

# Model-driven Schema Transformation for Graph Databases

Dominique Hausler<sup>1\*</sup>[0009–0004–2381–133X]✉,  
Torben Eckwert<sup>2\*</sup>[0000–0002–0501–9523], Meike Klettke<sup>1</sup>[0000–0003–0551–8389],  
Michael Guckert<sup>2</sup>[0000–0003–0523–3295], and  
Gabriele Taentzer<sup>3</sup>[0000–0002–3975–5238]

<sup>1</sup> University of Regensburg, Data Engineering Group, Bajuwarenstrasse 4, 93053 Regensburg, Germany {firstname.lastname}@ur.de

<sup>2</sup> KITE - University of Applied Sciences Mittelhessen, Wiesenstraße 7, 35390 Gießen, Germany {firstname.lastname}@mnd.thm.de

<sup>3</sup> Philipps-University Marburg, Biegenstraße 10, 35037 Marburg, Germany {lastname}@mathematik.uni-marburg.de

**Abstract.** In modern, agile software development continuous software evolution is the standard. Software changes can also affect the underlying database and its schema, resulting in schema evolution. Although schema evolution has traditionally been associated with relational databases, similar challenges arise in NoSQL databases, including graph databases. Graph databases are often described as schemaless because they do not enforce a predefined schema at the database level. However, they still have an implicit schema. When a graph database changes, the schema changes are typically implicit and often unknown, which can cause data quality problems. Making the evolutionary process between two schema versions explicit can help migrating the data to conform to the changed schema. In this paper, we present a model-driven approach that uses schema transformation techniques to reconstruct the evolutionary process between two schema versions. Our work is based on Graph Query Language, an ISO standard for graph databases, which makes our approach technology-independent. We present a semi-automated approach that first extracts the implicit schemas as graph models that ensure an abstract representation of the graph schemas. Next, we compare the graph models to identify possible transformations executed from the source and to the target model. An initial evaluation shows that schema extraction and schema transformation are both accurate and require only minimal manual user input.

**Keywords:** Model-driven engineering · Schema transformation · Schema evolution · Schema extraction · Graph database

## 1 Introduction

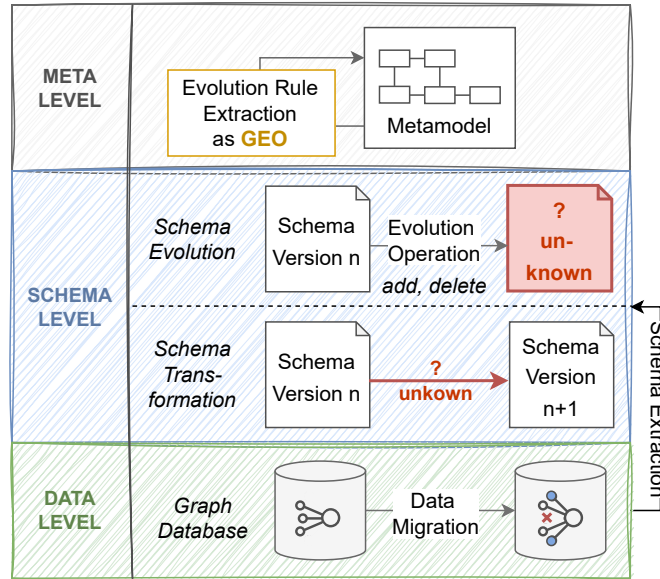
In modern, agile software development continuous software evolution is the standard. Changes to the software can also affect the domain model, which

---

\* Joined first authorship.

may require changes to the underlying database and its schema, resulting in schema evolution. Although schema evolution is traditionally associated with relational databases, similar challenges arise in NoSQL databases, including graph databases. Graph databases are known as schemaless, since they do not enforce a predefined schema at the database level. Nevertheless, graph databases still hold an implicit schema.

Graph schema transformation is realized through schema transformation techniques. These techniques refer to the process of converting a source schema into a target schema (see Figure 1). In contrast, schema evolution starts from a source schema and applies evolution operations. For example, [25] describes specific evolution operations for graph schemas, namely **add**, **rename**, **delete**, **copy**, **move**, **split**, **merge** and **transform**.



**Fig. 1.** Intertwine of schema evolution and schema transformation

**Problem Statement.** Changes to a graph database often occur during the development process, such as when adapting the data model to evolving requirements or introducing new features. These modifications can affect the implicit schema of the graph database. Making the evolutionary process between two schema versions explicit can support developers in understanding, documenting, and controlling the effects of such changes. By extracting their implicit schemas, an evolution can be visualized and analyzed in terms of the transformation steps required to convert the source schema into the target schema.

Schema transformation, as illustrated in Figure 1 presents a particularly challenging problem in the context of graph databases. These systems often contain heterogeneous data structures, such as optional properties, which increases the complexity of transformation tasks. The varying degree of disparity between the source and the target schema affects both the number and complexity of the evolution operations required to align them. It also affects the data level at which data migration occurs. Research on schema transformation for graph databases is still limited (see, e.g., [32, 17, 8]) and it is further complicated by the absence of standardized languages for describing schema transformations.

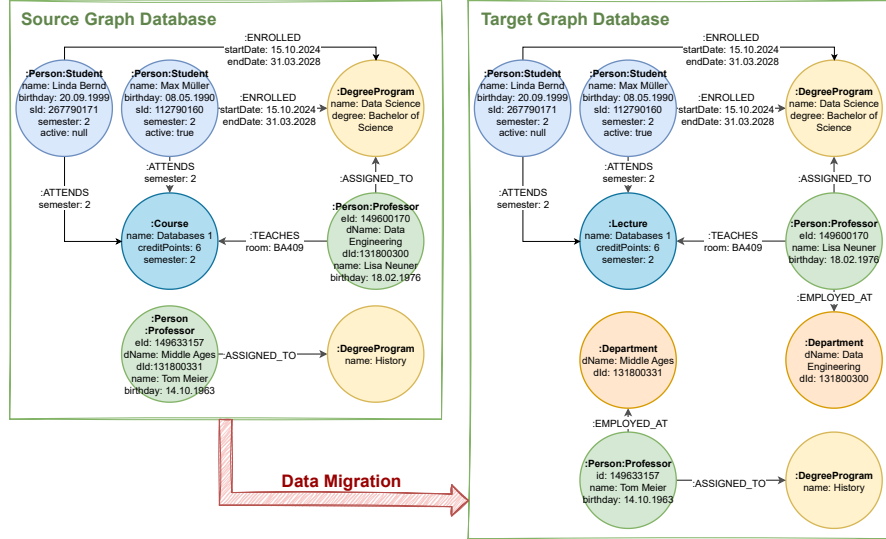
**Contribution.** Model-driven engineering [10], [19] (MDE) raises the level of abstraction by treating models as primary artifacts. In particular, modeling languages tailored to specific application domains can enhance the productivity and quality of software development. In this paper, we present a model-driven approach to adequately handle schema transformation in graph databases. To make our approach independent from specific graph database technologies, we base our approach on Graph Query Language (GQL) [27], the ISO standard for graph databases. GQL provides the conceptual basis for the meta level of our approach, which specifies Graph Evolution Operations (GEO) based on a metamodel. This metamodel defines the conceptual form of graph schemas based on GQL concepts (Figure 1: Meta level).

