P2KG: Declarative Construction and Quality Evaluation of Knowledge Graph from Polystores

Xiuwen Zheng(✉), Subhasis Dasgupta, and Amarnath Gupta

University of California San Diego, La Jolla, USA
{xiz675,sudasgupta,a1gupta}@ucsd.edu

Abstract. Constructing knowledge graphs from heterogeneous data sources and evaluating their quality and consistency are important research questions in the field of knowledge graph. We propose mapping rules to guide users to translate data from relational and graph sources into a meaningful knowledge graph, and design a user-friendly language to specify the mapping rules. Given the mapping rules and constraints on source data, equivalent constraints on the target graph can be inferred, which is referred as data source constraints. Besides this type of constraints, we design other two types: user-specified constraints and general rules that a high-quality knowledge graph should adhere to. We translate the three types of constraints into uniform expressions in the form of graph functional dependencies and extended graph dependencies, which can be used for consistency checking. Our approach provides a systematic way to build and evaluate knowledge graphs from diverse data sources.

Keywords: Knowledge graph construction · Knowledge graph evaluation · Graph functional dependency

1 Introduction

Knowledge graphs (KGs) are increasingly used as complex data products derived by integrating information from multiple sources [3]. Informally, an entity-centric knowledge graph [5,10] is a graph whose nodes represent real-world entities together with their properties, and edges (predicates) represent relationships between pairs of these entities. The edges may also have their own properties.

The broad topic of this paper is the issue of *quality* in knowledge graphs that are constructed from more than one data source. We specifically focus on a situation where a KG is constructed from independent, heterogeneous sources like relational databases and ontological graphs. An important metric of knowledge graph quality is ***consistency***, the property that asserts that the KG does not have contradictions. In other words, if we assume that the data sources from which the KG is constructed are already consistent (and accurate), the construction process of the KG should not introduce any inconsistency in the KG. In addition to consistency, we would like to ensure that the construction algorithm maintains a set of structural properties of the target KG. For instance, the KG should have no isolated nodes.

A. Abelló et al. (Eds.): ADBIS 2023, CCIS 1850, pp. 427–439, 2023.
https://doi.org/10.1007/978-3-031-42941-5_37

To achieve this goal, we adapt schema mapping techniques [1,2,9] that have been extensively used in the information integration literature. However, we note that a fundamental difference between information integration and KG-construction, is that in the former, the source and the target schema both exist and the role of the schema mapping is to establish the correspondence across the source and target schemas, whereas in our case, an a priori target schema does not exist. We show that with a slight abuse of intent, graph functional dependencies (GFDs) can be effectively used to express KG construction constraints even when the data sources have heterogeneous data models.

This paper makes the following contributions toward knowledge graph construction and quality evaluation,

- It defines a new generic language to specify mapping rules between data sources and the target knowledge graph. It is easily adaptable to different data sources.
- We adapt the theory of graph functional dependencies (GFD) [7] to generate equivalent GFDs on target KG from original constraints on data sources given the mapping rules users applied to construct the KG.
- We extend GFDs to graph dependency (GD) expressions to specify user-defined constraints and general-rule constraints.
- We translate these different constraints to uniform Graph Functional Dependencies and our extended graph dependencies for ease of evaluation.

2 Preliminaries

Definition 1 (Knowledge Graph). *A knowledge graph G is a tuple $(V, E, L, R, A_V, A_E, U_V, U_E, \lambda, \mu)$ where*

- *V is a set of vertices such that designate entites;*
- *E is the set of relationships between entity pairs;*
- *L is the set of entity categories such that $\lambda : L \mapsto V$ is a labeling function that assigns a category to every entity $v \in V$;*
- *R is the set of relationship categories such that $\mu : R \mapsto E$ is a labeling function that assigns a category to every entity $e \in E$;*
- *A_V (resp. A_E) is the set of node attributes (resp. edge attributes) such $u_a^i \in U_A$ (resp. U_E) is the domain of attribute $a^i \in A_V$, U_E is similarly defined.*

Definition 2 (Ontological Knowledge Graph). *An ontological KG is a KG such that the entity categories L map to the concepts $\mathcal{C}$ and the relationship categories R belong to the relationships $\mathcal{R}$ of an existing ontology $\mathcal{O}$.*

Thus, if v is a vertex in the knowledge graph and l is its label, the l must correspond to some concept c in ontology $\mathcal{O}$. Similarly, if $r(v_1, v_2)$ is an edge in the knowledge graph then its predicate name r will correspond to the object properties defined in $\mathcal{O}$. In reality however, it is not always feasible that all KG concepts and relationships can be mapped to the ontology. Hence we consider that the mappings of $L \mapsto \mathcal{C}$ and $R \mapsto \mathcal{R}$ to the ontology are **partial**.

