Modelling of Graph Databases

*Jaroslav POKORNY**

Faculty of Mathematics and Physics, Charles University, Malostranske nam. 25, 118 00, Praha, Czech Republic

*pokorny@ksi.mff.cuni.cz

(Received: 3-February-2017; accepted: 20-April-2017; published: 8-June-2017)
DOI: http://dx.doi.org/10.25073/jaec.201711.44

Abstract. *Comparing graph databases with traditional, e.g., relational databases, some important database features are often missing there. Particularly, a graph database schema including integrity constraints is mostly not explicitly defined, also a conceptual modelling is not used. It is hard to check a consistency of the graph database, because almost no integrity constraints are defined or only their very simple representatives can be specified. In the paper, we discuss these issues and present current possibilities and challenges in graph database modelling. We focus also on integrity constraints modelling and propose functional dependencies between entity types, which reminds modelling functional dependencies known from relational databases. We show a number of examples of often cited GDBMSs and their approach to database schemas and ICs specification. Also a conceptual level of a graph database design is considered. We propose a sufficient conceptual model based on a binary variant of the E-R model and show its relationship to a graph database model, i.e. a mapping conceptual schemas to database schemas. An alternative based on the conceptual functions called attributes is presented.*

Keywords

Attribute, graph conceptual modelling, graph database, graph database modelling.

1. Introduction

There are several application domains in which the data has a natural representation as a graph. Well-known applications using graph data structures include particular fields of the Semantic web, i.e. RDF data, Linked data, and other graph-oriented data as social networks and information networks. Influence of graph technologies is noticeable in areas like geography, spatial objects, and semi-structured data (e.g. XML). Graph databases are useful for storage and processing sensor networks in logistics, protein interaction pathways in life sciences, etc.

Graphs used in today's applications are usually directed graphs based simply on a set of nodes, together with a binary relation on these nodes to represent the edges among them. Also undirected graphs and multigraphs can be defined in this way. The labeling of the edges is represented as a function from the edges to the finite alphabet of symbols. The graphs are enriched mostly with properly defined attributes (properties).

A *graph database* (GDB) is any storage system that uses graph structures to represent and store data. An associated new brand category of data stores is called *graph database management systems* (GDBMS). Today, due to some their special properties, GDBMS belong among so called NoSQL databases.

Similarly to traditional databases, GDBs are based on a (logical) data model. Such a model is characterized by the following three features:

- Data and the database schema are represented by a graph or by data structures generalizing the notion of graph.
- Manipulation of data is expressed by graph transformations like graph-oriented operations and type constructors.
- Integrity constraints (IC) enforce data consistency.

Most of the graph database models proposed in the literature ignore at least one of the three components of a complete database model. Particularly, there has been much less work devoted to the formalisms other than graph reachability patterns or, e.g., the ICs such as labels with unique names, typing constraints on nodes, functional dependencies, and domain and range of properties [1].

A *graph database schema* reflecting the above features consists of three components: a set of data structure types, a set of operators or inference rules, and a set of ICs (often called only *constraints*). Often we talk about the *logical schema* of a GDB in this context. Formally, ICs are statements about a GDB which must be satisfied. In effect, a GDB can be considered as an *instance* of its schema. For a GDB designer, the logical schema refers to the organization of data, which describes how the database will be constructed. Graph schemas are also appropriate tools to understand and visualize the data in the GDB.

Current commercial GDBs still need more improvements to meet these traditional dfinitions. A graph database model is usually not presented explicitly, but it is hidden in constructs of data definition language (DDL) which is at disposal in the given GDBMS. Especially the IC possibilities and a declarative language for online querying of graph data are either limited or completely lacking. Also the notion of database schema is often understood in other way than usually. Many graph database vendors have opted to either support a weaker notion of schema or to avoid it entirely. For example, a Titan [21] database schema can either be explicitly or implicitly defined. The schema is defined implicitly when it is first used during the addition of an edge, node or the setting of a property. GDBMS Orient [22] even distinguishes three roles of graph database schema: schema-full, schema-less, schema-hybrid (more in Sec. 3.2).

For example, the advanced GRAD [2] is a schema-less database model. This is in accordance with a general approach to NoSQL databases, where the notion of database schema is usually not considered at all. Its authors support the idea to load the data into a graph repository without restricting it to any predefined schema. Then, the analyst can design an application-implicit, on-read schema and have the flexibility to explore more conveniently the graph at run-time according to specific scenario. On the other hand, schema-less model leads to sub-optimal query processing because the code that needs to make a decision on which paths to explore in the graph has to first ask each object it encounters what type it is. And this is an expansive operation.

Conversely, strict schema enforcement is sometimes considered disadvantageous by those who develop applications for dynamic domains, e.g., domains dealing with user-generated content, where the structure of data may change very often [3].

Our intention is to study approaches to GDBs which are capable of expressing a schema and/or ICs, while at the same time still have manageable model checking properties. In a top-down approach to development of a GDB, ICs depend on the conceptual model behind and on expressiveness of the DDL. Obviously, the complexity of such formalisms depends on how the underlying directed graphs of the databases are presented. An interesting approach to GDBs is presented just by GRAD database model, which although schema-less, uses conceptual constructs occurring in E-R conceptual model.

Related works. There is not too much literature about graph database models and graph conceptual models. ICs in graph databases environment are mentioned also only marginally. Books are rather devoted to commercial GDBMSs [4] or Big Graph applications (e.g., [5]). In cite6 we discuss limitations of graph databases, but without consideration of their conceptual properties including ICs. These parts of graph database technology are discussed in [7]. Some papers partially compare graph database models used in various commercial GDBMS (e.g., [3], [8]). Concerning graph database schemas, three most popular GDBMS from the DB-Engines Ranking of Graph DBMS [23] , Titan and OrientDB use the notion of schema, not Neo4j [7]. But they enable only some simple ICs (see, Sec. 3.2.).