The implemented workflow is as follows: First, the implicit graph schema is extracted as a graph model. Our approach identifies schema transformations by analyzing the differences between the source and target graph models. To reduce uncertainty and avoid incorrect assumptions, we use a human-in-the-loop approach, which allows application developers to interact with the schema transformation process and resolve conflicts. While this reduces the degree of automation, it mitigates the risk of false assumptions and incorporates the developer’s intentions and domain knowledge. Finally, the detected evolution operations are returned to the developer to support informed decisions about the further database development. We implemented this model-driven schema transformation approach for Neo4j using Eclipse Modeling Tools [37]. An initial evaluation using three selected Neo4j example datasets shows that schema extraction and transformation are both accurate and require only minimal manual user input.

**Outline.** The structure of this paper is as follows: Section 2 introduces an illustrative example. Section 3 discusses the differences between bottom-up and top-down approaches to schema transformation and presents key concepts of graph databases. Our proposed solution, based on the illustrative example, is presented in Section 4. Section 5 describes the implementation of our prototype, which is evaluated in Section 6. Section 7 gives a brief overview of the related work. Finally, Section 8 concludes the paper and outlines future work.

## 2 Illustrative Example

To illustrate the model-driven schema transformation, we use an example graph database involving two versions of an academic domain that store course data.



**Fig. 2.** Source and target database of our illustrative example

The source and target schema are shown in Figure 2. Our goal is to determine the schema transformation and the associated data migration operations.

The source graph database has five distinct node labels. The **Professor** and **Student** nodes exhibit multi-labeling, meaning they are associated with more than one label. Multi-labeling is considered one of the challenges of schema transformation, so proper processing is essential. Furthermore, the example illustrates two kinds of null values: The **degree** property in **DegreeProgram** is optional, while the **active** property in **Student** labeled nodes refers to unknown information. All edges, except **ASSIGNED\_TO**, contain properties. In addition, the data set contains different types of constraints. (1) *Type constraints* are defined for the following properties: **startDate** and **endDate** of **ENROLLED** edges and **birthday** of **Person** nodes must be dates. (2) Each **Person** node must have a **name** property. (3) The **sId** and **eId** properties must be keys.

```

GE01: 'rename' label Course 'of' node 'with' label Course 'to'
      Lecture.
GE02: 'move' properties dName, dId 'from' node 'with' label
      Professor 'to' node 'with' label Department.

```

**Listing 1.1.** Two of the Graph evolution operations describing the schema evolution with the evolution language GEO

The target dataset now includes **Department** nodes, and all **Professor** nodes no longer have the properties **dName** and **dId** of departments. Additionally, the label **Course** was renamed to **Lecture**. Some constraints may no longer apply to the evolved graph database. Understanding how the schema evolved allows

application developers to make better decisions and ensure data quality. When our approach is applied, it produces a set of graph evolution operators that describe the schema evolution from the source to the target database. A selection of these operators is illustrated in Listing 1.1 using the evolution language GEO.

### 3 Concepts of Graph Databases

In this section, we analyze the advantages and disadvantages of using a schema-first versus a schema-second approach to graph databases, whereby the latter does not require any presumptions about the schema. Next, we discuss existing methods for realizing schema evolution and transformation in graph databases. Our focus is on property graphs as they are addressed by the ISO GQL.

#### 3.1 Schema-first vs. Schema-second Approach

NoSQL database systems, such as JSON and graph databases, provide the ability to manage schemas and semantic constraints. These extensions allow the usage of NoSQL databases with a schema-first approach, which involves defining the structure of entities, data types, and constraints. Since the schema management component checks all added data, a homogeneous dataset is created, which reduces the number of errors resulting from reorganization tasks. In case of a schema-second approach users can directly start inserting data without pre-defining a schema, causing an increased degree of heterogeneity, such as optional properties, or typos, since no schema management mechanism is utilized. In addition, a growing demand to reorganize and restructure the dataset emerges. Thus, an explicit schema must be derived from the data to ensure the quality of the graph data and to make constraints explicit.

We consider the latter for our approach, as it is considered to be the state-of-the-art usage of schemaless databases. With such a bottom-up usage, the complexity of the transformation task increases because no preliminary considerations about the schema were made.

#### 3.2 Schema Evolution and Schema Transformation

Evolution operations for graph databases such as moving a property, directly cause data migration. Data migration and schema evolution tasks may include optimization, reorganization of the data, or adjustments due to changes in requirements. When comparing the source and target graph databases at the data level (green box in Figure 1), the following evolution operations can be identified: (1) adding nodes with its edges, colored blue, and (2) deleting a node and its edge, shown by the red cross.

