Semantic Web
XX(X):1–51

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Semantic Integration of Multidimensional Statistical Data: The CubeModeler Framework

Panagiotis Marios Filippidis[1,2], Euclid Keramopoulos[1], Rigas Kotsakis[1], Lazaros Ioannidis[1,2] and Charalampos Bratsas[1,2]

Abstract

Effective integration of heterogeneous statistical datasets remains a key challenge in semantic data publishing. Traditional approaches, ranging from ETL pipelines to OLAP and ontology-based solutions, often struggle with schema rigidity, limited reusability and complex transformation logic. This paper introduces a modeling-based integration approach that shifts the integration effort to the design of modular and reusable Data Structure Definitions (DSDs) within the RDF Data Cube framework. The method follows a clear sequence of modeling steps — including DSD construction, component and codelist definition, dataset description, semantic transformation and SPARQL querying — that support integration directly at the modeling stage. To operationalize this approach, we present CubeModeler, a lightweight semantic modeling environment that enables declarative integration through coded component hierarchies and facilitates dynamic querying via SPARQL over semantically aligned dimensions. Two real-world use cases, sports analytics and environmental measurements, demonstrate how the approach and its implementation in CubeModeler simplifies integration and querying across domains. A set of representative SPARQL queries illustrates its expressiveness in various contextual and temporal aggregations, while a comparative evaluation highlights its workflow simplicity, modular scalability and reusability for semantic multidimensional data integration.

Keywords

knowledge graphs, data cubes, data integration, multidimensional data modeling, SKOS, sports, environment,

[1]Department of Information and Electronic Engineering International Hellenic University, 57001 Thessaloniki, Greece
[2]Open Knowledge Greece, 54352 Thessaloniki, Greece

Corresponding author:
Panagiotis Marios Filippidis, Charalampos Bratsas
Email: filippidis.okfgr@gmail.com, cbratsas@ihu.gr

Prepared using sagej.cls *[Version: 2017/01/17 v1.20]*

1 Introduction

Data integration is the process of combining, sharing, or synchronizing data from multiple sources into a unified view [1]. It is a central requirement for various tasks in data analytics, decision-making and knowledge representation, in terms of analysis and querying. This is especially important in domains where datasets from multiple providers or systems need to be used together. Data integration remains a fundamental challenge in both research and practice due to the variety of data formats, modeling conventions, source contexts and intended uses [1, 2]. While numerous integration methods have been proposed and used in practice, the case of statistical datasets introduces additional requirements related to structure, semantics and interoperability.

Semantic Web technologies and RDF format have emerged as a common language for data integration [3]. Semantic data integration provides the means to achieve the meaningful combination of data necessary to support more complex analysis and conclusions [4]. Ontology-based data integration, in particular, involves the use of one or more ontologies to effectively unify data or information from multiple heterogeneous sources. It directly addresses semantic heterogeneity, a major challenge in integrating autonomous databases [5], by aligning data sources with shared conceptual models. Ontology alignment remains a complex challenge, as it involves determining correspondences between semantically related entities across distinct ontologies [6, 7]. To support this process, a range of interactive and user-assisted environments have been developed to enhance alignment effectiveness and usability [8, 9]. Ontology-based data integration can be seen as a specific method within the broader field of semantic data integration. This field focuses on meaning and context rather than just structure or format.

Statistical datasets are a typical example where integration is needed, as they often originate from heterogeneous sources, differ in structure and granularity and must be queried jointly for cross-domain or comparative insights. These datasets commonly include time-series data, measurements, or indicators and require a data model that supports multidimensional representation. The RDF Data Cube Vocabulary [10], a W3C standard, provides the appropriate framework to express such datasets semantically, with support for dimensions, measures, attributes and observations.

Traditionally, data integration has been addressed using a variety of techniques such as schema mapping, ontology alignment and Extract–Transform–Load (ETL) pipelines [11, 12]. While effective for well-predefined structured environments and closed-world settings, these methods often face significant challenges when extended to domains in which data is highly heterogeneous, semantically loose, or dynamically evolving. These methods usually require extensive upfront alignment and strict schema control. Maintaining interoperability becomes even harder as new data sources are added. This makes them less suitable for cases involving high-volume and inconsistent data that have no direct alignment options. This is also the case for statistical data, which often rely on metadata and contextual dimensions such as time, location, or measurement unit. These are core components of their structure, essential for interpreting and comparing values in semantic data integration.

The increasing volume and variety of statistical datasets published in decentralized ways highlights the need for a consistent and streamlined data integration process. These datasets often differ in their schema definitions, use custom identifiers or code systems and are described in isolation, hindering their combined use or comparative analysis. There is a growing need for a scalable, semantically aware data integration approach that is flexible enough to accommodate diverse data sources and robust enough to enable consistent querying and reasoning over them.

Within this context, an alternative modeling-based approach to data integration is explored, shifting the integration process to the modeling stage. Instead of transforming and aligning datasets during a post-processing phase, each dataset is semantically described using its own Data Structure Definition (DSD), structured according to a shared and extendable data model. Integration is then achieved at query time through common or hierarchically related components of the model, leveraging SPARQL queries over a knowledge graph that contains the datasets, their structures and the relevant codelists. This paper presents such a modeling-based integration approach using RDF Data Cubes and shows how it supports semantic alignment and interoperability across statistical datasets.

In practice, statistical data often appear as multiple datasets, each with its own `qb:dataSet` and `qb:DataStructureDefinition`. These datasets may rely on different codelists, identifiers, or modeling conventions, reflecting the diversity of their origins across domains, organizations, or data providers. Achieving integration in this setting requires a shared semantic foundation that enables unified access and querying across structurally and contextually heterogeneous data sources.

The proposed approach is demonstrated through two distinct use cases. The first involves basketball statistics, where data is centrally curated and structurally consistent across seasons and leagues. The second involves environmental datasets from public administration, which exhibit greater heterogeneity, decentralization and variation in structure and granularity. Both cases are addressed through modular data models, reusable components and alignment mechanisms based on RDF properties and SKOS codelists to ensure semantic interoperability and flexible integration.

By structuring integration around the modeling process and adopting semantic web standards for dataset representation, the approach promotes modularity, reuse and reasoning-based integration in a scalable way. Reasoning over the shared model enables coherent access and querying even across diverse datasets, providing a suitable method for statistical data domains. It aligns well with modern principles of data management and publishing, including the FAIR principles [13], and offers a practical solution to the challenge of interoperable statistical data integration.

To make the proposed approach transparent and reproducible, the workflow is organized into a sequence of methodological steps that guide users from modeling to semantic transformation: (i) defining a Data Structure Definition (DSD), (ii) creating or reusing components and codelists, (iii) describing dataset-level metadata, (iv) performing semantic transformation (RDFizing) using the established model, and (v) executing SPARQL queries for integrated access. These steps form the methodological backbone of the approach and are implemented in CubeModeler.

The remainder of this paper is structured as follows. Section 2 reviews related work on data integration, semantic modeling and the use of RDF Data Cubes. Section 3 introduced the modeling-based integration approach in detail, including the architectural rationale and the crucial role of Data Structure Definitions and data modeling principles with components hierarchies. Section 4 presents the methodology steps for data modeling, semantic transformation and semantic alignment in practice, through the CubeModeler and its core functionalities. Section 5 showcases the two main use cases - basketball statistics and environmental data - demonstrating how semantic integration is achieved in each context. Section 6 outlines the evaluation strategy and Section 7 concludes with the final remarks.

2 Related Work

Data integration has long been a central topic in database systems and knowledge representation. Classical approaches include schema mapping, entity resolution and Extract–Transform–Load (ETL) pipelines [1, 11]. These methods aim to align schema and data-level mismatches through structural mappings and transformation rules and have been widely applied in traditional data warehousing and IT infrastructures. However, such techniques often struggle when applied to decentralized, heterogeneous and semantically ambiguous sources, especially when data evolves or new sources are introduced dynamically [2].

Despite significant progress, these approaches typically expect well-defined schemas and preconfigured controlled environments, making them less suitable for open or decentralized contexts where data is described using diverse structures from various sources. Integration at this level often requires time and effort for schema alignment and maintenance, with limited support for semantic reasoning or dynamic interoperability.

To address these limitations, semantic data integration introduces meaning-aware mechanisms for aligning data across heterogeneous sources. It leverages standards such as RDF, OWL and SKOS to explicitly represent the semantics of data and to support integration at the conceptual level. Semantic data integration allows for meaningful combinations of datasets that enable more expressive analysis and inference [4]. This approach is particularly appropriate to cases with high structural heterogeneity, but also common conceptual ground among datasets.

Within this domain, ontology-based data integration emerges as a more formal subclass, using ontologies to mediate between schemas and unify data under a shared conceptual framework [5]. Ontologies offer a powerful formal modeling mechanism by providing standardized definitions and hierarchical relationships among concepts, enabling reasoning over aligned entities. Still, the effectiveness of ontology-based integration depends on the availability, alignment and maintenance of ontologies, which in practice can become a limiting factor, particularly in domains with evolving or loosely defined semantics [12].

Ontologies and vocabulary alignment tools have been extensively developed to support semantic interoperability [14] as well. Linking vocabularies and classifications provides an additional layer to the data integration process by mapping similar or related terms, thereby facilitating the broader integration task. From academic research [15] to sports analytics [16] and the fiscal domain [17], such classifications may play a pivotal role in enabling knowledge integration and semantic alignment.

In addition to traditional and ontology-driven integration strategies, several tools have been developed to support the semantic transformation of structured data into RDF. Karma [18] offers a semi-automated environment for mapping relational or tabular data to ontologies, combining user feedback with machine learning to suggest class and property alignments. D2RQ [19] and Morph-RDB [20] enable virtual RDF views over relational databases by defining R2RML-style mappings, allowing SPARQL queries without physical transformation. RMLMapper [21] generalizes this approach by supporting multiple input formats (e.g., CSV, JSON, XML), while SPARQL-Generate [22] extends SPARQL to express both transformation logic and data extraction from heterogeneous sources. Lightweight tools such as Tarql [23] focus on generating RDF directly from CSV or spreadsheet-like data, with minimal schema alignment. While these tools facilitate RDFization, they typically emphasize transformation over modeling and do not offer modular component reuse or hierarchical integration (as supported in the proposed approach).

Multidimensional data integration has been widely studied in Online Analytical Processing (OLAP). The goal is to combine fact tables and dimension hierarchies to support analytical tasks across domains such as sales, finance and public data [24]. OLAP integration typically involves schema mapping between dimensions and the alignment of temporal or geographic hierarchies. While these approaches support powerful aggregate queries and drill-down operations, they often rely on rigid schema definitions and centralized design, making them difficult to extend or adapt to weakly or semi-structured or evolving data sources.

Adapting multidimensional schemas to heterogeneous environments remains a persistent challenge [25]. Traditional OLAP tools lack support for semantic interoperability, offering few mechanisms for aligning concepts across diverse and dynamic schemas. OLAP and data warehousing systems reveal significant limitations when applied to the analysis of big multidimensional data, due to their reliance on static structures with limited support for inference [26]. In practice, this makes OLAP models difficult to maintain in decentralized settings under which data sources frequently evolve or diverge. OLAP operations have been mixed with RDF Data Cubes in various operations in practice [27].

The RDF Data Cube Vocabulary [10] provides a W3C standard for semantically modeling multidimensional data using RDF. It enables the definition of datasets, observations, dimensions, measures and attributes within a graph-based structure, making it well-suited for publishing statistical data in a machine-readable, interoperable format. The vocabulary has been widely adopted by national and regional authorities, including Eurostat and the UK Office for National Statistics (ONS), for releasing open government data.

However, these implementations often rely on standalone Data Structure Definitions (DSDs), with limited support for cross-dataset integration or shared semantics. Typically, each dataset defines its own schema without coordination and the absence of shared schema alignment restricts opportunities for reasoning or semantic querying across sources. Even extensions like QB4OLAP [28] have focused primarily on enabling SPARQL-based OLAP operations, without addressing the broader challenges of schema modularity and semantic integration.

In the research and tooling landscape, platforms such as CubeViz [29], part of the LOD2 project and OpenCube [30] have focused on enabling interactive exploration and visualization of RDF Data Cubes. These tools are designed for consuming statistical data that has already been modeled and transformed. They support interaction with predefined data cubes, allowing users to explore datasets across multiple dimensions or visualize summaries. Although effective in visualizing semantically structured data, CubeViz and OpenCube do not offer built-in support for schema construction, semantic alignment, or data transformation workflows, assuming instead that modeling and integration have been completed externally.

Efforts such as SSN-QB mappings have extended RDF Data Cubes to handle sensor and observational data [31], illustrating the vocabulary's versatility beyond traditional statistics. SSN [32] is a standard for describing sensors, actuators, observations and related concepts, providing a framework for representing the characteristics of sensors, the measurements they produce and the context in which they operate. However, even in those contexts, the emphasis is on data publishing rather than on semantic modeling or integration. The result is a pattern where RDF Data Cubes are used independently, often lacking a shared or extensible data model and without leveraging the hierarchical and semantic capabilities of RDF vocabularies such as `rdfs:subPropertyOf` or `skos:broader` and `skos:narrower`.

Recent advances further extend the landscape of semantic integration methods. The rise of large language models (LLMs) has introduced new opportunities in ontology matching and schema alignment, tasks directly related to the reuse and extension of components in modular data models. Approaches such as LLMs4OM [33] demonstrate that retrieval-augmented prompting strategies can outperform classical matchers in complex ontology alignment scenarios. Frameworks like Agent-OM [34] apply multi-agent LLM architectures to achieve accurate few-shot ontology matching. Comparative studies using GPT-3.5 and Flan-T5 models [35] highlight the potential of LLMs to support alignment and ontology reuse when carefully guided through prompting. Beyond matching, LLMs have also been applied to broader ontology and knowledge graph construction [36], proposing an LLM-supported pipeline that automates competency question generation, ontology design and KG population. These methods point towards AI-assisted strategies that could, in the future, be integrated into modeling workflows similar to the approach proposed in this paper, reducing manual alignment effort when building or enriching Data Structure Definitions.

In parallel, federated learning has recently been applied to knowledge graph embeddings, addressing privacy and scalability in distributed integration settings. Systems such as FedCKE [37] explored cross-domain knowledge graph embeddings in federated settings, showing how distributed representation learning can be achieved without centralizing sensitive data. FedCQA [38] enable federated complex query answering across multiple graphs, while FedMDKGE [39] introduces multi-granularity dynamic knowledge graph embedding, extending federated methods to temporal contexts. Although still emerging, these directions suggest complementary techniques for integration scenarios in which datasets remain distributed, contrasting with the paper's current focus on integration at the modeling stage within shared knowledge graphs and modeling environment.

Finally, extensions to RDF Data Cube Vocabulary itself reflect a growing demand for richer, real-time statistical publishing. The QB4ST extension [40] defines standardized ways to incorporate spatial and temporal metadata into Data Cubes, while the STAC Datacube Extension [41] adapts cube-style modeling to spatio-temporal assets such as Earth observation streams. These developments underline emerging directions toward dynamic, context-aware semantic data publishing, suggesting possible avenues where CubeModeler's modeling-driven approach could be adapted for streaming or sensor-based datasets.

Such advances highlight the growing landscape of semantic integration, yet they do not address key requirements for integration. These requirements include: (i) model modularity and reuse, where dimensions, measures and attributes can be shared or extended hierarchically across datasets; (ii) semantic alignment, enabling coded components to be linked consistently across domains or providers; (iii) configurable and repeatable transformations, so that new datasets can be integrated with minimal effort; and (iv) expressive and reusable querying, allowing cross-dataset SPARQL queries without detailed knowledge of each dataset's schema. Such requirements are crucial both for data publishers, who need scalable ways to release heterogeneous datasets in a consistent and reusable form and for data consumers and analysts, who depend on integrated access for cross-domain or comparative insights. Without explicit support for these dimensions, RDF Data Cube-based solutions risk remaining fragmented and underutilized for integration tasks.

The approach proposed in this work emphasizes modeling as the core of the integration process, incorporated directly into the design and transformation stages. Using modular data models with interconnected and reusable modeling elements, each dataset is defined through its own Data Structure

Definition but within a shared semantic space. Integration takes place dynamically during querying through alignments between components and codelists, without requiring rigid schema unification.

To support this approach in practice, the CubeModeler framework has been developed, as the operational environment that implements this modeling-based integration approach. This framework makes it feasible to define reusable components, construct DSDs and carry out semantic transformations in a repeatable way across heterogeneous datasets. It provides end-to-end services for constructing Data Structure Definitions, importing and linking codelists and transforming heterogeneous datasets into RDF - a well-established standard for semantic representation that is leveraged in this approach - with guided workflows. The framework has been designed to be user-friendly and semi-automatic, enabling the entire process to be carried out without specialized engineering skills or advanced knowledge of semantic technologies. By embedding the integration logic into the modeling process, CubeModeler lowers the effort of publishing interoperable Data Cubes and establishes a foundation for consistent, reusable and scalable semantic integration.

3 Materials and Methods

3.1 Data Integration with Data Cubes

An alternative method for the data integration task is proposed for statistical datasets, containing data that can be described through data cubes. This method has been applied to heterogeneous environmental datasets from the Public Administration pilot of the UPCAST project, but also to basketball datasets from various leagues, as the two major use cases of the current work, that showcase the application of the approach. The method can be applied to any other statistical use case as well.

