A language for graph database evolution and its implementation in Neo4j

Dominique Hausler[1,*], Meike Klettke[1] and Uta Störl[2]

[1]*University of Regensburg, Data Engineering Group, Faculty of Informatics and Data Science, Bajuwarenstrasse 4, 93053 Regensburg, Germany*

[2]*University of Hagen, Chair of Databases and Information Systems, Faculty of Mathematics and Computer Science, Universitätsstrasse 1, 58084 Hagen, Germany*

Abstract

In this article, we present an evolution language for graph databases and a method to realize evolution operations on graph databases and their schema. Graph database management systems like Neo4j can be used for different scenarios: they can store graphs where no semantic constraints are checked. It is also possible to use them to store graphs where the structure of nodes and relationships is regular, for example, because the nodes and relationships of a graph have been generated by an application. In the latter case, these structures may change, that means they can undergo evolution. Currently, there is no established evolutionary language for graphs. However, evolution operations (such as add, rename, delete, merge, copy, split and move) have already been developed for other database models. In this article, we will not only use the established evolution operations but additionally extend the operations appropriated. After analyzing evolution operators explicitly in the context of graph databases, we also created a new graph-specific operation: transform. Moreover, we suggest an evolution language to graph data and show how it can be executed on the graph database schema as well as on the instances (i.e. the graphs themselves). In doing so, some operations can be implemented native based on Cypher, others non-native using the APOC library for Neo4j.

Keywords

graph databases evolution, evolution, graph schema description, graph evolution language

1. Introduction and motivation

If systems are used successfully over a long period of time, they need to be updated from time to time. This is also valid for **G**raph **D**ata**b**ases (**GDBs**). GDBs allow *flexible storage of graphs* with different structures, but these graphs always contain implicit structures that can follow a certain structuring. To adapt this implicit structural information, we need a *language for the evolution of GDBs.* There are several reasons why an evolution in graph databases is necessary:

- Different representations of information can be chosen in GDBs, so information can be stored as node, node property, relationship, direction of a relationship, or relationship

ER2023: Companion Proceedings of the 42nd International Conference on Conceptual Modeling: ER Forum, 7th SCME, Project Exhibitions, Posters and Demos, and Doctoral Consortium, November 06-09, 2023, Lisbon, Portugal

*Corresponding author.

dominique.hausler@ur.de (D. Hausler); meike.klettke@ur.de (M. Klettke); uta.stoerl@fernuni-hagen.de (U. Störl)
0009-0004-2381-133X (D. Hausler); 0000-0003-0551-8389 (M. Klettke); 0000-0003-2771-142X (U. Störl)

property. Unlike relational databases, there is *no established design methodology* for GDBs yet. Due to these different representations, it can happen that these are to be changed for already used GDBs. In these cases, an evolution is necessary.

- For a given workload (read and write operations) on GDBs, it is often not clear which design of the database causes a more efficient execution. Because GDB optimization is still a very young field of research and there is not yet a standardized approach to design, *subsequent structural changes* are very likely.
- GDB refactoring may also be necessary when different GDBs with different structures shall be integrated.
- Further development of the database and the application or simple extensions of the graph data are another reason for evolution.

In all these cases, GDBs have to be evolved. Thereby, we have to consider the following aspects:

- With the evolution operations on GDBs the *context must also be considered*, for all operations on nodes it must be defined how the in– and outgoing relationships are proceeded.
- So far, there is *no standardized graph data query language* which we can build on.

We have to be able to make *simple extensions*, such as adding or renaming properties of either entity type. But we also need possibilities for a *complex graph restructuring*. For all of these cases, we will propose evolution operations in this article and illustrate, respectively, what changes to the schema and the graphs are performed by them.

The main contributions of this article are the following: We suggest an evolution language for graph data. In doing so, we extend existing approaches by *transform* to change the entity type (e.g. a node to a relationship). We show, how the evolution language can be executed on the GDB schema via $\mathrm{SMO_C}$ as well as on the instances ($\mathrm{GMO_C}$).

Structure of the article. The next section lists related work. In Section 3, we give an overview on the evolution approach. We continue with the definition of a schema for graph databases (Section 4). The main part is the definition of evolution operations (Section 5) and their execution in Neo4j. We conclude in Section 6 with a summary and some future work tasks.

2. Related work and state of the art

Our work is based on a variety of existing research, which is given here.

Graph query languages. There already exist some suggestions for GDB languages, e.g. GQL, PGQL, and Cypher. For our work, we decided to choose Cypher because of its popularity, [1, 2] its usage in commercial, well known projects [3] and the postulation of extending Cypher to a standardized language for property graph (**PG**) databases [4].

Schema of graph databases. Work has already been done on schema description in GDBs like in [5]. Even though the schema grammar of [5] was taken into account for the schema grammar proclaimed here, the complexity of it had to be extended to illustrate APOC functions, nodes with multiple relationships with different directions, and to allow the schema modification operations for Cypher to be illustrated. We use it to describe the database state before and after executing an evolution operation.

