



HAL
open science

Reproducible Mapping of Tabular Data into Semantic Knowledge Graphs with OntoWeaver and BioCypher

Johann Dreo, Claire Laudy, Sebastian Lobentanzer, Marko Baric, Ekaterina Gaydukova, Benno Schwikowski

► **To cite this version:**

Johann Dreo, Claire Laudy, Sebastian Lobentanzer, Marko Baric, Ekaterina Gaydukova, et al.. Reproducible Mapping of Tabular Data into Semantic Knowledge Graphs with OntoWeaver and BioCypher. 2024. hal-04509632

HAL Id: hal-04509632

<https://hal.science/hal-04509632v1>

Preprint submitted on 18 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Reproducible Mapping of Tabular Data into Semantic Knowledge Graphs with OntoWeaver and BioCypher

Johann Dreo¹, Claire Laudy², Sebastian Lobentanzer³, Marko Baric⁴, Ekaterina Gaydukova⁵, Benno Schwikowski⁶

March 18, 2024


Large-scale high-level information fusion and data integration is a pressing need in several scientific domains. Recently, the biomedical community established BioCypher, a tool to help create large Semantic Knowledge Graphs (SKGs) in a simple and reproducible way. In this article, we introduce OntoWeaver, a companion tool to BioCypher that allows to easily extract tabular data into SKGs by using a simple declarative mapping. The use of OntoWeaver and BioCypher is demonstrated in two different use cases: cancer database integration and invasive species monitoring. We believe that OntoWeaver and BioCypher, both free and open-source software, can help several scientific communities working on SKGs and high-level information fusion problems.


Introduction


Among scientific domains, the biomedical field operates on the largest scale [1]. This has led the biomedical community to curate scientific information in publicly available databases, each specialized on its specific subject, making it the ideal use case for information fusion problems at scale.


To use these fragmented resources, researchers need to integrate them into machine-readable databases with standardized representations. This has led the biomedical community to use knowledge graphs and ontologies as primary tools of integration. Recently, however, efforts have been made to scale up the process of producing such semantic knowledge graphs (SKG) with the objective of having a findable, accessible, interoperable, and reusable (FAIR) framework [2] for producing biomedical SKGs tailored to the task at hand.


BioCypher [3] is such a framework that simplifies the automated instantiation of SKGs through a combination of procedural “adapters” (which gather the data to integrate) and declarative “schema configurations” (which map the data types to an ontology). BioCypher allows the assembly of several ontologies into a single one by tailing in any subpart of a hierarchy of types into another one. By abstracting the SKG build process as a combination of modular input adapters, which can be aligned with schemas, BioCypher allows the building of SKGs made up of overlapping primary resources. It also supports several output


¹  johann.dreo@pasteur.fr Computational Systems Biomedicine Lab., Computational Biology Dept., Bioinformatics and Biostatistics Hub., Institut Pasteur, Université Paris Cité, Paris, France

²  claire.laudy@thalesgroup.com Thales, Palaiseau, France

³  sebastian.lobentanzer@gmail.com Institute for Computational Biomedicine, Heidelberg University, Faculty of Medicine and Heidelberg University Hospital, Heidelberg, Germany

⁴  marko.baric@pasteur.fr Computational Systems Biomedicine Lab., Computational Biology Dept., Institut Pasteur, Université Paris Cité, Paris, France

⁵  ekaterina.gaydukova@pasteur.fr Computational Systems Biomedicine Lab., Computational Biology Dept., Institut Pasteur, Université Paris Cité, Paris, France

⁶  benno.schwikowski@pasteur.fr Computational Systems Biomedicine Lab., Computational Biology Dept., Institut Pasteur, Université Paris Cité, Paris, France

The work presented in this paper was partly funded by two European projects. DECIDER is funded by the European Union under Horizon 2020 programme under grant agreement No 965193. ILIAD is funded under Horizon 2020 program under grant agreement No 101037643.

<https://biocypher.org/>

backends, allowing the creation of SKGs in various existing database formats/engines.

With BioCypher, adapters take the form of scripts that gather and prepare data as sets of nodes and edges tagged with their respective types. The task of implementing an adapter thus requires advanced programming skills, which may hinder the adoption of those tools by practitioners. Allowing adapters to be small programs makes sense for tasks where accessing the input data is difficult or where the targeted data structure is complex. However, most of the tasks we encounter in common use cases involve simple tabular data.

We thus introduce OntoWeaver, a software allowing the automated creation of BioCypher adapters by declaring a simple mapping from tabular to graph data. We believe that OntoWeaver and BioCypher can help several scientific communities working on SKGs and high-level information fusion problems.

