DTGraph: Declarative Transformations of Property Graphs

Angela Bonifati
Lyon 1 Univ., Liris CNRS & IUF
angela.bonifati@univ-lyon1.fr

Yann Ramusat
Lyon 1 Univ., Liris CNRS
yann.ramusat@liris.cnrs.fr

Filip Murlak
Univ. of Warsaw
f.murlak@uw.edu.pl

Amela Fejza
Lyon 1 Univ., Liris CNRS
amela.fejza@liris.cnrs.fr

Rachid Echahed
LIG – Univ. Grenoble Alpes
rachid.echahed@imag.fr

ABSTRACT

Current graph query languages, including the standards SQL/PGQ and GQL, define their semantics in terms of sets of tuples. This is largely inadequate for data interoperability tasks such as data migration or data integration which require queries to output new property graphs. This demonstration showcases DTGraph, an open-source declarative rule-based framework for easily specifying and efficiently executing property graph transformations. We describe a novel comprehensive system that allows the declarative specification of property graph transformations, by extending openCypher queries with a new **GENERATE** clause for creating new property graphs. The system includes several modules: a parser, a compiler for translating the transformation logic into an efficient executable openCypher script, and an interface assisting users in developing their transformations. The demonstration showcases the ability of our framework to scale to large graph data, and its suitability for transforming real-world datasets.

PVLDB Reference Format:
Angela Bonifati, Yann Ramusat, Filip Murlak, Amela Fejza, and Rachid Echahed. DTGraph: Declarative Transformations of Property Graphs. PVLDB, 17(12): 4265 - 4268, 2024.
doi:10.14778/3685800.3685851

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yannramusat/DTGraph.

1 INTRODUCTION

Property graph query languages are quickly evolving, with ongoing development and diverse proposals, including Neo4j's openCypher [7], Oracle's PGQL [12] and the international standards SQL/PGQ and GQL [6]. In data interoperability scenarios requiring the generation of new property graphs, the semantics of those query languages – defined as sets of tuples, becomes overly restrictive.

Current practical solutions for transforming property graphs either (i) rely on opaque external libraries, such as Neo4j's APOC [9], or (ii) involve complex handcrafted queries, or (iii) could even be externally implemented with general-purpose languages such as Python. Meanwhile, *declarative specifications* have long been recognized as pivotal for solving data programmability problems [3]. They have clear semantics that allow them to be reused, composed, and treated as first-class citizens in model management tasks. Due to their ad-hoc nature and inherent complexity, handcrafted queries written in openCypher rarely offer such benefits. On the other hand, the APOC library does not have out-of-the-box solutions for property graphs transformations. It supports graph projections with the concept of virtual nodes and relationships that do not exist in the graph (only returned by a query): this means that these projections *do not persist in the graph*. Moreover, this library is vendor-specific, hence it would not provide a system-agnostic universal solution for this task; for instance, Memgraph's MAGE [8] does not have such primitives.

We demonstrate a novel system, DTGraph, which is to the best of our knowledge the first system that allows users to formulate transformations of property graphs in a declarative and intuitive manner, aiming to follow the recommendations for data integration and data exchange scenarios discussed in [3]. Indeed, our solution goes well beyond implementing a simple transformation language by providing features to assist the user during the development process. The rules are independent of each other and the transformation can be built gradually; this means that independent business rules can be integrated into a complex transformation. Ambiguity in the data can be identified early in the process using built-in primitives to detect inconsistencies in the produced output. Finally, the runtime environment executing the transformation is simply the query processor of the underlying database system, so our solution offers full compatibility with several openCypher compatible backends such as Neo4j or Memgraph, without needing to modify those.

The purpose of this demonstration is to showcase in an interactive manner: (i) how users can build their own transformations on real-world graph data and benefit from the facilities offered by the system; and (ii) the efficiency of the system in building the target property graphs and (iii) its scalability in transforming large graph-shaped data.

2 FOUNDATIONS

In the following, we assume the property graph data model of [6]. Property graphs have the following characteristics:

- They contain nodes and edges having a *unique* identifier.
- Each node and edge can have zero or more labels.
- Each node and edge can have properties (key-value pairs).
- Each property is single-valued (*atomic*).
- Each edge is directed.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685851

Our demonstrated system, DTGraph, takes a declarative, rule-based approach to transforming property graphs. This system is built on top of openCypher queries and supports sets of *property graph transformation rules* that can be of two kinds: node rules and edge rules. These rules connect the output of a query over the input property graph with descriptions of new elements to be included in the output property graph. The full syntax and the semantics of the transformation language can be found in [5].

