ToposKG Manual

Kostas Plas
Sergios - Anestis Kefalidis
Manolis Koubarakis

Contents

1 Introduction 3
1.1 What is Topos and ToposKG? 3
1.2 What is this document? 3
1.3 Quick Start 3

2 ToposKG: A Modular Geospatial Knowledge Graph 4
2.1 Data Sources 5
2.1.1 The Global Administrative Unit Layers (GAUL) 5
2.1.2 OpenStreetMap 7
2.1.3 Marine Regions 9
2.1.4 GMBA Mountain Inventory 9
2.2 Ontology 11
2.2.1 Administrative Unit Representations 12
2.2.2 GeoSPARQL alignment and foundational layer 14
2.2.3 OSM attributes and alignment 15
2.2.4 Top-level thematic organization 16
2.2.5 Administrative hierarchy modeling 17
2.2.6 Natural feature taxonomy 18
2.2.7 External linking and shared semantic attributes 18
2.2.8 General and POI extensibility branch 19
2.2.9 Design principles embodied in the ontology 19
2.3 Class-level property overview 19
2.4 Querying ToposKG 23
2.4.1 Introductory Queries 23
2.4.2 Administrative governance and territorial analysis 30
2.4.3 Environmental and physical-geography analysis 31
2.4.4 3. Urban services, accessibility, tourism, and everyday POI discovery 33

3 The Topos Website and GUI 35
3.1 Website 35
3.1.1 Capabilities 35
3.1.2 Data Sources 36
3.1.3 Tooling 37
3.2 GUI for the Topos tool-chain 37

4 The `toposkg-lib` Toolchain 40
4.1 GeoKG blueprint builder 40
4.2 Materialization 43
4.3 Translation 44
4.4 From Raw Data to RDF Graphs 45
4.4.1 Parsing GeoJSON Naively 45
4.4.2 Parsing GeoJSON Files using RML Mappings 48

5 Building GeoKGs for GIS Applications using ToposKG and the Topos Framework 52
5.1 Building the GeoKG 52
5.2 Introducing thematic information 53
5.3 Interlinking the GeoKG with thematic information 57
5.4 Analysis Table for TerraQ and PnyQA usecases 64

6 Conclusion **65**

7 Acknowledgments **65**

1 Introduction

1.1 What is Topos and ToposKG?

The Topos framework and, by extension, the geospatial knowledge graph built upon it, ToposKG, represent our team's effort to develop a modular, geospatially enriched knowledge graph. It is supported by a suite of tools and functionalities designed to make it easier for end users to explore, use, and extend the graph with thematic or additional geospatial information.

1.2 What is this document?

This white paper presents the core functionalities of the Topos site and tooling, serving as an introduction to how users can work with, utilize, experiment with, and build their own knowledge graphs based on ToposKG. The contents of this document are the following:

- An in-depth exploration of the ToposKG ontology, design philosophy, how to extend your own copies of the modularized KG and a number of SPARQL and GeoSPARQL queries to start working with ToposKG data.
- An explanation of the Topos website and how a novice user can interact with it.
- A number of tutorials targeting every functionality provided by the topos-lib tool-chain
- An end-to-end tutorial on how to build a GIS application utilizing ToposKG, topos-lib and custom data provided by the user, based on our geospatial questions answering system, PnyQA

Users can refer to this document both as a manual for the Topos framework and a reference for the more technical parts of the design of our tool-chains and ontology, which did not make it to the original paper.

1.3 Quick Start

- Users interested specifically in the ToposKG ontology can refer directly to Section 2.2
- Users interested specifically in querying ToposKG can refer directly to Section 2.4
- Users looking for links to our resources can find them here: GitHub,Zenodo,Website, Ontology Website
- Tutorials for the ToposKG Python library begin in Section 4

2 ToposKG: A Modular Geospatial Knowledge Graph

In this section we will be examining the backbone of the Topos framework, the geospatial knowledge graph ToposKG. Our KG is built on a variety of geospatial data sources combined and presented to the user in a way that facilitates a modular knowledge base design based on the various geospatial information provided by ToposKG, combined with additional thematic or spatial information the user requires.

The core design principle of the Topos framework is modularity. Therefore, when collecting and parsing our various data sources, we decided to maintain a certain level of granularity based on the type of spatial resource. By organizing geospatial information into three main classes (`AdministrativeUnit`, `Natural` and `General`), users can seamlessly integrate their own data by extending one of these core classes. To enhance the customizability of GeoKGs constructed using the Topos framework, we also propose a more modularized approach to structuring incoming data with granularity in mind. Sources that can be encapsulated by the `AdministrativeUnit` classes, should be separated by country and by administrative division, enabling developers to work with only the subsets of ToposKG relevant to their specific GIS applications. Similarly, sources populating classes like `Forests`, `WaterBody`, and `POI` can categorized by country. Data sources that cannot be naturally divided into subsets can remain unified. For example, in constructing ToposKG, we integrated `Mountains` as a single resource, since dividing them by country would introduce significant redundancy due to their cross-border nature. This framework enables the creation of a repository of GeoKGs, that allows the selection of data sources as needed—without introducing unnecessary overhead or redundancy in specific use cases. Finally, we further expand on the modularity and extensibility of the ToposKG framework by introducing a tool chain for creating custom GeoKGs from its core sources.

2.1 Data Sources

This section will focus on the various data sources that were used to create ToposKG knowledge graph. The section will be separated in 3 cateragories: Administrative Data, Natural Data and Thematic Spatial(or General) Data.

2.1.1 The Global Administrative Unit Layers (GAUL)

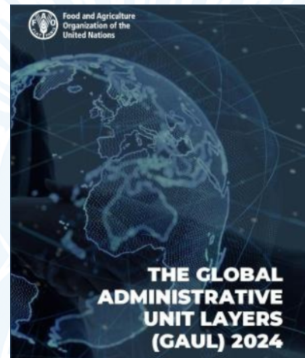


Figure 1: GAUL Logo

The Global Administrative Unit Layers (GAUL) dataset is developed and owned by the Food and Agriculture Organization of the United Nations (FAO). It provides a comprehensive spatial dataset of sub-national administrative units for all countries in the world, consistent with the United Nations (UN) delineation of international boundaries.

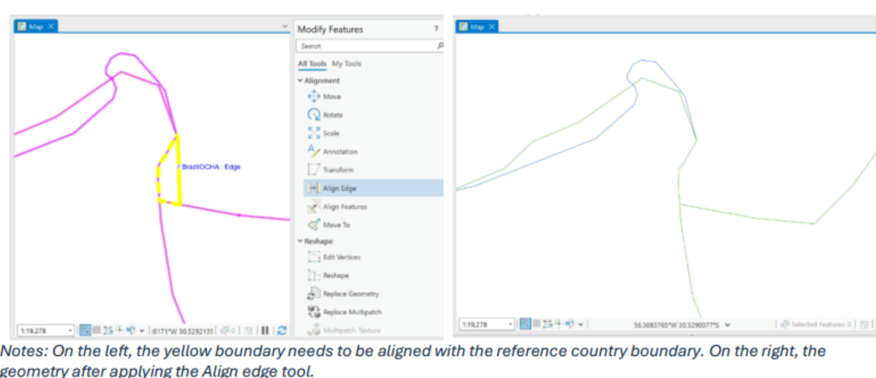
GAUL 2024 includes two datasets: GAUL 2024 Subnational Level 1 (GAUL_2024_L1) and GAUL 2024 Subnational Level 2 (GAUL_2024_L2). The dataset was developed by consolidating national-level data from relevant sources, which involved editing geometries, standardizing attributes into a unified table structure, and preparing metadata for each country.