This article shows the general idea and setup of OntoWeaver in Section , goes into more details of its main features in Section , then introduces two use cases demonstrating its use in Section , before concluding in Section .

<https://github.com/oncodash/ontoweaver>

Method

Note on Vocabularies

Translational research using SKGs is at the confluence of several research domains, each of which has its own vocabulary. Different domains often use similar terminology to describe unrelated concepts. For instance, terms used to describe ontologies differ from those used to describe SKGs or databases. However, in our case, some concepts are compatible, in the sense that we can (and want to) map one to another.

Table 1 describes the vocabularies used in OntoWeaver, and in this article, each row represents terms which we consider to be synonyms in our context.

Section goes into more detail about the synonym keywords used in the OntoWeaver declarations.

Mapping Declaration

The main feature of OntoWeaver is the transformation of tabular data into a graph. In the most simple setting, the input table is a collection of entries, each row representing a subject to be transformed into a node in the output graph. Each item in this row can also be transformed into a node, associated with an edge to the node representing the entry. All created nodes and edges are tagged with a type defined by the meaning of the column, or the meaning of the entry itself. Figure 1 shows such

Ontologies	Knowledge Graphs	Databases
Class	Node type Tag	Data Type
Class name	Label	Data
[Data] Property	Attribute Property Field	
Relation Class Object property Association	Edge type Link type	
Instance	Node	Entry Row
Range (of property)	Source	
Domain (of property)	Target	
Argument (of relation)	Node	

Table 1: Terms used as synonyms in this article and OntoWeaver.

a simple mapping, and section goes into detail about how to configure more complex mappings.

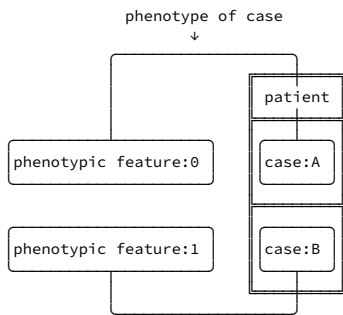


Figure 1: Example of a mapping from a table (drawn with double lines) of two entries represented as two rows and described in one column to an SKG (drawn with single lines) of four nodes and two edges. The rows about patients represent the subject of this SKG and are mapped to nodes of type “case” (synonym for “patient” in the Biolink ontology [4]), while the “patient” column is mapped to nodes of type “case” and linked from “phenotypic feature” node with edge “phenotype of case”.

This type of mapping can be described as a simple declaration, from the name of the column to a type of node and a type of edge. The example given in Figure 1 can thus be described by two declarations: “the subject is a phenotype” and “elements in the column are patients, linked from the subject via a specific association”. This can be described with a markup language, as shown in Figure 2.

```

1 subject: phenotypic feature
2 columns: # Introduces the list of columns.
3   patient: # This is a column name.
4     to_object: case
5     via_relation: phenotype of case
    
```

Figure 2: Example of an OntoWeaver declaration of a mapping written in the YAML markup language, implementing the diagram shown in Figure 1.

Implementation

OntoWeaver is implemented as a Python module. It relies on the Pandas [5] module to load a wide range of table formats.

It provides a simple layer of abstraction on top of BioCypher, which remains responsible for doing the ontology alignment, supporting several graph database backends, and allowing reproducible & configurable builds.

Under the hood, OntoWeaver relies on meta-programming, as it creates Python classes while parsing the mapping configuration. This allows a trained programmer to manipulate and create adapters at a low level, using the Python language itself as a basis. The UML diagram of a simplified view on the main classes of OntoWeaver is shown in Figure 3.

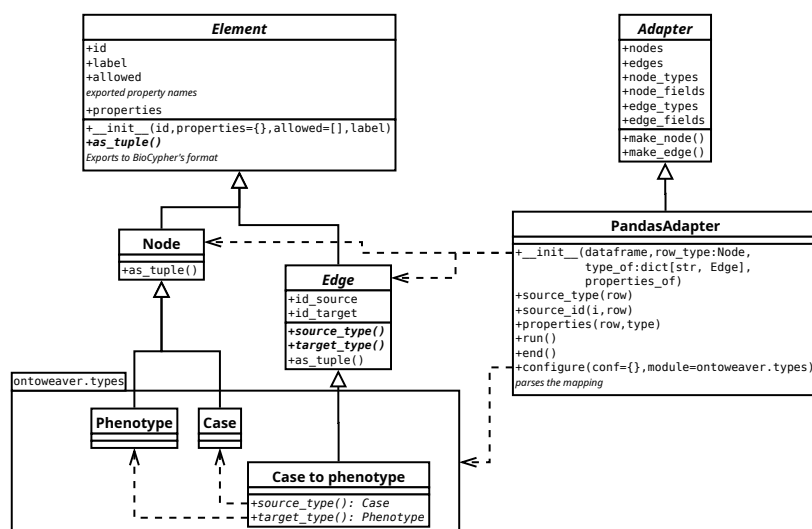


Figure 3: UML diagram showing the main classes of OntoWeaver (simplified for readability). The *Node* and *Edge* classes are the superclass of any node or edge type created on the fly by a *PandasAdapter* while parsing the YAML mapping.