Papers focused on graph database modelling such as [2] and [3] provide an exclusion and consider ICs in more detail. The graph database model GRAD supports advanced graph structures, a set of well-defined constraints over these structures, and a powerful graph analysis oriented algebra. An attempt to implement a graph database schema expressed in UML on top of Neo4j is described in [9].

Objective and contribution. In the paper, we discuss issues and current possibilities and challenges in graph database modelling. Also a conceptual level of a graph database design is considered. We propose a sufficient conceptual model and show its relationship to a graph database model.

We will also use a functional approach to a database modelling in which a database graph is represented by so called attributes, i.e. typed partial functions [10]. We use for this approach the HIT Database Model, see, e.g., [12], as a functional alternative variant of E-R model. Then a typed lambda calculus can be used as a data manipulation language. This approach reflects the graph structure of a GDB and, on the other hand, provides powerful possibilities for dealing with properties in querying the GDB content. The paper is an extension of the work [7].

The rest of the paper is organized as follows. In Sec.2 we describe some basic graph database models including main types of graphs usable in this area. Section 3. presents a discussion about categories of ICs and some examples from GDBMS Neo4j, Titan, OrientDB, and Stardog [24] . We focus also on modelling functional dependencies between entity types, which reminds modelling functional dependencies known from relational databases, and extend them to conditional functional dependencies. In Sec. 4 we will attempt to introduce a conceptual level for GDB, i.e., the level not considered in the graph database world at all. We use a binary variant of the E-R model and introduce a functional approach with so called attributes as first class citizen construct. Finally, Sec. 5 concludes the paper.

2. Logical Level of Graph Databases

2.1. Graph Definitions

All graph database models have their formal foundations in the basic mathematical definitions of various types of graphs. The most commonly used model of graphs in this context is called a (*labelled*) *property graph model* [4]. For example, current leading graph databases Neo4j and Titan are built on top of property graphs. The property graph contains connected entities (the *nodes*) which can hold any number of attributes (*properties*) expressed as key-value pairs with text keys in the simplest case. *Relationships* provide directed, semantically relevant connections (*edges*) between two nodes. A relationship always has a *direction*, a *start node*, and an *end node*. Nodes and edges are tagged with *labels* representing their different roles in application domain. There are exclusions, e.g. in GDBMS Titan unlike edge labels, node labels are optional. Some approaches refer to the label as the *type*. Labels may also serve to attach metadata - index or constraint information - to certain nodes and edges. Like nodes, relationships can have any properties. Often, relationships have quantitative properties, such as weight, cost, distance, ratings or time interval. Properties make the nodes and edges more descriptive and practical in use. The ability to type an edge and attach properties to it increases the semantic expressiveness of directed graphs. Both nodes and sometimes also edges are defined by a *unique identifier*.

As relationships are stored efficiently in GDB, two nodes can share any number or relationships of different types without sacrificing performance. Note that although they are directed, relationships can always be navigated regardless of direction. In fact, the property graph model concerns data structure called a *directed, labelled, attributed multigraph* in graph theory. Then, the definition of a database graph is as follows:

Definition 1: A database graph $G = (V, E, N, \Sigma, \rho, A, Att)$ is a directed, labelled, attributed multigraph, where V is a finite set of nodes with identifiers drawn from an infinite alphabet N, E is a set of edges, and ρ is an incidence function mapping E to $V \times V$. Node identifiers are called also *labels* (*node labels*). The *edge labels* are drawn from the finite set of symbols Σ, and α is an edge labelling function mapping E to Σ. A is a set of *attributes* (*properties*) represented by couples (A_i, $value_i$). Att is a mapping assigning to each node/edge a subset (possibly empty) of attributes from A.

Observe that Definition 1 accepts database graphs with different sets of attributes for nodes/edges of the same types. It occurs in practice, especially in cases where no graph database schema is at disposal, i.e., in schemaless GDBMSs.

Since an undirected edge can be always represented as two directed edges, each one in a reverse direction of the other, undirected graphs are a particular case of directed graphs. Often querying a GDB is formulated and performed regardless of direction type.

The currently known GDBMSs, categorized under the NoSQL umbrella, are mostly built on top of database graphs based on the Definition 1.

2.2. Graph database modelling

In a native graph database model, both the schema and its instances are modelled as graphs. Nodes and edges are first-class citizens. The model equips users by data as well as graph topology-aware data manipulation operators. These operators influence a development of graph query languages. They handle entire graphs and do not simply return sets of disconnected nodes. But there are exceptions, e.g. GDBMS Neo4j. Finally, the consistency of the graph data should be guaranteed by ICs dfined over the graph structures and maintained through, e.g., special algebraic operations. The constraints should be applied on whole subgraphs and not just on single nodes or edges.

Example 1: Suppose entity types `Language`, `Teacher`, and `Town`. Relationship types `Teaches` and `Is_born_in` describe teaching and to be born (in a town), respectively. An associated graph database schema is depicted in Fig. 2. Figure 1 shows an instance of this schema. We could suppose the following properties (without domains) for entity/relationship types in Fig. 2: `Language(Name, Textbook)`, `Teacher(#T_ID, T_name, Birth_year)`, `Town(Town_name, Population)`, `Teaches(Day, Hour, Room)`, `Is_born_in(Date)`. For example, the property `Textbook` could look as `Textbook:string(120)` for node type Language in Fig. 2 and as `Textbook:German for beginners` for node `German` in Fig. 1. Properties both of nodes and edges are omitted in Figs. 1 and 2.