Evolution for other data models (relational data, XML, JSON). Database evolution is needed in all database management systems. In relational databases, a evolution language that can evolve existing relational databases is part of the standard. The `alter table` statements define the evolution and modifies the tables. A bit more complicated is the evolution in XML and JSON databases because we have to consider heterogeneous datasets. XML evolution operations are defined in [6, 7, 8] and JSON evolution in [9, 10].

Updates and evolution of graph databases. Green et al. [11] suggest Cypher extensions to realize updates. It includes `add`, `delete` and `merge` operations proposing possible improvements for Cypher. These extensions are not an evolution language but can be used for graph data migration into a new version. In [12, 13, 14], a schema language is introduced and the validation of GDBs against the schema is described. In contrast to this article, we extended the evolution operations by `transform` to illustrate the complexity of GDBs. In consequence of writing the domain-specific SMO_C proclaimed in this paper, a precise implementation is possible. Moreover, this article presents an analysis of all evolution operations, categorizing them by the necessity to utilize a library.

Schema extraction. There are several works that are extracting an explicit schema from GDBs, some of which are [15, 16, 17, 14]. For the topic described in this article, we assume an existing schema, which may have been either pre-defined (schema-first, forward engineering) or derived from existing GDBs (reverse engineering). Thus, a schema-extraction can be a previous step for schema evolution. As far as we know, there currently is no graph database system that implements schema evolution with complex refactoring operations.

3. Overview

Figure 1 shows the basic components and the process of schema evolution in GDBs. ① is the most abstract level. It contains information about the **G**raph **E**volution **O**perations (GEO), describing them in general. On the next level ②, we see the schema. The light blue boxes represent the Schema Grammar SG_n, $\text{SG}_\text{n+1}$ before and after the execution of the evolution operation. SMO_C (**S**chema **M**odification **O**peration) defines the operation for the schema level – here on base of Cypher – as illustrated by ③. In this article, we use the **C** index for Cypher (Neo4j's query language) to indicate that a language is domain-specific. Number ④ illustrates the data level. Here the concrete entity types of the GDB are shown (before and after the evolution). The light green arrow ⑤ visualizes the implementation of the **G**raph Data **M**igration **O**peration in Neo4j, called GMO_C. All of the parts are described in detail in the following sections.

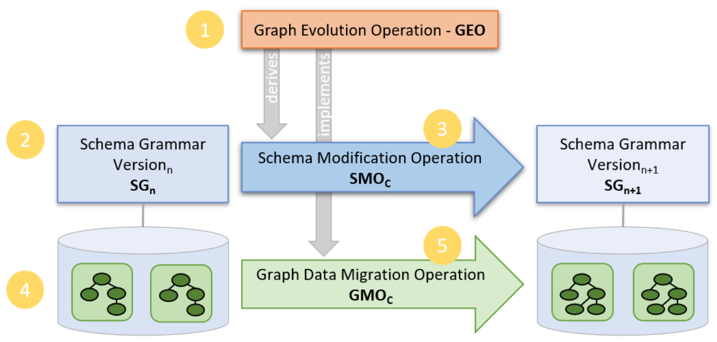
1
Graph Evolution Operation - GEO
derives
implements
2
Schema Grammar Version$_n$ SG_n
3
Schema Modification Operation SMO_C
Schema Grammar Version$_{n+1}$ SG_{n+1}
4
5
Graph Data Migration Operation GMO_C

Figure 1: Evolution of Graph Databases

4. Schema in graph databases

The structure of GDBs can be defined in an explicit schema. We developed an EBNF like[1] **S**chema **G**rammar (SG) inspired by [5] and modified it to fit the domain-specific SMO_C we invented. The SG in Figure 2 shows the definition of nodes, relationships and features. Nodes in GDBs can contain labels or node properties, whereas relationships can contain types or relationship properties. For the explanation, we name these four terms *features*. A node is represented in round brackets, a `rel`[2] is defined in squared brackets. Furthermore, the SG defines `connectedNode` and `multiConnectedNode` representing nodes with associated relationships. Properties are key-value pairs. Furthermore, `nodeLabel` and `relationshipType` are defined as strings; a `nodeLabel` can have several strings. In that case, nodes are called multi-labeled [3]. A `property` is defined as a map. In Figure 1, it is shown that an evolution operation evolves the schema and the graph data. The schema grammar versions SG_n and $\text{SG}_\text{n+1}$ show the schema before and after the evolution operation. In both cases, the schema is defined with the grammar shown in Figure 2.

```
entityType ::= node | rel
node ::= '(' ((nodeLabel (property)* (','
↪  nodeLabel (property)*)*) | ε) ')'
nodeEntity ::= node | connectedNode |
↪  multiConnectedNode
connectedNode ::= ((node '-' rel '→' node) |
↪  (node '←' rel '-' node))
multiConntectedNode = connectedNode ((('-'
↪  rel '→') | ('←' rel '-')) node)*
rel ::= '[' relationshipType (property)* (','
↪  relationshipType (property)*)* ']'
feature ::= property | nodeLabel |
↪  realtionType
property ::= '{' propertyName ':' value (','
↪  propertyName ':' value)* '}'
propertyName ::= String
nodeLabel ::= (String)+
relationshipType ::= String
```

Figure 2: EBNF like Schema Grammar for entity types and their features

5. Evolution operators

In this section, the evolution operations *add*, *rename*, *delete* which are so-called single-type operations and the multi-type operations *merge*, *copy*, *split* and *move* will be examined. Each operation changes the schema and graph data as described in Figure 1. Also, we categorize the operations according to whether they are directly available in Cypher as *native*, while functions implemented in the APOC library are called *non-native*.