GAUL 2024 was compiled from multiple sources in order of priority:

- Second Administrative Level Boundaries (SALB) geoportal - authoritative subnational boundary information managed by the UN Geographic Information Section
- Humanitarian Data Exchange (HDX) platform - hosted by the Office for the Coordination of Humanitarian Affairs (OCHA)
- National Providers - relevant national authorities identified through online research or FAO national representations
- GAUL 2015 - used when no other data sources were available

Editing Process The editing process aligned national data with an international reference dataset of country boundaries. For GAUL 2024, the UN 2018 geospatial dataset served as the primary reference. The dataset was initially downloaded as shapefiles from the United Nations Geospatial Data portal at `https://geoportal.un.org`. Although more recent versions of the UN dataset are now available, the UN 2018 version was used during the project's early stages and minimal differences are observed in newer releases, primarily along coastlines. Adjustments to international

borders to match the reference system were performed manually using the Align Edge tool in ArcGIS Pro.



Notes: On the left, the yellow boundary needs to be aligned with the reference country boundary. On the right, the geometry after applying the Align edge tool.

Figure 2: Example of align edge

The editing process was carried out in small regions to minimize the risk of unexpected vertex distortions in neighboring geometries. For international boundaries that fell short of the reference dataset, geometries were extended to meet the international boundary line. Conversely, when the national dataset exceeded the reference boundaries, the Clip function was applied to align them. Notably, the availability of "extended" CODs from the Fieldmaps application (`https://fieldmaps.io/data/cod`) proved valuable in this process.

After completing the full editing workflow, an initial topological validation was performed on the national dataset based on the following rules:

- Must not have gaps: Ensures there are no voids between or within polygons.
- Must not overlap: Verifies that polygons do not overlap one another.

Additional editing steps were applied as required by the characteristics of the original datasets. These included filling gaps, reprojecting data to the WGS84 coordinate system, and dissolving single polygons into multi-polygons. All the edits have been done at subnational level 2 and final results dissolved at level 1. Details of editing of coastline and waterbodies are discussed further in this document. To minimize the complexity of geometries and reduce file size, many countries applied a vertex reduction mechanism. In ArcGIS Pro, we utilized the Simplify Polygon tool with the Retain Critical Bends (Wang-Müller) algorithm to preserve significant bends. The simplification tolerance was set to 0.01 decimal degrees, while the Minimum Area parameter was left as Unknown. Additionally, the Resolve Topological Errors option was enabled to prevent overlaps or gaps. Optionally, country boundaries were used as a barrier layer to ensure that internationa borders remained unchanged. Finally, the table of attribute was consolidated into a standard set of attributes and each country aggregated into a final global dataset. Once again, a topological check was executed with same topology rules to confirm that no mistakes were introduced during the merging process. Searching for new available data sources stopped in November 2024, meaning any changes in international boundaries occurring after this date are not reflected in the GAUL 2024 release.

Aspect	GAUL administrative regions
Reference system	WGS 84 geographic coordinates, i.e., EPSG:4326.
Projection	No native projected CRS is specified; data are distributed in latitude/longitude coordinates.
Spatial resolution / scale	Vector dataset; older metadata reports an equivalent scale of approximately 1:1,000,000.
Spatial error / accuracy	No single numeric positional error is provided in the public metadata. GAUL documents source harmonization and geometry/topology validation.
Contested regions	Disputed or unsettled areas are explicitly represented using attributes such as `DISP_AREA` and status information.
Boundary change	Administrative changes are handled through temporal/versioned layers and attributes indicating creation and expiry years of administrative units.
Update handling	Historically released annually; GAUL 2024 is a recent revived release compiled from 2022–2024 sources.

Table 1: Summary of key characteristics of the GAUL administrative regions dataset.

How GAUL Handles Contested Boundaries

- Unsettled Territory Flag: Boundaries in the dataset contain specific attributes, including DIS_AREA (which is flagged as 'Yes' for unsettled territories) and STATUS to indicate the nature of the region.
- Overlapping Geometries: Unlike standardized borders that clip regions to prevent overlaps, GAUL maps the administrative regions of disputing nations in their entirety. This allows the user to see the respective territorial claims of all involved countries.
- UN Cartographic Unit Baseline: The dataset's country and regional coastlines are primarily drawn from the UN Cartographic Section, but are occasionally updated with satellite imagery for accuracy.

Metadata for the GAUL dataset are present in table 1. The GAUL dataset, since it was curated and carefully edited to not contain inconsistent topological data, was used as the administrative unit based for our geospatial knowledge graph.

2.1.2 OpenStreetMap

Figure 3: OpenStreetMap

OpenStreetMap (OSM) is a collaborative, open geospatial database that contains user-contributed information about real-world geographic features, including roads, buildings, land use, water bodies, points of interest, administrative boundaries, transport infrastructure, and many other feature types.

Aspect	OpenStreetMap administrative data
Reference system	OSM stores geographic coordinates in WGS 84, i.e., EPSG:4326.
Projection	No native projected CRS is used for raw OSM data. Web maps may render OSM using projected systems such as Web Mercator, but the source data are longitude/latitude coordinates.
Spatial resolution / scale	Vector dataset with no fixed global spatial resolution or cartographic scale. Detail depends on contributor activity, source imagery, imports, and local mapping practices.
Spatial error / accuracy	No single numeric positional error is provided. Accuracy and completeness vary by region and feature type, and OSM relies on community validation and quality-assurance tools.
Contested regions	Disputed areas are handled through community tagging and mapping conventions, e.g., disputed-boundary tags or alternative boundary relations. Representation may vary because OSM is not an official political authority.
Boundary change	Changes are applied continuously by contributors and recorded as changesets and object histories. Historical/full-history dumps can be used to reconstruct earlier versions.
Update handling	OSM is a live database rather than a periodic static product. Planet files provide regular snapshots of the current database, while full-history data preserve past object versions.

Table 2: Summary of key characteristics of OpenStreetMap administrative data.

Unlike centrally curated datasets such as GAUL, OSM is maintained by a global community of contributors who add, update, and correct features continuously.

OSM data are represented mainly through three primitive elements: nodes, ways, and relations. Nodes represent individual geographic points, ways represent linear or polygonal geometries, and relations group multiple elements together to model more complex objects, such as administrative boundaries, routes, or multipolygons. Semantics are expressed through flexible key-value tags, for example amenity=school, natural=water, boundary=administrative, or admin_level=4.

For administrative regions, OSM typically represents boundaries using relations tagged with boundary=administrative and an admin_level value. The meaning of admin_level is country-dependent, since different countries organize their administrative hierarchies differently. As a result, OSM administrative data are rich and frequently updated, but they may require normalization before being compared or integrated with more standardized datasets such as GAUL or GADM.

OSM does not provide a single guaranteed spatial resolution or positional accuracy. Accuracy varies by region, contributor activity, source material, and feature type. In highly mapped urban areas, the data can be very detailed, while in sparsely mapped regions it may be incomplete or less precise. Boundary changes and contested regions are handled through community edits and tagging practices, rather than through a fixed official temporal-versioning model. Therefore, OSM is highly useful for up-to-date and detailed geospatial information, but its heterogeneous structure and varying data quality require careful validation and alignment when used in knowledge graph construction.