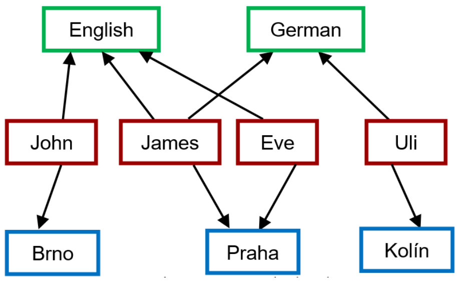

Fig. 1: A graph database.

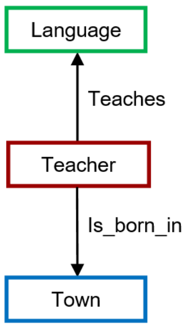

Fig. 2: Graph DB-schema 1.

The strings `Language`, `Teacher`, and `Town` as well as `Teaches` and `Is_born_in` can be used both for labeling in the graph database schema and in the associated GDB.

Because the human perceptual system is much more adept in working with graph data structures a good visualization is indispensable for GDB processing [5]. Authors of [13] mention the Neoclipse editor of Neo4j enabling visualizing and altering a GDB. Because a graph database schema is again database graph, it seems possible to use such tools for graph database modelling.

One can observe that the graph database schema 1 may not be sufficient for application using the GDB in Fig. 1. Obviously, each teacher can teach more languages and each teacher is born exactly in one town. These ICs should be already revealed at a conceptual level. Thus, there is M:N cardinality between languages and teachers, which is not expressed in Fig. 2. Then, a more sophisticated description would be needed. How can we expect, the answer is in the conceptual modelling and a conceptual schema designed for the GDB (see Sec. 4).

3. Integrity Constraints

In the case of the existence of a graph database schema, schema-instance consistency is required [14]. As in traditional databases, ICs provide a mechanism for capturing the semantics of the domain of interest represented by graphs. In the database area we usually distinguish three types of ICs. `Inherent constraints` are inherent to the data model itself and do not need to be specified explicitly in the schema but are assumed to hold by the definition of the model constructs. There are at least two inherent ICs in the graph data model considered:

1. Node labels in a GDB are unique.

2. Edges of the GDB are composed of the labels and nodes of the database graph in which the edge occurs.

These constraints correspond roughly to the ICs for the relational data model: (1) No component of the primary key of a base relation is allowed to accept `NULL` values; (2) The database must not contain any unmatched foreign key values.

An *explicit constraint* is any constraint that can be formulated in a DDL for a GDB. Sometimes also cardinalities of relationships are explicitly stated. Obviously, a goal is to develop a sufficiently expressive language for formulation of explicit ICs. Such languages are not yet common in the commercial GDBMSs. *Implicit constraints* are logical consequences of inherent and explicit ICs.

Another ICs concern property values both of nodes and edges. They include some domain constraints for particular properties, and possibly logical restrictions for their mutual relationships, e.g., functional dependencies known from relations in relational databases. However, GDBs are well-suited for situations in which the data complexity is contained in the relationships between the entities rather than in the property values associated with single nodes and edges. Consequently, we will notice such ICs only marginally in the paper.

Mostly the following ICs are studied [3]:

- types checking,
- node/edge identity, to verify that an entity or a relationship can be identified by either a value (e.g., name or ID) or the values of its attributes;
- referential integrity, to test that only existing entities are referenced;
- cardinality checking, to verify uniqueness of properties or relationships;
- functional dependencies, to test that an element in the graph determines the value of another;
- graph pattern constraints, to verify a structural restriction (e.g., path constraints).

It is remarkable, that for GDBMS DEX (now Sparksee [25]) and InfiniteGraph [26] the last three IC types were not supported at all in 2012. Today, a full schema model is currently used in InfiniteGraph. However, the developers of this GDBMS provide users with a schema-hybrid model involving strongly-typed objects for performance and loosely-typed objects for flexibility.

3.1. Formal approaches to integrity constraints

As a formal apparatus for some ICs the predicate calculus can be used. In the case of, e.g., quantifying nodes and edges labels, the second-order logic is needed. Authors of [15] show how description logics are well suited for this purpose. A usable approach is 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.

A natural and useful IC is a *functional dependency* (FD) on graph nodes and edges. For example, Yu and Heflin [16] proposed a value-clustered graph functional dependency for RDF data. Comparing to FDs known in the relational data model (see, e.g., [17]), in a GDB FDs require a special approach. An oriented edge in the graph database schema does not necessarily denotes a FD, e.g., `Teaches` in Fig. 2, and otherwise `Is_born_in` does. It means that FD specification has to be conceived as a formulation of explicit ICs on the database level. Due to the fact that graph database schemas are multigraphs, FD description needs node and edge labels, and direction, e.g., $\texttt{Teacher} \rightarrow_{\texttt{Is\_born\_in}} \texttt{Town}$ denotes such FD.

Often, FDs can be found for some edges coming from non-functional relationships, e.g., `Teaches`. In associated domain there is a rule, that teachers older than 70 teach at most one language. Such a dependency can be specified as e.g., as $\texttt{Teacher(Birth\_year} > \texttt{1994)} \rightarrow_{\texttt{Teaches}} \texttt{Language}$. We call such FDs *conditional functional dependencies*. Generally, they are described by expressions $\texttt{A}(\phi) \rightarrow_{\texttt{R}} \texttt{B}$, where `A`, `B` are node labels, `R` is edge label, and ϕ is a Boolean expression. Considering Armstrong's axioms used in the theory of FDs in the relational data model, only axiom of transitivity is relevant here. For example, $\texttt{A}(\phi_1) \rightarrow_{\texttt{R}} \texttt{B}$ and $\texttt{B}(\phi_2) \rightarrow_{\texttt{S}} \texttt{C}$ imply $\texttt{A}(\phi_1) \rightarrow_{\texttt{T}} \texttt{C}$. Obviously, it would be necessary to specify a meaningful name of the new edge `T`, which arises by composition of `R` and `S`. The axiom of reflexivity can be not applied here. For example, the statement $\texttt{Person} \rightarrow_{\texttt{Is\_friend\_of}} \texttt{Person}$ does not hold. A person can have more friends.

