Informatics

Verbesserung der maschinellen Verarbeitbarkeit von strukturierten Forschungsdaten

durch halbautomatisches Ontologie-Mapping von Datenattributen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Bugra Altug, BSc
Matrikelnummer 11938259

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.univ.Prof. Dr. Andreas Rauber

Wien, 16. August 2024

Bugra Altug	Andreas Rauber

Informatics

Increasing the Machine-actionability of Structured Research Data

via Semiautomatic Ontology Mapping of Data Attributes

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Bugra Altug, BSc
Registration Number 11938259

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.univ.Prof. Dr. Andreas Rauber

Vienna, August 16, 2024

Bugra Altug

Andreas Rauber

Erklärung zur Verfassung der Arbeit

Bugra Altug, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 16. August 2024

Bugra Altug

Danksagung

Zuallererst möchte ich meinem Betreuer, Andreas Rauber, meinen tiefsten Dank aussprechen. Ohne sein Fachwissen, seine Mitarbeit und sein wertvolles Feedback wäre diese Arbeit nicht möglich gewesen.

Mein Dank gilt auch Martin Weise, der das Forschungsdatenmanagement-Tool Database Repository an der Seite meines Betreuers konsequent gepflegt hat.

Außerdem bin ich Ádám Erdélyi für seine unermüdliche Unterstützung seit meiner Ankunft in Wien dankbar.

Zu guter Letzt möchte ich meiner Familie meine aufrichtige Anerkennung aussprechen. Meine Mutter, Aslı Altuğ, und mein Vater, Tayfun Altuğ, haben mich immer mit allem versorgt und mir zur Seite gestanden. Meine geliebte Frau, Fatma Odabaş Altuğ, hat meinem Leben einen wahren Sinn gegeben.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Andreas Rauber. This thesis would not have been possible without his expertise, collaboration, and invaluable feedback.

I would also like to extend my thanks to Martin Weise, who has consistently maintained the Database Repository research data management tool alongside my supervisor.

Additionally, I am grateful to Ádám Erdélyi for his unwavering support since my arrival in Vienna.

Lastly, I want to express my heartfelt appreciation to my family. My mother, Aslı Altuğ, and my father, Tayfun Altuğ, have always provided me with everything and stood by my side. My beloved wife, Fatma Odabaş Altuğ, has brought true meaning to my life.

Kurzfassung

Die Optimierung der Verwaltung von Forschungsdaten kann die Effizienz von Wissenschaftlern steigern. Ein Datenbank-Repository, das Richtlinien für Auffindbarkeit, Zugänglichkeit, Interoperabilität und Wiederverwendbarkeit befolgt, spielt dabei eine zentrale Rolle. Um die maschinelle Verwertbarkeit weiter zu verbessern, muss die Interoperabilität von Daten über verschiedene wissenschaftliche Bereiche hinweg gewährleistet sein. Dies kann durch die Definition der Semantik und der Maßeinheiten der Daten unter Verwendung mehrerer benutzerdefinierter Objektontologien und einer einheitlichen Einheitenontologie erreicht werden. Dies ermöglicht es Wissenschaftlern, mit Objekten in verschiedenen Bereichen zu arbeiten und Berechnungen mit einheitlichen Einheitenbezeichnungen durchzuführen.

Dies erfordert eine halbautomatische Zuordnung von Attributen zu Entitäten aus einer Objekt- und Einheitenontologie. Während die Zuordnung von Schema-Spalten zu Ontologie-Entitäten gut erforscht ist, befassen sich die bestehenden Systeme nicht speziell mit der Herausforderung, wissenschaftliche Daten abzubilden und gleichzeitig Entitäten zu isolieren, insbesondere bei der Zuordnung von Einheiten.

Wir schlagen daher ein halbautomatisches System vor, das eine Methode zur Berechnung der Ähnlichkeit von Einheiten, zwei Möglichkeiten zur Beeinflussung der Ergebnisse durch Benutzereingaben und eine Strategie zur Optimierung des Einsatzes dieser Methoden bietet.

Unser System ordnet wissenschaftliche Schemadatenspalten mit Kardinalitäten von $n : 1$ und $1 : 1$ auf der Elementebene sowohl Objekt- als auch Unit-Entitäten zu. Es nutzt ein Texteinbettungsmodell zur Kodierung von Spaltennamen und Entitätsbezeichnungen, wobei die Kosinusähnlichkeit die Relevanz berechnet. Dieser Ansatz schlägt 89.9% der korrekten Objektentitäten in den ersten 10% aller Objektentitäten vor (Entity Coverage) und erreicht einen Mean Reciprocal Rank (MRR) von 0.5259, was alle anderen Ansätze übertrifft. Ein ähnlicher Kodierungsansatz, der das Schlüsselwort ünit"hinzufügt, wird für die Ähnlichkeit zwischen Spalten und Einheitsentitäten verwendet, erzielt eine Abdeckung von 64.4% und einen MRR von 0.1164. Die einheitliche Ähnlichkeitsmetrik für Objekt- und Einheitenvorschläge ermöglicht zwei neue Methoden zur Verbesserung der Abdeckung und MRR durch Benutzereingaben während des Mapping-Prozesses.

Abstract

Optimizing the management of research data may increase scientists' efficiency. The Database Repository, as a research data management tool, plays a crucial role in achieving this optimization by following the Findability, Accessibility, Interoperability, and Reusability guidelines, which emphasize machine-actionability. However, to further enhance the machine-actionability of experiments, it is essential to ensure data interoperability across different scientific domains. Interoperability in scientific data can be achieved by defining the semantics of what the data represents and the used units of measurement. This approach involves employing multiple custom object ontologies alongside a unified unit ontology. This setup enables scientists to interact with objects from various scientific domains while maintaining the ability to perform calculations using consistent unit Internationalized Resource Identifiers.

Achieving this requires a semi-automatic mapping of attributes (e.g., columns in a relational database schema) to entities from both an object ontology and a unit ontology. While mapping schema columns to ontology entities is a well-established area, existing systems do not specifically tackle the distinct challenge of mapping scientific data while isolating units, particularly in the context of unit matching.

Therefore, we propose a semi-automatic system that introduces an approach for calculating unit similarity, two methods that use user input to influence the outcome of unit similarity results, and a strategy for utilizing these methods to achieve optimal performance.

Our system maps scientific schema data columns with $n : 1$ and $1 : 1$ element-level cardinalities to both object and unit entities. It utilizes a text embedding model to encode column names and entity labels of objects that employ cosine similarity to calculate relevance. This approach successfully suggests 89.9% of the correct entities within the first 10% of all entities (entity coverage) and achieves a Mean Reciprocal Rank (MRR) of 0.5259, outperforming all other approaches. For calculating the similarity between columns and unit entities, a similar approach is employed with an encoding method that adds the "unit" keyword at the end of entity labels. This achieves a 64.4% entity coverage and 0.1164 MRR, also surpassing all other tested approaches. The unified similarity metric used for object and unit suggestions allows for the application of two new indirect and direct influencing methods during the mapping process which threats users as an auxiliary linguistic resource. These methods improve the overall coverage and MRR of the mapping when used according to our introduced strategies.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

In numerous institutions, research teams, and laboratories, databases are essential for managing data before conducting actual research. Optimizing the management of such data may boost scientists' efficiency. FAIR[WDA+16] guideline assures to achieve this by machine actionability. FAIR contains four main principles. Findability can be achieved by providing rich metadata that can describe and identify the data. Accessibility by introducing authentication for the data with always visible metadata. Interoperability by data being integrateble with other data via metadata of broad knowledge representation languages and processable by other systems. Lastly reusibility by providing well-described, domain-relevant data and specifically licenses information. Applying these principles to a data management system is called FAIRification process.

Research Data Management Tool which is known as Database Repository (Database Repository (DBRepo)) [WMS+21] [wei22], has effectively addressed these challenges by providing a repository system. Researchers can use DBRepo to seamlessly integrate scientific data into (abstracted) external databases. In its current state, the Database Repository consists of multiple instances of Maria DB, each storing versioned data. All these database instances are connected to a metadata database that stores schema information and column data types. With this ability, *findability* and *accessibility* principles are highly achieved, thereby machine-actionability can be further improved.
Users of DBRepo have the ability to upload their scientific data in CSV format into DB instances within DBRepo. Subsequently, the table schema definition process takes place. Figure 1.1 illustrates the table schema definition within the DBRepo, where the value options of the columns are automatically detected. Additionally, primary key, null, and unique attributes are automatically suggested. In the newer version of DBRepo, users can add or remove columns, providing more flexibility.

As DBRepo serves as a research data management tool, it must accommodate a wide range of study areas while providing the machine-actionability of the FAIR requirements. Relational schema data, with attributes as freely set column names, can accommodate data from all scientific fields but requires human interpretation. Mapping a relational schema from any scientific field to an ontology of the same field will provide machine-actionability.

Currently, DBRepo allows manual mapping of attribute names to concepts from Wikidata [VK14] and is limited to the ontology. However, this process can be semi-automated, and custom object ontologies tailored specifically for scientific needs should be introduced. This would increase the system's ontology customization capabilities, improving its utility for diverse research domains and their requirements.

In addition to mapping the semantics of attribute names, measurement units can also be mapped to unit ontologies. Attribute mapping to objects and units will enable interoperability between different object ontologies through the same unit ontology. For example, two datasets containing meteorology and air quality measurements may share a common attribute name, "Temperature". The column from the meteorology dataset will be mapped to the meteorology ontology, while the same column from the air quality dataset will be mapped to the air quality ontology. These different object ontologies can define the "Temperature" differently (meteorology ontology [KRK12] defines "Temperature" as a class meanwhile air quality ontology [Cor19] defines it as an individual). However, since both columns share the attribute "Temperature", the measurements will be in either Celsius or Fahrenheit. Thus, a separate mapping will occur for the columns to Celsius or Fahrenheit. This approach allows "Temperature" measurements from the meteorology ontology to interoperate with the "Temperature" measurements from the air quality ontology.

Furthermore, selecting the most suitable ontology from the wide array of domain-specific ontologies overlapping in scope is also necessary, as the same object can have semantic differences in different scientific areas. For instance "Temperature" class in the meteorology ontology [KRK12] has subclasses such as "RoomTemperature" or "AboveRoomTemperature". Meanwhile, such room temperature distinction does not exist in the water quality ontology [Cam19].

In summary, the problem being faced is the lack of an efficient solution that guides the users through the process of mapping the data attributes and their measurement units to equivalent semantic concepts from object and unit ontologies in order to improve the machine-readability and actionability of data.

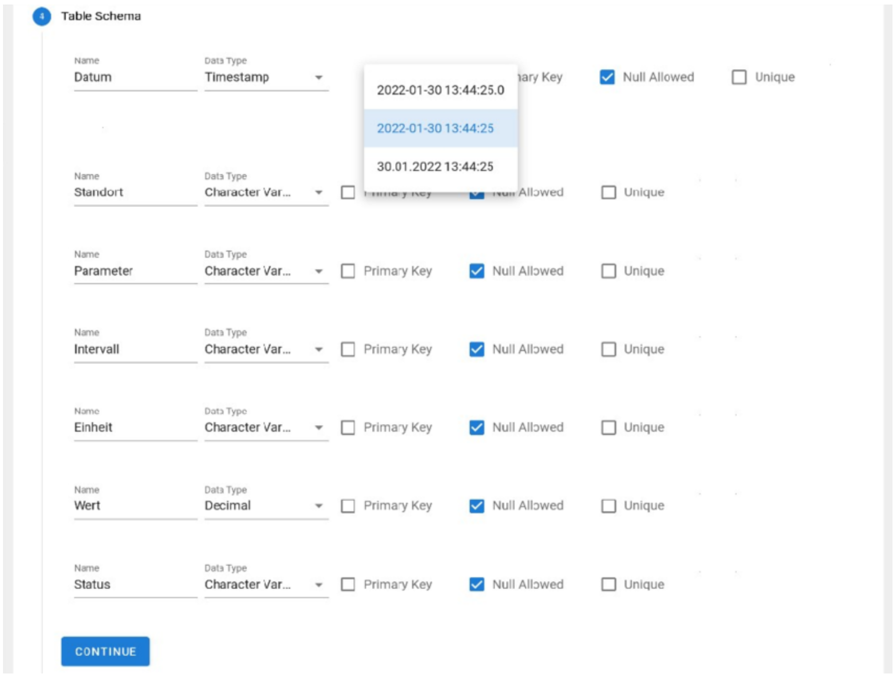


Table Schema

Name: Datum	Data Type: Timestamp	ary Key	✓ Null Allowed	☐ Unique
Name: Standort	Data Type: Character Var...	☐ Primary Key	✓ Null Allowed	☐ Unique
Name: Parameter	Data Type: Character Var...	☐ Primary Key	✓ Null Allowed	☐ Unique
Name: Intervall	Data Type: Character Var...	☐ Primary Key	✓ Null Allowed	☐ Unique
Name: Einheit	Data Type: Character Var...	☐ Primary Key	✓ Null Allowed	☐ Unique
Name: Wert	Data Type: Decimal	☐ Primary Key	✓ Null Allowed	☐ Unique
Name: Status	Data Type: Character Var...	☐ Primary Key	✓ Null Allowed	☐ Unique

2022-01-30 13:44:25.0
2022-01-30 13:44:25
30.01.2022 13:44:25

Figure 1.1: Defining the table schema in DBRepo

1.2 Aim of the work

We propose *semi-automated matching between the relational schema of user-provided scientific data to object and unit ontology.* This enhancement provides an object and a unit relevance list for each column of a given data set. Only the column names and their data types, calculated by DBRepo, are used. Relevance lists for units are not generated if the data type is string or datetime. Users have the freedom to select which concept, object property, data property, data type, or individual to match. The relevance list for a column is based on similarity score, data-type constraint satisfaction, and ontology name of the entities. If a desired entity is not included in the relevance list, a keyword-specific relevance list can be created with the ability to select *None* to keep matching blank. Additionally, a list of relevant ontologies is provided to ensure that users select entities from the correct ontology. Within the relevant ontology list, users can see averaged similarity scores of the highest-ranked entities and standard deviation. By implementing this system, users can streamline the process of adding metadata, thereby increasing the findability, interoperability, and reusability of their data.
Our system also enables administrators to provide Web Ontology Language (OWL), specifically OWL2 [HKP$^+$09], ontologies from any domain with minimal restrictions. Furthermore, administrators can specify whether the relevance list includes concepts, object properties, data properties, data types, or individuals, which will become handy when dealing with unit ontologies. This way, administrators can fine-tune the metadata that is broad enough to represent the underlying data while restricting the relevance

list from certain entities to improve relevance. The system can better capture the rich semantics of diverse scientific domains, thus improving data interoperability and reducing ambiguity in data interpretation. The ultimate goal is to create a flexible system that improves machine actionability. The system is running as a stateful Application Programming Interface (API) where the client states are handled in a caching mechanism. API ensures easy integration into DBRepo table schema creation process with various endpoints such as executing the Initial Mapping, keyword search, influencing mechanism, and adding/removing/editing the columns. Meanwhile, the caching mechanism enables clients to interact with their initial mapping data. Additionally, our front-end is designed to require as few user clicks as possible to find correct mappings for the given columns. To provide such a system, we need to address the following research questions:

1. *What are the core requirements for semi-automated scientific relational data schema to ontology matching?*: This research question is addressed in the Related Work Chapter 2 and System Overview Chapter 3. These chapters discuss the information provided by DBRepo, the need for unit entities and the importance of separating them from object entities, various matching approaches, and how to combine these propositions.

2. *To what extent can column names and data types of a scientific relational data schema be semi-automatically matched with entities of ontologies?*: Evaluation is conducted quantitatively by calculating the Mean Reciprocal Rank (MRR) and determining whether the ground truth entity is ranked first or falls within the top 5% or 10%. Additionally, a quantitative evaluation assesses our semi-automated User Interface (UI) by calculating the (required) minimum number of user clicks for selecting the correct entities and a qualitative evaluation assesses the ease and requirements for introducing new ontologies.

3. *To what extent can ontology relevance be semi-automatically calculated for scientific relational data schema?*: Evaluation is done quantitatively by checking if the winner object ontology is the ground truth or not.

1.3 Contribution

Our contributions are as follows:

1. Identifying core requirements for semi-automated scientific relational data schema to ontology matching combined with the DBRepo [wei22][WMS$^+$21] table schema requirements. In our context, the term *semi-automatic* entails automatically presenting the most pertinent ontology (or entity) for the given relational schema (or column).

2. Providing a semi-automated mapping system that utilizes embedding-based similarity metrics between the relational schema of user-provided scientific data and an object and unit ontology.

3. Applying text embedding to calculate similarities between schema column name-unit entity matches. Furthermore, using user-selected object entities as auxiliary information to improve unit entity matching results.

As previously mentioned, the system will be evaluated both qualitatively and quantitatively using scientific relational datasets. The system's performance will qualitatively imply the fulfillment of the core requirements.

1.4 Outline

The thesis begins with the Related Work Chapter 2 which discusses the concept of DBRepo and potential improvements (Section 2.1), ideas and the types of ontologies that can be used for scientific schema mapping (Section 2.2), World Wide Web Consortium (W3C) recommended approach (Section 2.3), ontology alignment evaluation initiative and ontology alignment systems (Section 2.4), preliminary schema-based matching approaches (Section 2.5), and state-of-the-art usage of embedding models in node classification.

Chapter 3 elaborates on the design of our system, encompassing the design philosophy (Section 3.1) and providing an overview of the architecture and data flow (Section 3.2).

Chapter 4 discusses the procedure our system follows when processing inputs. How administrators can customize input configurations (Section 4.1), the process of reading the ontology (Section 4.2), and then representing the extracted information as embedding vectors (Section 4.3).

Chapter 5 dives into how the user-provided schema is matched with ontologies. First, perform constraint satisfaction checks (Section 5.1). Next, calculate pairwise similarity scores between columns and entities (Section 5.2). Finally, represent the results as relevance lists (Section 5.3).

Chapter 6 demonstrates how users can interact with our system. The schema can be updated(e.g., adding/removing columns) (Section 6.1), indirect or direct influence can be applied to the match (Section 6.2), and the match can be reloaded in case of a timeout (6.3).

Chapter 7 provides a detailed explanation of our system's implementation. It covers the selected frameworks and libraries (Section 7.1), the implementation of the previously mentioned algorithms (Section 7.2), communication between the back-end and front-end (Section 7.3), and the user interface (Section 7.4).

Chapter 8 details the evaluation procedure, starting with the introduction of our simulation tool (Section 8.1), followed by the selected object and unit ontologies (Section 8.2), datasets (Section 8.3), similarity calculations including metrics, encoding methods, and influencing techniques (Section 8.4), and provides discussion of the evaluation results (Section 8.5). The chapter concludes by addressing the limitations of our system in Section 8.6 and limitations related to similarity calculation in Section 8.7.

Chapter 9 presents the conclusion (Section 9.1) and explores promising future directions for work (Section 9.2).

CHAPTER 2

Related Work

The following chapter discusses the related work for our schema columns to the ontology entities mapping system and its controlling user interface. Section 2.1 provides a brief introduction to the DBRepo, which our system is built upon. Next, the need for isolation in unit ontology is explained, along with an overview of the state-of-the-art unit ontologies in Section 2.2.

The remaining sections explain the approaches for schema entity-to-column matching. Section 2.3 presents the W3C recommendation for matching, while Section 2.4 discusses the state-of-the-art techniques used in the Ontology Alignment Evaluation Initiative. Section 2.5 showcases the fundamental schema to ontology matching approaches. Finally, Section 2.6 explores state-of-the-art approaches for using text embedding models in the schema to ontology matching.

2.1 Database Repository

DBRepo is a repository provides several APIs which can be seen in Figure 2.1. Each service of the repository operates within Docker containers [Mer14] to offer flexibility. This system enriches scientific research data by providing metadata mapping, versioning, data citation, and licensing features [wei22]. The collective goal is to ensure that data complies with FAIR (Findable, Accessible, Interoperable, and Reusable) principles [WDA+16] right from the beginning of the research, rather than addressing it later.

Data citation for a specific dataset is crucial. In DBRepo, this is achieved by storing the time stamps when the data is created (valid from) and when the data is expired or modified (valid to). This ensures that the data's temporal validity is captured, allowing for accurate citation and tracking of changes over time. It is integrated into the system by using Maria DB (`https://mariadb.com/`) instances.

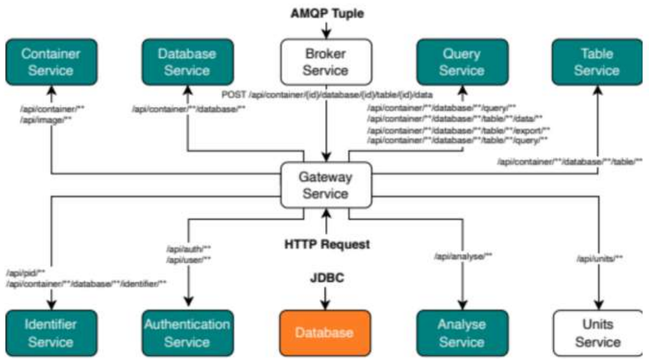

AMQP Tuple
Container Service
Database Service
Broker Service
Query Service
Table Service
POST /api/container/{id}/database/{id}/table/{id}/data
/api/container/**
/api/image/**
/api/container/**/database/**
/api/container/**/database/**/query/**
/api/container/**/database/**/table/**/data/**
/api/container/**/database/**/table/**/export/**
/api/container/**/database/**/table/**/query/**
/api/container/**/database/**/table/**
Gateway Service
HTTP Request
/api/auth/**
/api/user/**
/api/pid/**
/api/container/**/database/**/identifier/**
/api/analyse/**
/api/units/**
JDBC
Identifier Service
Authentication Service
Database
Analyse Service
Units Service

Figure 2.1: DBRepo Services Overview [wei22]

DBRepo supports both static and evolving (continuous) data. Consuming and inspecting time-series, such as real-time sensor data, is applicable through various APIs. The system protects the database's internal representation from the data source and generates SQL queries using the *Query Service* (see Figure 2.1). Researchers can also upload data through a graphical user interface, which communicates directly with the API. This functionality is introduced to assist users with limited Structured query language (SQL) expertise.
Research data can be imported into DBRepo as a new table (see Figure 2.2) or into an already existing table from comma-separated values (CSV) files. Initially, table information needs to be provided, defining the table name and description. Afterward, delimiter (comma or semicolon), line termination, and optional *NULL*, *TRUE*, and *FALSE* placeholders need to be set. Then, a CSV or Tab-separated values (TSV) file is uploaded. Lastly, the schema definition begins, as shown in Figure 1.1. In this step, any column can be selected as *primary key*, *enabling null values*, and *assuring uniqueness*.

In the newer version of DBRepo, column names can be edited, new columns can be added, and existing columns can be deleted. Most importantly, DBRepo suggests a data type for each column. This suggestion is provided by the *Analyse service*, and the suggested data types are MariaDB data types.

In one of the DBRepo publications [WMS+21], authors show their interest in extending the FAIR support by introducing an "ontology-based metadata mapping" by preserving the functionality and the intentions of their system.

2.2 Ontologies

The authors of DBRepo have acknowledged the promising possibility of enabling search across database repositories, spanning various research data from scientific domains. This can even include statistical calculations (such as min, max, etc.) when semantic information is provided.

Findability, reusability, and mainly interoperability can be improved in DBRepo by

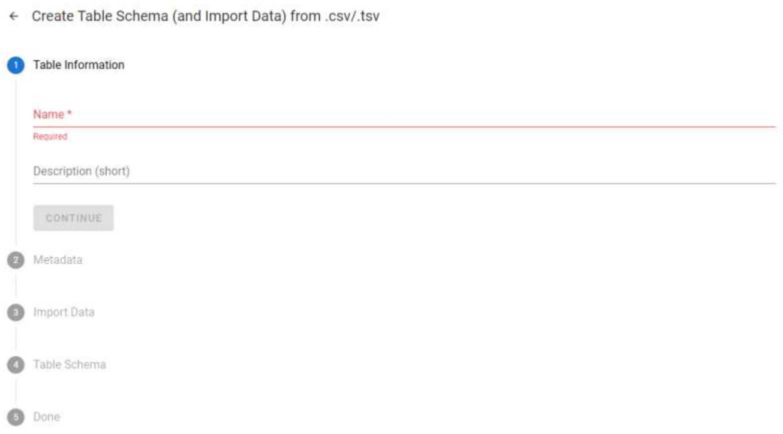


Figure 2.2: Table Schema Creation in DBRepo

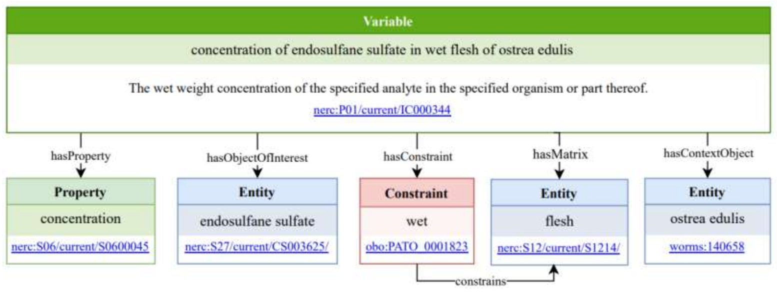

Variable
concentration of endosulfane sulfate in wet flesh of ostrea edulis
The wet weight concentration of the specified analyte in the specified organism or part thereof.
nerc:P01/current/IC000344
hasProperty
hasObjectOfInterest
hasConstraint
hasMatrix
hasContextObject
Property
concentration
nerc:S06/current/S0600045
Entity
endosulfane sulfate
nerc:S27/current/CS003625/
Constraint
wet
obo:PATO_0001823
Entity
flesh
nerc:S12/current/S1214/
Entity
ostrea edulis
worms:140658
constrains

Figure 2.3: I-ADOPT Example [MRS+21]

introducing ontology-based metadata mapping in the schema definition. However, usable ontologies that satisfy scientific needs have their own requirements.

2.2.1 InteroperAble Descriptions of Observable Property Terminolog (I-ADOPT)[MRS+21]

In the domain of biodiversity, data management has become increasingly challenging due to growing data-intensive needs. Different data sources often have different representations, leading to a lack of standardization, which becomes problematic. I-ADOPT is created to propose a solution and make the data more FAIR [WDA+16]. It contains high-level concepts to describe scientific observations, measurements, and derivations. One of the main goals is ensuring the interconnectivity of various resources, which DBRepo [wei22] can benefit from in the newer versions. The authors note that the focus of this project is environmental research; nevertheless, it remains relevant for other domains.

The framework can describe qualitative or quantitative, synthetic or real-world data, which the authors refer to as variable-centric. Furthermore, it supports encoding information about what, where, when, and how the data was collected. An example of concepts and object properties, as seen in Figure 2.3, are defined as follows:

- *Variable*: Isolated data of observation. The authors do not provide units, measurement methods, and time of observation. They urge to model the units independently from the variables because of the diversity of quantitative units [MRS+21]. Isolating the variable creates interoperability with other observations. Utilizing isolation requires having object and unit ontologies separately. That's why our system maps variables to both object and unit ontologies separately to ensure their isolation.
- *Property*: Characteristic of the object (e.g., concentration).
- *Entity*: Actor(s) of the observation. It can be an object or process (e.g., endosulfane sulfate, flesh, ostrea edulis).
- *hasObjectOfInterest*: Entity whose property is observed (e.g., endosulfane sulfate).
- *hasContextObject*: Entitiy to for additional background information (e.g., ostrea edulis).
- *hasMatrix*: Entity where the object of interest resides in (e.g., flesh).
- *Constraint*: Limits the scope of the observation (e.g., wet).

Isolating the variable requires two different types of entities: object and unit entities. Object entities describe the scientific observations of values (similar to how I-ADOPT describes them at a higher level), while unit entities describe the units of the measured variable. Using this approach ensures both the ability of statistical calculations for units and interoperability between different ontologies.

Instead of employing a complete modeling of object information, our system utilizes multiple object ontologies, each focusing on different domains. This choice allows our system to expand over time, covering a wide range of scientific fields while maintaining simplicity. Consequently, the modeling of complex object structures is delegated to external ontologies and their identifiers are then mapped to the database attributes.

2.2.2 Unit Ontologies

Various object ontologies could be found in DBPedia Archivo [FSG+20]. However, introducing interoperability between objects requires finding a unit ontology that contains all of the units of measurement across all of the scientific fields.

A survey by Zhang[ZLZP17] compares popular unit ontologies, which are defined in OWL2 [HKP+09], such as Quantity Unit Dimension Type Ontology (QUDT) [FAI22] that is created by National Aeronautics and Space Administration (NASA) and Ontology of units of Measure (OM) [RVAT13] which is using standards such as ones from National Institute of Standards and Technology (NIST) [Tay95]. Figure 2.4 shows the domains covered by each unit ontology. However, OM (now called OM2) and QUDT are continuously updated over time, which expands their coverage to include more domains. One such

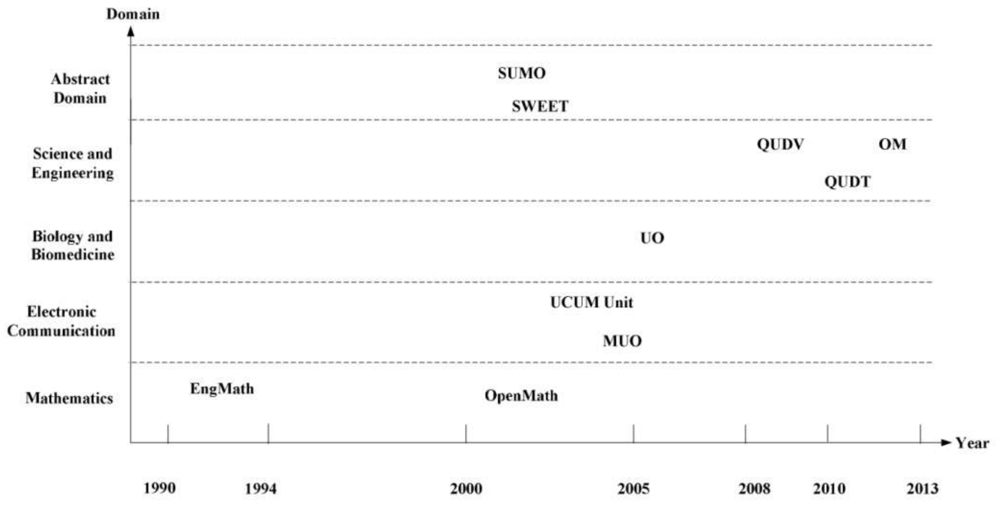


Figure 2.4: The timeline of units ontologies [ZLZP17]

example would be OM2, which includes biological units such as colony-forming units. Nevertheless, there is significant diversity among the unit ontologies. Noting that the unit overlap between OM2 and QUDT is 18.8% [KS19], our system utilizes OM2, QUDT, and Unified Code for Units of Measure (UCUM) to cover as many units as possible. Additionally, all of the utilized unit ontologies define units as instances.

Finally, there is a new unit ontology called Digital Representation of Units of Measurement (DRUM) [dru22] that is about to be released by Committee on Data (CODATA). The main goal of this ontology is to address the FAIR principles and advance the interoperability of cross-domain case studies. However, it is not available at the time of writing this thesis.

2.3 Direct Mapping

In 2012, World Wide Web Consortium (W3C) recommended a semi-automatic mapping approach called Direct Mapping [ABP+12]. This approach takes the relational schema and instance data as inputs and, in return, materializes an Resource Description Framework (RDF) graph known as the *direct graph*. Subsequently, similarity calculation can be conducted between the direct graph and ontology's graph structure.

To construct the directed graph, RDF triples are generated using the rows from the instance data. Each RDF triple of a row contains a subject Internationalized Resource Identifier (IRI), which is generated from the table name and the value of the primary key. Additionally, the base IRI and the column name of the primary key can be inserted. For example, in Figure 2.5, the student "*Alice*" is represented by the IRI "*S*01", where "*S*" stands for the table name and "01" stands for the primary key value of that specific row. Tables are represented as classes, while each data attribute is represented as a data property. RDF triples are generated according to the instance-specific data attributes

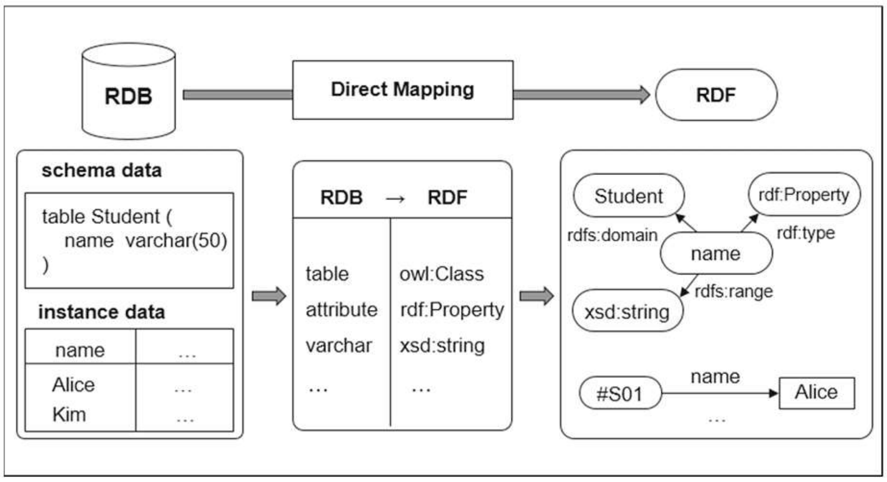


Figure 2.5: Direct Mapping [JI20]

and table names. Triple generation is skipped in cases where a data attribute contains a "*NULL*" value. On the other hand, range information of the data properties is extracted from the schema data (e.g., "*varchar*" is converted into "*xsd:string*"). Lastly, in simple terms, foreign keys are converted into RDF triples where the subject is the IRI of the instance, the predicate is the foreign key column name(s), and the object is the IRI of the referenced instance. Eventually, foreign keys are object properties, tables are concepts, data instances (rows) are instances, and data attributes are object properties in the direct graph.

2.3.1 Automatic mapping generation tools with graphs

A paper by Pieter Heyvaert (2017) [Hey17] proposes the use of existing data, schema, increasing query workload, and generated mappings (DSQM) for improvement of mappings and evaluates semi-automated ontology mappers which produce graphs. Automatic mapping generation tools can utilize the direct mapping approach to create a direct graph (bootstrap ontology), from a database, and then apply their algorithms to match it to a target ontology.

LogMap [JRG11] is one of the most popular matching generation tools for this purpose. It contains five steps which can be seen in Figure 2.6 and are described as follows:

1. *Lexical Indexation*: This step converts the label names (and synonyms) of the classes in each ontology into indexes. An external lexicon, such as WordNet [Mil95], can be optionally used.