Since OntoWeaver uses the Python type system, it inherits the consistency checks provided by the language. For instance, checking consistency across a type’s hierarchy is straightforward. OntoWeaver uses this ability to allow for the activation or deactivation of subtrees of types at runtime. In this way, the user can configure which part of the input data to extract each time they rebuild their SKG. Additionally, OntoWeaver can validate the source and target types of an edge against their node counterparts.

Features

OntoWeaver facilitates the integration of data into SKGs by targeting the ubiquitous tabular format, bridging the gap between the fields pro-

ducing data and the fields consuming them. Using an existing mapping on new data is straightforward, and creating a new mapping can be done by any practitioner who understands the input data and the targeted ontologies, even with very little knowledge of programming.

OntoWeaver essentially creates a BioCypher adapter from the description of a mapping. As such, its core input is a dictionary that takes the form of a YAML file. This configuration file indicates:

- to which (node) type to map each line of the table,
- to which (node) type to map columns of the table,
- with which (edge) types to map relationships between nodes.

Vocabularies

Because several communities are gathered around SKGs, several terms can be used (more or less) interchangeably (see Section).

OntoWeaver reflects this by providing several alternatives to name the different keywords it uses, thus allowing for the usage of one’s favorite vocabulary to write down the mapping configurations, as shown in Table 2.

Ontologies	Knowledge Graphs	Databases
subject	source	row entry line
from_subject	from_source	FIXME from_node
to_object	to_target to_node	N/A
via_predicate via_relation	via_edge	N/A
to_property	to_property FIXME: to_attribute	
N/A	N/A	columns fields

Table 2: Keywords that can be used by OntoWeaver interchangeably. Each line indicates a feature, each column an alternative (set of) name(s).

The meaning of those keywords is described in the following sections.

Common Mapping

The simplest configuration is to map lines and columns to node types and add an edge from the “line” node to each of the corresponding “column” nodes, as shown in Figure 1. This is achieved by indicating the node type of the *subject* (representing each entry), the node type of

each column (with *to_object*), and the edge type (*via_relation*) linking the two, as shown in Figures 2 and 4.

```

1 source: phenotypic feature
2 fields:
3   patient:
4     to_node: case
5     via_edge: phenotype of case
    
```

Relations Between Columns

Some columns of a table may not hold data about the entry itself but refer to complementary information that is linked to another column. In that case, the related graph should show an edge between the nodes extracted from two columns, and not add an additional edge from the node created for the entry/subject (see Figure 5).

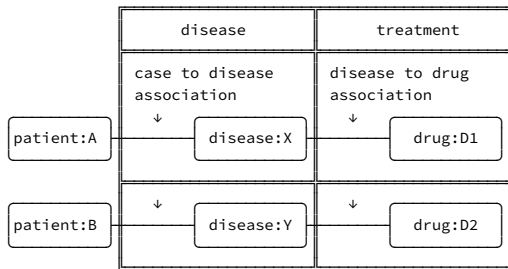


Figure 4: Example of an OntoWeaver declaration of a mapping written in the YAML markup language, implementing the diagram shown in Figure 1 in the same way as Figure 2, but with alternative keywords.

Figure 5: Example of a mapping showing edges added between nodes extracted from two columns. Nodes extracted from the column “disease” are associated with the nodes extracted from the column “treatment” through an edge of type “disease to drug association”.

OntoWeaver provides the keyword *from_node* (or *from_subject*) to indicate this case, as shown in Figure 6.

```

1 subject: patient
2 columns:
3   disease:
4     to_object: disease
5     via_relation: case to disease association
6   treatment:
7     from_subject: disease # Edge from this node type...
8     to_object: treatment # ... to this node type.
9     via_relation: disease to drug association
    
```

Figure 6: Example of an OntoWeaver declaration for linking nodes extracted from two columns, implementing the diagram shown in Figure 5.