For the purposes of ToposKG, we used OSM as additional information on administrative data when GAUL did not have available information. That is, for administrative levels greater than 3, or for administrative information that was incorrect in the GAUL dataset. Additionally, we used OSM to collect information about water bodies, points-of-interest and finally forests and wooded areas.

For water bodies and points-of-interest we collected a detailed number of classes to extend our ontology and knowledge graph with meaningful natural resource and thematic spatial information. Metadata for OSM is presented in Table 2.

2.1.3 Marine Regions

Figure 4: Marine Regions: Global Oceans and Seas

Marine Regions is an integration of the VLIMAR Gazetteer and the VLIZ Maritime Boundaries Geodatabase. The VLIMAR Gazetteer is a database with geographic, mainly marine names such as seas, sandbanks, seamounts, ridges, bays or even standard sampling stations used in marine research. The geographic cover of the VLIMAR gazetteer is global but initially focused on the Belgian Continental Shelf and the Scheldt Estuary and the Southern Bight of the North Sea. Gradually more regional and global geographic information was added to VLIMAR and combining this information with the Maritime Boundaries database, representing the Exclusive Economic Zone (EEZ) of the world, led to the creation of `marineregions.org`.

For the purposes of ToposKG, we extracted and used the *Global Oceans and Seas*. This dataset represents the boundaries between the 10 main oceans and seas (Arctic Ocean, North and South Atlantic Ocean, North and South Pacific Ocean, Southern Ocean, Indian Ocean, Baltic Sea, Mediterranean Region, South China and Eastern Archipelagic Seas). The boundaries are largely based on the publication 'Limits of Oceans and Seas, Special Publication No. 23', published by the IHO in 1953. The dataset is available in World Geodetic System of 1984 (WGS84). This dataset was composed by the Flanders Marine Data Centre.

NB: The Southern Ocean is not included in the IHO publication and its limits are subject of discussion among the scientific community. The Flanders Marine Institute acknowledges the controversy around this subject but decided to include the Southern Ocean in the dataset as this term is often used by scientists working in this area.

The need for spatial information about seas and oceans, came from our use cases in the TerraQ project of the European Space Agency. TerraQ, needed spatial information on seas and sea segments to monitor oil spills in the Mediterranean sea.

The metadata for Marine Regions: Global Seas and Oceans dataset is presented in Table 3.

2.1.4 GMBA Mountain Inventory

Figure 5: Marine Regions:Global Oceans and Seas

The GMBA Mountain Inventory v2.0 is a hierarchically organized spatial vector layer in ESRI shape format. The polygons represent several thousand mountain ranges across various scales (see

Aspect	Marine Regions dataset
Reference system	Marine Regions maritime-boundary products are available in WGS84 / EPSG:4326.
Projection / supported formats	The data are distributed as GIS vector products rather than in a single projected CRS. Available formats include GeoPackage, Shapefile, KML, and 0–360 degree variants for some products. The Marine Regions download page lists EEZ products in GeoPackage, Shapefile, KML, low-resolution, and 0–360 degree versions.
Spatial resolution / geometric detail	No single fixed global spatial resolution is specified. Boundary detail depends on the source and construction method. For median-line construction, baselines are densified so that the maximum distance between baseline points is reduced to 0.01 degrees..
Spatial error / accuracy	No single numeric positional error is documented. The dataset is intended for scientific, educational, and research use, not for legal, navigational, or economic-resource-exploration purposes.
Boundary change	Boundary changes are handled through versioned releases, change histories, corrections, and updates based on new treaties, expert input, and user feedback.

Table 3: Summary of key characteristics of the Marine Regions maritime-boundary dataset.

numbers below). Depending on the region, these polygons range from very small (e.g. High Tatras) to very large (e.g. Carpathians) across up to 10 levels, whereas in other regions, the depth of the inventory is more limited (e.g. South Siberian Mountains with 4 levels).

The hierarchical structure of the inventory allows the user to represent mountain ranges at different levels of detail, either at a coarse scale (e.g. European Alps) or at a refined scale (e.g. 134 polygons representing the smallest so-called basic subranges of the European alps). This flexibility in the use and application means that the default layer is complex as it is composed of all overlapping polygons (from large to small).

The construction of the mountain range polygons is the result of a semi-automated approach, where rivers form the borders between contiguous mountain ranges and the outer borders of the mountain systems correspond to a new, more generalized mountain definition. Details on the polygon delineation and additional information are available in

Version 2.0 introduces a hierarchical classification of the mountain ranges of the world, which allows for the partitioning of mountain systems into smaller ranges and subranges and enables spatially explicit and comparative mountain research across scales. This hierarchical structure of the GMBA dataset, was a good fit to our Topos ontology. When introducing the data to our ontology, we maintained the hierarchical levels, with a self-referential object property. Mountain ranges and mountain data provided by GMBA, were used extensively in our PnyQA system, which provided very useful statistical information on regions like the Appalachian Ranges and the voting habits of the people living in and around them. Metadata for GMBA can be found in Table **??**

Aspect	GMBA Mountain Inventory
Reference system	In WGS 84, originally in EPSG:3857 transformed to EPSG:4326 for compatibility.
Projection / supported formats	The inventory is distributed as spatial vector data in ESRI Shapefile format. During processing, the authors used Lambert Azimuthal Equal Area for high-latitude DEM zones and WGS 1984 Cylindrical Equal Area for the equatorial zone to reduce latitudinal distortion.
Spatial resolution / geometric detail	The inventory is a vector polygon dataset, so it does not have a single raster-like spatial resolution. Its geometric detail is hierarchical: ranges are represented from coarse mountain systems to smaller subranges, with up to 10 hierarchy levels. The associated GMBA Definition v2.0 raster has a resolution of 7.5 arc seconds.
Spatial error / accuracy	No single numeric positional error is documented. The authors describe some boundaries as approximate, because mountain-range extents can be fuzzy and difficult to identify, especially in old, eroded, or complex mountain systems. The dataset is not presented as an official gazetteer or replacement for national or regional mountain classifications.
Boundary definition / extent	The standard version uses outer borders corresponding to GMBA Definition v2.0. The broad version has larger outer borders based on the union of three mountain definitions plus a 5 km buffer, and should be intersected with a chosen mountain definition before use.

Table 4: Summary of key characteristics of the GMBA Mountain Inventory.

2.2 Ontology

A tree-like presentation of the ontology is available here. The ontology of ToposKG as described in the paper, a simplified version of which is shown in Figure 6, is designed with modularity and extensibility in mind. It is based on the OGC GeoSPARQL standard, which serves as the foundation for our ontology. The top-level class, `ToposKGFeature`, extends the `Feature` class defined in the GeoSPARQL ontology. All classes introduced in our knowledge graph are modeled as subclasses of `ToposKGFeature`. Administrative divisions are represented by the `AdministrativeUnit` class, which is further divided into seven subclasses corresponding to different levels of administrative hierarchy (levels 0 through 6). Natural resources are represented by the `Natural` class, which is extended by four subclasses: `Sea`, `Forest`, `Mountain`, and `WaterBody`. The `WaterBody` class is further refined into subclasses such as `Lagoon`, `Lake`, `River`, `Canal`, etc., based on water body types identified in OSM. POIs are represented by the `POI` class, which extends the more general `General` class. Although the General class is not currently extended further in this iteration of the KG, it was included to facilitate the integration of geospatial information that does not fit neatly into the `Natural` or `AdministrativeUnit` categories. This design supports our principles of modifiability and reusability. The data properties depicted in Figure 6 capture key attributes extracted from various data sources, including OSM and Wikidata (when references to wikidata were available in OSM entities). These properties include metadata such as addresses, official names, population statistics, and more.