Arbitrary openCypher queries can be used for extracting information from the source property graph, provided that they only return Node and Relationship structural types [10]. We connect their output to *constructors* which are expressed in a syntax resembling that of openCypher, contained in a new **GENERATE** clause that we introduce for this purpose. For instance, the following rule

```
1  MATCH (n:FirstName)-[r]->(m:LastName)
2  GENERATE (x = (r):FullName {value = n.value + m.value})
```

consists of a left-hand side extracting information from the source using a **MATCH** clause, and a right-hand side defined by a **GENERATE** clause which contains one *node constructor* consisting of three fields. The first field `(r)` specifies the identity of the node. It contains a list of expressions that can be source variables (as in the example), access keys $x.a$ for x a source variable and a an attribute name, or data values. The values of these expressions are passed to a *Skolem function*, which is an injective function defining the identity of a new element from the given arguments. This allows several rules to refer to the same element, provided that the expression lists in their respective constructors take the same values. The second field `:FullName` specifies the label, and the third field `{value = n.value + m.value}` specifies the properties of the element (+ denotes string concatenation). Importantly, the transformation rules do not forbid additional labels and properties, which will allow the user to split the description of an output element across multiple rules, if the user so desires. Section 4 showcases an interaction between two rules, where each rule defines its own label and properties.

Conflict detection. The data model we consider, unlike the one of openCypher, imposes single-valued properties. In openCypher, properties are multi-valued: they can store lists of data values. Restricting to atomic property values is crucial to transpose the achievements that have been made for relational databases to property graphs. Recall that in relational databases, the first normal form assumes atomicity. The work on normalization of property graphs [11] and the solutions proposed to cope with data inconsistencies in data integration tasks [3] all assume atomicity of the data. DTGraph is able to identify *conflicts*, which occur when two rules specify different values for a property of an output element; and provides suitable feedback to the user. Section 4 describes a complete scenario where users benefit from the interaction with the system to build their own meaningful transformation.

Advantages of this approach. The use of Skolem functions to implement a mapping language has already been studied, e.g., in [2, 4], but never to date on property graphs. This approach offers several benefits, (i) there is a unique well-defined output for each input, hence the set of rules defines a function mapping property graphs to property graphs, (ii) the order in which the rules are applied does not impact the produced output, (iii) it permits to keep track of conflicting attributes on the produced property graph.

We leverage this in DTGraph to design a system that allows users to: (i) incrementally build their transformations (because rules model independent business rules and do not depend on each other, they can be added in any order to a transformation scenario and the output can be visualized and investigated during the development process) and (ii) receive relevant feedback about the current output (the system includes an error reporting tool that can display relevant metadata that include the number of elements created, and the number of conflicts that have been generated).

In this demonstration, we will showcase in Section 4 concrete scenarios on real graph data where the user benefits from the unique features of DTGraph presented above to develop a transformation that meets their business needs.

3 SYSTEM ARCHITECTURE

Our system is implemented as an open-source Python3 package, freely available on GitHub[1]. It exposes a programmatic API that empowers users with the ability to specify their own transformations with declarative rules and to execute them on any openCypher compatible back-end. It has currently been tested on Neo4j 5.16.0 and Memgraph 2.14.0, the latest publicly available versions of both systems. We now present in detail the components of our system as shown in Figure 1.

The interface is based on interactive JupyterLab notebooks. It allows users to incrementally build and validate their transformations by adding new rules to currently active transformations, thus permitting an incremental development process. The interface also displays information and statistics on the execution context (query execution time, number of generated elements, information about conflicts, etc.). The output of the transformation is persisted in the database and can be visualized throughout the process using the vendor's built-in visualization tools such as Neo4j browser and Memgraph lab.

A parser and a compiler that parse the input rules and generate efficient openCypher scripts. The transformation rules introduced in Section 2 are expressed in our own *Domain Specific Language*, which complements the syntax of openCypher. The right-hand side of a rule corresponds to a new Cypher clause we introduce for this purpose, called **GENERATE**. We support a streamlined syntax close to the one of openCypher for our node and edge constructors. Examples of rules are given in Section 4.