Properties

In OntoWeaver, a property is any atomic piece of data that can be attached to the node but that is not a type tag. The keyword *to_property* allows mapping a column to a property of a node type. However, properties can be attached to many node types at once. For instance, properties are the classical way to attach references (e.g. version, original publication, etc.) to nodes and edges. Thus, the keyword *to_property* expects a list of node types after the name of the property (see Figure 7).

```

1 source: patient
2 columns:
3   disease:
4     to_object: disease
5     via_relation: case to disease association
6   sequence_variant:
7     to_object: variant
8     via_relation: patient has variant
9   pubmed_ID:
10    to_property:
11     reference: # Name of the property.
12     - disease # Node types to attach
13     - variant # the property to.

```

Figure 7: Example of an OntoWeaver declaration for a property. Here, the “pubmed_ID” column is mapped as a property attached to both “disease” and “variant” nodes.

Transformers

OntoWeaver allows for the manipulation of the content of a cell before passing it as a set of nodes and edges using *transformers* (also called *generators*).

For instance, it is possible to split the content of a cell into several parts and create a separate node for each part, using the *split* transformer, as shown in Figures 8 and 9.

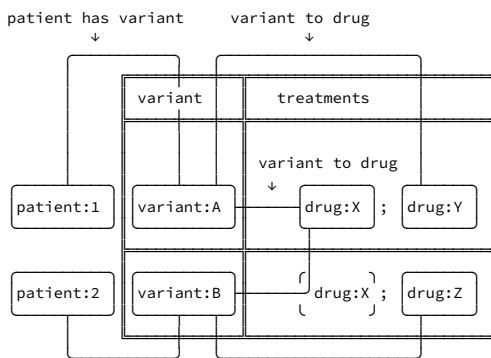


Figure 8: Example of a mapping showing a single column (“treatments”) being split as several (“drug”) nodes.

```

1 subject: patient
2 columns:
3   variant:
4     to_object: variant
5     via_relation: patient has variant
6   treatments:
7     into_transformer:
8       split: # Name of the transformer to use.
9       separator: ";" # Parameter passed.
10      # Apply a generic mapping on each part:
11      from_object: variant
12      to_object: drug
13      via_relation: variant to drug

```

Figure 9: Example of usage of the *split* transformer, implementing the diagram shown in Figure 8.

The implementation of transformers is straightforward and only requires the implementation of a class inherited from *EdgeGenerator*, with two functions that return lists of nodes and edges (see Figure 10). Table data parsing, loading, and import of the parameters passed in the mapping configuration (e.g. *separator* in Figure 9) are handled automatically.


```

1 class split(base.EdgeGenerator):
2     def nodes(self):
3         # 'separator' is added automatically
4         # to the 'self' instance.
5         for i in self.id.split(self.separator):
6             yield self.make_node(id = i)
7
8     def edges(self):
9         for i in self.id_target.split(self.separator):
10            yield self.make_edge(id_target = i)

```

Figure 10: Python code for the *split* transformer. The functions *nodes* and *edges* are called after the table data are parsed and loaded as the following member variables: *id*, *id_source*, *id_target*. The member variable *separator* is available because it has been defined as a parameter in the YAML mapping (see Figure 9).

User-defined Adapters and Dynamic Type Selection

OntoWeaver’s main entry point for the user is an object of the class “PandasAdapter”. But a user may create their own class, for instance, to provide a different way to disable the extraction of some node types at runtime.

Figure 11 shows an example of a new adapter that allows the passing of any node or edge types or property names that the user would like to see enabled. But if the user does not pass anything, then *all* the available types and properties are enabled.

```

1 class MYADAPTER(ontoweaver.tabular.PandasAdapter):
2     def __init__(self,
3                 df : pd.DataFrame,
4                 config : dict,
5                 node_types : Optional[Iterable[Node]] = None,
6                 node_props: Optional[list[str]] = None,
7                 edge_types : Optional[Iterable[Edge]] = None,
8                 edge_props: Optional[list[str]] = None,
9             ):
10        # All created classes go into the 'types' module.
11        from . import types
12        mapping = self.configure(config, types)
13
14        # If "None" is passed (the default),
15        # then do not filter anything
16        # and just extract all available types.
17        if not node_types:
18            node_types = types.all.nodes()
19        if not node_fields:
20            node_fields = types.all.node_fields()
21        if not edge_types:
22            edge_types = types.all.edges()
23        if not edge_fields:
24            edge_fields = types.all.edge_fields()
25
26        super().__init__(
27            df,
28            *mapping,
29            node_types,
30            node_fields,
31            edge_types,
32            edge_fields,
33        )

```

Figure 11: Example of user-defined adapter, which allows to enable a subset or all of node/edge types and properties. The *types.all* module is provided by OntoWeaver as a way to have access to all automatically defined types.

