Nautilus: Implementation of an Evolution Approach for Graph Databases

Dominique Hausler*
dominique.hausler@ur.de
University of Regensburg
Regensburg, Bavaria, Germany

Meike Klettke
meike.klettke@ur.de
University of Regensburg
Regensburg, Bavaria, Germany

ABSTRACT

Equivalent to relational databases, there is a need for an evolution language for graph databases that describes how evolution operations such as add, rename, delete, copy, move, split and merge are specified domain independent. Previous work proposes the graph evolution language called GEO, which we build upon.

In this paper, we present our program called Nautilus, implementing this formal language, used to define evolution and intuitively easing the usage of graph database systems. GEO can also be used to update implicit structures in the graph data. Users benefit not only from an easy-to-use interface to minimize syntax errors and to reduce the necessary knowledge of the evolution language, but also from additional statistics on database structures which are visualized in the tool. This visualization allows initial data exploration as well as identifying the effects of the development by comparing data versions.

Consequently, Nautilus is capable of widening the range of users and accessibility of graph databases for interdisciplinary research projects. Illustrating schema changes and performing schema evolution transparently builds the core of Nautilus. Complex operations like split and transform are part of the available evolution language, thus avoiding programming workarounds. An additional feature of the tool is a logging components that offers the traceability of all performed evolution operations.

CCS CONCEPTS

• **Information systems**; • **Data management systems**; • **Database administration**; • **Database utilities and tools**;

KEYWORDS

Graph Databases, Property Graph, Evolution Language, Graph Database Statistics, Profiles, Neo4j

ACM Reference Format:
Dominique Hausler and Meike Klettke. 2024. Nautilus: Implementation of an Evolution Approach for Graph Databases. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3652620.3687781

*Corresponding author.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike International 4.0 License.
MODELS Companion '24, September 22–27, 2024, Linz, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3687781

1 INTRODUCTION AND MOTIVATION

All graph databases which are used over long periods of time have to be evolved. Changing requirements as well as a re-design of their structures lead to changes e.g., new features may be added, renamed or a refactoring takes place. In all these cases, an evolution of the graph databases is necessary. Although query and update languages [16] are currently available, there is as far as we know no explicit evolution language. Whereas evolution approaches exist for other data models [10, 12, 22], a similar approach for graph databases is still missing, especially in the context of the arising research question: *How to make schema evolution in graph databases available to a bigger group of users, ensuring good usability?* We have developed our system for graph databases in general, due to the popularity of Neo4j [20]. This is the first system used in our prototype named Nautilus.

Contribution. The novelty of Nautilus consists of:

- Our evolution language GEO (Graph Evolution Operation) from [15], using an intuitive syntax to ease the comprehension of evolution operations.
- Due to filter options, GEO can also be used to perform updates for a specified part of the graph database.
- The graph-specific evolution operation `transform` is integrated, showing the difficulties when working with highly interconnected data.
- The integration of complex operations like split or move which are not available in **G**raph **Q**uery **L**anguages (GQLs).
- **S**tructural **D**atabase **S**tatistics (SDS) are used to show the impact of evolution and to estimate the evolutionary effort.

Structure. The rest of this article is sectioned as followed: First, preliminary and related work is analyzed. Section 3 focuses on Nautilus, including the target user group (3.1), followed by an overview over the program components (3.2) and the structural database statistics – a hybrid approach combining schema and statistical data (3.3). In 3.4 we discuss the realization of evolution operations, followed by the application spectrum (3.5) and an example usage of Nautilus in 3.6. A summary is given in Section 4, together with some future work tasks.

2 RELATED WORK & PRELIMINARY WORK

As a result of Cypher's popularity in research [1, 2, 9, 19] as well as in commercial applications [13, 20], the evolution language GEO was based on Neo4j's query language. Moreover, standard conformity to the newly released ISO GQL [16] is proposed by Neo4j [21]. Consequently, Nautilus also uses Neo4j.