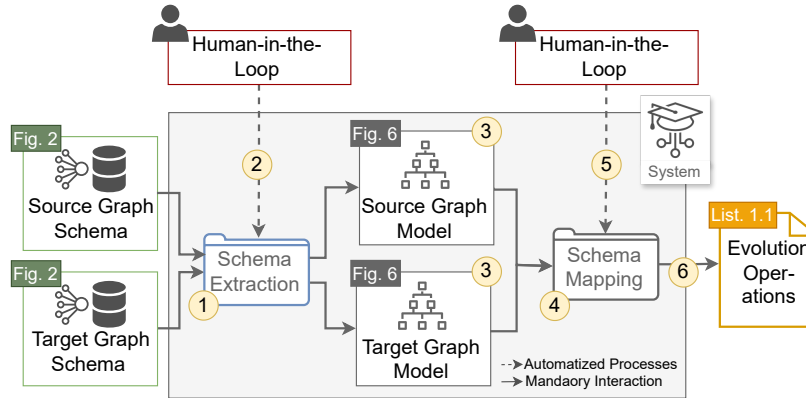
The schema level (i.e. the blue box in Figure 1) illustrates that both schema evolution and transformation return the same outcome, but differ in the known components. Schema evolution has the goal to obtain the target schema by a known source schema and specific evolution operations. In contrast, for schema

transformation, both the source and target schemas are given and the executed evolution operations are sought. Our approach focuses on transformation tasks by extracting the evolution operations conducted between schemas to introduce transparency on the conducted changes between schema versions.

## 4 Model-Driven Schema Transformation

In this section, we present our approach to model-driven schema transformation in graph databases. This presentation is based on the example in Section 2.

The key idea of model-driven schema transformation is the introduction of a meta level, as shown in Figure 1, which guarantees a generalizable approach. Schema transformation and evolution are handled directly based on technology-independent schema models. The meta level contains our metamodel, which controls schema transformation; it is shown in gray in Figure 1. Based on the metamodel, rules can be defined to specify valid changes of schema models. First, the source and target models are compared with each other to identify all changes. Then, model-based rule extraction is used to detect all executed evolution operations. The meta level enables the schema transformation process to be formulated in a reusable, technology-independent way.



**Fig. 3.** Workflow, showing all components of the model-driven schema transformation

**Workflow.** An overview of the implemented workflow for model-driven schema transformation is shown in Figure 3. Two versions of a graph database are provided as input at the data level (green boxes). Since graph databases are typically schemaless, schema extraction is required for Step ①. The implicit schema is extracted and transformed into graph models (see Step ③) that conform to our metamodel (the gray box in Figure 1). The schema extraction result for our example is shown in Figure 6. Schema extraction is actually performed twice, once

on the source database and once on the target database. If multi-labeling is detected, any resulting ambiguities are resolved by the human-in-the-loop (the red box in Figure 3) in Step ②, which allows for manual adjustments. These adjustments include assigning properties to an appropriate label (see Subsection 4.2). In Step ④, the differences between the two graph models are analyzed. The detected changes are then forwarded to the human-in-the-loop (Step ⑤), which allows for user validation and adjustment of the detected changes. Afterwards, the model mapping is updated accordingly. The detection of model changes is based on a set of evolution rules derived from the metamodel, which in turn is based on GQL. The detected model changes are then mapped to a graph evolution language called *Graph Evolution Operation* (GEO) [25] in Step ⑥. All GEOs applied are returned to the user as output (see Listing 1.1).

#### 4.1 Metamodel

To represent the graph schema at a higher level of abstraction, we have defined a metamodel using EMF Ecore [37], as shown in Figure 4. This metamodel specifies the key concepts of graph databases. To ensure system independency, the concepts are taken from the ISO standard GQL [27], while ensuring system-dependent, non-standard components such as constraints.

The root entity of our metamodel is *PropertyGraph*. Since we use GQL, every graph is considered a property graph. Consequently, we use the terms property graph and graph interchangeably. The enumeration *GraphType* specifies the type of graph and distinguishes between the following: (1) multigraphs, allowing multiple edges between node pairs, (2) directed, (3) undirected graphs, (4) mixed graphs, defining directions for edges, and (5) empty graphs.

Each graph contains zero or more *GraphItems*. The abstract class *GraphItem* serves as the common superclass for both *NodeType* and *EdgeType*. A *NodeType* defines the data type for nodes and consists of a set of zero or more *NodeLabels* as well as a set of *Property* elements. Similarly, an *EdgeType* defines the data type for edges, each of which may have zero or more *EdgeLabels* and a set of properties. Both *NodeLabel* and *EdgeLabel* inherit from the abstract superclass *Label*. Every *Label* and *Property* must have a unique name. All *NodeType*, *EdgeType*, and *Label* entities can have associated *Properties*, since *NodeTypes* may define properties that are not explicitly specified by any associated label. Each *Property* has a value type, represented by the abstract class *PropertyValue* and its concrete subclasses. The value type can be any supported property value type. If the type cannot be determined, a *UnionType* is used to allow values of arbitrary types. Each *PropertyValue* may have additional attributes specific to the type. For example, a *string type* has a *minlength* and a *maxlength*.

Although graph databases are schemaless, they often provide mechanisms to define constraints, which are not included in the standard, particularly when a certain degree of structure is required. Our metamodel supports different types of constraints, which are represented by the abstract class *Constraint* and its concrete subclasses. Each *Constraint* has a *name*, which serves as a unique identifier. The four subclasses represent the different kind of constraints:

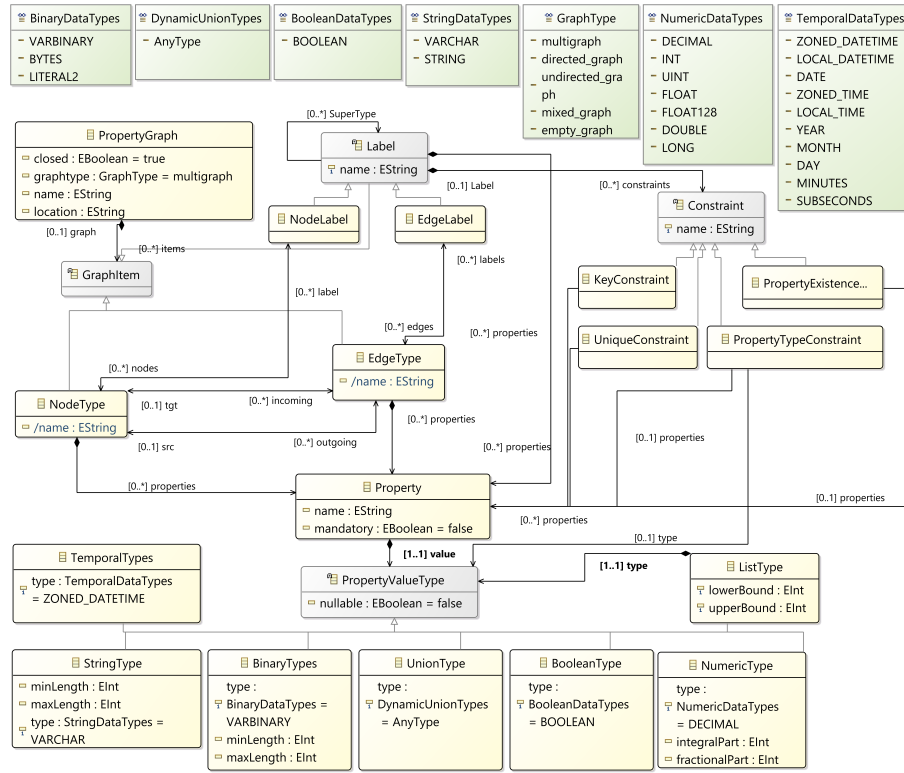


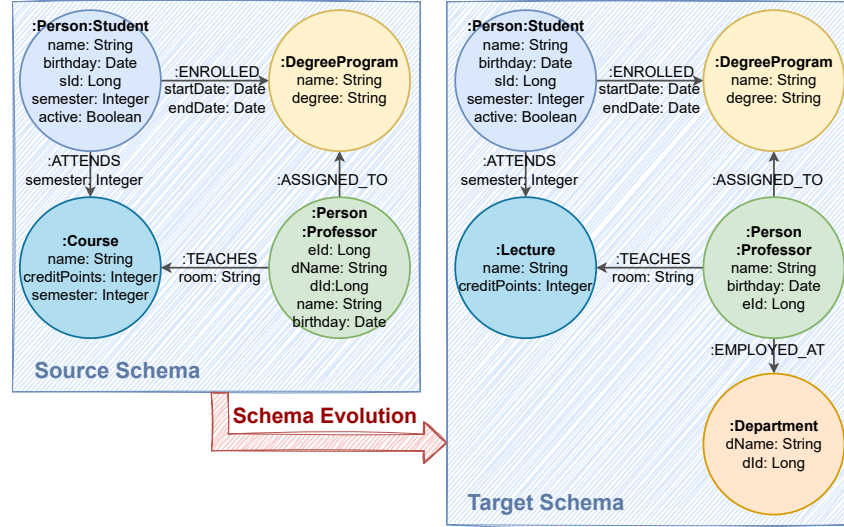
Fig. 4. Graph database metamodel

- *UniqueConstraint*: Unique property values across all specified entity types.
- *PropertyExistenceConstraint*: Requires existence of a given property within the stated entity type.
- *PropertyTypeConstraint*: Enforces that a property has a specific data type for all outlined entity types.
- *KeyConstraint*: Combines existence and uniqueness requirements.

## 4.2 Schema Extraction

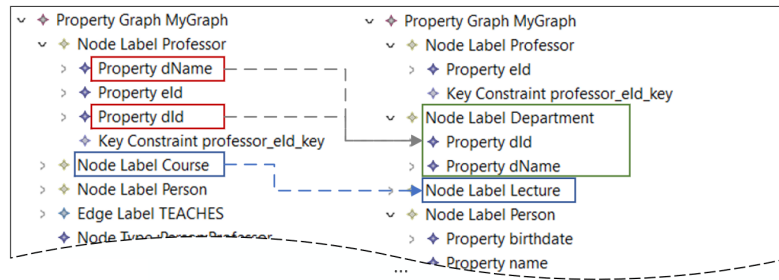
The implicit schemas of both database versions (see Figure 2) are extracted and transformed into instances of our metamodel (see Figure 3 Step ①). This process presents several challenges. For example, in the example dataset, two **Professor** and **Student** nodes are associated with multiple labels. During schema extraction, it is essential to determine which properties belong to which label. Figure 5 visualizes the schema extracted from the example dataset presented in Figure 2. In a multi-labeling context, if a label also appears in a single-labeled node, it could serve as a point of reference. However, such a node may not be





**Fig. 5.** Schema extraction, mapping and transformation of the illustrative example

present in the dataset. In our dataset, nodes labeled **Person** never occur without edges. However, all nodes that include this label, share the properties **name** and **birthday**. The constraints on these properties offer valuable insights, since they are associated with the **Person** label (see Paragraph Constraints in Section 2). Although this information could support a partially automated solution, such as a best-guess algorithm, we prefer a human-in-the-loop approach (see Figure 3 Step ②). This approach provides greater control to the user, eliminating the need for further system-sided validation. Moreover, the challenge of possibly optional properties for the **Person** label can only be solved by a human-in-the-loop approach, since there are no single-labeled **Person** nodes in the dataset.



**Fig. 6.** Graph database model: The original model is displayed on the left, while the edited model version is displayed on the right.

After the user has solved ambiguous cases, the graph models are created automatically. Figure 6 shows an excerpt of the graph model extracted from the source database on the left, and the model of the evolved graph database on the right. The evolved version reflects the conducted changes. In particular, a new node label **Department** was introduced, and both the department id **dId** and its name **dname** were moved to this new label. Additionally, the label **Course** was renamed **Lecture**.

### 4.3 Schema Comparison

The next step is to analyze the differences between the two model versions (Figure 3 Step ③). We use SiLift [28] for model comparison since it does not only compare models but also computes high-level change sets. The comparison process consists of three successive steps.

First, a matcher identifies pairs of corresponding elements that are considered to represent the *same* entity. Matching can be challenging. For example, in Figure 5, the department name **dName** was moved to the **Department** node, allowing for different interpretations. If it is seen as a **move** operation, matching both elements would be correct. Conversely, if it is treated as an **add** operation, matching would be incorrect. Therefore, as with schema extraction, the user is prompted to review the results of the matching and, if necessary, modify them by adding correspondences for unrecognized elements (Figure 3 Step ⑤). The outcome is a list of correspondences between the two models.

Second, a difference derivator identifies single-type changes based on established correspondences. These changes are atomic modifications that cannot be decomposed into smaller editing steps, such as **create** or **delete**. In practice, user editing operations often consist of multiple single-type changes, because even seemingly simple edits can result in numerous underlying modifications.