[1]Variables start with lowercase, fixed terms with uppercase letters. SMO_C commands are capitalized and Cypher specifics are violet in quotation marks. In GEO quotation marks are used as descriptions.
[2]The language uses `rel` as the short form for a relationship.
[3]The presence of two or more labels in a node.

5.1. Add

The first evolution operation add is needed for extensions of the GDB, either by new features or new entity types. In Figure 3, the GEO appends a new feature, specifically a property in the GMO$_C$. The SMO$_C$ defines how, within the FOREACH statement, features can be added. By choosing a function and a pattern that contains relationships, either properties, types or labels can be attached. In the SMO$_C$ the add operation is implemented as *native* operation. The corresponding GMO$_C$ in Figure 3 shows two variants to add the property `title` to a `Manager` node. Both generate the same result. Option A matches all `Manager` node paths, selects among them the nodes fulfilling a certain condition and sets a new property. It uses the FOREACH statement and realizes a *native* operation. Due to the pattern complexity of Option B the evolution operation can access all nodes without a FOREACH command *native*.

GEO

```
addFeatures ::= 'add' features 'to'
↪  entityTypes
```

SMO$_C$ for Option A

```
MATCH variable (*for path*) '='
↪  nodeEntity (filter)?
FOREACH '(' variable IN (NODES |
↪  RELATIONSHIPS) '(' variable (*for
↪  path*) ')' '|' (setProperty |
↪  setLabel | setType) ')'
(RETURN ((nodeVar)+ | (relVar)+)?
↪  (*depending on pattern and feature*)
```

GMO$_C$

```
//Option A
MATCH pattern = (n:Manager)
WHERE (n)<-[:WORKS_AT]-(:PlaceOfWork
↪  {city:'Regensburg'}) OR
↪  (n)<-[:WORKS_AT]-(:PlaceOfWork
↪  {city:'Passau'})
FOREACH (n IN nodes(pattern) | SET
↪  n.title = 'Prof. Dr.' )
RETURN n
//Option B
MATCH (n:Manager)-[:WORKS_AT]->
↪  (n2:PlaceOfWork)
WHERE n2.city='Regensburg' OR
↪  n2.city='Passau'
SET n.title = 'Prof. Dr.'
RETURN n, n2
```

Figure 3: GEO, SMO$_C$ and GMO$_C$ for: *Add* features

For adding entity types, the GEO would be defines as: `addEntityTypes ::= 'add' entityTypes 'to' database`. In contrast, the SMO$_C$ (Figure 4) differentiates between the entity types. Whenever there is uncertainty, MERGE is to be preferred over CREATE. This prevents the generation of duplicates [18]. At the precise level of GMO$_C$ there are two possible outcomes when utilizing a FOREACH command to create multiple nodes. If nodes are added with only a label that does not already exist in the GDB, the nodes will be displayed in gray with the automatically generated id naming them. In contrast, if the label was already assigned, the nodes will be empty, but colored according to the label assigned.

SMO$_C$

```
selectPattern WITH collect(NodeVar |
↪  RelVar) aliasCollection
FOREACH '(' variable IN collection '|'
↪  (addNode | addRel) ')'
```

Figure 4: Operation: *Add* entity types

Table 1 illustrates the available commands to add features. It differentiates between *native* and *non-native* options while *non-native* commands have the advantage of making *dynamic specifications*. Moreover, it shows that add is an evolution operation with a wide variety of available commands while the extent of *non-native* commands outnumber the *native* options. The categories *Sgl* and *Mul* can resemble the pattern complexity or the number of entity types or features that can be added.

Table 1
Add features in Neo4j

Operation	**Command**	**Feature**								**Status**	
		Node				**Relationship**				Native	Non-Native
		Label		**Property**		**Type**		**Property**			
		Sgl	**Mul**	**Sgl**	**Mul**	**Sgl**	**Mul**	**Sgl**	**Mul**		
Add	SET	✓	✓	✓	✓	–	–	✓	✓	✓	–
	apoc.create.setLabels	✓	✓	–	–	–	–	–	–	–	✓
	apoc.create.setType	–	–	–	–	✓	–	–	–	–	✓
	apoc.create.setProperty	–	–	✓	–	–	–	–	–	–	✓
	apoc.create.setProperties	–	–	✓	✓	–	–	–	–	–	✓
	apoc.create.setRelProperty	–	–	–	–	–	–	✓	–	–	✓
	apoc.create.setRelProperties	–	–	–	–	–	–	✓	✓	–	✓

5.2. Rename

Updating a database also incorporates renaming features. A rename operation is needed for this task. When looking at the GEO, rename can be seen as a change of an old to a new name or string. Rename is defined as: 'rename' features 'to new' name. There are several options to manipulate features, as the SMO_C in Figure 5 demonstrates. Before the pattern needs to be defined and the entities are collected.