Schema Extraction. **G**raph **d**ata**b**ases (GDB) can be schemaless, making schema-second approaches necessary to extract schema data from an existing database. Several extraction services for graph

Graph Evolution Operation – GEO

```
addFeatures ::= 'add' label Student 'to' node 'with'
    label Person
```

Schema Modification Operation – SMO_C

```
WITH parameter AS old_label selectPattern
((CALL addLabels YIELD node)| (CALL overwriteLabelsOf
    old_labelWithnew_label))
RETURN node
```

Graph Manipulation Operation – GMO_C

```
WITH $old_label AS old_label
MATCH (a) WHERE apoc.label.exists(a, $old_label)
CALL apoc.create.setLabels(a, $list_of_new_labels)
YIELD node RETURN node
```

Figure 1: The Evolution Operation "add label to node" and its Realization in Neo4j

databases are at hand [3, 9, 14]. For our approach we make use of the **A**wesome **P**rocedures **O**n **C**ypher (APOC) library's functions to extract schema data.

Evolution. In contrast to [4–6], the novelty of our evolution language implemented in Nautilus is the extension of the graph-specific `transform` operation, illustrating the complexity of GDBs. `transform` describes the process of changing the entity type e.g., converting a node into a relationship and vice versa.

Schema Visualisation. There are several tools to explore highly interconnected data such as [7, 11, 17, 25]. Such visualisations help to give a brief overview over the data already available. In the context of evolution an available tool including a visualization is presented in [23].

Own Preliminary Work & Theoretical Background. GEO defines operations similar to those used in other database systems. In relational databases, the `alter table` statements are applicable to specify single-type evolution operations. For JSON databases, many authors suggest more complex single and multi-type evolution operations e.g., [8, 18, 24]. Our program focuses on graph databases, adapting the operations to graph-specific features. In addition to these works, we implement a first collection of **filter functions**. Figure 1 shows an example for all three levels. From the platform independent GEO[1] our system derives specific **S**chema **M**odification **O**perations – SMO_C – for an exact translation into Neo4j. Platform dependency is indicated by the **C** index for Cypher. At the data level, the **G**raph **M**anipulation **O**peration – GMO_C, specifies what the graph query looks like. The orange box shows the evolution language, explaining how an operation is defined for graph databases in general. In the sample in Figure 1 the label `Student` is added to all `Person` nodes. SMO_C forms the theory behind the precise implementation on the schema level, while GMO_C represents the Cypher query.

[1]GEO file: https://zenodo.org/record/8311214

3 NAUTILUS

Nautilus implements GEO (**G**raph **E**volution **O**peration) presented in [15]. It is able to describe evolution operations domain independently. Due to GEO being realized through a formal language, it eases the understanding of how evolution operations such as *add*, *rename*, *delete*, *copy*, *move*, *split* or *merge* work. Nodes and relationships are referred to as *entity types* whereas labels, types and properties are called *features*.

3.1 Target User Group

As GEO mimics natural language, the program can be operated by a wide range of users consisting of:

- Non-computer scientists who want to use a graph database for storing, updating and manipulating their data.
- Computer scientists who want to realize updates and evolution operations with an intuitive language.

Consequently, the main impact emerges for interdisciplinary projects with the benefit of significantly reducing the time to learn a new query language. An example for bringing both user groups together is the `type` field (see Figure 3 number ①). Here headings are displayed in the drop-down menus, using technical terminology for professionals, while non-professionals are still able to use the program without this knowledge.

3.2 Overview of Nautilus

Figure 2 illustrates how the evolution language, the SMO_C and the precise implementation GMO_C are connected to each other in Nautilus. Nautilus makes use of `parameters` to increase the performance of the executed queries. This can be seen in Figure 1, illustrating how `add label(s)` is implemented. The program is based on the Django-framework, using Python backend and JavaScript, HTML and CSS for the frontend.