2. *Structural Indexation*: The ontology hierarchy is treated as two Directed Acyclic Graph (DAG)s (descendants and ancestors) to separate the descendant and ancestor

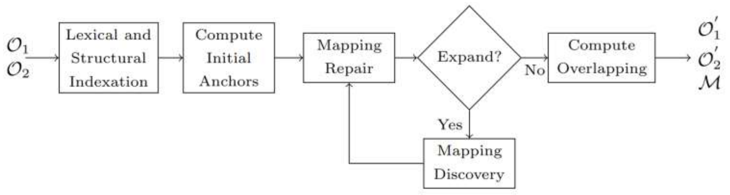


Figure 2.6: LogMap [JRG11]

relationships. An optimized data structure is used to store these DAGs. Using these DAGs, the visiting orders for each class (and their children) during a depth-first traversal are calculated. For each class, two intervals are created based on the visiting orders of the descendant and ancestor DAGs, where the starting point is the class's visiting order and the ending point is the highest visiting order of its child. These intervals enable efficient querying of the ontology hierarchy (e.g., determining if X is a subclass of Y).

3. *Initial Calculation of Anchors*: This process begins by identifying exact string matching between the two class labels from the ontologies. Once two identical class labels are found, their lexical indexes are marked as anchor points. Following this, the similarity of their neighboring classes is calculated using the ISUB similarity metric, which measures the length of common substrings. Finally, confidence scores are assigned to the anchor points based on the calculated neighbor similarities (principle of locality).

4. *Mapping Repair and Discovery*: This is an iterative process that begins with the mapping repair sub-step. It starts by checking the unsatisfiability of the classes using the Dowling-Gallier algorithm [DG84] when the two ontologies are extended using the active mappings. Next, the unsatisfiable classes are ordered by their topological level in the hierarchy. Using this order, for each unsatisfiable class, a minimal subset of mappings is removed to achieve satisfiability. If there are two possible repairs for a class, the one with the smallest confidence is chosen and removed from the mappings.

   On the other hand, the discovery sub-step involves creating and expanding *contexts* (lists of classes that belong to the neighborhood of the anchor's classes) for each anchor. Then, classes within these contexts are matched in pairs using the ISUB similarity metric. Upon finding pairwise similarities that exceed a mapping threshold, these pairs are used to extend the set of mappings.

   Overlapping Estimation: LogMap computes two fragments to represent overlaps in each ontology. Unlike anchors that are used in the previous steps, (weak) anchors in this step are defined by class-label similarities based on common substrings rather than exact matches. These weak anchors are then provided to domain experts as fragments to facilitate the manual creation of missing mappings.

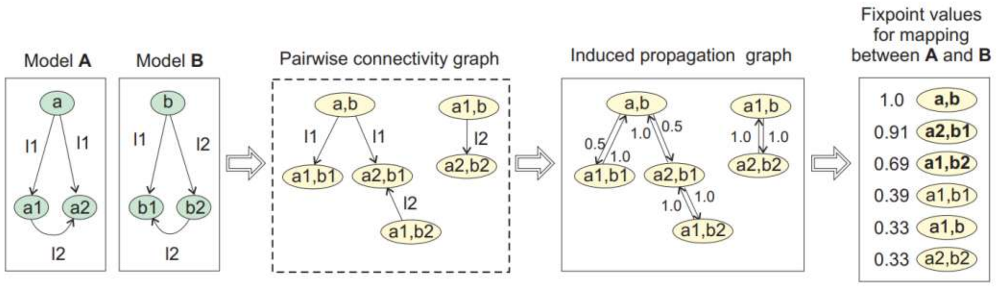


Figure 2.7: Similarity Flooding Example [MGMR02]

Another notable tool is called IncMap [PBKH13] which uses the similarity flooding algorithm [MGMR02]. Simply put, the initial similarity of any two nodes increases the similarity of their adjacent nodes. Authors noted that this approach resembles "flooding" the network when Internet Protocol (IP) packets are broadcast. The algorithm initially takes two schemas and converts them into directed labeled graphs in which data instances are converted into vertices and their relationships are represented as edges. Afterward, two graphs are joined together by some similarity metric. This final graph is called Pairwise Connectivity Graph (PCG). Afterward, inverse edges and edge weights are introduced to the PCG and fixpoint computation occurs over the similarity measurements until a certain threshold is reached. For example in Figure 2.7, let $sim(a, b)$ be our similarity measurement for a and b. Initial fixpoint computation is $sim^0(a, b)$. Each iteration increases the fixpoint computation for a, b by their neighboring similarity computations multiplied by the incoming edge weights. Thus $sim^1(a, b) = sim^0(a, b) + sim^0(a_1, b_1) * 1.0 + sim^0(a_2, b_1) * 1.0$. Finally, this similarity is normalized.

In conclusion, these approaches involve creating a graph from a schema and then applying graph algorithms for similarity assessment. While these methods provide various solutions for calculating similarity, they are not suitable for our purposes. We do not need to convert single-table relational data into a graph (with a single class); instead, we aim to describe simple string attributes (columns) using ontology concepts. Since the input consists of only a single table, there will be only one class (node) in the direct graph (or bootstrap ontology). This means that a single class (table) will be aligned to some ontology classes, rather than aligning the data properties (attributes). Therefore, alternative methods must be explored for semi-automated matching between the relational schema of user-provided scientific data and an object and unit ontology within the DBRepo framework.

2.4 Ontology Alignment Evaluation Initiative (OAEI)

In 2004, an international initiative called OAEI[oae] was founded. Its main aim is to evaluate ontology alignment methods under the same circumstances and create consensus on the right direction. Every year, they provide multiple challenges from various fields and offer test sets for evaluation purposes. This initiative has played a crucial role in advancing research in ontology alignment by providing standardized benchmarks and increasing collaboration among researchers. Each year, there are various challenges, or tracks as they are referred to, in the OAEI.

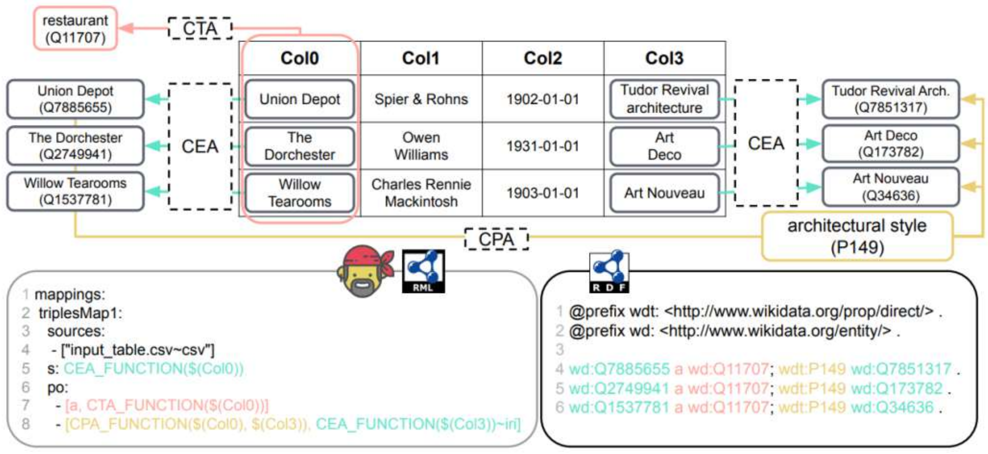
The 2020 edition [PAA+20] consisted of 12 tracks with 36 different test cases. These tracks can be classified into the following categories:

- *Schema matching*: Problems of matching between ontologies.
- *Instance Matching*: Problems of matching between individuals in the ontologies.
- *Instance and Schema Matching*
- *Complex Matching*: Introduces harder problems in-between entities from different ontologies.
- *Interactive*: Semi-automated matching problems that assess the impact of user interactions.

However, these tracks are not applicable for our use case in DBRepo. For instance, the *Schema Matching* track requires multiple ontologies, which cannot be created due to the constraints of the provided input, as described in section 2.3. Additionally, other tracks require the instance data, which, in our case described in section 2.3, is unavailable. Thus, possible solutions for such problems are also not applicable. Therefore, alternative approaches need to be explored to address the unique challenges posed by DBRepo.
There is a special track in the OAEI called Semantic Web Challenge on Tabular Data to Knowledge Graph Matching (SemTab) [oxf]. This track started in 2019 and focuses on CSV files as input, which can be interpreted as a single table. Their main goal is to improve data analytics and knowledge discovery, ultimately improving the FAIRification process as explained in the initial section 1. This track appears to be a suitable fit for our use case in DBRepo by improving the automation of Knowledge Base (KB) construction. Additionally, this problem requires matching elements of the table to Wikidata [VK14] and DBpedia [ABK+07] entities.
SemTab has three relevant tasks for our use case can be visualized in Figure 2.8:

- *Cell Entity Annotation (CEA)*: Matching each cell with a Wikidata or DBpedia entity.
- *Column Type Annotation (CTA)*: Matching each column with a Wikidata or DBpedia class.
- *Column Property Annotation (CPA)*: Finding object properties which can describe subject column(s) with their property column(s).

Meanwhile, there is a new task called *Topic Detection* which was introduced in 2023 [HAE+23]. This task focuses on the problem of matching the entire table to an entity.


restaurant (Q11707)
CTA
Union Depot (Q7885655)
The Dorchester (Q2749941)
Willow Tearooms (Q1537781)
CEA
CEA
Tudor Revival Arch. (Q7851317)
Art Deco (Q173782)
Art Nouveau (Q34636)
architectural style (P149)
CPA
RML
RDF

Col0	Col1	Col2	Col3
Union Depot	Spier & Rohns	1902-01-01	Tudor Revival architecture
The Dorchester	Owen Williams	1931-01-01	Art Deco
Willow Tearooms	Charles Rennie Mackintosh	1903-01-01	Art Nouveau

```
mappings:
 triplesMap1:
  sources:
   - ["input_table.csv~csv"]
  s: CEA_FUNCTION($(Col0))
  po:
   - [a, CTA_FUNCTION($(Col0))]
   - [CPA_FUNCTION($(Col0), $(Col3)), CEA_FUNCTION($(Col3))~iri]
```

```
@prefix wdt: <http://www.wikidata.org/prop/direct/> .
@prefix wd: <http://www.wikidata.org/entity/> .

wd:Q7885655 a wd:Q11707; wdt:P149 wd:Q7851317 .
wd:Q2749941 a wd:Q11707; wdt:P149 wd:Q173782 .
wd:Q1537781 a wd:Q11707; wdt:P149 wd:Q34636 .
```

Figure 2.8: SemTab Tasks [DCF22]

This task will occur again in 2024 challenge [oxf].

When some of the famous systems are inspected, such as JenTab[AS21], MantisTable [AC21], and SemTex [HKN$^+$23] (2023 SemTab CEA, CTA, and CPA best performer [HAE$^+$23]) follow the somewhat similar steps. At first, the table is pre-processed with subject column prediction services. Then all of them perform CEA followed by the CTA and CPA. It is important to note that CTA and CPA tasks depend on the outcome of the CEA [DCF22]. Thus common approach for all such systems is first attempt to map cells to entities or at least analyze similarities in-between. Afterward, using the aggregated knowledge, map entire columns to a Wikidata or DBpedia class.
Another example is KGCODE-TAB [LWZ$^+$22] (2022 SemTab accuracy track co-winner [ACC$^+$22]). KGCODE-TAB also has a subject column prediction service which assumes the table would have at least one subject column and the rest will be related properties. Service relies on spaCy [HMVLB20] for Named Entity Recognition (NER) to classify a cell is an entity (e.g., *PERSON*, *LOC*, *ORG*, ...) or not (e.g., *DATE* , *TIME*, *PERCENT*, ...). A column is classified as an entity if half of its cells are classified as entities. Then an entity column with the most entity diversity is selected as a subject column. This process is named *table structure analysis*. Column headers are ignored like LOD4ALL [CDPRS20], MantisTable[AC21] does. However, for CTA, instead of waiting for CEA, various similarity metrics are applied between all subject cells and all types of entities in the existing Knowledge Graph (KG). For example, one of the applied similarity metrics is *levenshtein distance*. Winner type proposes a similar subject entity for each subject cell which is then used for calculating the types of the properties cells.
The proposed approaches in SemTab challenges are important for us. However, they often rely on APIs such as the MediaWiki Action API(`https://www.mediawiki.org/w/api.php`) [HKN$^+$23]. Utilizing such APIs enables them to obtain annotations for each cell, thereby broadening the domain knowledge. In our case, the schema definition in

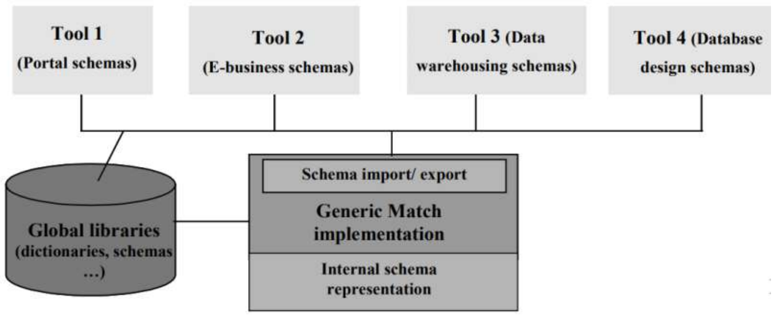

Tool 1
(Portal schemas)
Tool 2
(E-business schemas)
Tool 3 (Data warehousing schemas)
Tool 4 (Database design schemas)
Global libraries (dictionaries, schemas …)
Schema import/ export
Generic Match implementation
Internal schema representation

Figure 2.9: General architecture [RB01]

DBRepo is not dependent on a third-party API (see section 2.1).

Another problem with these approaches is that they include the cells of the table and the individuals of the ontology in the column-matching process (CTA). However, in our case (scientific research data), the cells mostly contain numerical data, whereas the cells in SemTab challenge generally contain string information. This indicates that we need to focus on the column headers (e.g., attribute names) rather than the cells themselves. In conclusion, approaches relying on external APIs and cell-level information are not suitable for our use case.

2.5 Schema-based matching approaches

In this section, more general approaches will be discussed. The main idea of these approaches is that they take two schemas as input, namely source and target schemas, and then create a mapping between them. Furthermore, schema types vary from approach to approach, such as relational schema, Extensible Markup Language (XML), and ontology. As indicated in sections 2.1 and 2.2, our requirements specify taking inputs of a relational schema and ontologies. Various surveys attempt to categorize and summarize these approaches. These surveys provide valuable insights into the different methods used for schema mapping and alignment.

2.5.1 A survey of approaches to automatic schema matching

The authors of a paper which is called *A survey on approaches to automatic schema matching* [RB01] highlight that schemas can differ significantly in terms of structure and terminology, especially when they originate from different domains. This is particularly relevant in our case, as we are dealing with schemas from various scientific domains.

The first step in handling the structure and terminology variation is understanding the inner relationships of different schemas and unifying them accordingly. In Figure 2.9, one such system design can be seen. Multiple schemas can be parsed into a unified

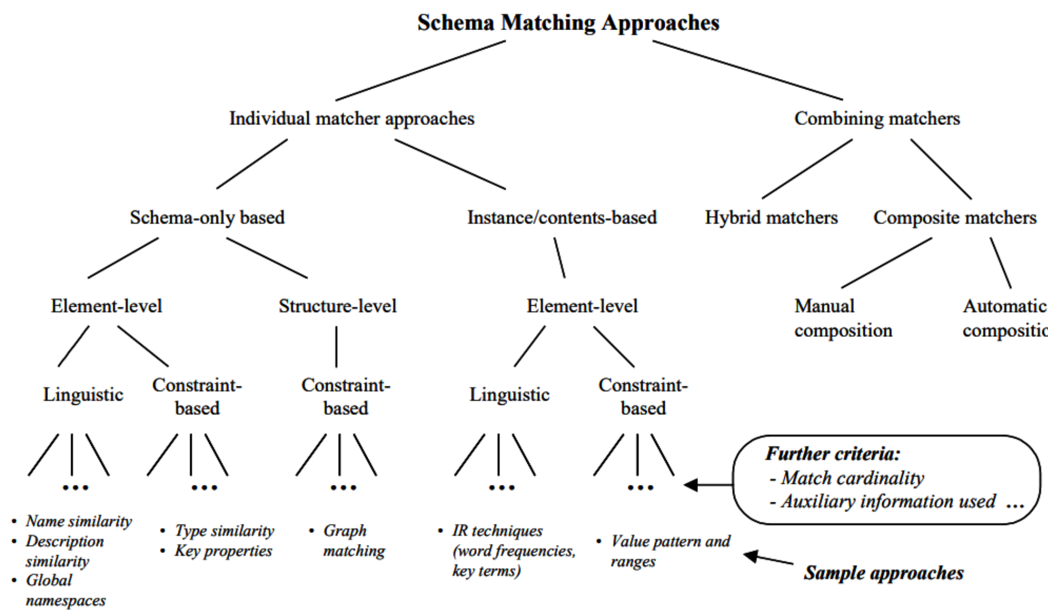


Figure 2.10: Schema Matching Approaches by Erhard Rahm [RB01]

representation. External libraries can be introduced to extend the metadata information. Afterward, matching can occur between these unified representations.

Since this design can handle multiple schemas, it provides the ability to introduce various ontologies from diverse scientific fields. Furthermore, customizable unified representations and global libraries offer flexibility in choosing matching approaches. Therefore, our system provides customization options for administrators, allowing them to flexibly read specific entity types from ontologies. For example, administrators can choose to read only individuals (measurement units) from unit ontologies to restrict the search space (see Section 2.2). Also, embedding models are utilized as global libraries (see Section 2.6). These approaches ultimately improve findability, interoperability, and flexibility in managing research data within DBRepo.

In Figure 2.10, the approaches are categorized based on whether they focus on *Element-level* or *Structure-level*. Matchers that focus on the element level solely consider entities, while matchers that focus on the structure level focus on analyzing the relationships among multiple entities. Similarity metrics are determined by combining *Linguistic* or *Constraint-based* approaches with Element or Structure level matching. This classification helps in understanding the various strategies employed in schema matching and alignment. Linguistic approaches are as follows:

- Name similarity involves calculating similarity based on the displayed names. One example is using the direct equivalence or Levenshtein distance. However, many pre-processing steps can be introduced. These include removing redundant prefix

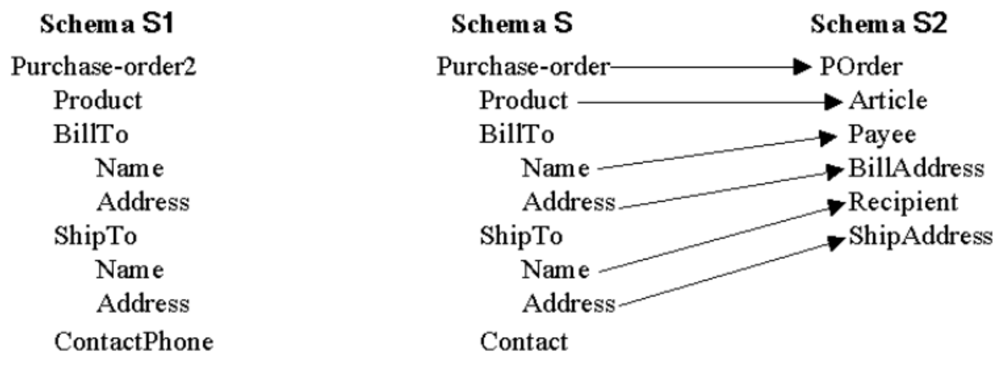


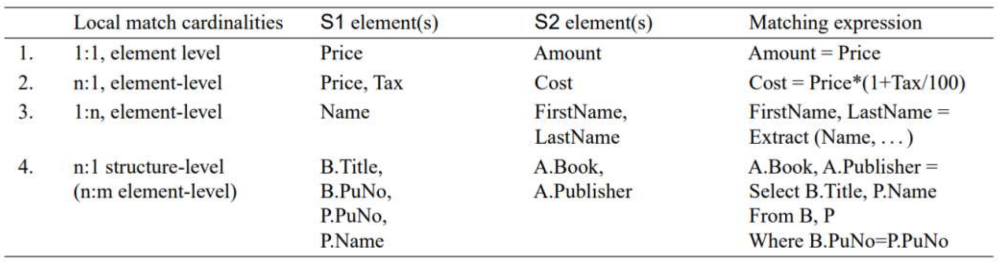
Figure 2.11: Schema Reuse [RB01]

and suffix symbols and generating synonyms, hypernyms, or even pronunciations. Nevertheless, pre-processing steps such as synonyms and hypernyms often require external data sources. Additionally, users can provide feedback on name equality or inequality to improve similarity calculations. This user feedback, specifically providing name equality, will be used in our influencing algorithm.

- Description similarity, on the other hand, utilizes annotations of the data elements. However, utilizing this approach would be limited in the DBRepo schema definition since the input is a CSV or TSV, which results in relying on the target ontology annotations. However, not every ontology has annotations.

Next, we discuss the Constraint-based approach which uses the structure information. Similarity can be assessed by checking whether the data types are equivalent or not. Conversion between the data types can also be satisfied (e.g., an integer can be converted to float, and data types between two enumeration types can be matched with each other). Using primary and foreign key relationships further relevant data types can be found. Constraint-based approach is also utilized within DBRepo using the column data types, data properties, and object properties. However, this approach can be combined with graph matching algorithms like similarity flooding algorithm [MGMR02] which was previously described in Section 2.3. Additionally, one-to-one relationship information can be used to constrain sources matching to the same target when there are multiple tables in the relational schema.

Another approach, known as schema reuse, is illustrated in Figure 2.11. The authors suggest that if a matching between **S** and **S2** already exists, the mapping could be reused for **S1** and **S2**. The requirement is that **S1** needs to be similar to **S**. Since our system will have one source as a relational data schema which is coming from the user and two separate target ontologies, namely object and unit ontologies (see section 2.2), our influencing procedure can utilize this approach with user feedback. In this approach, the user can select an object entity, and it can then be matched with unit entities. Subsequently, the matching between the object entity and unit entities can be reused



	Local match cardinalities	S1 element(s)	S2 element(s)	Matching expression
1.	1:1, element level	Price	Amount	Amount = Price
2.	n:1, element-level	Price, Tax	Cost	Cost = Price*(1+Tax/100)
3.	1:n, element-level	Name	FirstName, LastName	FirstName, LastName = Extract (Name, . . .)
4.	n:1 structure-level (n:m element-level)	B.Title, B.PuNo, P.PuNo, P.Name	A.Book, A.Publisher	A.Book, A.Publisher = Select B.Title, P.Name From B, P Where B.PuNo=P.PuNo

Figure 2.12: Match Cardinality Examples [RB01]

for the source column to unit entities. Performance regarding the schema reuse for such usage can be seen in the Evaluation Chapter 8.

Schema-matching approaches involve four possible cardinalities, which can occur at either the element level or the schema level. When matching happens between elements within the same schema, it's referred to as element-level matching. Conversely, when matching occurs between elements in different schemas, it's called schema-level matching. For the element level cardinalities (see Figure 2.12), when a source entity is matched with a target entity, it is called a $1:1$ cardinality. When $1:n$ corresponds to a source entity matching with multiple target entities. Consecutively, $n:1$ occurs when multiple source entities are matched to a single entity. Lastly, $n:n$ represents source entities matching with multiple target entities. In terms of DBRepo, there will be a single schema which means our system will support only element-level matching. Due to the provided User Interface (UI) which can be seen in Figure 1.1, DBRepo does not yet support $1:n$ element level cardinality. Thus only $n:1$ and $1:1$ element level cardinalities are being supported.

2.5.2 A Survey of Schema-Based Matching Approaches

Schema matching involves techniques that attempt to utilize implicitly defined information within the schemas, while ontology matching tries to utilize explicitly defined knowledge in the ontologies. However, schema-based matching can occur between both. *A Survey of Schema-Based Matching Approaches*[SE05] classifies such matching according to the inputs, matching process, and outputs.

Inputs can vary from XML, relational schema, ontology, etc. The matching process has two different characteristics. The first one is the similarity metrics, which can be syntactic, external, or semantic (see Figure 2.13). The other characteristic is whether exact or approximation algorithms are used for these metrics.

Output classification is based on multiple characteristics, such as post-processing requirements like match fine-tuning, matching cardinality (such as $n:1$), output of each match (confidence, probability, or an exact answer), and the match type, whether it is equivalence ($=$), subsumption ($\sqsubseteq$), or something else [RB01].

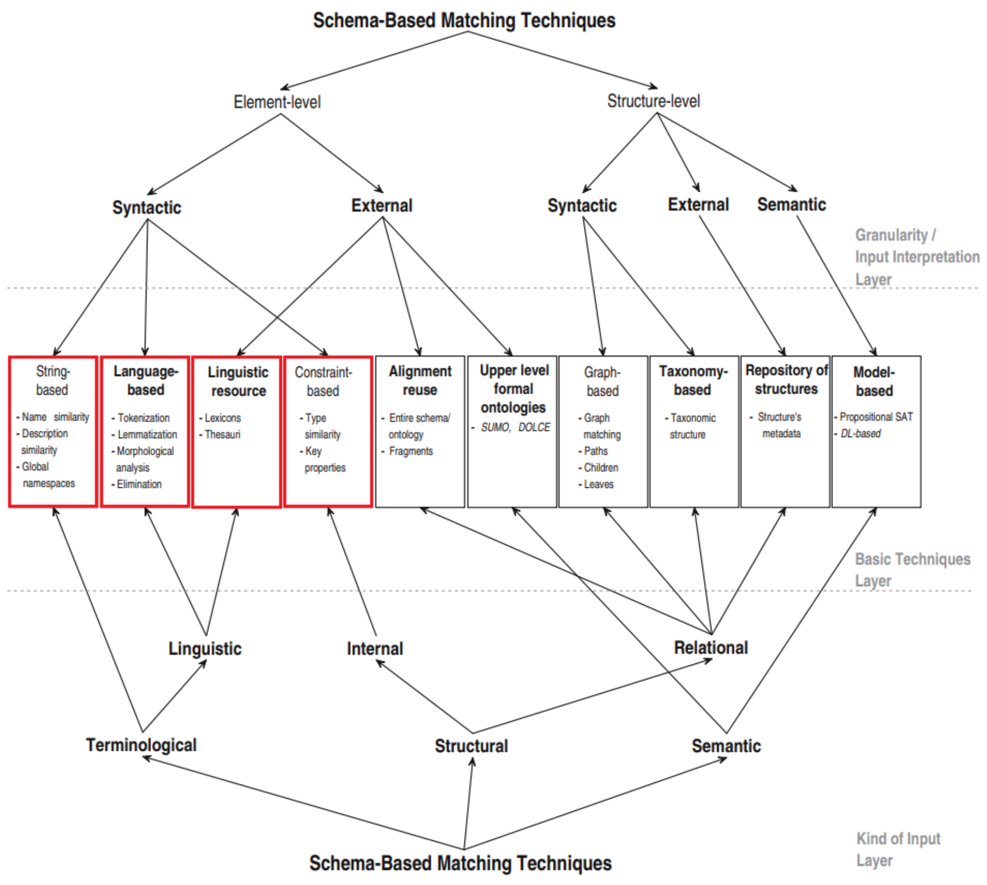


Figure 2.13: A Survey of Schema-Based Matching Approaches by Pavel Shvaiko [SE05] Utilized approaches are marked with red.

Accordingly, our system can be classified as taking one relational schema and two OWL ontologies (see Section 2.2). We employ syntactic and partially external similarity metrics with approximation algorithms (discussed in Chapter 5). The output requires fine-tuning, which will be further elaborated in the Similarity Calculation Limitations Section 8.7 and in the Future Work Section 9.2. The produced mappings are of types $n : 1$ and $1 : 1$, and the match type is equivalence (=).

In Figure 2.13, the top layer is the *Granularity layer*, which identifies how the input is *used*, containing three different similarity metric categories. Syntactic similarity metrics take inputs in their initial form, external resources use inputs to gather additional data, and semantic metrics utilize reasoning with the inputs.

The bottom layer is the *Basic Techniques Layer*, which identifies how the input is *treated*. Semantics treats its inputs as logical models, structural metrics treat them as structures, and terminological metrics treat them as strings. From this layer, we utilize terminological (specifically linguistic) and structural (specifically internal) similarity metrics.

The middle layer in Figure 2.13 is called Element level similarity metrics. String-based metrics are as follows:

- *Prefix*: Detects if the second word is starting with the first one (e.g., temp and temperature).
- *Suffix*: Detects if the second word is ending with the first one.
- *Edit distance*: Calculates the number of edits required between two words for them to be equal.
- *N-gram*: Detects whether consecutive N characters are equal or not between two words.

Language-based metrics employ Natural Language Processing (NLP) for tasks like tokenization. These metrics are often described as a standard pre-processing step before applying string-based metrics. However, in our approach, we exclusively utilize language-based metrics with the assistance of embedding models, which already contain their tokenizers. This approach is further elaborated on in Section 2.6 of the thesis.

Linguistic resources primarily leverage common or domain-specific knowledge to generate synonyms or hypernyms based on natural language inputs. In our system, the user-selected prefix, suffix, synonym, or hypernym information, represented as an embedding vector of an ontology entity, is automatically combined with the column name, also represented as an embedding vector. This combination process is crucial in our influencing step, which is why linguistic resources are utilized and will be elaborated in Chapter 6.

Taxonomy metrics rely on graph algorithms to assess the similarity between two entities by comparing their supersets and subsets. Similarly, model-based metrics begin with a graph-matching problem and transform it into a series of node-matching problems, often checking for unsatisfiability using SAT solvers. However, these metrics are not suitable for application within DBRepo due to the nature of user-provided input, which consists solely of a relational schema with a straightforward structure. Even if a direct mapping approach is employed (refer to Section 2.3), it is not feasible to construct applicable supersets or subsets.

A repository of structures is employed to accelerate the matching process. Upon the arrival of a new schema, its similarity with pre-existing structures in the repository is assessed. If a relevant structure is found, a more detailed matching process can occur, or existing alignments can be reused. The same reuse procedure illustrated in Figure 2.11 can be applied. However, such structures are currently not utilized and will be discussed in the Future Work chapter (see Future Work Section 9.2).

2.5.3 Combination Match (COMA)++

COMA/COMA++ [DR02] [ADMR05] is state-of-the-art ontology matching tool [ABM15] [Mat19] except the ones that are already discussed in OAEI Section 2.4 or in direct mapping [ABP+12] Section 2.3.

COMA++ [ADMR05] is a tool that can handle XML, relational schemas, and OWL ontologies. It utilizes ontology-based matching approaches as a hybrid matcher, employing string, language, data type, and auxiliary thesaurus metrics. Users can select which similarity metrics to use for automating the mapping. COMA++ can generate a DAG using foreign key relationships to execute matching, meanwhile user feedback can introduce biases. Although it utilizes graph-based approaches, it is not a necessity but a choice to use this metric. However, COMA++ does not convert different data types between different data models and performs constraint-based matching by analyzing all data instances of all elements (both schema and ontology) [EM07]. Additionally COMA only focuses on $1:1$ element-level matching cardinalities (see Figure 2.12) [Mat19].

The schema definition process in DBRepo includes the possibility of $1:n$ element-level matching cardinalities. Furthermore, our goal is to consume the already predicted data type information from the DBRepo, which only uses the data instances of the user-provided schema. Using this information, we will then match the predictions with RDF data types from ontologies. Lastly, a similarity metric for calculating the overall relevance of each ontology is required since our system will support multiple object ontologies.

2.6 Embedding Models

Embedding models take high-dimensional data, such as words, and encode it into a vector in a lower-dimensional embedding space. These models learn mappings from training data. In terms of NLP, similar words are encoded closer together. Their closeness can be determined by, for example, cosine similarity scoring. String-based similarity metrics, such as edit distance, can be applied between column names and object entities. However, these metrics will fail when unit entities are introduced. For example, the similarity of the "temp" column to the "Temperature" object can be detected with edit distance, but "Fahrenheit" will not be detected. This highlights the limitation of solely relying on string-based metrics for matching when dealing with units. Auxiliary dictionaries can be used to find such information, but covering all scientific areas would be unrealistic. Instead, having a text embedding model that produces dense embeddings that can be fine-tuned or completely replaced by another model provides flexibility to our system. Furthermore, the multilingual performance and the ability to represent synonyms, hypernyms, and canonical names in the embedding models can also be utilized. Additionally, averaging column and object embeddings can be used as auxiliary information in our influencing mechanism. Introducing an embedding model is beneficial for our system because it provides a more flexible and accurate way to represent entities and their relationships.
There are various approaches to construct node embeddings for node classification such as Instance Neighbouring by using Knowledge (INK) method [SVW+22], relational graph

convolution network [SKB+18], and rdf2vec [RP16]. The problem is that all of these approaches usually require training a new model. Since we do not have such resources and have time restrictions, utilizing a pre-trained word embedding model is ideal. In the research by Chen et al. [CHG+23], the authors utilized a pre-trained Bidirectional Encoder Representations from Transformers (BERT) [DCLT18] embedding model and fine-tuned it to classify ontology subsumption which is referred to as BERTSubs. They provided three different templates for name subsumption: isolated class, path context, and breadth-first context. The isolated class template does not consider class relationships, whereas the path context and breadth-first context templates do consider ancestors and descendants. Path context often outperforms all of the other templates and other state-of-the-art approaches in the inter-ontology named subsumption task [CHG+23]. The success of this template lies in the possibility of BERT to understand contextual information, such as graph paths. In more detail, BERT is a transformer architecture used in text encoding. Unlike the rest of the architectures, BERT can be trained on bidirectional representations by conditioning each token to both left and right contexts. This means the relationship between a vertex and its ancestors/descendants can be trained by representing the ancestors, the vertex itself, and the descendants bidirectionally.

BERT converts sequential text into tokens, and each token is then converted into an embedding vector. For this conversion from tokens to embeddings, the self-attention architecture [VSP+17] is used, rather than recurrence or convolution. In simple terms, self-attention converts each token, combined with its positional information, into three different vectors: query, key, and value. Attention scores for a token are computed by taking the dot product of its query vector with the key vectors of all other tokens. The final embedding (output) of a token is computed as a weighted sum of the value vectors, where the weights are the attention scores. Self-attention enables the model to effectively capture both local dependencies (by providing positional information) and global dependencies (by taking the dot product of key-query pairs) in the sequence.

Furthermore, the design of BERT provides symbols for pre-training or fine-tuning [DCLT18]. These symbols can be used to indicate a classification task ([CLS]), hiding some words for prediction ([MASK]) or separating distinct sentences ([SEP]). For the ontology subsumption [CHG+23], specifically the path context template, authors used a pre-trained BERT model and fine-tuned it to ontology subsumption downstream task between c_1 and c_2 by representing the descendent paths of c_1 and ancestor paths of c_2 as classes with "[SEP]" tokens (e.g., label of $class_1$ [SEP] label of $class_{1_A}$). They also tried using a custom token ([SUB]) rather than the [SEP] token. However, it gave worse results since the pre-trained model is already trained with [SEP] token. Furthermore, they used [CLS] token to indicate the classification task and added a classification layer (softmax layer) to output the probability of subsumption.

