Datenbank Spektrum (2018) 18:89–98
https://doi.org/10.1007/s13222-018-0286-9

SCHWERPUNKTBEITRAG

Using Queries as Schema-Templates for Graph Databases

Stephan Mennicke[1] · **Jan-Christoph Kalo**[1] · **Wolf-Tilo Balke**[1]

Received: 1 February 2018 / Accepted: 8 May 2018 / Published online: 31 May 2018

Abstract
In contrast to heavy-handed ER-style data models in relational databases, knowledge graphs (or graph databases) capture entity semantics in terms of entity relationships and properties following a simple collect-as-you-go model. While this allows for a more flexible and dynamically adaptable knowledge representation, it comes at the price of more complex querying: with varying degrees of information sparsity, it will gradually become more difficult to figure out what an entity actually represents. Thus, matching the intended schema as specified by a query against actually occurring entity patterns in the graph database needs severe attention on a conceptual level. In this article, we analyze graph patterns as schema information from a graph pattern matching perspective. We argue that every query consists of a mixture of conceptual information (how entities are structured) together with evaluation information (further dependencies and constraints on data) and that this mixture is not always easy to divide. To arrive at truly schema-aware graph query processing, we propose several matching mechanisms, each mandating a specific semantic meaning of the graph pattern, and discuss their practical applicability.

Keywords Graph databases · Graph queries · Conceptual modeling · Pattern matching

1 Introduction

Relational database design commences with creating a conceptual model of the domain of interest. Typically, the *universe of discourse* is captured using Entity-Relationship (ER) modeling [7] yielding a set of entity types (including their attributes) plus a set of possible relationship types between them. The conceptual model is transformed into a logical schema, which in turn defines the foundation for possible queries for applications. With the growing availability of *Web Data*, graph databases have aroused a vivid interest in the database community. They provide a powerful alternative to classical relational systems, promising more flexibility by not adhering to a rigid database schema. Indeed, it is enticing to open up the abundance of unstructured information on the Web through transformation into a dynamically growing collect-as-you-go manner. A good example is Google's Knowledge Graph supporting sophisticated features in Web search [9]. Generally speaking, in a graph databases entity-centric data from diverse domains coexist. Entities are represented as nodes connected by labeled edges (relations) to other entity nodes or attribute nodes (literals). These nodes of a graph database are often referred to as graph database objects. We present example graph database excerpts on books, movies and their (screen) writers in Fig. 1.

As an illustration, imagine a user querying a book database. Since in relational databases entity types are rigidly captured by the logical schema and a fixed set of keys, it is sufficient to query the following table:

ISBN	Author	Title
9780582186552	`J.R.R. Tolkien`	`The Hobbit`

However, entities with missing key information, here ISBN, cannot be inserted into the table. If we want to find all books from our example database, it is sufficient to use the simple SQL query

```
SELECT author, title, isbn FROM book.
```

✉ Stephan Mennicke
mennicke@ifis.cs.tu-bs.de

Jan-Christoph Kalo
kalo@ifis.cs.tu-bs.de

Wolf-Tilo Balke
balke@ifis.cs.tu-bs.de

1 Institut für Informationssysteme, TU Braunschweig, Mühlenpfordtstraße 23, 38106 Braunschweig, Germany

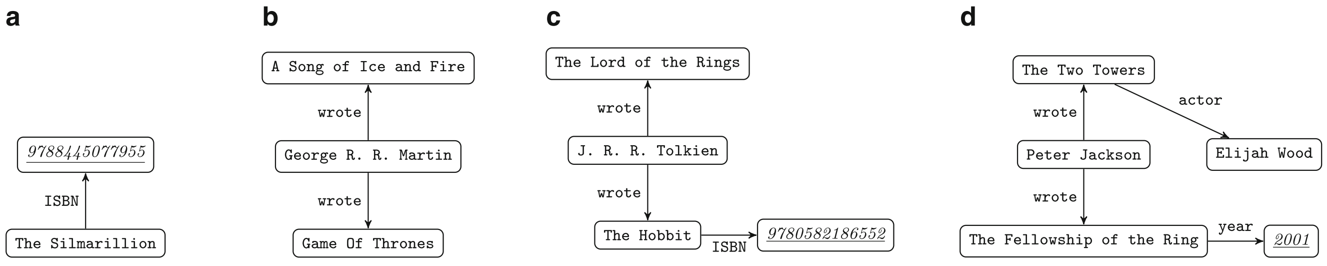

Fig. 1 A graph database example

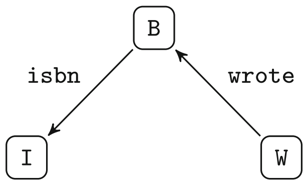

Fig. 2 A Graph Pattern

In contrast, for graph queries it is necessary to clarify the concept *book* first. As an example, all books in the sample graph database in Fig. 1 should be returned but since no schema information on book-type entities is guaranteed here, the query cannot be easily answered. Hence, a conceptualization step is necessary to find all graph patterns corresponding to the entities of a certain type (in our case books) within the graph database using some matching mechanism. (Sub-)graph isomorphism represents a popular pattern matching mechanism, allowing to relate graph patterns to subgraphs from a graph database such that matches *look identical*. Interpreted in a graph-isomorphic manner, the graph pattern depicted in Fig. 2 will correspond to the concept of a book, being related to some ISBN and an author. Yet, solely relying on graph isomorphism as matching relation (i.e. a one-to-one correspondence), the pattern would only return `The Hobbit` from our sample graph database, even though three other book entities might be contained that, however, cannot be safely identified as books due to data sparsity. Employing schema information, a pattern comprising only the `ISBN` relationship would increase recall but still lack a complete result set.

The example above demonstrated that flexible knowledge representations without a-priori schemas, as in graph databases, comes at a price. In real-world, entity information is often sparse and the representation in terms of entity types may be quite heterogeneous. This is why querying such a graph database requires a more complex query process that actively deals with schema heterogeneity and data sparseness. In fact, the lack of a fixed schema together with identifying key information, etc. requires a process of data conceptualization at query time. The graph database query process thus can be understood as a two-step process implicitly performed by the database user formulating queries. Since a query comprises information of several entity types being related to each other, as a first step, all structural representations of respective entities in the database have to be identified for each entity type mentioned in the query. The second step is the evaluation step, in which the actual query answers are computed by employing additional filtering constraints (mostly formulated as constraints on data) and by joining related entity types.