As already indicated, the main part of the program contains our evolution language GEO, shown by the Domain Independent Evolution Layer in Figure 2. The eye-symbol symbolizes the direct visibility to the user. GEO is – depending on the user's input via **2** – composed on the platform independent level. Subsequently, with each input the next part of the evolution language is displayed. Either one or multiple (indicated by the `1..n`) evolution operations can be executed sequentially at once. Additionally, for the applicability of the tool, all executed GEOs are saved in a history, directly showing GEO to the user **3.1** together with the option of a log file **3.2**. The entered connection data can be remembered, making only one intake necessary (**4**). To extract the initial **S**tructural **D**atabase **S**tatistics – SDS_n – on the Schema Layer (**6**) this information is sufficient to generate a scatter plot (**7**). After submitting the evolution form, the associated query (GMO_C) is identified. SMO_C defines the evolution operations by a grammar and considers graph characteristics (**9**). An example, illustrating all levels with GEO, SMO_C and GMO_C was given in Figure 1.

3.3 Structural Database Statistics – SDS

To show the impact and effects of evolution, Nautilus offers a visualization showing **S**tructural **D**atabase **S**tatistics. Moreover, the SDS before – SDS_n – and after – SDS_{n+1} – the evolution are compared.

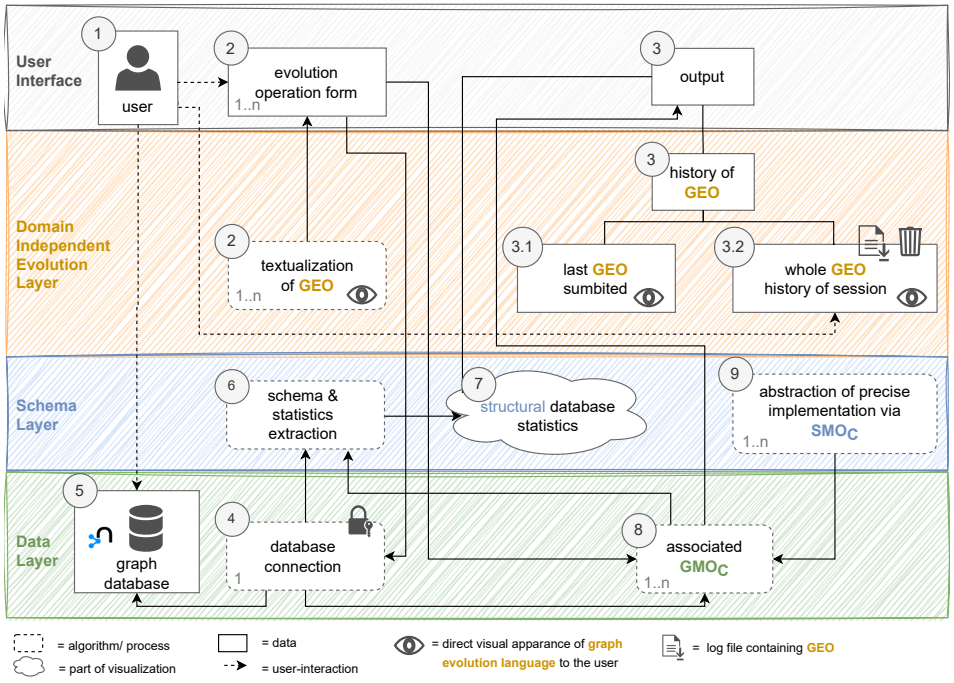

Figure 2: Process of Executing Evolution Operations in Nautilus

SDS represent a **hybrid approach** combining schema and statistical information. The SDS aim to help explore the data at version n as well as illustrating the impact of the evolution by comparing the SDS at n and n+1. In addition, the number of affected entities can be identified via the SDS.