As previously mentioned, the BERTSubs path context template outperforms all competitors for inter-ontology named subsumption, achieving a Mean Reciprocal Rank of 0.707. However, the isolated class template, which relies solely on label names and annotation properties for synonyms, follows closely with a score of 0.695 [CHG+23]. Our approach

to calculating the similarity between (user-provided) column names and entity labels follows a methodology similar to the isolated class template of BERTSubs. However, in the Future Work Section 9.2, we will discuss the incorporation of annotations and a variation of the path context template.
Currently, there are various BERT-based pre-trained embedding models which produce dense embeddings. BGE M3-Embedding [CXZ$^+$24] has state-of-the-art performance with multilingual and cross-lingual support, and it's capable of processing different input granularities (from short inputs to passages) into embedding vectors. This means that a small or large amount of sentences can be fed into the model, allowing for the possibility of fine-tuning the model with sequential relationships without even changing the model itself. Overall this model is being used in our system for its performance in Massive Text Embedding Benchmark (MTEB) [MTMR22] Semantic Textual Similarity benchmark meanwhile concerning the memory usage, multilingual capabilities, and capability to represent various semantic relationships.

CHAPTER 3

System Overview

The following chapter will explore our design philosophy in constructing a semi-automated mapping tool. It will demonstrate how our overall ideas are shaped by the requirements of the DBRepo and the related work we have covered. Section 3.1 starts with what are the main objectives by giving reasons. Afterward, section 3.2 will demonstrate a high-level view of how the data flows from two different perspectives: when new ontologies are introduced and when the user initiates the mapping.

Our system is called Schema-ontology (SO) Mapper, which takes a relational schema and calculates the similarities between the schema and object ontologies, as well as schema and unit ontologies. The produced outputs are relevance lists, which can be searched and influenced by users. Administrators can introduce new ontologies as well as remove them from the system. SO Mapper will be used within the DBRepo table schema definition interface to aid users by semi-automating the process of mapping columns with object and unit entities.

3.1 Objectives

Seamless integration of SO Mapper into DBRepo is achieved by converting it into a stateful API. Since DBRepo contains multiple APIs running in docker containers (see Figure 2.1), following the same approach will ensure compatibility. Additionally, user states, which include initial mapping results, are cached into files for further usage in case clients want to interact with them.

One of the design philosophies of DBRepo is to make the data FAIR (Findable, Accessible, Interoperable, and Reusable) right from the beginning of the research, rather than addressing it later [wei22]. This is why SO Mapper is integrated into the table schema creation process (see Figure 2.2) while retaining the overall workflow. This is achieved by utilizing SO Mapper in the table schema definition step, as visualized in Figure 3.1.

The interface in the table schema step of DBRepo involves multiple functionalities such as adding and removing columns, and updating column names or data types. These functionalities are reflected in the SO Mapper. For example, when a user adds a new column, SO Mapper will include the new column in the schema information, calculate the data type satisfaction and name similarity, and re-calculate the overall ontology relevance scores.

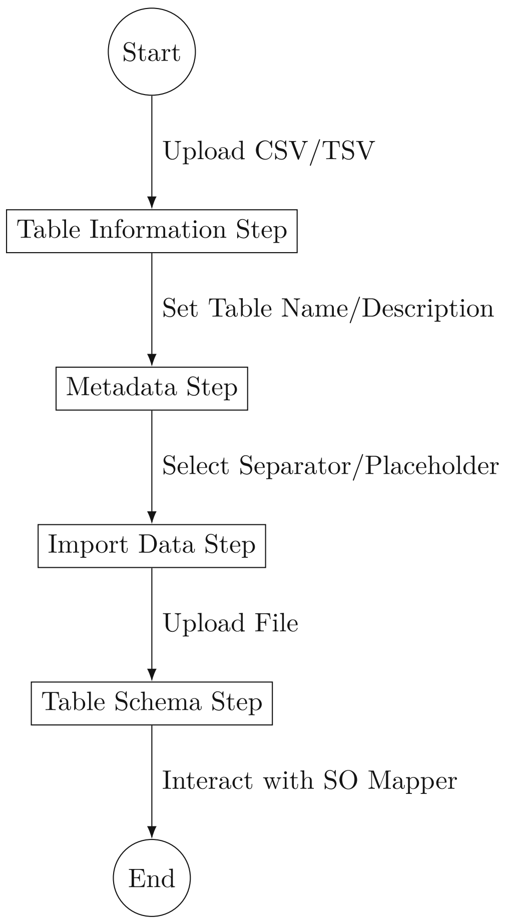


Figure 3.1: Workflow: DBRepo Table Schema Creation Process

Since DBRepo is a multi-user environment, SO Mapper must comply with it. This is achieved by introducing specific hashes for each user. For instance, there can be multiple users who are simultaneously working on the table schema creation process. When the front-end initiates the initial connection with SO Mapper, it receives a specific hash. Each user can use their hash to edit or influence their schema, which will be reflected individually to the SO Mapper. Note that these caches are thread-safe.

Our system contains two separate lists of ontologies, namely object and unit ontologies to make the underlying data FAIR. Administrators can introduce new ontologies to the system by simply putting the OWL2 files into specific file paths. Afterward, an Hypertext Transfer Protocol (HTTP) *load* request to the API needs to be sent. SO Mapper can load these new ontologies by identifying them, parsing the ontology data, encoding the display names, and finally appending them to their respective object or unit ontology list. Similarly, ontologies can be removed by deleting the files (deleting from file-system) and sending a removal request (deleting from memory). Furthermore, different ontologies can be parsed differently based on their specific modeling, such as using a reasoner or restricting the entity types. For example, the use of a reasoner is not needed for unit ontologies since some do not provide data types (e.g., units Vocabulary of QUDT [FAI24]) and others indirectly define data types through object properties (e.g., OM2). Additionally, the utilized unit ontologies define units as individuals which means parsing only the individuals is enough. All of these requests can be executed in run-time to give administrators flexibility in managing ontologies. Further information about the unit ontology data type definitions can be found in the Data Flow Section 3.2 and configuration options can be found in the Reading Configurations Section 4.1.

The back-end is designed to provide flexibility and be open to various approaches. Ontology information extraction can be configured to include only specific entity types or run the reasoner. The underlying embedding model can be changed efficiently, and new encoding methods can be introduced. Encoding methods for object ontologies, unit ontologies, and schema ontologies can be separated. Lastly, different scoring methods can be introduced for calculating the similarity scores, and object ontologies and unit ontologies can utilize different scoring methods.

Introducing semi-automation for mapping requires utilizing user feedback. In SO mapper, such feedback is handled by using a small number of user interactions. Ultimately will ease the usability by reducing the user effort and also increasing the quality of results. After the Initial Mapping finishes, SO Mapper front-end automatically maps each column to the most similar object and unit entity among all ontologies. The most suitable object and unit ontology is also automatically selected. In case the mappings are not perfect, users can select a specific object ontology to use. Then each column will be automatically mapped to the most relevant entities from the selected ontology. Furthermore, this selection will be used as auxiliary information and affect the similarities of unit entities. We call this *indirect* influence. However, users can also select an object entity which will change the similarities of unit entities. This enhancement is called *direct* influence.

3.2 Architecture & Data Flow

The SO Mapper comprises two main data flows. When administrators provide an OWL file that contains an object ontology O_{obj_i} or a unit ontology O_{unit_i}, reading configurations $Conf_{obj/unit_*}$ for object or unit ontologies and then trigger our loading process, our system locates the file and sends it to the ontology parsing process. The Entity Discovery and

Extraction (*discovery_extraction*) algorithm extracts ontology information $OI_{obj/unit_i}$ which can be defined as:

$$discovery\_extraction(O_{obj/unit_i}, Conf_{object,unit_i}) = OI_{obj/unit_i} = \{ei_1, ..., ei_m\} \tag{3.1}$$

Where m is the number of entities within the ontology $O_{obj/unit_i}$ and ei_y is the y'th entity information that contains relationships and ranges (e.g., data type, subclass), IRI, and label name (also data types in case an object ontology is provided). Our parsing algorithms can be seen in Chapter 4.

The ontology information $OI_{obj/unit_i}$ is then forwarded to the encoding process, where an encoding method *enc* returns string str_y for each entity information ei_y (e.g., label names of the entities can be returned). Subsequently, a text embedding model *emb* converts the strings into dense vector embeddings $E_{obj/unit_i}$.

$$\begin{aligned}
&E_{obj/unit_i} = \{v_1, ..., v_m\} \text{ , where} \\
&v_y = emb(enc(ei_y))\ y \in \{1, .., m\} \\
&enc(ei_y) = str_y = \begin{cases} label(ei_y), & \text{if } e_y \text{ belongs to an object ontology} \\ label(ei_y) + \text{" unit"}, & \text{if } e_y \text{ belongs to an unit ontology} \end{cases}
\end{aligned} \tag{3.2}$$

Lastly, our system generates logs to track the parsing and encoding processes for future reference and analysis. An illustration of such flow can be seen in Figure 3.2. Further information about the components is given in the Implementation Chapter 7.

Another crucial aspect of understanding the architecture is exploring the user-initiated data flow, which involves examining the processes behind the architecture that SO utilizes. This flow, called Initial Mapping, begins with the user uploading a relational table as a CSV or TSV file to DBRepo. The DBRepo analysis service then processes the given schema columns and suggests data types $D = \{d_1, ..., d_n\}$ for each column by examining the data instances. The SO front-end map initialization processes the schema columns and data types, excluding certain data types from the unit mapping (e.g., x'th column c_x is excluded for the unit mapping if d_x is a string). Subsequently, the column names $C = \{c_1, ..., c_n\}$ and data types are sent to the SO mapper to represent the user schema data in the back-end.

Afterward, the encoding process starts which utilizes the same encoding method *enc* and the text embedding model *emb* converts each column name C into a set of dense vector embeddings E_C.

$$\begin{aligned}
&E_C = \{v_1, ..., v_n\} \text{ s.t.} \\
&v_x = emb(enc(c_x))\ x \in \{1, ..., n\}, \\
&enc(c_x) = label(c_x)
\end{aligned} \tag{3.3}$$

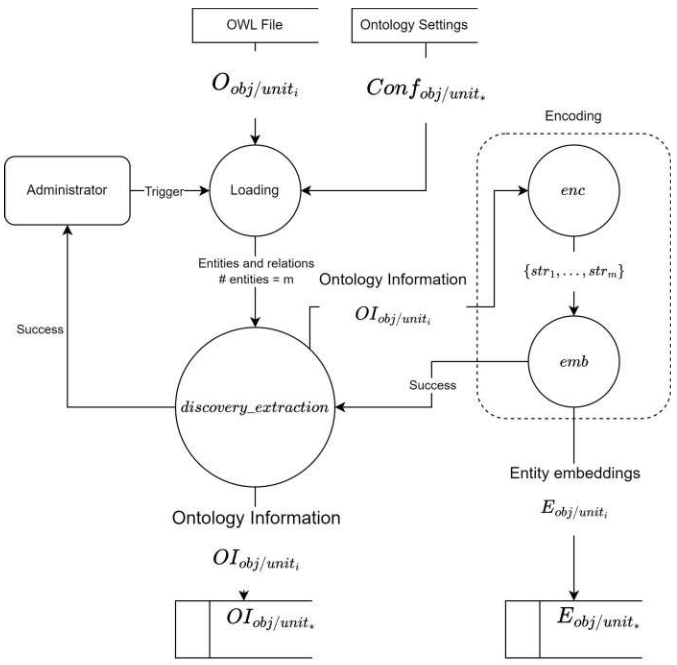


Figure 3.2: Data flow of new ontologies in SO Mapper

At this point, our system employs multi-threading for object and unit mapping. Note that for the sake of simplicity, the asterisk (*) symbol will be used to indicate all of the object or unit ontologies. First, Fields $F^{C,O_{obj/unit*}}$ are generated between the schema columns and all the ontologies ($\forall OI_i \in OI_*$). They can be defined as:

$$F^{C,O_{obj/unit*}} = \{F^{C,O_{obj/unit_1}}, \dots\}$$
$$F^{C,O_{obj/unit_i}} = \begin{cases} \{C, D, OI_{obj_i}, ST^{C,obj_i}, CS^{C,obj_i}\}, & \text{if } O_i \text{ is object ontology} \\ \{C, D, OI_{unit_i}, ST^{C,unit_i}\}, & \text{otherwise} \end{cases} \tag{3.4}$$

More information about the Fields can be found in Section 5.1. Fields are used to store following data structures:

- *SourceTarget matrix* $ST^{C,obj/unit_i}$: Holds column names C and entity labels in OI_i. It is initially defined as:

$$\begin{aligned} &ST^{C,obj/unit_i}_{x,y} = (name(c_x), label(ei\_y)) \ \forall x \in \{1, \dots, n\} \ \forall y \in \{1, \dots, m\}, where \\ &n \text{ is the number of columns in } C, \\ &m \text{ is the number of entities in } OI_{obj/unit_i} \end{aligned} \tag{3.5}$$

- *Constraint Satisfaction matrix* CS^{C,obj_i}: Holds the data type compatibility results between the columns and entities within the ST^{C,obj_i}. It can be defined as as:

$$\begin{aligned}
&DataConstSat(F^{C,obj_i}) = CS^{C,obj_i} \text{ , where}\\
&CS^{C,obj_i}_{x,y} = cs_{x,y} \ \forall x \ \forall y \in ST^{C,obj_i} \text{ and}\\
&cs_{x,y} = \begin{cases} true, & \text{if } d_x \in GetDataTypes(ei_y) \\ false, & \text{otherwise} \end{cases}
\end{aligned} \tag{3.6}$$

During the Field generation, the SourceTarget matrix gets created, and then constraint satisfaction *constraint_sat* starts. Only the object ontologies are checked for data type compatibility since unit ontologies do not provide any data types such as Units Vocabulary of QUDT [FAI24]. Meanwhile in some other unit ontologies, such as OM2 [HKP+09], indirectly define data types for certain units such as prefixed units (e.g., "centigram") through object properties (e.g., "hasUnit") of other classes (e.g., base units such as "gram"), which requires a specific algorithm for each new unit ontology. This restricts the systems' flexibility when new unit ontologies are provided, so using only the data type definitions in the object ontology is chosen to give visual information to the users.

When the Fields between S and all ontologies $OI_{obj/unit_i}$ are ready (SourceTarget and Constraint Satisfaction matrix are computed), Similarity Calculation (*sim_calc*) occurs for the column-entity pairs in the SourceTarget matrices $ST^{C,obj/unit_*}$ which returns similarity scores $R^{C,obj/unit_*}$ between column names and entities from ontologies. Again, note that an asterisk (*) symbol will be used to indicate all of the object or unit ontologies.

$$\begin{aligned}
&R^{C,obj/unit_i} \in R^{C,obj/unit_*}, \text{ where}\\
&R^{C,obj/unit_i} = sim\_calc(F^{C,obj/unit_i}, E_C, E_{obj/unit_i})\\
&R^{C,obj/unit_i}_{x,y} = cosine\_sim(v_x, v_y) \ \forall x \ \forall y \in \ ST^{C,obj/unit_i}
\end{aligned} \tag{3.7}$$

Our data type comparison and similarity calculation algorithms can be seen in Chapter 5. New similarity measurements (such as changing *sim_calc*, *enc*, *emb*) can be introduced to our system.

Once both unit and object processes are completed object and unit similarity scores $R^{C,obj/unit_*}$, Fields $F^{C,obj/unit_*}$, and schema column embeddings E_C are stored in a cache. Our system utilizes such caching to handle user interactions. For instance, one user interaction would be *changing the data type of a column* which changes the initial constraint satisfaction results within the Fields. All the supported user interactions can be seen in User Interactions Chapter 6.

Afterward, using the column names C, Fields $F^{C,obj/unit_*}$, and similarity scores $R^{C,obj/unit_*}$ two types of relevance list are created for the Initial Mapping:

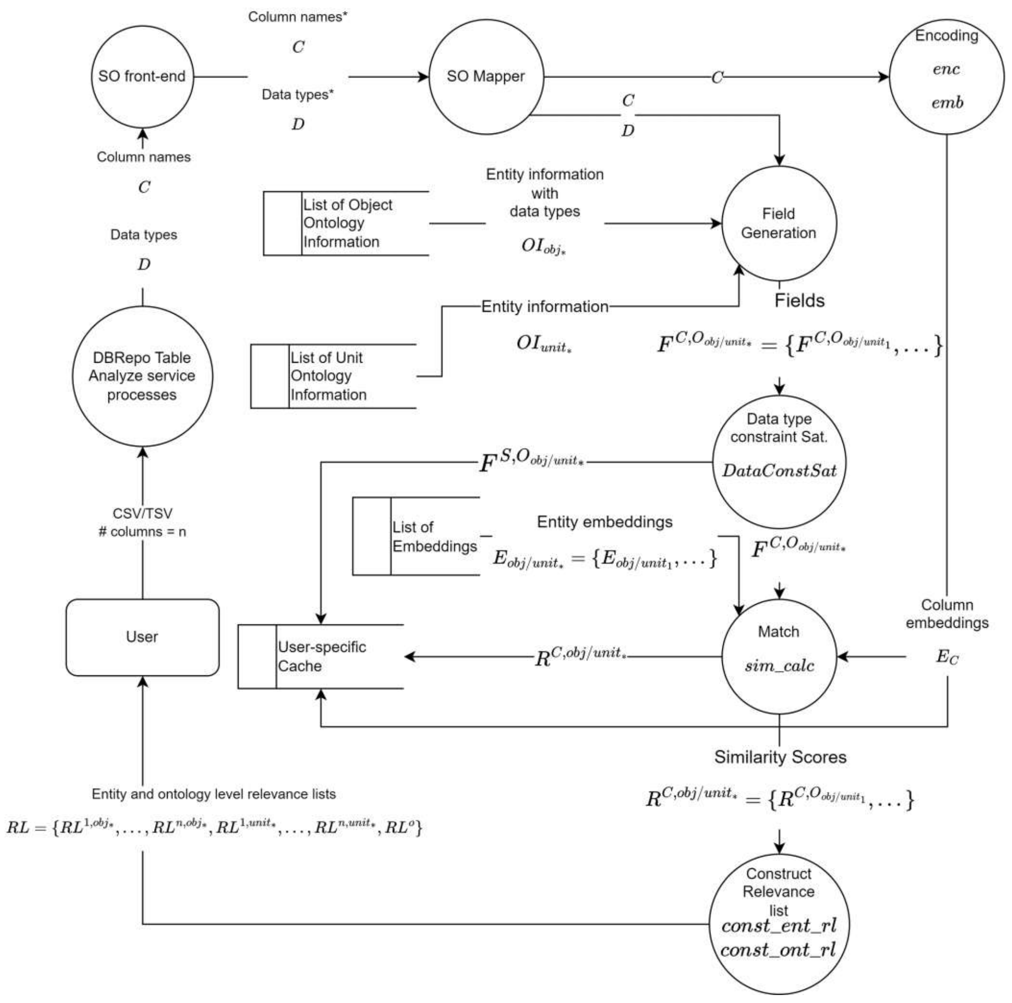

Column names*
C
Data types*
D
SO front-end
SO Mapper
C
C
D
Encoding
enc
emb
Column names
C
Data types
D
List of Object Ontology Information
Entity information with data types
OI_{obj_*}
Field Generation
Entity information
OI_{unit_*}
Fields
$F^{C,O_{obj/unit_*}} = \{F^{C,O_{obj/unit_1}}, \dots\}$
List of Unit Ontology Information
DBRepo Table Analyze service processes
Data type constraint Sat.
$DataConstSat$
$F^{S,O_{obj/unit_*}}$
List of Embeddings
Entity embeddings
$E_{obj/unit_*} = \{E_{obj/unit_1}, \dots\}$
$F^{C,O_{obj/unit_*}}$
CSV/TSV
# columns = n
User
User-specific Cache
$R^{C,obj/unit_*}$
Match
sim_calc
Column embeddings
E_C
Similarity Scores
$R^{C,obj/unit_*} = \{R^{C,O_{obj/unit_1}}, \dots\}$
Entity and ontology level relevance lists
$RL = \{RL^{1,obj_*}, \dots, RL^{n,obj_*}, RL^{1,unit_*}, \dots, RL^{n,unit_*}, RL^{o}\}$
Construct Relevance list
$const\_ent\_rl$
$const\_ont\_rl$

Figure 3.3: Data flow of schemas in SO Mapper Initial Mapping

- *Entity level relevance lists*: $\forall x \in C \quad const\_ent\_rl(x, F^{C,obj/unit_*}, R^{C,obj/unit_*}) = RL^{x,obj/unit_*}$
- *Ontology level relevance list*: $const\_ont\_rl(F^{C,obj/unit_*}, R^{C,obj/unit_*}) = RL^{o}$

Further information about the relevance lists can be seen in Section 5.3.

Finally, the relevance lists are sent back to the SO front-end and displayed to the user. This data flow is also illustrated in Figure 3.3.

CHAPTER 4

Reading the inputs

The following chapter explains how the SO mapping tool extracts and processes information from the provided schemas and ontologies. This task is crucial for our system because our data type checking, similarity metrics, and the entire mapping process rely on it. The first section 4.1 will discuss the customization of reading and representing ontologies. The second section 4.2 will explore our system's capabilities in extracting entity information, data types, and relationships. In the last section 4.3, we will discuss how text embedding models encode entities of ontologies and columns of user-provided schema data.

4.1 Reading configurations

Our system provides the following configuration capabilities:

- *Entity Types*: Parsing individuals, classes, objects, or data properties. By default, all of the entity types are parsed. This functionality is important because, in some ontologies, targets can be certain entity types. This is particularly common in unit ontologies. One such example is OM2 [HKP$^+$09] or QUDT. These ontologies represent units only as individuals. In OM2 and the QUDT Units Vocabulary [FAI24], no domain information about the units is provided. For instance, "biology" information is not provided for the Colony-forming unit. For such cases, our system provides a configuration option for what to include when extracting information from the ontologies. It allows administrators to specify entity types depending on the ontology structure. Our system currently uses this feature to extract only individuals from unit ontologies, while object ontologies extract all entity types.
- *Reasoner*: Enabling or disabling the Pellet [SPG$^+$07] reasoner. By default, the reasoner is enabled. A reasoner can deduce anonymous ancestor relationships

which can only be inferred through the axioms of ontology. This functionality can be applied to both object and unit ontologies. However, our system mainly utilizes anonymous ancestor relationships to extract data types in object ontologies rather than unit ontologies. Since the utilized unit ontologies do not provide data type information or indirectly define data types and represent units as individuals, parsing anonymous ancestor relationships is unnecessary. More information about the data types in unit ontologies can be found in Data Flow Section 3.2.

Configuring the entity types and reasoner can be advantageous when administrators want to reduce the entity types for the search space (e.g., parsing only the individual entity types) or reduce computational demand by ignoring additional data type constraints (e.g., disabling the reasoner).

- *Encoding Method enc*: Changing the generation of strings (e.g., using label names, relationships, etc.) that are used to represent entities.Default encoding methods are described in Section 4.3. In our system, various encoding approaches can be utilized, such as encoding labels of classes and super-classes into the same embedding representation.

- *Embedding Model*: Changing the text embedding model that converts strings that represent entities into vector embeddings. The default embedding model is given in Section 4.3.

  The goal of enabling the encoding method and embedding model configuration is to ensure SO mapper can incorporate advancements in encoding methodologies and use different text embedding models.

Entity type and reasoner configurations can be applied on the run-time of our system to customize reading new object or unit ontology $O_{obj/unit_i}$. For this, a JavaScript Object Notation (JSON) literal $Conf_{obj/unit_i}$ must be added into the object or unit configuration file $Conf_{obj/unit_*}$.

$$\begin{aligned}&Conf_{obj/unit_i} = i : \{\text{include:}inc, \text{use\_reasoner:}ur\} \text{ s.t.}\\&\text{name of the ontology } i\text{, list of unit types } inc\text{, and boolean } ur\end{aligned} \tag{4.1}$$

On the other hand, the encoding method *enc* and embedding model *emb* require restarting the system to compute all the embedding vectors. There are currently two encoding methods and one embedding model in our system which are addressed in Section 4.3. Administrators can use our components to introduce new embedding methods and encoding models. Our components can be found in Implementation Chapter 7.

4.2 Ontology Parsing

SO mapper parses ontologies for extracting the IRIs, display names, and data types for the constraint satisfaction step. Parsing can be customized as shown in section 4.1. Our

system relies on OwlReady2 to discover range and equivalence information. OwlReady2 is a newer version of the library OwlReady [Lam17] which supports OWL [WMS04] and OWL2 [mot12][GWPS09]. Further information about OwlReady2 is given in the Implementation chapter (see Chapter 7).
Our discovery and extraction algorithm takes an ontology O_i and returns the ontology information OI_i. Initially, the discovery part starts to find entities, relationships, equivalence definitions and ranges with or without using the reasoner. In each iteration, when a new entity e_y is discovered, entity information ei_y is created which contains IRI of the entity and label name. The extraction part (*ExtractRange* and *ExtractEquivalence*) takes place if entity e_y includes a range, super-class, or indirect equivalence definition. In such cases, enumerated classes (e.g., oneOf) logical constructs (e.g., A or B), class, and property restrictions (e.g., cardinality, quantification restrictions) are stripped away from the ranges and definitions. Finally, the ranges and relationships with only IRIs or data types are returned to fill ei_y in OI_i.

Figure 4.1 shows a small portion of three separate ontologies. Entities of the ontologies can be seen in Table 4.1, 4.2, and 4.3. Our discovery algorithm will detect all of these entities.

Class	Object property	Data property
Building	containsArea	hasAverageNumberOfFloorsValue
Area	containsSurface	hasSurfaceTypeValue
Surface	containsRectangularGeometry	hasNativeValue
RectangularGeometry	containsHeight	hasUnitValue
Height		
Custom Data type		Rdf schema data type
surfaceTypeEnum		Ceiling
		ExteriorWall

Table 4.1: Portion of entities in Building Information [KIVK13]

Class	Object property
business entity	fuel type
organization	
person	
engine specification	
quantitative value	
text value	

Table 4.2: Portion of entities in Vehicle Core (VC) [veh21]

Class	individual
Air Quality Property	benceno
	dioxidoDeAzufre

Table 4.3: Portion of entities in Calidad-aire [KRK12]

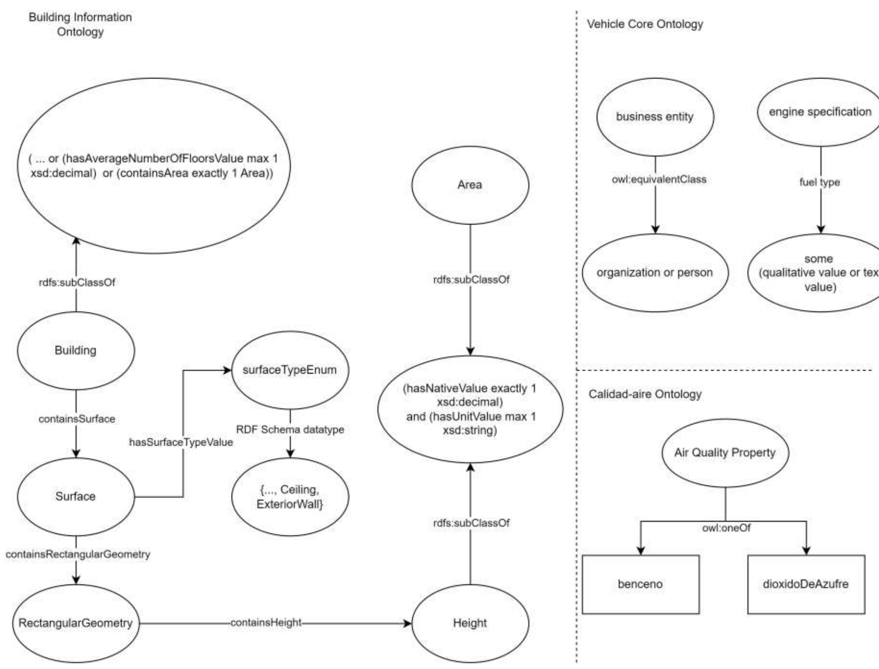


Figure 4.1: Small portion of the Building Information [KIVK13], VC [veh21], and Calidad-aire [KRK12] ontologies as object ontology examples

This discovery is carried out in the following cases:

- *Individuals*: Discover the IRI of each individual and the corresponding classes it belongs to (e.g., "benceno" has a class definition of "Air Quality Property" in the Calidad-aire ontology).
- *Classes*: Discover the IRI of each class, along with their super-class relationships, equivalence definitions, and equivalence definitions from anonymous ancestors (e.g., "Height" class is a subclass of "(hasNativeValue exactly 1 xsd:decimal) and (hasUnitValue max 1 xsd:string)" in building information ontology).
- *Object Properties*: Discover the IRI of each object property and the range classes associated with it (e.g., the "containsSurface" object property has a definition related to the "RectangularGeometry" class in building information ontology).
- *Data Properties*: Discover the IRI of each data property and the range of data types it is associated with (e.g., the "hasSurfaceTypeValue" data property has a definition related to the "surfaceTypeEnum" data type in building information ontology).
- *Data Types*: Discover the IRI of each datatype and their respective definitions (e.g., "surfaceEnum" includes definitions such as "Ceiling" and "ExteriorWal").

Our entity discovery and extraction process is outlined in Algorithm 4.1. OwlReady2 provides several functions to fetch information from the ontology such as:

- *GET_CLS* fetches all classes.
- *GET_OP* fetches all object properties.
- *GET_DP* fetches all data properties.
- *GET_IRI* takes an entity and returns the IRI of it.
- *GET_CLASSES* takes an individual and returns fetches classes of it.
- *GET_ALL_EQV* takes a class and fetches equivalencies (including anonymous ancestors).
- *GET_SUBCLS* takes a class and returns descendants of it.
- *GET_RANGES* takes an entity and returns ranges of it.
- *TYPE* fetches super-classes of a class or classes of an individual.

Furthermore, we introduced the *SparqlFetchDatatypes* algorithm to gather all data type definitions. It queries the RDF triples through OwlReady2. Output is being retrieved row by row. The executed SPARQL query is as follows:

```
SELECT ?dataproperty ?datatype
WHERE {
    ?dataproperty rdf:type owl:DatatypeProperty.
    ?dataproperty rdfs:range ?range.
    ?range owl:equivalentClass ?datatype.
}
```

The discovery part of the algorithm ensures that all relevant information about entities is collected from an OWL2 ontology. However, an extraction is necessary because object properties, data properties, and data types can contain complex ranges. Such cases are handled with the *ExtractRange* algorithm. Additionally, class equivalences and super-class definitions can contain complex definitions using the object and data properties. They are handled with the *ExtractEquivalence* algorithm.

For instance, while parsing the example portion of VC ontology, since object property "fuel type" has a range that is a logical combination, IRIs from the range will be extracted using *ExtractRange*. Similarly, *ExtractEquivalence* will extract the IRIs from the ("subClassOf" or "equivalentClass") definitions of "Building", "Height", and "Area" classes in Building Information ontology, the "business entity" class in VC ontology, and the "Air Quality Property" class in Calidad-aire ontology.

Algorithm 4.1: Entity Discovery and Extraction (*discovery_extraction*)

Input : *ontology*: Ontology O_i that is initially parsed by OwlReady2 (with or without the reasoner).

Input : *conf*: Configuration $Conf_{object,unit_i}$ that determines which entity types to discover. Options are Individual, Class, Objects, or Data properties.

Output : *ontologyData*: Ontology information data OI_i. Contains all IRIS, Class-Individual (CI), Class-Subclass (CS), Class-Data type (CD), Object property-Class (OC), Data property-Data type (DD), and Data type-Range (DR) relationships.

```
1  ontologyData ← []
2  if individuals ∈ conf.EntityTypes then
3      for individual IN GET_IND(ontology) do
4          ontologyData.CI ← GET_IRI(individual), GET_CLASSES(individual)
5      end
6  end
7  if classes ∈ conf.EntityTypes then
8      for class IN GET_CLS(ontology) do
9          identifiedEqv ← []
10         for equivalence IN GET_ALL_EQV(class) do
11             property, range ← ExtractEquivalence(equivalence)
12             identifiedEqv.APPEND(property,ExtractRange(range))
13         end
14         ontologyData.CS[GET_IRI(class)] ← GET_SUBCLS(class) ∪
            identifiedEqv.subCls
15         ontologyData.OC ← identifiedEqv.objectProperty
16         ontologyData.CD ← identifiedEqv.dataProperty
17     end
18 end
19 if objectProperties ∈ conf.EntityTypes then
20     for objectProperty IN GET_OP(ontology) do
21         identifiedRanges ← []
22         for range IN GET_RANGES(objectProperty) do
23             identifiedRanges.APPEND(ExtractRange(range))
24         end
25         ontologyData.OC[GET_IRI(objectProperty)] ← identifiedRanges
26     end
27 end
28 if dataProperties ∈ conf.EntityTypes then
29     for dataProperty IN GET_DP(ontology) do
30         identifiedRanges ← []
31         for range IN GET_RANGES(dataProperty) do
32             identifiedRanges.APPEND(ExtractRange(range))
33         end
34         ontologyData.DD[GET_IRI(dataProperty)] ← identifiedRanges
35     end
36     for dataType IN SparqlFetchDatatypes(ontology) do
37         ontologyData.DR[dataType] ← ExtractRange(dataType)
38     end
39 end
40 return ontologyData
```

The first extraction algorithm *ExtractRange* is used to extract the following IRIs/data types for complex ranges object properties, data properties, and data types:

- *Enumeration of Literals/Individuals* (owl:OneOf): Each IRIs in the "OneOf" definition can be extracted.
- *Classes* (owl:Class): Class IRIs can be extracted from object properties.
- *Data types*: XML Schema Definition (XSD) data types are converted into python data types (int, float, bool, str, datetime.date, etc.) implicitly using Owlready2.
- *Constrained data types* (ConstrainedDataType): Data types can be extracted within their constraints such as "xsd:float[>= 180.0f]".
- *RDF Schema [BG14] data types* (rdf-schema.Datatype): Custom enumerated data types can be extracted.
- *Cardinality restrictions* (owl:Restriction): IRIs in the "min", "max", etc. restrictions can be extracted.
- *Quantification* (owl:Restriction): IRIs in the universal ("all") and existential ("some") quantification can be extracted.
- *Logical combinations* (LogicalClassConstruct): IRIs in the nested logical expressions can be extracted.

Note that complement definitions (owl:inverseOf) are provided (discovered) implicitly by Owlready2 when the reasoner is running.

Since ranges can contain nested logical combinations, extracting them requires recursion. For instance, in the VC ontology, "engine specification" has an object property "fuel type" with the complex range "some (qualitative value or text value)". OwlReady2 treats such input as a whole, named "owl:Restriction". When *ExtractRange* encounters this complex range, it initially removes the quantification, leaving "(qualitative value or text value)", and recursively calls itself. During the recursion, since the current input is a *LogicalClassConstruct*, the logical connectives are stripped away, leaving two parts: "qualitative value" and "text value". The algorithm will recursively call itself for each part (owl:Class) to extract the IRIs. Once the algorithm completes, "fuel type" is marked as compatible with the data types of "qualitative value" and "text value" classes. Note that, in our example, the data types are not specified for these classes for the sake of simplicity.