The vast and diverse volume of data originating from various sources led to the search for a more targeted data integration solution tailored to the specific needs of statistical datasets that coexist in a common space and have a high degree of probability, or need, to interconnect. When the number of data providers is not fixed, new datasets with unknown content may arrive at any time. To handle this, our approach maps all heterogeneous datasets into a unified data model within a knowledge graph. This is to ensure a common representation across all data resources.

Thus, instead of applying traditional data integration methods for statistical datasets, the integration process, at least its initial phase, was shifted to the data modeling stage. In this way, all components that represent the data conform to a shared data model. As a result, links between different datasets are inherently established and integration takes place at query time, when asking for the data based on their common and related properties. SPARQL queries over the unified model become the main mechanism for integration.

This approach also stems from the fact that in many cases, the development of related data schemas and ontologies from different resources that would traditionally serve as inputs for conventional integration methods, is neither feasible nor realistic for their providers. Instead, a unified data model allows for the addition of new concepts (such as dimensions, measures, or attributes) when necessary, linking them through subproperty relationships.

It becomes evident that special attention is required in designing an appropriate, robust and adaptable data model to serve as the semantic backbone of the diverse data representation and their integration. To this end, the RDF Data Cube Vocabulary was selected as the foundational framework for modelling the

various statistical datasets with a common data model, onto which all incoming data would be mapped for each use case.

Each dataset has to be associated with its own Data Structure Definition (DSD), that is the schema in the Data Cube Vocabulary, allowing for schema flexibility while maintaining consistency at the model level. Datasets with similar structures may share DSDs, but in general, each is modelled independently to reflect the specific characteristics of the data source. The core idea is to unify data access through a shared set of common or related components, without forcing a rigid schema across all resources, instead, they can be aligned to the data model through DSD construction include related or common components. Dataset metadata provides additional linking possibilities, as depicted in Figure 1.

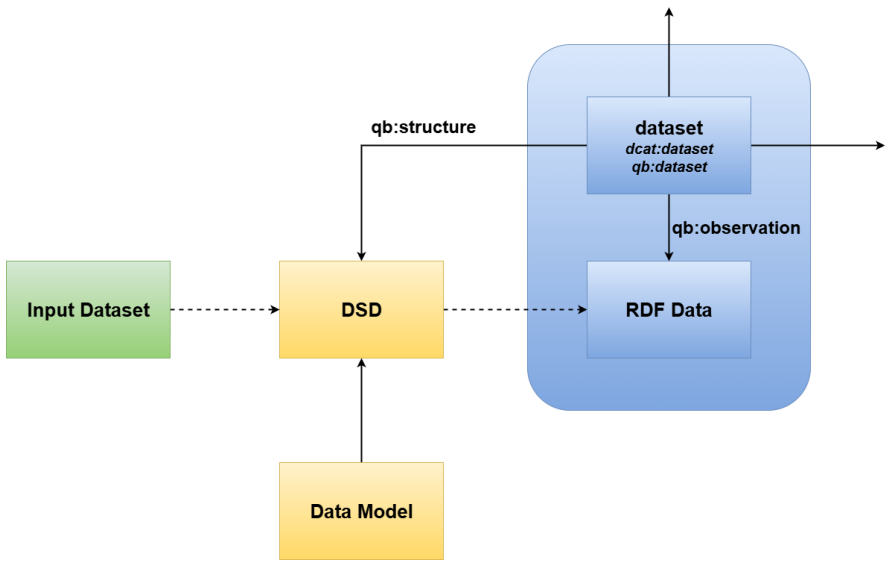

Figure 1. Abstract data procedures architecture.

All data, namely the semantic representation of the dataset in RDF, its corresponding DSD, but also the dataset metadata, are imported into a knowledge graph in a triple store. The triple store is a database management system for RDF Data, where also the data model, all its components and the respective codelists are included. A Virtuoso RDF triple store has been used for this task.

SPARQL queries can be written against any component in the model, as long as the corresponding DSD of a dataset has been defined. Thus, by defining the `qb:structure` property (indicates the DSD to which this dataset conforms) for the dataset RDF, the relevant DSD and all related components can be found. This, together with the data model components' hierarchical relations make any integration queries possible against any datasets that share a common or even a related component. This data integration approach architecture is presented in Figure 2.

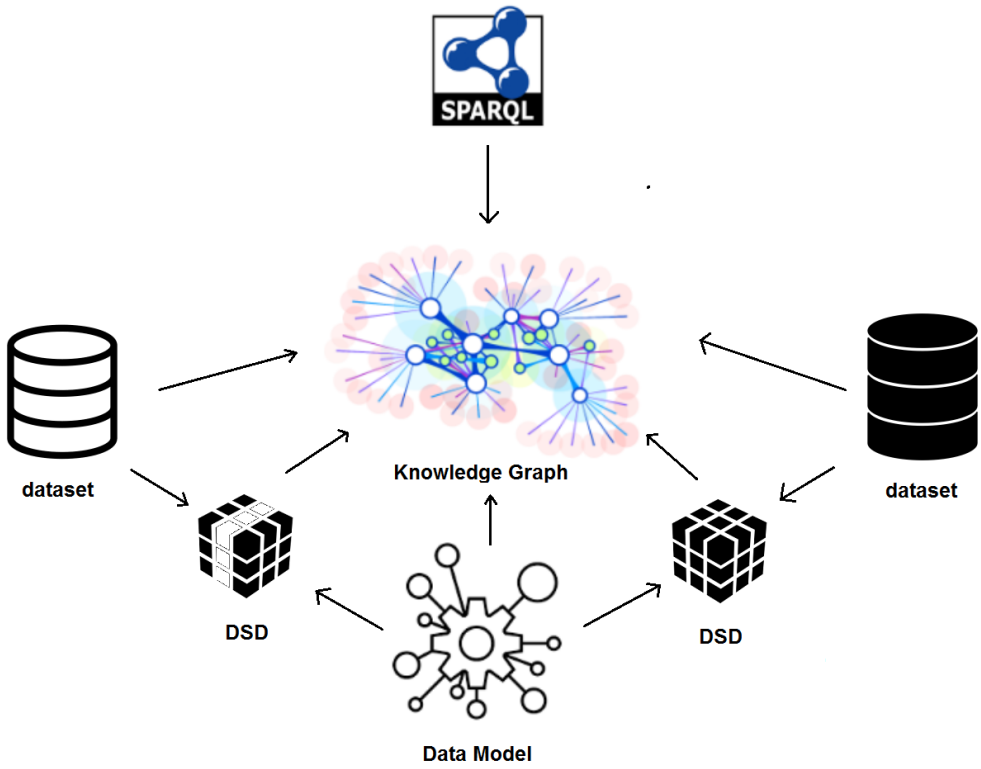

Figure 2. Data Integration with RDF cubes approach architecture.

3.2 The RDF Data Cube Vocabulary

Many datasets include statistical information that has to be published on the web in a machine-readable format, thus, the RDF Data Cube Vocabulary has been selected for their semantic representation. The RDF Data Cube Vocabulary is a W3C standard for representing multidimensional data using the Resource Description Framework (RDF). This vocabulary provides a structured way to describe datasets, observations and the relationships between them, making it easier to integrate and analyze data from diverse sources.

Each individual point within a dataset turns to a point within the data cube, that is an observation, which refers to the actual data, the measured values. An observation is a combination of dimensions and measures that "point" to that data. A dataset represents a collection of observations, as they are described within a data cube.

The most crucial component of the RDF Data Cube Vocabulary is the Data Structure Definition (DSD), which defines the schema of a dataset, namely the dimensions, measures and attributes that structure the data. Essentially, DSD provides a template structure for how the data within the dataset is organized,

ensuring semantic consistency and structural modularity, thus also enabling interoperability between different datasets.

A data cube may consist of some, or all of the following types of components, depending on the data modeling requirements:

Dimensions define the axes along which the data is organized. Each dimension is a variable like time, location, or a category. Dimensions are described using RDF properties and often linked to standardized codelists, as they have to have a specified range of values (i.e. there are various codelists for countries, regions and municipalities).

A measure represents the quantitative variable being measured, such as population, temperature, or air pollution. A data cube may contain multiple measures. Each measure is a property that indicates what is being observed in the dataset, i.e. air pollution, or noise pollution.

Attributes are additional properties that provide context to the observations and can include metadata like the unit of measure, an optional property, or annotations (i.e. speed is measured in km/h).

The values for dimensions within a data cube must be unambiguously defined. Codelists are controlled vocabularies that specify the values a dimension may have, ensuring consistency and interoperability between different cubes with common dimensions. Similarly, codelists may be used to specify the range of values of an attribute component, as well.

Additionally, in the context of the RDF Data Cube Vocabulary, a slice represents a subset of observations from a dataset that share specific values for one or more dimensions. This concept allows for grouping observations in a meaningful way, making it easier to access, analyze and query specific parts of the data. Slices can help users focus on particular aspects of the data, such as all observations for a specific year or region (i.e. environmental measurements only for the year 2023 and for the Municipality of Thessaloniki, or players stats only for the season 2024-25 for the Euroleague). Thus, each slice is defined by fixing one or more dimensions to specific values, while the other dimensions can vary within the slice.

Defining slices within data cubes is a good practice, as by structuring data into slices, the execution of targeted queries can be performed without processing the entire dataset, enabling enhanced querying and analysis, while observations that share the common values in the respective dimensions might only include values of the other dimensions in their description. For example, observations in a slice with the value ‘2023’ for the year dimension are not required to explicitly include the year property in their RDF description, as this is already implied by the slice.

Finally, defining concepts for RDF Data Cube components is essential for providing clear and unambiguous meanings to the dimensions, measures and attributes used in the data structure definition. Concepts can be defined using RDF vocabularies like SKOS (Simple Knowledge Organization System) or OWL properties within the concept definitions to ensure that the terms used in the data cube are well-defined, standardized and interoperable, as well as to link to external ontologies.

3.3 Data Modeling

The data modeling principles of the integration method proposed in this work are common between use cases. However, adjustments may be needed for different cases, depending on their special characteristics and structures, without altering the sense of the method. To this end, two different use cases are prsented to showcase the application of the approach in various fields.

For both use cases, environmental and basketball (and for any other future use case as well) a number of components have been defined to represent the related concepts and entities within a shared data model. Having been structured by the same data model, the datasets DSDs can be considered semi-connected, because they may include common or related (through hierarchical relations) components of the data model.

A common modeling practice in Data Cube is to define abstract components in a data model, if there is not a definite codelist that covers all possible values for that component (dimension). Then, a more specific component with the respective codelist can be defined (even in a Data Structure Definition that has been created for a dataset), as a subproperty of the abstract component of the data model, if this subproperty is still not abstract. Thus, dimensions, measures and attributes across datasets are semantically linked using `rdfs:subPropertyOf` relations to broader components defined in the common data model. The datasets properties can align with data model narrower components that are defined as subproperties of common broader components.

For example, in the environmental data model of the UPCAST Public Administration pilot, there is a dimension about the environmental stations that measure the air pollution, namely `upcast-env-dimension:airStation`. As this dimension may have different values if it applies in a Greek dataset, or in another country's dataset, this dimension is considered as an abstract dimension with no specific codelist related.

For each dataset, a subproperty of this dimension can be defined with a suitable codelist. For example, in a Greek dataset the list of air stations is fixed. Thus, a `gr-dimension:airStation` can be defined as a subproperty of `upcast-env-dimension:airStation`. The former has a range of values included in the `gr-codelist:air-station` codelist, that contains the Greek stations. Respectively, an `it-dimension:airStation` dimension with an `it-codelist:air-station codelist` can be defined for a corresponding case of an Italian dataset. Such hierarchical structure is presented in Figure 3.

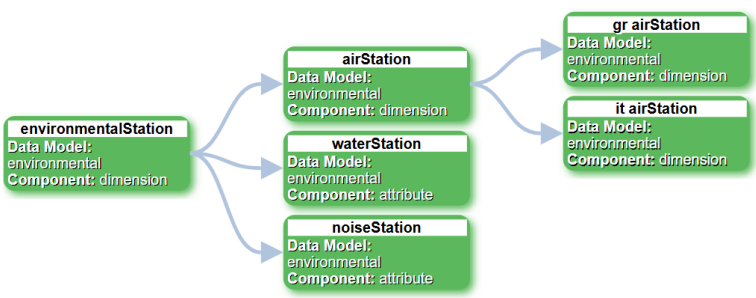

Figure 3. Hierarchical structure between broader and narrower (sub-properties) components.

In the case that the range of values of a dimension is definite, there is no need to define related subproperties. For example, the list of air pollutants is more or less concrete, thus the `upcast-env-dimension:airPollutant` dimension can already be related to the

`upcast-codelist:air-pollutant` codelist which includes the corresponding air pollutant types as concepts (i.e. CO2, PM10, O3, C6H6 etc.). This dimension can be used as-is for the respective datasets.

This practice can be followed thoroughly when developing a common data model that could potentially be extended to cover an increasing number of heterogeneous datasets. This is the case with the environmental data model of the UPCAST Public Administration Pilot, since its components cannot a priori cover all categories and properties of the public administration and environmental field. Therefore, a modular data model is required, allowing components to be reused or extended as needed.

This also supports the reuse of components across data models from different domains. Generic dimensions are defined (and can be enriched further) to capture common concepts such as temporal (Year, Date, DayInterval, Timestamp, Hour) and spatial (Country, Region, Municipality) concepts. Additional cross-domain components can also be introduced. An example is the generic Sensor component in the UPCAST use case that serves as a parent property for both environmentalSensor and trafficSensor components, which belong to different data models (shown in Figure 4).

A tree visualization of data model components (Sections 4.4 and Appendix 4.4) can be leveraged to illustrate these relationships, highlighting domain-specific components in different colors alongside shared generic ones. In this way, modeling in one domain can reuse or extend existing components of other domains, ensuring modularity and adaptability at the data model level.

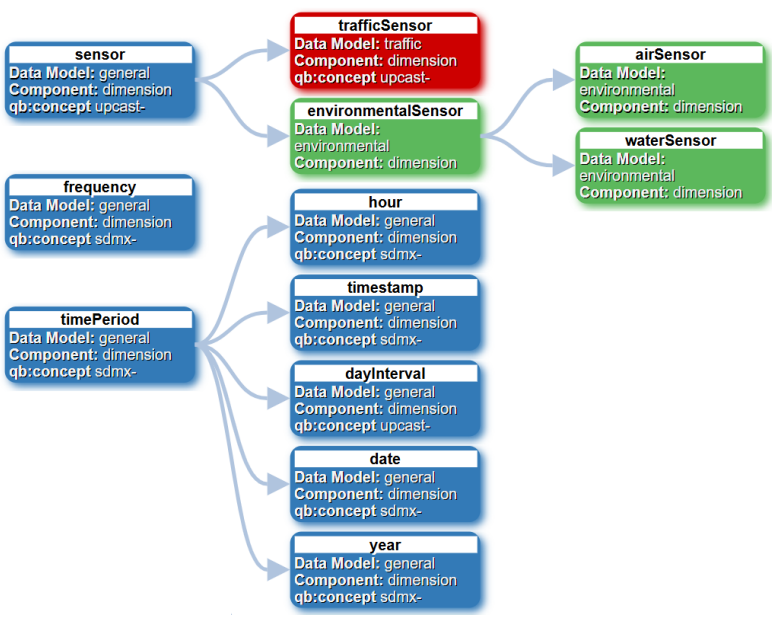

Figure 4. Illustration of relations between cross-domain components.

Abstract definitions ensure that the components conform to a standard modular structure, which promotes consistency and interoperability across different datasets and applications, when combined with more specific definitions through hierarchical relations. This facilitates the description of datasets

from various sources in a common way, in terms also of analysis and integration, as they might share similar and related dimensions and measures and enables reasoning engines to align and integrate data at query time.

Apparently, the same also stands for data models with no great necessity to be extended with narrower components, as the entities and concepts they represent already cover most of the corresponding datasets values. This is the case for the basketball use case, where most datasets categories from the various leagues have a similar structure. To this end, there is no need to extend the data model with numerous narrower components; instead, a standard set of components can be used for most cases. Consistency, interoperability and query time data integration remain in the spotlight (with more efficient and streamlined queries, exploiting one triple pattern across all cubes), by leveraging mostly the same (but still in some cases the related) components.

Thus, for the basketball data modeling [42] there is no use of subproperties for core identity-defining dimensions such as player, team, or court. These properties are used directly and uniformly across all datasets and competitions (e.g., NBA, Euroleague), with variation handled through season- or league-specific codelists rather than through property specialization.

This design choice reflects the fact that the semantics of a player and a team dimensions are stable and globally consistent in this domain, even as the value sets differ. Instead of introducing subproperties like `nba-dimension:team` or `el-dimension:team`, the model attaches different codelists at the level of the Data Structure Definition (DSD), allowing reuse of the same property across contexts without fragmenting the ontology, providing a clean and efficient modeling solution.

Another difference is the granularity level of the components in each data model. Basketball statistics usually have standard global vocabularies, while for environmental data, different codelists may have different temporal and spatial resolution (e.g. environmental sensors could be defined in country or municipality, or municipality block level, etc.), so the modeling and publishing context may vary. Thus, hierarchical relations via `rdfs:subPropertyOf` becomes essential, as it enables robust semantic querying across datasets, serving also as a fault-tolerant mechanism that compensates for missing or incomplete DSD metadata.

This is a reasonable possibility for the decentralized nature of the environmental use case, as there is no total control over the construction of a DSD by multiple publishers. If a DSD is incomplete or misses a critical component (e.g., the country for an air station), it may damage the knowledge graph. The result is a generic `airStation` dimension with values from multiple codelists but no context. To this end, the property-level hierarchy enables safer query time integration, but also supports localization.