In this article, we discuss different conceptual notions with respect to querying graph databases. After basic definitions (Sect. 2), we present a recall-oriented pattern matching relation as a starting point and provide a detailed analysis in the context of the graph query answering (Sect. 3). Moreover, we contribute a theory of failures, supporting matching relations in expressing negation (a feature step-wise integrated into modern graph query languages, see [4]). Hence, it is possible to capture even more complex concepts in a graph pattern without the need of complicated query language constructs (see Sect. 4). We show the limits of only using structural information in the form of graph patterns and relations to deal with sparsity and especially with a view towards heterogeneity (Sect. 5). Finally, we discuss new ideas based on Semantic Web technologies that may support graph database querying. Here, we rely on mining existing graph databases for entity schemata to use in query processes.

2 Graph Databases and Queries

For an extensive survey on graph database models, the reader is referred to Angles and Gutierrez [3]. Besides basic graph notions, this section covers *graph homomorphism*, a pattern matching relation strongly corresponding to the way modern graph query languages handle pattern queries [4], e.g. SPARQL. Throughout the paper, we make use of *labeled directed graphs*. A graph is a triple $G = (V, \Sigma, \longrightarrow)$ with a finite set of nodes V, a finite alphabet Σ, and a directed and labeled edge relation $\longrightarrow \subseteq V \times \Sigma \times V$. We abbreviate $(v_1, a, v_2) \in \longrightarrow$ to $v_1 \xrightarrow{a} v_2$. Furthermore, $v_1 \xrightarrow{a}$ means that a $v_2 \in V$ exists with $v_1 \xrightarrow{a} v_2$. Likewise, write $v_1 \not\xrightarrow{a}$ if there is no v_2 with $v_1 \xrightarrow{a} v_2$.

Since graph databases morally follow the lines of graphs, they also consist of nodes and (labeled) edges. In RDF, nodes refer to *objects*, stemming from an infinite supply of object identifiers $\mathcal{O}$, or *literals*, stemming from an infinite set of data points $\mathcal{L}$. $\mathcal{O}$ and $\mathcal{L}$ are assumed to be disjoint. In the graph databases, depicted in Fig. 1, objects are rounded corner nodes labeled in typewriter font, while all other nodes refer to literals (underlined). In graphical representations, the labels of nodes coincide with the chosen object identifiers or data points.

Database objects are related to other objects or literals, expressed as triples over objects, relation symbols and objects or literals. Therefore, $\mathcal{S}$ is the infinite supply of relation symbols, disjoint from $\mathcal{O}$ and $\mathcal{L}$. Triples form the edges of RDF graphs. Henceforth, we make synonymous use of the terms object and object identifier, literal and data point, and relation and relation symbol. We refer to edges between objects and literals as expressions of properties of the objects, for which literals define concrete values of the respective properties. Please note that literals may only be targets of triples. Reconsidering our example graph database in Fig. 1, objects `J. R. R. Tolkien` and `The Hobbit` are in a `wrote` relationship, intuitively expressing that object `The Hobbit` has been *written by* object `J.R.R. Tolkien`. Some objects are equipped with a property, e.g. `The Hobbit` relates to the literal $\underline{9780582186552}$.

Definition 1 A *graph database* is a triple

$$\mathrm{DB} = (O_{\mathrm{DB}}, \Sigma_{\mathrm{DB}}, \longrightarrow_{\mathrm{DB}}),$$

where $O_{\mathrm{DB}} \subset \mathcal{O} \cup \mathcal{L}$ is a finite set of *database objects*, $\Sigma_{\mathrm{DB}} \subset \mathcal{S}$ a finite set of *relation symbols*, and the *attributes of DB* $\longrightarrow_{\mathrm{DB}} \subseteq (O_{\mathrm{DB}} \cap \mathcal{O}) \times \Sigma_{\mathrm{DB}} \times O_{\mathrm{DB}}$.

Detour 1 *We have now decided to use an RDF-based data model. Of course, there are further graph models to mention. Property graphs allow for realizing attributes attached to database objects and relations between these objects. These graphs allow for a more compact representation of graph data but add no concept that cannot be encoded in RDF. For example, an attribute list $(p_1, v_1), ..., (p_k, v_k)$ attached to object o encodes to edges $o \xrightarrow{p_i} v_i$ $(i = 1, ..., k)$. Furthermore, there are unlabeled graphs such that every object may contain a set of attributes, sometimes also called data graphs. Encoding these attributes follows the same lines as for property graphs. In case of the missing edge labels, depending on the actual application, we may simply assign a single relation symbol to every edge in the graph that has not been used elsewhere. At least in theory, all notions we define are applicable also to other (data) graph models.*

Independent of concrete graph query languages, mechanisms answering a query against a graph database DB finds *subgraphs of DB*, representing an *answer to the query*. Graph $A = (O_A, \Sigma_A, \longrightarrow_A)$ is a *subgraph of DB* if $O_A \subseteq O_{\mathrm{DB}}$ and $\Sigma_A \subseteq \Sigma_{\mathrm{DB}}$, denoted by $A \preceq \mathrm{DB}$.

One of the most basic mechanisms in modern graph query languages is that of matching a *graph pattern* by *graph homomorphism*. Graph $P = (V_P, \Sigma_P, \longrightarrow_P)$ is a *graph pattern*. $A \preceq \mathrm{DB}$ is a *homomorphic match to* P, denoted $P \sqsubseteq_{\mathrm{hom}} A$, if mapping $\eta : V_P \to O_A$ exists with $v \xrightarrow{a}_P v'$ implying $\eta(v) \xrightarrow{a}_A \eta(v')$. η is called a *graph homomorphism between P and A*. The set of all graph-homomorphic matches is denoted by $[\![P]\!]^{\mathrm{DB}}_{\mathrm{hom}}$. Note that $P \sqsubseteq_{\mathrm{hom}} A \preceq \mathrm{DB}$ implies that $P \sqsubseteq_{\mathrm{hom}} \mathrm{DB}$, i.e. if there is one match, then the whole graph database is a match, because there is no restriction of having nodes in a match useless to the graph homomorphism. In graph query languages, graph homomorphisms are combined with a notion of projection to reduce the computational obligations.