Then our rules are compiled into efficient openCypher scripts. Different back-ends may have mild discrepancies, such as variations in the syntax for creating and removing indexes. Consequently, we may have minor differences in the compiled script depending on which back-end is used.

A back-end module which encapsulates a Neo4j bolt connector for Python to execute the transformation scripts produced by the compiler, and retrieves the statistics which are made available to the users. This module manages the transformation: it sets up the indexes and metadata needed to execute the transformation efficiently, and removes them when the transformation has been validated by the user.

[1] https://github.com/yannramusat/DTGraph

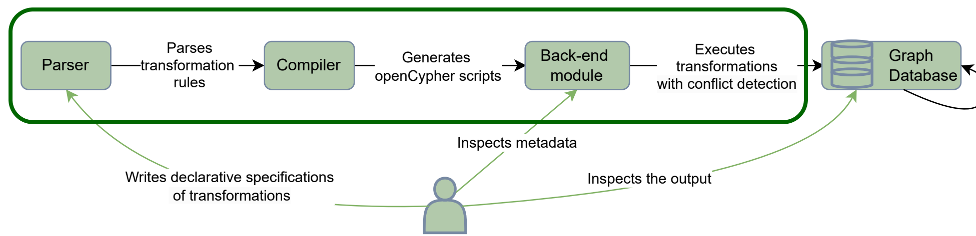

Figure 1: Architecture of the DTGraph system.

Advantages of this architecture. The demonstrated system is a complete and novel solution for extending openCypher with the ability to produce complex property graphs, given a specification consisting of a set of declarative rules. It is a layer on top of openCypher, implemented as a Python3 library. As such, it is compatible with any openCypher's compatible backends such as Neo4j or Memgraph, without the need to modify those.

Graph transformation API. Users can manage a transformation t by interacting with the system through the following functions.

- t.abort() sets the state of t to inactive and removes from the database the current output of the transformation.
- t.add($rule$) adds a declarative transformation rule to the transformation object t. If the transformation is already active on a graph, the rule is parsed, compiled into an openCypher script and this script is executed on this graph and the output of the current transformation is updated.
- t.apply_on($graph$) executes all the rules of the transformation on the given $graph$ database. The output is created in the same database, but is disconnected from the input data. t is now active on $graph$.
- t.diagnose() displays the possible output elements that have a conflict in one of their attributes.
- t.eject($destructive$) removes all internal bookkeeping data on the output graph and sets the state of t to inactive. If $destructive$ is set to $true$, the input data is also removed.
- t.exec($graph$, $destructive$) is a shorthand for the composition of apply_on($graph$) followed by eject($destructive$).

The following section showcases how the user can interact with our library to build meaningful transformations.

4 DEMONSTRATION SCENARIOS

In order to demonstrate the novelty, efficiency, and applicability of this approach for transforming property graphs, our demonstration scenario will use the following datasets:

- the Movies[2] dataset from Neo4j which contains a small graph of movies and people related to those movies as actors, directors, producers, etc. It is small and simple enough to showcase the capabilities of the framework in a demonstration setting;
- two real data exchange scenarios adapted to the data model of property graphs: Amalgam1ToAmalgam3 which describes metadata about bibliographical resources and GUSToBioSQL which maps fragments of the Genomics Unified Schema (GUS) to the generic Biological Schema (BioSQL), that have been provided as part of the iBench benchmark suite [1].
- the *Offshore Leaks Database and guide from the International Consortium of Investigative Journalists (ICIJ)*, a popular real-world property graph[3] with 1,908,466 nodes, 3,193,390 edges.

The users will also be free to write their own queries, helping them get familiar with DTGraph's features and limitations.

Movies dataset. First, we will start with an introductory transformation scenario using the Movies dataset (171 nodes and 253 edges). In this dataset, people are connected with movies through relationships whose type indicates their role in the movie, e.g., actor, director, etc. This is inadequate if we want to efficiently filter the people that have co-directed a movie, as such a query would greatly benefit from a label containing all people who directed at least one movie, thus avoiding an expensive scan of all person nodes and their outgoing relationships. The following transformation can be used to solve the above problem. It generates a new property graph disconnected from the input one, containing the refactoring specified with the three declarative transformation rules below. The steps correspond to those shown in Figure 2.