The nature of the basketball statistical datasets can be characterized as more centralized, in the sense that there are no multiple providers that can transform and publish semantic data at any time. In fact, in this use case, there is only one central entity that manages the data. However, there are still components that form an internal hierarchy, while additional classes and properties need to be defined to represent all the information.

An example is the `bco-dimension:homeTeam` and `bco-dimension:awayTeam` of the games' cube, each one referring to the home and the away team that played in that game. Both dimensions are `rdfs:subPropertyOf` the `bco-dimension:team` which refers to a team entity. For each league and each season, a different codelist of the participating teams has to be defined as the range of possible values for all these dimensions, for that data, as explained previously.

For example, for the Euroleague 2024-25 season, a `bco-codelist:teams-euroleague-2025` conceptScheme is defined with the 18 participating teams, a `skos:concept` for each one with a `skos:prefLabel` and a `skos:altLabel` for the specific season's official (sponsor) name. As there are common teams between seasons and a team entity also may contain additional properties, a `bco:Team` class is defined to represent the related info (such as country, dbpedia link etc.). The connection between them is made by a foaf:focus property, used to indicate specific individual things that are mentioned in different SKOS schemes, within each codelist concept, pointing to the `bco:Team` class of the corresponding team. In that way, every codelist concept (league-season team) is linked to the main team entity.

The same logic is also followed for the players, where the core semantic `bco:Player` class, represents a real-world basketball player as a global identity and a different codelist of the participating players for each league and each season is defined as the range for the possible values for the `bco-dimension:playerConcept` dimension. However, an extra class is introduced, the `bco:PlayerParticipation` class, which refers to a player's participation in a specific team for any season and league. It links to the core `bco:Player` class via the property `bco:player` and includes season-bound metadata, as shown in Figure 5. In the previous example, Sasha Vezenkov has two `bco:PlayerParticipation` instances for the 2024-25 season, one for the Euroleague and one for the Greek BasketBall League, both with the same team, Olympiacos.

The `bco:PlayerParticipation` class is introduced to capture context of a player's unique combination of team, league and season, as changes such as mid-season transfers (mainly), different listed names, jersey numbers, or status (e.g., injured, active), which cannot be represented cleanly with the core `bco:Player class` or the respective codelist.

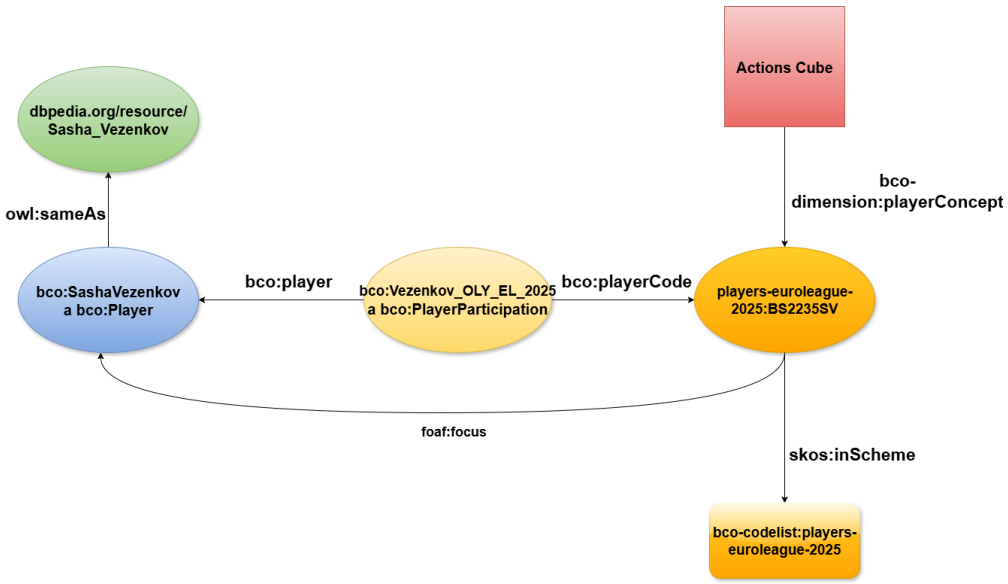

Figure 5. Player-related classes and concepts and their interconnection.

These extra classes and properties operate as auxiliaries to the data integration approach.

It is evident that providing the data cubes with the appropriate codelists and defining the corresponding components, when necessary, is a very critical part when using the data models, with respect to the modularity and the hierarchical structure of them.

RDF descriptiveness also allows for the integration of multilingual content, with the use of the RDF lang tag that denotes the language of the respective entity. This is the case for both dataset data (content) and metadata (description); both are foreseen to have multilingual content. SPARQL queries with respective lang filters can then handle and integrate multilinguality of datasets

Further data modeling challenges regarding data cubes exceed the scope of the current paper. However, presenting two different use cases showcases that multiple modeling options can work well for the data integration approach with data cubes that is presented in this work.

4 Semantic Integration with CubeModeler

CubeModeler is a suite of tools and services facilitating the processes related to the creation and management of the data cubes components and the various processing steps. It is designed to assist in defining, describing and transforming statistical datasets into interoperable Data Cubes, bringing consistency and connectivity to the data lifecycle and data integration needs. It can be reused in any statistical context, including operations that can be performed for generic semantic tasks, like building structured data models and RDFizing datasets tailored to the RDF Data Cube Vocabulary. The most characteristic functions of CubeModeler regard the development of data structures (DSDs), the semantic description of datasets, the semantic transformation of datasets and the data model enrichment with new components and codelists.

CubeModeler is currently in use within the UPCAST project's internal infrastructure. While it is not yet publicly available, a demo version is planned to be released in GitHub in Q1 2026.

The following subsections outline the practical steps of the proposed methodology as implemented in CubeModeler. The workflow begins with the creation of a Data Structure Definition (DSD) and the optional definition of new components and associated codelists when required. The dataset is then described through metadata, semantically transformed into RDF based on the selected DSD and finally accessed through SPARQL queries. These steps correspond directly to the modeling-based integration process presented in Section 3 and demonstrate how CubeModeler operationalizes it in practice. For a detailed walkthrough of the CubeModeler UI, workflows and configuration scenarios, see Appendix A.

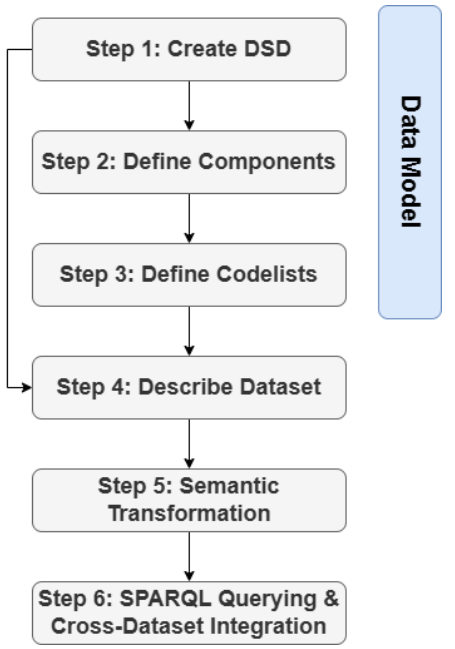

Figure 6. Data Integration Steps

4.1 Creating the Data Structure Definition

Building Data Structure Definitions (DSD) for RDF Data Cubes is a crucial step towards the semantic transformation of the dataset, as this DSD will be the map in which the dataset's columns will be aligned with the data model properties. The DSD maker service of CubeModeler allows the user to select the data model, the components of which will be used to build the DSD, while a filtering option of component type (dimension, measure, attribute) is also offered for more convenience.

Users can configure the DSD metadata, including prefix, IRI, label, the attachment level of components (dataset, slice, observation) and additional slice metadata (slice key, prefix, IRI, label), if any, as shown in Figure 7. The system also allows defining the temporal and spatial scope of the DSD for internal indexing and reuse. For datasets that share structural similarity, CubeModeler supports duplicating existing DSDs with adjustments, facilitating rapid iteration. After definition, DSDs are previewed and exported in TTL format, ready for reuse and upload to the knowledge graph.

For a detailed walkthrough of the Data Stuctures Interface, see Appendix A.2.

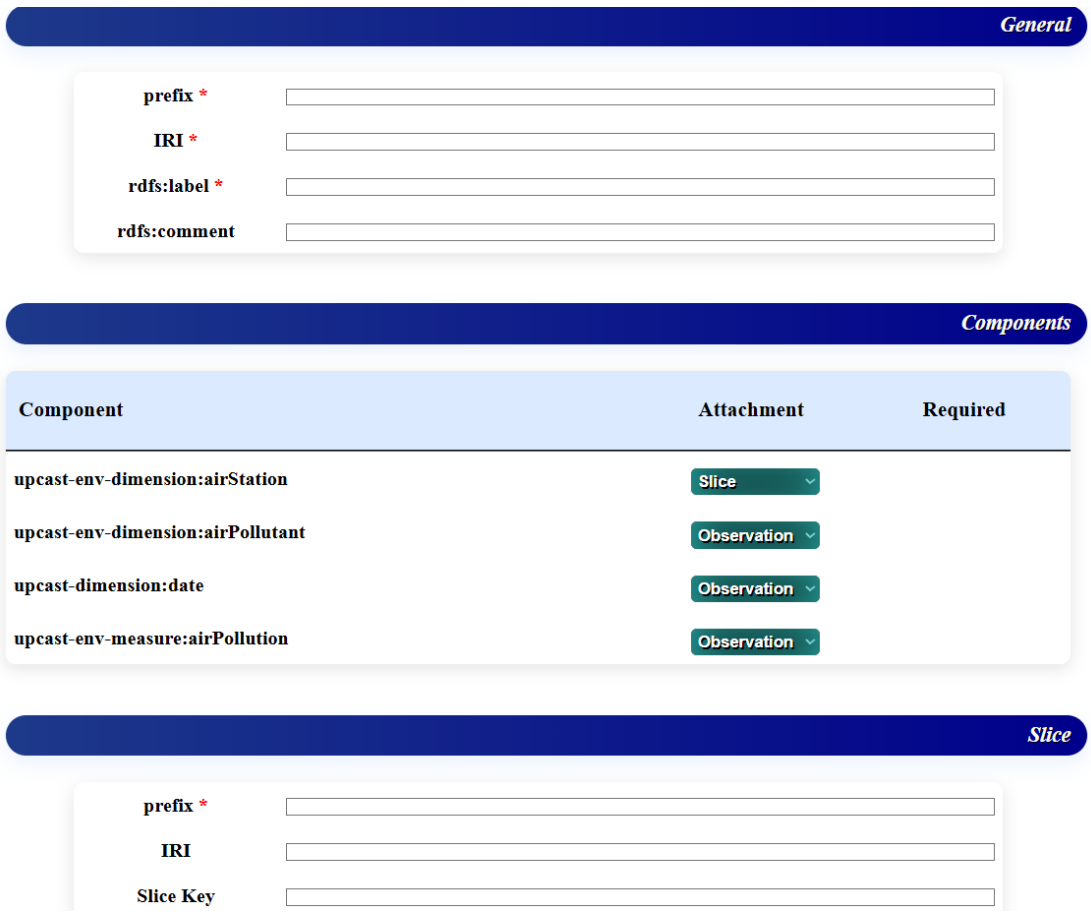

Figure 7. DSD configuration interface towards the final structure of the DSD.

4.2 New Components Definition & Data Model Editing

Ideally, all required components when creating a DSD would be available in the data model. However, for cases when a required component of the dataset has not been defined yet to include it in the DSD, that component can be created within CubeModeler.

Adding new components in the data model through CubeModeler is carried out through a tree visualization of the data model and all its components (dimensions, measures, attributes). The whole hierarchical structure of the data model is presented in the visualization (as shown in Figure 8), using arrows to represent `rdfs:subPropertyOf` relationships. It supports filtering by type and data model, helping users understand component roles and select appropriate ones for DSD definition.

Any component from the tree visualization can be selected along with a relation type (broader/narrower/related) as a starting point for adding a new component in the data model. If no

relations can be identified with the data model's current components, a new component can be added from scratch, with no explicit relations to the existing ones.

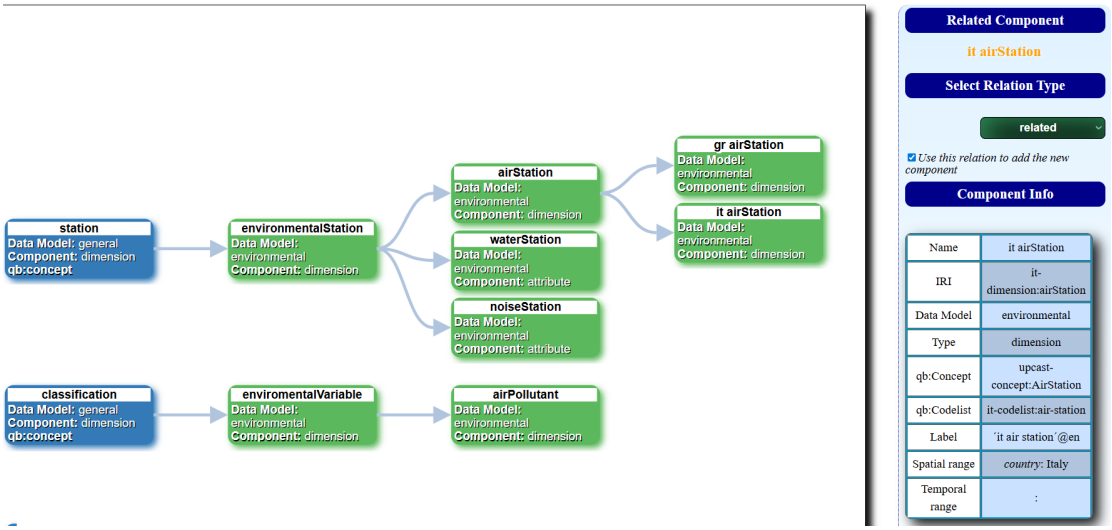

Figure 8. A tree visualization of the data model is used for adding new components.

The new component is defined by inserting the required input for specific properties, such as the IRI, the label, the component type, the range, the concept, any related codelist, but also the `rdfs:subPropertyOf`, if any. Finally, the new component is added to the CubeModeler's database, the source for the data modeling feature, as well as to the updated data model, after validation.

Moreover, the new component has been placed appropriately in the components' hierarchy and the data model structure leveraging the relation property selected at the beginning by the user. For direct `rdfs:subPropertyOf` relations, this task is straightforward. Selecting a "related" relation places the new component at the same level as its reference, sharing the same parent if applicable. If a broader (parent) component is specified but not yet in the model, CubeModeler prompts the user to import it.

Using this service, the data model(s) that CubeModeler handles can be enriched at any time and this also could facilitate the building of the data model (and its import to the database) from scratch. Now, the new component(s) needs to be validated and then it can be included in the DSD, as in step 1.

4.2.1 Components Validation

After a new component is inserted into the data model, it must be validated by a reference admin user. This is to prevent any wrong input that could alter the data model and affect the whole semantic transformation and data integration process. Components can be activated or deactivated at any time (Appendix A.4.1), ensuring that the data model consistently reflects its most up-to-date form .

4.2.2 Relations Editing

Another intervention in the data model is featured through the relations editor functionality (Appendix A.4.2), where a component can be selected as an `rdfs:subPropertyOf` value for any data model component that has not yet defined this property. This allows for supplementary configuration of the modular and hierarchical structure of the data models that are built through the data model tree visualization, with simple and easy to make updates that may be needed when enriching the data model.

4.3 Codelist Definition

It is probable that a new codelist has to be defined and included along with the definition of a new component (in case there is a definite range of values for this component). To this end, CubeModeler offers two possibilities to the user, the codelist import and the codelist creation.

The codelist import implies a ready-to-use codelist from the user and is carried out through a metadata form where the user defines the codelist's information. These include an indicative name for storing in the CubeModeler database, but also basic metadata such as the codelist's IRI and prefix, the label and the Concept Scheme. Optionally, the user may point out the data model's component that uses this codelist (related component) and then this codelist could be directly used as this component's range of values in the upcoming semantic transformations. After providing these metadata, the codelist in TTL format must be uploaded and imported into the knowledge graph. Only after validating the syntax of the codelist, with the TTL Validator, is it uploaded to the knowledge graph and also recorded in the CubeModeler's database.

The codelist creation, on the other side, allows for the definition of a codelist based on a dataset uploaded by the user. There, CubeModeler identifies the distinct values of the selected column that represent the codelist and provides the necessary SKOS properties (inScheme, notation, prefLabel, broader, if available) to form the new codelist. This can now be uploaded in the Knowledge Graph through the codelist import functionality and be used in the definition of the new aforementioned component.

To summarize, if all necessary components have already been defined in the data model, user can only create the corresponding DSD. If not, CubeModeler provides the path to define the respective component along with its codelist. Then, the semantic transformation process can take place.

4.4 Semantic Transformation

The RDFizing process is based on the built DSD for the dataset and an interface where the user can configure all the transformation settings. One part of these settings concerns aligning the dataset's columns to the DSD components, thus the respective RDF property with the column value is created. Another part concerns mapping the coded components (components that use a codelist) columns values to a specified property of the respective codelist, i.e. the dataset value could match either the `skos:notation` or the `skos:prefLabel` property of the concept, while they can also make a direct match, using the codelist prefix and the dataset value.