The *ExtractRange* algorithm can be seen in Algorithm 4.2. Line 1 detects whether the incoming data is a XSD data type (e.g., "xsd:decimal"). Line 4 detects whether the incoming range information is an enumerated data type from RDF Schema (e.g., "Ceiling" and "ExteriorWall" in Building information ontology). Constrained data types (e.g., "xsd:float[>= 180.0f]") are detected in line 7. Line 10 detects the enumeration of literals (e.g., assume "someObjectProperty {benceno, dioxidoDeAzufre}" in Calidad-aire ontology). Line 17 detects whether the incoming construct is a class or not. Additionally, cardinality restrictions (e.g., "containsArea exactly 1 Area"), existential quantification, and universal quantification (e.g., "some (qualitative value or text value)") are covered in line 20. Since object and unit properties can contain complex definitions with logical operators, this is covered in line 23.

Algorithm *ExtractEquivalence* can extract the property-complex range pairs from complex class equivalence and super-class relationship definitions. Extraction will occur for both classes and their anonymous ancestors. Such complex definitions can be the combination of properties using the following:

- *Enumeration of Individuals* (owl:OneOf)
- *Classes* (owl:Class)
- *Cardinality restrictions* (owl:Restriction)
- *Quantifications* (owl:Restriction)
- *Logical combinations* (LogicalClassConstruct)

Since properties in the equivalence/superclass definitions can contain logical combinations, extracting them requires recursion. There are four extraction examples from the ontologies in Figure 4.1):

- The "Area" and "Height" classes in the Building Information Ontology are subclasses of "(hasNativeValue exactly 1 xsd:decimal) and (hasUnitValue max 1 xsd:string)". OwlReady2 treats this complex definition as *LogicalClassConstruct*. The *ExtractEquivalence* algorithm will be called twice for both inputs. In each call, the logical connectives are stripped away, leaving two parts: "hasNativeValue exactly 1 xsd:decimal" and "hasUnitValue max 1 xsd:string", and then recursively calls itself for each part. Since the inputs are now an "owl:Restriction", the recursions will return "(hasNativeValue, decimal)" and "(hasUnitValue, string)". Once algorithms are complete, the "Area" and "Height" classes are marked as compatible with "decimal" and "string." Furthermore, the data properties "hasNativeValue" and "hasUnitValue" are marked as compatible with "decimal" and "string" respectively.
- The "Building" class in the Building information ontology is a subclass of "(... or (hasAverageNumberOfFloorsValue max 1 xsd:decimal) or (containsArea exactly 1 Area))". OwlReady2 treats this complex definition as "LogicalClassConstruct". When *ExtractEquivalence* encounters this complex definition, logical connectives will be removed, leaving two parts: "(hasAverageNumberOfFloorsValue max 1 xsd:decimal)" and "(containsArea exactly 1 Area)". The algorithm will recursively call itself for each part. Since the inputs are "owl:Restriction", the recursions will

Algorithm 4.2: Extract Range

Input : *construct*: Target(s) of the range. It can be from object property, data property, and data type.

Output : *List of IRIs*: These IRIs can be classes, data types, and data type definitions.

```
if construct IS XSD data type then
    return [GET_IRI(construct)]
end
else if construct IS rdf-schema.Datatype then
    return [GET_IRI(construct)]
end
else if construct IS ConstrainedDatatype then
    return [GET_IRI(BASE_DATATYPE(construct))]
end
else if construct IS owl:OneOf then
    tmp ← []
    for instance IN construct.instances do
        tmp.APPEND(GET_IRI(instance))
    end
    return tmp
end
else if construct IS owl:Class then
    return [GET_IRI(construct)]
end
else if construct IS owl:Restriction then
    return [GET_IRI(construct.value)]
end
else if construct IS LogicalClassConstruct then
    tmp ← []
    for part IN SPLIT(construct) do
        tmp ← CONCAT(tmp, ExtractRange(part))
    end
    return tmp
end
else
    return []
end
```

return "(hasAverageNumberOfFloorsValue, decimal)" and "(containsArea, Area)". Once the algorithm completes, the "Building" class and "hasAverageNumberOfFloorsValue" data property will be marked as compatible with the "decimal" data type. Furthermore, the "containsArea" object property will be marked as compatible with the data types of the "Area" class (in our case, "decimal" and "string"). The reasoning behind this is that when a column is mapped to an object property ("containsArea") it is mapped to one of the ranges of that object property (such as "Area").

- The "business entity" class in the VC ontology is equivalent to "organization or person". Once again, the OwlReady2 treats this definition as "LogicalClassConstruct". *ExtractEquivalence* will remove the logical connectives, leaving the "organization" and "person" parts. Two recursions will occur for these parts that return "(super-class of organization, organization)" and "(super-class of person, person)" classes. Note that since these classes have no object property, their super-classes are returned but not utilized in our algorithms. Once the *ExtractEquivalence* algorithm is completed, "business entity" is marked as compatible with the data types of "organization" and "person" (data types are not specified for these classes for the sake of simplicity).

- The "Air Quality Property" in Calidad-aire ontology has the instances of "{benceno, dioxidoDeAzufre}". OwlReady2 treats the definition as $OneOf$ thus *ExtractEquivalence* will return "[(Air Quality Property, benceno), (Air Quality Property, dioxidoDeAzufre)]". Once the algorithm is completed, "benceno" and "dioxidoDeAzufre" are marked as compatible with the compatible data types of "Air Quality Property".

ExtractEquivalence can be seen in Algorithm 4.3. Line 1 detects cardinality restrictions, existential quantification, and universal quantification. Furthermore, it uses recursion to extract property-value pairs from nested object properties. Line 9 detects the logical class constructs, removes the logical connectives, and uses recursion to extract property-value pairs. Line 16 detects class relationships that do not have the object property relationship and returns superclass-class pairs. Line 20 detects instance definitions and returns a superclass of individual-individual pairs.

Even though *ExtractEquivalence* can extract properties from complex definitions, their ranges can be complex. For example, when "objectProperty only (xsd:int and xsd:float)" is provided as an input, *ExtractEquivalence* will return "objectProperty"-"(xsd:int and xsd:float)" pair where the range is still complex (contains a logical combination). This is why when the *ExtractEquivalence* is completed, ranges of the output are provided to the *ExtractRange* algorithm as input (see line 12 in algorithm 4.1).

On completion of *ExtractRange* and *ExtractEquivalence*, entity labels and ranges from the properties, equivalences, and super-classes are stored in 2-dimensional dictionaries with their respective IRIs as follows (also see algorithm 4.1):

Algorithm 4.3: Extract Equivalence

```
Input  : classConstruct: Equivalent definition of a class.
Output : List of property-value pairs: A list of properties and their value pairs
         are extracted from a definition.
if classConstruct IS owl:Restriction then
    return [(classConstruct.property, classConstruct.value)]
end
else if classConstruct IS LogicalClassConstruct then
    tmp ← []
    for part IN classConstruct.Split do
        tmp ← CONCAT(tmp, ExtractEquivalence(part))
    end
    return tmp
end
else if classConstruct IS owl:Class then
    return [(TYPE(classConstruct), classConstruct)]
end
else if classConstruct IS OneOf then
    tmp ← []
    for instance IN classConstruct.instances do
        tmp.APPEND((TYPE(classConstruct), instance))
    end
    return tmp
end
```

- *Class-Individual (CI)*: Holds the class and individual relations.
- *Class-Subclass (CS)*: Holds the class and subclass relations.
- *Class-Data type (CD)*: Holds the class and all possible data types.
- *Object property-Class (OC)*: Holds the object property and class relations.
- *Data property-Data type (DD)*: Holds the data property and data type relations.
- *Data type-Range (DR)*: Holds the custom data type (enum) and value relations.

Additionally, our system employs two additional dictionaries. The first one is used to hold the type of the entities (e.g. Table 4.1, 4.2, and 4.3). The second one is used to convert IRIs to label names and vice versa.

For the ontologies in Figure 4.1, Ontology information data (OI_i) are as follows:

- Building information ontology $OI_{building}$:
  - *Class-Data type (CD)* = {Building ← {decimal}, Area and Height ← {decimal, string}}
  - *Object property-Class (OC)* = {containsArea ← {Area}, containsSurface ← {Surface}, containsRectangularGeometry ← {RectangularGeomerty}, containsHeight ← {Height}}
  - *Data property-Data type (DD)*= { hasAverageNumberOfFloorsValue ← {decimal}, hasSurfaceTypeValue ← {surfaceTypeEnum}, hasNativeValue ← {decimal}, hasUnitValue ← {string}}
  - *Data type-Range (DR)* = {surfaceTypeEnum ← {Ceiling, ExteriorWall}}

  Where the Class-Individual (CI) and Class-Subclass (CS) dictionaries are empty.

- VC ontology OI_{vc} :
  - *Object property-Class (OC)* = {fuel type ← {qualitative value, text value}}
  - *Class-Subclass (CS)* = {business entity ← {organization, person}}

  Where the Class-Data type (CD), Data property-Data type (DD), and Class-Individual (CI) dictionaries are empty.

- Calidad-aire ontology $OI_{calidad}$:
  - *Class-Individual (CI)* = {Air Quality Property ← {benceno, dioxidoDeAzufre}}

  Where the Class-Data type (CD), Data property-Data type (DD), Object property-Class (OC), and Class-Subclass (CS) dictionaries are empty.

In summary, the SO mapper utilizes Extract Range (Algorithm 4.2) to remove complexities (cardinality restrictions, constrained data types, quantifications, and logical combinations) from ranges and Extract Equivalence (Algorithm 4.3) to remove complexities between the properties. These two algorithms are used inside Entity Discovery and Extraction *discovery_extraction* (Algorithm 4.1) which ultimately extracts the data type values of all entities. Classes will have all possible data types. Data properties will use their ranges as data types. Object properties will use the data types of all their relatable classes. Individuals will use the data type of their corresponding classes.

4.3 Embeddings

Computing embeddings is the final step before fully representing the inputs in the SO mapper back-end. User-provided columns C directly reach this step without needing parsing. Similarly, the parsed ontology information data OI_i also progresses to this step. Our encoding method *enc* takes all column names c_x in C or all entity information ei_y in

OI_i and returns the column names or entity labels as strings (str_x or str_y). However, if the entities belong to a unit ontology, we construct the strings as $str_y = label(ei_y) +$ " unit". This approach is influenced by prompt engineering and has improved the results in the test datasets (see Chapter 8).

When the strings are ready, our embedding model *emb* takes them as input and computes the embedding vectors. Column vectors E_C are stored in a user-specific cache whereas entity vectors $E_{obj/unit_i}$ are stored in the file system. Furthermore, entity vectors are saved in Hierarchical Data Format version 5 (HDF5) format [The]. This file format offers greater efficiency when reading large amounts of data [FHK+11] and provides flexibility for adding more information (e.g., introducing new entities) into the same file.

SO mapper can switch between encoding methods *enc* and embedding models *emb* while maintaining the saved embeddings from other methods or models. For instance, if the text embedding model *emb* is utilizing a BERT-based transformer architecture [DCLT18] to transform bi-directional textual information into numerical representations, ontology information data OI_x can provide ancestor information for each class and individual. This auxiliary information can be combined with the entity labels to include the representation of class relationship information in the strings. However, this method would require fine-tuning the utilized embedding model to understand these representations.

Moreover, different encoding methods for schema, object, or unit ontologies can be used. These options are introduced to enable the SO mapper to evolve and be open to various approaches. Such flexibility ultimately leads administrators to experiment with different encoding strategies tailored to the specific characteristics of ontological data.

SO mapper currently utilizes BGE M3-Embedding [CXZ+24] as a text embedding model *emb*. The reason for selecting this model is the performance which is explained in Section 2.6. Nevertheless, new text embedding models can be easily introduced. This introduction capabilities will be further discussed in the Implementation chapter (see Chapter 7).

For our previous example from the shown ontologies in Figure 4.1, labels of all entities in the $OI_{building}$, OI_{VC}, and $OI_{calidad}$ (e.g., Building, containsSurface, , surfaceTypeEnum, benceno, etc.) are converted to embedding vectors and then stored as $E_{building}$, E_{VC}, and $E_{calidad}$.

In conclusion, our system creates ontology information data OI_* for all of the ontologies O_*. The Entity Discovery and Extraction (*discovery_extraction*) algorithm takes an ontology O_i as input and returns the ontology information data O_i which contains the IRIs, label names, and relationships of the entities $\{ei_1, ...\}$. The Entity Discovery and Extraction (*discovery_extraction*) algorithm can be customized using the configuration file $Conf_{object,unit}$. Customization options are disabling the reasoner or extracting only certain entity types.

Afterward, encoding method *enc* takes all entity information ei_y in OI_i, and returns entity labels as strings. Next, an embedding model *emb* takes the strings as input and computes the embedding vectors E_i. Finally, the ontology information OI_i and embedding vectors

E_i are stored in the file system. For the schema column names C, only the embedding vectors are computed E_C and stored in the cache.

CHAPTER 5

Matching

The following chapter explains how the extracted entity data types and their embeddings are compared with the columns and their data types. Section 5.1 will discuss how the data types are compared to verify compatibility. Section 5.2 will demonstrate how the embedding similarity is calculated. Finally, the last section, 5.3, will explain how the winners are determined and the relevance lists are created. Altogether this chapter discusses the whole mapping process.

5.1 Constraint Checking

When the user provides the columns C, and our encoding method and embedding model convert them into vector embeddings E_C, the Initial Mapping begins. The first step in the Initial Mapping is creating Fields $F^{C,O_{obj/unit_*}}$ between the schema columns C and all ontologies OI_*. There is only one Field between the schema columns and an ontology. A Field $F^{C,O_{obj/unit_i}}$ serves two purposes:

1. Storing the column names as sources and their candidate entities as targets in a SourceTarget matrix $ST^{C,O_{obj/unit_i}}$. Initially, the SourceTarget matrix contains all columns from the schema (as sources) and all entities from the ontology information OI_i (as targets).
2. Checking the data type constraint compatibility ($DataConstSat$) between the source columns and target entities in the SourceTarget matrix and storing the similarity scores in the Constraint Satisfaction matrix $CS^{C,O_{obj/unit_i}}$. In the $CS^{C,O_{obj/unit_i}}$, column-entity pairs where the data types are not compatible are marked as $false$, and cases where the data types are compatible are marked as $true$. Initially, the Constraint Satisfaction matrix is empty and requires the SourceTarget matrix and execution of the Data Constraint Compatibility Check ($DataConstSat$) Algorithm 5.1.

The data type satisfiability check is utilized just to provide further insights to the user about entity types. The reasoning behind it could be summarized as follows:

- *Suggested data types*: User-provided schema data might contain columns with null cells, which is supported in DBRepo. In return, the DBRepo analyzer service can suggest incorrect data types to the Initial Mapping of the SO mapper. Nevertheless, users can change the suggested data types from the table schema definition interface of DBRepo, and this can be reflected in the SO mapping back-end.
- *Enumerations*: There are two different cases for the enumeration satisfaction:
  1. *Satisfaction of enumerated data type to enumerated data type*: For instance, a column in a building schema [Dat22] contains an enumeration type "Multi-family Housing" that can be satisfied by the "Multifamily" enumeration type in the building ontology of TU Wien [KIVK13]. Therefore, a text similarity calculation must be made between "Multifamily Housing" and "Multifamily".
  2. *Satisfaction of data type (which is not an enumeration) to enumerated data*: For example, in the previously mentioned building ontology, the "densityUnitEnum" enumeration has an enumeration type called "GramsPerCubicCm," which implicitly indicates and therefore can be satisfied by the float data type of a column. Thus, another similarity calculation must be made.

Although the SO mapping parser can extract each custom enumerated data type from the entities, the free-form naming conventions require a similarity metric to check satisfaction. Handling such similarity checks presents unique challenges beyond this thesis's scope and will be addressed in the Similarity Calculation Limitations Section 8.7 and the Future Work Section 9.2.

Table 5.1 shows a portion of the building schema used in our evaluation as an example. The remaining column in the schema is not displayed here to reduce complexity in the Building Information Ontology example in Table 4.1 and the Initial Mapping examples provided later.

Column name c_x	Column data type d_x
Area	decimal
Height	decimal
Num_Floors	integer

Table 5.1: Portion of building relational schema [Red24] as example

After the vector embeddings for the columns of the Building schema are computed, the Fields are created between Building schema and Building Information, VC, and Calidad-aire ontologies ($F^{C_{bld},O_*} = \{F^{C_{bld},O_{building}}, F^{C_{bld},O_{VC}}, F^{C_{bld},O_{calidad}}\}$). Afterward, the SourceTarget matrices are created inside the Fields. The SourceTarget matrix between the building schema columns and building ontology contains sources as Area, Height, Num_Floors and each source has targets as all of the entities (Building, containsArea, hasAverageNumberOfFloorsValue, Area, containsSurface, hasSurfaceTypeValue, Surface, containsRectangularGeometry, hasNativeValue, RectangularGeometry, containsHeight, hasUnitValue, Height).

When the creation finishes, the Data Constraint Compatibility Check ($DataConstSat$) Algorithm 5.1 starts. Schema columns are separated based on their data types. For each data type, the first column is selected as a sample. The reasoning behind using only one column for each data type is that if two columns have the same data type, their data type compatibilities will be the same. Thus, the data type compatibilities (satisfactions) of one column can be used for the other columns with the same data type.

The target entities of the sample column are collected from the SourceTarget matrix ST. Their data types are checked for compatibility with the sample source using the ontology information data OI. Finally, the data type constraint satisfaction results are stored for the sample column and all the columns with the same data type in the Constraint Satisfaction matrix. Note that if an entity does not have any data type, the constraint satisfaction result will be $false$ by default.

For the previously given ontologies, a Constraint Compatibility Check ($DataConstSat$) Algorithm will be executed for each Field. Since "Area" and "Height" have *decimal* data type, the "Area" column is selected as a sample. For "Area", all of the target entities (e.g., for the Building Information ontology "Height", "containsArea", ...) are collected. Ontology information data $OI_{building}$, OI_{VC}, and $OI_{calidad}$ are then used to check the data type constraint satisfaction such as:

- Class-data type (CD) dictionary of $OI_{building}$ will provide "{decimal, string}" data types for the "Height" class. Since "Area" has the "decimal" data type, $CS^{C_{bld},O_{building}}_{Area,Height}$ will be $true$.

  Note that the Class-data type dictionary stores all possible data types of all entities. Further information about the dictionaries in ontology information data can be seen in Section 4.2.

- Data property-range (DR) of $OI_{building}$ will provide a "string" data type for the "hasUnitValue" data property. Thus $CS^{C_{bld},O_{building}}_{Area,hasUnitValue}$ will be $false$.

- Object property-Class (OC) and Class-Data type (CD) will provide "surfaceType-Enum" data type for the "containsSurface" object property thus $CS^{C_{bld},O_{building}}_{Area,containsSurface}$ will be $false$.

When the satisfaction results are computed for the "Area" column, the same results will be used for the "Height" column ($CS^{C_{bld},O_{building}}_{Area,y} = CS^{C_{bld},O_{building}}_{Height,y}\ \forall y \in OI_{building}$).

Our Data Constraint Compatibility Check ($DataConstSat$) Algorithm 5.1 takes a Field and a boolean *remove* as inputs. Between line 2 and 12, the list *distinctT* is filled to store columns that are separated by their data types. Note that the columns from the SourceTarget matrix are being used in line 3. For each distinct data type within $distinctT$, a sample column is selected in line 14. Get Data Types (GetDataTypes) Algorithm 5.2 is executed for all of the target entities in line 16. If an entity has a data type as the sample column then lines $17 - 19$ are executed which sets the constraint satisfaction to *true* for the sample column and all of the columns with the same data type. However, if an entity does not have any data type same as the sample column, lines $22 - 28$ are executed. If the boolean *remove* is *true*, then the entities that do not have any compatible data types are removed from the SourceTarget matrix which means they are removed from the similarity calculation. Otherwise, they are marked as $false$ in the Constraint Satisfaction matrix.

5.1

The Get Data Types (GetDataTypes) Algorithm 5.2 uses the dictionaries in ontology information data OI_{obj_i} (see Section 4.2 for dictionaries). to provide data types of an entity to the Data Constraint Compatibility Check ($DataConstSat$) Algorithm. It provides the data types according to the input entity's type:

- *Class entities*: Provides the data types from the sub-class, equivalence, and data property definitions that are stored in the Class-data type (CD) dictionary of ontology information data OI_{obj_i}. Occurs between lines $3 - 5$.
- *Individual entities*: Collects classes of the individual that are stored in Class-Individual (CI) dictionary. Finally provides data types of the related classes. Occurs between lines $6 - 10$.
- *Object Property entities*: Collects range classes of the object property that are stored in the Object property-Class (OC) dictionary. Finally provides data types of the related classes. Occurs between lines $11 - 15$.
- *Data Property entities*: Provides the data types from their ranges that are stored in the Data property-Data type (DD) dictionary. Occurs between lines $16 - 18$.

When the Field between building schema columns and Building Information ontology is provided to the Data Constraint Compatibility Check Algorithm 5.1 as an input, the Constraint Satisfaction matrix output is shown in Table 5.2.

Note that when the SourceTarget and Constraint Satisfaction matrices are computed for a Field, it is then stored in a cache. Furthermore, Fields (SourceTarget and Constraint Satisfaction matrices) have the capability of concatenating or removing column data. This capability is used when users interact with our system (see Chapter 6). For instance, when the user adds a new column c_{new} to the schema C, temporary Fields $F^{C_{c_new},O_*}$ are created between the new column and ontologies. SourceTarget and Constraint Satisfaction matrices are computed. Finally Fields of the initial schema columns F^{C,O_*}

Algorithm 5.1: Data Constraint Compatibility Check (DataConstSat)

```
Input  : field ($F^{C,obj_i}$): Field which contains columns $C$, column data types $D$,
         object information $OI_i$, and SourceTarget matrix $ST^{C,obj_i}$.
Input  : remove: Boolean whether to remove unsatisfied entities or not. $false$
         by default.
Output : CS ($CS^{C,O_{obj_i}}$): Constraint Satisfaction matrix that holds pairwise
         compatibility information between column and entity data types.
CS ← []
distinctT ← []
for column IN field.ST do
    CS[column] ← []
    columnT ← field.$D$[column]
    if columnT IN distinctT then
        distinctD[columnT].APPEND(column)
    end
    else
        distinctT[columnT] ← [column]
    end
end
for type IN distinctT do
    sampleColumn ← distinctT[type][0]
    for entity IN field.ST[sampleColumn] do
        if type IN GetDataTypes(entity, field.$OI_{obj_i}$) then
            for column IN distinctT[type] do
                CS[column][entity] ← true
            end
        end
        else
            for column IN distinctT[type] do
                if remove IS TRUE then
                    DELETE(field.$ST$[column][entity])
                end
                else
                    CS[column][entity] ← false
                end
            end
        end
    end
end
return CS
```

Algorithm 5.2: Get Data Types (GetDataTypes)

```
Input  : entity: Name of the entity.
Input  : OI ($OI_{obj_i}$): Ontology information data which contains the Class-Data
         type ($CI$), Class-Individual ($CI$), Object property-Class ($OC$), and
         Data property-Data type ($DD$) relations.
Output : dataTypes: Possible data types of the entity.
1  dataTypes ← []
2  entityType ← OI.Type[entity]
3  if entityType IS owl:class then
4      dataTypes ← OI.CD[entity]
5  end
6  else if entityType IS Individual then
7      for class IN GetClasses(OI.CI, entity) do
8          dataTypes.APPEND(OI.CD[class])
9      end
10 end
11 else if entityType IS owl:ObjectProperty then
12     for class IN OI.OC(entity) do
13         dataTypes.APPEND(OI.CD[class])
14     end
15 end
16 else if entitytype IS owl:DatatypeProperty then
17     dataTypes ← OI.DD[entity]
18 end
19 return dataTypes
```

and the temporary Fields F^{C_{cnew},O_*} are concatenated from their Columns C, SourceTarget matrices $ST^{C,obj/unit_i}$ and Constraint Satisfaction matrices CS^{C,obj_i} as:

$$
\begin{aligned}
&C = C \cup C_{c\_new},\\
&ST^{C,obj/unit_i} = ST^{C,obj/unit_i} \cup ST^{C_{c\_new},obj/unit_i},\\
&CS^{C,obj_i} = CS^{C,obj_i} \cup CS^{C_{c\_new},obj_i} \text{ s.t.}\\
&\forall F^{C,O_{obj/unit_i}} \in F^{C,O_{obj/unit*}}\\
&F^{C,O_{obj/unit_i}} = \begin{cases} \{C, D, OI_{obj_i}, ST^{C,obj_i}, CS^{C,obj_i}\}, & \text{if } O_i \text{ is object ontology}\\ \{C, D, OI_{unit_i}, ST^{C,unit_i}\}, & \text{otherwise}\end{cases}
\end{aligned}
\tag{5.1}
$$

More information about the list of interactions can be seen in User Interactions Chapter 6.

Constraint Satisfaction matrix $CS^{C_{bld},O_{building}}$	Area (decimal)	Height (decimal)	Num_Floors (integer)
Area (decimal, string)	true	true	false
hasUnitValue (string)	false	false	false
hasNativeValue (decimal)	true	true	false
Building (decimal, string)	true	true	false
containsArea (decimal, string)	true	true	false
hasAverageNumberOfFloorsValue (decimal)	true	true	false
containsSurface (surfaceTypeEnum)	false	false	false
hasSurfaceTypeValue (surfaceTypeEnum)	false	false	false
Surface (surfaceTypeEnum)	false	false	false
containsRectangularGeometry ()	false	false	false
RectangularGeometry ()	false	false	false
containsHeight (decimal, string)	true	true	false
Height (decimal, string)	true	true	false

Table 5.2: Constraint Satisfaction matrix between building schema columns and building ontology examples

5.2 Similarity Calculation

Calculating similarity is another crucial aspect of our system. This is because the creation of relevance lists relies on the accuracy of similarity scores when sorting the entities. Currently, similarity calculation is based on the embeddings produced by the state-of-the-art BGE M3-Embedding model [CXZ+24]. These embeddings are vector representations of extracted labels from object and unit entities, as well as user-provided column names.

SO mapper currently utilizes cosine similarity. Fundamentally it calculates the cosine angle between two vectors, such as the cosine angle between the label of an object entity vector and the column name vector. The intuition behind cosine similarity is that an embedding model represents similar objects, such as "vehicle" and "car", as two vectors that are closer to each other than other objects such as "airplane". Nevertheless, our system is capable of utilizing different scoring mechanisms, including scoring metrics that are not necessarily based on embedding models. For instance, instead of relying on text embedding models, the Levenshtein distance can be utilized as a similarity metric.

In this last step, our Similarity Calculation (*sim_calc*) Algorithm 5.3 is executed which takes three inputs: all of the Fields $F^{C,O_{obj/unit*}}$, ontology embeddings $E_{obj/unit*}$, and column embeddings E_C. In return, outputs the column name-entity label similarity scores. This algorithm runs separately for object and unit ontologies.

Initially, the algorithm iterates through ontologies. For each ontology, a list of ontology embeddings is called in line 2. Then, for each column, the column's embedding (line

4) and its target entity embeddings (lines 5 – 6) are collected. Following this, in line 7, pairwise cosine similarity is computed for the collected embeddings, and the similarity scores are returned.

Algorithm 5.3: Similarity Calculation for Embedding-based Representations (*sim_calc*)

```
Input  : fields ($F^{C,O_{obj/unit_*}}$): List of object or unit ontology fields.
Input  : sEmbedding ($E_C$): Embedding representations of each column name.
Input  : oEmbeddings ($E_{obj/unit_*}$): Embedding representations of each object or
         unit entity label.
Output : sims ($R^{C,obj/unit_*}$): Similarity scores between column names and
         entities from ontologies
1  for field IN fields do
2      oEmbedding ← oEmbeddings[field.name]
3      for column IN field.SourceTarget.sources do
4          columnEmbedding ← sEmbedding[column]
5          entities ← field.SourceTarget[column]
6          entityEmbeddings ← oEmbedding[entities]
7          sims[field.name][column] ← COSINE_SIMILARITY(columnEmbedding,
            entityEmbeddings)
8      end
9  end
10 return sims
```

The outputs of the Similarity Calculation (*sim_calc*) algorithm for our previously defined example schema 5.1 and ontologies 4.1 can be seen in Table 5.3 for Building Information ontology, Table 5.4 for VC ontology, and Table 5.5 for Calidad-aire ontology.

5.3 Relevance Lists

Once the Similarity Calculation (*sim_calc*) algorithm completes, resulting similarity scores R^{C,O_*} are processed to construct relevance lists RL. There are three types of relevance lists:

- *Entity level relevance list* ($RL_x^{C,obj/unit_*}$ for a column x): It is a list of rows where each row contains the data of an entity with the highest similarity score within its ontology. Data of a row contains the label name, similarity score, belonging ontology, and constraint satisfaction result of the entity. Rows in the list are sorted by similarity scores in descending order.

  Construct Entity level relevance list (*const_ent_rl*) Algorithm 5.4 is responsible for the construction of an entity level relevance list. For each Field, $F^{C,obj/unit_i}$, line

Similarity Scores $R^{C_{bld},O_{building}}$	Area	Height	Num_Floors
Area	**1.0**	0.536	0.491
hasUnitValue	0.467	0.506	0.474
hasNativeValue	0.437	0.436	0.421
Building	0.60	0.601	0.571
containsArea	0.66	0.434	0.469
hasAverageNumberOfFloorsValue	0.472	0.488	**0.703**
containsSurface	0.466	0.457	0.498
hasSurfaceTypeValue	0.476	0.440	0.443
Surface	0.531	0.499	0.521
containsRectangularGeometry	0.421	0.461	0.421
RectangularGeometry	0.479	0.477	0.460
containsHeight	0.421	0.707	0.438
Height	0.536	**1.0**	0.487

Table 5.3: Example similarity scores for the Building schema and Building Information ontology using Algorithm 5.3 and BGE M3-Embedding model [CXZ+24]

Similarity Scores $R^{C_{bld},O_{VC}}$	Area	Height	Num_Floors
business entity	0.551	0.469	0.434
height	0.541	**0.917**	0.496
organization	0.566	0.514	0.426
person	**0.617**	0.642	**0.5**
vehicle	0.560	0.544	0.431
quantitative value float	0.358	0.403	0.428

Table 5.4: Example similarity scores for the Building Schema and VC ontology using Algorithm 5.3 and BGE M3-Embedding model [CXZ+24]

Similarity Scores $R^{C_{bld},O_{calidad}}$	Area	Height	Num_Floors
Air Quality Property	**0.5**	**0.48**	**0.479**
benceno	0.393	0.392	0.345
dioxidoDeAzufre	0.34	0.344	0.348

Table 5.5: Example similarity scores for the Building schema and Calidad-aire ontology using Algorithm 5.3 and BGE M3-Embedding model [CXZ+24]

3 collects the similarity scores between the column name x and entities of ontology information data OI_i, and line 4 finds the highest similarity score in that list. Line 5 collects the name of the entity with the highest similarity score. Constraint satisfaction of the entity is collected from the Constraint Satisfaction matrix in line 6. The name of the ontology i is collected in line 7. Finally, the data of the entity with the highest similarity score is stored in entity level relevance list in line 8.

- *Entity level relevance list on search* ($RL_{x,t}^{C,obj/unit_*}$ for a column x with search term t): It is a list of rows where each row contains the data of an entity that starts with the search term t. Data of a row is the same as entity level relevance list and

Algorithm 5.4: Construct Entity level relevance list (*const_ent_rl*)

```
    Input  : column ($c_x$): Name of a column.
    Input  : fields ($F^{C,O_{obj/unit*}}$): List of object or unit ontology fields.
    Input  : sims ($R^{C,obj/unit_*}$): Similarity scores.
    Output : eRl ($RL^{C,obj/unit_x}$): Entity level relevance list for the column $x$.
 1  eRl ← []
 2  for field IN fields do
 3      entitySims ← sims[field.$OI$.name][column]
 4      eHighestSim ← MAX(entitySims)
 5      eName ← entitySims.GET_NAME(eHighestSim)
 6      eSat ←field.$CS$[column][eName]
 7      ontName ←field.ontologyName
 8      eRl.APPEND([eName,eHighestSim,eSat,ontName])
 9  end
10  return eRl
```

rows are sorted in the same order.

Construct Entity level relevance list on search (*const_ent_rl_s*) 5.5 is responsible for constructing this list type. For each Field $F^{C,obj/unit_i}$, line 3 collects the similarity scores between the column x and entities of ontology information data OI_i. Line 4 collects the entity names that start with the given search term t. For each collected entity that starts with the search term t, their label names, similarity scores, constraint satisfactions, and ontology names are stored in the relevance list.

- *Ontology level relevance list* (R^o): It is a list of rows where each row contains the data of an ontology. Data of a row contains the ontology name, the average similarity score of the first-ranking entity for each column, and the standard deviation of the average similarity score.

  Construct Ontology level relevance list (*const_ont_rl*) is responsible for constructing this list type. For each Field $F^{C,obj/unit_i}$, lines $4-5$ collect the highest similarity scores for each column in C and entities in ontology information data OI_i. Afterward, line 7 calculates the mean value, and line 8 calculates the standard deviation. Finally, the data is stored in the list in line 9.

Furthermore, these relevance lists are created in the following cases:

- *Initial Mapping*: When the user initially uploads a relational table as a file to the DBRepo. An example of such a case is given in the Data flow of schemas in SO Mapper Figure 3.3. Two types of relevance lists are created. Entity level relevance

Algorithm 5.5: Construct Entity level relevance list on search (*const_ent_rl_s*)

```
Input  : column ($c_x$): Name of a column.
Input  : term ($t$): Search term. By default none.
Input  : fields ($F^{C,O_{obj/unit*}}$): List of object or unit ontology fields.
Input  : sims ($R^{C,obj/unit*}$): Similarity scores
Output : eRlS ($RL^{C,obj/unit*}_{x,t}$): Entity level relevance list on search for the
         column $x$ and search term $t$.
1  eRlS ←[]
2  for field IN fields do
3      entitySims ← sims[field.$OI$.name][column]
4      entityNames ← field.$OI$.LabelsStartWith($t$)
5      for eName IN entityNames do
6          eSim ← entitySims[eName]
7          eSat ←field.$CS$[column][eName]
8          ontName ←field.ontologyName
9          eRlS.APPEND([eName,eSim,eSat,ontName])
10     end
11 end
12 return eRlS
```

Algorithm 5.6: Construct Ontology level relevance list (*const_ont_rl*)

```
Input  : fields ($F^{C,O_{obj*}}$): List of object ontology fields.
Input  : sims ($R^{C,obj*}$): Similarity scores
Output : oRl ($R^o$): Ontology level relevance list. Only used for the object
         ontologies.
1  oRl ←[]
2  for field IN fields do
3      population ← []
4      for column IN sims[field.OI.name] do
5          population.APPEND(MAX(sims[field.$OI$.name][column]))
6      end
7      mean ← SUM(population) / SIZE(sims[field.$OI$.name])
8      std ← CALCULATE_STD(population,mean)
9      oRl.APPEND([field.ontologyName,mean,std])
10 end
11 return oRl
```

list $RL_x^{C,obj/unit_*}$ for each column x in the schema columns C and an Ontology level relevance list for the object ontologies R^{oo}.