Next, we move on to three other matching mechanisms, each capturing diverse graph properties, thus expressing different structural semantics, including assumptions on missing data.

3 Schematic Pattern Matching

The reader knowledgeable in SPARQL recognizes graph-homomorphic pattern matching as the semantics of basic graph patterns and conjunctions thereof [4, 27]. For more sophisticated queries, evaluation processes become more complex, e.g. by setting up implications between (sub-)patterns and requiring matches to satisfy conditions over the matched literals. For example, when looking for every book/movie released after $\underline{1987}$, the very first step is to conceptualize books/movies by those structures representing books/movies in a database, e.g. the one in Fig. 1, and only secondly, the release year is used to differentiate resulting matches according to the query. Thus, a graph database query formulation process is, at least virtually, split up into two core tasks: (1) identifying structures representing the entities the query speaks about and (2) additional constraints and articulated dependencies between the entities of the query important for the actual evaluation. For both tasks, current graph query languages provide support for expressing (1) and (2) in mixture that often hardly allows to tear them apart. Finding structures representing certain entities in a graph database, i.e. step (1), is a non-trivial task, since there is no a-priori schema restricting the sort of information (i.e. triples) inserted into the database.

As a consequence, not all objects must be equipped with key attributes, making it often complex to clearly identify entities of the same type.

In light of heterogeneity and sparsity inherent to graph data, we focus on how to support this quite creative task from a graph pattern matching perspective. We present different pattern matching strategies, compare them, and discuss the subordinate meaning a these matching relations mandate on graph patterns. Let us therefore recapitulate the foundational terms we deal within the forthcoming sections:

1 the graph database $\mathrm{DB} = (O_{\mathrm{DB}}, \Sigma_{\mathrm{DB}}, \longrightarrow_{\mathrm{DB}})$,
2 a graph pattern $P = (V_P, \Sigma_P, \longrightarrow_P)$,
3 a preorder ($\sqsubseteq$)[1] or equivalence ($\equiv$)[2] capturing the matching mechanism, and
4 a match graph $A = (O, \Sigma, \longrightarrow)$, a subgraph of DB such that $P \sqsubseteq A$ ($P \equiv A$, resp.).

3.1 Graph Isomorphism

Maybe the most prominent case makes pattern matching via graph isomorphism. For short, a graph isomorphism is a bijective graph homomorphism, implying that each node in the match graph has exactly one pattern node it matches.

Definition 2 (Graph Isomorphism) Let DB be a graph database, $A \preceq \mathrm{DB}$, and P a graph pattern. A *graph isomorphism between P and A* is a bijective function $\iota : V_P \to O_A$ such that ι is a graph homomorphism. $A \preceq \mathrm{DB}$ is a *graph-isomorphic match to P*, denoted $P \equiv_{\mathrm{iso}} A$, iff there is a graph isomorphism between P and A. $[\![P]\!]^{\mathrm{DB}}_{\mathrm{iso}}$ denotes the set of all such subgraphs.

Please note that, as our notation implies, $\equiv_{\mathrm{iso}}$ is an equivalence relation. In contrast to homomorphisms, $P \equiv_{\mathrm{iso}} A \preceq \mathrm{DB}$ does not imply that $P \equiv_{\mathrm{iso}} \mathrm{DB}$. Graph-isomorphic matches and graph-homomorphic ones are strongly corresponding, since $[\![P]\!]^{\mathrm{DB}}_{\mathrm{iso}} \subseteq [\![P]\!]^{\mathrm{DB}}_{\mathrm{hom}}$.

As exemplified in Sect. 1, graph-isomorphic matching is often too strict when sparsity and heterogeneity appear. Matches to a pattern contain exactly the information asked for. The example pattern given in Fig. 2 may ask for all books, identified by `wrote` and `ISBN` relationships (but exactly one of each). If one of the edges is missing, as e.g. in Fig. 1a but also in Fig. 1b, the subgraph is not considered a match. Further note that the graph in Fig. 1c only partially matches, ignoring node `The Lord of the Rings`. Hence, graph isomorphism should always be asked in precision-orientation applications. Simulations, as introduced and discussed during the next subsection, constitute a weaker relation, allowing for recall-oriented querying.

Related Work 1 *Although homomorphisms and isomorphisms are common to many graph query languages* [4, 20], *the degree of structural identity is often assessed as too restrictive for many applications, see e.g.* [6, 10, 11]. *Furthermore, the core task of finding matches is to solve the respective subgraph problem, known to be* NP*-complete* [8, 16].

Besides promoting pattern matching relations more liberal than (sub-)graph isomorphism, we need to discuss tractability of the proposed matching relations. Such a discussion is inevitable, since solving the *evaluation problem* (i.e. the second step of query answering) alone for query languages may be quite time- and space-consuming, e.g. SPARQL's evaluation problem is PSPACE-complete [27].

Detour 2 *Up to now, we have learned about two pattern matching relations, a preorder (graph homomorphism) and an equivalence (graph isomorphism). All of the forthcoming pattern matching relations have formulations as preorders and equivalences. We choose equivalences here, because all the preorders $\sqsubseteq$, we would discuss, share the property that*

$$P \sqsubseteq A \preceq \mathrm{DB} \quad \text{implies} \quad P \sqsubseteq \mathrm{DB}\,, \qquad (1)$$

i.e. whenever we identify any subgraph due to pattern P and $\sqsubseteq$, then also the whole database is identified to match the pattern (cf. graph homomorphic matching). There is hardly any application that needs the whole database as answer to almost any query, or phrased differently, in these applications, no information about the stored information is needed anyway.