Formal Description. A graph G is defined as tuple consisting of a set of vertexes (=nodes) V and relationships (= edges) E. Like in [14] this is extended by a database name n_{db}.

$$G = (n_{db}, V, E)$$

From G a set of *SDS* for both entity types SDS_V and SDS_E are extracted. As there are currently two ways of illustrating data, *SDS* are defined for the tabular and the graphical visualization.

$$f(G) = (SDS_V, SDS_E)$$

The SDS_V in the **table** are defined as single label l, a set of property keys P solely limited to their names n_p together with a set of types of associated relationships $T|_{E_a}$. Examples are shown in Figure 3 part ④. In contrast, the **diagram** shows the number of occurrences c_l, a set of property keys – with their name and datatype dt_p – as well as associated relationships E_a. E_a are specified as a single type t, count c_t and direction D, defined as boolean. The graphical illustration is shown in Figure 3 number ②.

$$SDS_V = \begin{cases} \{(l, P|_{n_p}, T|_{E_a})\} & , \text{ table} \\ \{(l, c_l, P, E_a)\} & , \text{ diagram} \end{cases}$$

$$P = (n_p, dt_p)$$

$$E_a = \{(t, c_t, D) \mid t \neq \text{''}\}$$

In opposition, SDS_E has the condition $t \neq$ '' i.e., that a type t can not be an empty string. The tabular SDS_E are defined as a boolean for the direction D, a set of start L_{SN} and end node labels L_{EN} and a set of property keys P limited to their names n_p – analogous to SDS_V. For the scatter plot each relationship is defined as its type t together with the count c_t, the direction D and the property keys P. The SDS of the relationships are visualized through a scatter plot and a table similar to those of the nodes in Figure 3 number ② and ④.

$$SDS_E = \begin{cases} \{(t, D, L_{SN}, L_{EN}, P|_{n_p}) \mid t \neq \text{''}\} & , \text{ table} \\ \{(t, c_t, D, P) \mid t \neq \text{''}\} & , \text{ diagram} \end{cases}$$

The visualization of the SDS serves two purposes. First, to assist the user in writing GEO by providing an overview over the initial data. Second, to compare the data before and after executing evolution operations in one graphic and to demonstrate data changes.

3.4 Realization of Evolution Operations

Table 1 demonstrates the precise realization of evolution operations at the data level (Figure 2 number **8**). For some functions **subcategories** are available. For instance, adding a label can be executed in two different ways, as illustrated by SMO_C in Figure 1. Either all given labels can be added by `addLabels` or the old values can be overwritten. The Neo4j logo shows that the GMO_C can be performed by a **workaround** with Cypher and the APOC library. `Move labels` for example has subcategories depending on the number of labels to move while using a workaround. The check mark inherits another difference. For instance, a `delete node` is

Table 1: Realization of Evolution Operations

	Single-type				Multi-type			
	Add	Rename	Delete	Transform	Copy	Move	Split	Merge
Node	✓✗	–	✓	✓/↻	✓✗	✓✗/↻	↻	↻/🐍
Rel-ship	✓✗	–	✓	✓/↻	–	✓✗/↻	↻	↻/🐍
Label	✓✗	✓	✓	–	✓	✓✗/↻	–	–
Type	–	✓	–	–	–	–	–	–
Property	✓	✓	✓	↻	↻/🐍	↻/🐍	–	–
Direction	–	–	–	–	–	↻	–	–

✓✗ Subcategories existing
– Unavailable operation for graph databases
✓ *Native* or *non-native* command
↻ Workaround with Cypher (and APOC)
↻/🐍 Combined approach enabling collection handling

conducted *native*, meaning with Cypher commands only, whereas all `rename` operations make use of the APOC library and thus, are called *non-native*.

The operation `copy properties` is a **combined approach** utilizing Cypher to access the selected entities together **with Python**, parsing the data. Using `setProperties` would result in overwriting keys already present in the entity to copy to.

Some evolution operations are not available for graph databases such as `add type`. This is due to the internal structure, only allowing one `type` for a relationship, which has to be defined upon their creation. Subsequently, `delete type` is equivalent to `delete relationship`. Moreover, `split` and `merge` are only applicable for entity types. Accordingly, `rename` is merely at hand for features. `Transform` is exclusively for graph databases, describing how a node is converted to a relationship and vice versa or a property to an entity type.

3.5 Application Spectrum

Nautilus offers a range of functions and features for various application scenarios. The three most important functions are:

(1) Execution of evolution operations without a GQL.
(2) Analysis of performed evolution operations via a log file.
(3) Visualization of the evolutionary impact through SDS.