Step (i). The user first applies the transformation t containing the following two rules on our current Movies database:

```
1  MATCH (n:Person)-[:ACTED_IN]->(:Movie)
2  GENERATE (x = (n):Actor { name = n.name, born = n.born })
```

```
1  MATCH (n:Person)-[:DIRECTED]->(:Movie)
2  GENERATE (x=(n):Director { name = n.name, born = n.born })
```

At this point, the transformation is executed on the database and its output can be visualized with tools such as Neo4j browser or Memgraph lab. It is important to understand that all these rules *collaborate* together and that the output of the transformation *consolidates* the output of all rules. For instance, because the same argument list (n) is provided in both rules, we create only one node (with the two appropriate labels) for a person which is both an actor and a director in some movies.

Step (ii). The user now add rules to the currently executed transformation to incorporate some relationships in the output graph between these newly created nodes, one of which could be:

```
1  MATCH (n:Person)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(o:Person)
2  GENERATE
3      (x = (n):)-[():COLLEAGUE { movie = m.title }]->(y = (o):)
```

Note that endpoints will be recognized by the system as matching previously created nodes because of their identical id field, so there is no need to specify their contents again.

Step (iii). Dealing with conflicts is an important aspect of property graph transformations. We recall that a conflict occurs when

[2] https://neo4j.com/docs/getting-started/appendix/example-data/

[3] https://github.com/ICIJ/offshoreleaks-data-packages

(i) Writing the transformation

```
generate_actors = Rule('''
MATCH (n:Person)-[:ACTED_IN]->(:Movie)
GENERATE (x = (n):Actor { name = n.name, born = n.born }) ''')
generate_directors = Rule('''
MATCH (n:Person)-[:DIRECTED]->(:Movie)
GENERATE (x=(n):Director { name = n.name, born = n.born}) ''')
my_transform = Transformation([generate_actors, generate_directors])
my_transform.apply_on(graph)
```

```
Index: Added 1 index, completed after 2 ms.
Rule: Added 204 labels, created 102 nodes, completed after 147 ms.
Rule: Added 51 labels, created 23 nodes, completed after 72 ms.
```

(ii) Adding a new rule to an existing transformation

```
generate_colleague = Rule('''
MATCH (n:Person)-[:DIRECTED]->(m:Movie)<-[:DIRECTED]-(o:Person)
GENERATE (x = (n):)-[():COLLEAGUE { movie = m.title }]->(y = (o):) ''')
my_transform.add(generate_colleague)
```

```
Rule: Added 0 labels, created 6 relationships, completed after 114 ms.
```

(iii) Dealing with conflicts

```
my_transform.diagnose()
```

```
NodeConflicts: There is currently no node in the database which have a conflict.
EdgeConflicts: There is currently 1 edge in the database which have a conflict.
The edge (:Director {born: 1965, name: Lana Wachowski})
  -[:COLLEAGUE {}]->
(:Director {born: 1967, name: Lilly Wachowski})
  has a conflict on attributes ['movie']
```

```
my_transform.abort()
gen_coll_fixed = Rule('''
MATCH (n:Person)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(o:Person)
GENERATE (x = (n):)-[(m):COLLEAGUE { movie = m.title }]->(y = (o):) ''')
transform_fixed = Transformation([gen_actors, gen_directors, gen_coll_fixed])
transform_fixed.apply_on(graph)
```

```
Index: Removed 1 index, completed after 2 ms.
Abort: Deleted 125 nodes, deleted 6 relationships, completed after 10 ms.
Index: Added 1 index, completed after 1 ms.
Rule: Added 204 labels, created 102 nodes, set 446 properties, compl. after 109 ms
Rule: Added 51 labels, created 23 nodes, set 111 properties, compl. after 59 ms.
Rule: Added 0 labels, created 768 relationships, completed after 114 ms.
```

```
my_transform_fixed.diagnose()
```

```
NodeConflicts: There are currently 0 nodes in the database which have a conflict.
EdgeConflicts: There are currently 0 edges in the database which have a conflict.
```