The decision in favor of the equivalence matching relations comes to the price of immediately facing subgraph equivalence problems which in turn give rise to intractability of algorithmic solutions. In all our cases, NP*-completeness results are obtained using the pattern graph as input for the problem* [8, 16, 26]. *Since queries and therefore patterns are small relative to the usual size of a graph database, we may at least rely on an adequate data complexity* [29]. *Therefore, while studying the equivalence matching mechanisms, we focus on what a pattern expresses under these matching relations. In cases of huge patterns, we provide insights from related work tweaking the preorders such that we evade problem (1) while maintaining tractability of the associated matching problems.*

[1] A preorder is a binary relation $\sqsubseteq \subseteq A \times A$ that is reflexive (i.e. for all $a \in A$, $a \sqsubseteq a$) and transitive (i.e. for all $a, b, c \in A$, $a \sqsubseteq b$ and $b \sqsubseteq c$ implies $a \sqsubseteq c$).

[2] An equivalence relation is a preorder $\equiv \subseteq A \times A$ that is symmetric (i.e. for all $a, b \in A$, $a \equiv b$ implies $b \equiv a$).

3.2 Simulations

Dating back to its origins in 1971, a simulation is a binary relation over the node sets of two graphs [24]. In Milner's paper, nodes represent states of a program. The goal was to capture the intuitive meaning of simulation in an algebraic manner. Whenever we have two program states related by a simulation, each step (i. e. outgoing edge) from the first program state must be mimicked by the second one such that the successor states form a pair in the simulation.

The definition may be directly used in a (graph) database setting. Indeed, it has been proposed to compare conceptual data graphs very early as in [2]. However, exchanging graphs representing program states for graphs representing data and their inter-linkage implies adaptations to the notion of simulation. First, while for programs, only considering outgoing edges from nodes/states is sufficient, for data nodes also incoming edges are relevant. A simulation variant respecting incoming and outgoing edges of simulated nodes has been coined to *dual simulation* by Ma et al. [21].

Detour 3 *A second important difference between program graphs and graph databases is the existence/absence of an initial state. Since a program needs to start somewhere, the initial states of two program graphs under investigation need to be in the simulation. On one hand, we would not dare to introduce such a notion to graph databases, since it would restrict the information we obtain as query answers massively. On the other hand, the existence of initial states and their handling make simulation pattern matching a non-trivial notion. A natural way to reach the same degree of non-triviality is to require a simulation to be a non-empty set. Alternatively, we might also require that any node of the match graph simulates some node of the pattern or that the simulation satisfies a maximality criterion (cf. [22]).*

As a last point, we mention a peculiarity of the original definition of simulation. A node with no outgoing edges can be simulated by any node. This is because only outgoing edges need to be simulated. When considering also incoming edges of nodes, this problem is reduced, since every edge needs to be reflected in a match. Only disconnected nodes, i. e. nodes with neither incoming nor outgoing edges, may be simulated by any other node and thus by any other graph.

Therefore, what Ma et al. [21, 22] call dual simulation is the only simulation notion we put forward in this paper, here simply called *graph simulation*.

Definition 3 Let $G_i = (V_i, \Sigma_i, \longrightarrow_i)$ $(i = 1,2)$ be two graphs. A *graph simulation between* G_1 *and* G_2 is a non-empty relation $\mathcal{S} \subseteq V_1 \times V_2$ s.t. for any $(v_1, v_2) \in \mathcal{S}$,

1 $v_1 \xrightarrow{a}_1 w_1$ implies $\exists w_2 \in V_2 : v_2 \xrightarrow{a}_2 w_2$ and $(w_1, w_2) \in \mathcal{S}$, and
2 $u_1 \xrightarrow{a}_1 v_1$ implies $\exists u_2 \in V_2 : u_2 \xrightarrow{a}_2 v_2$ and $(u_1, u_2) \in \mathcal{S}$.

Write $G_1 \sqsubseteq_{\text{sim}} G_2$, iff a graph simulation between G_1 and G_2 exists. Define $G_1 \simeq G_2$ iff $G_1 \sqsubseteq_{\text{sim}} G_2$ and $G_2 \sqsubseteq_{\text{sim}} G_1$.

For a graph database DB and pattern P, define $[\![P]\!]_{\text{sim}}^{\text{DB}}$ as the set of all simulating matches to P.

For a detailed example on how to work with simulation relations to match subgraphs, we refer the reader to our previous work, especially Sect. 3.2 in [23]. Please note that the witnesses for graph simulating matches are pairs of simulations, one showing that the pattern is simulated by the match and one showing that the pattern simulates the match. For establishing simulation equivalence, the respective simulations $\mathcal{S}_1, \mathcal{S}_2$ do not need to coincide in that $\mathcal{S}_1 = \mathcal{S}_2^{-1}$. In case they coincide, they are called *bisimulations*.

Reconsidering the pattern in Fig. 2, the graph in Fig. 1c completely matches the pattern under simulation equivalence, although `Lord of the Rings` misses the identifying ISBN attribute. This is due to our simulation equivalence check, asking also the pattern to simulate the match, because `Lord of the Rings` is simulated by pattern node `B`. What we learn from this example, though, is that simulation does not strictly rule out objects as part of a match graph with missing relations (in terms of incoming/ outgoing edges), as long as some subgraph of a match simulates the whole edge structure of a pattern and the additional edges have something to do with the pattern. From a modeling perspective, the pattern interpreted under graph simulation asks for *all book authors together with their works (i. e. not necessarily books identified by an ISBN)*. Alternatively, we may find that an item that has been written by a person, who clearly qualifies as a book author, is probably a book. Further note that if several authors wrote one book, then also all co-authors belong to a match. Hence, one single pattern may have simulating matches for diverse queries.

Neither simulation nor graph isomorphism captures absence of relationships as a requirement expressed in a query. Suppose we want to identify only movies and their authors. One way to identify authors is to have a `wrote` relationship between an author and another object, which shall be a movie. Compared to books from the previous examples, we may use that movies do not feature an ISBN. Hence, by excluding edges labeled by `ISBN` we purely get non-book objects as results. Consider therefore the altered pattern depicted in Fig. 3. Herein, the striked-through edge indicates that there is no outgoing edge labeled `ISBN` from node `B`, being only made explicit for illustrative purposes. Please note that from a graph simulation perspective, the graph

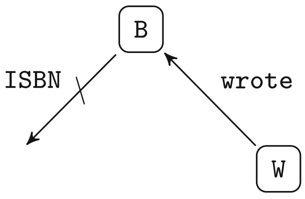

Fig. 3 A Graph Pattern with Failure of *ISBN* in *B*

depicted in Fig. 1b is a match to the pattern in Fig. 3 and also the subgraph excluding the ISBN of Fig. 1c matches up to graph simulation. Requiring missing edges to be reflected by pattern and match is the subject of Sect. 4, where we develop a database theory of failures.