- *User search*: After the Initial Mapping, since columns and their respective Entity level relevance lists are provided to the users, they can search entities for a column x using a search term t (see Figure 7.6 where $t = "TotalOrganic"$). In this case an Entity level relevance list on the search $RL_{x,t}^{C,obj/unit_*}$ is created.
- *User Interactions*: After the Initial Mapping, users can update the schema and provide feedback for the similarity calculation using our UI. These interactions change the data within Fields and Similarity Calculations. The details of such interactions can be seen in Chapter 6. Their relevance lists are as follows:
  - *Removing columns*: After the user clicks on the "Remove" button of a column in our UI (see 8 in Figure 7.4). A new Ontology level relevance list is dispatched.
  - *Setting column(s) as primary key*: After the user clicks on the "Primary Key" checkbox in our UI (see 5 in Figure 7.4). A new Ontology level relevance list is dispatched.
  - *Introducing new columns*: After the user clicks on the "Add Column" button at the bottom of our UI. A new Ontology level relevance list and a new Entity level relevance list ($RL_{new\_column}^{C,obj/unit_*}$) are dispatched.
  - *Changing the column name*: After the user changes the name of a column by typing to the textbox in our UI (see 3 in Figure 7.4). A new Ontology level relevance list and a new Entity level relevance list ($RL_{changed\_name}^{C,obj/unit_*}$) are dispatched.
  - *Changing the data type of column*: After the user changes the data type of a column using the dropdown in our UI (see 4 in Figure 7.4). A new Ontology level relevance list and a new Entity level relevance list are dispatched.
  - *Direct influencing*: After the user selects an entity from the Object Match dropdown in our UI (see 6 in Figure 7.4). A new Entity level relevance list is dispatched.
  - *Indirect influencing*: After the user selects an object ontology from the Object Ontology dropdown in our UI (see 1 in Figure 7.4). New Entity level relevance lists are dispatched for all columns.

Note that more information about our UI is given in Section 7.4 and user interactions in Chapter 6.

For example, using the previously defined Building schema columns C_{bld}, Building Information ontology $OI_{building}$, VC ontology OI_{VC}, and Calidad-aire ontology $OI_{calidad}$. Initial Mapping will produce Entity level relevance lists for all of the columns ("Area", "Height", and "Num_Floors") and an Ontology level relevance list for object ontologies (Building Information, VC, and Calidad-aire ontologies). Note that in the examples we do not have any unit ontologies, thus only one Ontology level relevance list will be produced. The Entity level relevance list for the column "Num_Floors" is in Table 5.6 and the Ontology level relevance list is in Table 5.7.

Entity Name	Score	Constraint Sat.	Ontology
hasAverageNumberOfFloorsValue	0.703	false	Building Information
person	0.5	false	VC
Air Quality Property	0.479	false	Calidad-aire

Table 5.6: Example Entity level relevance list for column Num_Floors

Ontology	Average of first-ranking entity scores	Standard deviation of first-ranking entity scores
Building Information	0.901	+/-0.140
VehicleCore	0.678	+/- 0.175
Calidad-aire	0.486	+/- 0.009

Table 5.7: Ontology level relevance list of Building schema

Lastly, when the user searches for an entity for the "Height" column using the term "h", an Entity level relevance list on search will be produced which is given in Table 5.8.

Entity Name	Score	Constraint Sat.	Ontology
Height	1.0	true	Building Information
height	0.917	false	VC
hasUnitValue	0.506	false	Building Information
hasAverageNumberOfFloorsValue	0.488	true	Building Information
hasSurfaceTypeValue	0.440	false	Building Information
hasNativeValue	0.436	true	Building Information

Table 5.8: Entity level relevance list on search for column Height with search term "h"

In conclusion, when the Initial Mapping occurs, Fields F^{C,O_*} are created between column names C and ontology information data OI_*. Furthermore, the Data Constraint Compatibility Check ($DataConstSat$) algorithm is executed for the Fields of object ontologies. These satisfaction checks utilize the ontology information data OI_i to find data types of the entities. Afterward, Similarity Calculation for Embedding-based Representations (sim_calc) is executed for all of the Fields. Finally, similarity scores from the sim_calc algorithm are used to produce the Entity level relevance lists for all columns and an Ontology level relevance list for object ontologies. This concludes the Initial Mapping. The cache contains the Fields F^{C,O_*}, Similarity scores R^{C,O_*}, and column embeddings E_C. Now users can search for entities for a column using a search term. This will create an Entity level relevance list on search.

CHAPTER 6

User Interactions

The following chapter will explain how the user can interact with the SO mapper back-end (specifically the Fields and Similarity Calculations). In section 6.1, handling the addition, removal, or update operations for schema columns will be discussed. Thereafter, section 6.2 will explain how user feedback is utilized as an influencing mechanism. Lastly, section 6.3 will showcase how the SO mapper front-end can handle cache time-outs.

6.1 Updating the Schema

DBRepo[wei22] front-end provides various schema manipulationsto the users. Since our system is built to cooperate with DBRepo, we must comply with such schema manipulations. Compliance is achieved by integrating the SO mapper front-end into the table schema definition interface, which sends API requests to SO mapper back-end. Further information about the API requests can be found in Section 7.3.

For the schema manipulations, SO mapper back-end utilizes the previously defined Fields $F^{C,O_{obj/unit*}}$, Embeddings E_*, and Similarity Scores $R^{C,obj/unit_*}$ and executes the Add Column Algorithm 6.1 or Remove Column Algorithm 6.2. This implies that an Initial Mapping (see Figure 3.2) is required before users can reflect their interactions to the SO mapper. When Initial Mapping is done, SO mapper front-end will receive a schema key. Using this key, SO mapper back-end can identify which schema is being manipulated.

DBRepo table schema definition interface provides the following schema manipulations:

- *Removing columns*(see 8 in Figure 7.4): SO mapper back-end will receive the schema key and the column name. Afterward, Remove Column Algorithm 6.2 will be executed.

- *Setting column(s) as primary key* (see 5 in Figure 7.4): The same procedure with the "Removing columns" will be executed. The reason behind this is that a primary column (e.g., "id") cannot be mapped to an object or unit entity.
- *Introducing new columns*: SO mapper back-end will receive the schema key, a column name, and a data type. Add Column Algorithm 6.1 will be executed.
- *Changing the column name*(see 3 in Figure 7.4): SO mapper back-end will receive the schema key, the new column name, and the data type. Remove Column Algorithm 6.2 and then the Add Column Algorithm 6.1 will be executed.
- *Changing the data type of column*(see 4 in Figure 7.4): SO mapper back-end will receive the schema key, the column name, and the new data type. Remove Column Algorithm 6.2 and then the Add Column Algorithm 6.1 will be executed.

The Add Column algorithm can be seen in Algorithm 6.1. It takes a schema key for the schema columns, the name of the column that will be added (c), the data type of the column (d), Ontology information data of the object and unit ontologies ($OI_{obj/unit_*}$), embeddings of the object and unit ontologies ($E_{obj/unit_*}$) the cache of our SO mapper back-end, and directive data (dir).

Our system supports users in providing feedback on the mapping for units. The directive data is used to apply the previous feedback to the new columns. Two types of feedback can occur:

- *User selecting the correct object entity for a column*: In this case, the Direct Influence ($DirectInfluence$) Algorithm 6.3 is executed to apply the user feedback. Directive data contains the column name (c), the name of the correct object entity (e_x), and the name of the ontology (i) of the correct object entity.
- *User selecting the correct object ontology*: For this case, the Indirect Influence ($IndirectInfluence$) Algorithm 6.4 is executed to apply the user feedback. Directive data contains the column names (C) of the schema and the correct ontology name (i).

When the Add Column algorithm is executed, a check is performed to ensure that the Initial Mapping for the schema (see Figure 3.3) has been completed and not expired. If this is not the case, the SO mapper will return a "HTTP 404" not found message, and the SO mapper front-end will request a reload, as discussed further in the Reload section 6.3.

However, if the Initial Mapping has been completed beforehand, then the cache contains the Fields $F^{C,O_{obj/unit_*}}$, Similarity Scores $R^{C,obj/unit_*}$, and Column embeddings E_C, and the algorithm will proceed.

Algorithm 6.1: Add Column

```
Input : key: Id of the user-provided schema in SO mapper back-end.
Input : dir: Directive data for user feedback.
Input : column ($c$): Name of the new column.
Input : datatype ($d$): Data type of the new column.
Input : objOIs ($OI_{obj*}$): Ontology information data for object ontologies.
Input : unitOIs ($OI_{unit*}$): Ontology information data for unit ontologies.
Input : oEmbeddingObj ($E_{obj*}$): Embeddings of object entities.
Input : oEmbeddingUnit ($E_{unit*}$): Embeddings of unit entities.
Input : cache: SO mapper cache which holds the Fields, Column embeddings,
        and Similarity scores.
Output: eRlObj ($RL^{C,obj_c}$): Object Entity level relevance lists for the column.
Output: eRlUnit ($RL^{C,unit_c}$): Unit Entity level relevance lists for the column.
Output: oRL ($RL^o$): Ontology level relevance list.
1  if EXISTS(cache[key]) IS FALSE then
2      return HTTP 404
3  end
4  tmp ← Initial_Match(column, datatype, objOIs, unitOIs, oEmbeddingObj,
     oEmbeddingUnit)
5  if dir.direct IS true then
6      tmp.Unit.sims ← DirectInfluence(dir.direct.data, tmp.Column.Emb,
         oEmbeddingObj, oEmbeddingUnit, tmp.Unit.sims, unitOIs)
7  end
8  else if dir.indirect IS true then
9      tmp.Unit.sims ← IndirectInfluence(dir.indirect.data, tmp.Column.Emb,
         tmp.Object.sims, oEmbeddingObj, oEmbeddingUnit)
10 end
11 for field IN cache[key].Object.Fields do
12     field ← Concatenate(field, tmp.Object.Fields[field.$OI$.name])
13     cache[key].Object.sims[field.$OI$.name][column] ←
         tmp.Object.sims[field.$OI$.name][column]
14 end
15 for field IN cache[key].Unit.Fields do
16     field ← Concatenate(field, tmp.Unit.Fields[field.$OI$.name])
17     cache[key].Unit.sims[field.$OI$.name][column] ←
         tmp.Unit.sims[field..$OI$.name][column]
18 end
19 eRlObj ← const_ent_rl(column, cache[key].Object.Fields,
     cache[key].Object.sims)
20 eRlUnit ← const_ent_rl(column, cache[key].Unit.Fields, cache[key].Object.sims)
21 oRL ← const_ont_rl(cache[key].Object.Fields, cache[key].Object.sims)
22 return eRlObj, eRlUnit, oRL
```

In line 4, an Initial Mapping (only the SO mapper back-end) occurs using the column c and data type d. The data flow of the Initial Mapping can be seen in Figure 3.3) which uses the algorithms that are previously defined. Note that instead of saving the Column embeddings, Fields, and Similarity stores to cache, they are used in the variable *tmp*. Between the lines $5 - 7$ and $8 - 10$, user feedback is applied using the previously defined directive data.

All object Fields ($F^{C,O_{obj_i}}$) and new Fields in the *tmp* are concatenated between lines $11 - 14$. This concatenation is defined in the Equation 5.1. The same procedure occurs for the unit Fields between lines $15 - 18$.

Lastly, two Entity level relevance lists (RL^{c,obj_*} and $RL^{c,unit_*}$ for the new column c) and an Ontology level relevance list are created and then sent to the SO mapper front-end (between lines $19 - 22$).

Our Remove Column can be seen in Algorithm 6.2. It takes a schema key for the schema columns C, the name of the column that will be removed, and the cache of our SO mapper back-end. Initial lines $(1 - 3)$ are the same as Add Column Algorithm 6.1.

Fields, Similarity scores, and column embeddings from the Initial Mapping are collected between the lines of $4 - 8$. Afterward, data of column c is deleted from each object Field $F^{C,O_{obj_i}}$ between lines $9 - 11$. Column data consists of targets in the SourceTarget matrix ($ST^{C,O_{obj_i}}_{column}$) and constraint satisfactions in the Constraint Satisfaction matrix ($CS^{C,O_{obj_i}}_{column}$). Furthermore, Similarity scores of the column c are deleted from all of the object ontologies in line 12. Note that each Field contains an ontology information data OI and the ontology name can be collected using $field.OI.name$ (see Field definition in Section 3.2).

The same delete process occurs for the unit ontologies between the lines $14 - 18$. Next, embedding of the column is deleted from the Column embeddings E_C in line 19. Fields, Similarity scores of object and unit ontologies, and Column embeddings are saved to the cache in line 20. Finally, an Ontology level relevance list is created using the previously defined Algorithm 5.6, and the output is sent to the SO mapper front-end.

The reason behind changing column names and data types is using the Remove Column Algorithm 6.1 and Add Column Algorithm 6.1 is Data Constraint Compatibility Check (DataConstSat) Algorithm can remove the unsatisfied entities using a boolean *remove* (see Section 5.1). The unsatisfied constraints are not removed by default. However, this functionality can be utilized by the administrators by changing the default value and continue using the aforementioned algorithms.

6.2 User Feedback

After the Initial Mappinging finishes, users can provide feedback on the mapping for units. This feedback changes the Similarity scores between column(s) and unit entities. There are two types of feedback. The first one is called Direct Influencing (DirectInfluence).

Algorithm 6.2: Remove Column

```
Input  : key: Schema key of the user-provided schema in SO mapper back-end.
         Key represents the id of schema.
Input  : column ($c$): Name of a column that will be removed from the schema $C$.
Input  : cache: SO mapper cache which stores Fields $F^{C,O_{obj/unit*}}$, Similarity
         Scores $R^{C,obj/unit*}$, and Column embeddings $E_C$.
Output : oRl ($RL^o$): Ontology level relevance list. Only used for the object
         ontologies.
1  if EXISTS(cache[key]) IS FALSE then
2      return HTTP 404
3  end
4  oFields ← cache[key].$F^{C,O_{obj*}}$
5  oSims ← cache[key].$R^{C,O_{obj*}}$
6  uFields ← cache[key].$F^{C,O_{unit*}}$
7  uSims ← cache[key].$R^{C,O_{unit*}}$
8  embeddings ← cache[key].$E_C$
9  for field IN oFields do
10     DELETE(field.$ST$[column])
11     DELETE(field.$CS$[column])
12     DELETE(oSims[field.$OI$.name][column])
13 end
14 for field IN uFields do
15     DELETE(field.$ST$[column])
16     DELETE(field.$CS$[column])
17     DELETE(uSims[field.$OI$.name][column])
18 end
19 DELETE(embeddings[column])
20 cache[key] ← oFields, oSims, uFields, uSims, embeddings
21 oRl ← const_ont_rl(oFields, oSims)
22 return oRl
```

When the user selects an object entity (e_y) from the ontology for a column (c_x), both the object entity and the column will participate in the similarity calculation (sim_calc) for the unit entities. Participation is defined as averaging the embeddings of the object entity and column such as:

$$\begin{aligned} &v^{avr}_{x,y} = \frac{v_x + v_y}{2}, \text{ where} \\ &v_x \text{ is the embedding of the column,} \\ &v_y \text{ is the embedding vector of the object entity} \end{aligned} \tag{6.1}$$

Although only the selected object entity can calculate the similarity scores for unit entities, the column name can be a synonym or contain the unit abbreviation. Enabling both the column and object entity to participate in similarity calculation provides broader information.

For instance, if the column name is *WindAvgMPH* and the user maps to an object entity such as *Wind*. Embedding of the column name ($v_{WindAvgMPH}$) and embedding of the object entity label names (v_{Wind}) are averaged ($v^{avr}_{WindAvgMPH,Wind} = \frac{v_{WindAvgMPH} + v_{Wind}}{2}$). Subsequently, the cosine similarity between the averaged embedding and the unit entity embedding from each unit ontology is calculated using the previously defined Similarity Calculation (sim_calc) Algorithm 5.3. In this way, the user can provide feedback to the mapping for units (such as *Mile per Hour* for the example above).

The Direct Influence Algorithm 6.3 begins by taking the column name (c_x) and embedding, the label name of the selected object entity (e_x), the ontology name of the object entity (i), Column Embeddings E_C, object and unit Embeddings ($E_{obj/unit_*}$), unit Similarity scores ($R^{C,unit_*}$), and the Ontology information data (OI_{unit_*}) of each unit ontology. The Algorithm starts by computing the average embedding using column name embedding and label name embedding in line 1. Afterward, temporary Fields are created between the column name and unit ontologies ($F^{c,unit_*}$). This occurs between lines $2-6$. In line 7, the Similarity Calculation (sim_calc) algorithm is executed using the temporary Fields, averaged embedding, and unit ontology Embeddings. The output is the influenced Similarity scores ($R^{c_x,unit_*}$) between the averaged embedding (column name and object entity label name) and unit embeddings. Lastly, for each unit ontology, new unit Similarity Scores are saved and returned (line $8-11$).

The second interaction the SO front-end provides is called Indirect Influence (Indirect-Influence). It is initiated when the user selects an object ontology (i).For each column, both the column (c_x) and the object entity with the highest Similarity score in the selected object ontology (e_y s.t. $R^{C,obj_i}_{x,y} = max(R^{C,obj_i}_x)$) will participate in the similarity calculation (sim_calc) for unit entities. In contrast to the Direct Influence, participation is defined as weighted averaging embeddings of the object entity and column such as:

Algorithm 6.3: Direct Influence (DirectInfluence)

```
Input  : column ($c_x$): Column name.
Input  : object ($e_y$): Label name of the object entity that influences the unit
         results of the column.
Input  : ontology ($i$): Ontology name of the object.
Input  : cEmbeddings $E_C$: Column Embeddings.
Input  : oEmbeddingObj ($E_{obj_*}$): Embeddings of entities in object ontologies.
Input  : oEmbeddingUnit ($E_{unit_*}$): Embeddings of entities in unit ontologies.
Input  : unitSims ($R^{C,unit_*}$): Unit Similarity scores of each unit ontology.
Input  : unitOIs ($OI_{unit_*}$): Ontology information data of unit ontologies.
Output : unitSims ($R^{C,unit_*}$): Updated unit results for each unit ontology.
1  averagedEmbedding[column] ← (cEmbeddings[column] +
     oEmbeddingObj[ontologyName][object])/2
2  tmpFields ← []
3  for unitOI IN unitOIs do
4      tmpField ← Field(column, unitOI)
5      tmpFields.APPEND(tmpField)
6  end
7  influencedSims ← sim_calc(tmpFields, averagedEmbedding, oEmbeddingUnit)
8  for unitOI IN unitOIs do
9      unitSims[unitOI.name][column] ←
         influencedSims.Unit.sims[unitOI.name][column]
10 end
11 return unitSims
```

$$\begin{aligned} v_{x,y}^{wAvr} &= \frac{v_x * 1 + v_y * w}{1 + w}, \text{ where} \\ w &= R_{c,y}^{C,obj_i}, \\ & v_x \text{ is the embedding vector of the column, and} \\ & v_y \text{ is the embedding vector of the object entity} \end{aligned} \tag{6.2}$$

For example, the similarity between the column *WindAvgMPH* and the object entity *Wind* is 0.647. When the user selects the ontology of *Wind* entity, the embeddings between *WindAvgMPH* and *Winde* are weighted averaged, where the weight of *WindAvgMPH* is 1 and the weight of *Wind* is 0.647.

The Indirect Influence can be seen in Algorithm 6.4. For each column, the embedding of the object entity with the highest similarity score from the selected object ontology is collected between lines $2 - 4$. Embedding of the column is collected in line 5. Afterward, the collected embeddings are weighted averaged in line 6. When the new embeddings are computed for each column, the algorithm will proceed to line 8. For each unit ontology, the Similarity Calculation (*sim_calc*) Algorithm will be executed using the new column embeddings between the lines 8 – 13. Finally, the new similarity scores are saved and returned between lines 14 – 17.

Algorithm 6.4: Indirect Influence (IndirectInfluence)

```
Input  : columns: Column names.
Input  : ontologyName ($i$): Name of the object ontology.
Input  : cEmbeddings ($E_C$): All of the column embeddings.
Input  : objectSims ($R^{C,obj_*}$): Object and unit similarity scores for each unit
         ontology.
Input  : oEmbeddingObj ($E^{C,obj_*}$): All of the embeddings from all of the object
         ontologies.
Input  : oEmbeddingUnit ($E^{C,unit_*}$): All of the embeddings from all of the unit
         ontologies.
Input  : unitOIs ($OI_{unit_*}$): Ontology information data for unit ontologies.
Output : results: Updated unit results for each unit ontology.
1  for column IN columns do
2      eHighestSim ← MAX(objectSims[ontologyName][column])
3      eNname ← objectSims[ontologyName].GET_NAME(eHighestSim)
4      eEmbedding ← oEmbeddingObj[ontologyName][eNname]
5      columnEmbedding ← cEmbeddings[column]
6      wAveragedEmbeddings[column] ←
         ((eHighestSim*eEmbedding)+columnEmbedding)/(1+eHighestSim)
7  end
8  fields ← []
9  for unitOI IN unitOIs do
10     field ← Field(columns, unitOI)
11     fields.APPEND(field)
12 end
13 influencedSims ← sim_calc(fields, wAveragedEmbeddings, oEmbeddingUnit)
14 for unitOI IN unitOIs do
15     unitSims[field.$OI$.name] ← influencedSims.Unit.sims[field.$OI$.name]
16 end
17 return unitSims
```

An additional remark is that cosine similarity uses the orientation of each vector rather than considering their magnitudes. However, in the Indirect Influence Algorithm 6.4, we change the magnitudes of the vectors but then add them together. This weighted addition impacts the direction of the vectors, which changes the orientation. Thus, cosine similarity can calculate the reflected changes. Furthermore, division operations are executed in case the previously defined Similarity Calculation for Embedding-based Representations (*sim_calc*) Algorithm 5.3 is changed by the administrators to use another vector similarity calculation metric such as Euclidean distance.

Our system is also capable of utilizing the unit entities for influencing. However, designing the front-end would make the interface harder to use for users. Since DBRepo is designed to be a user-friendly platform, we are not utilizing this functionality.

6.3 Reload

The SO mapper front-end communicates with the back-end using schema hash keys. Our back-end saves the schema-ontology results of the user and these schema hash keys are used to identify which schema-ontology results are related to which user. More information regarding the caching mechanism and overall service architecture is given in the Implementation Chapter 7.

However, schema-ontology results are deleted from the back-end after some time. When this occurs while the user is still using the DBRepo table schema interface, SO mapper front-end requests multiple reloads. These reloads contain the Initial Mappinging as well as applied Indirect Influencing and Direct Influencing if there were any.

Our back-end provides reloads one by one rather than executing all at once. Mainly there are three reloads that can occur:

- *Reloading the Initial Mapping*: Restart the Initial Mappinging in the back-end and then save the results.
- *Reloading the indirect influence*: Applies Indirect Influence using the selected object ontology, and then updates the results. The most important feature of this reload is that it excludes already selected object entities and unit entities.
- *Reloading the direct influence*: Applies multiple Direct Influences by using the selected entities of each column. That means all of the direct influences will occur in one reload.

Note that since Indirect Influence excludes the already selected object entities, Indirect and Direct influence will not intervene with each other's Similarity scores. Thus the whole integrity of the results is being secured.

CHAPTER 7

Implementation

The upcoming chapter elaborates on how the stateful API architecture is designed to ensure resilience and flexibility while providing control for scalability. Each section will offer insights into both the front-end and back-end. The first section 7.1 discusses the underlying programming language, libraries, and frameworks utilized to operate our system. Meanwhile, the second section 7.2 explains the implementation of the previously mentioned algorithms into services that construct the overall architecture. Section 7.3 provides details about how the communication is being done between the back-end and front-end, while the last section 7.4 describes our proposed user interface.

7.1 Methodologies

DBRepo is deployed through Docker [Mer14] containers, each containing various APIs, communicating through HTTP requests. These APIs are depicted in Figure 2.1. Since the SO mapper must work synchronously with the DBRepo table schema definition interface, our system needs to be integrated into the same environment. Therefore, the SO mapper back-end is created as an API. The underlying programming language is Python 3 [VRD09] for several reasons. It provides an intuitive syntax which promotes programmers to easily understand and change the underlying code for future works. However, most importantly, Python contains continuously supported, easy-to-use, and well-documented libraries and frameworks for our use cases when it comes to constructing the API, parsing ontologies, or utilizing text embedding models.

To build the API using Python, we utilize the Flask micro web framework [Gri18]. The reasoning behind this choice is that Flask is a lightweight framework, saving resources while enabling faster development for our system. Additionally, we incorporate the Flask-Caching extension. This caching mechanism stores schema-ontology mapping data in temporary files. Administrators can control the system's scalability by restricting the maximum number of simultaneous cache files and defining a timeout period for their deletion. Another extension used for the API is Flask-Cors, which enables communication using Cross Origin Resource Sharing (CORS).

One of the fundamental mechanisms in our system is parsing OWL ontologies. Several Python libraries exist for this purpose, such as RDFLib [Boe18], which is one of the most popular library for parsing RDF data models. RDFLib supports various serialization formats such as RDF/XML, Turtle, N-Triples, etc. However, it does not directly parse OWL 2 vocabulary. For example, when "class A and class B" is being read, RDFLib constructs an intermediate node called a Blank Node (BNode). Subsequently, "class A" is linked with the BNode, which contains the information of "and" and "class B". This limitation is not feasible for our use case, so we are not utilizing RDFLib.

NetworkX [HSSC08] is another Python module capable of extracting information from the structure of complex networks. However, it requires Open Biomedical Ontologies (OBO) vocabulary, meaning OWL2 vocabularies must be converted to be compatible with NetworkX. It's important to note that RDFLib can convert the given graphs into NetworkX graphs. However, the OWL2 vocabulary is still not being parsed directly, and issues such as blank nodes persist.

Owlready2 [Lam17] is another well-known Python module mainly used for parsing OWL2 vocabularies, especially in biomedical ontologies. It can convert classes, instances, object properties, and data properties into Python objects, enabling fetching the relations of entities. Additionally, it introduces reasoning capabilities through HermiT [GHM+14] and Pellet [SPG+07] reasoners for OWL 2 DL with queriying capabilities of inferred classification (e.g., getting the equivalences for a class). Supported formats include RDF/XML, N-Triples, and OWL/XML. Lastly, it features a dedicated SPARQL engine with better performance than RDFlib. Due to these capabilities and native support for OWL2, our system utilizes this library for parsing the given ontologies.

Our similarity metrics come from various libraries and frameworks, divided into two categories for name similarity. The first category, although not utilized in our current system, uses string-based approaches from the *textdistance* library [lif17]. It supports metrics such as Levenshtein distance, Hamming distance, etc., and can be extended to support phonetic-based techniques as well. The second category is based on text embedding models, which convert texts into embedding vectors. There are two frameworks based on PyTorch [PGM+19] that offer a large collection of pre-trained state-of-the-art text embedding models. These frameworks are Sentence Transformers [Reite] and FlagEmbedding [Fla23]. They are currently being used and provide the ability to flexibly change from one embedding model to another by simply changing the model name. Furthermore, PyTorch is used for vector addition, division, and multiplication. The Sentence Transformers framework is utilized for cosine similarity scoring.

SO mapper pre-encodes entity names into vector representations and saves these representations locally. This design choice allows us to compute embeddings only once and then repeatedly use them when similarity calculations are needed. These vectors are saved in HDF5 format using the Python library *h5py* [Col13], which is a de facto data management standard capable of easily reading and writing large amounts of numerical data [The], making it suitable for storing vector representations of thousands of entities. Furthermore, *h5py* enables us to set attributes as metadata, which is used to tag which ontology, embedding model, or encoding method is being used. Conversely, the native Python module called *pickle* [VR20] is used to serialize the Python objects that contain extracted information from the ontologies. This is suitable for our use case since the resulting pickles are less than 100KB, even for the largest ontology we utilize, namely QUDT. This indicates that advanced storage techniques are not necessary.

Several other Python 3 modules are utilized in our system. These include the *regex* module for searching entities by a given keyword, the *logging* module for diagnosing the system in case a bug is encountered, the *hashlib* module for generating schema-ontology specific hash keys, and the *threading* module to introduce multi-threading for calculating the object and unit similarity scores concurrently.

Lastly, our system also needs to update the old table schema creation interface in DBRepo, as depicted in Figure 1.1. This interface requires enhancement to introduce users to freely selecting the mapped entities, especially in cases where the SO mapper fails to find the correct object or unit entities. Therefore, the SO mapper front-end is built on top of the DBRepo schema definition interface. Since the DBRepo front-end is developed using the Vue.js 2.0 [Vue24] JavaScript framework, our front-end utilizes the same framework.

7.2 Architecture

Our architecture is divided into two parts: the back-end is responsible for mapping relevant entities, while the front-end is used for manipulating the schema and interacting with the back-end. The front-end implementation of the SO mapper consists of several components designed to introduce visualization and interaction capabilities. The *Table schema definition* interface from the DBRepo is utilized for integration into the overall DBRepo system. Our design principles aim to provide new interactions and a user-friendly interface. More details about these interactions can be found in the User Interactions Chapter 6. Additionally, the SO mapper front-end component diagram can be seen in Figure 7.1. Apart from the interface, there are mainly three different components used as services: *SO Interaction* service, *SO Communication* service, and *SO Extraction* service.

SO Interaction service is responsible for storing various information as well as requesting API calls from *SO* communication service. The stored information is as follows:

- *Schema hash key*: This key is used to communicate with the SO mapper back-end. Without this key, our system cannot identify which schema-result pairs are being interacted with.
- *Column names and data types*: These are used in manipulating the schema data and reflecting such changes to our back-end. In case the user updates the name or data type of a column, or adds or removes a column, these changes are stored.

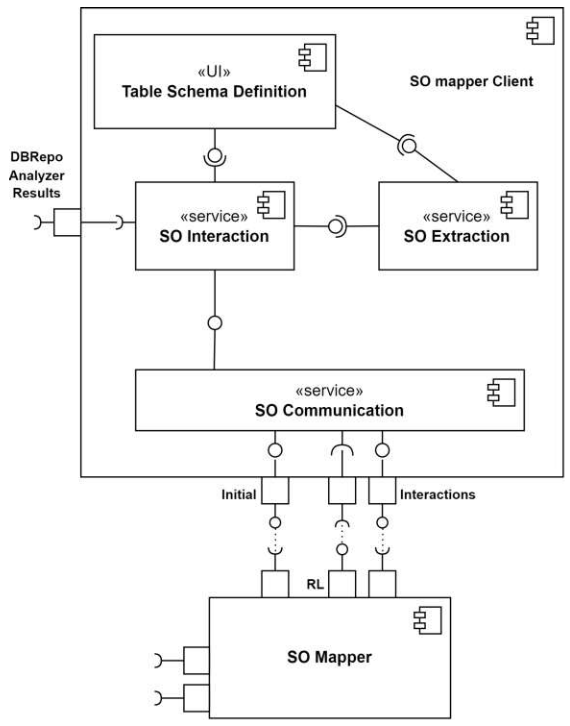


Figure 7.1: SO mapper front-end components

- *Primary keys*: Users can set columns as primary keys. In such cases, these columns are flagged, and their names are sent to the back-end.
- *Ignored Columns*: Columns with "string" data types are detected and ignored for unit mapping.
- *Mapped Entities*: User-selected and auto-suggested schema-entity pairs are stored separately. Note that when a user selects an entity, direct influencing is applied by design. By storing auto-suggested and user-selected pairs separately, our system can distinguish when to apply direct influencing.
- *Mapped ontologies*: Auto-suggested and user-selected ontology selections are also stored. Indirect influencing can be applied by identifying the user-selected object ontology.

The *SO Communication* service is responsible for executing the HTTP API requests to communicate with our back-end. We utilize two functionally separate API call types. The first type occurs when the user initially uploads the file, and the DBRepo analyzer service provides the data types. Subsequently, this service collects column names and their data types from the *SO Interaction* service and executes the Initial Mapping API call. In return, SO mapper back-end responds with a schema hash key and relevance lists.

The second type of API call occurs when the Initial Mapping is done, and the user interacts with the schema or mapped entities. This call is made by retrieving the schema hash key and relevant information for the interaction through the *SO Interaction* service. Finally, the SO mapper back-end responds with relevance lists. For both types of calls, once the schema hash key and relevance lists are received, the *SO Interaction* service is updated accordingly.

The last service for our front-end is the *SO Extraction* service which reads the relevance lists from *SO Interaction* service and passes them to the underlying JavaScript framework. In our case, Vue.js 2.0 combo-boxes (marked as 6 and 7 in Figure 7.4) are populated with entity data are the object and unit entities with the highest similarity scores from each ontology or entities from the keyword search. Further information about the relevance list data can be seen in Relevance Lists Section 5.3.

Meanwhile, the SO mapper back-end consists of several components, all of which are services except the cached data (Fields, Similarity scores, and column embeddings), object and unit Ontology information data ($OI_{obj/unit_*}$), and their embeddings ($E_{obj/unit_*}$). Nevertheless, all these services utilize algorithms described in previous chapters. Our component diagram is depicted in Figure 7.2.