Following the definition of our ontology, we will go into further detail for each subpart of the ontology, which data sources populate the various subclasses and additional information that was not presented in ToposKG's paper due to space limitations.

Rather than treating the ontology as a flat taxonomy of geographic objects, ToposKG adopts a layered semantic architecture in which interoperability, domain specialization, and extensibility are separated into different modeling levels. At the uppermost layer, the ontology inherits foundational geospatial semantics from GeoSPARQL. A middle layer introduces the ToposKG domain abstraction

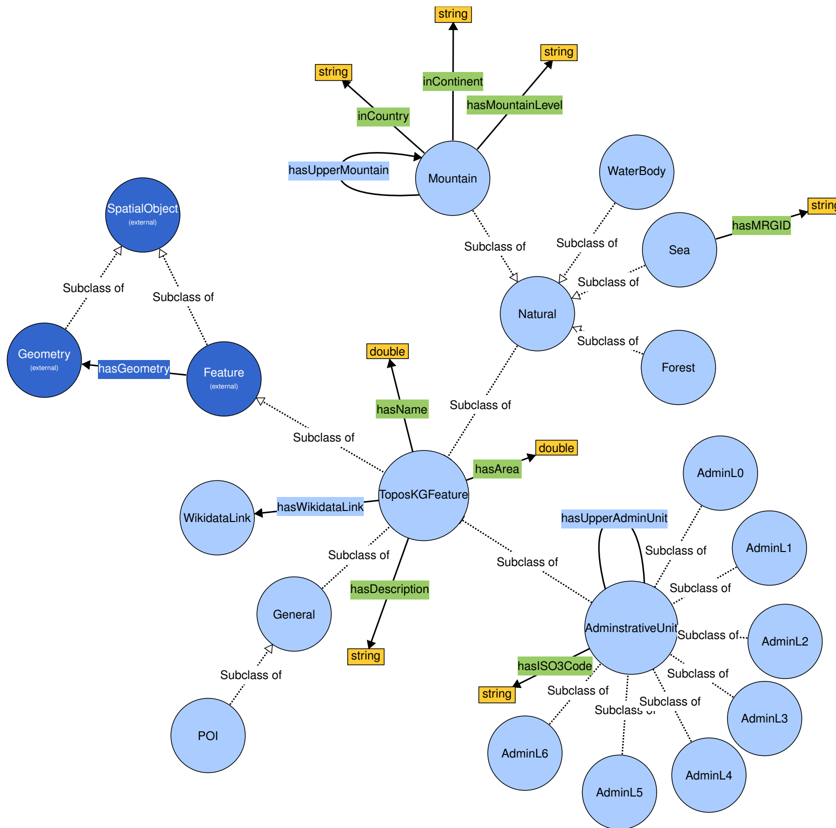


Figure 6: A simplified visual representation of the ToposKG ontology

through `ToposKGFeature`. A lower domain layer then specializes this abstraction into thematic branches corresponding to administrative, physical, and general-purpose geospatial entities. This structure supports both semantic reasoning and controlled expansion of the ontology as new feature categories are introduced.

2.2.1 Administrative Unit Representations

The basis of our architecture can be seen on older ontologies and geospatial knowledge graphs that model administrative unit information. Mainly, we built upon previous knowledge graph ontologies like GAG, GADM and YAGO2geo, the latter providing a combined ontology of GAG and GADM to represent the spatial information.

GAG Ontology The Greek Administrative Geography (GAG) ontology models the administrative hierarchy of Greece. It represents Greek administrative units such as Country, Decentralized Administration, Region, Regional Unit, Municipality, Municipality Unit, Municipality Community, and Local Community. A recent evaluation lists it as a relatively small ontology, with 9 classes and 2,914 individuals.

Its core modelling idea is hierarchical containment: lower-level administrative units belong to, or are contained in, higher-level administrative units. For example, a Municipality belongs to a Regional Unit, a Regional Unit belongs to a Region, and so on. The evaluated version of GAG does

not import GeoSPARQL; instead, it defines its own relationship for the containment hierarchy. It also includes a more general semantic association relation between regions, but this relation is not clearly a spatial/topological relation. The GAG ontology is presented in figure 7

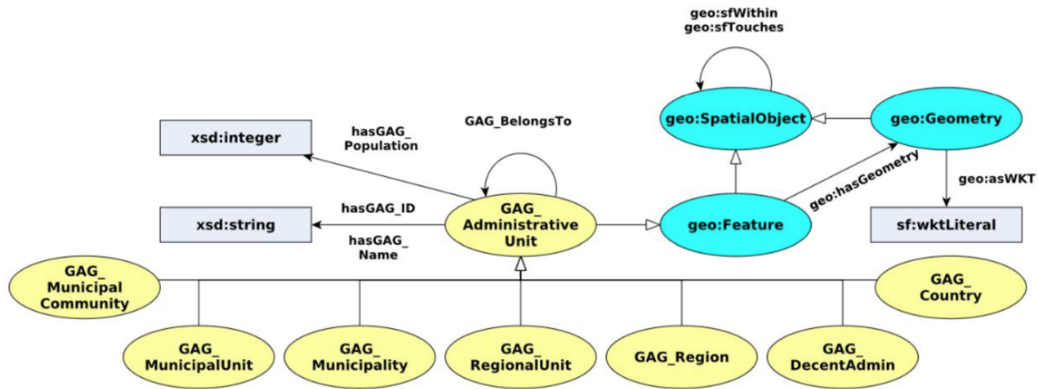


Figure 7: YAGO2geo: GAG ontology

So, GAG is useful as a domain-specific administrative ontology for Greece, especially for answering questions about the official administrative structure. Its main weakness is that its spatial semantics are limited: it can answer containment-style competency questions, but not proximity/topological questions such as adjacency, overlap, touching, or distance-based relations unless those are added externally or inferred from geometries.

GADM Ontology GADM itself is primarily a global administrative boundary dataset, not originally an ontology. It provides spatial data for countries and their subdivisions worldwide. The dataset schema is level-based: level 0 is the country, and lower administrative levels are represented with fields such as GID_i, NAME_i, TYPE_i, ENGTYPE_i, VALIDFR_i, and VALIDTO_i. The GID encodes the hierarchy by concatenating country and subdivision identifiers, e.g., AFG.1, AFG.1.1, etc.

The GADM ontology used in linked geospatial data work is a constructed RDF/OWL ontology for representing GADM administrative units. In GeoQA/App-Lab related work, it was created by converting GADM shapefiles into RDF using GeoTriples, with resources under a GADM namespace and ontology terms under a GADM ontology namespace. In later Copernicus/App Lab work, the GADM ontology is described as extending GeoSPARQL, using geo and sf namespaces for spatial modelling, while introducing GADM-specific classes and properties with a gadm prefix. It can be used either to materialize GADM data as RDF or to query GADM data virtually through OBDA/mappings.