SMO_C

```
selectPattern WITH collect(NodeVar |
↪  RelVar) aliasListOfEntities
rename RETURN COMMITTEDOPERATIONS
```

Figure 5: Operation: *Rename* features

A *native* realization can only be done by combining a delete and an add operation. Instead of this workaround, the APOC library offers all *non-native* options to rename features. All of them are illustrated in Table 2. For a concrete implementation (GMO_C) use the desired rename function at rename of the SMO_C. Table 2 points out that for each feature, a rename command is available.

5.3. Delete

Another important operation is the option of deleting no longer needed entity types or features in a database. Consequently, the evolution operation delete will be shown in this section. The GEO in Figure 6 makes a clear determination between *restricted* and *cascade*[5] delete

[5]As in other database technologies, the keyword cascade specifies that with a delete operation of a node also all of its associated relationships are deleted.

Table 2
Rename features in Neo4j

Operation	Command	Feature								Status	
		Node				Relationship				Native	Non-Native
		Label		Property		Type		Property			
		Sgl	Mul	Sgl	Mul	Sgl	Mul	Sgl	Mul		
Rename	apoc.refactor.rename.label	✓	✓	–	–	–	–	–	–	–	✓
	apoc.refactor.rename.nodeProperty	–	–	✓	✓	–	–	–	–	–	✓
	apoc.refactor.rename.type	–	–	–	–	✓	✓	–	–	–	✓
	apoc.refactor.rename.typeProperty	–	–	–	–	–	–	✓	✓	–	✓

and `remove`. The first two describe how to remove entity types whereas `remove` deals with features, also described in the SMO$_C$. Since all relationships including the selected nodes are deleted by DETACH DELETE it has to be utilized carefully [19]. A DELETE is always preferred to prevent unintended deleting. How to flexibly change the code and use any of the three options is shown by the SMO$_C$. Moreover, the SMO$_C$ gives the alternative of a `non-native` cascade delete as well as *non-native* remove options.

The GMO$_C$ in Figure 6 deletes the property `role` from all relationships of the type WORKS_FOR. In addition to the relationship type, the selected pattern defines start nodes to be labeled `Person` and end nodes to be `Manager`. When looking at the concrete level of implementation (GMO$_C$) a feature is being removed from all relationships of type WORKS_FOR. To access the property `role`, REMOVE is executed. To accomplish the deletion of the relationship itself, DELETE would be utilized. Even tough REMOVE can be executed directly in Cypher there are also *non-native* options [20, 21, 22]. The difference between the Cypher command and an APOC function is that, in the Cypher command no dynamic specifications can be made [23]. The *non-native* option of DETACH DELETE called `apoc.nodes.delete` requires the `ids` of the entities as input.

GEO

```
delete ::= restrictedDelete |
↪  cascadeDelete | remove
restrictedDelete ::= 'delete'
↪  entityTypes 'from' database
cascadeDelete ::= 'delete'
↪  connectedNodes 'from' database
remove ::= 'remove' features 'from'
↪  entityTypes
```

SMO$_C$

```
selectPattern (restrictedDelete |
↪  nativeCascadeDelete |
↪  (nonNativeCascadeDelete RETURN
↪  VALUE) | nativeRemove | (WITH
↪  collect(NodeVar | RelVar)
↪  aliasCollection nonNativeRemove
↪  RETURN NODE))
```

GMO$_C$

```
MATCH (n:Person)-[r:WORKS_FOR]->
↪  (n2:Manager)
REMOVE r.role
```

Figure 6: GEO, SMO$_C$ and GMO$_C$ for: *Remove* entity types and features

5.4. Transform

Since GDBs have a different structure than other types of databases, further graph-specific evolution operations are needed. This comprises the change of entity types and features to entity types. These operations are named `transform`.

GEO

```
nodeToRel ::= 'transform' node 'with
↪  its features into a' relationship
relToNode ::= 'transform a' relationship
↪  'with its features into a' node
```

SMO_C

```
selectPattern
WITH (collectNodeVar aliasNodesToMove|
↪  collectRelVar aliasRelsToMove)
(nodeToRel | relToNode)
RETURN OUTPUT
```

Figure 7: GEO and SMO_C for: *Transform* entity types

How entity types can be transformed into one another is illustrated by the GEO (Figure 7). The SMO_C in Figure 7 points out that this is possible through the APOC functions `collapseNode` and `extractNode`. With this `transform` operation, all initial features of the originals can be kept. It is also possible to specify that a property is transformed into a node entity, through GEO as: `'select a' property 'of n' nodes 'and transform it into a' node 'connected by a' relationship 'to the nodes prior containing this property'`. Describing a specific case to `transform` a feature to an entity type. On the precise level of GMO_C the *non-native* function `apoc.refactor.categorize` realizes the transformation of a node property to a node. Analog the SMO_C in Figure 8 illustrates its components, including the possibility to `rename` the key. Upon the transformation, the new entity will be utilized to categorize the nodes, former containing this property. To achieve this, a uniqueness constraint must first be defined for the new node entity.