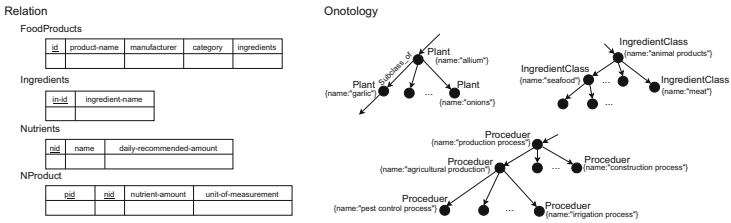


Fig. 1. Source data for running example.

Ontology. One of the data sources used in our knowledge graph is an ontology. For this paper, we assume that the ontology is expressed as a DL-Lite (specifically, $DL-Lite_{\mathcal{A},id}$) [4] that corresponds to OWL 2 QL, a tractable profile of the OWL ontology. DL-Lite is less expressive than other ontologies like OWL-Full (e.g., it cannot express subproperty relationships). However, as [4] elegantly elaborates, ontological expressions in DL-Lite cleanly translate to relational queries. In our setting, we translate a DL-Lite compatible OWL ontology into a faithfully encoded property graph. The translation process is out of scope for this paper, but see [2] for a comparable approach. However, we will present examples throughout this paper to illustrate the graph representation of the $DL-Lite$ ontology.

Graph Functional Dependency. Graph functional dependency (GFD) [7,8] is a concept in database theory that extends the idea of functional dependency from traditional relational databases to graph databases. A GFD is a constraint that describes a relationship between properties of nodes. GFDs are used to ensure data integrity in graph databases and to help ensure that queries can be executed efficiently.

A GFD is formally defined as a pair $(Q[\bar{x}], X \rightarrow Y)$ where Q defines a graph pattern; X and Y are two sets of literals of $\bar{x}$. The literal can be a constant literal, e.g., $x.A = c$, or variable literal, e.g., $x.A = y.A$ where $x, y \in \bar{x}$.

3 Running Example

We use a food knowledge graph as our running example (Fig. 1). We consider a relational data source having tables modeled after the 2022 USDA Database on Branded Food Products[1], and an ontology graph by extending the Food Ontology (FOODON) [6]. Our target KG incorporates information regarding the products, their ingredients, and their nutrient content from the relational source, as well as relevant ontology information concerning the product ingredients and procedures for producing the products.

The relational source (Fig. 1) has four tables. The `FoodProducts` table keeps the information for food products where `id` is the primary key and `[ingredients]` is a list of ingredient names. The `Ingredients` table has ingredients information with their ids and names. There is a domain constraint for

[1] https://fdc.nal.usda.gov/.

the attribute [ingredients] of FoodProducts table and ingredient-name attribute of Ingredients table: the domain for any item in the union list of FoodProducts.ingredients must belong to Ingredients.ingredient-name, which can be stated as:

$$\forall p \in FoodProducts, dom(unnest(p.ingredients)) \subset dom(ingredients.ingredient\text{-}name) \quad (1)$$

The Nutrients table keeps the information of nutrients where nid is the primary key. The NProduct table keeps the mapping information between products and nutrients which satisfies the PK-FK constraints.

```
NProduct.pid = FoodProducts.id
NProduct.nid = Nutrients.nid.
```

The ontology graph models food production procedures, ingredients class and plants. There are three node labels Procedure, IngredientClass and Plant and a node property name and transitive edge Subclass_of. It models information like "agricultural production" is a subclass of "production process". Since the ontology graph may contain information not relevant to food products(e.g., "pest control process"), only a subset of the ontology dataset should be preserved in the target knowledge graph.

4 The P2KG Mapping Language and Mapping Rules

The P2KG mapping language specifies how elements of the knowledge graph are defined as views over one or more data sources. A mapping statement has the following structure.

```
for row/node/edge/path <vars-1> in <source-query-1>
    for row/node/edge/path <vars-2> in <source-query-2> ...
      <knowledge-graph-construction-statement>
```

The mapping statement we employ uses multiple for constructs that may be nested, followed by a <knowledge-graph-construction-statement>. The syntax of the <knowledge-graph-construction-statement> is inspired by Cypher, which is a standard language used for querying graph databases. To construct nodes or edges, we use a subset of Cypher language that includes the Create, Merge, Where, and Set clauses, among others.