SO mapper back-end provides various configuration options for the initial start-up. These options are namely:

- *Directory locations of Object and unit ontologies*: The object and unit ontologies are stored in separate directories. Each directory contains multiple folders representing different ontologies. Inside each folder, there is an OWL2 file along with parsed object ontology data and computed embeddings generated by our system.
- OWL/XML (".owx") and RDF/XML (".rdx") syntax can be used for reading the ontologies. By default, both of them are enabled.
- *Location of the object and unit ontology reading configurations* $Conf_{obj/unit_*}$: Settings for reading object or unit ontologies can be configured using two separate JavaScript Object Notation (JSON) files. Each file contains multiple objects with their corresponding ontology name, specification of which entity type(s) should be used during parsing, and whether to use the reasoner or not. By default, all of the entity types are parsed and the reasoner is enabled. The location of the configurations can be changed. For further information refer to Section 4.1.
- *Text embedding model (emb)*: Administrators have the option to change the used embedding model. However, vector embeddings from the previous model will still be stored, necessitating manual removal if desired. By default, BGE M3-Embedding [CXZ+24] text embedding model is being used. However, the nasa-smd-ibm-st-v2 [NAS24] text embedding model can be used.
- *Schema, object, and unit encoding methods (enc)*: Encoding methods convert the entity information (label name, relationships) into strings. Afterward, the

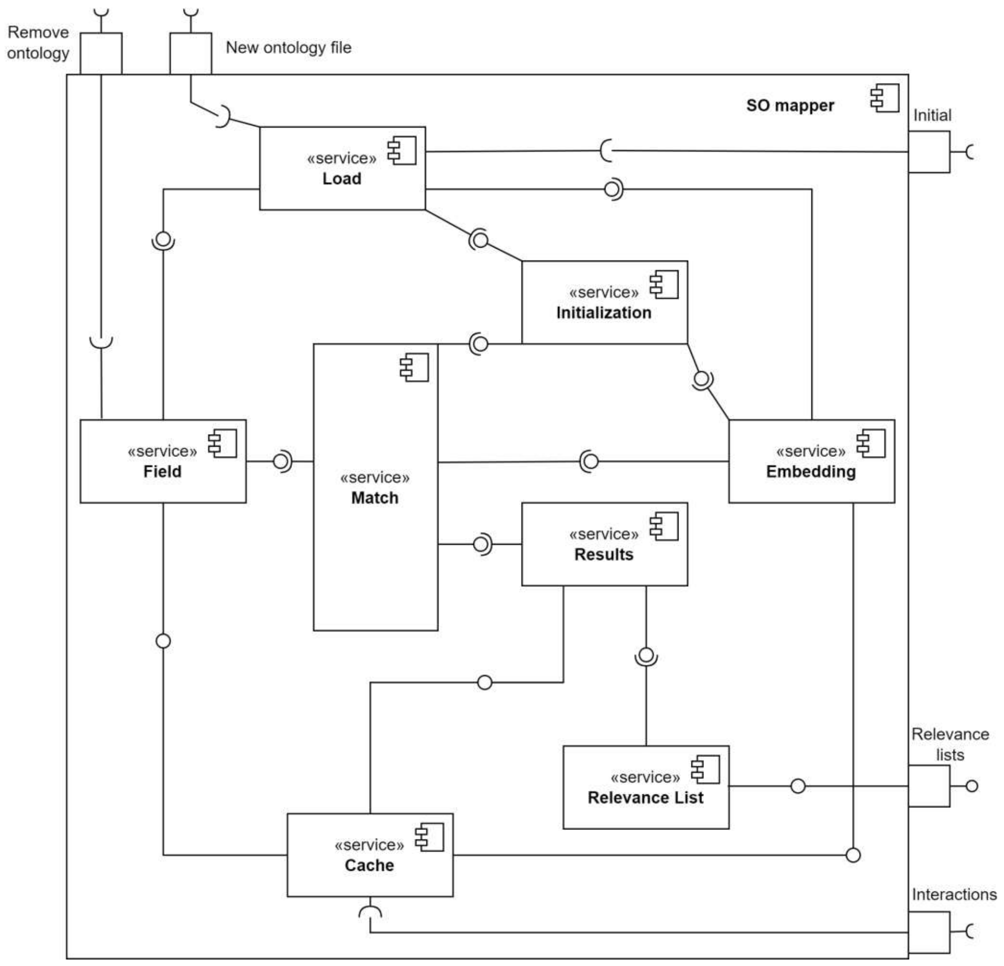


Figure 7.2: SO mapper back-end components

embedding model converts the string to entity embedding. Schema columns, object entities, or unit entities can use different encoding methods; however, the underlying embedding model will remain the same for each of them. By default, the encoding method that is used for the schema columns and object ontology entities is converting the label names into strings. The encoding method that is used for the unit ontologies is converting the entity label names into strings and adding " unit" term at the end of each string. For further information refer to Section 5.2. New encoding methods can implemented into the system such as converting the entity label and label of its ancestor into a string.

- *Object and unit similarity metric*: Used similarity metric within the similarity calculation (*sim_calc*) can be changed, and they can differ for column-object entities and column-unit entities. By default, cosine similarity is used to calculate similarities between column-object entity embeddings and column-unit entity embeddings. Options are using Levenshtein, JaroWinkler, Jaccard, Longest Common Subsequence similarity to calculate column name-entity label name similarities (see Section 8.4).

- *Cache configurations*: Our system utilizes the Flask-Caching extension, and these configurations are also exposed to the administrators. Cache data can be stored data in the file system, memory, or Redis. If stored as files, the directory location of the cache files can also be modified. When a cache is not accessed for a certain amount of time, a timeout occurs and the cache gets automatically deleted. Cache timeout value can be configured or timeout can be disabled. Similarly, the maximum number of concurrent cache files can also be configured. By default, caches are stored as files in the file system, cache timeout is set to 10 minutes, and the maximum number of concurrent caches is set to 50.

The SO mapper starts with the *Initialization* service. It reads the previously defined back-end configurations and provides this information to start other services. The *Load* service is started by receiving directory locations and reading configuration ($Conf_{obj/unit_*}$) locations for both object and unit ontologies. The *Embedding* service is initiated with the embedding model (*emb*) and encoding methods (*enc*) for schema, object, or unit ontologies. Finally, the *Match* service is initiated with the type of scoring methods (e.g., cosine similarity) for the object or unit ontologies. The main responsibility of this service is to read configurations and start other services.

The *Load* service loads the schema data and ontologies. For loading the object and unit ontologies into the memory, the service first loads the reading configurations $Conf_{obj/unit}$ and then loads the ontologies accordingly. This load operation can involve either loading the Ontology information data (OI_i) and ontology embeddings from the file system or parsing the OWL files and computing the embeddings.

In the following cases, the ontology file will be parsed (using Owlready2 and the Entity Discovery and Extraction (*discovery_extraction*) Algorithm 4.1), and ontology embeddings are computed:

- *Ontology information data* (OI_i) *does not exist*: If a new ontology file is placed into the SO mapper object or unit ontology directory, it will not have an Ontology information data in the file system. Thus the ontology file will be parsed (Algorithm 4.1) and embeddings will be computed. Finally, the Ontology information data and the ontology embeddings will be saved to the file system, in the same folder as their respective OWL file. An example data flow can be seen in Figure 3.2.

- *Entity types (e.g., individual, class, ...) in the Ontology information data* (OI_i) *are not equal to the Entity types in the reading configuration* $Conf_i$: The ontology will be parsed (using the new entity types as input to the Algorithm 4.1) and embeddings will be computed. Old Ontology information data and the ontology embeddings are replaced with the new ones in the file system in the same folder as their respective OWL file.

Further information about the reading configurations can be found in Section 4.1 and information about the Ontology information data can be found in Section 4.2.

In cases where the Ontology information data exists (OI_i) and the parsed entity types are the same with the reading configurations $Conf_i$, Ontology information data and the ontology embeddings will be loaded from the file system.

Conversely, the schema data is provided to the *Field* service when schema column names (C) and their data types (D) are received from the client. This transaction occurs when a user initiates the Initial Mapping call. Subsequently, the schema data is received by the *Load* service and then forwarded to the *Field* and *Embedding* services.

The second service initiated by the *Initialization* component is the *Embedding* service. This component is responsible for controlling the embeddings of both schemas columns (E_C) and ontologies ($E_{obj/unit_*}$). It contains a text embedding model (emb), encoding methods (enc), and object or unit entity vector embeddings ($E_{obj/unit_*}$).

A similar load mechanism occurs for the entity embeddings: they can be either read from a previously saved file or new computations can occur. In cases where there is no embedding file in the file system, utilized embedding model or encoding method changes, new embeddings ($E_{obj/unit_*}$) will be computed using the encoding method (enc) and embedding model (emb). These embeddings are then stored as HDF5 in the same folder as their respective Ontology information data. Note that an ontology can have multiple embedding files that use different embedding models and encoding methods. The utilized embedding model and encoding method will be written in the file name.

The schema data comes to the *Embedding* component through the *Load* service. Upon arrival, their embeddings are computed and then sent to the *Cache* component for future use and the *Match* component to execute the Similarity Calculation (sim_calc) algorithm 5.3.

Another responsibility of this service is manipulating the schema embeddings. Such cases occur when the user interacts with the SO mapper front-end. Updating column names causes their respective embeddings to be recomputed and replaced. Likewise, removing columns causes removing the column's embedding. When the Indirect Influence (IndirectInfluence) Algorithm 6.4 is executed, this service computes the weighted averaging between embeddings. Similarly, averaging between embeddings in the Direct Influence (DirectInfluence) Algorithm 6.3 is computed in this service.

Field service controls Ontology information data from all object and unit ontologies ($OI_{obj/unit_*}$). New Ontology information data ($OI_{obj/unit_i}$) can be added to this dataset by executing the Load API request. Furthermore, ontologies can be removed by executing the Remove ontology API request. However, it is important to note that removing an Ontology information data OI_i from the memory does not delete OWL, Ontology information data, and ontology embeddings in the file system. Further information about the API requests is given in Section 7.3.

The *Field* service is used for constructing Fields ($F^{C,O_{obj/unit_i}}$) between the column names (C) and all object and unit ontologies ($O_{obj/unit_i}$) (see Field definition in Equation 3.4). This service is also responsible for executing the Data Constraint Compatibility Check ($DataConstSat$) Algorithm 5.1 which computes the Constraint Satisfaction matrix that holds pairwise compatibility information between column and entity data types.

Furthermore, Field concatenation in the Add Column Algorithm 6.1 and deleting column data from the Field in the Remove Column Algorithm 6.2 are handled by this service.

Lastly, when the user initiates the Direct Influencing Algorithm 6.3, *Field* service is responsible for creating a temporary Fields ($F^{c_x,unit_*}$) between the column (c_x) and entities of each unit ontology. Similarly, for the Indirect Influencing Algorithm 6.4, *Field* service is responsible for creating temporary Fields ($F^{C,unit_*}$) between columns (C) and entities of each unit ontology. More information about the Fields can be found in the Ontology Parsing Section 4.2.

Once the *Field* service provides all Fields ($F^{C,O_{obj/unit_i}}$) between the schema and object or unit ontologies, *Embedding* service provides the column embeddings and entity embeddings ($E_{obj/unit_*}$) then the *Match* service is responsible for executing the Similarity Calculation for Embedding-based Representations (sim_calc) Algorithm 5.3. The output of the algorithm is Similarity scores between columns and all entities from object or unit ontologies ($R^{C,obj/unit_*}$). Note that two Similarity calculations occur, one for object ontologies and another one for unit ontologies.

Furthermore, when the user initiates the Direct or Indirect Influencing, the *Match* service is responsible for executing the Similarity Calculation for Embedding-based Representations (sim_calc) Algorithm 5.3 using the temporary Fields that are coming from the *Field* service.

The *Results* service controls the Similarity Scores $R^{C,obj/unit_*}$. This service has the responsibility to find the entity with the highest similarity score in the Similarity scores $R^{c_x,obj/unit_*}$ for a column c_x. This is utilized in the Construct Entity level relevance list ($const\_ent\_rl$) Algorithm 5.4 and Construct Entity level relevance list on search Algorithm 5.5.

Furthermore, this service filters Similarity scores for entity label names using the given search term. This is utilized to enable users to search entity names in our front-end.

The last responsibility of this service is adding (or removing) the Similarity scores of a column $R^{c_x,obj/unit_*}$ to the Similarity scores of schema columns $R^{C,obj/unit_*}$. These abilities are utilized in the Add Column Algorithm 6.1 and in the Remove Column Algorithm 6.2.

The *Cache* service stores and updates the Fields ($F^{C,O_{obj/unit_*}}$), schema embeddings (E_C), and Similarity scores ($R^{C,obj/unit_*}$). Additionally, it generates a random but unique schema hash key for users to access their cached data. This key is requested from the users for each interaction to ensure correct data is being accessed. Once the Fields, schema embeddings, and Similarity scores are cached for a user, Fields and Similarity scores are sent to the *Relevance List* service.

The *Relevance List* service can generate three types of relevance lists: Entity relevance list (Algorithm 5.4), Entitly level relevance list on search (Algorithm 5.5), and Ontology level relevance list (5.5). More details about these relevance lists can be found in Section 5.3. This component constructs these lists and subsequently dispatches them to our front-end.

7.3 API Requests

Clients can communicate with SO mapper using HTTP requests. Our system employs different types of requests for default users, who are either initiating or manipulating their schema-ontology mapping results. Default user requests are as follows:

- *Initial mapping*: This is a HTTP POST request that occurs on the initial communication from the client to our SO mapper back-end. The JSON request body contains the schema name, schema columns along with their data types, and names of the columns that will be removed from the unit mapping process. The rationale for ignoring these columns is that columns with string data types typically lack units, hence unit entity mapping should not apply to them. In return, our back-end provides a schema hash key as well as entity-level and ontology-level relevance lists for both object and unit ontologies. More information for the relevance lists can be found in the Relevance Lists Section 5.3.

- *Keyword search*: This action involves a HTTP GET request that occurs when the user initiates a search interaction in our front-end interface by entering a keyword in the column's object or unit suggestions drop-down. The front-end sends the schema key, the corresponding column name, and the entered keyword. In response, the SO mapper sends a relevance list containing entity names that begin with the provided keyword.

- *Indirect Influence*: When the user selects an object ontology, a HTTP POST request is sent to the server, containing a JSON request body with the schema hash key, the selected object ontology, and the columns that will be ignored in the Indirect Influencing. This decision is based on the rationale that if the user selects an object entity for a column, indirect influence will not be applied on top of the direct influence. Subsequently, the SO mapper will return a new unit entity relevance list for each column that is not being ignored, as well as the new unit ontology relevance list. Since new entities with the highest similarity score can emerge from the output of Indirect Influencing, the overall unit ontology similarities can change.

- *Direct Influence*: Direct influence takes place when the user selects an object entity for a specific column. Subsequently, the schema hash key, column name, influencer entity for that column, and the entity's ontology are sent to the back-end through a HTTP POST request. The SO mapper will then respond with a new unit entity relevance list and a new unit ontology relevance list. It is important to note that a single API request can handle multiple columns that are influenced by entities belonging to the same object ontology.

- *Remove column(s)*: When a user deletes a column, an HTTP POST request is sent to the back-end, which includes the schema hash key and the name of the deleted column. In return, the front-end receives new object and unit ontology relevance lists.

- *Add column(s)*: When a user introduces a column, another HTTP POST request will be sent. The request body will include the schema hash key, name, and data type of the new column, as well as any Indirect or Direct influences if they were applied. Following this, the back-end processes the request and returns object and unit entity relevance lists for the new column, along with new object and unit relevance lists.

- *Reload*: Finally, a reload occurs when a user sends one of the previously defined requests, which contains a schema hash key. If the cached user data is expired in the SO mapper back-end, the front-end will receive a 404 "schema key not found" error. Subsequently, multiple reload requests will be sent to the back-end one by one. Each reload request contains a directive, which will be in the following order: "initial mapping", "indirect influence" if applied, and multiple "direct influences" if there are any. Additionally, each "direct influence" directive can contain multiple influencer entities belonging to the same object ontology. Upon receiving each reload directive, the back-end returns a success message. If there are no more waiting reload requests left in the client, the client will automatically send the pending API request.

The sequence diagram illustrating these default user requests can be seen in Figure 7.3. These communications ensure that the system updates and synchronizes its data effectively, maintaining consistency between the front-end and back-end for each user.

Another type of communication is designed for administrators, with two different possibilities of requests that are done to the SO mapper. Both requests are HTTP POST. The first one is for introducing new ontologies. This request has an empty JSON body and is sent to the "/load/ontologies" API endpoint. This action prompts the SO mapper to automatically load object or unit ontologies that do not exist within the system but are present in the ontology directories.

The second administrator communication is for removing a certain ontology. The body of this request requires the ontology name and needs to be sent to the "/remove/ontology"

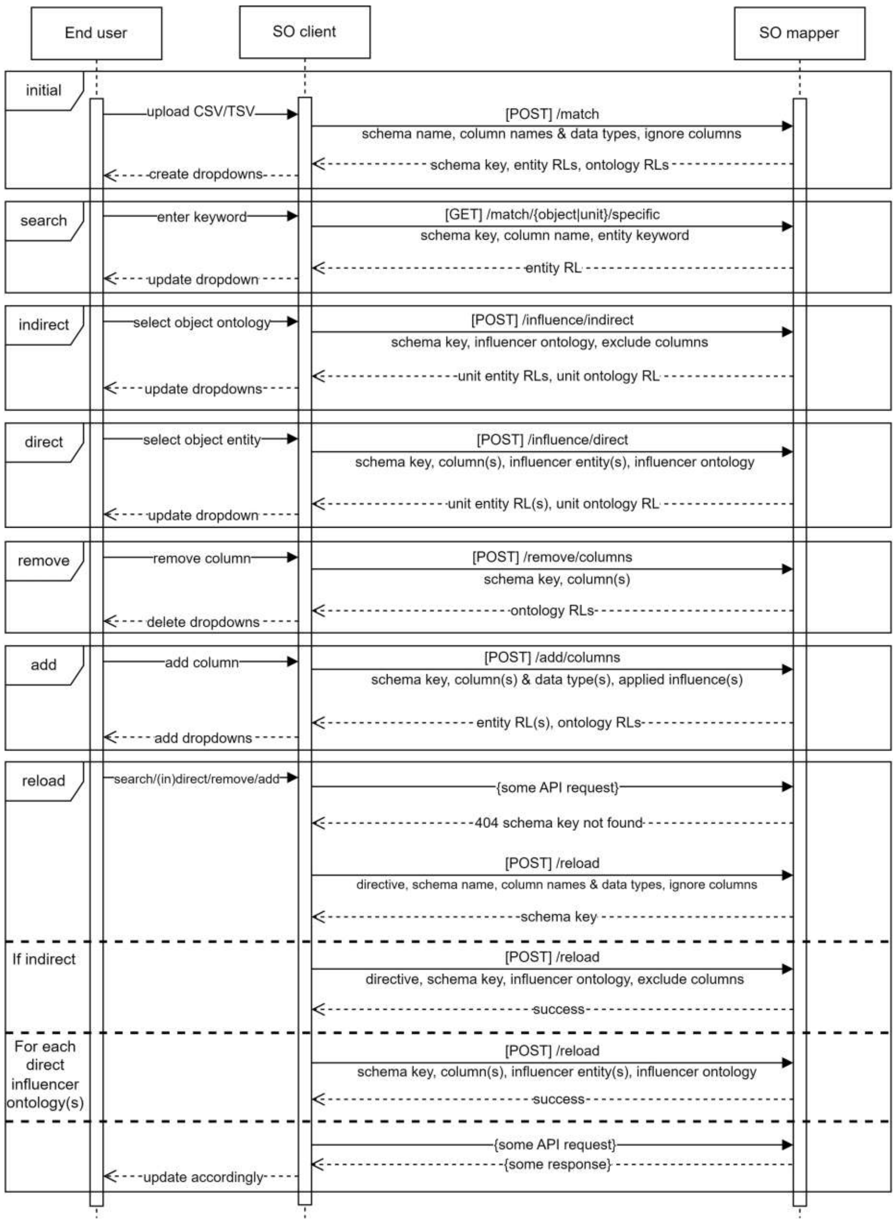

End user
SO client
SO mapper
initial
upload CSV/TSV
[POST] /match
schema name, column names & data types, ignore columns
schema key, entity RLs, ontology RLs
create dropdowns
search
enter keyword
[GET] /match/{object|unit}/specific
schema key, column name, entity keyword
entity RL
update dropdown
indirect
select object ontology
[POST] /influence/indirect
schema key, influencer ontology, exclude columns
unit entity RLs, unit ontology RL
update dropdowns
direct
select object entity
[POST] /influence/direct
schema key, column(s), influencer entity(s), influencer ontology
unit entity RL(s), unit ontology RL
update dropdown
remove
remove column
[POST] /remove/columns
schema key, column(s)
ontology RLs
delete dropdowns
add
add column
[POST] /add/columns
schema key, column(s) & data type(s), applied influence(s)
entity RL(s), ontology RLs
add dropdowns
reload
search/(in)direct/remove/add
{some API request}
404 schema key not found
[POST] /reload
directive, schema name, column names & data types, ignore columns
schema key
If indirect
[POST] /reload
directive, schema key, influencer ontology, exclude columns
success
For each direct influencer ontology(s)
[POST] /reload
schema key, column(s), influencer entity(s), influencer ontology
success
{some API request}
{some response}
update accordingly

Figure 7.3: SO mapper User API call sequences

API endpoint. Subsequently, the SO mapper will remove the object or unit ontology that has the provided name. This ensures that administrators can manage ontologies efficiently within the system.

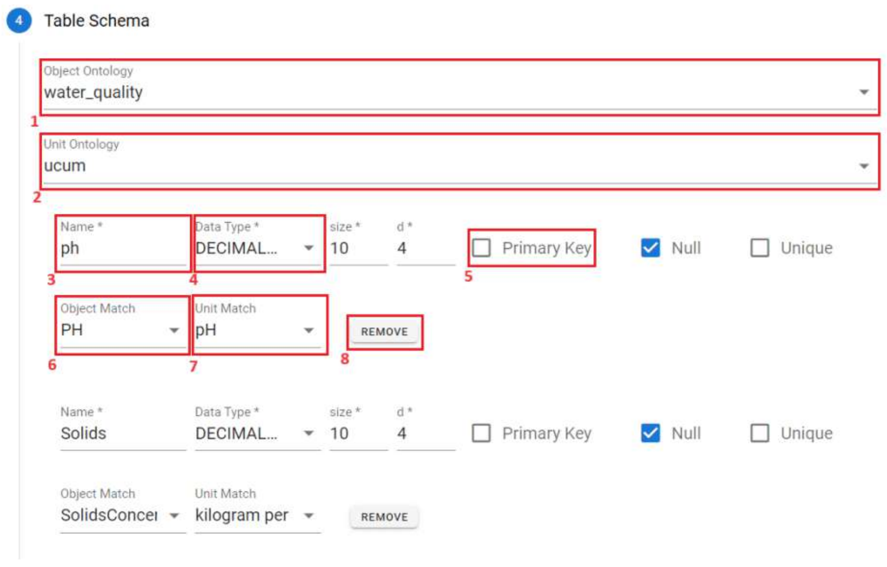


Figure 7.4: New table schema definition interface

7.4 Table Schema Definition Interface

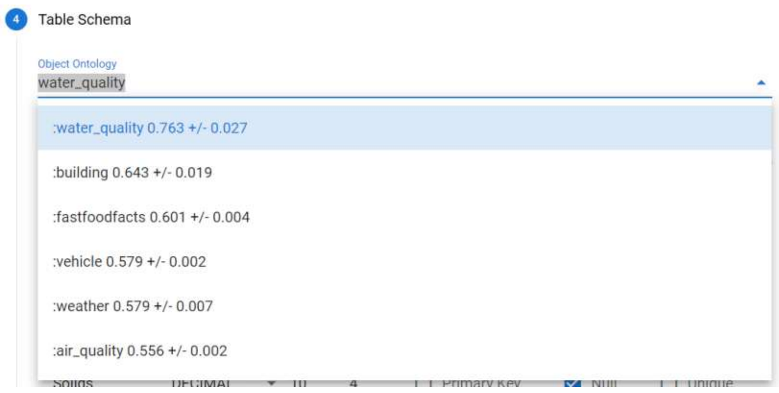
Our front-end is built on top of the DBRepo table schema definition interface. We are introducing two new dropdown lists on top of the interface to present the object and unit Ontology level relevance lists. Additionally, two new dropdown lists are added next to each column, representing unit and object entity mappings for each column. Our interface can be seen in Figure 7.4.

In our front-end, when the user selects an object or unit entity, this selection is treated as verified mapping. These mapping verifications are ignored for the Direct Influencing API request.

Our object and unit Ontology level relevance list in the front-end can be seen in Figure 7.5. When the user manually selects a certain object ontology. An Indirect Influence API request will be executed. This request will ignore the columns with verified object or unit entity mapping.

Our Entity level relevance list based on the keyword search in the front-end can be seen in Figure 7.6. Users can type search terms, and our front-end will execute the Keyword search API request that will display the object or unit entities starting with the search term. The order of the listed entities is based on descending similarity scores.

Direct Influencing API request will occur when an object entity has been verified. That entity will directly influence the respective column's unit ontology similarity scores. However, if the user has verified a unit entity mapping, the Direct and Indirect Influence will not occur. It is important to note that if the column has already been Indirectly Influenced, Direct Influence can still occur on the Initial Mapping similarity scores rather


4 Table Schema

Object Ontology
water_quality

:water_quality 0.763 +/- 0.027

:building 0.643 +/- 0.019

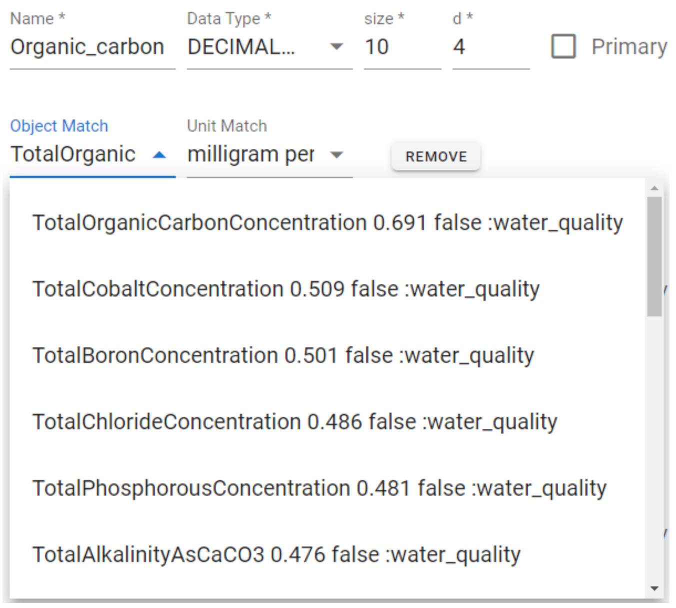
:fastfoodfacts 0.601 +/- 0.004

:vehicle 0.579 +/- 0.002

:weather 0.579 +/- 0.007

:air_quality 0.556 +/- 0.002

Figure 7.5: Ontology level relevance list


Name * Organic_carbon | Data Type * DECIMAL... | size * 10 | d * 4 | Primary

Object Match TotalOrganic | Unit Match milligram per | REMOVE

TotalOrganicCarbonConcentration 0.691 false :water_quality

TotalCobaltConcentration 0.509 false :water_quality

TotalBoronConcentration 0.501 false :water_quality

TotalChlorideConcentration 0.486 false :water_quality

TotalPhosphorousConcentration 0.481 false :water_quality

TotalAlkalinityAsCaCO3 0.476 false :water_quality

Figure 7.6: Entity level relevance list on search

than being applied on top of each other.

Our interface is designed to minimize the required user interactions for achieving correct mappings between all schema columns to object or unit entities. This is done by automatically mapping each column (that does not have a verified object or unit mapping) to the object and unit entity with the highest similarity score. If a column has an entity with a higher similarity score than the automatically mapped entity, the label of the automatically mapped entity will have an asterisk symbol.

Our interface is designed to minimize user interactions needed to achieve correct mappings between schema columns and object or unit entities. It automatically maps each column (without a verified object or unit mapping) to the (object and unit) entity with the highest similarity score. If a column has an entity with a higher similarity score than the automatically mapped entity, the label of the automatically mapped entity will include an asterisk.

For each user interaction, our front-end will memorize the semi-automatically mapped object and unit entities for each column. If a reload is needed, this information is provided to the SO mapper back-end. Once the mappings are finished, the column-object entity and column-unit entity pairs are persisted in the DBRepo metadata database, updating the corresponding schema information. This ensures that the mappings established by the user are recorded and stored for future use within the DBRepo.

CHAPTER 8

Evaluation

The following chapter provides an evaluation of the SO mapper. A custom simulation tool was created for the evaluation environment, which is described in Section 8.1. Subsequently, the selected unit and object ontologies are detailed in Section 8.2. Similarly, the datasets targeting the selected ontologies and their grounding methods are described in Section 8.3. Before the actual evaluation, the applied similarity metrics, embedding models, and encoding methods are presented in Section 8.4. The quantitative evaluation results for the object and unit mappings are shown in Section 8.5. The current limitations of our system are discussed in Section 8.6, and the limitations of our selected similarity calculation approach are given in Section 8.7.

8.1 Simulation Tool

The SO mapper has been evaluated by a custom simulation tool. This tool is capable of running the services and algorithms that are previously defined. It uses the previously defined SO mapper back-end services and can simulate user clicks using the relevance list outputs. This enables thorough testing of the SO mapper's functionality in various scenarios, showcasing its reliability and performance.

At first, each sample table schema data with its ground truth must be stored in separate folders. When our tool is started, it will automatically detect all of the sample scenarios and extract the table schema data using the *Load* service. Ground truths will also be read separately.

Once all of the sample scenarios are loaded, the simulation tool starts by calling the *Initialization* service and sets the configuration parameters as previously described. Afterward, sample columns that do not possess a ground truth can be deleted. This is utilized to mimic user behavior in case the user removes the column rather than mapping it to a special *None* target. Since this is an optional user behavior, we will provide a separate evaluation result for such cases when the absence of a relevant object entity is assured.

Subsequently, initial match, indirect influence, and direct influences occur one after another. They all utilize the *Embedding*, *Field*, and *Match* services for executing the mappings, and *Results* and *Relevance List* services to generate the relevant output information.

The Initial Match does not rely on any ground truth data. In contrast, both Indirect and Direct influences utilize the object ontology ground truths. Indirect influence uses the ground truth of object ontology to mimic user interaction, Indirectly affecting the unit entity similarity scores. This behavior assesses the performance of Indirect influence when the user sets the correct object ontology. Similarly, Direct influence utilizes the ground truths of object entities to mimic user interactions, directly affecting unit entity similarity scores. This behavior measures the overall performance of direct influences when correct object entities are set by the user.

Overall, the Initial Match will assess the accuracy of the mappings of object and unit entities, as well as determine whether the correct object ontology is identified. In contrast, the Indirect and Direct influences will solely measure the accuracy of the mappings of unit entities.

At the end of the Initial Match, Indirect or Direct influence, results are measured in various ways using the ground truths. Firstly, a measurement is conducted to determine whether the suggested object ontology matches the ground truth or not. Then, the mapped object or unit entities are checked to see if they are the ground truth entities or fall within the first 5% or 10%. Furthermore, Mean Reciprocal Rank (MRR) scores are measured to determine the quality of the ranked lists for entities. Finally, the minimum number of user interactions required to achieve the correct mapping of each column-object entity pair and column-unit entity pair is calculated.

The minimum number of user interactions for a column-entity pair is determined by simulating our front-end functionalities. The ontology with the highest similarity score in the ontology-level relevance list is automatically selected, costing zero clicks. However, users can change this selection, which will cost one click.

Similarly, entities with the highest score in entity-level relevance lists are automatically selected for mapping. Additionally, when an ontology is automatically or manually selected, the automatic selection of entities is limited to those within the same ontology. This means that if the correct ontologies and all of the correct entities have the highest similarity scores, the system will function fully automatically without any human interference. However, if the correct ontologies are not automatically selected, this will cost zero clicks. Our simulation will always select the correct object ontology if it is not automatically selected and exists within the list of object ontologies.

If the correct entities do not have the highest scores in the Entity-level relevance lists, users need to expand the drop-downs, which costs one click each, and proceed to search. Searching can be done by typing into the drop-downs and hitting enter, which costs the number of characters typed by the user and an additional key-press to initiate the search. Subsequently, Entity-level relevance lists for search results are received. Each drop-down displays six entities (if expanded) at no cost, and each mouse wheel movement displays two new entities, costing one interaction each.

If a correct entity resides in the list with the highest similarity score (also residing in the selected ontology), it will be automatically selected. If that's not the case, users can continue the search by typing additional characters or scrolling down the relevance list.

Each entity search is simulated by typing the first character of the correct entity, and then recursively continuing the search by either scrolling down or continuing to type. When a search simulation ends, the search combination that costs the minimum number of user clicks is used.

When the correct entity is found, users can select it to map, which costs one click. In cases where there is no correct entity for a column, users have the option to open the drop-down and select the special "*None*" mapping. This functionality is also reflected in our simulation for entities without ground truths, which directly costs two interactions.

It is important to note that several unit ontologies are being utilized within the system. Some of these ontologies have semantically similar (overlapping) entities, all of which are correct unit entities in certain scenarios. This is why, all of the overlapping units are used and the manual selection of a certain unit ontology is not utilized in our simulations.

8.2 Selected Ontologies

Since sample object and unit ontologies are necessary for the evaluation, we searched ontologies on the internet. It is important to note that finding ontologies alone is not sufficient; we also need to find datasets that can appropriately fit into these ontologies.

For our evaluation, we use OM2, QUDT, and UCUM for the unit ontologies. More information about these ontologies can be found in the Ontologies Section 2.2.

For the object ontologies, we employ six different ontologies from the fields of environmental sciences, meteorology, architecture, mechanical engineering, and nutritional sciences. The specific ontologies used are as follows:

- Calidad-aire ontology [Cor19]: The domain of this ontology is environmental sciences. It references air pollutants from the European Environment Agency thesaurus and uses properties from the W3C Sensor, Observation, Sample, and Actuator (SOSA) within the Semantic Sensor Network Ontology [JHC+19] to represent air quality data. It can represent individual pollutants such as PM10, NO2, and O3. There are 44 distinct entities (concept, individual, object, and data properties) within the ontology. Distinctness refers to having a unique IRI within the same ontology. Additionally, unlike the rest of the ontologies, entity names in Calidad-aire are in Spanish.

- Weather ontology [KRK12]: This ontology originates from TU Wien and is an ontology in the meteorology domain. Its main purpose is to represent weather properties for smart home systems. It can represent weather states, reports, conditions (e.g., cloud, fog, rain), and phenomena (e.g., air pollution, precipitation, humidity, temperature). Unlike the Calidad-aire ontology, which extensively covers air quality, this ontology focuses on general weather phenomena and exterior conditions. For instance, this ontology represents air pollution as a European Air Quality Index value instead of individual pollutants. There are 162 distinct entities within the ontology.

- The Doce water quality ontology [Cam19]: This ontology, originating from the domain of environmental sciences, is used to represent the water quality of the Doce River Basin. For example, it includes individuals such as concentrations of aluminum, organic carbon, and sodium. The ontology contains 179 distinct entities.

- Building Information Ontology [KIVK13]: This is another ontology from TU Wien, with its application domain in architecture and partially civil engineering. We use it to represent the building context, including attributes such as area, height, age, energy, and emissions. There are 694 distinct entities within the ontology.

- The Vehicle Core (VC) ontology [veh21]: Produced by the Enterprise Data Management Council, this ontology focuses on the structure, configuration, and emission of various vehicle types. VC incorporates concepts from other vehicle ontologies, such as the Car Ontology [ZIMS15] from Toyota and the Vehicle Emissions Ontology (VEO) [NJT14]. Its domain is mechanical engineering, specifically automotive engineering. VC contains 375 distinct entities such as model type, fuel consumption, CO2 emission, engine power, etc.