In practice, an important problem is that GDBs might be inconsistent, i.e., the database might fail to satisfy all ICs. In the case of GDB applications, such inconsistencies appear due to interoperability and graph distribution. For example, inconsistency might arise while integrating several sources into a single RDF graph, or while performing statistical inference on a scientific or social network [18]. Similar issues occur in integration of any heterogeneous databases. Thus, there is a need for developing an inconsistency-tolerant semantics for GDB.

3.2. Examples of integrity constraints in GDBMSs

We will show a number of examples of often cited GDBMSs and their approach to database schemas and ICs specification.

Neo4j: Neo4j is a schema-less GDBMS. In terminology of Neo4j so called schema is a persistent database state that describes available indexes and enabled constraints for the data graph, i.e. GDB. On the other hand, Neo4j helps enforce data integrity with the use of ICs. ICs can be applied to either nodes or relationships. Unique node property constraints can be created, as well as node and relationship property existence constraints.

Suppose nodes with the label `Teacher`. Then the following ICs can be specified:

```
CREATE CONSTRAINT
ON(teacher:Teacher) ASSERT
teacher.#T_ID IS UNIQUE,
CREATE CONSTRAINT
ON(teacher:Teacher) ASSERT
exists(teacher.Birth_year),
```

i.e. all nodes with the label Teacher have a certain property.

```
CREATE CONSTRAINT ON()-
[teaches:Teaches]-() ASSERT
exists(teaches.Room),
```

i.e. all relationships with the label Teaches have a certain property.

Titan: GDBMS Titan enables to define some ICs in the graph schema definition, e.g. cardinality settings for node and edges properties, to distinguish simple graphs and multigraphs, and 1:1, 1:N, and N:1 cardinalities. For example,

```
town = mgmt.makeEdgeLabel('Is_born_in')
.multiplicity(MANY2ONE).make()
```

i.e. the edge label `Is_born_in` is an example with `MANY2ONE` multiplicity since each teacher has at most one town where he/she is born, but towns can have multiple teachers born there. In Titan DDL schema elements can be even redefined during the existence of a graph.

OrientDB: GDBMS OrientDB brings together the power of graphs and the flexibility of documents into one scalable database even with an SQL layer. In OrientDB, classes for node types and edge types are defined, e.g.,

```
orientdb>CREATE CLASS Teacher
EXTENDS V orientdb>CREATE CLASS
Language EXTENDS V
```

```
orientdb>CREATE CLASS Teaches
EXTENDS E
```

Nodes and edges (records of particular classes) are created by commands `CREATE VERTEX` and `CREATE EDGE`, respectively. During this process identifiers of nodes and edges are automatically generated. To require that the `Teaches` edge only exists between the node of type `Teacher` and the node of type `Language`, it is necessary to specify

```
orientdb> CREATE PROPERTY
Teaches.out LINK Teacher
orientdb> CREATE PROPERTY
Teaches.in LINK Language
```

Properties are defined as class fields, e.g. by command

```
teacher.createProperty("Birth_year",
OType.int)
```

They can be constrained by ICs like: Minimum Value: `setMin()`, Maximum Value: `setMax()`, Mandatory: `setMandatory()`, Read Only: `setReadonly()`, Not Null: `setNotNull()`, and `Unique`.

The role of graph database schema can be precisely specified in OrientDB:

- *schema-full* - enables strict-mode at a class-level and sets all fields as mandatory.
- *schema-less* - enables classes with no properties. Default is non-strict-mode, meaning that records can have arbitrary fields.
- *schema-hybrid* - enables classes with some fields, but allows records to define custom fields. This role is also sometimes called *schema-mixed*.

Stardog: A more sophisticated approach to ICs is offered by GDBMS Stardog. Stardog supports RDF graph model and property graph model. It takes the IC validation as a data quality mechanism via closed world reasoning. ICs can be specified in languages as OWL, SWRL, and SPARQL and serve for validation of RDF data. The authors of Stardog argue that ICs in Stardog may be arbitrarily complex and include many conditions.

A special problem of ICs is their checking. If an IC is enabled, data is checked as it is entered or updated in the database, and data that does not conform to the IC is prevented from being entered. Relatively easy is the case when an IC concerns a node and its neighbors. More complex can be to verify a structural restriction.

4. Conceptual Level of graph Databases

Graph-based conceptual schemas are an effective communication medium between users of any a

GDB. They can significantly help to GDB designers. We describe a graph database by defining its conceptual schema in a binary variant of E-R conceptual model in Sec. 4.1. In Sec. 4.2, we present some mapping rules transforming a graph conceptual schema expressed in E-R model into graph databases schema in a weaker variant of this E-R model. This approach is necessary due to the inherent properties of database graphs used for GDB in this paper.