Conceptually, the GADM ontology usually centres on a generic class such as AdministrativeUnit, with properties for names, geometries, and administrative hierarchy. Example queries over the GADM RDF representation use gadm:hasName, geo:hasGeometry, geo:asWKT, and hierarchy properties such as gadm:belongsToAdm2. The GADM ontology is presented in figure 8

The idea behind our main ontological structure can be observed in the two previous ontologies. We provided a "recursive" object property for geospatial administrative units that can be extended to more levels if necessary and covers the nesting inherent in almost all countries and administrative divisions with great accuracy. This administrative division was the basis of our Topos ontology. This

ontology design, inspired by GADM's administrative hierarchy, has been widely useful in geospatial applications that require structured territorial information. Such applications include answering questions about administrative containment, evaluating how effectively ontologies model administrative hierarchies, and using administrative units as a backbone for linking thematic information to standardized territorial entities. Further extensions that concern geospatial resources other than the administrative divisions are explored in the next subsections.

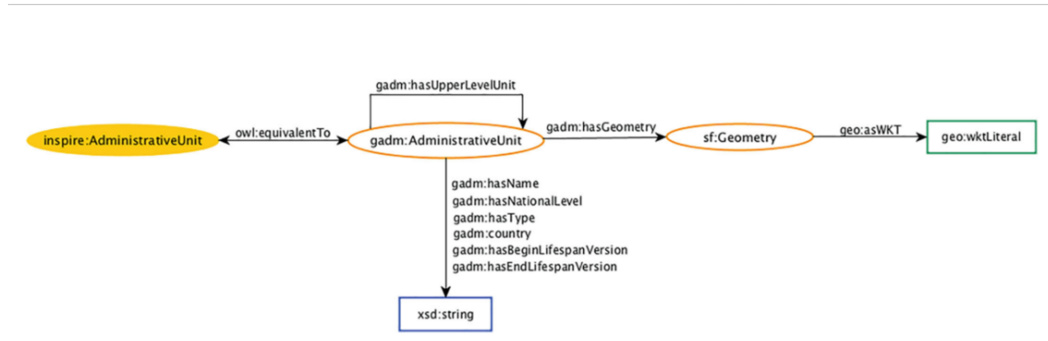


Figure 8: GADM Ontology

2.2.2 GeoSPARQL alignment and foundational layer

The foundational layer establishes ToposKG as an extension rather than an isolated vocabulary. The ontology anchors itself in `geo:SpatialObject`, distinguishes feature identity from geometric representation through `geo:Feature` and `geo:Geometry`, and preserves compatibility with geospatial RDF tooling through reuse of GeoSPARQL object properties.

This alignment serves several modeling goals. First, it separates what an entity *is* from how it is spatially represented. A municipality, mountain, or river can exist as a semantic object independent of whether it is described by a polygon, line, or point geometry. Second, it enables direct interoperability with external linked geospatial datasets using a standard vocabulary. Third, it constrains the ontology to established geospatial semantics, preventing domain-specific modeling choices from drifting away from reusable standards. Figure 9 illustrates this foundational semantic layer.

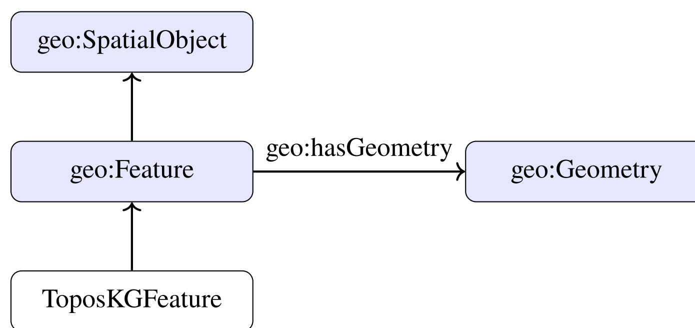


Figure 9: GeoSPARQL-based root of the ontology

2.2.3 OSM attributes and alignment

ToposKG separates its compact conceptual core from a larger source-derived attribute layer. The conceptual core contains the main geospatial categories of the graph, such as `ToposKGFeature`, `AdministrativeUnit`, `Natural`, `Waterbody`, `Forest`, `Sea`, `Mountain`, `General`, and `POI`. In contrast, the OpenStreetMap (OSM) layer preserves source-specific descriptive attributes encountered during data extraction. These attributes are represented as datatype properties rather than as new ontology classes, in order to keep the model lightweight and compatible with the original ToposKG data.

Each OSM-derived predicate is aligned with the original OSM tagging scheme through explicit annotation metadata. More specifically, ToposKG uses `topos:osmKey` to record the original OSM key, `topos:osmValue` when the predicate corresponds to a fixed key–value pair, and `topos:osmTag` to record the complete OSM tag. For example, `topos:hasAddrCity` is aligned with the OSM key `addr:city`, whereas `topos:hasAmenityCafe` is aligned with the fixed OSM tag `amenity=cafe`. The predicate remains a ToposKG datatype property, but its source semantics are made explicit through the alignment metadata and links to the corresponding OSM documentation.

OSM-derived attributes. The OSM-derived attributes are attached to the ToposKG classes whose instances are populated from OSM data. In the current ontology, OSM-derived predicates are mainly associated with four classes:

`AdministrativeUnit`	OSM administrative, address, boundary, name, source, and contact metadata.
`Forest`	OSM land-cover, vegetation, taxonomic, access, tourism, and environmental metadata.
`Waterbody`	OSM water, waterway, reservoir, harbour, usage, access, and physical-description metadata.
`POI`	OSM amenity, shop, tourism, healthcare, transport, accessibility, service and other metadata.

This design preserves the expressive coverage of OSM without turning every tagged feature type into a separate ontology class. As a result, ToposKG can store detailed OSM annotations while maintaining a simple and reusable upper ontology.

OSM-derived classes. In addition to OSM-derived predicates, ToposKG introduces a set of OSM-derived subclasses under `Waterbody`. These classes correspond to commonly used OSM `water=*` values and provide a controlled hydrological specialization of the broader `Waterbody` class:

`Basin`, `Canal`, `Ditch`, `Drain`, `Fish_pass`, `Harbour`, `Lagoon`, `Lake`,
`Lock`, `Moat`, `Oxbow`, `Pond`, `Rapids`, `Reflecting_pool`, `Reservoir`,
`River`, `Stream`, `Stream_pool`, `Wastewater`.

These classes are not arbitrary copies of the full OSM schema. They are introduced only where an OSM tag corresponds to a stable and semantically useful geographic category in ToposKG, namely a recognizable type of water body.

In similar fashion, a subset of amenity classes are extracted and integrated into our ontology. These classes correspond to frequent and semantically stable OpenStreetMap `amenity=*` tags that denote recognizable types of points of interest, such as restaurants, schools, hospitals, parking facilities, public services, and cultural venues. Each generated class remains part of the ToposKG conceptual model through the `POI` hierarchy, while its source semantics are made explicit through `topos:osmTag`, `dcterms:source`, and `rdfs:isDefinedBy` links to the corresponding OSM documentation.

Mobility and parking	{Parking, Parking_space, Bicycle_parking, Fuel}
Food and drink	{Restaurant, Fast_food, Cafe, Bar, Pub}
Education and culture	{School, University, Library, Theatre, Cinema, Studio}
Health and emergency services	{Hospital, Clinic, Doctors, Pharmacy, Veterinary, Fire_station, Police}
Public and civic services	{Townhall, Post_office, Community_centre, Place_of_worship, Bank, Atm}
Public facilities and urban furniture	{Bench, Shelter, Toilets, Fountain, Vending_machine}
Waste and recycling infrastructure	{Waste_basket, Waste_disposal, Recycling}
Commercial and social places	{Marketplace, Grave_yard}

Non-OSM class-specific attributes. ToposKG also contains a small set of core and non-OSM attributes. These properties are not part of the OSM alignment layer; instead, they express shared ToposKG metadata, administrative hierarchy, external identifiers, or domain-specific relations.