- Ontology of Fast Food Facts (OFFF) [AOL+21]: The original aim of this ontology is to standardize knowledge of fast food, fast food chains, and nutritional data for analysis. OFFF includes concepts such as fat, sodium, sugar, calcium, and calories. Its domain is nutritional sciences. OFFF contains 478 distinct entities.

In total, there are 1,932 object entities and 3,811 unit entities utilized for evaluation purposes. Among the employed object ontologies, as well as among the unit ontologies, numerous entities semantically overlap. For instance, the "gram" unit from the OM2 unit ontology overlaps with the "Gram" unit in QUDT unit ontology. In our evaluation, the overlaps in unit ontologies are not considered since we are not using the units but rather evaluating the column-to-unit mapping capabilities of SO mapper (Initial Macth, Indirect, and Direct influencing). For each column-to-unit mapping, in case the unit ground truth overlaps with a unit from another ontology, the overlapped entity is also considered a unit ground truth. A unit evaluation is based on the ground truth that has the highest similarity score. For example the column "Trans Fat (g) " has the unit ground truths "gram" and "Gram". During the Initial Mapping, similarity scores between the column and unit entities are "0.443" and "0.446" respectively. Then the "Gram" unit entity is used in the Initial Mapping column-to-unit entity mapping evaluation. Note that differences such as "$\frac{kg}{cm^2}$" and $\frac{g}{cm^2}$ are not considered as unit overlap.

For the overlapping object entities, only the entity that belongs to the correct object ontology ground truth is considered as object ground truth. However, overlapping object entities cause having multiple potentially correct column-to-object entity mappings. This allows us to demonstrate how the schema-to-ontology mapping determines the most representative ontology. Take, for example, the weather ontology from TU Wien and the Calidad-aire ontology, both contain a temperature entity. However, while the former primarily addresses weather conditions, the latter concentrates specifically on pollutants. The presence of such overlapping object entities allows us to evaluate our system's schema-to-ontology mapping ability.

8.3 Selected Data Sets and Grounding

Upon discovering the ontologies, we proceeded to search for datasets. Following this, ground truth data for each dataset is manually created. This ground truth data includes the correct object ontology name for the overall table data (object ontology ground truth), the correct object and unit entity name for each column (object/unit entity ground truth), and the data type result for each column.

We conduct two evaluation rounds to measure the effectiveness of object and unit suggestions. In the first round, various approaches are evaluated (see Section 8.4). The best-performing approach is then evaluated once more in the second round with different datasets. Each round consists of ten datasets, with at least one dataset for each object ontology. These datasets, along with their manually created ground truth data, are provided to our simulation tool for evaluation. The specific details regarding the datasets are as follows:

- *First round*: In this round, two air quality datasets [Vit16][Poh23] are grounded to the Calidad-aire ontology. Similarly, two meteorology datasets [Mis23] [Gru17] are grounded to the weather ontology from TU Wien. Two water quality datasets [Kad21][Ver23] are grounded to the Doce water quality ontology. Additionally, two nutrition datasets [Arv23] [Sin10] are grounded to the OFFF ontology, and one building dataset [Dat22] is grounded to the Building Information Ontology from TU Wien. Lastly, one automation dataset [Qui93] is grounded to the VC ontology. In total, there are 137 columns, out of which 107 columns contain target object entities and 86 columns contain target unit entities. Columns that do not have any target object or unit entities either provide auxiliary information (e.g., id) or lack a semantically similar entity within the target object ontology (e.g., the Calidad-aire ontology does not include an entity for absolute humidity) or unit ontology (e.g., the city column does not have a measurement unit).

Dataset	Domain	# data columns	# auxiliary columns	Unit names in columns
Air Quality [Vit16]	Environmental sciences	14	2	No
Air Quality Index in Jakarta (2010 - 2021)[Poh23]	Environmental sciences	8	3	No
The Scientific Investigation of Weather in Istanbul[Mis23]	Meteorology	25	7	No
Historical temperature, precipitation, humidity, and windspeed for Austin, Texas[Gru17]	Meteorology	16	4	Yes
Drinking water potability[Kad21]	Environmental sciences	8	2	No
Water Quality Testing[Ver23]	Environmental sciences	5	1	Yes
McDonald's Nutrition[Arv23]	Nutrition	11	2	Yes
Starbucks Nutrition Facts[Sin10]	Nutrition	5	3	No
Manhattan NYC Building Energy Data[Dat22]	Architecture	8	3	Yes
Auto MPG [Qui93]	Automotive	7	2	No

Table 8.1: First round datasets

- *Second round*: As in the first round, each ontology has at least one corresponding dataset to be mapped. One air quality dataset[Gup23] is grounded to the Calidad-aire ontology. Two meteorology datasets [Wer22] [Dey18] are grounded to the weather ontology from TU Wien. Additionally, two water quality datasets [Zub23] [Agr20] are grounded to the Doce water quality ontology, and two building datasets [Red24][Ade21] are grounded to the Building Information Ontology from TU Wien. Two automation datasets [Pod20] [EE24] are grounded to the VC ontology. Lastly, one nutrition dataset [Nem21] is grounded to the OFFF ontology. In total, there are 136 columns, out of which 112 columns have target object entities and 87 columns have target unit entities.

Grounding for each table is performed manually by searching for semantically equivalent entities within the respective ontology for each column and marking which entity it needs to map to. Many of the authors of the selected datasets [Vit16], [Kad21], [Ver23], [Sin10], [Gup23], [Wer22], [Zub23], [Agr20], [Pod20], [EE24] provide additional information about each column. This information offers further insight into the column names and, at times, their unit types. For example, a column from one of the air quality datasets [Vit16] is called "CO(GT)." The authors provided further details for this column, such as "True hourly

Dataset	Domain	# data columns	# auxiliary columns	Unit names in columns
Air Quality in Hyderabad: Pollution Analysis[Gup23]	Environmental sciences	8	2	Yes
Historical London weather data from 1979 to 2021[Wer22]	Meteorology	10	0	No
Jaipur Weather Forecasting[Dey18]	Meteorology	13	0	No
Telangana Ground Water Quality Data[Zub23]	Environmental sciences	16	11	No
Water Quality Measurement Data[Agr20]	Environmental sciences	10	1	No
Fast Food Chains - Nutrition Values[Nem21]	Nutrition	14	1	Yes
Building Dataset for predicting the price[Red24]	Architecture	6	1	No
Location and characteristics data for buildings in the City of Philadelphia[Ade21]	Architecture	6	6	No
CO2 Emission by Vehicles[Pod20]	Automotive	11	1	Yes
Real-world CO2 emissions from new cars and vans, reporting year 2022[EE24]	Automotive	18	1	Yes

Table 8.2: Second round datasets

averaged concentration CO in mg/m^3 (reference analyzer)." Thus, "monoxidoDeCarbono" is marked as the ground truth in the Calidad-aire ontology, and "milligram per cubic metre" is marked as the ground truth in the OM2 unit ontology.

Furthermore, in some datasets [Dat22][Ver23][Arv23] [Gru17] [Gup23][Pod20] [EE24] [Nem21] unit types are written in the column names (e.g., column "Turbidity (NTU)" has unit type "Nephelometry Turbidity Unit"). These same unit types are used in the ground truth assignment to the respective unit entities in the unit ontologies. If there are special characters in column names, they are manually removed. For instance "æ" is converted to "ae". It is also worth noting that all of the utilized datasets contain real-world data, except for one water quality dataset [Kad21], which is synthetically created by the author. Since we do not deal with the data itself and only read the column names, the synthetic nature of this dataset can be disregarded.

In cases where the authors do not explicitly provide information about the measurement units, the cell values are first evaluated to infer the correct unit (e.g., Celsius cannot be 90° as a weather temperature for some province, and $100,000$ millibars of sea level pressure is highly unlikely compared to $100,000$ pascals of sea level pressure). Unit assignments are done as follows:

- *Air quality datasets*: Only one air quality dataset [Poh23] did not contain any unit information. The remaining datasets [Vit16] [Gup23] are used to assign the same units to the same objects.
- *Water quality datasets*: Two datasets [Zub23][Agr20] did not contain any unit information. The remaining datasets [Kad21] [Ver23] are used to assign the same units to the same objects. Additionally in one dataset [Agr20], the unit "milligram per litre" is assigned to "BOD" (biological oxygen demand), and the unit "colony forming unit per millilitre" is assigned to "TOTAL_COLIFORM" and "FECAL_COLIFORM".
- *Automotive datasets*: One dataset [Qui93] did not contain any unit information. Furthermore, it contains an "mpg" column (miles per gallon). This indicates that the units in this dataset are in the imperial system. The remaining vehicle datasets [Pod20] [EE24] are used to assign the imperial system equivalents of the same units to the same objects.
- *Building datasets*: Two datasets [Red24] [Ade21] did not contain the unit information. For the first one [Red24], "square metre" unit assigned to the "Area" column, "metre" assigned to the "Height" column, and "Year" assigned to the "Age" column.

  The second one [Ade21] was coming from Philadelphia, indicating the units should be in the imperial system. "Foot" unit is assigned to the columns "BASE_ELEVATION", "APPROX_HGT", "MAX_HGT" and "Shape___Length". "Square Foot" unit is assigned to the "Shape___Area".
- *Meteorology datasets*: Two datasets [Mis23][Dey18] did not contain any unit information. The remaining datasets [Gru17][Wer22] are used to assign the metric system equivalents of the same units to the same objects. In one weather dataset [Mis23] "Kilowatt hour per square metre" unit is assigned to the column "solarenergy".

Lastly, the "okta" unit which is used to describe the amount of cloud cover, and the "miles per gallon" unit do not exist in our utilized unit ontologies.

8.4 Selected Approaches

Various similarity calculations and influence techniques are selected for the evaluation of object and unit suggestions. Approaches for the initial matching focus on calculating the similarity scores between schema columns and entities. The similarity calculations vary by the underlying similarity metric, text embedding models (*emb*), and encoding methods (*enc*). The tested similarity metrics are as follows:

- *Edit distance*: Also known as Levenshtein distance, measures the number of edits needed to transform one string into another. An N-gram approach is also utilized, which splits the string into n-sized segments and calculates the number of edits within these segments.

- *Jaro-winkler*:Calculates similarity based on the formula $(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}) * \frac{1}{3}$, where m is the total number of matching characters, s_1 and s_2 are the lengths of the strings, and t is the number of transpositions.
- *Jaccard index*: Also known as Intersection-Over-Union, it calculates similarity by dividing the number of intersecting characters by the total number of characters, regardless of their order.
- *Longest common subsequence*: Finds the longest common subsequence between two strings and divides its length by the geometric mean of the lengths of the two strings.
- *Cosine similarity*: Measures the cosine of the angle between two text embedding vectors. Further information can be found in the Embedding Models section 2.6.

Edit distance and Jaro-Winkler are edit-based similarity metrics that focus on individual characters. In contrast, the Jaccard index treats the entire word as a token and calculates similarity accordingly. The longest common subsequence captures sequences of characters between two words. As discussed in the Related Work Chapter 2, these similarity metrics have limitations because column and object names might be synonyms or hyponyms. Additionally, they cannot calculate the similarity of unit entities. External knowledge thesauri can help mitigate this issue to some extent, but finding or creating a thesaurus for each scientific domain is a challenging task. Therefore, cosine similarity combined with text embedding models is also utilized in the evaluation. We primarily focus on two state-of-the-art text embedding models. The first is the BGE M3-Embedding [CXZ+24] (see Section 2.6). It has been chosen because of its performance in MTEB [MTMR22] Semantic Textual Similarity benchmark with respect to its memory usage. It uses a multi-lingual subword tokenizer XLM-RoBERTa [CKG+20]. The second is nasa-smd-ibm-st-v2 [NAS24], which is trained by the datasets from the NASA Science Mission Directorate. These datasets contain data from various scientific research in areas such as the biological sciences, physical sciences, earth science, and applied sciences. It uses an English subword tokenizer RoBERTa [LOG+19].

Furthermore, two encoding methods (*enc*) are evaluated. The first method involves converting the column or entity names into the string and then using the embedding model to create embeddings. The second method appends the "unit" keyword at the end of each unit entity name, inspired by prompt engineering.

For influencing, three different techniques are evaluated. The first two are described in the User Interactions Chapter 6 (namely Direct and Indirect Influence), which involves averaging the embeddings of the column and the selected object entity and then calculating the similarity. The third technique is a variation of schema reuse (see Section 2.5). In our variation, when the user selects the correct object entity, instead of averaging column name and object entity label name embeddings (Direct Influencing), we execute two separate similarity calculations: the first between column names and unit entity labels, and the second between selected object entities label names and unit entity label names. Finally, for each unit, the highest similarity score from the column-unit or selected object-unit similarity score is used as the final similarity score.

8.5 Results

Two evaluation rounds are conducted with different datasets. The first evaluation round covers all of the previously mentioned approaches. Datasets in the first round are used as validation sets to explore these different approaches and select the best-performing one as a default for object and unit suggestions. The second evaluation round covers only the best-performing approaches. Hence datasets in the second round are used as test sets to evaluate the performance of our systems' final form.

Five evaluation criteria are based on the suggestions through entity-level relevance lists. When the ground truth entities fall within the top 10% of the respective column's entity-relevance list, they are marked as covered. Since our system currently utilizes a total of $1,932$ object entities and $3,811$ unit entities, the ground truth must have as high similarity score as to be within the top 193 object entities or 381 unit entities. Furthermore, if the ground truth of a column has the highest similarity score in the respective entity-level relevance list, it is marked as 1^{st}. Otherwise, the ground truth can be ranked between 1^{st} and 5%, or between 5% and 10%. Additionally, MRR evaluation metric is calculated to showcase the overall quality of our entity suggestion ranking system. Note that the MRR is highly dependent on the number of first-placing entities however within our front-end users have the option to search keywords which means the importance of entities that reside within the first 10% is not well reflected with the MRR scores. Our measurements use micro-averaging, which gives equal weight to each column, rather than macro-averaging, which gives equal weight to each dataset. The rationale behind this approach is that, with macro-averaging, datasets with fewer columns would end up giving more weight to each column compared to datasets with more columns. Micro-averaging, on the other hand, eliminates this imbalance and offers a more precise evaluation of our system's overall performance. However, results from the macro average are still shown to provide further information.

Another evaluation criteria is related to the ontology-level relevance lists which examine the total number of correct ontology suggestions. Lastly, the final evaluation criteria is for measuring the performance of our front-end which examines the required minimum number of user clicks for selecting the correct entities. These two evaluation criteria are treated separately due to the nature of our front-end. When a (ground) correct ontology is selected, the highest-scoring entities within that ontology are likely to be (ground) correct. Since our front-end automatically selects the highest-scoring entities, the number of clicks will correspondingly decrease.

8.5.1 Object Mapping Results

Object suggestions are evaluated solely by the initial matching, as indirect and direct influencing does not affect the object relevance lists. Table 8.3 shows the entity-level coverage results. Edit distance, Jaro-Winkler, Jaccard index, and longest common subsequence demonstrate relatively good total coverage. These methods perform well because some columns in our datasets have slightly different name variations, allowing these techniques to achieve relatively good results. However, certain column names, such as "T" (which corresponds to "Temperature"), cannot be accurately matched using these techniques. Conversely, cosine similarity with text embedding models achieves much higher scores. The nasa-smd-ibm-st-v2 model [NAS24] is the best-performing approach for total coverage, achieving a score of 91.7.

Our goal is to achieve the highest total coverage possible as well as achieve high coverage in the top-ranked results to enable full automation which significantly impacts the MRR score. The embedding model from NASA achieves the highest total coverage, but 5.6% of the entities have similarity scores between 5% and 10%. This indicates that the similarity predictions provide relatively lower precision.

In contrast, the BGE M3-Embedding model [CXZ+24] performs differently. It has the highest correct entity coverage in the top rank, with 43.5% of the entities being correctly identified which is also reflected in the total MRR score as 0.5259 being the highest among the rest of the approaches. It also has the second-highest total coverage at 89.8%, and one of the smallest percentages of entities with similarity scores between 5% and 10%.

Approach	Total Coverage (%)	1^{st} (%)	1^{st} - 5% (%)	5% - 10% (%)	Total MRR
Edit distance $n = 1$	75.9	26.9	40.7	8.3	0.3441
Edit distance $n = 2$	74.1	27.8	45.4	0.9	0.3776
Jaro–winkler	67.6	26.9	34.3	6.5	0.3363
Jaccard index	71.3	13.9	**47.2**	**10.2**	0.1857
Longest common subsequence	70.4	21.3	42.6	6.5	0.2624
Cosine Similarity with BGE [CXZ+24]	89.8	**43.5**	45.4	0.9	**0.5259**
Cosine Similarity with Nasa-SMD [NAS24]	**91.7**	40.7	45.4	5.6	0.5093

Table 8.3: First round object entity coverage with micro average

Table 8.4 shows the results of the same evaluation when the macro average is being used. The best-performing approaches are the same as the ones from micro averaging.

Table 8.5 displays the results for the average number of clicks and the number of correctly

Approach	Total Coverage (%)	1^{st} (%)	1^{st} - 5% (%)	5% - 10% (%)	Total MRR
Edit distance $n = 1$	74.9 +-/ 5.4	32.3 +/- 8.3	35.2 +/- 3	7.3 +/- 0.5	0.384 +/- 0.077
Edit distance $n = 2$	73.8 +/- 7	33 +/- 8.2	39.5 +/- 5.3	1.3+/- 0.1	0.407 +/- 0.075
Jaro–winkler	68 +/- 9.3	32.4 +/- 9.2	31 +/- 5.5	4.6 +/- 0.4	0.377 +/- 0.089
Jaccard index	71.1 +/- 6.4	17.4 +/- 5.7	**43.4 +/- 5.3**	**10.4 +/- 1.2**	0.223 +/- 0.071
Longest common subsequence	69.1 +/- 6.5	26.2 +/- 7.8	37.1 +/- 6.3	5.8 +/- 0.4	0.304 +/- 0.074
Cosine Similarity with BGE [CXZ+24]	90.5 +/- 1.3	**51.4 +/- 9.8**	37.9 +/- 7.4	1.3 +/-0.1	**0.593 +/- 0.077**
Cosine Similarity with Nasa-SMD [NAS24]	**92.7 +/- 0.6**	43.9 +/- 4.4	42.7 +/- 1.7	6.1 +/- 0.9	0.538 +/- 0.044

Table 8.4: First round object entity coverage with macro average and standard deviation

selected ontologies from the ontology-level relevance lists. It is important to note that the average character length of the correct object entities in the first round is 13.4 characters.

This evaluation was conducted twice: first, without removing columns that do not have any ground object entities, and second, with such columns removed. A crucial point in this evaluation is that when columns that do not belong to any ontology are included in the match, they can confuse the similarity calculation process by producing misleading entity similarities (e.g., hallucination).

Edit, token, or sequence-based approaches are, generally, not affected when misleading columns are removed. This indicates that while they seem to be robust for hallucination (e.g., edit distance is not affected), these approaches are unable to understand the underlying semantic meaning necessary to detect any changes within the schema.

When users remove misleading columns, the number of correct ontologies identified by the text embedding models increases. The nasa-smd-ibm-st-v2 model achieves a perfect score, correctly identifying all ten ontologies. However, the BGE M3-Embedding model generates a lower average number of clicks, which is expected given that it achieves more first-place rankings in the relevance lists, as shown in Table 8.3. This increase in user clicks indicates that our system can be approximately ten times faster than users manually typing entity names.

In more detail, the nasa-smd-ibm-st-v2 model produces lower values when column-object entity similarities are difficult to identify. These challenging similarities often occur in air quality datasets that are targeted at the Calidad-aire ontology which means the main problem is due to the model using the RoBERTa [LOG+19] which is trained in an

	Without removal			With removal		
Approach	(Micro) Average Clicks	(Macro) Average Clicks	Correct Ontology	(Micro) Average Clicks	(Macro) Average Clicks	Correct Ontology
Edit distance $n = 1$	1.98	1.998 +/- 0.639	6/10	1.98	2.044 +/- 1.024	6/10
Edit distance $n = 2$	2.06	2.097 +/- 1.15	6/10	2.07	2.168 +/- 1.814	6/10
Jaro–winkler	2.38	2.197 +/- 1.003	6/10	2.38	2.3 +/- 1.589	6/10
Jaccard index	2.69	2.598 +/- 0.712	5/10	2.88	2.856 +/- 1.29	4/10
Longest common subsequence	2.35	2.328 +/- 0.781	5/10	2.44	2.49 +/- 1.341	5/10
Cosine Similarity with BGE [CXZ+24]	**1.51**	**1.379 +/- 0.393**	7/10	**1.36**	**1.17 +/- 0.676**	9/10
Cosine Similarity with Nasa-SMD [NAS24]	1.54	1.537 +/- 0.356	**8/10**	1.4	1.408 +/- 0.545	**10/10**

Table 8.5: First round user clicks and ontology coverage with micro average

English dataset. One example would be a column labeled "NMHC," meaning "Nonmethane hydrocarbons," which is defined in the Calidad-aire ontology as "hidrocarburosNoMetano." When these hard similarity calculations arise, the nasa-smd-ibm-st-v2 model generates lower similarity scores. However, the model also incorrectly selects the VC ontology for one of the air quality datasets, with the average of the highest scoring entities from the VC ontology being 0.296, compared to 0.282 for the Calidad-aire ontology. On the other hand, the BGE M3-Embedding model tends to produce much higher values for column-object entity similarities. It incorrectly selects two air quality datasets, with the average of the highest scoring entities from the VC ontology being 0.617, compared to 0.591 for the Calidad-aire ontology.

When the user does not remove misleading columns, the lower similarity scoring behavior of the nasa-smd-ibm-st-v2 model results in more correct ontologies overall. This is because it assigns low confidence to difficult column-entity similarities and misleading columns, so the similarity values of these "hallucinations" are not high enough to affect the selection of the correct ontology. Consequently, the nasa-smd-ibm-st-v2 model achieves better overall accuracy in our evaluations. In contrast, the higher similarity scoring behavior of the BGE M3-Embedding model exacerbates hallucination problems. This results in the BGE M3-Embedding model identifying one less correct ontology compared to the NASA-SMD-IBM-ST-v2 model.

In summary, the lower confidence of the nasa-smd-ibm-st-v2 model increases the system's resilience against misleading or harder-to-detect similarities. In contrast, the higher confidence of the BGE M3-Embedding model leads to higher similarity values for correct column-entity pairs. Consequently, the BGE M3-Embedding model is preferred for ranking higher correct object entities, bringing us one step closer to a fully autonomous mapping system. Another factor favoring the selection of the BGE M3-Embedding model is its performance in unit suggestions and its use of a single embedding model for both object and unit embeddings, which facilitates influence purposes.

In the second round, the average length of the object entities is 14.402. When the misleading columns are not removed, BGE M3-Embedding model achieves 88.4% total coverage (89.8 +/- 2.1 using macro) with first, top 1st - 5%, and top 5% - 10% results as 41.1% (41.0 +/- 11.4), 45.5% (47.0 +/- 8.6), and 1.8% (1.7 +/- 0.1) respectively. Achieved MRR score is 0.486 (0.49 +/- 0.09) which is lower than the first round. The reason behind this difference is that the entities between top 1st - 5% are ranked lower than before. A minimum of 1.779 (1.603 +/- 0.987) average object clicks must be made to achieve perfect mapping. Lastly, 7/10 cases achieved correct ontology identification. In case the misleading columns are removed, the average number of user clicks decreases to 1.723 (1.6 +/- 0.980) and correct object ontology identification increases to 8/10 cases.

8.5.2 Unit Mapping Results

Entity-level and ontology-level unit relevance lists are initially produced by the initial matching. However indirect or direct influencing can occur and change the results of the initial matching. That is why unit mappings are evaluated by initial matching, initial matching combined with indirect influencing, and initial matching combined with direct influencing.

In the unit evaluation, unit information provided in the domain-specific object ontologies is disregarded since units are not acting differently from different domains. For instance, the behavior of the same object can change from scientific domain to domain but units are universal. For this reason, only the unit entities in the unit ontologies are evaluated. Additionally, if a column does not have any object entities, for that column, unit entities are not evaluated.

Note that the average character length of units is 14.26 in the first round. In Table 8.6, it is evident that the BGE model [CXZ+24] achieves the best coverage results, specifically when the encoding method is changed into adding the "unit" keyword at the end of each unit entity. This approach attains 64.4% entity coverage and has the highest percentage of first-place entities at 6.9%. Without using the unit keyword encoding method, the total coverage for the BGE model [CXZ+24] drops by 2.3%, indicating that this method provides a performance boost. Conversely, the nasa-smd-ibm-st-v2 model [NAS24] achieves 57.5% unit entity coverage. Meanwhile, the edit distance method has the lowest total coverage at 31%.

In terms of the MRR score, the BGE model [CXZ+24] with the unit encoding method achieves the highest score among all tested approaches. However, as previously mentioned, a higher MRR score does not necessarily indicate a better suggestion system. The nasa-

Approach	Total Coverage (%)	1^{st}(%)	1^{st}-5%(%)	5%-10%(%)	Average user clicks	Total MRR
Edit distance $n = 2$	31	**6.9**	16.1	8	5.77	0.0839
Nasa-SMD [NAS24]	57.5	1.1	**50.6**	5.7	6.23	0.0615
BGE [CXZ+24]	62.1	5.7	41.4	**14.9**	5.781	0.1049
BGE [CXZ+24] + unit keyword	**64.4**	**6.9**	47.1	10.3	**5.745**	**0.1164**

Table 8.6: First round unit entity mapping results by approaches.

smd-ibm-st-v2 model [NAS24] has a lower MRR score than the edit distance model, but its overall coverage is significantly better.

The reason edit distance has a higher MRR score is that it achieves one of the highest percentages of first-ranking entities. This is because some column names are directly or partially named as unit types (e.g., calories, year built, etc.). Since our text embedding models use subwords in the tokenization, they do not compare the strings as a whole like text distance. Thus the text distance tends to perform better in such exact string matches.

For the average user clicks, the winner is the same but the situation is rather different. BGE model [CXZ+24] with unit keyword encoding leads with 5.745 clicks, followed by the edit distance method with 5.77 clicks. This behavior occurs because of the nutrition datasets [Arv23] [Sin10], where many columns use the "gram" unit. In the OM2 and QUDT unit ontologies, there are approximately 258 unit types that start with the "gram" keyword (e.g., "gram per centilitre"). When users search the "gram" keyword in our UI, the BGE model [CXZ+24] tends to favor units that start with the "gram" keyword (e.g., gram per day) rather than the "gram" unit itself, which are usually ranked between the 1^{st} and 10% positions. This often forces users to scroll down the drop-down menu, requiring many clicks to select the correct unit entity (see standard deviation in the Table 8.9). Conversely, the edit distance method prioritizes shorter character lengths, causing the "gram" entity to rank higher in the search relevance list. Excluding the nutrition datasets, the BGE model [CXZ+24] achieves the best average minimum user clicks with 4.793, while the edit distance method achieves an average minimum of 5.819 user clicks.

Table 8.7 shows the results of the same unit evaluation when the macro average is being used. The best-performing approaches are similar to the ones from micro averaging. However, there is a high spike in the standard deviation of the minimum number of user clicks. The reason behind this is due to the previously explained behavior. For instance, in case the same nutrition datasets [Arv23] [Sin10] are removed from the evaluation, BGE [CXZ+24] with unit keyword performs 4.843 macro average user clicks with 1.933

standard deviation. Meanwhile, the edit distance performs 5.818 macro average user clicks with a 2.015 standard deviation.

Approach	Total Coverage (%)	1^{st}(%)	1^{st}-5%(%)	5%-10%(%)	Average user clicks	Total MRR
Edit distance $n = 2$	33.5 +/- 7.1	7.3 +/- 0.9	16.3 +/- 3.1	10.0 +/- 1.6	**5.667 +/- 2.454**	0.092 +/- 0.008
Nasa-SMD [NAS24]	58.6 +/- 8.7	2.0 +/- 0.4	**51.4 +/- 7.5**	5.2 +/- 0.5	6.272 +/- 8.705	0.077 +/- 0.006
BGE [CXZ$^+$24]	60.2 +/- 3.8	8.6 +/- 0.9	36.8 +/- 4.5	**14.9 +/- 1.4**	5.984 +/- 14.75	0.118 +/- 0.01
BGE [CXZ$^+$24] + unit keyword	**64.9 +/- 6.7**	**9.2 +/- 0.9**	46.0 +/- 2.7	9.7 +/- 1.7	5.791 +/- 12.236	**0.13 +/- 0.012**

Table 8.7: First round unit entity mapping results by approaches with macro average and standard deviation.

In conclusion, because of the performance comparison which is shown in the Table 8.6, cosine similarity using the BGE [CXZ$^+$24] text embedding model with unit keyword encoding approach has been chosen as the default approach. However, further improvement can be achieved with influencing mechanisms.

Table 8.8 compares each influencing method. Influencing is applied to each column. For example, if direct influencing is being evaluated, all columns are directly influenced. Cosine similarity using the BGE [CXZ$^+$24] text embedding model with unit keyword encoding is referred to as initial matching which is also presented in Table 8.8 for comparison.

When users initiate our Indirect Influence method, both the percentage of total coverage and the number of first-ranking unit entities decrease by 1.2% as well as a decrease happens for the MRR. This indicates that the indirect approach reduces overall results. On the other hand, when our direct influencing method is utilized, total coverage increases by 3.4%, while the number of first-ranking entities decreases. Ultimately MRR continues to decrease to 0.1047. This means that direct influencing causes certain correct entities to move into the top 10% while pushing others back. The reason for this behavior is that when the embeddings of column names and object labels are averaged, the encoded information behind the vectors is also combined. This results in both new information and information loss. For instance, in certain datasets [Dat22][Ver23][Arv23][Gru17], unit types are specified in the column names (e.g., the column "Dissolved Oxygen (mg/L)" has the unit type "milligram per liter"). When the evaluation occurs only in those datasets, the initial match scores are 86.1% entity coverage with 11.1% first-ranking entities (0.1973 MRR). Indirect influencing scores 75% entity coverage with 8.3% first-ranking entities and 0.1777 MRR score due to the weighted averaging technique. However, direct influencing has the same coverage but a significant decrease in first-ranking entities, with only 2.8% which further decreases the MRR to 0.1656. The main reason Indirect Influence

Method	Total Coverage (%)	1^{st}(%)	1^{st}-5%(%)	5%-10%(%)	Average user clicks	Total MRR
initial	64.4	**6.9**	47.1	10.3	5.745	0.1164
indirect	63.2	5.7	44.8	12.6	5.803	0.111
direct	67.8	3.4	51.7	12.6	5.774	0.1047
influence with schema reuse	63.2	5.7	41.4	**16.1**	5.81	0.1057
initial + influence with schema reuse	67.8	**6.9**	44.8	**16.1**	5.774	0.1131
initial + indirect	67.8	**6.9**	47.1	13.8	5.752	**0.1192**
initial + direct	**72.4**	4.6	**52.9**	14.9	**5.723**	0.1128

Table 8.8: First round unit entity mapping results by influencing methods and case-by-case combinations with micro average. BGE [CXZ+24] model with unit keyword encoding method is utilized.

experiences a decrease is that some object entities with the highest similarity scores are incorrect. As a result, when their embeddings are automatically weighted and averaged, the similarity scores of the correct unit entities decrease.

If we examine another part of the dataset where unit types are not added to the column names, the initial match scores show 49% entity coverage and a MRR of 0.0592. With indirect influencing, the scores increase to 54.9% entity coverage and an MRR of 0.0639. Direct influencing yields an even higher entity coverage of 62.7%, with a lower MRR than indirect influencing at 0.0616, but still higher than the initial match. Notably, all methods show the same percentage of top-ranking entities at 3.9%.

The significant impact of column names on unit mappings highlights why our front-end is designed to make influencing methods optional. Users have the option to choose which influencing method to use or to opt out of using any influencing methods altogether.

For the influencing method with the variation of schema reuse where the embeddings of a column name and its selected object entity are used to calculate two similarity scores and the highest similarity score is being used (see Section 8.4 for more information). In the first round, it achieves the worst performance. Although it has similar total coverage to our indirect influence, it falls short in the percentage of entities ranked between 5% and 10% which is reflected in its MRR score. In more detail, when only the datasets where columns contain the unit names are evaluated, the influence with schema reuse achieves 8.3% first-ranking entities and 0.169 MRR which is once again lower than the indirect influence. Furthermore, in the evaluation of the datasets with columns that do not contain any unit names, this method achieves the same top-ranking entities (3.9%), the same entity coverage with indirect influence (54.9%) with 0.0607 MRR which is lower than both indirect and direct influence methods.

As previously mentioned, our front-end is designed to make influencing methods optional, allowing for combinations of initial, indirect, and direct influencing, as well as initial matching with schema reuse. These combinations are applied on a case-by-case basis, and the results are also shown in Table 8.8. For instance, one nutrition dataset might use the results of initial matching, which offers higher total coverage, while another dataset from meteorology might use directly influenced results to achieve higher total coverage.

When initial matching and direct influencing are optimally chosen, they achieve the best total coverage at 72.4%, with the lowest number of user clicks at 5.723. For the optimal combination of initial matching with indirect influencing, we observe a lower total coverage score of 67.8%, but one of the highest numbers of first-ranking entities at 6.9% and the highest MRR score.

Notably, in both combination tests, influenced results are preferred in the same eight out of ten datasets in terms of total coverage. However, initial matching is preferred for the other two datasets [Ver23][Arv23], which contain columns with unit names.

Finally, the combination of initial matching and schema reuse has the same percentage of total coverage and first-placing entities as the combination of initial and indirect influencing. However, the main issue with this approach is that the total coverage is achieved by a higher number of entities ranked between 5% and 10%. In contrast, the combination of initial and indirect influencing achieves total coverage with entities ranked within the top 1% to 5%. This is evident in their MRR score difference. The combination of Initial Matching and schema reuse achieves 0.1131 meanwhile Initial Matching and indirect influence achieve 0.1192.

The macro average results of the same unit evaluation can be seen in Table 8.9. Best performing methods are the same with similar results.