Therefore, in the third step, we use SiLift’s Semantic Lifting Engine to group changes into *semantic change sets*. Each set represents the effect of a user-level editing operation. In order to adapt SiLift to our modeling language, we need a set of edit rules that define the available editing operations and specify how the structure or content of a model can be modified or updated. These rules describe the permitted or intended changes when evolving a model. The rule set, which is specific to the metamodel used (Section 4.1) and was created once for our setup; it does not need to be redefined for each use.

Figure 7 shows two exemplary edit rules specified with Henshin (introduced in 5), a model transformation engine for EMF models. The simple edit rule **create Node Label** (top) identifies the creation of a new **NodeLabel** element. The lower rule illustrates how the **delete** and **add** operations are combined to represent a **moveProperty** operation. Each object is annotated as **preserve**, **delete**, or **create**. When a **Property** is moved, the reference to the original label is deleted, and a new reference to the target label is created. SiLift uses these rules to search the list of correspondences for patterns that match the criteria and conditions defined by the respective edit rule. If a pattern matches, the corresponding single-type changes are combined into one edit operation.

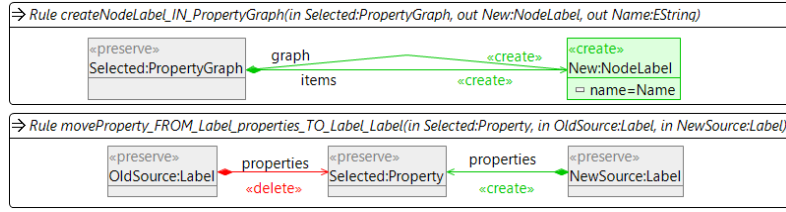


Fig. 7. Edit rules for graph database schemata

#### 4.4 Evolution Language

SiLift compares models to produce a symmetric difference model containing all correspondences, differences, and grouped single-type changes, called *semantic change sets*. These changes represent the edits between the both models.

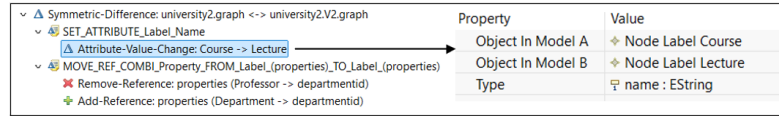


Fig. 8. Symmetric difference model

Figure 8 shows part of the symmetric difference model for our illustrative example (Figure 2). The set *SET\_ATTRIBUTE\_Label\_name* represents the renaming of **Course** to **Lecture**. The property “Object in Model A” refers to the modified element in the original version, and “Object in Model B” refers to the corresponding element in the evolved version. The *Type* property indicates which aspect of the element was changed. In our case, the value of **name**. The second set shown, describes the relocation of the property **did** from **Professor** to **Department**. Accordingly, the reference of this property is moved as well.

In the next step, these sets are mapped to their corresponding GEOs (see Figure 3, Step ⑥). This approach is beneficial over using commands, as GQL lacks a dedicated evolution language. Without GEO, expertise in GQL is required to understand complex operations. GEO is designed to describe evolution operations in a clear and intuitive manner. A full list of all supported GEOs can be found in<sup>4</sup>. The simple edit rule shown at the top of Figure 8 is translated into a **rename label** operation. The second edit rule represents a **move** operation, which is considered a complex evolution operation. Such operations can be decomposed into a combination of single-type operations, namely **add**, **rename** and **delete**. The mapping results in the two GEOs shown in Listing 1.1. The first GEO corresponds to the **rename** operation, while the second represents the **move** operation. Finally, these GEOs are visualized as output for the application user.

<sup>4</sup> GEO file: <https://zenodo.org/records/15958412>

## 5 Implementation

We implemented our approach for Neo4j<sup>5</sup> because of its widespread use in industry and research. Moreover, GQL is based on the Neo4j’s query language, Cypher, making it an ideal candidate. Additionally, we used the APOC<sup>6</sup> (Awesome Procedures on Cypher) library as the basis for our schema extraction algorithm. APOC provides access to useful procedures and functions that extend the use of the Cypher query language into areas such as data integration, graph algorithms, and data conversion. Our prototype is implemented as a feature for the Eclipse platform. We did so, because the Eclipse Modeling Tools and the Eclipse Modeling Framework [37] provide a powerful platform for creating, editing, and managing structured data models. EMF provides a comprehensive framework for code generation and model handling, making it highly efficient for model-driven development. For model comparison, we used Henshin<sup>7</sup> and the SiLift [28] framework. Henshin supports in-place model transformations, enabling changes directly within the existing model structure. We chose the SiLift framework for model comparison, since SiLift can group low-level changes into *semantic change sets*, each of which represents the impact of a user-level editing operation. The source code of our implementation is available in our GitHub repository<sup>8</sup>.

Our current implementation supports the key functionalities of model-driven graph transformation. However, some features and system-specific refinements have yet to be implemented. As described in Section 4.1, the metamodel design aligns with the GQL standard. Using this standard ensures that our model-driven approach is domain-independent and allows for the integration of various graph database systems. However, several aspects of the ISO standard are not directly applicable when working with a specific graph database. For example, Neo4j enforces exactly one label per relationship, whereas GQL allows zero to many. Such system-specific constraints must be considered in future refinements. In addition, some complex schema transformation operations, such as copying or splitting of nodes, are not currently accurately detected.

## 6 Initial Evaluation

We evaluated the effectiveness of our approach in handling schema extraction and transformation using three Neo4j example graph databases<sup>9</sup>.

*Set up.* To ensure diversity, we selected three sample databases from the Neo4j example collection that differ in both schema size and number of values. Table 1 provides an overview. We applied realistic operations to each dataset to simulate typical schema changes and obtain the evolved version. Specifically, six evolution

<sup>5</sup> <https://neo4j.com/>

<sup>6</sup> <https://neo4j.com/docs/apoc/5/introduction/>

<sup>7</sup> <https://projects.eclipse.org/projects/modeling.emft.henshin>

<sup>8</sup> <https://github.com/tekw24/evolveDB>




<sup>9</sup> <https://github.com/neo4j-graph-examples>

operations were executed on the *Twitch* dataset, while the *Pole* dataset was evolved with five operations. Eight operations were carried out on the *Bloom* dataset. Taking advantage of real-world user behavior, the **add** operation was executed with the highest frequency across all datasets, followed by **rename** and **delete** (for single-type), **copy**, **move** and **split** (for multi-type operations).