Because OntoWeaver uses Python’s metaprogramming for type declaration, it is also possible to declare user-defined classes that will be reused at the mapping declaration stage. For that purpose, it is sufficient to declare subclasses of *Node* or *Edge*, possibly without additional code, as shown in Figure 12.

```

1  class variant(ontoweaver.Node):
2      @staticmethod
3      def fields(): # = properties.
4          return ["timestamp", "version"]
5
6  # Simple subtypes:
7  class amplification(variant):
8      # Properties are automatically
9      # gathered from parent class.
10     pass
11  class loss(variant):
12     pass
13
14  class MYADAPTER(ontoweaver.tabular.PandasAdapter):
15     # [...]
16     def source_type(self, row):
17         from . import types
18         # Type can vary depending on some column value:
19         if row["alteration"].lower() == "amplification":
20             return types.amplification
21         elif row["alteration"].lower() == "loss":
22             return types.loss
23         else:
24             return types.variant

```

Figure 12: Example of user-defined node types, and of an adapter having a dynamic source node type.

User-defined Functions

As a low-level API, OntoWeaver provides a way to implement complex functions that are called at certain key steps of its processing.

For example, if the user needs to change the type of the subject node depending on the value of a column, they can overload the *source_type* function of their adapter, as shown in Figure 12.

Illustration

To show the ease of use and adaptability to any business domain of OntoWeaver and BioCypher, we illustrate their use in two use cases. The first is a biomedical case that targets the integration of cancer databases, and the second deals with the monitoring of invasive species. Both use cases presented here are simplified for the sake of readability.

Cancer Database Integration

The aim is to extract, from curated biomedical databases, a semantic graph of interaction involving alteration of genes, drugs, and other genomics objects of interest. The domain ontology that we use for this illustration is Biolink [6] which provides an information model of biological entities such as genes, diseases, phenotypes, pathways, individuals, and substances, together with a model of their potential associations. Here we demonstrate the integration of two databases containing this type of information: the Cancer Genome Interpreter (CGI, [7]) and OncoKB [8], [9] databases. The complete application is part of the OncoDashKB project [10]. The data shown in this article is produced by random sampling of the original data.

In tables 3 and 4 we show a few examples of simplified data from the CGI and OncoKB databases, respectively. Each row in these tables represents a genetic sequence variant that can be associated with the patient, gene details (existing in both databases), as well as additional information specific to each database.

Patient	Gene	Transcript	Oncogenic Summary
patient_1	NRG1	ENST00000523534	non-oncogenic
patient_2	IRF4	ENST00000380956	non-oncogenic
patient_3	NOTCH1	ENST00000651671	oncogenic

Table 3: Example data from CGI database

ID	Patient	Hugo Symbol	Treatments	PubMed Citation
0	patient_1	NRG1	Palbociclib	PM_1
1	patient_2	IRF4	Tazemetostat	PM_2
2	patient_3	NRG1	Olaparib	PM_3

Table 4: Example data from OncoKB database

```

1 subject: variant
2 columns:
3   patient:
4     to_object: patient
5     via_relation: patient_has_variant
6   gene:
7     to_object: gene_hugo
8     via_relation: variant_in_gene
9   transcript:
10    from_subject: gene_hugo
11    to_object: transcript
12    via_relation: transcript_to_gene_relationship
13  oncogenic_summary:
14    to_object: disease
15    via_relation: variant_to_disease

```

Figure 13: OntoWeaver mapping file for CGI database

The OntoWeaver mapping files for the CGI and OncoKB databases are shown in Figures 13 and 14. Figure 15 represents a simplified version of the Biolink ontology that was used for this example.

By leveraging Biocypher output adapters, we can export and visualize the integrated data in the Neo4j graph database. The graph on Figure 16 shows the integrated information from both CGI and OncoKB databases, associated with the sequence variants from both databases.

<https://neo4j.com>

Invasive Species Monitoring

The second illustrative use case deals with the monitoring of invasive species in the Mediterranean Sea. The collected data is made of reports of jellyfish sightings, achieved through the Meduzot [11] citizen science app. In order to study the contexts of occurrences and the monitoring of jellyfish, the data gathered is processed and injected into a knowledge graph, on which further processes such as frequent pattern mining are done.

```

1 subject: variant
2 columns:
3   patient:
4     to_object: patient
5     via_relation: patient_has_variant
6   hugoSymbol:
7     to_object: gene_hugo
8     via_relation: variant_in_gene
9   treatments:
10    to_object: drug
11    via_relation: variant_affected_by_drug
12  citationPubMed:
13    from_subject: drug
14    to_object: publication
15    via_relation: treatment_has_citation

```

Figure 14: OntoWeaver mapping file for OncoKB database

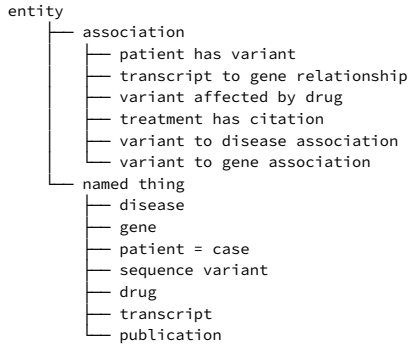


Figure 15: Subset of the Biolink ontology used for data integration from CGI and Oncokb databases.