ToposKGFeature [
hasName : human-readable feature name,
hasDescription : textual feature description,
hasArea : area associated with the feature,
hasWikidataLink : link to an associated Wikidata resource]

AdministrativeUnit [
hasISO3Code : ISO 3166-1 alpha-3 country code,
hasUpperAdminUnit : link to the broader administrative unit]

Mountain [
hasMountainLevel : mountain granularity or hierarchy level,
hasUpperMountain : link to a broader mountain or mountain range,
inCountry : country associated with the mountain,
inContinent : continent associated with the mountain]

Sea [hasMRGID : Marine Regions Gazetteer identifier]

2.2.4 Top-level thematic organization

All ToposKG-specific concepts are organized beneath ToposKGFeature, which acts as a semantic pivot between imported geospatial abstractions and domain-specific knowledge. Rather than organizing all entities in a single heterogeneous hierarchy, the ontology partitions them into thematic branches representing different principles of classification.

AdministrativeUnit captures jurisdictional entities whose meaning depends on governance or territorial organization. Natural captures entities whose identity derives from physical geography. General functions as a residual but intentional abstraction for entities that are neither naturally occurring landforms nor formal administrative constructs.

This organization reflects a faceted design strategy. Instead of overloading one branch with incompatible semantics, each branch groups entities that share common structural assumptions and likely relations. This improves schema coherence while also making reasoning patterns more predictable. Administrative containment, for example, differs fundamentally from ecological or topographic relations, and this partitioning preserves that distinction.

Figure 10 summarizes this branching structure.

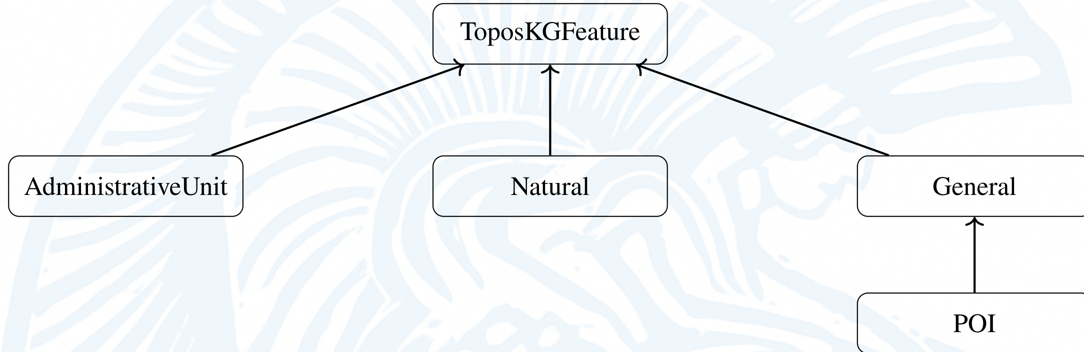


Figure 10: Top-level ToposKG class organization

2.2.5 Administrative hierarchy modeling

The administrative branch models nested territorial units as a hierarchy of explicit subclasses corresponding to administrative granularity levels. The intent is not merely taxonomic classification but scalable representation of jurisdictional nesting.

A notable design choice is that hierarchical containment is represented both intensionally and relationally. It is captured intensionally through subclasses corresponding to levels of administration, and relationally through `hasUpperAdminUnit`, which encodes upward administrative dependency among instances. These two mechanisms serve different purposes: subclasses represent structural categories, while the object property supports traversable territorial hierarchies.

This dual representation supports several forms of reasoning. Queries may operate at the schema level (e.g., retrieve all level-2 administrative regions), at the instance level (e.g., traverse from municipality to country), or across both simultaneously.

Another important design property is granularity neutrality. The hierarchy is not tied to a single national administrative model but can accommodate countries with very different subdivision depths. This is especially important for integrating heterogeneous administrative data sources.

Figure 11 details this branch.

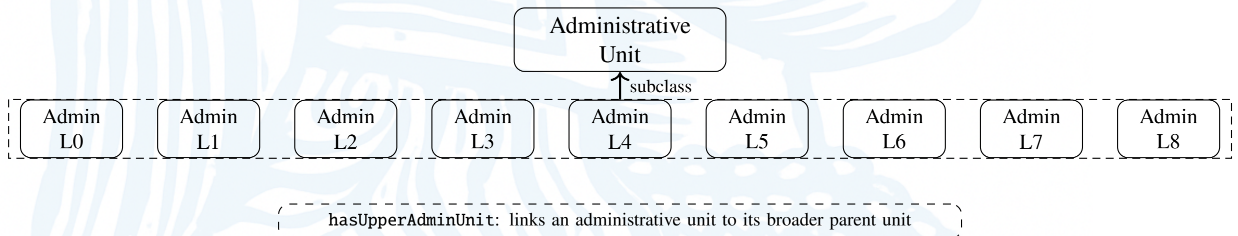


Figure 11: Administrative-unit levels and hierarchical containment relation.

Properties of Administrative Units

2.2.6 Natural feature taxonomy

The natural branch organizes entities according to geomorphological and environmental identity rather than political boundaries. Its subclasses represent broad natural categories that can support finer thematic refinement without requiring redesign of the upper ontology.

A central design feature here is progressive specialization. Broad categories such as `WaterBody` act as abstraction layers from which finer distinctions can descend. This avoids prematurely hard-coding exhaustive environmental taxonomies while still enabling later specialization into rivers, lakes, lagoons, canals, and additional hydrological categories.

The branch also illustrates differentiated relation design. Some subclasses are primarily characterized through datatype descriptors, such as mountain-level attributes or marine identifiers, while others may support richer topological or hydrological relations in future extensions. This indicates an ontology designed not simply for classification, but for incremental semantic enrichment.

Importantly, the natural hierarchy remains conceptually independent from data-source-specific classifications. Although some subclasses align with categories derived from OSM, they are modeled as ontology concepts rather than direct replicas of source schemas, which preserves abstraction from source-system idiosyncrasies.

Figure 12 shows the structure of this branch.

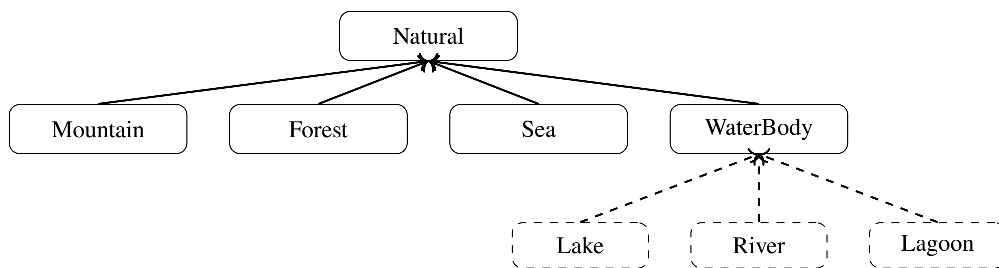


Figure 12: Natural feature taxonomy and progressive specialization.

2.2.7 External linking and shared semantic attributes

ToposKG distinguishes feature identity from descriptive annotations and external references through a lightweight but important metadata layer.

The class `WikidataLink` functions as a bridge entity rather than reducing external identifiers to simple literals attached directly to features. This choice makes alignment itself representable and extensible. Additional provenance, confidence measures, or mappings to other external sources could be introduced around the link entity without modifying feature definitions.