Related Work 2 *As mentioned earlier, simulation equivalence matching is an* NP*-complete problem [26]. Tractability is rectified by considering the preorder* $\sqsubseteq_{\text{sim}}$ *[17, 22]. Unfortunately,* $P \sqsubseteq_{\text{sim}} A \preceq \text{DB}$ *implies* $P \sqsubseteq_{\text{sim}} \text{DB}$*. Ma et al. [21] proposed simple syntactic restrictions on the matches to be considered in that the size of a match graph is bounded by the size of the pattern. Furthermore, they require that every node of the match graph simulates some node of the pattern. The resulting pattern matching relation is called strong simulation, playing along with graph isomorphic/homomorphic matching. In fact, Ma et al. compared the matches obtained due to strong simulation with those by graph isomorphism. It turns out that up to 80% of strong simulation matches are also graph-isomorphic ones [22]. Beyond that promising connection to graph isomorphism, strong simulation has been applied in a query paradigm called Exemplar Queries [25], presenting those results similar to an example the user provides as input.*

4 A Failures Theory for Graph Databases

We have seen pattern matching relations querying for data in different resolutions. In this section, we develop a theory of failures for graph databases to incorporate statements of absent relationships. Of course, users formulating such queries must be aware of the semantics of a missing relationship in the database. It might mean that the object must not have such a relationship but the information can also simply be missing due to extraction imprecision or any other reason of that kind. The pattern depicted in Fig. 3 indicates the wish to express that we are looking for all written items (hence, the `wrote` relationship) that are not books (missing `ISBN`). Recall that the struck through edge is only made explicit for illustrative purposes. In order to take the missing edge into account, we define the notion of failures and then build a simulation notion incorporating negated relationships. As in case of graph simulation, the original notions appeared as part of the analysis of abstract programs/program graphs [5].

4.1 Failures

As a first attempt, a node v fails a relation symbol a if $v_1 \not\xrightarrow{a}$, i.e. v has no outgoing edge labeled a. However, as discussed for graph simulation in Sect. 3, the direction of relationships, here a failure, plays an important role in the study of graph databases. We therefore observe each relation symbol a in two shapes. First, we have a failure $\overrightarrow{a}$ of v, meaning that there is no outgoing edge from v labeled a, i.e. as before. Second, we also encode failures due to ingoing edges by $\overleftarrow{a}$. To this end, we denote by $\not\xrightarrow{a} v$ that there is no node w such that $w \xrightarrow{a} v$. Since the relation symbols relevant for the query are specified by the graph pattern, the notion of failures is parameterized by some alphabet Γ, later replaced by the concrete set of relation symbols of the pattern graph, i.e. $\Gamma = \Sigma_P$. In a graph, a node v *fails* Γ if for some $a \in \Gamma$, $v_1 \not\xrightarrow{a}$ or $\not\xrightarrow{a} v$, expressed by the notation introduced above, i.e. $\overrightarrow{a}$ or $\overleftarrow{a}$.

Definition 4 Let $G = (V, \Sigma, \longrightarrow)$ be a graph, $v \in V$, $a \in \Sigma$, and Γ any finite alphabet alphabet. Define the directed alphabet as $\overleftrightarrow{\Gamma} := \{\overrightarrow{a}, \overleftarrow{a} \mid a \in \Gamma\}$. α is a *Γ-failure of v* iff $\alpha \in \overleftrightarrow{\Gamma}$ with $v_1 \not\xrightarrow{a}$ if $\alpha = \overrightarrow{a}$ and $\not\xrightarrow{a} v$ if $\alpha = \overleftarrow{a}$. $\mathcal{F}_G^\Gamma(v)$ denotes the set of all Γ-failures of v.

As an example, reconsider the graph pattern P given in Fig. 3. Regarding $\Gamma = \{\texttt{wrote}\}$, node B has the Γ-failure $\overrightarrow{\texttt{wrote}}$. In contrast, $\overleftarrow{\texttt{wrote}} \notin \mathcal{F}_P^\Gamma(\texttt{B})$.

4.2 Failure Simulation

We exemplify our theory of failures with the notion of graph simulation. Other matching mechanisms may be extended similarly. In this example, an object not only has to simulate a pattern node but must further reflect its failures, encoding explicit statements of absent relationships. Following the general concept of Γ-failures defined above, we may directly derive the notion of a graph simulation combined with Γ-failures as follows. Let $G_i = (V_i, \Sigma_i, \longrightarrow_i)$ $(i = 1{,}2)$ be two graphs and Γ be any finite alphabet. A non-empty relation $\mathcal{S} \subseteq V_1 \times V_2$ is a *Γ-failure simulation between G_1 and G_2* iff $\mathcal{S}$ is a graph simulation (cf. Def. 3) a nd for all $(v_1, v_2) \in \mathcal{S}$, $\mathcal{F}^{\Gamma}_{G_1}(v_1) = \mathcal{F}^{\Gamma}_{G_2}(v_2)$. G_2 *Γ-failure simulates* G_1, denoted $G_1 \sqsubseteq^{\Gamma}_{\text{fail}} G_2$, iff there is a Γ-failure simulation between G_1 and G_2. Γ-failure simulations are sufficient for the examples we have given so far. Assume now $\Gamma = \{\texttt{wrote}, \texttt{ISBN}\}$ for pattern P in Fig. 3. For the nodes of P we obtain Γ-failure sets

$\mathcal{F}^{\Gamma}_{P}(\texttt{W}) = \{\overleftarrow{\texttt{ISBN}}, \overleftarrow{\texttt{wrote}}, \overrightarrow{\texttt{ISBN}}\}$ and

$\mathcal{F}^{\Gamma}_{P}(\texttt{B}) = \{\overleftarrow{\texttt{ISBN}}, \overrightarrow{\texttt{wrote}}, \overrightarrow{\texttt{ISBN}}\}$.