In conclusion, direct influencing yields the best overall results but is prone to losing some column information therefore having a lower MRR score. On the other hand, indirect influencing aims to balance this information loss (by weighted averaging) and produce a better MRR but is less effective when used alone (since it automatically selects the first-ranking object entities). Therefore, our front-end allows for a combination of influencing mechanisms on a case-by-case basis. If a column contains essential information, such as unit types, users can bypass the direct influence mechanism by pre-selecting the mapped units. Additionally, our system supports influence combinations within the same dataset (column-by-column combination). Users can select unit entities for some columns in a dataset through initial matching, then apply indirect and direct influencing to further discover new unit entities for the rest of the columns. In our tests, total entity coverage is 64.4% for the initial match, with an additional 3.4% from indirect influencing and another 8.0% from direct influencing can be discovered. This results in a total coverage of 75.8% when influencing methods are applied consecutively to each dataset. Furthermore, the total MRR becomes 0.139 when the correctly mapped first placing entities from

Method	Total Coverage (%)	1^{st}(%)	1^{st}-5%(%)	5%-10%(%)	Average user clicks	Total MRR
initial	64.9 +/- 6.7	**9.2 +/- 0.9**	46.0 +/- 2.7	9.7 +/- 1.7	5.791 +/- 12.236	0.13 +/- 0.012
indirect	60.6 +/- 3.6	6.7 +/- 0.7	44.5 +/- 2.1	9.4 +/- 1.7	5.92 +/- 11.9	0.11 +/- 0.01
direct	64.4 +/- 3.0	5.4 +/- 0.7	47.9 +/- 3.1	11.0 +/- 1.3	5.9 +/- 12.003	0.106 +/- 0.009
influence with schema reuse	62.2 +/- 3.3	6.7 +/- 0.7	41.1 +/- 1.9	**14.4 +/- 1.8**	5.904 +/- 11.744	0.106 +/- 0.009
initial + influence with schema reuse	67.4 +/- 5.6	**9.2 +/- 0.9**	43.8 +/- 2.3	**14.4 +/- 1.8**	5.803 +/- 12.22	0.127 +/- 0.011
initial + indirect	67.4 +/- 5.6	**9.2 +/- 0.9**	46.3 +/- 2.4	11.9 +/- 1.8	5.803 +/- 12.081	**0.133 +/- 0.011**
initial + direct	**71.2 +/- 4.6**	7.9 +/- 1.0	**48.8 +/- 3.3**	**14.4 +/- 1.7**	**5.784 +/- 12.181**	0.128 +/- 0.01

Table 8.9: First round unit entity mapping results by influencing methods and case-by-case combinations with macro average and standard deviation. BGE [CXZ+24] model with unit keyword encoding method is utilized.

the Initial Match are selected, and then direct influencing is applied for the rest of the columns.

Approach	Total Coverage (%)	1^{st}(%)	1^{st}-5%(%)	5%-10%(%)	Average user clicks	Total MRR
initial	60.9	**14.9**	31	14.9	**6.162**	**0.1861**
indirect	64.4	10.3	40.2	13.8	6.243	0.1455
direct	73.6	3.4	51.7	18.4	6.471	0.0882
initial + indirect	66.7	10.3	40.3	16.1	6.243	0.1444
initial + direct	78.2	3.4	**52.9**	21.8	6.493	0.0821
initial + indirect + direct	**79.3**	3.4	**52.9**	**23**	6.485	0.0866

Table 8.10: Second round unit entity mapping results by influencing method case-by-case combinations with micro average. BGE [CXZ+24] model with unit keyword encoding method is utilized.

Table 8.10 shows the evaluation results for testing the performance of the selected influencing methods in the second round. Note that the average character length of unit entities is 14.552. It can be seen that when initial matching, indirect influencing, or direct influencing are optimally used for different cases, a total of 79.3% unit coverage can be achieved. However, the highest MRR score comes from the initial match which has a total coverage of 60.9. The reason behind this difference is the same as before; a lower number of first-ranking entities causes lower MRR. Similarly, when direct influencing is applied to all columns in all cases, 73.6% unit coverage can be achieved, but with a poor MRR score.

Similarly to the first round, we have four datasets [Gup23][Pod20][EE24][Nem21] where the unit types are included in the column names. In our influencing combination evaluations, direct influencing is preferred in terms of total entity coverage for eight out of ten datasets. The remaining two datasets[Gup23][Nem21] contain unit types in their names. Furthermore, indirect influencing achieves the best total coverage in one [Gup23] and the initial match achieves the best total coverage in the other [Nem21] dataset. This shows that the combination of the influencing methods is crucial for achieving the best performance. This combination heavily depends on the given information in each column.

This assumption becomes evident once again when we evaluate only the datasets with unit types added to the column names. The same decreasing trend observed in the first round continues: initial, indirect, and direct influence achieve total entity coverage of 92.5% (0.3901 MRR), 90% (0.3007 MRR), and 82.5% (0.1733 MRR), respectively. This decrease is due to lost information. Furthermore, if we examine the part of the dataset where unit types are not added to the column names, the trend reverses. Without unit type information, the initial match starts with 34% entity coverage (0.0124 MRR), which increases to 42.6% with indirect influence (0.0134 MRR), and further increases to 66% with direct influence (0.0157 MRR).

Lastly, if the influencing methods are optimally applied on a column-by-column basis (e.g., some columns from a single dataset are affected by indirect and the other columns are affected by direct influencing), the initial match provides 60.9% total coverage. Indirect Influence adds 9% more, and Direct Influence adds another 19.5%. In total, our system can achieve 89.3% coverage. Furthermore, the total MRR becomes 0.203 when the first placing entities of the initial match are selected, and then the direct influence is applied.

The table 8.11 displays the results of macro averaging. The best approaches are the same as before.

In conclusion, unit mappings can be generated using a text embedding model, and unit encoding method, and further increased by our influencing methods. Our system also provides the possible strategy of selecting the correctly mapped first-placing entities from the Initial Match and then utilizing the Influencing methods to secure high MRR meanwhile discovering new correct entities. The optional utilization of these influencing methods provides performance boosts. However, users need to consider the information embedded in column names. If the unit type is provided in the column name, initial matching will yield better results; otherwise, influencing methods will enhance the correct unit discovery of our system. Lastly, our front-end offers correct unit entity selection that

Approach	Total Coverage (%)	1^{st}(%)	1^{st}-5%(%)	5%-10%(%)	Average user clicks	Total MRR
initial	62.2 +/- 11.5	**14.0 +/- 6.6**	33.7 +/- 6.7	14.4 +/- 2.6	**6.029 +/- 17.612**	**0.177 +/- 0.075**
indirect	65.7 +/- 9.6	8.8 +/- 2.1	43.9 +/- 8.8	13.0 +/- 2.1	6.085 +/- 17.12	0.139 +/- 0.033
direct	74.1 +/- 4.3	2.3 +/- 0.3	56.1 +/- 11.0	15.7 +/- 3.0	6.23 +/- 16.483	0.086 +/- 0.007
initial + indirect	67.3 +/- 10.2	8.8 +/- 2.1	43.9 +/- 8.8	14.6 +/- 2.7	6.085 +/- 17.12	0.138 +/- 0.033
initial + direct	78.6 +/- 3.4	2.3 +/- 0.3	**56.9 +/- 10.6**	19.4+/- 3.1	6.26 +/- 16.472	0.078 +/- 0.007
initial + indirect + direct	**80.0 +/- 3.4**	2.3 +/- 0.3	**56.9 +/- 10.6**	**20.8 +/- 3.6**	6.25 +/- 16.474	0.084 +/- 0.006

Table 8.11: Second round unit entity mapping results by influencing method case-by-case combinations with macro average and standard deviation. BGE [CXZ+24] model with unit keyword encoding method is utilized.

has an average of 2.2 times faster compared to manually typing out the full entity names.

8.6 System Limitations

The primary limitation of our system arises from the use of Owlready2 [Lam17] and Python. Owlready2 converts OWL2 entities to Python classes, and if the provided ontology includes an inheritance schema that is not permitted by Python (e.g., A is a B, B is a C, C is an A), a "metaclass conflict" error occurs. This prevents our system from parsing the ontology.

Another limitation is the lack of support for enumeration data types. Currently, we treat enumeration data types of the user-provided schema as string data types, even though the elements inside such enumerations can be satisfied with the elements of enumerations within OWL ontologies, our system currently does not support it (e.g., enumeration type "Multifamily Housing" [Dat22] can be satisfied with "Multifamily" enumeration type [KIVK13]). Furthermore, in certain cases, categorical enumerations in the ontology data types implicitly indicate a numerical data type (e.g., the float data type can be satisfied with "GramsPerCubicCm" [KIVK13]). While these two functionalities can be easily implemented in our system, a similarity calculation must be used for such cases which should be determined and evaluated. This is why it has not been implemented and will be discussed in Future Work Section 9.2.

Regarding the unit ontologies, I-ADOPT [MRS+21], discussed in the Related Work Chapter 2, suggests the isolation of a single unit ontology to enhance functionality, such as enabling statistical calculations. However, in our system, we use multiple unit ontologies because there currently does not appear to be any single ontology that is comprehensive enough in terms of unit completeness.

Lastly, in terms of constraint checks, our system does not support "Not" data property declarations. For instance, if a concept has a property "Not string," our system will not capture this behavior. However, such detection is not necessary for us because if our system does not recognize a unit (e.g., string) in the data property deceleration, it is implicitly marked as "not."

8.7 Similarity Calculation Limitations

Our system utilizes the BGE [CXZ+24] text embedding model with a unit keyword encoding approach. In the Evaluation Chapter 8, the performance of object and unit mappings is examined. Although object entity mappings achieve a good MRR score, unit entity mappings have a lower MRR score. Furthermore, the MRR scores are inconsistent across different entities.

This inconsistency is due to the varying rankings of entities. For instance, a well-known entity like Celsius from Meteorology is likely to be suggested within $1^{st} - 5\%$, while a lesser-known entity like oktas is more likely to be suggested within the 5%-10% range or not covered at all. This discrepancy is due to the dataset on which the underlying text embedding model was trained. Therefore, training for a downstream task is a viable solution to minimize these inconsistencies and increase the average MRR scores. However, the training will require column-to-unit mapping datasets from different fields. This topic will be discussed further in the Future Work Section 9.2.

Finally, users can combine our influence methods on a column-by-column basis (e.g., selecting unit entities for some columns in a dataset through Initial Mapping, then applying indirect and direct influencing to discover new unit entities for the remaining columns) to ensure a high Mean Reciprocal Rank (MRR) score meanwhile having extensive coverage, this process is not automated. Currently, the best strategy involves manually selecting the top-ranking entities during the Initial Mapping and then applying direct influencing.

CHAPTER 9

Conclusion and Future Work

This chapter concludes the thesis by summarizing our approaches and the overall system. Finally, potential ideas for further improving our system are presented.

9.1 Conclusion

In this work, we discuss the Schema Column to Ontology Entity Mapping System, referred to as the SO Mapper. This system includes a user interface that operates on top of DBRepo [wei22][WMS+21], where the back-end is built as an API for seamless integration.

The schema column mapping is designed to meet scientific data requirements by mapping columns to both an object and a unit entity simultaneously. By isolating unit ontologies [MRS+21] from multiple object ontologies, it becomes possible to perform statistical calculations across different ontologies (e.g., identifying trends influenced by temperature across various environmental science domains).

Administrators can easily introduce new ontologies to the system or remove existing ones without interrupting its operation. The system allows for optional use of a reasoner when parsing newly introduced ontologies, enabling the identification of constraints or limiting entity types. Further information about the ontology parsing configurations can be seen in the Reading Configurations Section 4.1.

Our user interface is built on top of the DBRepo table schema creation interface. Once a user schema is imported into DBRepo and the DBRepo analysis service predicts the column data types, the SO Mapper utilizes the column names and the predicted data types for each column. Then, an initial similarity calculation is performed between each column and every entity within each ontology in the system as well as automatically checking the satisfiability between column data types and entity data types. Further information regarding the constraint satisfiability check can be seen in Contraint Checking Section 5.1.

The SO Mapper generates two entity-level relevance lists for each column, sorted in descending order by similarity scores: one list for object entities and another for unit entities. Each entry in these lists includes the English entity label (or the entity IRI if no English label exists), the similarity score, the corresponding ontology, and the result of constraint satisfaction between the column and the entity. Additionally, the system provides users with ontology-level relevance lists, which include the ontology name, the average similarity scores of the top-ranked entities, and the standard deviation of these scores for each column. Detailed information on the relevance lists can be found in the Relevance Lists Section 5.3.

Users can interact with our interface in various ways. For example, they can select an ontology or an entity as correct. Selecting an ontology causes the first-ranking entities to be automatically chosen for the columns that the user has not yet selected. Selecting an entity marks it as a mapping between the column and the entity. A full list of user interactions can be found in the User Interactions Chapter 6.

Administrators can configure our system to optionally remove entities with unsatisfied constraints from similarity calculation. Additionally, they can easily customize the similarity metrics for column-object entity and column-unit entity similarity calculations as well as introduce new text embedding models and encoding methods.

We have also proposed a unified approach to calculate similarity between column-object and column-unit entities using a text embedding model called BGE M3-Embedding [CXZ+24] with cosine similarity. Additionally, we have increased the similarity calculation performance for column-unit entities by introducing a new encoding method that simply adds the keyword "unit" at the end of each unit entity.

Cosine similarity with the BGE M3-Embedding model outperformed all other similarity metrics for unit suggestions across all aspects, including MRR, the average number of user clicks required for a full column-unit entity mappings, and covering the correct entities within the top 10%. For object suggestions, our approach also excelled in MRR and the average number of user clicks required for a full column-unit entity mappings, with only a slight reduction in performance for covering the correct entities within the top 10% compared to another text embedding model.

Due to our unified approach for suggesting object and unit entities, we have been able to propose two new methods for improving unit entity suggestions using user-provided input as an auxiliary linguistic resource. The first method, indirect influencing, outperforms an existing method called schema reuse in terms of MRR and the average number of user clicks required for a fully correct match. The second method, direct influencing, surpasses schema reuse in covering the correct entities within the top 10% and in the average number of user clicks required for a fully correct match.

Lastly, our front-end allows the combination of initial matching with indirect and/or direct influencing on a column-by-column basis. Within the same schema, some columns can be mapped to entities using initial matching, while indirect or direct influencing can be applied to the remaining columns. Due to this flexibility in our interface, selecting the first-ranking correct unit entities from the initial match and then applying direct influencing to the rest of the columns outperforms every other approach in terms of MRR while sharing the same amount of highest coverage of the correct entities within the top 10%.

9.2 Future Work

Both the front-end and back-end of our system can be further improved. Additionally, the existing embedding model can be fine-tuned, and a new matching approach can be introduced.

The minimum number of user clicks for selecting the correct entities can be further reduced in our front-end. As explained in the Unit Suggestion Results Section 8.5.2, the higher number of required user clicks is primarily due to the BGE model [CXZ+24] favoring composite units over singular units (e.g., "gram per day" instead of "gram"). Since our front-end search functionality returns entities starting with the given keyword, a user searching for a singular unit like "gram" might be presented with composite units ranked higher than the singular unit.

To address this issue, an additional similarity metric can be implemented, such as Levenshtein distance, to refine the results when similarity scores are close enough to a certain threshold. This would help display the correct entities more accurately.

Our back-end can be improved by utilizing a vector database for storing text embedding vectors, rather than storing them in HDF5 files. This would increase the maintainability of our system in general.

Another improvement for our system would be introducing constraint satisfaction check for enumeration data types. As addressed in the Constraint Checking Section 5.1, there are two cases. First one is checking the constraint satisfaction between column enumerated data type and entity enumerated data types (e.g., enumeration type "Multifamily Housing" [Dat22] can be satisfied with "Multifamily" enumeration type [KIVK13]). This case can be handled by utilizing a text embedding model (e.g., BGE M3-Embedding [CXZ+24]) for text similarity calculation. However there needs to be a threshold for the similarity results to improve accuracy. Other case is checking the constraint satisfaction between column data type (that is not a enumeration) and entity enumerated data types (e.g., float data type can be satisfied with "GramsPerCubicCm" [KIVK13]). For such cases, a text embedding model should be trained to a downstream task to calculate similarity between a text and a data type.

In terms of introducing a new matching approach, certain ontologies include entity annotations that provide additional descriptions of the entities. To leverage this, users can provide separate descriptions for their columns. Similarity can then be calculated between these user-provided descriptions and the entity annotations.

A more straightforward matching approach that can be introduced is a variation of repository of structures [SE05]. When a user provides schema, it can first be matched with previously matched schemas using similarities between their ontology-level relevance scores. Subsequently, schema reuse can be employed, existing column-entity mappings from the most similar previously matched schema can be suggested to the users.

Additionally, the currently utilized text embedding model [CXZ+24] provides inconsistent similarity scores when the target entity occurs less frequently in its training dataset. For instance, the "oktas" unit is more likely to be suggested in the lower ranks. This issue can be resolved by fine-tuning the model for the column-object/unit similarity task. Our system can be used to collect manually selected user mappings to form scientific column-object or column-unit datasets for training. The use of such custom domain-specific training datasets, which are not collected from the web, is also emphasized by OAEI as an interesting approach [HAE+23].

Lastly, the utilized text embedding model can be fine-tuned similarly to the ontology subsumption approach [CHG+23]. User-provided schema columns can be encoded by combining the schema name and column name with the "[SEP]" token. Similarly, entities can be encoded by combining entity names with the names of their descendants.

List of Figures

List of Tables

List of Algorithms

Acronyms

API Application Programming Interface. xvi, 4, 7, 16, 17, 27, 29, 63, 73–77, 80, 82–85, 111, 115

BERT Bidirectional Encoder Representations from Transformers. 24, 25, 47

BNode Blank Node. 74

CEA Cell Entity Annotation. 15, 16

CODATA Committee on Data. 11

COMA Combination Match. 23

CORS Cross Origin Resource Sharing. 74

CPA Column Property Annotation. 15, 16

CSV comma-separated values. 8, 15, 19, 30

CTA Column Type Annotation. 15–17

DAG Directed Acyclic Graph. 12, 13, 23

DBRepo Database Repository. 1–5, 7–9, 14, 15, 17–20, 22, 23, 27, 28, 30, 50, 58, 63, 71, 73, 75, 76, 85, 87, 111, 115

DRUM Digital Representation of Units of Measurement. 11

HDF5 Hierarchical Data Format version 5. 47, 74, 80, 113

HTTP Hypertext Transfer Protocol. 29, 73, 76, 82, 83

I-ADOPT InteroperAble Descriptions of Observable Property Terminolog. 9, 10, 109, 115

INK Instance Neighbouring by using Knowledge. 23

IP Internet Protocol. 14

IRI Internationalized Resource Identifier. 11, 12, 30, 36–41, 43–45, 47, 91, 112

JSON JavaScript Object Notation. 36, 77, 82

KB Knowledge Base. 15, 75

KG Knowledge Graph. 16

MRR Mean Reciprocal Rank. 90, 98–100, 102–110, 112, 113

MTEB Massive Text Embedding Benchmark. 25, 97

NASA National Aeronautics and Space Administration. 10

NER Named Entity Recognition. 16

NIST National Institute of Standards and Technology. 10

NLP Natural Language Processing. 22, 23

OAEI Ontology Alignment Evaluation Initiative. xv, 14, 15, 23, 114

OBO Open Biomedical Ontologies. 74

OFFF Ontology of Fast Food Facts. 92–94

OM Ontology of units of Measure. 10, 11, 29, 32, 35, 91, 92, 95, 103

OWL Web Ontology Language. 3, 10, 29, 37, 74, 77, 79, 80

PCG Pairwise Connectivity Graph. 14

QUDT Quantity Unit Dimension Type Ontology. 10, 11, 29, 32, 35, 75, 91, 92, 103

RDF Resource Description Framework. 11, 23, 39, 74

SemTab Semantic Web Challenge on Tabular Data to Knowledge Graph Matching. 15–17, 115

SO Schema-ontology. 27–31, 33, 35, 36, 46, 47, 50, 55, 58, 63–68, 71, 73–80, 82–84, 89, 92, 111, 112, 115

SOSA Sensor, Observation, Sample, and Actuator. 91

SQL Structured query language. 8

TSV	Tab-separated values. 8, 19, 30
UCUM	Unified Code for Units of Measure. 11, 91
UI	User Interface. 20, 60, 103
VC	Vehicle Core. 37–39, 41, 44, 46, 47, 50, 56, 57, 60, 61, 92–94, 101, 115, 117
VEO	Vehicle Emissions Ontology. 92
W3C	World Wide Web Consortium. 11, 91
XML	Extensible Markup Language. 17
XSD	XML Schema Definition. 41, 43

Bibliography

[ABK+07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *international semantic web conference*, pages 722–735. Springer, 2007.

[ABM15] Fatima Ardjani, Djelloul Bouchiha, and Mimoun Malki. Ontology-alignment techniques: survey and analysis. *International Journal of Modern Education and Computer Science*, 7(11):67, 2015.

[ABP+12] Marcelo Arenas, Alexandre Bertails, Eric Prud'hommeaux, Juan Sequeda, et al. A direct mapping of relational data to rdf. *W3C recommendation*, 27:1–11, 2012.

[AC21] Roberto Avogadro and Marco Cremaschi. Mantistable v: A novel and efficient approach to semantic table interpretation. In *SemTab@ ISWC*, pages 79–91, 2021.

[ACC+22] Nora Abdelmageed, Jiaoyan Chen, Vincenzo Cutrona, Vasilis Efthymiou, Oktie Hassanzadeh, Madelon Hulsebos, Ernesto Jiménez-Ruiz, Juan Sequeda, and Kavitha Srinivas. Results of semtab 2022. *Semantic Web Challenge on Tabular Data to Knowledge Graph Matching*, 3320, 2022.

[Ade21] Adebayo Adejare. Location and characteristics data for buildings in the City of Philadelphia, 2021. Dataset.

[ADMR05] David Aumueller, Hong-Hai Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with coma++. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 906–908, 2005.

[Agr20] Utcarsh Agrawal. Water Quality Measurement Data, 2020. Dataset.

[AOL+21] Muhammad "Tuan Amith, Chidinma Onye, Tracey Ledoux, Grace Xiong, and Cui Tao. The ontology of fast food facts: conceptualization of nutritional fast food data for consumers and semantic web applications. *BMC Medical Informatics and Decision Making*, 21, 11 2021.

[Arv23] Joakim Arvidsson. McDonald's Nutrition, 2023. Dataset.

[AS21] Nora Abdelmageed and Sirko Schindler. Jentab meets semtab 2021's new challenges. In *SemTab@ ISWC*, pages 42–53, 2021.

[BG14] Dan Brickley and R.V. Guha. RDF Schema 1.1 - W3C Recommendation 25 February 2014, February 2014.

[Boe18] Carl Boettiger. *rdflib: A high level wrapper around the redland package for common rdf applications*, 2018.

[Cam19] Patricia Marçal Carnelli Campos. Designing a Network of Reference Ontologies for the Integration of Water Quality Data. Master's thesis, 2019.

[CDPRS20] Marco Cremaschi, Flavio De Paoli, Anisa Rula, and Blerina Spahiu. A fully automated approach to a complete semantic table interpretation. *Future Generation Computer Systems*, 112, 05 2020.

[CHG$^+$23] Jiaoyan Chen, Yuan He, Yuxia Geng, Ernesto Jiménez-Ruiz, Hang Dong, and Ian Horrocks. Contextual semantic embeddings for ontology subsumption prediction. *World Wide Web*, 26(5):2569–2591, 2023.

[CKG$^+$20] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. Unsupervised cross-lingual representation learning at scale. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8440–8451, Online, July 2020. Association for Computational Linguistics.

[Col13] Andrew Collette. *Python and HDF5*. O'Reilly, 2013.

[Cor19] Oscar Corcho. Vocabulario sobre calidad del aire, 2019.

[CXZ$^+$24] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. Bge m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *arXiv preprint arXiv:2402.03216*, 2024.

[Dat22] NYC Open Data. Manhattan NYC Building Energy Data, 2022. Dataset.

[DCF22] Anastasia Dimou and David Chaves-Fraga. Declarative description of knowledge graphs construction automation: Status & challenges. In *Proceedings of the 3rd International Workshop on Knowledge Graph Construction (KGCW 2022) co-located with 19th Extended Semantic Web Conference (ESWC 2022)*, volume 3141, 2022.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[Dey18] Rajat Dey. Jaipur Weather Forecasting, 2018. Dataset.

[DG84] William F Dowling and Jean H Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267–284, 1984.

[DR02] Hong-Hai Do and Erhard Rahm. Coma—a system for flexible combination of schema matching approaches. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 610–621. Elsevier, 2002.

[dru22] Digital representation of units of measurement. *Chemistry International*, 44(3):30–31, 2022.

[EE24] European Environment Agency and European Environment Agency. Real-world co2 emissions from new cars and vans, reporting year 2022, 2024.

[EM07] Daniel Engmann and Sabine Massmann. Instance matching with coma++. In *BTW workshops*, volume 7, pages 28–37, 2007.

[FAI22] FAIRsharing.org. FAIRsharing.org: QUDT; Quantities, Units, Dimensions and Types. `https://doi.org/10.25504/FAIRsharing.d3pqw7`, 2022. Last Edited: May 6th, 2022, Last Accessed: April 24th, 2024.

[FAI24] FAIRsharing.org. QUDT - Quantities, Units, Dimensions and Data Types - Units Vocabulary, 2024.

[FHK$^+$11] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 workshop on array databases*, pages 36–47, 2011.

[Fla23] FlagOpen. Flagembedding: Source code for the flagembedding project, 2023.

[FSG$^+$20] Johannes Frey, Denis Streitmatter, Fabian Götz, Sebastian Hellmann, and Natanael Arndt. Dbpedia archivo - a web-scale interface for ontology archiving under consumer-oriented aspects. In *Semantic Systems. The Power of AI and Knowledge Graphs*, volume 16 of *Lecture Notes in Computer Science*. Springer, 2020.

[GHM$^+$14] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. Hermit: an owl 2 reasoner. *Journal of automated reasoning*, 53:245–269, 2014.

[Gri18] Miguel Grinberg. *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.", 2018.

[Gru17] GrubenM. Historical temperature, precipitation, humidity, and windspeed for Austin, Texas, 2017. Dataset.

[Gup23] Manav Gupta. Air Quality in Hyderabad: Pollution Analysis, 2023. Dataset.

[GWPS09] Christine Golbreich, Evan K Wallace, and Peter F Patel-Schneider. Owl 2 web ontology language new features and rationale. *W3C working draft, W3C (June 2009) http://www. w3. org/TR/2009/WD-owl2-new-features-20090611*, 2009.

[HAE+23] Oktie Hassanzadeh, Nora Abdelmageed, Vasilis Efthymiou, Jiaoyan Chen, Vincenzo Cutrona, Madelon Hulsebos, Ernesto Jiménez-Ruiz, Aamod Khatiwada, Keti Korini, Benno Kruit, et al. Results of semtab 2023. In *CEUR Workshop Proceedings*, volume 3557, pages 1–14, 2023.

[Hey17] Pieter Heyvaert. Ontology-based data access mapping generation using data, schema, query, and mapping knowledge. In *Proceedings of the 14th Extended Semantic Web Conference: PhD Symposium*, May 2017.

[HKN+23] Emil G Henriksen, Alan M Khorsid, Esben Nielsen, Adam M Stück, Andreas S Sørensen, and Olivier Pelgrin. Semtex: A hybrid approach for semantic table interpretation, 2023.

[HKP+09] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F Patel-Schneider, Sebastian Rudolph, et al. Owl 2 web ontology language primer. *W3C recommendation*, 27(1):123, 2009.

[HMVLB20] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python. 2020.

[HSSC08] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[JHC+19] Krzysztof Janowicz, Armin Haller, Simon JD Cox, Danh Le Phuoc, and Maxime Lefrançois. Sosa: A lightweight ontology for sensors, observations, samples, and actuators. *Journal of Web Semantics*, 56:1–10, 2019.

[JI20] Hee-Gook Jun and Dong-Hyuk Im. Semantics-preserving rdb2rdf data transformation using hierarchical direct mapping. *Applied Sciences*, 10(20):7070, 2020.

[JRG11] Ernesto Jiménez-Ruiz and Bernardo Grau. Logmap: Logic-based and scalable ontology matching. pages 273–288, 10 2011.

[Kad21] Aditya Kadiwal. Drinking water potability, 2021. Dataset.

[KIVK13] Mario Kofler, Felix Iglesias Vazquez, and Wolfgang Kastner. An ontology for representation of user habits and building context in future smart homes. In *Proceedings EG-ICE 2013*, pages 1–10, 2013.

[KRK12] Mario J Kofler, Christian Reinisch, and Wolfgang Kastner. An ontological weather representation for improving energy-efficiency in interconnected smart home systems. 2012.

[KS19] Jan Martin Keil and Sirko Schindler. Comparison and evaluation of ontologies for units of measurement. *Semantic Web*, 10(1):33–51, 2019.

[Lam17] Jean-Baptiste Lamy. Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. *Artificial intelligence in medicine*, 80:11–28, 2017.

[lif17] life4. Textdistance - python library for comparing distance between two or more sequences by many algorithms, 2017.

[LOG$^+$19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[LWZ$^+$22] Xinhe Li, Shuxin Wang, Wei Zhou, Gongrui Zhang, Chenghuan Jiang, Tianyu Hong, and Peng Wang. Kgcode-tab results for semtab 2022. In *SemTab@ ISWC*, pages 37–44, 2022.

[Mat19] Sahil Nakul Mathur. Automatic generation of relational to ontology mapping correspondences. *Thesis*, 2019.

[Mer14] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[MGMR02] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings 18th international conference on data engineering*, pages 117–128. IEEE, 2002.

[Mil95] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[Mis23] Ruken Missonnier. The Scientific Investigation of Weather in Istanbul, 2023. Dataset.

[mot12] Owl 2 web ontology language. structural specification and functional-style syntax (second edition). 2012.

[MRS+21] Barbara Magagna, Ilaria Rosati, Maria Stoica, Sirko Schindler, Gwenaelle Moncoiffe, Anusuriya Devaraju, Johannes Peterseil, and Robert Huber. The i-adopt interoperability framework for fairer data descriptions of biodiversity. *arXiv preprint arXiv:2107.06547*, 2021.

[MTMR22] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. Mteb: Massive text embedding benchmark. *arXiv preprint arXiv:2210.07316*, 2022.

[NAS24] NASA-IMPACT. nasa-smd-ibm-st-v2 (revision d249d84), 2024.

[Nem21] Deniz Nemli. Fast Food Chains - Nutrition Values, 2021. Dataset.

[NJT14] Bojan Najdenov, Milos Jovanovik, and Dimitar Trajanov. Veo: an ontology for co2 emissions from vehicles. 09 2014.

[oae] Ontology alignment evaluation initiative (oaei) 2023. `https://oaei.ontologymatching.org/2023/`. Accessed on April 22, 2024.

[oxf] Semantic web challenge on tabular data to knowledge graph matching.

[PAA+20] Mina Abd Nikooie Pour, Alsayed Algergawy, Reihaneh Amini, Daniel Faria, Irini Fundulaki, Ian Harrow, Sven Hertling, Ernesto Jiménez-Ruiz, Clement Jonquet, Naouel Karam, et al. Results of the ontology alignment evaluation initiative 2020. In *15th International Workshop on Ontology Matching (OM 2020)*, volume 2788, pages 92–138. CEUR Proceedings, 2020.

[PBKH13] Christoph Pinkel, Carsten Binnig, Evgeny Kharlamov, and Peter Haase. Incmap: pay as you go matching of relational schemata to owl ontologies. In *OM*, pages 37–48, 2013.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[Pod20] Debajyoti Podder. CO2 Emission by Vehicles, 2020. Dataset.

[Poh23] Taufiq Pohan. Air Quality Index in Jakarta (2010 - 2021), 2023. Dataset.

[Qui93] R. Quinlan. Auto MPG. UCI Machine Learning Repository, 1993. DOI: https://doi.org/10.24432/C5859H.

[RB01] Erhard Rahm and Philip A Bernstein. A survey of approaches to automatic schema matching. *the VLDB Journal*, 10:334–350, 2001.

[Red24] R Kiran Kumar Reddy. Building Dataset for predicting the price, 2024. Dataset.

[Reite] Nils Reimers. Sentence transformers, No date.

[RP16] Petar Ristoski and Heiko Paulheim. Rdf2vec: Rdf graph embeddings for data mining. In *The Semantic Web–ISWC 2016: 15th International Semantic Web Conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part I 15*, pages 498–514. Springer, 2016.

[RVAT13] Hajo Rijgersberg, Mark Van Assem, and Jan Top. Ontology of units of measure and related concepts. *Semantic Web*, 4(1):3–13, 2013.

[SE05] Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. In *Journal on data semantics IV*, pages 146–171. Springer, 2005.

[Sin10] Utkarsh Singh. Starbucks Nutrition Facts, 2011-03-10. Dataset.

[SKB$^+$18] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *The semantic web: 15th international conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, proceedings 15*, pages 593–607. Springer, 2018.

[SPG$^+$07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

[SVW$^+$22] Bram Steenwinckel, Gilles Vandewiele, Michael Weyns, Terencio Agozzino, Filip De Turck, and Femke Ongenae. Ink: knowledge graph embeddings for node classification. *Data Mining and Knowledge Discovery*, 36(2):620–667, 2022.

[Tay95] Barry Taylor. *Guide for the use of the International System of Units (SI): The metric system.* DIANE Publishing, 1995.

[The] The HDF Group. Hierarchical Data Format, version 5.

[veh21] Vehiclecore ontology, 2021.

[Ver23] Shreyansh Verma. Water Quality Testing, 2023. Dataset.

[Vit16] Saverio Vito. Air Quality. UCI Machine Learning Repository, 2016. DOI: https://doi.org/10.24432/C59K5F.

[VK14] Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

[VR20] Guido Van Rossum. *The Python Library Reference, release 3.8.2.* Python Software Foundation, 2020.

[VRD09] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual.* CreateSpace, Scotts Valley, CA, 2009.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[Vue24] Vue.js. Vue.js 2.0, 2024.

[WDA+16] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. The fair guiding principles for scientific data management and stewardship. *Scientific data*, 3(1):1–9, 2016.

[wei22] *DBRepo: A Semantic Digital Repository for Relational Databases*. Zenodo, June 2022.

[Wer22] Emmanuel F. Werr. Historical London weather data from 1979 to 2021, 2022. Dataset.

[WMS04] Chris Welty, Deborah L McGuinness, and Michael K Smith. Owl web ontology language guide. *W3C recommendation, W3C (February 2004) http://www. w3. org/TR/2004/REC-owl-guide-20040210*, 48, 2004.

[WMS+21] Martin Weise, Cornelia Michlits, Moritz Staudinger, Kirill Stytsenko, and Andreas Rauber. FDA-DBRepo: A Data Preservation Repository Supporting FAIR Principles, Data Versioning and Reproducible Queries, July 2021.

[ZIMS15] Lihua Zhao, Ryutaro Ichise, Seiichi Mita, and Yutaka Sasaki. An ontology-based intelligent speed adaptation system for autonomous cars. In *Semantic Technology: 4th Joint International Conference, JIST 2014, Chiang Mai, Thailand, November 9-11, 2014. Revised Selected Papers 4*, pages 397–413. Springer, 2015.

[ZLZP17] Xiaoming Zhang, Kai Li, Chongchong Zhao, and Dongyu Pan. A survey on units ontologies: architecture, comparison and reuse. *Program*, 51(2):193–213, 2017.

[Zub23] Sadiya Zubair. Telangana Ground Water Quality Data, 2023. Dataset.