Shared datatype properties occupy a similarly generic layer. Rather than embedding descriptive attributes deep within each subclass, common descriptive predicates are attached at or near the `ToposKGFeature` level. This promotes reuse across branches and avoids repeating attribute patterns throughout the ontology.

This reflects a modular separation between conceptual classification, relational structure, and descriptive annotation. Such separation is particularly valuable in knowledge graphs expected to evolve under heterogeneous source integration.

Figure 13 illustrates this semantic linking layer.

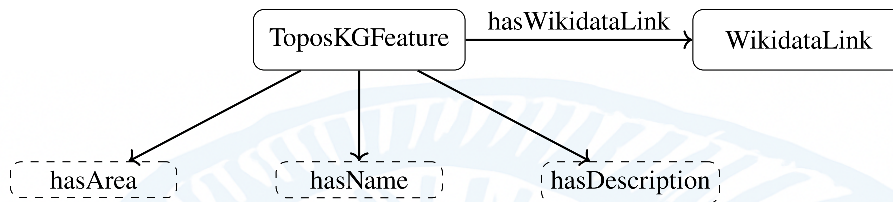


Figure 13: Shared semantic attributes and external linkage layer

2.2.8 General and POI extensibility branch

Although comparatively shallow in the current ontology, the `General` branch plays an architectural role disproportionate to its size. It acts as an extensibility buffer that prevents forcing heterogeneous place-like entities into inappropriate natural or administrative abstractions.

This is particularly important for points of interest, whose semantics are often functional, cultural, or infrastructural rather than territorial or physical. By locating `POI` under `General`, the ontology reserves space for future thematic subdivisions such as cultural sites, transportation facilities, public services, or commercial entities without restructuring upper layers.

This reflects an open-world design philosophy: absence of current specialization is treated as intentional incompleteness rather than a closed taxonomy. In ontology engineering terms, this improves maintainability because expansion occurs through subclass growth rather than structural revision.

2.2.9 Design principles embodied in the ontology

Taken together, the ontology exhibits several recurring design principles:

- **Layered modularity**, separating foundational geospatial semantics from domain specialization;
- **Controlled extensibility**, using abstract superclass nodes as expansion points;
- **Dual structural and relational modeling**, combining subclass taxonomies with object-property hierarchies;
- **Interoperability by reuse**, grounding semantics in GeoSPARQL and external linked data;
- **Abstraction from source schemas**, avoiding direct mirroring of source-specific taxonomies such as OSM tags.

These principles make ToposKG not simply a catalog of geographic entity types, but an ontology engineered for long-term growth, integration, and semantic reuse.

2.3 Class-level property overview

Table 5 summarizes the main properties associated with each major class in the ToposKG ontology. The table separates structural ontology properties, such as parent-unit and external-link relations, from the large number of source-derived attributes imported from OSM and other datasets. For OSM-heavy classes, only a representative subset of properties is shown to keep the table readable; the ontology contains many additional source-specific attributes, especially for `POI`, `Forest`, `WaterBody`, and `AdministrativeUnit`.

Table 5: Main ToposKG classes and associated properties.

Class	Associated properties
ToposKGFeature	hasName, hasDescription, hasArea, hasWikidataLink. These are generic properties shared by all ToposKG feature types.
geo:Feature	geo:hasGeometry. This links a semantic feature to its geometric representation.
AdministrativeUnit	hasUpperAdminUnit, hasISO3Code, hasOSMLevel, hasPopulation, hasPopulationDate, hasPopulationCensus2020, hasBoundary, hasBoundary_type, hasBorder_type, hasCapital, hasPlace, hasType, hasTimezone, hasWebsite, hasUrl, hasImage, hasFullName, hasDesignation, hasIs_in, hasIs_inCountry, hasIs_inCountry_code, hasIs_inState, hasIs_inState_code, hasIs_inIso_3166_2, hasLand_area, hasEle, hasLayer, hasNetwork, hasOperator, hasOperatorShort, hasOperatorWikidata, hasNote, hasNoteGeometry, hasSource, hasSource_ref, hasStart_date, hasTwitter, hasWikimedia_commons, hasWikipediaEs, hasContactWebsite, hasContactYoutube, hasAddrCity, hasAddrStreet, hasAddrHousenumber, hasAccess, hasAmenity, hasBuilding, hasBuildingLevels, hasHeritage, hasNatural. Many additional OSM-derived administrative attributes are included in the full ontology.
AdminUnit_Level0–AdminUnit_Level8	These classes inherit the administrative-unit properties through AdministrativeUnit. They primarily specialize administrative entities by granularity level rather than introducing separate properties.
Natural	No major direct properties are assigned at this abstraction level. Its role is mainly classificatory, grouping natural geographic entities such as mountains, seas, forests, and water bodies.
Mountain	hasUpperMountain, hasMountainLevel, inCountry, inContinent.
Sea	hasMRGID.

Class	Associated properties
Forest	hasGenus, hasGenusEn, hasSpecies, hasSpeciesEn, hasSpeciesWikidata, hasSpeciesWikipedia, hasTaxon, hasTaxonEn, hasTaxonFamily, hasLeafType, hasLeafCycle, hasWood, hasWoodland, hasWoodDensity, hasWoodDamage, hasNatural, hasLandcover, hasLanduse, hasLanduseHistoric, hasSurface, hasWetland, hasWater, hasWaterwayType, hasTree, hasTrees, hasHeight, hasLevel, hasTourism, hasSport, hasHistoric, hasHistoricPeriod, hasHeritageInscriptionDate, hasInformation, hasHiking, hasHorse, hasHunting, hasInside, hasIntermittent, hasInternetAccess, hasIsland, hasIucnLevel, hasLegalStatus, hasSource, hasSourceDate, hasSourceGeometry, hasSourceWebsite, hasSurveyDate, hasStartDate, hasWebsiteMap, hasWebsitePark. Many additional OSM-derived forest and land-cover attributes are included in the full ontology.
WaterBody	hasWaterway, hasLake, hasBasin, hasReservoirType, hasDam, hasBarrier, hasDepth, hasLength, hasEstWidth, hasUsage, hasAttraction, hasBoat, hasCanoe, hasMotorboat, hasShip, hasFishing, hasMaritime, hasHarbour, hasSeamarkHarbourCategory, hasTunnel, hasUnderground, hasCovered, hasCrossing, hasConstruction, hasMaterial, hasBuildingMaterial, hasBuildingPart, hasFenceType, hasWall, hasTemperature, hasHotWater, hasDrinkingWater, hasFountain, hasBathType, hasKneippWaterCure, hasDisusedWater, hasWatermillDisused, hasEmergency, hasOutdoor, hasPhone1, hasGps, hasVhf, hasFee, hasDate, hasCreatedBy, hasContent, hasHistoricCivilization, hasRefGRWwf, hasWdbSource. Additional water-related OSM properties may also be present depending on the source entities.
General	No major direct properties are assigned at this abstraction level. It acts as an extensibility class for geospatial entities that do not fit naturally into the administrative or natural branches.