4.1. A binary E-R model as a variant for graph conceptual modelling

As usually in the database world, we use a variant of E-R model for conceptual modelling of GDB. Based on the general definition of E-R model, in practice it is possible to define various E-R notations and more or less restricted variants of E-R models. As we deal with directed, labelled, attributed multigraphs, only a binary E-R variant can be considered. We propose a graph conceptual schema definition based on the binary E-R model with *strong entity types, weak entity types, binary relationship types, attributes, identification keys, partial identification keys, ISA-hierarchies, and min-max ICs.*

In a binary E-R model, we are limited to only binary relationship types with the cardinalities 1:1, 1: N, and M:N. Fig. 3 uses the constructs coming from the binary E-R model used in the Oracle Designer CASE Notation (see, e.g., [19]). Its original notation, however, considers only 1:1 and 1:N cardinalities, i.e., M:N relationship types must be decomposed into two 1:N relationship types. This is not too necessary for our graphs, since M:N cardinalities can be modelled directly. Thus, schemas like the one in Fig. 3(a.) are acceptable. Both entity and relationship types can have attributes. Remind that in the Binary E-R model by Oracle relationship types have no attributes. This fact reflects so called semantic relativism existing from the decomposition of M:N relationship types. Observe that cardinalities 0 and 1 are not precisely distinguishable in Fig. 3(a.). For this purpose, the min-max ICs are usable. In this model variant, min-max cardinalities are expressed using the crow's foot notation used for the start node and the end node of some edges (see, Fig. 3(b.)). A straight and dotted line express mandatory and optional relationship, respectively. Min-max ICs could be expressed equivalently by expressions $(E1 : (a, b)$, $E2 : (c, d))$, where $a, c \in \{0, 1\}$, $b, d \in \{1, N\}$, and N means "any number greater than 1".

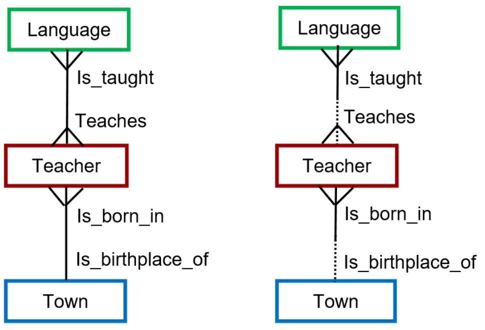

Fig. 3: (a) Graph C-schema 1. (b) Graph C-schema 2.

Weak entity types are identification- and existence-dependent on some other entity type. Suppose a weak entity type E_W with a partial identification key that distinguishes instances of E_W that are related to the same instance of a strong entity type E. The full identification key of E_W then has to include the identification key of E. In Fig. 4, `Street` is a weak entity type. Its partial identification key is `Street_name`. On the database schema level, the identification key of Street would be `Town_name, Street_name`. The perpendicular line denotes the identification and existence dependency. Identification dependency implies existence dependency.

Subtyping (ISA-hierarchies) can be simply expressed in this conceptual model as well, e.g., `Teacher ISA Person`. Identification key of `Teacher` would be `#Person_ID`. Due to the explicit `Is_a` edge, the inherited information is not necessary in subtypes. Obviously, `Teacher` has yet its own identification key `#T_ID`. The supertype identification key in the `Teacher` type can only simplify querying in the associated GDB. A possible associated graph database schema is in Fig. 5.

Weak entity types can be quite complex. To reach a strong entity type for the weak entity

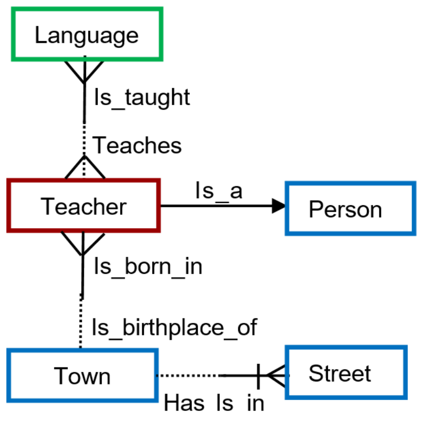

Fig. 4: Graph C-schema 3.

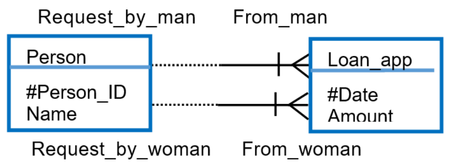

Fig. 6: Graph C-schema 4.

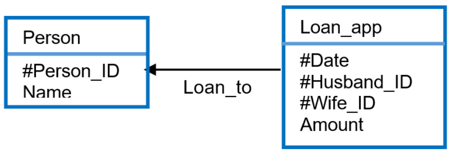

Fig. 7: Graph DB-schema 3.

type E_W, it can be necessary to form a sequence of other weak entity types or even a few such sequences for obtaining the resulted identification key of E_W.

Example 2: Consider entity types Person and `Loan_app`. Two persons are necessary for a loan transaction (e.g., a husband and his wife). Loans of couples are numbered locally by dates (see, Fig. 6). Consequently their full identification key will be `#Husband_ID, #Wife_ID, #Date`, where a referential integrity exists in `Loan_app`, i.e. `#Husband_ID` $\subseteq$ `#Person_ID` and, similarly, `#Wife_ID` $\subseteq$ `#Person_ID`.

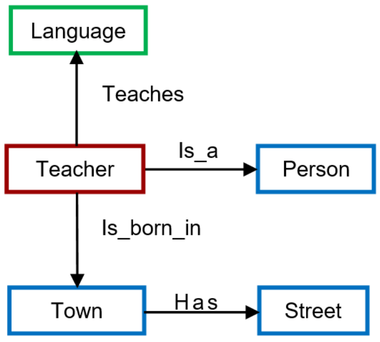

Fig. 5: Graph DB-schema 2.