In the following section, we present a series of examples that demonstrate how P2KG mapping rules.

4.1 Mapping from Relations

There are several ways to map relations to a knowledge graph, and the choice of mapping rules depends on the intended purpose of the knowledge graph. In this

section, we present different mapping rules, and then demonstrate how they can be used to create different knowledge graphs using the running dataset.

Rule 1: Mapping Table Columns to KG Nodes. One way to create nodes in the knowledge graph is by using columns from a table. The columns can be used to create nodes with a specific label in the KG, and the columns are mapped to the node's properties. For instance, if a user is interested in the manufacturer of the products, they can create nodes with the label Manufacturer" using the following statement:

```
for row r in (select manufacturer from FoodProducts)
    Merge (n:Manufacturer {name:r.manufacturer})
```

The Manufacturer nodes have a `name` property that is derived from the `manufacturer` column in the `FoodProducts` relation. The Merge clause is used to ensure that the nodes created are distinct, i.e., there are no two Manufacturer nodes with the same name. Nodes can also be created with multiple columns as properties. For example, ProductName" nodes can be created from both the `product-name` and `manufacturer` columns using the following statement:

```
for row r in (select product-name, manufacturer from FoodProducts)
    Merge (n:ProductName {name:r.product-name, manufacturer:r.manufacturer})
```

Rule 2: Mapping Table Rows to KG Edges. The rows in the table can be mapped to edges in the KG as both relational tuples and node edges depict relationships. Each row will be mapped to an edge in the KG. For instance, one can create Product nodes using certain columns, such as "id," as attributes and create Manufacturer nodes using rule 1. Then, the Product nodes can be linked to the Manufacturer nodes using the statement:

```
for row r in (select id, manufacturer from FoodProducts)
    MATCH (p:Product {id: r.id})
    MATCH (m:Manufacturer {name: r.manufacturer})
```

As the Product and Manufacturer nodes are pre-built, the Match clause is used to match the corresponding nodes with the given property values. If the edge does not exist before, it creates an edge between the two matched nodes.

Different mapping rules can be applied based on the user's requirements. Figure 2 shows various mappings from the same data. Different mapping rules can generate KGs with different space costs and different source constraints, which will be explained in detail in Sect. 5.

In the first example, Product, ProductName, Manufacturer and Category nodes are created using columns `id`, `productname`, `manufacturer` and `category` respectively using rule 1. Edges between nodes are created using rule 2. The total number of nodes will be $disc(id)+disc(product-name)+disc(manufacturer)+disc(category)$ where $disc(\cdot)$ is the distinct count of values in the column $\cdot$. It

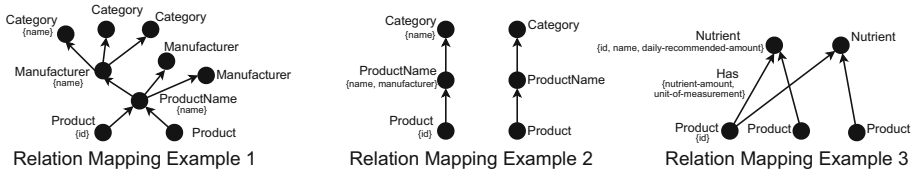


Fig. 2. Different mapping examples for the relational source in the running example.

saves space compared to other mapping rules, however, it loses some source constraints which will be introduced in Sect. 5. A Product node can only link to one node with a certain label, for example, it can only connect to one ProductName (or Manufacturer) node because `id`is the primary key of the relation. For the other relationships, they are m to n mappings. For example, different Product nodes can connect to the same ProductName node, and different ProductName nodes can connect to the same Manufacturer node, and the same ProductName may connect to different Manufacturer nodes. There are other ways to do the mapping. As example 2 shows, the columns `product-name` and `manufacturer` are used together to create the ProductName nodes. Since `product-name` and `manufacturer` columns can determine a product, ProductName nodes have a one-to-one mapping relationship with other types of nodes.