The transformation settings for the dataset can be configured by the user, using the semantic transformation interface of CubeModeler, presented in Figure 9. For each component (dimension, measure, attribute) of the selected DSD, the corresponding dataset column which refers to that component, has to be aligned. For example, a points measure of a basketball player (in a players statlines

cube) should be aligned with the points column of a boxscores dataset. To this end, one of the dataset's columns has to be selected for each of the DSD components; this is the basic transformation scenario.

Additional configuration options define how each component attaches to the dataset: at the dataset, slice, or observation level. Dataset-level components use fixed values, while slice-level components can be fixed or dynamically mapped, depending on modeling needs. Observation-level components must always be matched to dataset columns.

CubeModeler supports multiple DSD-to-dataset alignment strategies based on the dataset's structure and user preference. For slice-level components, it allows either one slice per row or grouping rows with identical values into shared slices, the latter being the more typical approach. Moreover, when alignment isn't obvious, CubeModeler offers a "smart slice mapping" feature that automatically detects shared values across slice-level components to generate grouped slices. The algorithm ensures each slice includes all observations with matching values in the relevant columns.

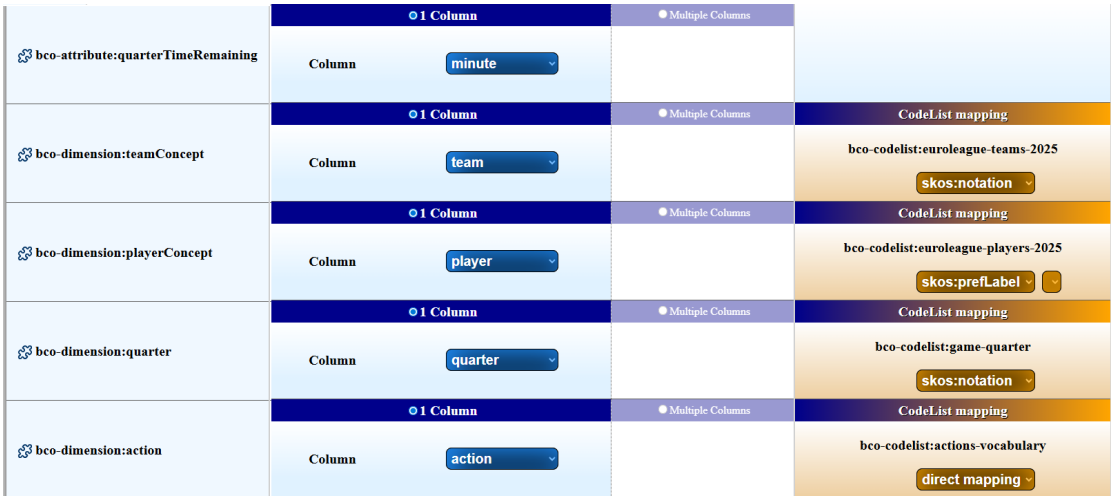

Figure 9. CubeModeler's user interface for the configuration of the RDFizing process.

As there would be several coded components within the DSD - dimensions and attributes that use a codelist - their aligned columns values should be exploited to make the final match with a specified property of the respective codelist, as this has been specified in their definition. Then, the dataset values of the column that has been aligned with the component, are matched to the respective codelist property values. The property may also be configured by the user, as `skos:notation`, or `skos:prefLabel`. In the latter case, the user can select any of the existing languages of the `skos:prefLabel` of a codelist entity to make the mappings for multilingual entities and datasets. During RDFizing, CubeModeler queries the knowledge graph to match dataset values with SKOS concepts from imported codelists, returning the appropriate concept IRI. Alternatively, it supports direct value-to-IRI mapping by combining the codelist prefix with dataset values.

All the settings that are configured in the RDFizing process can be stored in the CubeModeler database as a type of workflow, in order to be able to insert and use them directly for similar datasets that relate

to the specific DSD. Thus, if a semantic transformation task for the same DSD has to take place later with another dataset, it can leverage the respective configuration workflow and automatically all settings will be imported and adjusted directly to the mapping interface. In this way the user can simply click the execute button to proceed to the RDF transformation and have the final RDF file in a blink of an eye. If needed, any customization of the configuration settings can be applied in the interface before execution and also this new set of settings can be saved as a new workflow as well.

Finally, the dataset is totally transformed to RDF in TTL format, based on the user configuration settings and is checked through a TTL validator so it can be uploaded to the knowledge graph. Only if the syntax of the produced TTL is correct, the upload can be performed, otherwise, there is an alternative path.

An RDF editor has been developed for fixing any syntax errors caused by malformed source datasets, or any unexpected factors, after the RDFizing process. Thus, if not directly uploadable to the KG, there is the capability of fixing any issue, by exploiting the error/warning messages of the TTL validator within the editor and when it is all correct, the final TTL can be uploaded to the knowledge graph.

Regarding the knowledge graph, it is created and enriched within an RDF triple store and specifically a Virtuoso server, where all CubeModeler products are uploaded; the DSDs, the datasets descriptions, the datasets in RDF/TTL format, as well as the codelists.

Uploading to Virtuoso is not always straightforward, because of size limitations on each INSERT query of the server. Thus, various scenarios have been taken into account, to split the TTL dataset into smaller parts and finally upload them successively in the triple store. The algorithm analyzes the dataset structure and identifies key separators in the TTL output to group data into size-based chunks. Smaller chunks are uploaded in bulk, while larger ones may be further partitioned using additional separators for efficient ingestion. Finally, all the triples are uploaded in the knowledge graph stored in Virtuoso.

Having uploaded the dataset TTL in the knowledge graph, its triples may be returned as results by the relevant SPARQL queries.

For a more detailed walkthrough of the RDFizing interface, see Appendix A.2.

4.5 Dataset Description

Describing datasets regards the dataset metadata. It is another straightforward process, as the user has to insert the dataset metadata directly to a specific form. The latter includes fields from established vocabularies, such as DCAT, PROV, TERMS and others, as fields, depending on the level of metadata detail provided. The required fields concern the basic metadata of the dataset, such as the IRI, the relevant DSD (through the `qb:structure` property) and a label. Finally, an RDF dataset description in TTL format is created and can be uploaded to the knowledge graph as well.

Within the Dataset Description section, critical properties related to the version and the licence of the dataset can be defined as shown in Figure 10. Versioning is declared through the DCAT dcat:distribution property, defined for any dataset description, while the dcmi:licence property is used to include the licence related to that dataset, if any. These properties are then included in the RDF representation of the dataset metadata description, enabling the version and licensing control. There is also the possibility of inserting various lang entities through the respective input fields to support multiple languages in the dataset metadata. Finally, the IRI of the dataset could be used directly to the RDFizer interface to automatically connect the rdfized dataset with the respective dataset description.

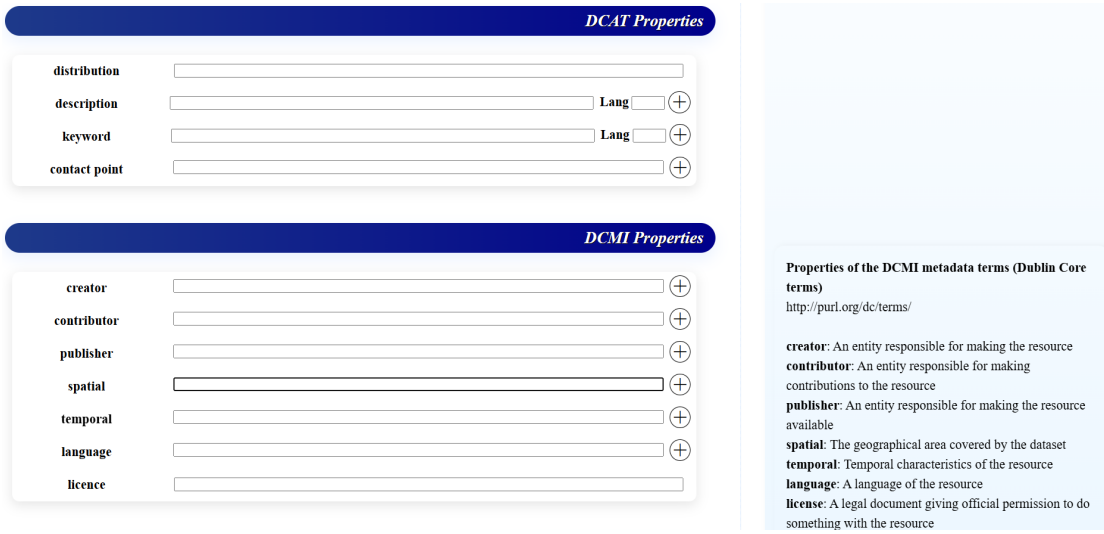

Figure 10. Data Description page where metadata properties are defined.

Having all definitions (DSD, metadata, RDF data) into one place, that is the Knowledge Graph, users can now perform SPARQL queries to extract the corresponding knowledge, together with knowledge that connects with these data through common or related properties. Next sections provide detailed examples to demonstrate the integration of various datasets through SPARQL queries.

5 Use Cases

In this section we showcase various data integration examples regarding two major use cases of statistical data, basketball and environmental data. Having been modeled and semantically transformed by CubeModeler, according to the previous sections, a series of SPARQL queries can be executed to the respective knowledge graphs, to get the relevant information in RDF triples.

5.1 Basketball Data

Basketball data were derived from league-provided records (NBA, Euroleague), which represent standardized and widely used statistical feeds. They have been modeled with RDF Data Cube Vocabulary and some additional classes (e.g. players, teams), as mentioned before. There are three main types of cubes, the games, the actions and the statistics per player and per team, which may share some common components, based on the datasets schemas. The games cube contains information about a game played for a competition/league, such as the opposing teams, the final score, temporal info (date and time) and others. The statistics cube includes the boxscores stats of players and teams in each game. Finally, the actions cube comes from the play-by-play of the game, that is the series of actions a player has taken during the game, with valuable gametime-related data. Datasets are usually per league (e.g. Euroleague, Greek Basketball League) and per season, while they can also be split by round or phase, for example, NBA Finals 2025, Euroleague Regular Season Round 1, etc. Integrated results are provided in query time

combining the common components of different cubes (per type, league, or season). The game entity commonly serves as the shared link across cubes, but integration is also further supported by additional components that are either directly shared or semantically related. Prefixes and namespaces declarations are omitted from the queries examples for brevity; they are presented in total in the Appendix B.

5.1.1 Player Points in Each Game (for All Leagues in One Season)

A first example combining the various cubes of basketball data includes a query that sorts the games played by a player during a season (for any team) by the points he scored in descending order.

```
SELECT ?player ?points ?league ?round ?opponentName
FROM <http://data.basketontology.org/cube/data/graph/>
WHERE {
    ?player a bco:Player .
    ?player rdfs:label ?playerLabel .
    ?playerParticipation a bco:PlayerParticipation .
    ?playerParticipation bco:player ?player .
    ?playerParticipation bco:playerCode ?playerConcept  .
    ?playerParticipation bco:season bco_season:2024_25 .
    ?playerParticipation bco:league ?league .
    ?statline bco-dimension:playerConcept ?playerConcept.
    ?statline a qb:Observation .
    ?statline bco-dimension:teamConcept ?teamConcept .
    ?statline bco-measure:points ?points .
    ?slice qb:observation ?statline .
    ?slice bco-dimension:game ?game .
    ?game bco-dimension:phaseround ?round .
    ?game bco-dimension:homeTeamConcept ?home .
    ?game bco-dimension:awayTeamConcept ?away .

BIND(
    IF(?teamConcept = ?home, ?away, ?home) AS ?opponent
    )
    ?opponent skos:prefLabel ?opponentName .

FILTER(?playerLabel="Sasha Vezenkov"@en)
}
ORDER BY DESC(?points)
```

Listing 1: SPARQL query that sorts all player's season games by points

This query combines two types of cubes (games, player statistics) for different leagues for a specific player. It could be extended also to multiple seasons; there, the 2024-25 season is used for Sasha Vezenkov as the only filtering options. Thus, this query could be generated (in a form) by a simple name input of the player, to find all his scoring performances in all the games he played during that specific season. The statlines cube has been sliced per game, so each game slice has only the respective players statlines observations. The integration of the games for all leagues he played (Euroleague, Greek Basketball League) is done implicitly, showcasing a simple but genuine example of semantic

data integration, as they have been modeled similarly and each `bco-dimension:game` value of the statlines cubes refers to an observation of the games cube.

The player name could be also used within the `PlayerParticipation` or the `PlayerConcept` label properties where it is also recorded. However, the core `bco:Player` class however was preferred as an example, to demonstrate the connection between all player related classes and concepts. This class is also connected to dbpedia entities through `owl:sameAs` properties, making it a more complete case.

5.1.2 Player Clutch Shooting

A use case that identifies the clutch element of a basketball player is presented through the next SPARQL query that calculates the field goal percentage (FG) of a player (Sasha Vezenkov) during the final two minutes of the 4th quarter in playoffs games, for each season. It integrates information from multiple cubes, such as the actions cube (shot events) and the games cube (game metadata) for all relevant seasons/cubes.

```
SELECT ?season (COUNT(?madeShot) AS ?made) (COUNT(?allShots) AS ?attempts) (
    xsd:decimal(COUNT(?madeShot)) / COUNT(?allShots) AS ?fgPercentage)
FROM <http://data.basketontology.org/cube/data/graph/>
WHERE {
    ?player a bco:Player .
    ?player rdfs:label ?playerLabel .
    FILTER(?playerLabel="Sasha Vezenkov"@en)

    ?participation a bco:PlayerParticipation .
    ?participation bco:player ?player .
    ?participation bco:playerCode ?playerConcept .
    ?participation bco:season ?season .
    ?participation bco:league bco_league:Euroleague .

    ?obs a qb:Observation ;
    ?obs bco-dimension:playerConcept ?playerConcept .
    ?obs bco-dimension:action ?action .
    ?obs bco-dimension:quarter ?quarter .
    ?obs bco-attribute:quarterTimeRemaining ?timeLeft .
    ?obs bco-dimension:game ?game .

    FILTER(?action IN (
    bco_action:two-pointer-made, bco_action:two-pointer-missed,
    bco_action:three-pointer-made, bco_action:three-pointer-missed
    ))
    FILTER(xsd:integer(?timeLeft) <= 120)
    FILTER(?quarter IN (game-quarter:Q4))

    ?game bco-dimension:phase bco-el-phr:Playoffs .

    BIND(?obs AS ?allShots)
    FILTER(?action IN (bco_action:two-pointer-made, bco_action:three-pointer-
        made))
```

```
    BIND(?obs AS ?madeShot)
}
GROUP BY ?season
ORDER BY ?season
```

Listing 2: SPARQL query that combines various cubes and stats to identify the clutch element of a player

As in the previous example, the core `bco:Player` class is used to filter by the player name. Through the bco:player property of the `bco:PlayerParticipation` class join, information about the season and the league (in that case, only Euroleague) is derived. The query demonstrates semantic data integration across multiple axes, as the `?playerConcept` is used to identify the related actions cube observations. Only the actions related to field goal attempts (two pointer made, two pointer missed, three pointer made, three pointer missed) are searched, while the clutch time scenario restricts the game time to the last 2 minutes of the game (so, the `game-quarter:Q4` has to be filtered). Finally, through the games cube (the actions-games cross-cube linkage takes place via the `?game` variable), only the games that belong to the `bco-el-phr:Playoffs` phase are taken into account.

The result aggregates the total number of made and attempted shots per season for that context and computes the shooting efficiency as a decimal percentage. A crucial part is the definition of appropriate codelists for the respective components, enabling interoperability between the various cubes and allowing the direct multidimensional condition filtering. By grouping results by season, the query integrates cubes in temporal resolution, while filtering by specific actions and game context showcases multidimensional RDF modeling in action.

5.1.3 Individual Performance Comparison Across Different Leagues

Another data integration example is finding the top 4th quarter scoring performances by players either in Euroleague or the NBA, during the 2024-25 season. The next SPARQL query performs a multidimensional, cross-cube integration over three semantically aligned types of cubes, the actions, the players statlines and the games cubes. The scoring performance is measured by the total points resulting from the made field goals and free throws.

```
SELECT ?playerName ?league ?pointsScored ?minutesPlayed ?gameDate ?
    opponentName
FROM <http://data.basketontology.org/cube/data/graph/>
WHERE {
    ?participation a bco:PlayerParticipation .
    ?participation bco:season bco_season:2024_25 .
    ?participation bco:league ?league .
    ?participation bco:player ?player .
    ?participation bco:playerCode ?playerConcept .

    FILTER(?league IN (bco_league:Euroleague, bco_league:NBA))

    ?player rdfs:label ?playerName .
    ?actionObs a qb:Observation .
    ?actionObs bco-dimension:playerConcept ?playerConcept .
    ?actionObs bco-dimension:action ?action .
```

```
    ?actionObs bco-dimension:quarter game-quarter:Q4 .
    ?actionObs bco-dimension:game ?game .

    FILTER(?action IN (
        bco_action:two-pointer-made,
        bco_action:three-pointer-made,
        bco_action:free-throw-made
    ))

    ?slice a qb:Slice .
    ?slice bco-dimension:game ?game .
    ?slice qb:observation ?statlineObs .
    ?statlineObs a qb:Observation .
    ?statlineObs bco-dimension:playerConcept ?playerConcept .
    ?statlineObs bco-measure:minutes ?minutesPlayed .
    ?statlineObs bco-dimension:teamConcept ?teamConcept .

    ?game bco-dimension:date ?gameDate .
    ?game bco-dimension:homeTeamConcept ?home .
    ?game bco-dimension:awayTeamConcept ?away .

    BIND(IF(?teamConcept = ?home, ?away, ?home) AS ?opponent)
    ?opponent skos:prefLabel ?opponentName .

    BIND(
        IF(?action = bco_action:two-pointer-made, 2,
        IF(?action = bco_action:three-pointer-made, 3,
        IF(?action = bco_action:free-throw-made, 1, 0))) AS ?pointValue
    )
}
GROUP BY ?playerConcept ?playerName ?league ?minutesPlayed ?gameDate ?
    opponentName
ORDER BY DESC(SUM(?pointValue))
LIMIT 10
```

Listing 3: SPARQL query that performs a multidimensional, cross-cube integration over three semantically aligned types of cubes

Player identity is resolved via `bco:PlayerParticipation` instances, from which a season-specific `bco:playerCode` is extracted. This code acts as the unifying key across cubes, linking in-game actions and player statlines. The name of each player is extracted from the core `bco:Player` class.