	Data Statistics			Schema Statistics			
	Total Nodes	Total Edges	Total Properties	Node Labels	Edge Labels	Property keys	$\Sigma$
<b>Twitch</b>	4.680.870	10.076.938	4.710.878	10	15	13	38
<b>Pole</b>	61.521	105.840	200.364	22	35	34	91
<b>Bloom</b>	30.960	29.731	101.353	36	38	60	134

**Table 1.** Data and schema statistics for each dataset used in the evaluation

*RQ1: How accurate is the schema extraction from a given graph database?* Accurate results require that the user correctly assigns properties to the appropriate labels in cases of multi-labeling. This is a prerequisite for the subsequent automatic schema extraction. After the assignment, the duration of automatic schema extraction increases with the size of the database. The largest dataset was extracted in under 20 seconds. *With the human-in-the-loop support, the system can achieve 100% accuracy in schema extraction.*

*RQ2: How accurate is the schema transformation?* A detailed quantification of the evaluation results is given in Table 2. In addition to the total number of executed operations ( $\Sigma$ ), the number of evolution operations that were automatically identified is shown by . The  symbol indicates that no evolution operator of this type was applied to the source version.  represents operations requiring additional manual assignments from the user. The **split** and **copy** operations are not reliably detected by the system. This outcome was expected, as the system currently lacks full support for these complex evolution operators. In such cases, user input is necessary to resolve ambiguities and ensure the transformation is interpreted correctly. *With the human-in-the-loop support, the system can achieve 100% accuracy in schema transformation.*

*RQ3: What proportion of manual user input is required for schema extraction and transformation?* More multi-labeled nodes require more user input for property assignment. Based on real-world data, our datasets show that multi-labeling rarely occurs. Specifically, the source graphs contain only two cases of multi-labeling, while the target graphs contain an additional instance, resulting in three cases of multi-labeling in the target graph and five cases in total. During schema comparison, the user is prompted to review and modify the matching results by adding correspondences for unrecognized elements, if necessary. Consequently, the interaction rate indicates the proportion of unmatched or incor-

	Executed Evolution Operations															Results				
	Single-type									Multi-type						Extr.		Mapping		
	add			rename			delete			copy			move			split			S	T
	$\Sigma$			$\Sigma$			$\Sigma$			$\Sigma$			$\Sigma$			$\Sigma$				
<b>Twitch</b>	3	3	0	1	1	0							1	1	0				1	1
<b>Pole</b>	2	2	0	1	1	0	2	2	0	1	0	1							38	37
<b>Bloom</b>	5	5	0	2	2	0							1	1	0	1	0	1	134	131

**Table 2.** Number and kind of executed evolution operations together with the results of the schema extraction and schema mapping

rectly matched elements. Table 1 summarizes the number of element pairs that could potentially form correspondences between the source and target models in the rightmost column. Table 2 shows information about schema extraction and transformation results in the five rightmost columns. The *Twitch* example includes 38 correspondences, 37 were automatically identified correctly, and one element pair was missing, resulting in an interaction rate of 2.63%. The source model of the *Pole* example contains 91 elements. Due to deleted elements in the target model, only 88 potential correspondences remained. Of these, 82 were automatically identified correctly, five were not detected, and one pair was incorrectly matched. This results in an interaction rate of 6.81%. The *Bloom* example contains 134 expected correspondences. Of these, 131 were identified automatically. Three element pairs were not recognized, resulting in an interaction rate of 2.23%. The evaluation also revealed that applying an increasing number of changes to a single element in the *Pole* example leads to a higher likelihood of unrecognized or incorrect correspondences. Of the three examples, five cases of multi-labeling required user interaction during the schema extraction. Ten out of 260 correspondences required user intervention during schema mapping, corresponding to an overall interaction rate of 3.84%. *These findings show that although most of the work can be automated, carefully reviewing the extraction and mapping results is crucial, as undetected or incorrect correspondences can compromise the accuracy of the schema transformation.*

*Threats to Validity.* One potential threat is that our evaluation was limited to three example databases, which constrains the generalizability of the results. Additionally, the evolution operations applied to the graph databases were defined by the authors and may not accurately reflect the full range of changes encountered in other systems or contexts. Consequently, the findings may not be directly transferable to different settings. Moreover, the authors performed the required user interaction for schema extraction and mapping, as well as the validation of the resulting correspondences. To address this issue, we deliberately applied a diverse set of evolution operators and had two authors validate the results independently. Future studies should incorporate graph databases with real-world data to ensure broader coverage of schema transformation scenarios and to validate the generalizability of the results.

## 7 Related Work

This section discusses related work on *graph schemas*, *schema extraction*, *schema evolution*, *schema transformation*, and *model-driven engineering*.

**Graph Schema.** The most renowned paper on graph schemas is [3], which emphasizes the necessity of formalized schema descriptions and proposes *PG-Schema*. The importance of schema constraints for query optimization is discussed in [35]. [23] uses a formalization of the schema through description logic. In addition, ensuring data quality through constraints is done in [33], with [31] focusing on integrity constraints. Regular path constraints to measure inconsistency are presented in [22]. Work has been done on keys [4] and triggers for property graphs [14]. An extendable schema language is presented in [6]. *All these works highlight the necessity of deriving schemas with constraints for semi-structured data in schema-less graph databases.* To this end, we present a metamodel derived from the GQL standard, and use it for model-based schema extraction.

**Schema Extraction.** Several papers focus on graph schema extraction with a schema-second method, e.g., [20, 9, 16]. The relevance of schema extraction for NoSQL databases, here document stores, is emphasized in [30]. *Extracting the schemas of given source and target graph data is a prerequisite for schema transformation.* In contrast to other approaches, we use a conceptual model – our metamodel – to transform the graph schema into a schema model of the graph. Consequently, (1) raising the abstraction, (2) ensuring technology-independence by integrating GQL and (3) integrating a human-in-the-loop.

**Schema Transformation.** Schema mapping is often discussed when migrating data between a NoSQL and a relational database, as described in [13, 2, 5]. [34] shows how to transform data warehouses which are based on multidimensional conceptual models into graph models. To overcome the absence of a schema description standard, FLASc is presented in [36], which defines a schema through formal algebra. We also use a model independent of graph database technology to represent a graph schema, but our approach is closer to implementation. [32] focuses on schema transformation between RDF and property graphs. [1] illustrates a transformation approach by using a meta level. *However, a domain-independent language for implementing schema transformations is still missing.* Our metamodel, which performs model-driven schema transformation, is technology-independent and guarantees standard conformity. GEO [25], the language of graph evolution operations, is based on this metamodel and is also technology-independent.