On the other hand, the associated graph database schema in Fig. 7 will be simpler, due to the one-way orientation and union of partial keys. Somebody could ask why only one edge label is used in the graph database schema 3. Obviously, two edges will lead from each `Loan_app` node in a GDB instance. This should be ensured by an IC. For better distinguishing the husband and his wife in personal data, two different edge labels would be more appropriate. In Definition 2 we will not consider this possibility, i.e. we will use the variant with only a unique sequence of weak entity types.

People experienced in E-R conceptual modelling may lack other details of supertype/subtype hierarchies, such as two important constraints on the subtype entities: disjointness and completeness. For example, the `Student` entity type could exist as a further subtype of `Person`. Then ICs like `Student` $\cap$ `Teacher` $= \emptyset$, and `Student` $\cup$ `Teacher` $=$ `Person` are meaningful.

Definition 2: A *graph conceptual schema* in the binary E-R model is 4-tuple $< \boldsymbol{E}, \boldsymbol{R}, \boldsymbol{H}, \boldsymbol{CC} >$, where $\boldsymbol{E}$ is a set of entity types, each of them is given by its name E_i and a set of attributes A_{E_i}. One or more attributes from A_{E_i} determine the identification key K_{E_i} of E_i. $\boldsymbol{R}$ is a set of binary relationship types, while each relationship type R is given by a couple (E_{i1}, E_{i2}) and a set of attributes A_R. There are two inverse relationship names for each relationship type. If $E_{i1} = E_{i2}$ for R, then such relationship type is called recursive. $\boldsymbol{H}$ is a set of ISA-hierarchies of entity types, and $\boldsymbol{CC}$ is a set of ICs.

There is a set $E_W \subset E$ (possibly empty) of weak entity types. For each weak entity type E_W there is at least one sequence $E_1, \ldots, E_s$, such that $E_1 = E$, E_{i-1} is identification dependent on E_i, $i = 2, \ldots, s-1$, and E_s is a strong entity type. Identification key of E_W is the union of all partial and complete identification keys from this sequence. In each ISA-hierarchy $H_E \in \boldsymbol{H}$, $H_E \subseteq \boldsymbol{E} \times \boldsymbol{E}$,

- entity type E is the source of H_E with identification key K_E,
- the graph associated to H_E is a tree with the root E,
- there is no hierarchy $H_E{}' \in \boldsymbol{H}$ such that the tree associated to H_E is a subtree of tree, which is associated to hierarchy $H_E{}'$, except of the case, when H_E has only a root.

For each relationship type $R \in \boldsymbol{R}$ there are two min-max ICs in $\boldsymbol{CC}$ and vice versa, to each min-max IC from $\boldsymbol{CC}$ there is at least one relationship R in $\boldsymbol{R}$ having this IC as its min-max IC.

Conceptually, other generic relationship types, e.g., `is-part-of` relationships, could be considered in the binary E-R model. They can be described simply with graph conceptual constructs as well.

4.2. Mapping conceptual schemas to database schemas

A correct graph conceptual schema may be mapped into an equivalent (or nearly equivalent) graph database schema with the straightforward mapping algorithm but with a weaker notion of a database schema, i.e. some inherent ICs from the conceptual level will be neglected to satisfy usual notation of directed, labelled, attributed multigraphs. Then the mapping algorithm transforming a graph conceptual schema C into a graph database schema D can be described by the following rules:

Rule 1. *Strong entity types.* For each strong entity type $E \in \boldsymbol{E}$, create a node type D_E which includes all attributes from A_E and the same identification key K_E.

Rule 2. *Weak entity types.* Let $E_1, \ldots, E_s$ is a sequence of entity types from E, where E_s in a strong entity type, and E_i, $i = 1, \ldots, s-1$, are weak entity types with partial keys PK_i. For each weak entity type E_i create a node type D_{E_i} which includes all the attributes of A_{E_i} and identification key K_{i+1} from $D_{E_{i+1}}$. The resulted key $K_i = PK_i \cup K_{i+1}$, $i = 1, \ldots, s-1$.

Rule 3. *Relationship types.* For each relationship type $R \in \boldsymbol{R}$, create an edge type D_R which includes all attributes from A_R. Specify the label and direction belonging to the edge type.

Rule 4. *Integrity constraints.* Inherent ICs such as cardinalities become explicit ICs in D. Explicit ICs from $\boldsymbol{CC}$ become also explicit ICs in D.

Rule 5. *ISA-hierarchies.* They are transformed directly into ISA relationships in the GDB level. Associated edges will be labelled as `Is_a`. For better manipulation of data from ISA-hierarchies, we recommend to propagate the identification key K_E of the source of each ISA-hierarchy into all nodes of this hierarchy in D.

An *instance* of the graph database schema is a database graph containing nodes labelled with the associated entity types or identifiers and labelled edges according to the schema.

Rule 3 needs an explanation. The label used for the edge should be chosen accordingly to the chosen edge direction. For example, M:N relationship between type `Teacher` and `Language` in Fig. 3 (graph conceptual schema 1) can be transformed to the `Teaches` edge with direction from `Teacher` to `Language` (see Fig. 2) or `Is_taught` with reverse direction. Such edges express the relationship semantics only in one direction. Our binary E-R model uses two inverse relationship names for better readability of the conceptual schema. We can use both on the conceptual level, but in the database schema only one of these labels is used as well as only one direction.

Similarly, a question is why the relationship type `(Town, Street)` is not inverse in Fig. 5. Yes, it could be, obviously with a different assumption on the database implementation. For application requirement to have more queries on streets of a town, the first choice is more adequate. Consequently, from a graph conceptual schema we can propose several different graph database schemas. The final selection is influenced by the analytical phase of the GDB development.