Since from an application-perspective, queries formulated by users dictate which relationships are relevant, the graph pattern specifies the relation symbols, either used as matching or failure information. While the former is expressed by explicitly including an edge in the pattern, the latter can only be achieved by excluding certain edges. The graph pattern alphabet in Fig. 3 needs to include `wrote`, due to the labeled edge between `W` and `B`, as well as `ISBN`, due to the indicated absent relationship. For this concrete pattern P, with query semantics of obtaining authors and their works that are not books, it is sufficient to have $\Sigma_P = \{\texttt{wrote}, \texttt{ISBN}\}$. In the query process, failures of each node of a pattern are evaluated w. r. t. its own alphabet of relation symbols. Hence, the failures specified by pattern P of the example coincide with the above-mentioned ones for Γ.

Regarding matches, i. e. subgraphs A of a database DB, we need to take special care about the subgraph relationship, which we defined quite liberally. Consider our graph pattern P requesting non-books from the graph database. We surely want to rule out matches including `The Hobbit` (cf. Fig. 1c). Unfortunately, the subgraph relation allows for picking arbitrary nodes from the graph database, regardless of their importance for a given pattern. In this case, we would simply leave out the ISBN of `The Hobbit`. Together with `J. R. R. Tolkien` and the connecting relation `wrote`, `The Hobbit` constitutes a match up to failure simulation, although the graph database associates relation *ISBN* with it. If we incorporate the database knowledge about nodes of match graphs, `The Hobbit` is no longer considered a match. We achieve this by regarding Σ_P-failures of the match graph nodes w. r. t. the database, i. e. instead of using $\mathcal{F}^{\Sigma_P}_{A}$ we compare the failures of the pattern nodes with that from the database by $\mathcal{F}^{\Sigma_P}_{\text{DB}}$.

Definition 5 Let DB be a graph database, $A \preceq \text{DB}$, and P a graph pattern. A is a *failures simulating match* to P, denoted $P \equiv_{\text{fail}} A$, iff $P \sqsubseteq_{\text{sim}} A$ (by graph simulation $\mathcal{S}_1$) and $A \sqsubseteq_{\text{sim}} P$ (by graph simulation $\mathcal{S}_2$) such that for all $(p, o) \in \mathcal{S}_1$, $\mathcal{F}^{\Sigma_P}_{P}(p) = \mathcal{F}^{\Sigma_P}_{\text{DB}}(o)$ and for all $(o, p) \in \mathcal{S}_2$, $\mathcal{F}^{\Sigma_P}_{\text{DB}}(o) = \mathcal{F}^{\Sigma_P}_{P}(p)$.

By using failure simulation, `The Hobbit` never occurs in a match graph. The theory of failures for graph databases is now complete. Thereby we followed the lines of combining the failures semantics with simulation semantics, as summarized in van Glabbeek's spectrum [13]. The resulting relation is there called *ready simulation* with the property that any ready simulating match of a pattern is also a match up to simulation and failures. This is a property we cannot repeat for our failure simulation matching, because we also evaluate failing relations of those matched nodes not being part of the subgraph, thus breaking with the definition we discussed at the beginning of this subsection. Compared to the analysis in our previous work [23], this break reveals as a necessary step to make failures a useful notion for retrieving answers from a graph database.

Regarding existing query languages, we find explicit statements for absent information in `NOT EXISTS` clauses of SQL and SPARQL (version 1.1). For instance, by

```
SELECT ?A ?B WHERE {
?A wrote ?B.
FILTER NOT EXISTS { ?B ISBN ?I .}}
```

we express a similar graph pattern as the one in Fig. 3 in a SPARQL syntax. This query asks for all matches to *?A* and *?B*, being in a *wrote* relationship, such that *?B* does not feature an `ISBN`. Please note that our graph pattern in Fig. 3 further requires absence of a `wrote` edge in *?B* as well as absence of `ISBN` in *?A*. Regarding the real-world context, illustrated by this example, the restrictions required by the graph pattern are quite realistic, since books only rarely write other books. In order to faithfully characterize the above-mentioned query by a graph pattern we needed to extend the notion of failure simulation.

We would like to stress the fact that the failures simulation matching (cf. Def. 5) constitutes a graph matching mechanism, promising interesting properties w. r. t. graph queries. Compared to the other relations, we observe that it disregards subgraphs as matches, the other relations (graph isomorphism, homomorphism, and simulation) consider a match, e. g. subgraphs containing `The Hobbit` as matches for the pattern in Fig. 3. This is because failure simulation uses information that is not explicitly represented in the graph structure (i. e. by nodes and edges), the only source to obtain a matching decision for the other mechanisms. Fortunately, the information needed to evaluate failures is bounded, since we require Σ_P to be finite. Regarding complexity, we end up with the same bounds as for graph simulation. Surely, the described extensions due to Ma et al. [22] pulls failure simulation back to tractability. However, side-effects the failures pose upon the syntactic restrictions need to be studied in future work.

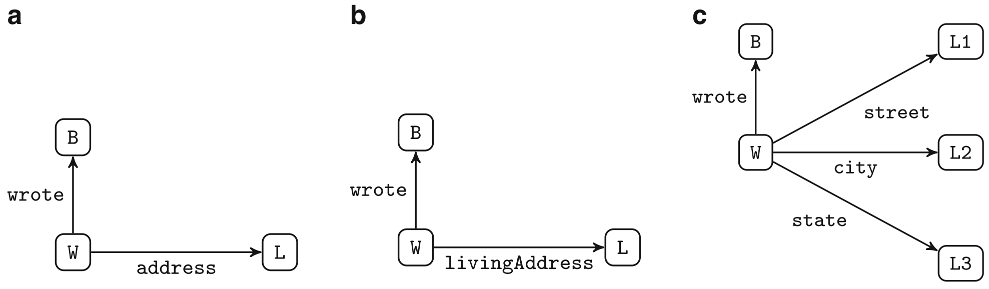

Fig. 4 Example Graph Patterns

The next section discusses problems occurring in graph database pattern matching that cannot be handled by solely relying on the pattern matching relations defined in the last two sections.