SMO_C

```
CALL APOC.REFACTOR.CATEGORIZE '('
↪  propertyName (*from selected
↪  pattern*) ',' type ',' direction ','
↪  label (*of newly created node*) ','
↪  propertyName (*new property name can
↪  be set for new node*) ','
↪  (listOfStrings (*to be copied*) |
↪  '[' ε ']' ) ']' ',' batchSize ')' ';'
```

Figure 8: Operation: *Transform* a node property

5.5. Merge

Multiple entities with duplicate information can occur within the database. The `merge` operation is used to combine them. The GEO in Figure 9 for `merge` includes: inner, outer, right and left join, analogous to SQL. The `join` operation is defined as combining the data; it can only be applied to entity types. The SMO_C demonstrates that there are just *non-native* commands available. Inside of the *non-native* `apoc.refactor.mergeNodes` command, either specific keys or the property maps can be set to `combine`, `discard` or `overwrite/ override`. There is also the alternative to `merge` relationships according to the defined pattern.

GEO

```
fullOuterInclusive ::= copy properties
↪  'of' entityTypes 'to' entityTypes
↪  'and' (cascadeDelete |
↪  restrictedDelete) 'originals'
leftJoin ::= 'keep all features of
↪  left' entityTypes 'add duplicates
↪  from right' entityTypes
rightJoin ::= 'keep all features of
↪  right' entityTypes 'add duplicates
↪  from right' entityTypes
```

SMO$_C$

```
selectPattern
WITH collect(NodeVar | RelVar)
↪  aliasEntitiesToJoin
nonNativeMerge
RETURN NODE
```

Figure 9: GEO and SMO$_C$ for: *Merge* of entity types

To perform a full outer inclusive join in Cypher (on the level of GMO$_C$) the parameter `properties` is set to `combine`. In case of multiple values for identical keys, an array emerges. When using `discard`, the first property value would be kept, whereas `overwrite` would use the last one. Both options are based on the automatically generated `id`, which shows the order in which the data was generated. This determines the oldest and the most recent entity. In order to perform a left or right join, there is the availability of defining the type of join for each property key[24].

5.6. Copy

There are multiple ways of copying entities in Cypher. In the GEO in Figure 10, we describe three options to do so. `copyEntitiyTypes` points out that either a node or a node with associated relationships can be copied. When applying a copy operation merely to relationships, start and end nodes would need to be assigned for them to avoid loose ends. Through the SMO$_C$ (Figure 10) it becomes obvious that features are copied by adding them to another entity type. `cloneNodes` offers the possibility to copy the entire entity, either with or without all relationships (`withRelationships`). Moreover, properties that are not supposed to be copied to the

GEO

```
copyEntityTypes ::= 'copy'
↪  (connectedNodes | nodes) ('keep'
↪  relationships | delete
↪  relationships)
copyFeatures ::= 'copy' features 'to'
↪  entityTypes
copySubgraph ::= 'copy' connectedNodes
↪  'linked as' subgraph
```

SMO$_C$

```
copyNodes* ::=  APOC.REFACTOR.
↪  CLONENODES '(' listOfNodes (','
↪  withRels (',' listOfStrings (*to
↪  skip in clone*) )? )? ')' YIELD
↪  INPUT ',' OUTPUT ',' ERROR
withRels = Boolean
```

Figure 10: GEO and SMO$_C$ for: *Copy* nodes with or without relationships

new node entity can be omitted (`skipPropertiesInClones`). GMO$_C$ in Figure 11 shows the use of a workaround to set properties from the `PlaceOfWork` nodes to `Copy`. Consequently, the GEO to chose here would be `copyFeatures`. In the SMO$_C$ the features to be copied can be specified manually within `addPropToNode` or `addPropToRel`, but this will result in an `add` operation. Therefore, selecting the entities to be copied inside the `CALL` function makes it a `copy`. In GMO$_C$ all keys and values of a `map` are picked. Through `addPropsToRel` the picked features can be attached to relationships.

SMO$_C$

```
selectPattern WITH collect(NodeVar |
↪  RelVar) aliasAttachProps
(CALL '{'
    selectPattern
    WITH getPropertiesFrom(NodeVar |
    ↪  RelVar) aliasMap
    WITH getKeysFromMap  aliasKeys ','
    ↪  getValuesFromMap aliasValues
    RETURN keys, values '}' )?
(addPropToNode | addPropToRel)
RETURN NODE
```

GMO$_C$

```
MATCH (p:Copy)
WITH collect(p) AS nodes
CALL{ MATCH (n:PlaceOfWork{city:
↪  'Regensburg'})
    WITH properties(n) AS map
    WITH [k IN KEYS(map) | map[k]] AS
    ↪  values, keys(map) AS keys
    RETURN values, keys }
CALL apoc.create.setProperties (nodes,
↪  keys, values)
YIELD node RETURN node
```

Figure 11: SMO$_C$ and GMO$_C$ for: *Copy* properties

5.7. Split

Another evolution operation needed for database refactoring is `split`. When splitting relationships, the dilemma of loose ends can occur. Figure 12 illustrates all split cases for nodes with relationships. SG$_n$ shows what pattern can exist before and SG$_{n+1}$ what could emerge from the `split` operation. The nodes in SG$_n$ contain two colors representing the properties. Consequently, an entity will be split at a defined property. Just the pattern of the relationships in the SG$_{n+1}$ vary. *Cases C, D* and *F*, include their vice versa cases for SG$_{n+1}$. The GEO demonstrates the split of nodes with associated relationships. Analogously, relationships - and their attached properties - can be split.