In practice some weaker variants of graph database schemas are used. For example, such a schema has only subsets of entity/relationship attributes for some nodes/edges. Consequently, node and edges can own only subsets of key-value couples. It is in accordance to key-value NoSQL databases which do not represent explicitly a missing information. Similarly, ICs may be missing in the graph database schema at all. An advantage of such approach is that graphs, which allow for rich data structures without the ICs of schema, are naturally extensible and amenable to continuous data evolution.

A special problem is the language for ICs, i.e. the associated DDL. For example, we can state the IC on the database level requiring that each teacher in the database graph teaching German should be born after 1980. That is, such a teacher has to be related to the German language. If these conditions are not met, then, e.g., the insert transaction of `Teaches` edge in the database graph should fail. Using a graph pattern (see the approach in Section 3.1) we can obtain IC in Fig. 8. We can observe that the pattern is a generalization of a conditional functional dependency.

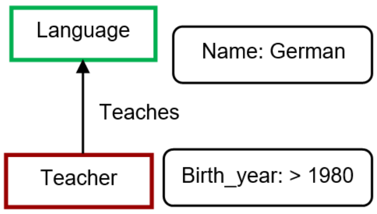

Fig. 8: Integrity constraint pattern.

A significant problem is how to use these patterns in practice, reminding that the problem of graph matching using subgraph isomorphism is known to be NP-complete.

4.3. Functional approach graph conceptual modelling

A conceptual modelling can be based on the notion of attribute viewed as an empirical typed function that is described by an expression of a natural language [12]. A lot of papers are devoted to this approach studied mainly in 90ties (see, e.g., [20]).

Types

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

If $S, R_1, ..., R_n (n \geq 1)$ are types, then
(i) $(S : R_1, ..., R_n)$ is a (*functional*) *type*,
(ii) $(R_1, ..., R_n)$ is a (*tuple*) *type*.

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

The elementary type *Bool* = {TRUE, FALSE} is also in **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 notion of a set is then redundant here.

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

Attributes

Object structures usable in building a database can be described by some expressions of a natural language. Suppose **B** = { *Language, Teacher, Town, School, Name, Birth_year*, ...}. Then, e.g, "the language thought by a teacher at a school" (abbr. LTS) is a $(Language : Teacher, School)$-object, i.e. a (partial) function $f : Teacher \times School \to Language$. Such functions are called *attributes* in [12].

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. We can omit parameters w and t in type(M). In the case of LTS attribute we consider possible worlds, where teachers teach at most on language in each school. For GDBs we can elementary entity types conceive as sets of node 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 $((Bool : (Bool : Student)) : School)$, i.e. the classes contain sets of students and the CS returns a set of classes (of students) for a given school.

We can also consider other functions that need no possible world. For example, the aggregate function like COUNT, + (adding) provides such function. These functions have the same behavior 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. The range of these functions are again functions. Analytical functions are of type R, where R does not depend on W and T. In the conceptual modelling, each base **B** consists of descriptive and entity types. Descriptive types (*String, Number,* etc.) serve for domains of properties.

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.

Now we add properties. Properties describing entity types can be of types $(S_1, ..., S_m : R)$, 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 $(S_1, ..., S_m, R_1 : R_2)$ or $(Bool(S_1, ..., S_m, R_1) : R_2)$.

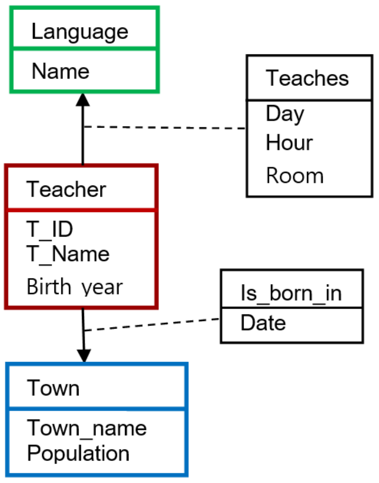

Fig. 9: Graph DB-schema 1 with properties.

Then a functional database schema corresponding to the graph database schema in Fig. 9 can look as

Language/((*Name, Textbook):Language*)
Teacher/((*T_ID, T_Name, Birth_year):Teacher*)
Town/((*Town_name, Population):Town*)
Teaches/(*Bool:(Day, Hour, Room, Language) :Teacher*)
Is_born_in/((*Date, Town):Teacher*)

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.

Then, the associated typed lambda calculus with applications of functions and lambda abstractions provides a powerful tool for querying graph data conceived as functions [10].

5. Conclusion

In this paper, we proposed an approach to modelling GDBs based on a classical technique, here a binary variant of the E-R model, known from the world of relational DBMSs. We also proposed rather non-traditional functional approach to modelling graph data based on the notion of attribute. Attributes are conceptual objects with extension enabling to conceive a property graph as a set of functions.

We used the notions of graph conceptual model and graph database model. We also discussed relationships between schemas in both models, particularly the transformation of a graph conceptual schema to a graph database schema. Comparing to similar approaches in the world of relational DBMSs, the resulted schema is not given uniquely in this approach, both in terms of graph structure and ICs. We discussed also some types of ICs reminding functional dependencies known from a relational theory. Both graph data modelling and ICs formulation are yet maturing and offer an interesting theme for future research.

Acknowledgment

The work was supported by the Charles University project Q48.

References

[1] LARRIBA-PEY, J. L., N. MARTINEZ-BAZAN and D. DOMINGUEZ-SAL. Introduction to Graph Databases. In: *10th International Summer School*. Athens: Springer, 2014, pp. 171–194. ISBN 978-3-319-10586-4. ISSN 0302-9743.

[2] GHRAB, A., O. ROMERO, S. SKHIRI, A. VAISMAN, and E. ZIMANYI. BGRAD: On Graph Database Modelling. In: Cornel University Library [online]. 2016.