5 Heterogeneity and Sparsity in Practice

In the last two sections, we have introduced structural approaches based on graph pattern matching to capture the schema of an entity. These ideas enable dealing with sparsity issues in graph databases, by choosing an appropriate matching mechanism. For example in sparse databases, simulations will return more results than homomorphism or simulation failure-based querying. These two should be used in non-sparse databases since, in general, they return more precise result sets there. However, the quality of query results that would be desired is heavily application-dependent. Therefore, the choice of the right pattern matching relation usually needs some manual tuning. Recently, techniques to measure data completeness automatically [12] have been proposed. By employing these techniques, it would be possible to automatically adapt the matching mechanism depending on data sparsity. Beyond data sparsity, the choice for a specific matching mechanism is also dependent on the problem to be solved. In [23], we therefore outlined important differences of simulation-based and failure-based pattern matching.

In real-world graph databases, knowledge about entities additionally shows data heterogeneity going hand in hand with data sparsity. Heterogeneity alone can hardly be captured by structural graph pattern matching alone. Therefore, this section deals with solutions for solving data heterogeneity with existing techniques from the Semantic Web and give a more detailed view on sparsity issues in graph databases. Furthermore, we briefly discuss the role the Semantic Web plays in solving emerging heterogeneity and sparsity problems.

5.1 Data Heterogeneity

Data heterogeneity not captured by structural matching graph patterns alone, is synonymous relationship names. The pattern in Fig. 4a models an author and her address. Similarly, the same information may be expressed by the pattern in Fig. 4b. The graph patterns are structurally similar but use different relationship symbols, although `address` and `livingAddress` might have the same meaning for an application or user. A more complex case is sketched in Fig. 4c. Here, the same address information as before is modeled completely different to Fig. 4a, b. The pattern uses three single relationships `street`, `city` and `state` to express (possibly) the same semantics as before. By solely relying on our graph pattern matching relations, we are not able to capture any of these examples.

In a Semantic Web setup, several ways exist to overcome the mentioned heterogeneity problems originating from the conceptual differences shown in the previous paragraph. In SPARQL 1.1, entailment regimes [15] are introduced as an extension of query semantics by entailment rules. Entailment regimes basically rely on additional ontological knowledge in the form of *Web Ontology Language* (OWL) and RDF-Schema (RDF-S). Typical Semantic Web ontologies comprise:

- entity type information (*rdfs:type*),
- class hierarchies (*rdfs:subClassOf*),
- mappings between properties (*owl:equivalentProperty*),
- relationship hierarchies (*rdfs:subPropertyOf*), and
- instances (*owl:sameAs*).

This knowledge may be used to map the graph pattern from Fig. 4a, b, if it is known upfront that `address` and `livingAddress` are equivalent properties. Unfortunately, this idea requires an extensive set of ontological information which usually requires a manual conceptualization of the graph database.

Several ideas from the Semantic Web can be directly integrated into our pattern matching relations [14]. For example, the *owl:equivalentProperty* mappings could easily be integrated into the definitions of all matching relations. Addressing the issue illustrated by Fig. 4c means to relate single relationships/sub-graphs with other sub-graphs. Initial ideas to integrate such pattern matchings within matching mechanisms may be found in so-called *isotactics* [28]. However, we strongly believe that, again, hand-crafted schema information is necessary to make isotactics a valuable asset to graph database querying. Hence, as future work, we plan to address heterogeneity issues by an automatic schema extraction and matching mechanism used prior to the actual query process.

5.2 Data Sparsity

We have already addressed some sparsity issues in various examples throughout the paper. We have also demonstrated that pattern matching may be recall-oriented (e. g. simulation), also returning unexpected query results, and others are rather precision-oriented (homomorphism or failures). Similarly to heterogeneity issues mentioned in the previous subsection, sparsity problems cannot be overcome by graph pattern matching alone. In the same fashion, we believe that additional techniques for automatic schema extraction or property-based data access can be used as a preprocessing step for the graph query process. Approaches to automatically create schema information for relational databases have already been developed in the field of *data profiling* [1]. Profiling techniques analyze semantic properties of the data to infer schema information, e. g. key candidates, type information, data domains and functional dependencies. This information can be used to better deal with heterogeneity and sparsity issues where only few schema information is known. In the field of Semantic Web, there are approaches dealing with missing schema information for entity queries. Homoceanu et al. came up with the idea of identifying entity types by a set of typical attributes [18, 19]. They have shown that missing schema information in graph databases must not be a problem for querying, because it is possible to automatically determine a small set of attributes that may be used to uniquely distinguish entity types from each other. Particularly, they show that their approach finds around 5 typical attributes for movies and books potentially used by entity queries. However, existing approaches are not yet ready to deal with extremely sparse data, as often found in modern knowledge graphs.

Therefore for future work, we plan to further investigate this line of research by transferring more concepts from relational databases to graph databases, and build approaches automatically identifying entity types by learning identifying properties to enhance graph pattern matching for graph queries in sparse and heterogeneous data environments.

6 Conclusion

In this article, we discussed different conceptual notions w. r. t. querying graph databases. We therefore went from precision-oriented graph pattern matching relations over recall-oriented ones, finally yielding a discussion of how negation may be incorporated. To this end, we developed a theory of failures, enriching arbitrary pattern matching relations by the notion of failures, here exemplified with graph simulation. To the best of our knowledge, there is no intensionally defined pattern matching relation supporting negation of relations. We discussed what matching mechanisms mandate as structural semantics of entities, from a user perspective (what the user wants) as well as from a graph database retrieval perspective (what the user gets).

Surely, by the presented graph pattern matching mechanisms we are far away from solving data sparsity and heterogeneity issues in graph databases. We strongly conjecture, however, that these matching mechanisms combined with powerful tools from, for instance, the Semantic Web may mean a major step towards a schema-aware graph query processor.

Besides the points for future work we mentioned in earlier sections, we give one more outlook here. We have looked, so far, at simulation-based graph pattern matching. Regarding the possibilities program graph analysis offers, path-based pattern matching relations are readily available. By these, we would also cover path queries, one of the strongest features of modern graph query languages. In [23], we have analyzed a first notion of such a trace-based matching relation in a graph database context. We were also interested in the notion of failures and reused one of the standard definitions (i. e. without adaptations) from [13]. Further analysis of the interplay of path queries from graph query languages and these trace-based matching relations might open our considerations for more query types.