In this example, we show how to create an SKG from one database, using two combined ontologies to model different aspects of the domain. The first ontology models the specific monitoring domain, whilst the second one, the URREF ontology [12], models the level of accuracy attached to observations made by citizens.

User_ID	ObsID	datetime_ori	Location_20_Zones_ID	Species	Gold_User
A	1091	2023-07-03 18:08:00	Habonim Beach Nature Reserve	Rhopilema nomadica	0
B	6	2023-01-02 17:10:00	Habonim Beach Nature Reserve	Rhopilema nomadica	1
B	7	2023-01-03 08:06:00	Beit Yanai	Unknown	1

Table 5: Example of jellyfish observations

In Table 5 we show a few examples of our dataset. Each line describes an observation made by one of the Meduzot users of jellyfish on the Israeli coast. Figures 17 and 18 represent the parts of the two ontologies that we use for our simple illustration. The OntoWeaver mapping script is shown in Figure 19 and the BioCypher configuration is shown in Figure 20.

The OntoWeaver mapping defines that each line of the dataset is a node of type *jellyfishobservation*, linked to a *namedzone* node (whose value is provided by the *Location_20_Zone_ID* column) through a *has-forlocation* relation. The observation node is also linked to a node of type *user*, through a *reportedby* relation. All these node types are classes de-

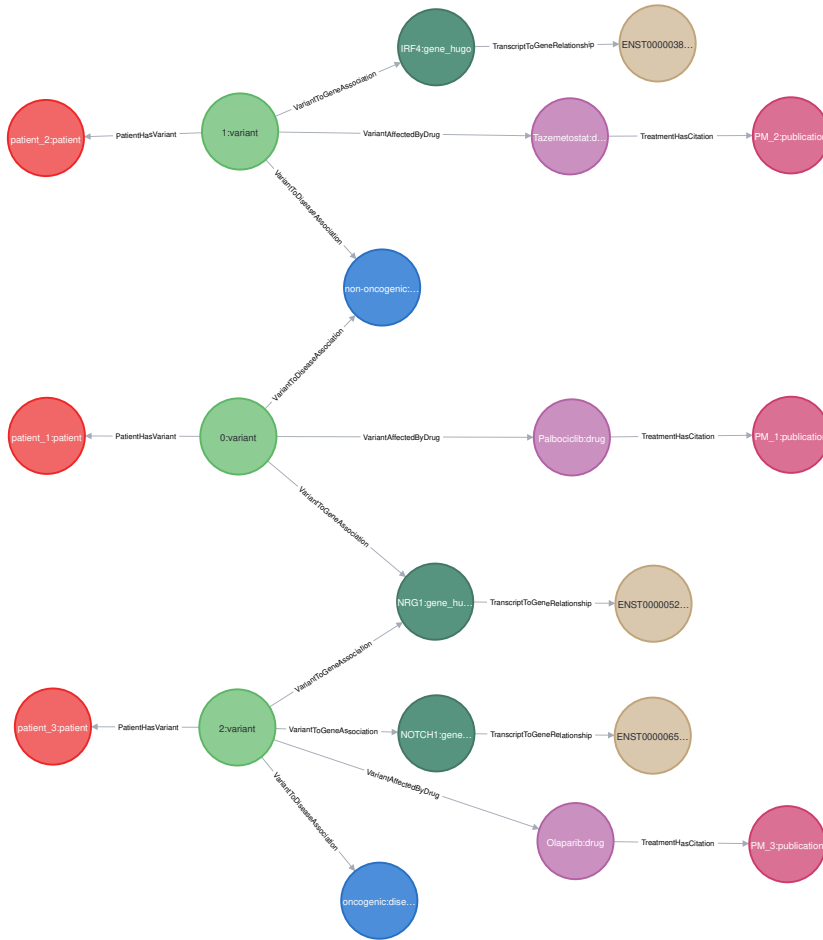


Figure 16: Visualization of integrated data from CGI and OncoKB databases. From left to right: red nodes represent patients, each of whom is connected to sequence variants, represented by green nodes. Each variant is further associated with a drug (purple), a gene (dark green), and an oncogenic summary (blue). If a type in the database is associated with multiple variants, it is integrated as a single node with various associations in the graph. Such is the case of the NRG1 gene, and the non-oncogenic summary. Drugs are further associated with the corresponding PubMed citations (pink), and genes with their corresponding transcripts (beige).