Rule 3: Mapping Multiple Table Columns to KG Nodes. There may be multiple columns, which could be from different tables, that refer to the same entities and can be mapped to the same nodes. Such columns will be used together to create nodes with the same label, while nodes from different columns can have their own associated labels. For instance, consider the `ingredients` column in the `FoodProdcuts` table and the `ingredient-name` column in the `Ingredients` table; these two columns can be mapped to the same nodes. The two columns are used together to create the `AllIngredient` nodes, and the ingredients from the `FoodProdcuts` table have another label `ProductIngredient`, and those from the `Ingredients` table have an `Ingredient` label.

```
for row r in (select unnest(ingredients) as in from FoodProdcuts)
    MERGE (i:AllIngredient{name:r.in}) SET i:ProductIngredient
for row r in (select ingredient-name as in from Ingredients)
    MERGE (i:AllIngredient{name:r.in}) SET i:Ingredient          (1)
```

In this statement, the two `for` loops are used to traverse both tables and create nodes for the ingredients. The first loop handles the `ingredients` column in the `FoodProdcuts` table and creates nodes with the label `ProductIngredient`. The second loop handles the `ingredient-name` column in the `Ingredients` table and creates nodes with the label `Ingredient`. Both loops create nodes with the label `AllIngredient` by using the `MERGE` clause to either match an existing node with the same name or create a new one with the given name. The `SET` clause sets the appropriate label for each node created by the two loops.

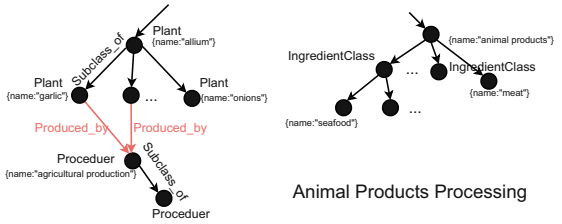


Fig. 3. Mapping example for the graph source in the running example.

Mapping From Query Results. More complex SQL queries including joins between tables can be applied before applying any mapping rules introduced above. The SQL queries is inside the <source-query> syntax, then any mapping rule introduced before can be applied on the query result. For example, in Fig. 2 Example 3, the `Nutrient` and `Product` nodes come from information in the `Nutrient` and `FoodProducts` tables respectively, and the edges between them are mapped from the `NProduct` table. To create the KG, one can write a query to join the three tables and then apply the mapping rules:

```
for row r in (select p.id as pid, n.nid as nid, name as name,
daily-recommended-amount, nutrient-amount, unit-of-measurement
from Nutrient n, FoodProdcuts p, NProduct np
where n.nid = np.nid, p.id = np.pid)
    MERGE (p:Product{id:r.pid})
    MERGE (n:Nutrient{id:r.nid, name:r.name, daily-recommended-amount:
   r.daily-recommended-amount})
    MERGE (p) - [:Has{nutrient-amount:r.nutrient-amount,
    unit-of-measurement:r.unit-of-measurement}] -> (n)
```

4.2 Mapping from Graph Source

When mapping from the graph source to the KG, the <source-query>, which can also be written in Cypher, is used to match the part of the graph data that the user wants to keep in the target KG. Users can also specify some customized rules using Cypher SET clauses to create new edges. Figure 3 shows an example of mapping the ontology graph to the target KG for the running dataset. As stated in the dataset section, the production procedures ontology graph is large and contains unrelated information about food production, and users are only interested in the relevant information about "agricultural production" or "animal products processing." This can be achieved by the following statement:

```
for edge(n, r, m) in (MATCH (a:Procedure WHERE a.name in ['agricultural
production', 'animal products processing'])-[:Subclass_of*]->(n),
(n)-[r:subClassOf]-(m) return (n, r, m))
    MERGE (n)-[r]-(m)
```

Besides, each Plant node should be connected to the agricultural production Procedure node, which can be achieved by the following statement:

```
for node n in (MATCH (n:Plant) return n)
    MATCH (m:Procedure{name:'agricultural production'})
    MERGE (n)-[:Produced_by]->m
```