Within the actions cube, observations are filtered to include only successful scoring actions (two pointer made, three pointer made, free throw made) that have been made in the fourth quarter. These are linked to corresponding game entities as the `bco-dimension:game` values of the actions observations refer to game cubes observations. from these entities the query extracts the game date and the participating teams. This information is used to compare the player's team to the home and away teams, which then lead to derive the opponent based on the player's team affiliation (retrieved from the statline).

Finally, the query accesses the statlines cube that includes a `qb:Slice` structure where the `bco-dimension:game` is attached, retrieving each player's total minutes played in the same game. These two are joined by matching both the `bco:playerCode` and the `bco:game` values, which ensures that each action and the stat summary refer to the same individual and game instance. Grouping is done by player, league and game context and the results are ordered by total points scored in the fourth quarter, returning the top 10 performances across both leagues (filtered through the `bco:league` property of the `bco:PlayerParticipation` class).

The approach enables precise integration of time-sensitive actions, aggregate statistics and contextual metadata across different levels of granularity, leveraging RDF's graph-based linking and the data model's structural modularity with the Data Cube. The query result is a coherent, unified analytical view generated from semantically interoperable data sources.

5.2 Environmental Data

A significant use case that leverages the Cubemodeler suite to its full extent is the Public Administration Pilot of the UPCAST project. The pilot uses environmental data and datasets for Thessaloniki Metropolitan area* that come from various sources and public municipal and national authorities and have been grouped into four thematic categories, for each one a specific data model was developed. Thus, there are: a data model about environmental measurements, a data model about demographics, a traffic data model and a data model about urban statistics. These data models can be considered semi-connected, because they relate to a more generic data model, which includes common components that either are shared between the specific data models too, or for which particular components have been defined in the specific data models as subproperties, so the latter are connected through the same parents. Examples of this kind of generic components are the year and the municipality dimensions, as well as the sensor dimension, which has an environmental sensor subproperty in the environmental data model and a traffic sensor subproperty in the traffic data model. Therefore, the Public Administration data model is a combination of components of four different data models, along with components that can be shared among them, as an umbrella data model.

All this environmental-related data has been modeled with RDF Data Cube Vocabulary with numerous components that have been defined to cover the many aspects and categories of the abstract environmental field. There is no definitive number of the types of the structures (DSDs), as additional datasets may be imported to extend the data model and create new cubes within the UPCAST Public Administration Pilot scope. As there is an extensive hierarchical structure in the data model between the components, integrated results are provided in query time combining the common and related components of the different cubes coming from a wider range of resources, potentially leveraging the `rdfs:subPropertyOf` property for the hierarchical relations. Prefixes and namespaces declarations are omitted from the queries examples for brevity; they are presented in total in the Annex.

5.2.1 Analysing Variations in Air Pollution Levels Between Local and Regional Scales

A use case of comparing the air pollution measurements in a specific air station of the Municipality of Thessaloniki and the average air pollution measurements of the Municipality is presented below.

*https://tds.okfn.gr/

```
SELECT ?maxLabel ?date ?airPollution ?airPollutionAvg ?maxPollution
FROM <http://data.upcast.eu/cubemodeler/pilot/data>
WHERE {
?dataset1 qb:slice ?slice1 .
?dataset1 rdfs:label ?label1 .
    ?slice1 a qb:Slice .
    ?slice1 upcast-dimension:date ?date .
    ?slice1 gr-dimension:airStation ?airStation .
    ?slice1 qb:observation ?obs1 .
    ?obs1 a qb:Observation .
    ?obs1 upcast-env-measure:airPollution ?airPollution.

?airStation a gr-codelist:AirStation .
?airStation skos:prefLabel ?stationLabel .

?dataset2 qb:slice ?slice2 .
?dataset2 upcast-dimension:municipality ?municipality .
?dataset2 rdfs:label ?label2 .
    ?slice2 a qb:Slice .
    ?slice2 qb:observation ?obs2 .
    ?obs2 a qb:Observation .
    ?obs2 upcast-dimension:date ?date .
    ?obs2 upcast-env-measure:airPollutionAvg ?airPollutionAvg .

BIND(IF(?airPollution > ?airPollutionAvg, ?airPollution, ?airPollutionAvg) AS
     ?maxPollution)
BIND(IF(?airPollution > ?airPollutionAvg, ?label1, ?label2) AS ?maxLabel)
FILTER(?stationLabel = "Sindos"@en )
FILTER(?municipality = "http://data.europa.eu/nuts/code/EL522" )
}
```

Listing 4: SPARQL query that for spatial comparison in air pollution levels

The SPARQL query focuses on the `Sindos` airStation and the broader municipality of Thessaloniki, using filters to isolate observations from the respective datasets (Sindos airStation via its `skos:prefLabel` of the corresponding codelist and Thessaloniki municipality identified by its NUTS code). It could also specify a particular air pollutant type (i.e. NO2) or the specific hour of the measurements (for the Sindos air station) but these are omitted for the example (supposing we have only one specific hour of the day for each observation).

One dataset contains daily pollution measurements (`airPollution`) from the Sindos airStation, while the other includes daily average pollution values (`airPollutionAvg`) at the municipality level.

The query joins two datasets on a shared temporal dimension, that is the date. For each date where both types of data are available, the query compares the local and regional pollution values. Using conditional logic (IF within BIND statements), it identifies the higher pollution value and records the corresponding dataset label, either that of the Sindos airStation or the Thessaloniki municipality.

The output, illustrated partially in Figure 11, includes the date, both pollution readings, the maximum value between them and the label of the source with the higher measurement. This enables a structured

comparison between localized and aggregated environmental indicators, helping to assess the air pollution levels under local conditions compared to municipality averages and identify significant variations.

date	airPollution	airPollutionAvg	maxPollution
"21-01-2021"^^<http://www.w3.org/2001/XMLSchema#date>	32	90.06	90.06
"22-01-2021"^^<http://www.w3.org/2001/XMLSchema#date>	32	96.74	96.74
"23-01-2021"^^<http://www.w3.org/2001/XMLSchema#date>	20	29.56	29.56
"24-01-2021"^^<http://www.w3.org/2001/XMLSchema#date>	10	26.01	26.01
"25-01-2021"^^<http://www.w3.org/2001/XMLSchema#date>	12	34.63	34.63
"26-01-2021"^^<http://www.w3.org/2001/XMLSchema#date>	34	11.7	34
"27-01-2021"^^<http://www.w3.org/2001/XMLSchema#date>	7	19.94	19.94
"28-01-2021"^^<http://www.w3.org/2001/XMLSchema#date>	8	26.89	26.89
"29-01-2021"^^<http://www.w3.org/2001/XMLSchema#date>	4	78.02	78.02
"30-01-2021"^^<http://www.w3.org/2001/XMLSchema#date>	27	30.85	30.85
"31-01-2021"^^<http://www.w3.org/2001/XMLSchema#date>	15	43.85	43.85

Figure 11. SPARQL results of air pollution comparison between local and regional levels.

5.2.2 Exploring the Relationship Between Air Pollution Levels and Population Size

Another example is to correlate air pollution measurements with the municipality population and calculate how the air pollution of a Municipality is "distributed" in its population, that is, how much air pollution corresponds on average to each resident of the municipality. This SPARQL query integrates environmental and demographic datasets based on the respective data models' components.

```
SELECT ?municipality ?airPollution/?population as ?PollutionPerPerson
FROM <http://data.upcast.eu/cubemodeler/pilot/data>
WHERE {
?dataset1 rdfs:label ?label1 .
?dataset1 qb:slice ?slice1 .
?dataset1 upcast-dimension:year ?year .
    ?slice1 a qb:Slice .
    ?slice1 upcast-dimension:municipality ?municipality .
    ?slice1 qb:observation ?obs1 .
    ?obs1 a qb:Observation .
    ?obs1 upcast-dimension:date ?date .
```

```
    ?obs1 upcast-env-measure:airPollutionAvg ?airPollution .

?dataset2 rdfs:label ?label2 .
?dataset2 qb:slice ?slice2 .
    ?slice2 a qb:Slice .
    ?slice2 upcast-dimension:municipality ?municipality .
    ?slice2 qb:observation ?obs2 .
    ?obs2 a qb:Observation .
    ?obs2 upcast-dimension:year ?year .
    ?obs2 upcast-dem-measure:population ?population .
FILTER(?population > 0)
FILTER(?date = "2025-06-01"^^xsd:date)
}
ORDER BY DESC(?PollutionPerPerson)
```

Listing 5: SPARQL query that integrates environmental and demographic data

The query retrieves observations of air pollution and population (`upcast-env-measure:airPollutionAvg` and `upcast-dem-measure:population`) for each municipality. These values are coming from two separate datasets which use components-measures of different specific data models (environmental, demographic), each organized into slices and observations according to the environmental model. The two datasets are joined by the common dimension of municipality.

For each municipality, the query calculates a derived indicator, pollution per person, by dividing the average air pollution value by the corresponding population count. This is expressed in the `SELECT` clause as `?airPollution/?population AS ?PollutionPerPerson`. The result provides a normalized measure that facilitates more meaningful comparisons across municipalities of varying sizes.

The results include the municipality identifier and the computed air pollution per person (`PollutionPerPerson`) and are ordered in descending order of `PollutionPerPerson`, highlighting the municipalities with the highest per capita exposure to pollution. The results are aligned by the same `?year`, while a specific `?date` can be used for reference

5.2.3 Environmental Monitoring Data Collected by Multiple Countries

A use case involving environmental monitoring data collected by multiple countries exploits this data integration approach. Each country publishes its own dataset of air pollution measurements, structured independently, using country-specific stations, frequency intervals (e.g., daily, monthly) and air pollutant types. Traditional integration would require flattening these datasets into a single schema or building complex mappings between each structure, something that can be fragile and time-consuming.

```
SELECT ?station ?airPollutant ?frequency ?year ?annualAvg
FROM <http://data.upcast.eu/cubemodeler/pilot/data>
WHERE {
  {
    ?dataset1 rdfs:label ?label1 ;
              upcast-dimension:frequency sdmx-code:freq-A ;
              qb:slice ?slice1 .
```

```
    ?slice1 ?stationProperty ?station ;
            upcast-env-dimension:airPollutant ?airPollutant ;
            upcast-dimension:year ?year ;
            qb:observation ?obs1 .
    ?obs1 a qb:Observation ;
          upcast-env-measure:airPollution ?aVal .

    ?stationProperty rdfs:subPropertyOf upcast-env-dimension:airStation .

    BIND(sdmx-code:freq-A AS ?frequency)
    BIND(?aVal AS ?annualAvg)
  }
  UNION
  {
    SELECT ?station ?airPollutant ?frequency ?year (AVG(xsd:decimal(?mVal))
        AS ?annualAvg)
    FROM <http://data.upcast.eu/cubemodeler/pilot/data>
    WHERE {
      ?dataset1 rdfs:label ?label1 ;
                upcast-dimension:frequency sdmx-code:freq-M ;
                qb:slice ?slice1 .

      ?slice1 ?stationProperty ?station ;
              upcast-env-dimension:airPollutant ?airPollutant ;
              qb:observation ?mObs .

      ?mObs a qb:Observation ;
            upcast-env-measure:airPollution ?mVal ;
            upcast-dimension:date ?mDate .

      ?stationProperty rdfs:subPropertyOf upcast-env-dimension:airStation .

      BIND(sdmx-code:freq-M AS ?frequency)
      BIND(xsd:integer(SUBSTR(STR(?mDate), 1, 4)) AS ?year)
    }
    GROUP BY ?station ?airPollutant ?frequency ?year
  }
}
```

Listing 6: SPARQL query that integrates environmental data accross multiple sources (countries) and time granularities

Each country's dataset has been modeled with its own DSD, preserving local semantics and metadata. The airStation dimension in each dataset is defined accordingly (`it-dimension:AirStation`, `gr-dimension:AirStation`, or `es-dimension:AirStation`) and is semantically linked via `rdfs:subPropertyOf` to the common broader property `upcast-env-dimension:airStation`. This subproperty hierarchy ensures that all semantically aligned data is returned, enabling unified access and querying across the various datasets. Moreover,

frequency values are harmonized through a shared SDMX codelist (e.g., daily, monthly, annual), according to the data model.

In this context, the query retrieves air pollution observations from the environmental Knowledge Graph, combining information about monitoring stations, pollutant types, temporal frequency and air pollution measurements. The query dynamically resolves dataset-specific properties for the station via `rdfs:subPropertyOf`. This makes it possible to query over the shared dimensions `upcast-env-dimension:airStation` and return relevant results across datasets that vary structurally, having defined different components for the respective concept. Additionally, a filter ensures that only records with standard monthly or annual reporting frequencies (`sdmx-code:freq-M`, `sdmx-code:freq-A`) are included, further refining and aligning the results.

To ensure comparability and integration between datasets of different temporal granularities, the query filters results to monthly and annual reporting frequencies (`sdmx-code:freq-M`, `sdmx-code:freq-A`). It then normalizes them to a common scale, leveraging the frequency dimension that has been defined for each dataset. Annual datasets are returned directly, while monthly datasets values are aggregated into yearly averages. In this way, results preserve their original temporal semantics while still allowing meaningful comparison across countries and datasets. Therefore, datasets of different temporal levels can be integrated with the appropriate queries. Additional component properties (e.g. exact observation dates) could be included if finer temporal resolution were required.

6 Evaluation

Evaluation focuses on the main aspects that determine the practical value of CubeModeler: workflow simplicity, model reusability and semantic flexibility. These dimensions reflect how easily users can define and adapt data structures, how much effort is required when new datasets are added and whether the integration process remains consistent as schemas evolve.

Traditional data integration methods often demand repeated schema design and complex mappings, whereas CubeModeler concentrates effort in the modeling phase. Once reusable components and Data Structure Definitions (DSDs) are defined, new datasets can be integrated with only minimal changes. This upfront investment enables predictable reuse and consistent interoperability across evolving datasets.

The evaluation of CubeModeler follows a twofold perspective: (i) the ability to support semantic reuse and modular scalability across heterogeneous datasets and (ii) the efficiency of RDFizing performance on representative cubes. At the modeling level, the focus is on reuse of components, adaptability under schema drift and the ability to maintain semantic consistency across evolving datasets. Performance measurements capture execution time, CPU and memory footprint. Together, these criteria provide a balanced assessment of CubeModeler in comparison with established mapping tools, highlighting both its practical efficiency and its support for evolutionary data integration.

6.1 Modular Scalability and Integration

This section evaluates CubeModeler's capacity to support modular scalability and semantic consistency in comparison with established mapping tools. The analysis considers semantic reuse, measuring how many components can be retained when integrating new datasets and verifying their alignment with the reference data model. It also examines integration scalability, where cumulative reuse ratios serve as a proxy for modeling effort as increasingly heterogeneous datasets are combined. Finally, dynamic

evolution is assessed through the incremental cost of incorporating a new dataset under schema drift, highlighting CubeModeler's ability to absorb changes smoothly while preserving consistency.

We designed seven scenarios (A2–D2) to reflect increasing levels of schema drift and heterogeneity. The A2 represents an identical dataset schema, B1–B3 introduce minor changes (renamed columns, alternative identifiers and an additional measure), C1 considers an aggregated cube (team totals) and D1–D2 involve new cubes (games schedule and play-by-play actions) that partially reuse existing dimensions while introducing new ones. A1 is the initial dataset used to establish the baseline data model and is not included in the comparison tables. It refers to player statlines, which includes dimensions such as Player, Team, Game, League, Season and basketball statistics measures (points, fouls etc.). Specifically, the eight total scenarios are:

- A1: players statlines across a season (base DSD, initial modeling)
- A2: players statlines across a different season (identical schema to A1)
- B1: players statlines across a season with two renamed columns (i.e. Index Rating $\rightarrow$ Ranking, Points $\rightarrow$ Pts)
- B2: Players Statlines with two different value types (i.e. playerID $\rightarrow$ playerName, teamID $\rightarrow$ teamName)
- B3: players statlines with one extra component (i.e. plus/minus measure)
- C1: team aggregated statlines (based on its players boxscores)
- D1: multi-cube integration $\rightarrow$ Play-by-Play Cube (dataset about the players actions in a game)
- D2: multi-cube integration $\rightarrow$ Games Cube (dataset about the schedule)

To ensure comparability across tools, reuse, effort and semantic accuracy are defined consistently but adapted to each workflow.