References

1. Abedjan Z, Golab L, Naumann F (2015) Profiling relational data: a survey. Vldb J 24(4):557–581
2. Abiteboul S, Buneman P, Suciu D (1999) Data on the web: from relations to semistructured data and XML. Morgan Kaufmann, Burlington
3. Angles R, Gutierrez C (2005) Querying RDF data from a graph database perspective. In: Gómez-Pérez A, Euzenat J (eds) ESWC 2005, LNCS. Springer, Berlin, Heidelberg, pp 346–360 https://doi.org/10.1007/11431053_24
4. Angles R, Arenas M, Barceló P, Hogan A, Reutter J, Vrgoč D (2017) Foundations of modern query languages for graph databases. Acm Comput Surv 50(5):68:1–68:40. https://doi.org/10.1145/3104031
5. Brookes SD, Hoare CAR, Roscoe AW (1984) A theory of communicating sequential processes. JACM 31(3):560–599. https://doi.org/10.1145/828.833

6. Brynielsson J, Högberg J, Kaati L, Mårtenson C, Svenson P (2010) Detecting social positions using simulation. ASONAM 2010, pp 48–55 https://doi.org/10.1109/ASONAM.2010.52
7. Chen PPS (1976) The entity-relationship model – toward a unified view of data. ACM Trans Database Syst 1(1):9–36. https://doi.org/10.1145/320434.320440
8. Cook SA (1971) The complexity of theorem-proving procedures. Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71. ACM, New York, pp 151–158 https://doi.org/10.1145/800157.805047
9. Dong X, Gabrilovich E, Heitz G, Horn W, Lao N, Murphy K, Strohmann T, Sun S, Zhang W (2014) Knowledge vault: a web-scale approach to probabilistic knowledge fusion. Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14. ACM, New York, pp 601–610 https://doi.org/10.1145/2623330.2623623
10. Fan W (2012) Graph pattern matching revised for social network analysis. ICDT 2012. ACM, New York, pp 8–21 https://doi.org/10.1145/2274576.2274578
11. Fan W, Li J, Ma S, Tang N, Wu Y, Wu Y (2010) Graph pattern matching: from intractable to polynomial time. PVDLB Endow 3(1):264–275. https://doi.org/10.14778/1920841.1920878
12. Galárraga L, Razniewski S, Amarilli A, Suchanek FM (2017) Predicting completeness in knowledge bases. Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, WSDM '17. ACM, New York, pp 375–383 https://doi.org/10.1145/3018661.3018739
13. van Glabbeek RJ (1990) The linear time - branching time spectrum. In: Baeten JCM, Klop JW (eds) CONCUR 1990. Springer, Berlin, Heidelberg, pp 278–297 https://doi.org/10.1007/BFb0039066
14. van Glabbeek R, Goltz U (1989) Equivalence notions for concurrent systems and refinement of actions. In: Kreczmar A, Mirkowska G (eds) Mathematical foundations of computer science 1989. Springer, Berlin, Heidelberg, pp 237–248
15. Glimm B, Krötzsch M (2010) Sparql beyond subgraph matching. The Semantic Web – ISWC 2010. Springer, Berlin, Heidelberg, pp 241–256
16. Hell P, Nešetřil J (1990) On the complexity of h-coloring. J Comb Theory Ser B 48(1):92–110. https://doi.org/10.1016/0095-8956(90)90132-J
17. Henzinger M, Henzinger T, Kopke P (1995) Computing simulations on finite and infinite graphs. In: FOCS 1995. IEEE Computer Society, pp 453–462 https://doi.org/10.1109/SFCS.1995.492576
18. Homoceanu S, Balke WT (2015) A chip off the old block - extracting typical attributes for entities based on family resemblance. Database systems for advanced applications. Springer, Cham, pp 493–509
19. Homoceanu S, Wille P, Balke WT (2013) Proswip: property-based data access for semantic web interactive programming. Proceedings of the 12th International Semantic Web Conference - Part I, ISWC '13. Springer, New York, pp 184–199 https://doi.org/10.1007/978-3-642-41335-3_12
20. Lee J, Han WS, Kasperovics R, Lee JH (2012) An in-depth comparison of subgraph isomorphism algorithms in graph databases. PVLDB Endow 6(2):133–144. https://doi.org/10.14778/2535568.2448946
21. Ma S, Cao Y, Fan W, Huai J, Wo T (2011) Capturing topology in graph pattern matching. PVLDB Endow 5(4):310–321. https://doi.org/10.14778/2095686.2095690
22. Ma S, Cao Y, Fan W, Huai J, Wo T (2014) Strong simulation: capturing topology in graph pattern matching. Acm Trans Database Syst 39(1):1–4. https://doi.org/10.1145/2528937
23. Mennicke S, Kalo JC, Balke WT (2017) Querying graph databases: what do graph patterns mean? Springer, Cham, pp 134–148 https://doi.org/10.1007/978-3-319-69904-2_11
24. Milner R (1971) An algebraic definition of simulation between programs. Proceedings of the 2Nd International Joint Conference on Artificial Intelligence, IJCAI'71. Morgan Kaufmann Publishers, San Francisco, pp 481–489
25. Mottin D, Lissandrini M, Velegrakis Y, Palpanas T (2016) Exemplar queries: a new way of searching. VLDB J 25(6):741–765. https://doi.org/10.1007/s00778-016-0429-2
26. Nardo LD, Ranzato F, Tapparo F (2009) The subgraph similarity problem. IEEE Trans Knowl Data Eng 21(5):748–749. https://doi.org/10.1109/TKDE.2008.205
27. Pérez J, Arenas M, Gutierrez C (2009) Semantics and complexity of sparql. Acm Trans Database Syst 34(3):16:1–16:45. https://doi.org/10.1145/1567274.1567278
28. Polyvyanyy A, Weidlich M, Weske M (2012) Isotactics as a foundation for alignment and abstraction of behavioral models. In: Barros A, Gal A, Kindler E (eds) Business process management. Springer, Berlin, Heidelberg, pp 335–351
29. Vardi MY (1982) The complexity of relational query languages (extended abstract). Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, STOC '82. ACM, New York, pp 137–146 https://doi.org/10.1145/800070.802186