The SMO$_C$ in Figure 13 shows the `split` of a node resulting in two nodes keeping all relationships of the originals (see *Case E* in Figure 12). As indicated by the domain-specific SMO$_C$, there is no predefined function to perform a `split` resulting in a workaround.

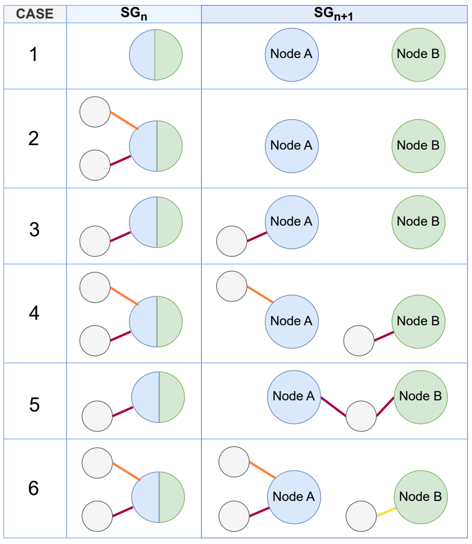

Figure 12: All split cases

First the selected nodes are copied inside of `getInitialNodesAndClones`. Then the value to `split` at (`splitKeyValueListAtProperty`) is defined and all clones and originals are collected to iterate over them, resulting in `ResultsA` and `ResultsB`. These are collections containing the maps used in `overwriteMaps*OfNodes*` to attach them to the nodes.

GEO

```
splitNodes ::= 'split' nodes 'at'
↪  propertyName ('keep' relationships
↪  | delete relationships | 'keep'
↪  relationships 'of' (partA | partB))
partA ::= properties 'till' propertyName
partB ::= properties 'from' propertyName
```

SMO$_C$

```
getInitialNodeAndClones
CALL '{' MATCH node (*key to split at
↪  is defined inside of*)
```

```
    splitKeyValueListAtProperty '}'
WITH collectOutput aliasOutput ','
↪   collectNodeVar aliasInitialNodes
↪   ',' keysA ',' keysB
loopOverKeysARemoveFromOutputAs
ResultsA','
↪   loopOverKeysBRemoveFromOutputAs
ResultsB ',' output ',' initialNodes
overwriteMapsResultsAOfNodesOutput
overwriteMapsResultsBOfNodes
InitialNodes
```

Figure 13: GEO and SMO$_C$ for: *Split* nodes at a defined property

5.8. Move

To change the graph structure by repositioning entity types, a move operation is needed. Associated relationships must be taken into account to avoid nodes being accidentally disconnected. The GEO for the evolution operation `move` (Figure 14) consists of two parts: moving an entity type or a feature. Another consideration has to be made for `moveNodes` as `connectedNodes` can sum up to an entire subgraph. We suspect that `moveNodes` is far more often used than moving a relationship because relationships are utilized to specify a precise connection between nodes. Substantial in the SMO$_C$ – in Figure 14 – to `move` nodes is the `extendedFilter`

GEO

```
moveNodes ::= 'move' connectedNodes 'to
↪  new' node
moveFeatures ::= 'move' features 'from
↪  original' entityTypes 'to new'
↪  entityTypes
```

SMO$_C$

```
selectPattern WITH (collect* alias*
↪  (*depdening on extended filter*))?
↪  ',' nodeVar (*old start node*) ','
```

```
    nodeVar (*new start node*)
selectSubgraph CALL copySubgraph
CALL '{'  selectPattern WITH
↪   collectNodeVar aliasOldPattern
    FOREACH
    ↪   rangeOfCollectionForOldPattern
    ↪   '|' FOREACH '('
    ↪   elementOfCollectionsByVar '|'
    ↪   nativeCascadeDelete ')' ')' '}'
RETURN INPUT ',' OUTPUT ',' ERROR
```

Figure 14: GEO and SMO$_C$ for: *Move* nodes with relationships

option via `apoc.path.subgraphAll`. This ensures flexibility as it offers a wide variety of filtering options like whitelisting nodes [25]. In order of moving all specified nodes connected to a particular end node, a subgraph is copied. Instead of keeping the clones – like in a copy operation – the former pattern is defined in the CALL block and deleted afterwards.