For CubeModeler, reuse is measured at the level of unique components (dimensions, measures, attributes) in the integrated data model. A component that appears in multiple cubes (e.g., player) is only counted once. The cumulative reuse ratio at each scenario is defined as:

$$\text{Reuse Ratio} = \frac{\text{Components already present before scenario}}{\text{Total components after scenario}}$$

New components are those introduced by the current dataset. This method reflects the incremental growth of a shared semantic model and avoids double-counting. Precision and recall remain by design at 1.0, since CubeModeler enforces alignment to the defined DSD and model hierarchy.

For Karma, reuse is evaluated at the level of predicate–object maps (POMs) in the exported R2RML mapping file. A mapping is considered reused if it preserves the same predicate IRI, subject template and

column binding as in the reference mapping. Changes due to renamed columns, new measures, or altered key templates are counted as new. Reuse is reported per cube, not cumulatively, because each dataset is mapped in a separate project/file and there is no global semantic model shared across cubes. This distinction highlights CubeModeler's main advantage: it enables true cumulative reuse, while baseline tools only replicate mappings independently. Precision/recall ranges capture the difference between ideal mappings (1.0/1.0) and mappings that drift under schema changes (e.g., one misaligned column out of nine gives recall 8/9 = 0.89).

For RMLMapper reuse is measured over the predicate–object mappings in the RML Turtle file. A mapping is reused if the same predicate and subject template are preserved and the column source remains stable. Any new or renamed column, or change in subject construction, requires new mapping lines. Like Karma, reuse is reported per cube, not cumulatively, since each dataset is handled in a separate file. Precision/recall follows the same logic as Karma: perfect alignment is possible in theory, but schema drift introduces risk, represented as ranges (considering a likely drift, not a worst-case scenario).

Manual effort was estimated using realistic weights derived from interaction costs. For CubeModeler, reusing a component requires ∼0.2 minutes (dropdown selection) and inserting a new component into a duplicated "ready" Cube, ∼1 minute; defining a new cube with all components incurs ∼5 minutes overhead. For Karma, effort reflects both project setup (∼5 minutes to load a CSV and define the subject template) and per-binding edits (∼1 minute each for renamed or new columns). Additional operations, such as adjusting subject templates or verifying joins and datatypes, are also included in the estimates (typically adding 1–3 minutes depending on the scenario). For RMLMapper, effort corresponds to editing lines in a Turtle mapping file (∼0.5–0.8 minutes per predicate-object map), with ∼4 minutes for boilerplate setup when starting a new mapping file. Subject template updates or structural changes may add another 0.5–1 minute. These values ensure that the effort estimates in Table 1 reflect the full set of manual actions required - not only the new components - and provide a consistent basis for comparing tools.

Table 1. CubeModeler component reuse and consistency across scenarios.

Scenario	Total comps	Reused	New	Reuse Ratio (%)	Consistency
A1	25	–	25	–	✓
A2	25	25	0	100	✓
B1	25	25	0	100	✓
B2	25	25	0	100	✓
B3	26	25	1	96	✓
C1	25	25	0	100	✓
D1	29	25	4	86	✓
D2	34	29	5	85	✓

Table 1 presents the measured component reuse within CubeModeler. Scenarios A2–B1–B3 show ∼100% reuse, confirming that column renames and identifier changes require no new modeling, as they are mapped directly to the appropriate DSD components. Especially for B3, this is accomplished by selecting the relevant property of the corresponding codelist. B3, where a new measure is introduced and C1, which shifts the granularity to team aggregates, each require only one new component, yielding reuse ratios above 95%, as the user can duplicate and alter accordingly the previous scenarios' DSD. In

D1 (Play-by-Play cube) and D2 (Games cube), cumulative reuse drops (∼86% and ∼85%, respectively) because new cubes introduce genuinely new dimensions, but overall reuse remains high thanks to shared components. Crucially, semantic consistency is enforced across all scenarios. These results demonstrate CubeModeler's modular scalability: most of the model is reused even as more heterogeneous datasets are integrated.

Table 2. Cross-tool comparison of reuse, effort and semantic accuracy. CubeModeler values are cumulative; Karma and RMLMapper are per-cube.

Scenario	Metric	CubeModeler	Karma	RMLMapper
A2	Reuse (%)	100	100	100
	Effort (min)	0.0	0.0	0.0
	Precision/Recall	1.0/1.0	1.0/1.0	1.0/1.0
B1	Reuse (%)	100	∼92	∼92
	Effort (min)	0.4	1.2	0.8
	Precision/Recall	1.0/1.0	0.92–1.0	0.92–1.0
B2	Reuse (%)	100	∼80	∼85
	Effort (min)	0.4	4.0	3.0
	Precision/Recall	1.0/1.0	0.92–1.0	0.92–1.0
B3	Reuse (%)	96	∼95	∼95
	Effort (min)	1.0	1.2	0.8
	Precision/Recall	1.0/1.0	0.95–1.0	0.95–1.0
C1	Reuse (%)	100	∼70	∼75
	Effort (min)	1.0	17.0	11.2
	Precision/Recall	1.0/1.0	0.75–1.0	0.80–1.0
D1	Reuse (%)	86	∼65	∼65
	Effort (min)	5.0	16.0	10.6
	Precision/Recall	1.0/1.0	0.84–1.0	0.84–1.0
D2	Reuse (%)	85	∼60	∼60
	Effort (min)	5.0	14.0	9.4
	Precision/Recall	1.0/1.0	0.85–1.0	0.85–1.0

Table 2 compares CubeModeler with Karma and RMLMapper across all scenarios. For identical schemas (A2), all tools perform equally well. Under schema drift (B1, B2), CubeModeler maintains 100% reuse and perfect precision/recall with negligible effort (¡1 min). Karma and RMLMapper require manual edits, reducing reuse (to ∼92% for renames and ∼80–85% for identifier changes) and precision/recall to ranges such as 0.92–1.0. In B3 (+1 measure), CubeModeler again requires only a single new component (96% cumulative reuse), while baselines achieve similar reuse but with higher manual effort and no guarantee of consistency.

For C1 (team aggregates), CubeModeler reuses all components, achieving 100% cumulative reuse and full consistency. Karma and RMLMapper must essentially rebuild the mapping, reducing reuse (70–75%) and increasing manual effort (9–17 minutes). Finally, in the multi-cube scenarios D1 (Play-by-Play) and D2 (Games), CubeModeler retains high cumulative reuse (∼86% and ∼85%) and semantic consistency, whereas Karma and RMLMapper must remap large portions of the schema independently,

resulting in reuse of ∼65% and ∼60%, respectively, with higher effort and precision/recall dropping into ranges such as from 0.84–0.85 to 1.0.

Taken together, these results highlight CubeModeler's distinctive advantage: semantic consistency and high reuse are guaranteed by design, whereas baseline tools can, in principle, achieve perfect mappings but in practice rely on manual editing and are exposed to drift as datasets evolve. These findings further confirm that CubeModeler front-loads the integration effort at the modeling phase. By defining reusable components (dimensions, measures, attributes), configuring DSDs and linking coded values through SKOS hierarchies, subsequent datasets can be aligned without altering the core structure. Once the model is established, transformations become guided and repeatable, and queries can be written against shared components rather than dataset-specific schemas. This upfront investment yields measurable savings in later phases, as demonstrated by the consistently high reuse ratios and minimal effort requirements across scenarios.

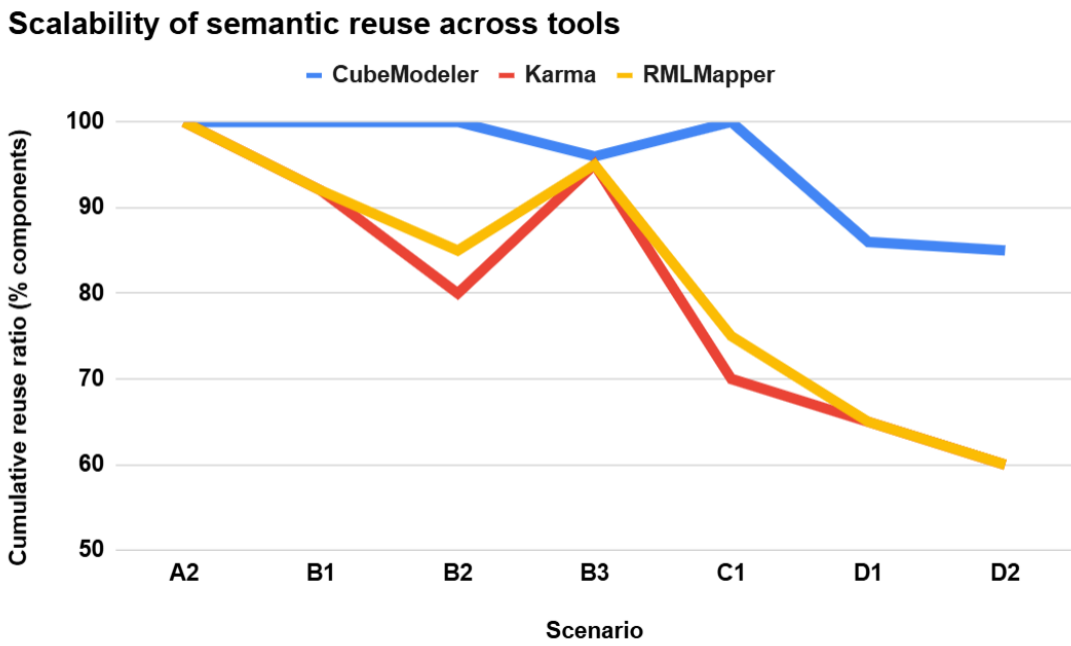

Figure 12. Scalability of semantic reuse across tools

To visualize these trends, we plotted (figure 12) the cumulative reuse ratio (%) across scenarios (A2→D2). CubeModeler maintains consistently high reuse levels, ranging between 85–100% across all integration steps. In contrast, Karma and RMLMapper show a much steeper decline in reuse as scenarios become more complex, with values falling toward 60% in the final cubes. This divergence demonstrates CubeModeler's ability to sustain semantic reuse even in heterogeneous and evolving datasets, while baseline tools lose efficiency and consistency as integration demands increase.

6.1.1 Dynamic evolution

Beyond scalability, we examined the incremental cost of integrating a new dataset into an existing model. Scenario B1 (two renamed columns) illustrates this case. In CubeModeler, the renamed fields were simply rebound to existing components, requiring less than one minute with guaranteed correctness. Karma and RMLMapper required reopening or editing the mapping and manually reassigning the renamed fields, with higher effort (1–2 minutes) and risk of inconsistency. While this difference is modest for a single case, its impact accumulates across scenarios: CubeModeler sustains high reuse and consistency throughout, whereas the baselines show a progressive decline in reuse and accuracy as schema changes and new cubes are introduced. This micro-view confirms CubeModeler's strength in dynamic evolution, where incremental changes are absorbed smoothly without compromising semantic alignment.

To this end CubeModeler achieves near minimal-effort scalability. Once a data model and DSDs are in place, new datasets can be integrated through preconfigured settings with little or no structural modification. Hierarchical concept schemes further allow new components to be introduced without disrupting existing queries, ensuring that RDFization workflows remain stable even as schemas evolve. This makes CubeModeler particularly effective for heterogeneous inputs, decentralized publishing contexts, and integration tasks involving multiple data providers.

6.2 Performance Overview

The RDFizing performance of CubeModeler was evaluated on representative datasets from both the sports and environmental domains. These included a full season of boxscores statistics (players statlines), a play-by-play dataset covering random rounds, a games dataset and a longitudinal air pollution dataset (56 years). Each dataset had different complexity in terms of number of columns and contained between 8K and 20K rows, yielding 200K–250K RDF triples after transformation.

Tests were performed on a standard workstation (Windows 10, Apache + PHP/XAMPP, 8GB RAM, Intel(R) Core(TM) i5-8500T @ 2.10GHz CPU) without any special optimization. Resource usage was monitored using Windows Performance Monitor with one-second sampling. For each run we measured the active duration (from the time CPU usage exceeded 5% until it returned to idle), the peak CPU load (per process) and the peak memory consumption (Working Set).

Across datasets with 8K–20K rows, CubeModeler RDFizing completed within 17–34 seconds, consistently saturating one CPU core (peak ∼100%). For smaller datasets, the process was near-instantaneous. Most importantly, memory consumption remained extremely low, with peaks between 55 MB and 117 MB depending on dataset complexity, as shown in Table 3.

Table 3. RDFizing performance of CubeModeler across representative datasets.

Dataset	Rows	Triples	Duration (s)	CPU peak (%)	RAM peak (MB)
Games – 1 season	330	4K	1	–	52.3
Boxscores – 1 season	8K	227K	17	101.0	60.0
PbP – 1 round	4.5K	72K	2	101.3	44.0
PbP – 3 rounds	13.5K	210K	30	101.5	117.0
Air Pollution – 56 years	20K	250K	34	101.6	55.4

For comparison, RMLMapper requires considerably higher memory usage even for small datasets. A dataset of 10K rows × 20 columns (∼200K triples) required ∼8 seconds and ∼1.4 GB RAM, while a dataset of 1M rows × 30 columns (∼20M triples) required ∼262 seconds and ∼13.2 GB RAM; at 10M

rows, RMLMapper typically fails due to out-of-memory errors [43]. Karma, while designed as a semantic integration environment with broad source support, has no peer-reviewed performance benchmarks to our knowledge; nevertheless, its Java-based, in-memory architecture suggests memory demands comparable to or higher than RMLMapper when scaling beyond small datasets.

More recent engines such as SDM-RDFizer [44, 45] and Morph-KGC [46] introduce duplicate-aware operators, mapping partitions and optimized execution strategies that allow them to reduce memory overhead and improve performance on medium and large tabular datasets. These outperform RMLMapper and can process datasets in the million-row range with significantly lower peak RAM, though still in the hundreds of MB to several GB depending on scale.

Table 4. RDFizing performance comparison on small–medium datasets.

Tool	Dataset Size	Triples	Time	Peak Memory	Notes
CubeModeler	8K–20K rows, 10–25 cols	∼200K–250K	17–34 s	**55–117 MB**	Low footprint, saturates one CPU core; nearly instantaneous on very small datasets.
RMLMapper	10K rows × 20 cols	∼200K	∼8 s	∼1.4 GB	Comparable time to CubeModeler, but much higher memory usage.
RMLMapper	1M rows × 30 cols	∼20M	∼262 s	∼13.2 GB	Scales poorly; fails at ∼10M rows with out-of-memory.
SDM-RDFizer	10K–100K rows (GTFS-Madrid-Bench)	200K–2M	Seconds–tens of seconds	Hundreds of MB	Faster than RMLMapper; duplicate-aware execution reduces memory/time overhead.
Morph-KGC	100K–1M rows (GTFS-Madrid-Bench)	∼2M–20M	Seconds–minutes	<8 GB	Uses mapping partitions to reduce peak memory; outperforms RMLMapper and SDM-RDFizer in medium-scale scenarios.

These results confirm that CubeModeler can transform and integrate regular cubes quickly and with minimal resource requirements, making it practical for public administrations, researchers and data providers with limited infrastructure. While it is not designed as a large-scale RDFizing optimizer and query optimization for very large datasets is deferred to future work, the framework already provides a competitive, lightweight solution for typical multidimensional cubes. Its efficiency ensures that semantic interoperability can be achieved without heavy infrastructure, complementing more performance-oriented tools in the knowledge graph construction landscape.

Beyond RDFizing performance, we also conducted preliminary tests on query execution performance to assess how CubeModeler handles integrated queries over varying dataset sizes. For the first environmental query (Section 5.2.1), execution times remained low on smaller slices of the dataset (0.315s for one year and 0.622s for five years), while increasing more noticeably with larger temporal spans (9.34s for ten years). Although these tests are not exhaustive, they indicate that CubeModeler already supports interactive querying on small-to-medium data ranges, while performance on larger datasets will require optimization. As noted in the Future Work section, query optimization strategies (e.g., plan tuning, caching) remain a priority for further development.

6.3 Limitations

While CubeModeler approach offers significant advantages in terms of modularity, semantic alignment and cross-dataset querying, especially for RDF Data Cubes, it is not without effort or constraints.

The initial modeling phase, including defining the data model, building reusable components and configuring Data Structure Definitions (DSDs), requires a clear understanding of the data domain and structure. Creating accurate codelists and aligning coded values to SKOS properties also demands careful curation, particularly when integrating heterogeneous datasets, but this task can be common to every approach. The graphical interface simplifies transformation tasks. However, users must still have a solid understanding of SPARQL and the semantics of modeled components to formulate expressive queries. These modeling demands are shared, to varying degrees, across all approaches.

Query optimization [47, 48] for large-scale datasets and multi-cube integrations [49] has not yet been explored. Efficient query planning, caching strategies and indexing mechanisms are recognized as critical for ensuring scalability and responsiveness, as the number of datasets and users grows. These aspects exceed the current scope of this paper but are planned as next steps in the development of CubeModeler, aiming to strengthen its applicability to larger, production-scale environments.

It is also important to recognize that alternative integration approaches may outperform CubeModeler in contexts in which data are fully tabular, stable, or domain-specific (e.g., OLAP for high-performance aggregations, or traditional ETL for quick one-off integrations). Ontology-based or federated semantic methods may also be preferable when dealing with existing published RDF datasets or when deeper reasoning over formal ontologies is required. Thus, while CubeModeler is well-suited for scalable integration of multidimensional statistical datasets, its effectiveness is strongest when there is modeling control, repeated structure and a need for long-term interoperability across evolving data sources.