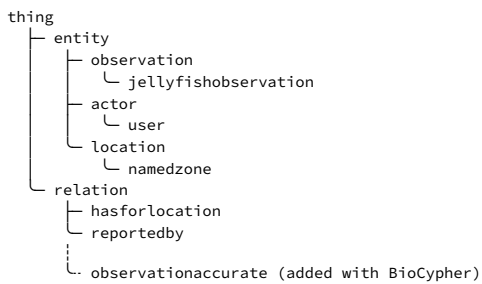


Figure 17: Subset of the domain specific ontology developed in the ILIAD project.

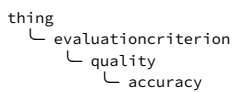


Figure 18: Subset of the URREF ontology

```

1 subject: jellyfishobservation
2 columns:
3   Location_20_Zones_ID:
4     to_object: namedzone
5     via_relation: hasforlocation
6   User_ID:
7     to_object: user
8     via_relation: reportedby
9   Gold_User:
10    to_object: accuracy
11    via_relation: observationaccurate

```

Figure 19: OntoWeaver mapping for the invasive species monitoring use case, showing how columns of the tables are mapped to typed node and edges.

```

1 biocypher:
2   head_ontology:
3     url: illiad.ttl
4     root_node: Thing
5
6   tail_ontologies:
7     urref:
8       url: URREF.ttl
9       head_join_node: entity
10      tail_join_node: EvaluationCriterion

```

Figure 20: BioCypher configuration for the invasive species monitoring use case, showing how ontologies are assembled.

fined in the first “ILIAD” ontology.

Each observation is also linked to a level of accuracy. This level depends on the level of expertise of the app user that provides the observation. Some citizens have been trained to recognize and report the presence of jellyfish. These users are called “Gold Users” by the marine biologists, and their status is recorded in the data file in the *Gold_User* column. The domain ontology does not provide means to describe the quality of the data stored in the dataset. Therefore, we use the URREF ontology to map the accuracy of each observation. Furthermore, no relation existed in the ILIAD ontology to capture this relation between the fact that a user was trained and the quality of their observation. The relation *observationaccurate*, depicted with a dotted line in Figure 17 does not exist in any ontology, but was added manually through the BioCypher schema configuration, as shown on Figure 21.

The resulting SKG is shown on Figure 22.

Conclusion

OntoWeaver and BioCypher are software tools that enable mapping a tabular database into a Semantic Knowledge Graph (SKG), according to a model of the application domain expressed through one or more ontologies. The only required configuration is a declarative mapping, describing how to extract data cells as typed nodes and edges in the SKG. The creation of the SKG is thus reproducible and can be rerun every time the database changes.

Our approach improves on the use of BioCypher alone, since OntoWeaver allows users with few programming skills to write a mapping. It actually creates a BioCypher “adapter” on-the-fly, so that BioCypher can assemble the required ontologies, check their consistencies, and send

```

1  jellyfishobservation:
2     represented_as: node
3     label_in_input: jellyfishobservation
4  namedzone:
5     represented_as: node
6     label_in_input: namedzone
7  hasforlocation:
8     represented_as: edge
9     label_in_input: hasforlocation
10     source: observation
11     target: location
12  user:
13     represented_as: node
14     label_in_input: user
15  reportedby:
16     represented_as: edge
17     label_in_input: reportedby
18     source: observation
19     target: user
20  accuracy:
21     represented_as: node
22     label_in_input: accuracy
23  observationaccurate:
24     is_a: relation # Manual insertion in the ontology.
25     represented_as: edge
26     label_in_input: observationaccurate
27     source: jellyfishobservation
28     target: evaluationcriterion

```

Figure 21: BioCypher schema for the invasive species monitoring use case, showing how types are handled. The type *observationaccurate* is manually added as a subtype of *relation*.