6. Conclusion and further work

The basic components and the workflow of schema evolution in GDBs are visualized in Figure 1. A precise realization is illustrated by Figure 15, showing how the levels analyzed in this paper work together. Displayed at the top is the GEO[6] in the orange box, representing how the `move` of nodes in GDBs can be generalized. The example chosen here is the same as in Section 5.8 that moves nodes and keeps all relationships. SG_n shows the GDB before the execution of the `move` operation and $\text{SG}_{\text{n+1}}$ is the result of the evolutionary process. On the bottom, the GDB is visualized referring to a precise GDB. There are two colored arrows labeled SMO_C[7] and GMO_C. Blue represents the schema level and green the operation on the graph. These directly show at what position and level SMO_C and GMO_C occur. The index C indicates that these levels are domain-specific for Cypher. In contrast, GEO is system independent for GDBs.

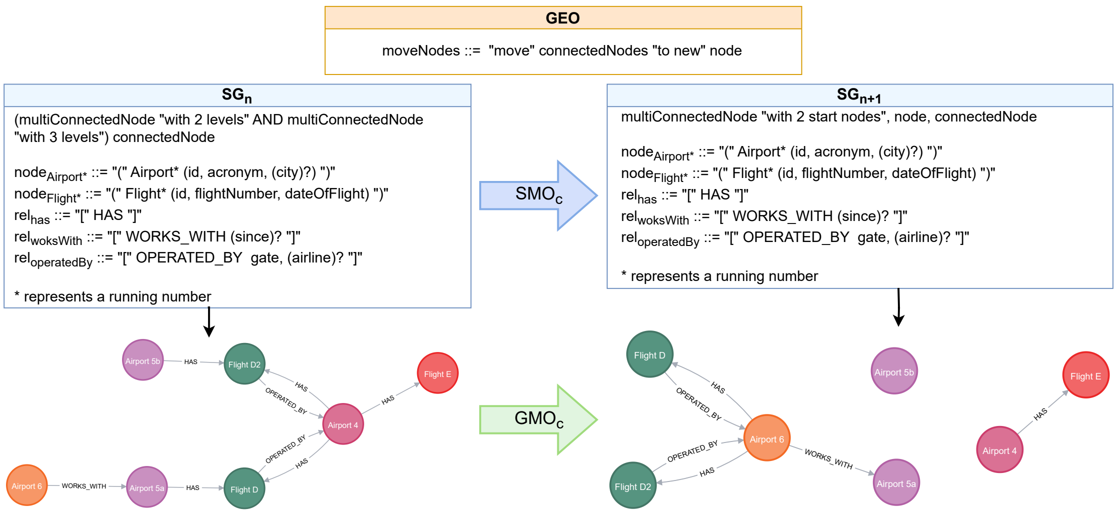
GEO
moveNodes ::= "move" connectedNodes "to new" node
SGn
(multiConnectedNode "with 2 levels" AND multiConnectedNode "with 3 levels") connectedNode
nodeAirport* ::= "(" Airport* (id, acronym, (city)?) ")"
nodeFlight* ::= "(" Flight* (id, flightNumber, dateOfFlight) ")"
relhas ::= "[" HAS "]"
relwoksWith ::= "[" WORKS_WITH (since)? "]"
reloperatedBy ::= "[" OPERATED_BY gate, (airline)? "]"
* represents a running number
SMOc
SGn+1
multiConnectedNode "with 2 start nodes", node, connectedNode
nodeAirport* ::= "(" Airport* (id, acronym, (city)?) ")"
nodeFlight* ::= "(" Flight* (id, flightNumber, dateOfFlight) ")"
relhas ::= "[" HAS "]"
relwoksWith ::= "[" WORKS_WITH (since)? "]"
reloperatedBy ::= "[" OPERATED_BY gate, (airline)? "]"
* represents a running number
GMOc
Airport 5b
HAS
Flight D2
Flight E
OPERATED_BY
Airport 4
Airport 6
WORKS_WITH
Airport 5a
Flight D

Figure 15: `Move` operation visualized analogous to Figure 1

One of the most important findings is that due to the structure of a GDB, consisting of nodes and relationships, a variety of evolutionary operations must be considered. This becomes apparent in Figure 12, displaying the variety of structures possible when performing a `split`. Moreover, this consideration led to the development of the `transform` operation, which only exists in GDBs. This kind of operation realizes a change in entity types, i.e. the process of changing a node into a relationship and vice versa. Also, we differentiated between *native* and

[6]GEO file: https://zenodo.org/record/8311214
[7]SMO_C file: https://zenodo.org/record/8311186

non-native options to show what is possible in Cypher and in which cases the APOC library is necessary. To accomplish a `move` or `split` operation workarounds were presented.

As implied in Section 5.1 in some cases the representation in Cypher differs depending on how, one or more entity types are created. This leads to the questions how these different presentations do affect the evolution. Our evolution does not consider virtual entity types yet. One of our future tasks is to conceptualize which types of evolution apply to which application scenarios involving virtual entities and to develop the evolution operations for them.

Acknowledgments

This work has been funded by Deutsche Forschungsgemeinschaft (German Research Foundation) - 385808805.