**Schema Evolution.** Work on RDF-Graph is shown in [15, 7], nevertheless, we focus on property graphs for which GQL was designed. The importance of adequately handling schema evolution is evident from the variety of database systems for which evolution has been considered. [21, 18, 40] focus on relational, and [38] on document stores. MigCast [26] can handle data migration tasks for evolving NoSQL databases. [39] presents an approach for adapting wide-column stores when an underlying conceptual model evolves. Nautilus, a tool for evolving graph databases is presented in [24]. The preservation of constraints during

evolutionary tasks is presented in [12]. [11] proposes using a graph schema to properly model heterogeneous web data and manage evolution through version control. *These approaches to schema evolution are all technology-dependent, so they cannot be directly reused for other graph database technologies.* In contrast, our approach to schema transformation is applicable to all graph databases because it is based on GQL. It also forms the basis for technology-independent schema evolution.

**Model-Driven Engineering.** Since we use a model-driven approach, we also analyze approaches that use model-driven techniques for schema evolution, transformation, and data migration. [41] extracts schemas to JSON files, which are then mapped to Ecore. This approach aims to transform schemas to improve the data quality of an extracted schema. The relevance of a model-driven approach for handling transformation tasks, especially in at the meta level, is demonstrated in [29], whereas our approach specializes on graph databases by using an abstract meta level containing a domain-independent meta model to perform schema transformation. *Both approaches are model-driven but not GQL compatible.* This is in contrast to our approach.

## 8 Conclusion

Changes to a graph database can affect the implicitly given schema, which may lead to data quality issues. Explicitly documenting the evolution between two schema versions can help developers understand, document, and control the effects of such changes, as well as migrate the graph data to conform to the updated schema. In this paper, we present a model-driven approach and tool to reconstruct the evolution between two implicitly given schema versions of a graph database. With this tool, developers do not need to manually construct schema transformations. To ensure independence from specific graph database technologies, our metamodel is based on the recent GQL standard.

The initial evaluation of our model-driven approach revealed that schema extraction and transformation are both accurate and require only minimal user intervention. However, the results also underscore the importance of carefully reviewing the matching results, as undetected or incorrect correspondences can compromise the accuracy. Consequently, we integrated a human-in-the-loop approach, allowing users to validate the results and make manual adjustments.

In the future, we plan to improve the proposed solution by addressing its current limitations. Currently, during schema extraction, users need to decide which properties belong to which label in cases of multi-labeling. To reduce manual input further, we plan to integrate an algorithm that provides suggestions to support this decision. Based on our initial evaluation, we are confident that our approach has great potential for practical application. We plan to demonstrate this by applying it to various real-world graph databases in the future.

**Acknowledgments.** The work of Dominique Hausler has been funded by Deutsche Forschungsgemeinschaft (German Research Foundation) – 552691270.



## References

1. Abdelhédi, F., Brahim, A.A., Atigui, F., Zurfluh, G.: UMLtoNoSQL: Automatic Transformation of Conceptual Schema to NoSQL Databases. In: AICCSA. pp. 272–279. IEEE Computer Society (2017)
2. Aftab, Z., Iqbal, W., Almustafa, K.M., Bukhari, F., Abdullah, M.: Automatic NoSQL to Relational Database Transformation with Dynamic Schema Mapping. *Sci. Program.* **2020**, 8813350:1–8813350:13 (2020)
3. Angles, R., Bonifati, A., Dumbrava, S., Fletcher, G., Green, A., Hidders, J., Li, B., Libkin, L., Marsault, V., Martens, W., Murlak, F., Plantikow, S., Savkovic, O., Schmidt, M., Sequeda, J., Staworko, S., Tomaszuk, D., Voigt, H., Vrgoc, D., Wu, M., Zivkovic, D.: PG-Schema: Schemas for Property Graphs. *Proc. ACM Manag. Data* **1**(2), 198:1–198:25 (2023)
4. Angles, R., Bonifati, A., Dumbrava, S., Fletcher, G., Hare, K.W., Hidders, J., Lee, V.E., Li, B., Libkin, L., Martens, W., Murlak, F., Perryman, J., Savkovic, O., Schmidt, M., Sequeda, J.F., Staworko, S., Tomaszuk, D.: PG-Keys: Keys for Property Graphs. In: *SIGMOD Conference*. pp. 2423–2436. ACM (2021)
5. Bansal, N., Sachdeva, S., Awasthi, L.K.: Query-based denormalization using hypergraph (QBDNH): a schema transformation model for migrating relational to NoSQL databases. *Knowl. Inf. Syst.* **66**(1), 681–722 (2024)
6. Beeren, N., Fletcher, G.: A Formal Design Framework for Practical Property Graph Schema Languages. In: *EDBT*. pp. 478–484. OpenProceedings.org (2023)
7. Bobed, C., Maillot, P., Cellier, P., Ferré, S.: Data-driven assessment of structural evolution of RDF graphs. *Semantic Web* **11**(5), 831–853 (2020)
8. Boneva, I., Groz, B., Hidders, J., Murlak, F., Staworko, S.: Static Analysis of Graph Database Transformations. In: *PODS*. pp. 251–261. ACM (2023)
9. Bonifati, A., Dumbrava, S., Martinez, E., Ghasemi, F., Jaffré, M., Luton, P., Pickles, T.: DiscoPG: Property Graph Schema Discovery and Exploration. *Proc. VLDB Endow.* **15**(12), 3654–3657 (2022)
10. Brambilla, M., Cabot, J., Wimmer, M.: *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers (2017)
11. Braunschweig, K., Thiele, M., Lehner, W.: A Flexible Graph-Based Data Model Supporting Incremental Schema Design and Evolution. In: *ICWE Workshops. Lecture Notes in Computer Science*, vol. 7059, pp. 302–306. Springer (2011)
12. Calvanese, D., Ortiz, M., Simkus, M.: Evolving Graph Databases under Description Logic Constraints. In: *Description Logics. CEUR Workshop Proceedings*, vol. 1014, pp. 120–131. CEUR-WS.org (2013)
13. Ceresnák, R., Dudás, A., Matiaszko, K., Kvet, M.: Mapping rules for schema transformation : SQL to NoSQL and back. In: *IDT*. pp. 52–58. IEEE (2021)
14. Ceri, S., Bernasconi, A., Gagliardi, A., Martinenghi, D., Bellomarini, L., Magnanimiti, D.: PG-Triggers: Triggers for Property Graphs. In: *SIGMOD Conference Companion*. pp. 373–385. ACM (2024)
15. Chabin, J., Eichler, C., Ferrari, M.H., Hiot, N.: Graph rewriting rules for RDF database evolution: optimizing side-effect processing. *Int. J. Web Inf. Syst.* **17**(6), 622–644 (2021)
16. Comyn-Wattiau, I., Akoka, J.: Model driven reverse engineering of NoSQL property graph databases: The case of Neo4j. In: *IEEE BigData*. pp. 453–458. IEEE Computer Society (2017)
17. Dumbrava, S., Oudemans, M.W.M., Ozkan, B.K.: Fuzzing Graph Database Applications with Graph Transformations. In: *ICGT. Lecture Notes in Computer Science*, vol. 15720, pp. 135–156. Springer (2025)