4.3 Mapping from Multiple Sources

Data from different sources can be linked based on specific rules. For example, in the running example (as shown in Fig. 4), a product from the FoodProducts table can be linked to a Plant node created from the ontology graph if the product contains an ingredient from the plant. Similarly, an Ingredient node from the relations can be linked to an IngredientClass node from the ingredient class ontology using the `subclass_of` edge, if that ingredient is a sub-class of the ingredient class. To create the KG, mapping rules are applied to each single source separately. Then, the cross-source relationships are established by executing a Cypher query in the target KG as the <source-query>,

```
for nodes (n, m) in KG:(MATCH (n:Product)->(x:Ingredient), (m:Plant)
where x.name=m.name return n, m)
    MERGE n-[:Derived_from]->m
for nodes (n, m) in KG:(MATCH (n:Ingredient), (x:IngredientClass)
-[:Subclass_of]->(m:IngredientClass) return n, m)
    MERGE n-[:Subclass_of]->m
```

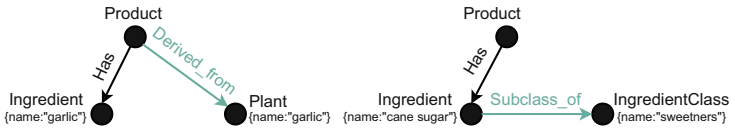


Fig. 4. Mapping examples for multiple source in the running example.

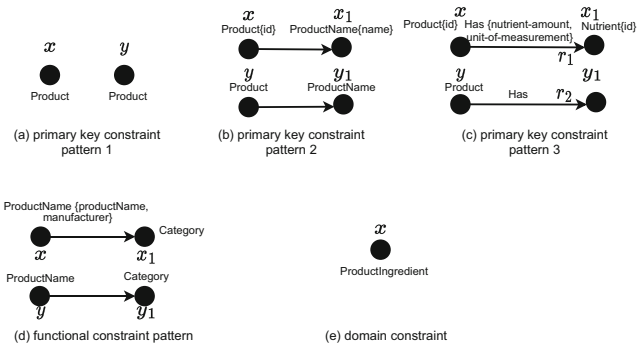


Fig. 5. Source constraints example.

5 Constraints

Consistency checking is important to ensure that a knowledge graph is reliable and accurate. To achieve this, we define three types of constraints. The first type is **source constraints**, which refer to the constraints specified by the data sources used to build the knowledge graph. The target KG must comply with these constraints to ensure consistency. The second type is **general rules**, rules that a good knowledge graph should follow. These rules ensure that the knowledge graph is structured in a coherent and meaningful way. The third type is **user-specified constraints**, which are constraints that are specified by the user to ensure that the knowledge graph meets their specific requirements. To check if a knowledge graph complies with these constraints, we translate these different constraints to our extended graph dependencies which will be used for consistency checking. By checking if the knowledge graph meets these constraints, we can evaluate if it is consistent and of good quality.

5.1 Source Constraints

Constraints from the data sources can be kept in the target KG based on what mapping rules are applied. We mainly consider relational constraints and show how to translate them on the source relation to the graph functional dependency on the KG given the mapping rules applied.

Primary Key Constraint. In relational databases, a primary key constraint is a constraint that ensures that each row in a table has a unique identifier or key value. Figure 5(a)-(c) illustrates the primary key constraint on KG.

For the `FoodProducts` table, the column `id` is a primary key. In Fig. 5(a), the Product nodes are created from the columns from this table including `id` column, thus we have the following graph functional dependency

$$Q_a[x, y], X_1 \rightarrow Y_1$$

where $Q_a[x, y]$ is the structure shown in Fig. 5(a), X_1 is $x.id = y.id$ and Y_1 is $x.A = y.A$ where A is any node property of Product. In Fig. 5(b), Product

nodes are created from `id` column and the `product-name` column is used to create ProductName nodes and the edges are created between them, then in the KG, we have the GFD as

$$Q_b[x, y], X_2 \rightarrow Y_2$$

where $Q_b[x, y]$ is the structure shown in Fig. 5(b), X_2 is $x.id = y.id$ and Y_2 is $x_1.name = y_1.name$. The primary key can be multiple columns, for example, in the `NProduct` table, the `pid, nid` together serves as primary key, in Fig. 5(c), Product nodes and Nutrient nodes are created from these two columns respectively and edges created between them by applying mapping rule 2 and the other two columns serve as edge properties. For this KG, we have the following GFD to express the primary key constraint:

$$Q_c[x, x_1, y, y_1, r, r_1], X_3 \rightarrow Y_3$$

where $Q_c[x, x_1, y, y_1, r, r_1]$ is the structure in (c), X_3 is $x.id = y.id, x_1.id = y_1.id$ and Y_3 is $r.nutrient\text{-}amount = r_1.nutrient\text{-}amount, r.unit\text{-}of\text{-}measurement = r_1.unit\text{-}of\text{-}measurement$.