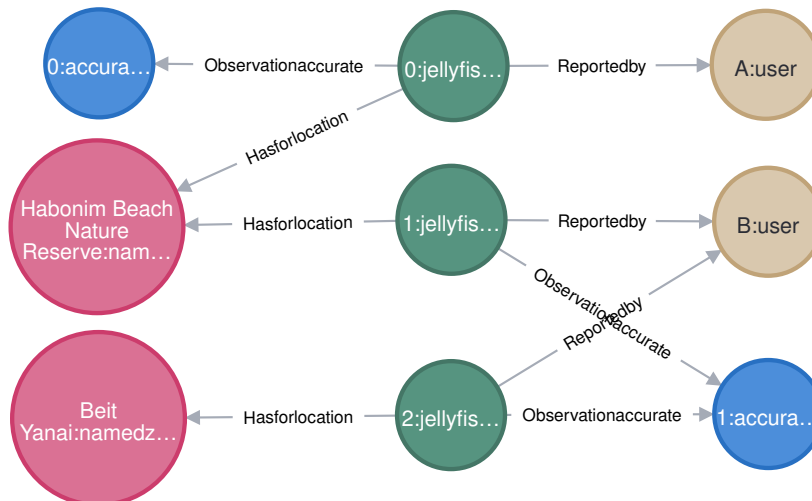


Figure 22: Visualisation of a sample of the Meduzot dataset, mapped to two ontologies. Each entry of the dataset is mapped to an observation (green nodes), associated with the user reporting it (brown) and its location (red). A level of accuracy (blue), is associated with each observation.

the SKG to any of its supported graph database engines.

Furthermore, in a data fusion perspective, OntoWeaver and BioCypher allow to declare several mappings for several tables, and then to run them sequentially in a single pass, effectively creating a single, integrated, SKG.

BioCypher and OntoWeaver are agnostic of the specific data domain. To demonstrate their adaptability, we presented two very different use cases: cancer database integration and invasive species monitoring.

Both BioCypher and OntoWeaver are available as free and open-source software.

See <https://biocypher.org> and
<https://github.com/oncodash/ontoweaver>

References

- [1] M. Ginda, B. W. Herr, and K. Börner, Introducing the open biomedical map of science, in *Frontiers Media SA*, Oct. 2023, vol. 8. DOI: [10.3389%2Ffrma.2023.1274793](https://doi.org/10.3389/fm.2023.1274793).
- [2] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, *et al.*, “The fair guiding principles for scientific data management and stewardship,” *Scientific data*, vol. 3, no. 1, pp. 1–9, 2016.
- [3] S. Lobentanzer, P. Aloy, J. Baumbach, *et al.*, “Democratizing knowledge representation with biocypher,” *Nature Biotechnology*, pp. 1–4, 2023.
- [4] D. R. Unni, S. A. Moxon, M. Bada, *et al.*, “Biolink model: A universal schema for knowledge graphs in clinical, biomedical, and translational science,” *Clinical and translational science*, vol. 15, no. 8, pp. 1848–1855, 2022.
- [5] W. McKinney, “Data Structures for Statistical Computing in Python,” in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 56–61. DOI: [10.25080/Majora-92bf1922-00a](https://doi.org/10.25080/Majora-92bf1922-00a).
- [6] D. R. Unni, S. A. T. Moxon, M. Bada, *et al.*, “Biolink model: A universal schema for knowledge graphs in clinical, biomedical, and translational science,” *Clinical and Translational Science*, vol. 15, no. 8, pp. 1848–1855, 2022. DOI: <https://doi.org/10.1111/cts.13302>. eprint: <https://ascpt.onlinelibrary.wiley.com/doi/pdf/10.1111/cts.13302>.
- [7] D. Tamborero, C. Rubio-Perez, J. Deu-Pons, *et al.*, “Cancer genome interpreter annotates the biological and clinical relevance of tumor alterations,” *Genome medicine*, vol. 10, pp. 1–8, 2018.
- [8] D. Chakravarty, J. Gao, S. Phillips, *et al.*, “Oncokb: A precision oncology knowledge base,” *JCO precision oncology*, vol. 1, pp. 1–16, 2017.

- [9] S. P. Suehnholz, M. H. Nissan, H. Zhang, *et al.*, “Quantifying the expanding landscape of clinical actionability for patients with cancer,” *Cancer Discovery*, vol. 14, no. 1, pp. 49–65, 2024.
- [10] J. Dreo, S. Lobentanzer, E. Gaydukova, *et al.*, “High-level biomedical data integration in a semantic knowledge graph with onco-dashkb for finding personalized actionable drugs in ovarian cancer,” in *Cancer Genomics, Multiomics and Computational Biology conference*, European Association for Cancer Research, to appear.
- [11] D. Edelist, Ø. Knutsen, I. Ellingsen, *et al.*, “Tracking jellyfish swarm origins using a combined oceanographic-genetic-citizen science approach,” in *Frontiers in Marine Science*, 2022. [Online]. Available: <https://api.semanticscholar.org/CorpusID:248268498>.
- [12] P. Costa, K. Laskey, E. Blasch, and A.-L. Joussemme, “Towards unbiased evaluation of uncertainty reasoning: The urref ontology,” Jan. 2012, pp. 2301–2308, ISBN: 978-1-4673-0417-7.