All three components refer to the target user groups defined in Subsection 3.1 as being either non-experts or experts making use of an intuitive language. They further define when Nautilus is beneficial and what this support looks like.

3.6 Example Usage/ Capabilities of Nautilus

To explore the initial data, the scatter plot with the SDS_n can be requested. The advantages are a) getting an overview over the available data and b) simplifying the definition of GEO. To accomplish that, a user merely needs to connect to the database and generate the visual output displayed in Figure 3 part ②. Figure 3 also displays how an `add` together with a `rename` operation would be shown. A `delete` operation would result in an identical illustration as the `rename` operation when merely looking at an extract without taking the `history` (Figure 2 number **3**) into account.

Afterwards, several evolution operations can be executed. The `evolution operation form` (Figure 3 number ①) offers the advantage of displaying each operation in our formal language GEO. Hence, users can straightforwardly read how their database will be effected. As submit output, the SDS_{n+1} are displayed together with the SDS_n to visualize data changes like in Figure 3 number ②. The evolution *history* is displayed in the output section numbered ③. The other tabs – e.g., the tab shown in ④ – contain tabular information of the SDS_{n+1} separated by entity types. Besides all GEOs being directly illustrated, recapitulating the actions taken during the session is possible via a text file (**3.2**). This file contains all performed queries and can be downloaded. In case of switching the database, the file can be cleared.

4 CONCLUSION AND FUTURE WORK

Nautilus offers a wide range of evolution operations implemented via the Django framework and executed on a Neo4j graph database. Users benefit from the implementation of GEO by enabling them to access and update their database without the knowledge of a GQL. GEO is intuitive to use because drop-down menus automatically assemble the language to describe what each operation does. Moreover, GEO offers a first selection of filter options to perform updates. We will add additional subcategories momentarily not available. This, for example, includes filtering for a selected label in combination with one or more property keys. To gain insight into the understandability of GEO and the UI, a task-based study using a **thinking aloud** approach will follow. In the long term, we plan to implement a **preview** option that will allow the user to get an overview of the impact an evolution operation has on the graph database, along with an estimation of the time this operation will take upon its execution.

ACKNOWLEDGMENTS

This work has been funded by Deutsche Forschungsgemeinschaft (German Research Foundation) – 385808805. Special thanks goes to Tanja Auge for the input during our discussions.

REFERENCES

[1] Junhua Bai and Lei Che. 2021. Construction and Application of Database Microcourse Knowledge Graph Based on Neo4j. In *The 2nd International Conference on Computing and Data Science* (Stanford, CA, USA) *(CONF-CDS 2021)*. Association for Computing Machinery, New York, NY, USA, Article 68, 5 pages. https://doi.org/10.1145/3448734.3450798
[2] Ioannis Ballas, Vassilios Tsakanikas, Evaggelos Pefanis, and Vassilios Tampakas. 2021. Assessing the computational limits of GraphDBs' engines - A comparison study between Neo4j and Apache Spark. In *Proceedings of the 24th Pan-Hellenic Conference on Informatics* (Athens, Greece) *(PCI '20)*. Association for Computing Machinery, New York, NY, USA, 428–433. https://doi.org/10.1145/3437120.3437356
[3] Angela Bonifati, Stefania-Gabriela Dumbrava, Emile Martinez, Fatemeh Ghasemi, Malo Jaffré, Pacome Luton, and Thomas Pickles. 2022. DiscoPG: Property Graph Schema Discovery and Exploration. *Proc. VLDB Endow.* 15, 12 (2022), 3654–3657.
[4] Angela Bonifati, Stefania Dumbrava, and Nicolas Mir. 2022. Hierarchical Clustering for Property Graph Schema Discovery. In *Proc. EDBT*. OpenProceedings.org, 2:449–2:453.