Functional Dependencies. A functional dependency (FD) on relations states that the value of a set of attributes determines the value of another set of attributes. A FD of relation with schema $R(U)$ is an expression of the form $R : X \rightarrow Y$ where $X \subseteq U$ and $Y \subseteq U$.

For example, in the `FoodProducts` table, the `productname` together with `manufacturer` columns determine the category of the product. In Fig. 5 (d), the Product nodes are created from `productname` and `manufacturer`, and they are connected to the Category nodes. We have the following GFD equivalent to the relational FD:

$$Q_d[x, x_1, y, y_1], X_4 \rightarrow Y_4$$

where Q_d is the topological structure in (d), X_4 is $x.productName = y.productName$, $x.manufaturer = y.manufacturer$ and Y_4 is $x_1.category = y1.category$.

Inclusion Dependencies. An inclusion dependency (IND) on pairs of relations of schemas $R(U)$ and $S(V)$ (with R and S not necessarily distinct) is an expression of the form $R[X] \subseteq S[Y]$ where $X \subseteq U$ and $Y \subseteq V$. There is an inclusion dependency between `FoodProducts` and `Ingredients` table as stated in 1. Suppose that these two columns are mapped to the target KG using mapping statement (1), then we have the following GFD: $Q_e[x], X_5 \rightarrow Y_5$ where $Q_e[x]$ is shown in Figure (e) which matches any node with $ProductIngredient$ label, X_5 is $\emptyset$ and Y_5 is $x.label = Ingredient$.

5.2 General Rule Constrains

To ensure a high-quality knowledge graph, there are several general constraints that should be satisfied. These constraints are illustrated in Fig. 6.

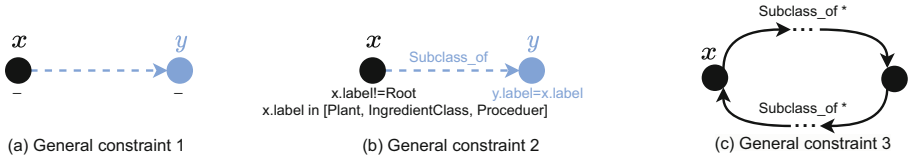
– The graph should not contain any isolated nodes, as shown in Fig. 6 (a).

- All nodes created from the ontology data source, except for the root nodes, should have a subClassOf parent, as shown in Fig. 6 (b).
- The graph should display edge-label acyclicity i.e., if you just consider a single edge label, the graph will be acyclic (symmetric edge labels are implicit), as shown in Fig. 6 (c).

To provide a uniform way to express different types of constraints, we propose an extension to the graph functional dependency called graph dependency (GD). In this extension, we support node/edge existence statements, allowing us to express constraints on the existence of nodes and edges in the knowledge graph. As stated in Sect. 2, in GFD, X and Y are two sets of literals of $\bar{x}$, and a literal of $\bar{x}$ can either be a constant literal or variable literal. We extended the form of X and Y. For X, it is extended to support the existence statement which says that there exists a node/edge in the knowledge graph with certain properties value or labels which can be constant or related to $\bar{x}$. For example, X can be $\exists node\ n \in KG, n.label = L, n.A = c, n.A' = x.A'$ where $x \in \bar{x}$. For Y, it is extended to support connection between nodes defined in X and nodes in $\bar{x}$, for example, y can be $n \rightarrow x$ which states that there is an edge from n (node defined in X) to x. With extended GDs, the general constraints can be expressed as follows.

- $Q_a[x], X_1 \rightarrow Y_1$, X_1 is $\exists node\ y \in KG$, Y_1 is `x -> y`.
- $Q_b[x], X_2 \rightarrow Y_2$, X_2 is $\exists node\ y \in KG, y.label = x.label$, Y_2 is `x-[:Subclass_of]->y`.
- $Q_c[x], X_3 \rightarrow Y_3\ X_3 = \emptyset, Y_3 = false$.

where Q_a, Q_b, Q_c match the topological structure depicted in black color in Fig. 6 (a) - (c) respectively.



(a) General constraint 1 (b) General constraint 2 (c) General constraint 3

Fig. 6. General constraints example.

5.3 Constraints from Users

Users can specify constraints on the knowledge graph directly using graph dependency expressions to check if the created graph satisfy their specific requirements. We show some examples in Fig. 7, and explain them as follows:

- Any Plant node should be connected to the Procedure node with name "Agricultural Process".