This does not imply that the approach fails when applied to noisy or more complex datasets. On the contrary, the concept of modular data models and reusable components can also be leveraged in such cases. For unstructured datasets where no explicit relations exist between entities, the respective components can still be defined, with CubeModeler initially serving as an RDFizing and data model definition tool rather than a direct integration framework. At later stages, relations between components may be established to support scalability and enable semantic integration. In this way, related components can progressively construct the links needed for integrating heterogeneous datasets. Moreover, generic concepts such as time period, frequency and reference area remain broadly applicable across diverse cases, while the underlying infrastructure supports the semantic enrichment of each domain's data ecosystem.

7 Conclusions

This work presented a modeling-based integration approach for heterogeneous statistical datasets, realized through the RDF Data Cube Vocabulary. The approach shifts integration to the modeling stage, where datasets are described through modular Data Structure Definitions and reusable components, ensuring consistency and interoperability across diverse domains such as sports statistics and environmental measurements. To operationalize this methodology, we developed CubeModeler, a semantic modeling framework that supports the construction of DSDs, coded hierarchies and reusable workflows for semantic transformation and querying.

The proposed approach demonstrates that semantic integration can be achieved without rigid schema unification. Instead, it leverages shared components, hierarchical relations and SKOS-coded codelists to support expressive and reusable SPARQL queries, as illustrated through six representative examples ranging from cross-league player statistics in basketball to comparative analysis of environmental measurements. These examples highlight how the approach supports the combination of contextual, temporal and multidimensional conditions in a seamless way, showing how complex yet intuitive querying across semantically aligned datasets can be performed.

While the case studies focus on basketball statistics and environmental monitoring, the proposed approach and its operationalization through CubeModeler are generalizable to any domain involving statistical datasets. By centering on modular DSDs, reusable semantic components and coded hierarchies, the framework can be applied to contexts such as health indicators, fiscal statistics, transportation, or demographics. In fact, a data model has already been implemented and connected with environmental indicators within the UPCAST pilot. The diversity of the two case studies was intentional: basketball data demonstrates integration in a centralized and structurally stable environment, while environmental data highlights decentralized, heterogeneous and evolving sources. Together, they illustrate the transferability of the approach across very different integration scenarios.

Overall, this work aims to contribute a practical and extensible solution for semantic data integration. Beyond the two use cases presented, the modeling-driven approach is generalizable to a wide variety of statistical domains. By promoting modularity, reuse and interoperability, the CubeModeler framework offers a practical and extensible foundation for interoperable knowledge graphs, enabling unified access to structurally diverse datasets and supporting richer analytical possibilities across domains.

The approach also adheres to the FAIR principles (Findability, Accessibility, Interoperability and Reusability) by promoting standardized, semantically rich data modeling practices. Leveraging RDF, RDF Data Cube Vocabulary and SKOS-coded components, it enables structured and machine-readable representations that support both human and programmatic access. Interoperability is enhanced through modular Data Structure Definitions (DSDs) and the reuse of well-defined dimensions and measures, ensuring semantic alignment across datasets. Furthermore, the declarative integration model facilitates dataset discoverability and reproducibility by making all components and codelists explicitly defined and queryable within a knowledge graph. This standards-driven approach aligns with broader efforts in open data ecosystems and scientific data management to ensure that data remains not only technically accessible but also contextually meaningful across domains and over time.

In the context of open government data, the proposed approach promotes transparency and accountability by enabling semantic access and structural comparability across public datasets. Its support for standardized metadata, harmonized identifiers and shared conceptual models aligns with practices adopted by national and international open data portals. While the FAIR principles address technical stewardship, the CARE principles (Collective benefit, Authority to control, Responsibility and Ethics) offer a complementary perspective, particularly in contexts involving sensitive or community-governed data. Although CubeModeler is primarily designed for semantic and structural interoperability, its modular and transparent architecture can accommodate CARE-aligned extensions in which ethical data governance is required.

8 Future Work

Future work will extend CubeModeler with services that further facilitate manual tasks and enhance scalability. A key direction is the integration of semantic alignment and ontology matching mechanisms into the data modeling process. These will provide suggestions when building or enriching the model, supporting users in defining appropriate components and relations more efficiently. Exploring large language models (LLMs) for ontology matching also highlights promising opportunities to reduce manual effort and could be adapted into this workflow. They will also improve reuse and hierarchy development, while promoting links to established domain ontologies. In this context, the ability to construct custom RDF codelists directly from user input will also be explored, complementing the current codelist import functionality. Together, these advances will strengthen the modeling process and make CubeModeler applicable even to noisy or unstructured datasets. In such cases, incomplete or inconsistent structures can still be represented in modular form and progressively integrated. Furthermore, RDF Data Cube Vocabulary extensions toward spatio-temporal and real-time contexts suggest additional directions for CubeModeler's evolution, especially when scaling to larger datasets and integrating heterogeneous sources. Another priority is query optimization for large-scale datasets and multi-cube integrations. Efficient strategies such as query planning, caching mechanisms and indexing will be investigated to ensure scalability and responsiveness, particularly in production-scale environments.

Funding

This research was funded by HORIZON EUROPE Digital, Industry and Space grant number 101093216 (project name UPCAST).

Appendix A CubeModeler – Functional Walkthrough

This appendix expands on Section 4 by providing a more technical walkthrough and user interface configuration details of CubeModeler. This section presents the CubeModeler functionalities in more detail, with some explanatory steps included to guide the reader through the UI. The primary focus, however, is to demonstrate how the framework addresses various challenges regarding data integration, as well as the proper data modeling that supports it.

Appendix A.1 DSD Creation Workflow

Building a Data Structure Definition (DSD) is essential for aligning dataset columns with data model properties in RDF Data Cubes. CubeModeler supports selecting relevant components from a data model and filtering by type (dimension, measure, attribute).

From the list of all related components that appear, any of them can be selected to be included in the new DSD, while the show button next to each component displays information about them, such as their RDF/TTL definition of the respective component in the data model. Thus the user has a clearer view of what a component is about and what components the new DSD has to include in its definition, facilitating the non-trivial task of building the appropriate structures.

After the selection of the suitable components for the new DSD, additional but also necessary configurations should be made towards the final structure of the DSD. These configurations concern the DSD metadata (prefix, IRI, label and comment), the components' attachment level to the cube (dataset, slice, observation) and if it is required (in case of an attribute) and finally, the slice metadata (slice key, prefix, IRI, label and comment), if any. This input is inserted in the DSD configuration feature of CubeModeler, after the selection of the components.

Finally, two supplementary fields are available for extended use: the spatial and temporal ranges of the DSD. Defining these ranges explicitly clarifies the datasets for which the DSD can be applied. For example, if a DSD needs to be adjusted for a different year (i.e. because datasets from previous years have fewer columns-components than the more recent ones), the temporal unit that could be defined would be the year. To this end, country, region, municipality spatial units and year, month temporal units can be defined to distinguish the use of similar but not the same DSDs. This additional input is intended rather for internal use in CubeModeler and is not included in the DSD definition nor during the semantic transformation of the data. However it is useful when dealing with a big number of datasets variations that may differ in that specific temporal and/or spatial unit and need to use another DSD, facilitating also the user to build the appropriate structures.

To this end, an option to duplicate a current DSD is provided, after it has been created. This feature supports the definition of a new DSD having as a starting point the selected, already created DSD, for cases in which the similarities are more than the differences between those DSDs. In this case, the same components of the initial DSD can be reused and after any necessary adjustments to the structure (components and components attachment level updates), the new DSD is created. If the latter applies to a different temporal or spatial unit, it would be useful to denote this, for clarification.

To finalize the creation of the initial DSD, a preview of the produced DSD is displayed to the user, to validate its final definition and proceed to any updates on the aforementioned properties, if needed. Finally, a new DSD that will form the base for the RDF cube transformation for the corresponding datasets is created in TTL format. For all the DSDs created, the view, edit, download and duplication options are available for the users through the DSDs catalog of the CubeModeler, where the capability of uploading them to the knowledge graph is provided too. The created DSD can be reused for similar datasets with the same structure. An example DSD is presented in Figure A1, as a product of CubeModeler.

Figure Appendix A.1. example DSD (view through the CubeModeler's DSD Catalog).

Appendix A.2 Semantic Transformation Workflow

The RDFizing process is based on the built DSD for the dataset and an interface where the user can configure all the transformation settings, thereby generating RDF properties. The process is straightforward and after a dataset

is uploaded by the user, a mini preview of it is displayed for validation. An option to select whether the dataset includes or does not headers for each column is also provided. The most important part is the selection of the DSD, that has already been defined through the respective CubeModeler service, from a list with CubeModeler DSDs; that will be the DSD on which the mapping will take place. Also, the dataset IRI for which triples will be created for, can be defined here. If the dataset has already been described within CubeModeler, that means it includes a `qb:structure` property, then a list of the corresponding datasets IRIs where the selected DSD has been "assigned to" is suggested to the user. These options are presented in Figure A2.

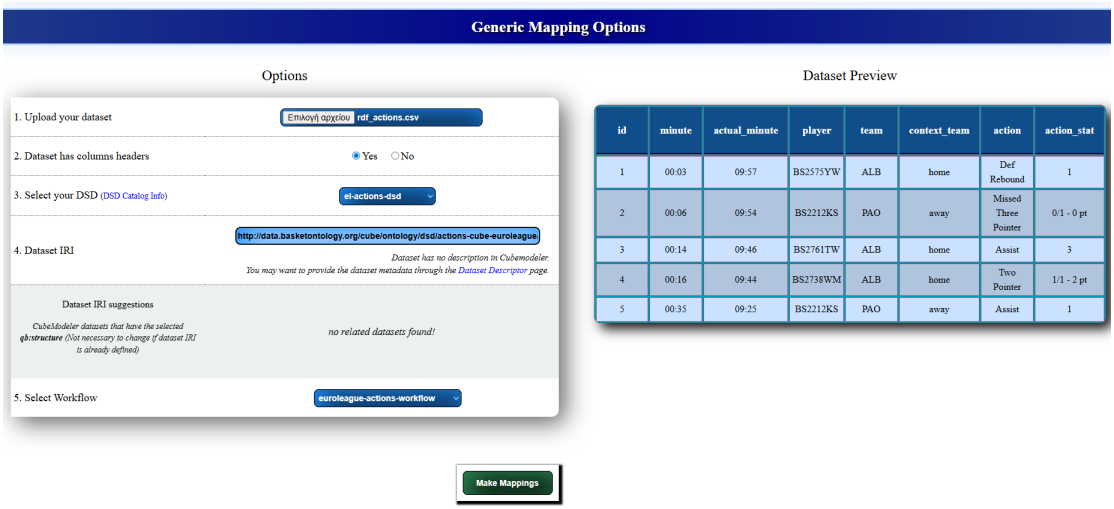

Figure Appendix A.2. Basic input page for the RDFizing process of a dataset.

Having this preliminary but requisite input, users configure transformation settings through CubeModeler's semantic transformation interface, shown in Figure A3 by aligning each DSD component (dimension, measure, attribute) with a corresponding dataset column. For example, the "points" measure in a player statistics DSD would be matched to the "points" column in the source dataset. Thus, one of the dataset's columns has to be selected for each of the DSD components.

Additional settings define the attachment level of each component. Dataset level attached components include only a text field as input, because their value doesn't really relate to the dataset's columns and values, rather it is a standard value that can be inserted manually. However, this can also be the case for the slice level attached components, but not always. It depends on modeling decisions. Thus, the user can select whether the slice value should be an external fixed value for the dataset, or whether numerous slices should be created based on the dataset values. Finally, all observation level attached components should be matched to one or more dataset columns.

Regarding the DSD - dataset alignment, there are also several scenarios, depending on the contents of the dataset. The CubeModeler service allows for various mapping options, depending on the case, that the user is more familiar with. Thus, for slice level attached components mapping, there are two routes: in the first, the final RDF will include one slice for each individual dataset row, while in the second, dataset rows with the same values should be grouped

together and form only one slice. While the latter is the most common scenario, there should be cases in which the flexibility of one to one mapping of slice attached components could be useful.

If there is no apparent direction on how to make the DSD - dataset alignment (probably when there are several slice level attached components), CubeModeler offers the "smart slice mapping" feature. In this case, CubeModeler's algorithm recognizes common values within the dataset and formulates accordingly the slices of the RDF dataset, if any. The algorithm takes into account all the components that are attached to the slice level, so the grouping extends to all these dataset columns, identifying the sets of common values within them to create the corresponding slices and of course, their containing observations. Thus, the number of observations per slice is determined by the mapping options through this interface.

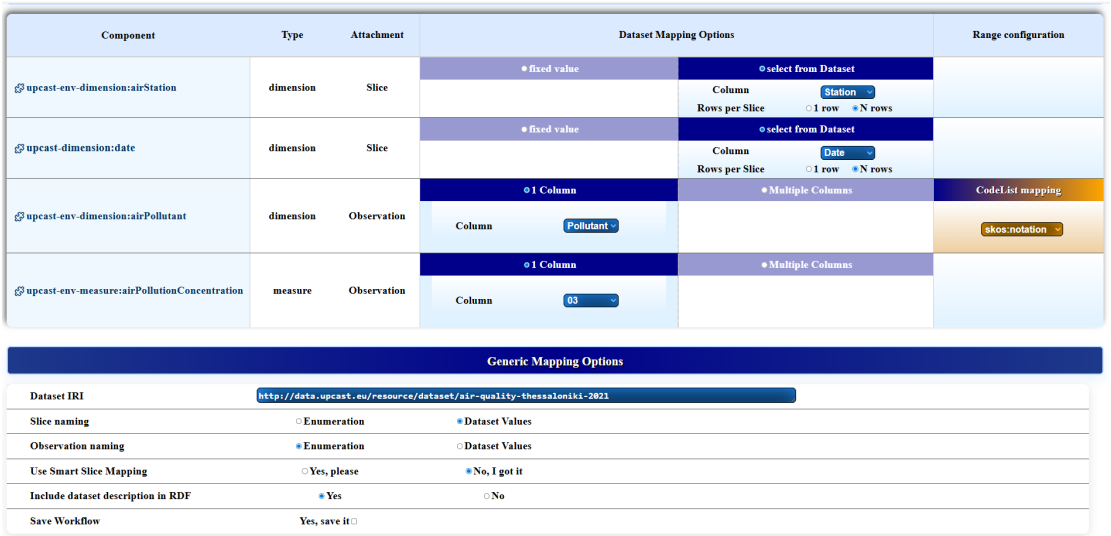

Figure Appendix A.3. CubeModeler's user interface configuration setttings for semantic transformation of datasets.

For coded components like dimensions or attributes using codelists, the aligned dataset column values are matched to a specific property of the corresponding codelist, as defined in the component's configuration. If this is not defined, in case of generic components with no specified codelist in their definition, the user may select a codelist that exists in CubeModeler. Then, the dataset values of the column that has been aligned with the component, are matched to the respective codelist property values, where the property can be also configured by the user (`skos:notation`, or `skos:prefLabel`).

During the RDFizing process, the algorithm queries the knowledge graph that already contains that codelist (through the Codelist Import feature presented later) and matches the dataset values with the respective codelist property values. It then replaces the dataset values with the IRIs of the matched concepts, thus creating valid RDF triples. Besides this nested matching process, CubeModeler also allows for direct mappings with the codelist, constructing essentially the component value by composing the codelist prefix and the respective dataset value.

Once the main RDFizing settings are configured, users can set additional parameters before running the semantic transformation. There, the options of whether the slices and observations IRIs should

be identified by a specific number, or by concrete columns values are provided. In the first case, a distinct counter number is assigned to each slice and observation, while in the second case, the values of the respective aligned columns of the slice and observations form the final IRI. Combinations of these selections (e.g. values for slices and numbers for observations) are also allowed. For example, the following observation IRI `http:/data.basketontology.org/cube/resource/dataset/statlines-players-cube-euroleague/observation/5/2361` results from a number selection for both slices and observations, while the following `http:/data.upcast.eu/cube/resource/dataset/greek-municipalities/air-quality-sindos-2021/observation/05-01-2021_PM10/20` combines the values of the slice level attached components to form the slice identifier part in the IRI (the last part '20' is also a value, denoting the hour of the day, for that observation).

Other input regards the selection of the smart slice mapping feature mentioned before while CubeModeler also allows users to save RDFizing configurations as reusable workflows tied to a specific DSD. When transforming a similar dataset later, the saved workflow can be loaded to automatically apply all previous settings, enabling quick execution with minimal input. Users can also customize these settings as needed and save them as a new workflow for future use.

Finally, the dataset is totally transformed to RDF in TTL format, based on the user configuration settings and is checked through a TTL validator in order so it can be uploaded to the knowledge graph. If the syntax of the produced TTL is correct, then the options of local downloading, as well as the upload in the knowledge graph are displayed, among others, to the user. If not totally correct, however, there is an alternative path. CubeModeler includes an integrated RDF editor for correcting syntax errors that may arise from malformed source data or other issues during RDFizing. Users can review error messages from the TTL validator, make necessary fixes within the editor and then upload the corrected RDF to the knowledge graph.

The knowledge graph is hosted on a Virtuoso RDF triple store, where all CubeModeler outputs are stored. Due to server limitations on INSERT query size, large TTL files are automatically split into smaller parts for sequential upload. The algorithms identify the structure of the dataset (i.e. whether there are slices or not in it), as well as the major separators (usually a punctuation mark or the greater than character with which an entity ends, or both) of the produced TTL and then group the chunks by size, thus the ones of smaller size can be inserted more massively (e.g. by groups of 10-20) than the bigger ones. Additional partitioning may apply in the bigger chunks, by extra separators. Finally, all the triples are uploaded in the knowledge graph stored in Virtuoso.