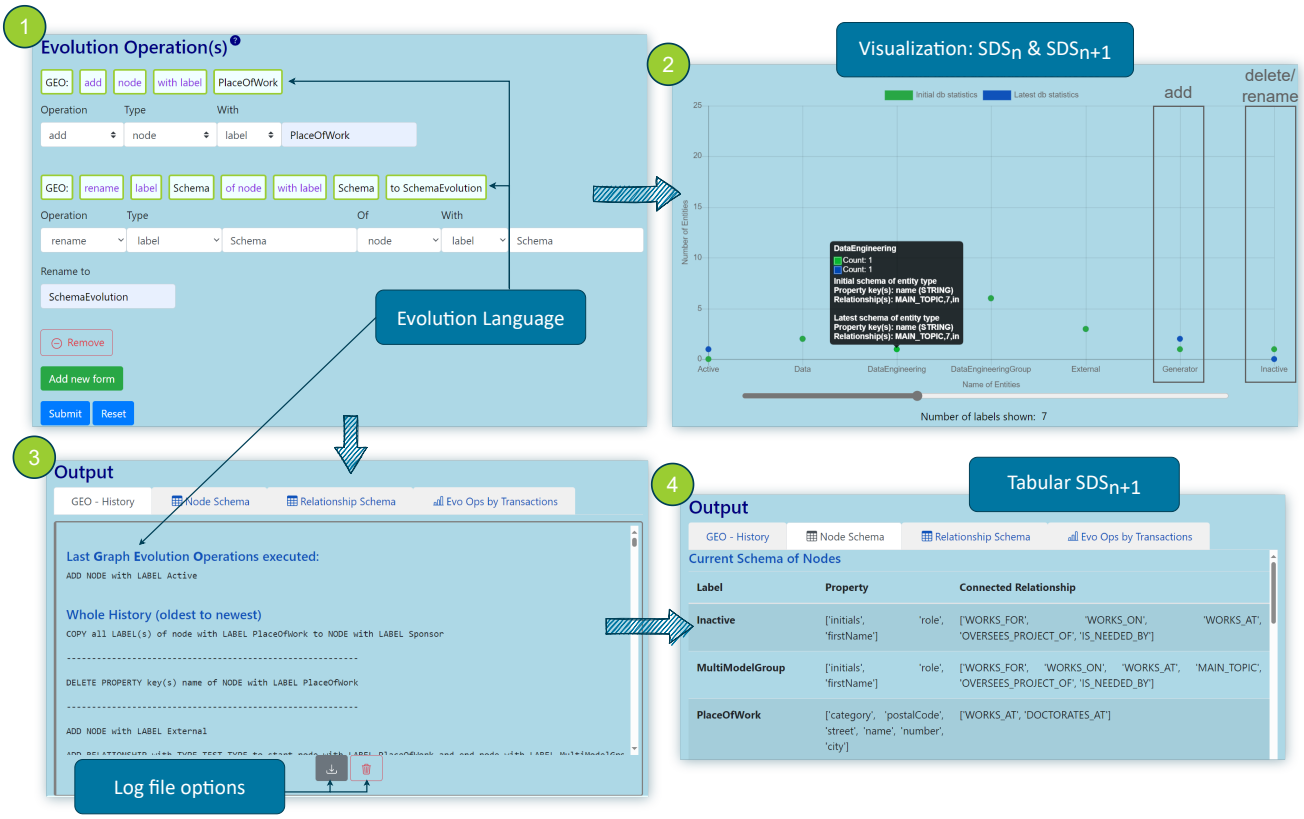

Figure 3: Components of Nautilus

[5] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. 2019. Schema Validation and Evolution for Graph Databases. In *ER (Lecture Notes in Computer Science, Vol. 11788)*. Springer, 448–456.

[6] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. 2019. Schema Validation and Evolution for Graph Databases. https://arxiv.org/abs/1902.06427

[7] Enrico Giacinto Caldarola, Antonio Picariello, and Antonio M. Rinaldi. 2015. Experiences in WordNet Visualization with Labeled Graph Databases. In *IC3K (Communications in Computer and Information Science, Vol. 631)*. Springer, 80–99.

[8] Alberto Hernández Chillón, Meike Klettke, Diego Sevilla Ruiz, and Jesús García Molina. 2022. A Taxonomy of Schema Changes for NoSQL Databases. *CoRR* abs/2205.11660 (2022).