- A Product node which has "garlic" as ingredient should be connected with the Plant node whose name is garlic by the `Derived_from` edge.
- If a product has ingredient which is a subclass of meat or seafood, then the product should be non-vegetarian.

Users can specify these constraints in the extended GD expressions as follows:

- $Q_a[x, y], X_1 \rightarrow Y_1$, X_1 is $\exists edge\ r \in KG, r.label = Produced\_by$, Y_1 is `(x)-[r]->(y)`;
- $Q_b[x, y], X_2 \rightarrow Y_2$, X_2 is $\exists edge\ r \in KG, r.label = Derived\_from$, Y_2 is `(x)-[r]->(y)`;
- $Q_c[x], X_3 \rightarrow Y_3$, $X_3 = \emptyset$, Y_3 is `x.type='non-vegetarian'`.

where Q_a, Q_b, Q_c match the topological structure depicted in black color in Fig. 7 (a) - (c) respectively.

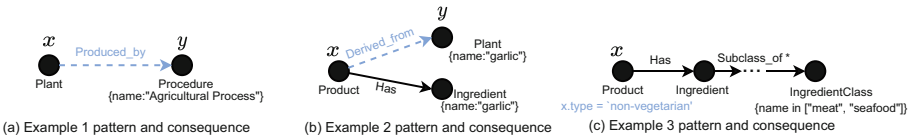


Fig. 7. User-specified constraints example.

5.4 Evaluation of Constraints

There is prior work such as [8] which proposes algorithm to evaluate GFDs on property graph, and the prior algorithm can be directly introduced to evaluate the GFDs derived from the source constraints. However, we extended the original GFDs to graph dependencies (GD) to support more user-specified constraints and general rule constraints, and a new algorithm should be designed to evaluate the extended GDs on property graph which we leave as future work.

6 Conclusion and Future Work

In this paper, we have presented mapping rules for efficiently mapping data from different sources, namely relational and graph data, to a target knowledge graph. We have also defined three types of constraints, derived from source data, user specifications, and common rules of a good knowledge graph, and translated them into unified expressions in the form of GFDs and extended GDs to evaluate the quality, correctness, and consistency of an existing knowledge graph.

Future work includes developing an algorithm to efficiently evaluate the GDs in large-scale knowledge graphs and automating the process of generating equivalent GFDs based on the constraints on the data sources and mapping rules used to create the KG. These contributions will facilitate the development and maintenance of high-quality knowledge graphs.

References

1. Alexe, B., Hernández, M., Popa, L., Tan, W.C.: Mapmerge: correlating independent schema mappings. VLDB J. **21**, 191–211 (2012)
2. Angles, R., Thakkar, H., Tomaszuk, D.: Mapping RDF databases to property graph databases. IEEE Access **8**, 86091–86110 (2020)
3. Asprino, L., Daga, E., Gangemi, A., Mulholland, P.: Knowledge graph construction with a façade: a unified method to access heterogeneous data sources on the web. ACM Trans. Internet Technol. **23**(1), 1–31 (2023)
4. Calvanese, D., et al.: Ontologies and databases: the *DL-Lite* approach. In: Tessaris, S., et al. (eds.) Reasoning Web 2009. LNCS, vol. 5689, pp. 255–356. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03754-2_7
5. Dong, X., et al.: Knowledge vault: a web-scale approach to probabilistic knowledge fusion. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 601–610 (2014)
6. Dooley, D.M., et al.: Foodon: a harmonized food ontology to increase global food traceability, quality control and data integration. NPJ Sci. Food **2**(1), 23 (2018)
7. Fan, W., Hu, C., Liu, X., Lu, P.: Discovering graph functional dependencies. ACM Trans. Database Syst. (TODS) **45**(3), 1–42 (2020)
8. Fan, W., Wu, Y., Xu, J.: Functional dependencies for graphs. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1843–1857 (2016)
9. Mazilu, L., Paton, N.W., Fernandes, A.A., Koehler, M.: Dynamap: schema mapping generation in the wild. In: Proceedings of the 31st International Conference on Scientific and Statistical Database Management, pp. 37–48 (2019)
10. Vrandečić, D., Krötzsch, M.: Wikidata: a free collaborative knowledgebase. Commun. ACM **57**(10), 78–85 (2014)