18. Eckwert, T., Guckert, M., Taentzer, G.: EvolveDB: a tool for model driven schema evolution. In: *MoDELS (Companion)*. pp. 61–65. ACM (2022)
19. Fowler, M.: *Domain-specific languages*. Pearson Education (2010)
20. Frozza, A.A., Jacinto, S.R., dos Santos Mello, R.: An Approach for Schema Extraction of NoSQL Graph Databases. In: *IRI*. pp. 271–278. IEEE (2020)
21. Giachos, F., Pantelidis, N., Batsilas, C., Zarras, A.V., Vassiliadis, P.: Parallel lives diagrams for co-evolving communities and their application to schema evolution. In: *ER (Companion)*. CEUR Workshop Proceedings, vol. 3618. CEUR-WS.org (2023)
22. Grant, J., Parisi, F.: On measuring inconsistency in graph databases with regular path constraints. *Artif. Intell.* **335**, 104197 (2024)
23. Gutiérrez-Basulto, V., Gutowski, A., Ibáñez-García, Y.A., Murlak, F.: Containment of Graph Queries Modulo Schema. *Proc. ACM Manag. Data* **2**(2), 77 (2024)
24. Hausler, D., Klettke, M.: Nautilus: Implementation of an Evolution Approach for Graph Databases. In: *MoDELS (Companion)*. pp. 11–15. ACM (2024)
25. Hausler, D., Klettke, M., Störl, U.: A language for graph database evolution and its implementation in Neo4j. In: *ER (Companion)*. CEUR Workshop Proceedings, vol. 3618. CEUR-WS.org (2023)
26. Hillenbrand, A., Levchenko, M., Störl, U., Scherzinger, S., Klettke, M.: MigCast: Putting a Price Tag on Data Model Evolution in NoSQL Data Stores. In: *SIGMOD Conference*. pp. 1925–1928. ACM (2019)
27. Information technology — Database languages — GQL. Standard, International Organization for Standardization, Geneva, CH (Apr 2024)
28. Kehrer, T., Kelter, U., Ohrndorf, M., Sollbach, T.: Understanding model evolution through semantically lifting model differences with SiLift. In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. pp. 638–641. IEEE (2012)
29. Khelladi, D.E., Kretschmer, R., Egyed, A.: Change Propagation-based and Composition-based Co-evolution of Transformations with Evolving Metamodels. In: *MoDELS*. pp. 404–414. ACM (2018)
30. Klettke, M., Störl, U., Scherzinger, S.: Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In: *BTW. LNI*, vol. P-241, pp. 425–444. GI (2015)
31. Pokorný, J., Valenta, M., Kovacic, J.: Integrity constraints in graph databases. In: *ANT/SEIT. Procedia Computer Science*, vol. 109, pp. 975–981. Elsevier (2017)
32. Rabbani, K., Lissandrini, M., Bonifati, A., Hose, K.: Transforming RDF Graphs to Property Graphs using Standardized Schemas. *Proc. ACM Manag. Data* **2**(6), 242:1–242:25 (2024)
33. Reina, F., Huf, A., Presser, D., Siqueira, F.: Modeling and Enforcing Integrity Constraints on Graph Databases. In: *DEXA (1)*. Lecture Notes in Computer Science, vol. 12391, pp. 269–284. Springer (2020)
34. Sellami, A., Nabli, A., Gargouri, F.: Transformation of Data Warehouse Schema to NoSQL Graph Data Base. In: *ISDA (2)*. Advances in Intelligent Systems and Computing, vol. 941, pp. 410–420. Springer (2018)
35. Sharma, C., Genevès, P., Gesbert, N., Layaïda, N.: Schema-Based Query Optimisation for Graph Databases. *Proc. ACM Manag. Data* **3**(1), 72:1–72:29 (2025)
36. Sharma, C., Sinha, R.: FLASc: a formal algebra for labeled property graph schema. *Autom. Softw. Eng.* **29**(1), 37 (2022)
37. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: *EMF: eclipse modeling framework*. Pearson Education (2008)

38. Störl, U., Klettke, M.: Darwin: A Data Platform for Schema Evolution Management and Data Migration. In: EDBT/ICDT Workshops. CEUR Workshop Proceedings, vol. 3135. CEUR-WS.org (2022)
39. Suárez-Otero, P., Mior, M.J., Suárez-Cabal, M.J., Tuya, J.: CoDEvo: Column family database evolution using model transformations. *Journal of Systems and Software* **203**, 111743 (2023)
40. Vassiliadis, P., Karakasidis, A.: Time-Related Patterns Of Schema Evolution. In: EDBT. pp. 310–323. OpenProceedings.org (2025)
41. Wischenbart, M., Mitsch, S., Kapsammer, E., Kusel, A., Pröll, B., Retschitzegger, W., Schwinger, W., Schönböck, J., Wimmer, M., Lechner, S.: User Profile Integration Made Easy: Model-Driven Extraction and Transformation of Social Network Schemas. In: WWW (Companion Volume). pp. 939–948. ACM (2012)