References

[1] D. Fernandes, J. Bernardino, Graph databases comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4j, and OrientDB, in: DATA, SciTePress, 2018, pp. 373–380.

[2] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, D. Vrgoc, Foundations of modern query languages for graph databases, ACM Comput. Surv. 50 (2017) 68:1–68:40.

[3] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, A. Taylor, Cypher: An evolving query language for property graphs, in: SIGMOD Conference, ACM, 2018, pp. 1433–1445.

[4] F. Holzschuher, R. Peinl, Performance of graph query languages: Comparison of Cypher, Gremlin and Native Access in Neo4j, in: EDBT/ICDT Workshops, ACM, 2013, pp. 195–204.

[5] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, A. Green, J. Hidders, B. Li, L. Libkin, V. Marsault, W. Martens, F. Murlak, S. Plantikow, O. Savkovic, M. Schmidt, J. Sequeda, S. Staworko, D. Tomaszuk, H. Voigt, D. Vrgoc, M. Wu, D. Zivkovic, PG-Schema: Schemas for property graphs, Proc. ACM Manag. Data 1 (2023) 198:1–198:25.

[6] M. Polák, M. Necaský, I. Holubová, DaemonX: Design, adaptation, evolution, and management of native xml (and more other) formats, in: iiWAS, ACM, 2013, p. 484.

[7] G. Guerrini, M. Mesiti, M. A. Sorrenti, XML schema evolution: Incremental validation and efficient document adaptation, in: XSym, volume 4704 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 92–106.

[8] J. Klímek, J. Malý, M. Necaský, I. Holubová, eXolutio: Methodology for design and evolution of XML schemas using conceptual modeling, Informatica 26 (2015) 453–472.

[9] A. H. Chillón, D. S. Ruiz, J. G. Molina, Towards a taxonomy of schema changes for NoSQL databases: The Orion language, in: ER, volume 13011 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 176–185.

[10] U. Störl, M. Klettke, Darwin: A data platform for schema evolution management and data migration, in: EDBT/ICDT Workshops, volume 3135 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022.

[11] A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Schuster,

P. Selmer, H. Voigt, Updating graph databases with Cypher, Proc. VLDB Endow. 12 (2019) 2242–2253.

[12] A. Bonifati, P. Furniss, A. Green, R. Harmer, E. Oshurko, H. Voigt, Schema validation and evolution for graph databases, in: ER, volume 11788 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 448–456.

[13] A. Bonifati, P. Furniss, A. Green, R. Harmer, E. Oshurko, H. Voigt, Schema validation and evolution for graph databases, 2019. URL: https://arxiv.org/abs/1902.06427.

[14] A. Bonifati, S. Dumbrava, N. Mir, Hierarchical clustering for property graph schema discovery, in: Proc. EDBT, OpenProceedings.org, 2022, pp. 2:449–2:453.

[15] I. Comyn-Wattiau, J. Akoka, Model driven reverse engineering of NoSQL property graph databases: The case of Neo4j, in: Proc. IEEE BigData, IEEE Computer Society, 2017, pp. 453–458.

[16] I. Comyn-Wattiau, J. Akoka, Query-based reverse engineering of graph databases - from program to model, in: Proc. ADBIS (Short Papers and Workshops), volume 1064 of *Communications in Computer and Information Science*, Springer, 2019, pp. 188–197.

[17] A. A. Frozza, S. R. Jacinto, R. dos Santos Mello, An approach for schema extraction of NoSQL graph databases, in: Proc. IRI, IEEE, 2020, pp. 271–278.

[18] Neo4j, Inc., Merge, 2023. URL: https://neo4j.com/docs/cypher-manual/current/clauses/merge/, accessed: 2023-06-07.

[19] Neo4j, Inc., Delete, 2023. URL: https://neo4j.com/docs/cypher-manual/current/clauses/delete/, accessed: 2023-08-29.

[20] Neo4j, Inc., apoc.create.removeLabels, 2023. URL: https://neo4j.com/docs/apoc/current/overview/apoc.create/apoc.create.removeLabels/, accessed: 2023-08-30.

[21] Neo4j, Inc., apoc.create.removeProperties, 2023. URL: https://neo4j.com/docs/apoc/current/overview/apoc.create/apoc.create.removeProperties/, accessed: 2023-08-30.

[22] Neo4j, Inc., apoc.create.removeRelProperties, 2023. URL: https://neo4j.com/docs/apoc/current/overview/apoc.create/apoc.create.removeRelProperties/#usage-apoc.create.removeRelProperties, accessed: 2023-08-30.

[23] Neo4j, Inc., Creating data, 2023. URL: https://neo4j.com/labs/apoc/4.0/graph-updates/data-creation/, accessed: 2023-06-22.

[24] Neo4j, Inc., Merge nodes, 2023. URL: https://neo4j.com/labs/apoc/4.3/graph-updates/graph-refactoring/merge-nodes/, accessed: 2023-06-07.

[25] Neo4j, Inc., apoc.path.subgraphAll, 2023. URL: https://neo4j.com/labs/apoc/4.3/overview/apoc.path/apoc.path.subgraphAll/, accessed: 2023-08-24.