[3] ANGLES, R. A Comparison of Current Graph Database Models. In: *28th International Conference on Data Engineering Workshops*. Arlington: IEEE, 2012, pp. 171–177. ISBN 978-0-7695-4748-0.

[4] ROBINSON, I., J. WEBBER and E. EIFREM. Graph databases. 1st ed. O'Reilly Media, 2013. ISBN 978-1-4493-5626-2.

[5] POKORNY, Jaroslav and Vaclav SNASEL. *Graph-based social media analysis: In Graph Based Social Media Analysis*. Chapman and Hall/CRC, 2015, pp. 391–416. Chapman. ISBN 978-1-4987-1904-9.

[6] POKORNY, J. Graph Databases: Their Power and Limitations. In: *Proceedings of 14th Int. Conf. on Computer Information Systems and Industrial Management Applications (CISIM 2015)*. vol. 9339. Warsaw: Springer, 2015, pp. 58–69.

[7] POKORNY, J. Conceptual and Database Modelling of Graph Databases. In: P*roceedings of the 20th International Database Engineering*. Montreal: ACM Press, 2016, pp. 370–377. ISBN 978-1-4503-4118-9.

[8] JADHAV, P. and R. OBEROI. Comparative Analysis of Different Graph Databases. *Int. Journal of Engineering Research & Technology*, vol. 3, iss. 9, pp. 820–824, 2014. ISSN 2278-0181.

[9] DELFOSSE, V., R. BILLEN and P. LECLERCQ. UML as a schema candidate for graph databases. In: *Proceedings of NoSQL Matters*. 2012, pp. 1–8.

[10] POKORNY, J. Functional Querying in Graph Databases. In: *Asian Conference on Intelligent Information and Database Systems*. Kanazawa: Springer, 2017, vol. 10191. pp. 291–301. ISBN 978-3-319-54471-7.

[11] MENDELZON, A. O. and P. T. WOOD. Finding Regular Simple Paths in Graph Databases. *SIAM Journal on Computing.* 1995, vol. 24. no. 6, pp. 1235–1258.

[12] POKORNY, J. A function: unifying mechanism for entity-oriented database models. In: *Entity-Relationship Approach: A Bridge to the User, Proceedings of the Seventh International Conference on Entity-Relationship Approach*, North-Holland: Elsevier Science Publishers B.V., pp. 165–181, 1989.

[13] KAUR, K. and R. RANI. Modeling and querying data in NoSQL databases. In: *International Conference on Big Data.* Silicon Valley: IEEE, 2013, pp. 1–7. ISBN 978-1-4799-1293-3.

[14] ANGLES, R. and C. GUTIERREZ. Survey of graph database models. *ACM Computing Surveys (CSUR).* 2008, vol. 4, iss. 1. ISSN 0360-0300.

[15] CALVANESE, D., M. OORTIZ, and M. SIMKUS. Evolving Graph Databases under Description Logic Constraints. In: *Proceedings of the 26th Int. Workshop on Description Logics (DL 2013).* 2013, vol. 1014,

[16] YU, Y. and J. HEFLIN. Extending Functional Dependency to Detect Abnormal Data in RDF Graphs. In: *The Semantic Web – ISWC.* 2013, Berlin: Springer, vol. 7031, pp. 794–809. ISBN 978-3-642-25072-9.

[17] SILBERSCHATZ, A., H. KORTH, and S. SUDARSHAN, *Database System Concepts.* 6th ed. McGraw-Hill, 2010.

[18] BARCELLO, P. and G. FONTAINE, On the Data Complexity of Consistent Query Answering over Graph Databases. In: *Proceedings of 18th International Conference on Database Theory (ICDT'15).* Leibniz Int. Proceedings in Informatics, pp. 380–397, 2015. ISSN 0022-0000.

[19] BARKER, B., *CASE*METHOD: Entity Relationship Modeling.* Addison-Wesley Publishing Company, New York, 1990.

[20] POKORNY, J. Database semantics in heterogeneous environment. In: *Proceedings of 23rd Seminar SOFSEM 96: Theory and Practice of Informatics*, Springer-Verlag, pp. 125-142, 1996.

[21] Titan: Distributed Graph Database [online]. 2017. Available at: `http://titan.thinkaurelius.com/`.

[22] OrientDB Ltd. OrientDB [online]. 2017. Available at: `http://orientdb.com/`.

[23] DBengines. DB-Engines Ranking of Graph DBMS [online]. 2017. Available at: `https://db-engines.com/en/ranking/graph+dbms`.

[24] Stardog Union. Stardog 5 [online]. 2017. Available at: `http://www.stardog.com/`.

[25] Sparcity Technologies. Scalable high-performance graph database [online]. 2017. Available at: `http://sparsity-technologies.com/#sparksee`.

[26] Objectivity. InfiniteGraph [online].

2017. Available at: `http://www.objectivity.com/products/infinitegraph/#.U8O_yXnm9I0`.

About Authors

Jaroslav POKORNY is a full professor of the Faculty of Mathematics and Physics at Charles. His research interests include database technologies, information retrieval, data organization, and XML. He has published more than 300 papers and 6 books. He organized ADBIS-DASFAA, EDBT, ISD, ICADIWT, and ADBIS international conferences in Czech Rep. in 2000-2016, in other conferences he served as their general-chair. He is a member of the Editorial Boards of Computing and Informatics, J. of Systems Integration, Int. J. of Web Information Systems, and J. of Advanced Engineering and Computation. He is a member of ACM and IEEE. From 2005 he serves as a representative of Czech Rep. in IFIP.