(iv) Validating the transformation

```
my_transform_fixed.eject()
```

```
Index: Removed 1 index, completed after 1 ms.
Eject: Removed 125 labels, erased 893 properties, completed after 232 ms.
```

Figure 2: Transformation management in DTGraph.

two rules specify different values for a property of an output element. Indeed, in the property graph data model, each attribute must be single-valued. In our scenario, Lana and Lilly Wachowski have produced together many movies, hence the only COLLEAGUE relationship between them would store more than one title on the *movie* attribute. To solve this issue, the user replaces the last rule with one in which the identifier list of the COLLEAGUE relationship is (m), to specify that there should be as many relationships between two people as there are movies in which they both starred. Users can now see through their favorite graph visualization tool that there are several relationships between the two producers, and DTGraph indicates that there is no longer a conflict in the output.

Step (iv). The transformation obtained so far meets the needs of the user and can therefore be validated. DTGraph offers an option to remove all internal bookkeeping data used by the system for handling the transformations.

Figure 2 shows all the metadata returned by the system during the development of a transformation (Steps 1–4): execution time, number of elements built, elements in conflicts, creation and destruction of internal bookkeeping data such as indexes, etc.

Scalability assessment. Then, we will move to the realistic data exchange and data integration scenarios (i.e., Amalgam1ToAmalgam3 and GUSToBioSQL) to demonstrate the scalability of our system. These scenarios consist of two relational schemas and a mapping between them expressed as a set of SO-tgds. We transform the input instance, a rudimentary property graph obtained after importing the input relational data using a generic ingestion method, into a full-fledged property graph following the output schema and modeling join tables as relationships.

We will showcase the scalability of our system using synthetic data generated with the iBench tool [1], generating arbitrarily large input instances. The users will have access to plots and charts demonstrating the scalability of the system and its other desirable features (e.g., the order in which the rules are applied does not have an impact on the time to construct the output). Users will also be able to interactively: load reasonably sized input data, examine the transformation rules implementing the required mapping, modify them if desired, execute them, and visualize the output produced.

Offshore Leaks dataset. We end our demonstration scenario by showcasing a comprehensive refactoring on the last dataset, using approximately 20 rules. This illustrates how our system can be used for deep refactoring of the data, still maintaining practical efficiency on a large scale. The user will have access to a notebook providing the motivation behind each rule, and how they are tied together.

ACKNOWLEDGMENTS

Filip Murlak was supported by Poland's NCN grant 2018/30/E/ST6/-00042 and the other authors by ANR-21-CE48-0015 VeriGraph.

REFERENCES

[1] Patricia C. Arocena, Boris Glavic, Radu Ciucanu, and Renée J. Miller. 2015. The IBench Integration Metadata Generator. *VLDB* 9, 3 (2015), 108–119.

[2] Patricia C. Arocena, Boris Glavic, and Renee J. Miller. 2013. Value Invention in Data Exchange. In *SIGMOD*. 157–168.

[3] Philip A. Bernstein and Sergey Melnik. 2007. Model Management 2.0: Manipulating Richer Mappings. In *SIGMOD*. 1–12.

[4] Iovka Boneva, Benoît Groz, Jan Hidders, Filip Murlak, and Slawek Staworko. 2023. Static Analysis of Graph Database Transformations. In *PODS*. 251–261.

[5] Angela Bonifati, Filip Murlak, and Yann Ramusat. 2024. Transforming Property Graphs. arXiv:2406.13062 [cs.DB] https://arxiv.org/abs/2406.13062

[6] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023. GPC: A Pattern Calculus for Property Graphs. In *PODS*. 241–250.

[7] 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*. 1433–1445.

[8] Memgraph. 2023. Memgraph Advanced Graph Extensions. Retrieved February 13, 2024 from https://github.com/memgraph/mage

[9] Neo4j. 2023. APOC user guide for Neo4j 5. Retrieved November 9, 2023 from https://neo4j.com/docs/apoc/current/

[10] Neo4j. 2024. Cypher Query Language Reference, V9. Retrieved February 27, 2024 from https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf

[11] Philipp Skavantzos and Sebastian Link. 2023. Normalizing Property Graphs. *Proc. VLDB Endow.* 16, 11 (jul 2023), 3031–3043.

[12] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A Property Graph Query Language. In *GRADES*. 1–6.