Once the dataset TTL is uploaded to the knowledge graph, its triples become accessible via SPARQL queries, while the user may start again the RDFizing process with another dataset, or use another functionality of CubeModeler.

Appendix A.3 Dataset Description

Describing datasets involves filling out a metadata form using fields from standard vocabularies like DCAT, PROV and TERMS. Required inputs include the dataset's IRI, associated DSD (`qb:structure`) and a descriptive label. Additional properties can be inserted through a TTL code frame, for any further definitions not included in the current fields (for example, defining the RDF entities that are used within the standard form properties). This feature also provides a helping sidebar with tips about the correct input in respect to each field. Finally, an RDF dataset description in TTL format is created and can be uploaded to the knowledge graph as well.

For all the datasets descriptions defined, the view, edit and download options are available for the users through the datasets descriptions Catalog of CubeModeler, where the capability of uploading them to the knowledge graph

is provided too. An extra button linking to the CubeModeler RDFizer and using the respective Dataset IRI in the relevant field for direct proceeding to the semantic transformation process, is also present in the Catalog.

Appendix A.4 Data Model Editing

Adding new components in the data model through CubeModeler is carried out through a tree visualization of the data model and all its components (dimensions, measures, attributes). The whole hierarchical structure of the data model is presented in the visualization which has a horizontal orientation. It begins with the broader components on the left side and progresses to the more specific components on the right, connected by arrows that represent an `rdfs:subPropertyOf` property between two related components. The tree visualization allows for the filtering of components per type and per data model (if more are used). This visualization can also be used as a reference, for a better understanding of the meaning of each component and where it stands within the data model's hierarchy, making clearer which ones should be selected when defining a DSD.

In the tree visualization, users can add a new component by selecting an existing one and choosing a relation type. A narrower component means that the new component is an `rdfs:subPropertyOf` the selected (from the tree visualization) component, a broader component means that the selected component is a `rdfs:subPropertyOf` of the new component, while a related component stands in the same hierarchy level and may have the same "parent", later. This enables structured placement of components within the data model. If no appropriate relation exists, components can also be added independently, without linking to existing ones. There is also an infobox displaying the basic properties of the respective component's definition, when clicking on it in the visualization, for assistance.

The new component is then defined by providing values for properties such as IRI, label, type, range, concept, optional codelist and `rdfs:subPropertyOf` if applicable. As it is the case with the CubeModeler's DSDs, spatial and temporal resolution of the component can also be defined, indicating the specific range that applies. For example, a `gr-dimension:airStation dimension` (country spatial unit with value "Greece") can be defined as a `rdfs:subPropertyOf` of a generic `airStation` dimension. Once validated, the new component is stored in CubeModeler's database and integrated into the updated data model. It is positioned within the component hierarchy based on the initially selected relation, maintaining the model's structural integrity.

For direct `rdfs:subPropertyOf` relations, this task is straightforward (once a component has a `rdfs:subPropertyOf` property, it can't have another one, while a parent may have multiple children). When selecting the "related" relation, CubeModeler places the new component in the same level with the referred component, having the same parent (if the related component has already). Moreover, if a broader component is to be defined, a new import component functionality is displayed to the user, if the respective `rdfs:subPropertyOf` property has been filled in by them with an unknown yet data model component.

This service allows ongoing enrichment of data models within CubeModeler and supports building models from scratch, though for large models, direct SQL insertion into the database is recommended for efficiency.

Appendix A.4.1 Validate Components

Newly added components in CubeModeler must be validated by an admin user to ensure correctness and prevent disruptions to the semantic transformation and data integration process. This task takes place through the components validator functionality, where the components that are related to the data model can be activated or deactivated. Thus, at any time, a new, updated version of the data model, regarding the components it includes, can be formatted and be automatically presented in the updated tree visualization and subsequently used for the RDFizing task.

Appendix A.4.2 Edit Relations

The relations editor allows users to define `rdfs:subPropertyOf` relationships by assigning a selected component as the parent of another that lacks this property. This is the case when a child component has been inserted before the parent and their relationship hasn't been defined yet (because the parent for example has been inserted as a narrower component of another component and not as a broader of this one, through the tree visualization, thus no relation between them exists yet). This feature supports flexible updates to the modular, hierarchical structure of data models, enabling easy enrichment as new components are added. The algorithm used takes into account any inconsistencies within the data model's hierarchy, so a component of a lower level in the hierarchy can't be an option for the `rdfs:subPropertyOf` value of the selected component.

Appendix A.5 Importing Codelists

Codelists are hierarchical structures of concepts, described mainly through the SKOS vocabulary. In the view of CubeModeler, any codelist is defined as a SKOS codelist. The codelist import is carried through a metadata form where the user defines the codelist's information. To import a codelist, users provide basic metadata such as its IRI, prefix, label and Concept Scheme, along with an internal name for storage. Optionally, they can link it to a related component in the data model for automatic use in future transformations. The codelist is then uploaded as a TTL file, validated for syntax and stored in both the knowledge graph and CubeModeler's database. A catalog of all the codelists that have been imported into the CubeModeler but also in the knowledge graph and can be used directly for mapping and semantic transformation options is present through the Codelists Catalog of the CubeModeler, where the metadata info and the view (codelist in TTL format), edit (metadata) and download (codelist in TTL file) options are available to the users. All CubeModeler's products are downloadable.

Appendix A.6 User Guidance

The CubeModeler interface has been designed to support users who may not be deeply familiar with semantic technologies. Its primary goal is to make semantic operations accessible through a guided and intuitive user experience. Most interactions are carried out via straightforward text forms, while in certain cases a TTL-syntax input must be provided. To further assist users, each page includes helping sidebars that display information about components and properties. Components are typically presented in TTL format across CubeModeler pages, while the Dataset Description page explicitly shows the required information for each property together with a direct link to the corresponding vocabulary definition (see Figure 8 in Section 4.3). The RDFizer interface also offers convenient customization options, such as smart slice mapping, entity naming selections and selective inclusion of relevant datasets in the final TTL output. Overall, CubeModeler deliberately minimizes the need for direct code writing. Instead, it enables users to focus on the conceptual aspects of their domain and on the basic understanding of RDF, aligning with the tool's overarching design philosophy of accessibility and usability.

Appendix B Prefixes

```
prefix qb: <http://purl.org/linked-data/cube#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix skos: <http://www.w3.org/2004/02/skos/core#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
```

```
prefix foaf: <https://xmlns.com/foaf/0.1/>
prefix dbo: <http://dbpedia.org/ontology/>
prefix schema: <http://schema.org/>
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix sdmx-dimension: <http://purl.org/linked-data/sdmx/2009/dimension#>
prefix upcast-dimension: <http://data.upcast.eu/cube/ontology/dsd/dimension/>
prefix upcast-env-dimension: <http://data.upcast.eu/cube/ontology/environment
    /dsd/dimension/>
prefix upcast-env-measure: <http://data.upcast.eu/cube/ontology/environment/
    dsd/measure/>
prefix upcast-dem-measure: <http://data.upcast.eu/cube/ontology/demographics/
    dsd/measure/>
prefix bco: <http://data.basketontology.org/cube/ontology/>
prefix bco-dimension: <http://data.basketontology.org/cube/ontology/dimension
    />
prefix bco-attribute: <http://data.basketontology.org/cube/ontology/attribute
    />
prefix bco-measure: <http://data.basketontology.org/cube/ontology/measure/>
prefix bco-concept: <http://data.basketontology.org/cube/concept/>
prefix bco-codelist: <http://data.basketontology.org/resource/codelist/>
prefix teams-euroleague-2025: <http://data.basketontology.org/cube/resource/
    codelist/euroleague-teams-2025/>
prefix players-euroleague-2025: <http://data.basketontology.org/cube/resource
    /codelist/euroleague-players-2025/>
prefix bco_league: <http://data.basketontology.org/cube/resource/codelist/
    league/>
prefix bco_season: <http://data.basketontology.org/cube/resource/codelist/
    season/>
prefix player-position: <http://data.basketontology.org/cube/resource/
    codelist/player-position/>
prefix injury-status: <http://data.basketontology.org/cube/resource/codelist/
    injury-status/>
prefix active-status: <http://data.basketontology.org/cube/resource/codelist/
    active-status/>
```

Listing 1: prefixes and namespaces used in SPARQL queries examples

References

References

1. Lenzerini M. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. pp. 233–246.
2. Doan A, Halevy A and Ives Z. *Principles of Data Integration*. Elsevier, 2012.
3. Sabou M, Ekaputra FJ and Biffl S. Semantic web technologies for data integration in multi-disciplinary engineering. In Biffl S, Lüder A and Gerhard D (eds.) *Multi-Disciplinary Engineering for Cyber-Physical Production Systems*. Berlin, Germany: Springer International Publishing, 2017. pp. 301–329.

4. Cheatham M and Pesquita C. Semantic data integration. In *Handbook of Big Data Technologies*. Springer, 2017. pp. 263–305.
5. Wache H, Voegele T, Visser U et al. Ontology-based integration of information–a survey of existing approaches. In *OIS@IJCAI*.
6. Euzenat J and Shvaiko P. *Ontology Matching*, volume 18. Springer, 2007.
7. Jiménez-Ruiz E, Cuenca Grau B, Zhou Y et al. Large-scale interactive ontology matching: Algorithms and implementation. In *ECAI 2012*. IOS Press, pp. 444–449.
8. Ivanova V, Bach B, Pietriga E et al. Alignment cubes: Towards interactive visual exploration and evaluation of multiple ontology alignments. In *International Semantic Web Conference*. Springer, pp. 400–417.
9. Sicilia Á, Nemirovski G and Nolle A. Map-on: A web-based editor for visual ontology mapping. *Semantic Web* 2017; 8(6): 969–980.
10. Cyganiak R, Reynolds D and Tennison J. The rdf data cube vocabulary. `https://www.w3.org/TR/vocab-data-cube/`, 2014. W3C Recommendation.
11. Rahm E and Bernstein PA. A survey of approaches to automatic schema matching. *VLDB Journal* 2001; 10(4): 334–350.
12. Shvaiko P and Euzenat J. Ontology matching: state of the art and future challenges. *IEEE Transactions on Knowledge and Data Engineering* 2011; 25(1): 158–176.
13. Wilkinson MD, Dumontier M, Aalbersberg IJ et al. The fair guiding principles for scientific data management and stewardship. *Scientific Data* 2016; 3(1): 1–9.
14. Karampatakis S, Bratsas C, Zamazal O et al. Alignment: a hybrid, interactive and collaborative ontology and entity matching service. *Information* 2018; 9(11): 281.
15. Bratsas C, Filippidis PM, Karampatakis S et al. Developing a scientific knowledge graph through conceptual linking of academic classifications. In *2018 13th International Workshop on Semantic and Social Media Adaptation and Personalization (SMAP)*. IEEE, pp. 113–118.
16. Filippidis PM, Dimoulas C, Bratsas C et al. A unified semantic sports concepts classification as a key device for multidimensional sports analysis. In *2018 13th International Workshop on Semantic and Social Media Adaptation and Personalization (SMAP)*. IEEE, pp. 107–112.
17. Filippidis PM, Karampatakis S, Koupidis K et al. The code lists case: Identifying and linking the key parts of fiscal datasets. In *2016 11th International Workshop on Semantic and Social Media Adaptation and Personalization (SMAP)*. IEEE, pp. 165–170.
18. Knoblock CA, Szekely P, Ambite JL et al. Semi-automatically mapping structured sources into the semantic web. In *Extended Semantic Web Conference*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 375–390.
19. Bizer C and Seaborne A. D2rq-treating non-rdf databases as virtual rdf graphs. In *Proceedings of the 3rd International Semantic Web Conference (ISWC2004)*. Hiroshima: Springer.
20. Priyatna F, Corcho O and Sequeda J. Formalisation and experiences of r2rml-based sparql to sql query translation using morph. In *Proceedings of the 23rd International Conference on World Wide Web*. pp. 479–490.
21. Dimou A, De Nies T, Verborgh R et al. Automated metadata generation for linked data generation and publishing workflows. In *LDOW2016*. CEUR-WS.org, pp. 1–10.
22. Lefrançois M, Zimmermann A and Bakerally N. A sparql extension for generating rdf from heterogeneous formats. In *European Semantic Web Conference*. Cham: Springer International Publishing, pp. 35–50.
23. Cyganiak R. Tarql (sparql for tables): Turn csv into rdf using sparql syntax. Technical Report, Available at: http://tarql.github.io, 2015.

24. Vassiliadis P. Modeling multidimensional databases, cubes and cube operations. In *Proceedings. Tenth International Conference on Scientific and Statistical Database Management*. IEEE, pp. 53–62.
25. Oueslati W and Akaichi J. A survey on data warehouse evolution. *International Journal of Database Management Systems (IJDMS)* 2010; 2(4): 11–24.
26. Cuzzocrea A, Song IY and Davis KC. Analytics over large-scale multidimensional data: the big data revolution! In *Proceedings of the ACM 14th International Workshop on Data Warehousing and OLAP*. pp. 101–104.
27. Ioannidis L, Bratsas C, Karabatakis S et al. Rudolf: An http api for exposing semantically represented fiscal olap cubes. In *2016 11th International Workshop on Semantic and Social Media Adaptation and Personalization (SMAP)*. IEEE, pp. 177–182.
28. Etcheverry L and Vaisman AA. Qb4olap: a new vocabulary for olap cubes on the semantic web. In *Proceedings of the Third International Conference on Consuming Linked Data (COLD)*, volume 905. pp. 27–38.
29. Martin M, Abicht K, Stadler C et al. Cubeviz: Exploration and visualization of statistical linked data. In *Proceedings of the 24th International Conference on World Wide Web*. pp. 219–222.
30. Kalampokis E, Nikolov A, Haase P et al. Exploiting linked data cubes with opencube toolkit. In *ISWC (Posters & Demos)*. pp. 137–140.
31. Brizhinev D, Toyer S, Taylor K et al. Publishing and using earth observation data with the rdf data cube and the discrete global grid system. W3C Working Group Note and OGC Discussion Paper, 2017. W3C, 20170928, 16–125.
32. Compton M, Barnaghi P, Bermudez L et al. The ssn ontology of the w3c semantic sensor network incubator group. *Journal of Web Semantics* 2012; 17: 25–32.
33. Babaei Giglou H, D'Souza J, Engel F et al. Llms4om: Matching ontologies with large language models. *arXiv preprint arXiv:240410317* 2024; URL `https://arxiv.org/abs/2404.10317`.
34. Qiang Z, Wang W and Taylor K. Agent-om: Leveraging llm agents for ontology matching. *arXiv preprint arXiv:231200326* 2023; URL `https://arxiv.org/abs/2312.00326`.
35. He Y, Chen J, Dong H et al. Exploring large language models for ontology alignment. *arXiv preprint arXiv:230907172* 2023; URL `https://arxiv.org/abs/2309.07172`.
36. Kommineni VK, König-Ries B and Samuel S. From human experts to machines: An llm supported approach to ontology and knowledge graph construction. *arXiv preprint arXiv:240308345* 2024; URL `https://arxiv.org/abs/2403.08345`.
37. Huang W, Liu J, Li T et al. Fedcke: Cross-domain knowledge graph embedding in federated learning. *IEEE Transactions on Big Data* 2022; 9(3): 792–804. DOI:10.1109/TBDATA.2022.3144603.
38. Hu Q, Jiang W, Li H et al. Fedcqa: Answering complex queries on multi-source knowledge graphs via federated learning. *CoRR* 2024; abs/2402.14609. URL `https://arxiv.org/abs/2402.14609`.
39. Huang W, Chen J, Wang D et al. Fedmdkge: Multi-granularity dynamic knowledge graph embedding in federated learning. *International Journal of Computational Intelligence Systems* 2025; 18(1): 131. DOI: 10.1007/s44196-025-00878-5.
40. W3C. Qb4st: Rdf data cube extensions for spatio-temporal components. `https://www.w3.org/TR/qb4st/`, 2021. Accessed: 2025-09-08.
41. STAC Community. Stac datacube extension. `https://github.com/stac-extensions/datacube`, 2023. Accessed: 2025-09-08.
42. Filippidis PM, Bratsas C, Veglis A et al. Creating and using sports linked data: Applications and analytics, 2015. Unpublished technical report.

43. de Vleeschauwer E et al. Rmlstreamer with reference conditions in the kgcw 2023 challenge. In *Proceedings of the Knowledge Graph Construction Challenge (KGCW)*, *CEUR Workshop Proceedings*, volume 3471.
44. Iglesias E, Chaves-Fraga D, Toledo J et al. Sdm-rdfizer: An rml interpreter for the efficient creation of rdf knowledge graphs. *arXiv preprint arXiv:200807176* 2020; .
45. Chaves-Fraga D, Iglesias E, Toledo J et al. What are the parameters that affect the construction of a knowledge graph? In *Proceedings of the 18th International Semantic Web Conference (ISWC)*.
46. Arenas-Guerrero J, Toledo J, Chaves-Fraga D et al. Morph-kgc: Scalable knowledge graph materialization with mapping partitions. *Semantic Web Journal* 2024; .
47. Schmidt M, Meier M and Lausen G. Foundations of sparql query optimization. In *Proceedings of the 13th International Conference on Database Theory*. pp. 4–33.
48. Le W, Kementsietsidis A, Duan S et al. Scalable multi-query optimization for sparql. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, pp. 666–677.
49. Khan Y, Saleem M, Mehdi M et al. Safe: Sparql federation over rdf data cubes with access control. *Journal of Biomedical Semantics* 2017; 8(1): 5.