[9] Isabelle Comyn-Wattiau and Jacky Akoka. 2017. Model driven reverse engineering of NoSQL property graph databases: The case of Neo4j. In *Proc. IEEE BigData*. IEEE Computer Society, 453–458.

[10] Carlo Curino, Hyun Jin Moon, and Carlo Zaniolo. 2008. Graceful database schema evolution: the PRISM workbench. *Proc. VLDB Endow.* 1, 1 (2008), 761–772.

[11] Niels De Joung. 2013. 15 Tools for Visualizing Your Neo4j Graph Database. https://neo4j.com/developer-blog/15-tools-for-visualizing-your-neo4j-graph-database/ Accessed: 2024-07-04.

[12] Torben Eckwert, Michael Guckert, and Gabriele Taentzer. 2022. EvolveDB: a tool for model driven schema evolution. In *MoDELS (Companion)*. ACM, 61–65.

[13] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD Conference*. ACM, 1433–1445.

[14] Angelo Augusto Frozza, Salomão Rodrigues Jacinto, and Ronaldo dos Santos Mello. 2020. An Approach for Schema Extraction of NoSQL Graph Databases. In *IRI*. IEEE, 271–278.

[15] Dominique Hausler, Meike Klettke, and Uta Störl. 2023. A language for graph database evolution and its implementation in Neo4j. In *ER (Companion) (CEUR Workshop Proceedings, Vol. 3618)*. CEUR-WS.org.

[16] ISO/IEC 39075:2024 2024. *Information technology— Database languages — GQL*. Standard. International Organization for Standardization, Geneva, CH.

[17] Weihao Jiang, Li Yan, Yaofeng Tu, Xiangsheng Zhou, and Zongmin Ma. 2022. PG-explorer: Resource Description Framework data exploration with property graphs. *Expert Syst. Appl.* 198 (2022), 116789. https://doi.org/10.1016/J.ESWA.2022.116789

[18] Pavel Koupil, Jáchym Bártík, and Irena Holubová. 2022. *MM-evocat:* A Tool for Modelling and Evolution Management of Multi-Model Data. In *CIKM*. ACM, 4892–4896.

[19] Haibo Liu, Guoyi Jiang, Linhua Su, Yang Cao, Fengxin Diao, and Lipeng Mi. 2020. Construction of power projects knowledge graph based on graph database Neo4j. In *CITS*. IEEE, 1–4.

[20] Neo4j, Inc. 2024. Graph Database Case Studies. https://neo4j.com/case-studies/ Accessed: 2024-02-27.

[21] Neo4j, Inc. 2024. Neo4j Welcomes New GQL International Standard in Major Milestone for Database Industry. https://neo4j.com/press-releases/gql-standard/ Accessed: 2024-05-15.

[22] Stefanie Scherzinger, Stephanie Sombach, Katharina Wiech, Meike Klettke, and Uta Störl. 2016. Datalution: a tool for continuous schema evolution in NoSQL-backed web applications. In *QUDOS@ISSTA*. ACM, 38–39.

[23] Uta Störl and Meike Klettke. 2022. Darwin: A Data Platform for Schema Evolution Management and Data Migration. In *EDBT/ICDT Workshops (CEUR Workshop Proceedings, Vol. 3135)*. CEUR-WS.org.

[24] Pablo Suárez-Otero, Michael J. Mior, María José Suárez Cabal, and Javier Tuya. 2023. CoDEvo: Column family database evolution using model transformations. *J. Syst. Softw.* 203 (2023), 111743.

[25] Michael Thane, Kai M. Blum, and Dirk J. Lehmann. 2023. CatNetVis: Semantic Visual Exploration of Categorical High-Dimensional Data with Force-Directed Graph Layouts. In *25th Eurographics Conference on Visualization, EuroVis 2023 - Short Papers, Leipzig, Germany, June 12-16, 2023*, Thomas Höllt, Wolfgang Aigner, and Bei Wang (Eds.). Eurographics Association, 91–95. https://doi.org/10.2312/EVS.20231049