Class	Associated properties
POI	hasAmenityBar, hasAmenityCafe, hasAmenityCinema, hasAmenityRestaurant, hasAeroway, hasTourism, hasShop, hasOffice, hasHealthcare, hasLeisure, hasHistoric, hasArchaeologicalSite, hasAtm, hasBank, hasBakery, hasBar, hasCafe, hasCinema, hasTheatre, hasToiletsAccess, hasToiletsFee, hasWheelchair, hasWheelchairDescription, hasOpeningHours, hasCheckDateOpeningHours, hasPhone, hasAltPhone, hasWebsite, hasWebsiteEn, hasWebsiteEl, hasWebsiteMenu, hasContactWebsite, hasEmail, hasBrand, hasBrandWikidata, hasBrandWikipedia, hasOperator, hasOperatorWikidata, hasCuisine, hasDietVegan, hasDietVegetarian, hasTakeaway, hasDelivery, hasOutdoorSeating, hasSmoking, hasInternetAccess, hasWifi, hasCapacity, hasFee, hasPaymentCash. The full ontology contains many more POI-specific OSM attributes, covering accessibility, services, commercial metadata, transport, food, culture, public facilities, and operational details.
WikidataLink	No major datatype properties are emphasized in the simplified presentation. The class is mainly used as the range of hasWikidataLink.
geo:Geometry	Geometry literals and serializations are handled through GeoSPARQL-compatible geometry modeling rather than ToposKG-specific class properties.

2.4 Querying ToposKG

In this subsection, we will be presenting GeoSPARQL queries users can experiment with and start working on their own iterations of ToposKG. Users can use the public SPARQL endpoint of ToposKG or locally with their own instances of our GeoKG. First we will be presenting the prefixes used in the queries below:

```
PREFIX topos: <http://toposkg.di.uoa.gr/ontology/>
PREFIX geo:   <http://www.opengis.net/ont/geosparql#>
PREFIX geof:  <http://www.opengis.net/def/function/geosparql/>
PREFIX uom:   <http://www.opengis.net/def/uom/OGC/1.0/>
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:   <http://www.w3.org/2001/XMLSchema#>
```

2.4.1 Introductory Queries

Query 1: Retrieve features with names and geometries

```
SELECT ?feature ?name ?geometry
WHERE {
   ?feature rdf:type topos:ToposKGFeature ;
        topos:hasName ?name;
        geo:hasGeometry ?geometry .

  ?geometry geo:asWKT ?wkt .
}
LIMIT 100
```

Query 2: Points of Interest with geometry

```
SELECT ?adminUnit ?name ?wkt
WHERE {
  ?adminUnit rdf:type topos:POI ;
            topos:hasName ?name ;
            geo:hasGeometry ?geometry .
  ?geometry geo:asWKT ?wkt .
}
LIMIT 100
```

Query 3: Select level-0/country units ordered by their names

```
SELECT DISTINCT ?name
WHERE {
  ?country rdf:type topos:AdminUnit_Level0 ;
           topos:hasName ?name ;
           geo:hasGeometry ?geometry .

  ?geometry geo:asWKT ?wkt .
}
ORDER BY ?name
LIMIT 20
```

Query 4: Mountains with thematic information

```
SELECT ?mountain ?name ?country ?level ?desc
WHERE {
  ?mountain rdf:type topos:Mountain ;
            topos:hasName ?name ;
            geo:hasGeometry ?geometry .
  ?mountain topos:inCountry ?country .
  ?mountain topos:hasLevel ?level .
  ?mountain topos:hasDescription ?desc .

  ?geometry geo:asWKT ?wkt .
}
LIMIT 100
```

Query 5: Seas with MRGID identifiers

```
SELECT ?sea ?name ?mrgid ?wkt
WHERE {
  ?sea rdf:type topos:Sea ;
       topos:hasName ?name ;
       geo:hasGeometry ?geometry .
  ?sea topos:hasMRGID ?mrgid .

  ?geometry geo:asWKT ?wkt .
}
LIMIT 5
```

Query 6: Administrative hierarchy

```
SELECT ?lowerUnit ?lowerName ?upperUnit ?upperName
WHERE {
  ?lowerUnit rdf:type topos:AdminstrativeUnit ;
             topos:hasName ?lowerName ;
             topos:hasUpperAdminUnit ?upperUnit .

  ?upperUnit topos:hasName ?upperName .
}
ORDER BY ?upperName ?lowerName
LIMIT 100
```

Query 7: Mountain hierarchy

```
SELECT ?mountain ?mountainName ?upperMountain ?upperMountainName
WHERE {
  ?mountain rdf:type topos:Mountain ;
            topos:hasName ?mountainName ;
            topos:hasUpperMountain ?upperMountain .

  ?upperMountain topos:hasName ?upperMountainName .
}
```

Query 8: Largest Level 1 Administrative units in Greece by area

```
SELECT DISTINCT ?feature ?name
WHERE {
  ?feature rdf:type topos:AdminUnit_Level1 ;
           topos:hasName ?name ;
           topos:hasArea ?area ;
           geo:hasGeometry ?g_level1 .
    ?g_level1 geo:asWKT ?wkt_level1.

  ?feature_0 rdf:type topos:AdminUnit_Level0 ;
           geo:hasGeometry ?g_level0 .
    ?g_level0 geo:asWKT ?wkt_level0.

    FILTER(regex(str(?feature_0),"grc") && regex(str(?feature),"grc"))
    FILTER(geof:sfWithin(?wkt_level1,?wkt_level0))
}
ORDER BY DESC(?area)
```

Query 9: Polygonal geometries only

```
SELECT ?feature ?name ?wkt
WHERE {
 ?feature rdf:type topos:ToposKGFeature ;
          topos:hasName ?name ;
          geo:hasGeometry ?g .

 ?g geo:asWKT ?wkt .

 FILTER(
   regex(STR(?wkt),"POLYGON") ||
   regex(STR(?wkt),"MULTIPOLYGON")
 )
}
LIMIT  10
```

Query 10: Features inside a bounding box

```
SELECT ?feature ?name ?wkt
WHERE {
 ?feature rdf:type topos:ToposKGFeature ;
          topos:hasName ?name ;
          geo:hasGeometry ?g .

 ?g geo:asWKT ?wkt .

 BIND(
 "POLYGON((19 34,29 34,29 42,19 42,19 34))"
 ^^geo:wktLiteral AS ?bbox
 )

 FILTER(geof:sfWithin(?wkt, ?bbox))
}
LIMIT 100
```

Query 11: Level 2 administrative units near mountains

```
SELECT ?feature ?name ?mname (geof:distance(?wkt,?mwkt) as ?distance)
WHERE {
 ?feature rdf:type topos:AdminUnit_Level2 ;
          topos:hasName ?name ;
          geo:hasGeometry ?g .

 ?g geo:asWKT ?wkt .

 ?m rdf:type topos:Mountain ;
          topos:hasName ?mname ;
          geo:hasGeometry ?mg .

 ?mg geo:asWKT ?mwkt .

    FILTER(?distance < 200)
}
LIMIT 100
```

Query 12: Water bodies intersecting administrative units

```
SELECT ?adminName ?waterName
WHERE {

 ?admin rdf:type topos:AdminstrativeUnit ;
        topos:hasName ?adminName ;
        geo:hasGeometry ?ag .

 ?water rdf:type topos:Waterbody ;
        topos:hasName ?waterName ;
        geo:hasGeometry ?wg .

 ?ag geo:asWKT ?awkt .
 ?wg geo:asWKT ?wwkt .

 FILTER( geof:sfIntersects(?awkt, ?wwkt) )
}
LIMIT 10
```

Query 13: Forests contained in administrative units

```
SELECT ?adminName ?forestName
WHERE {

 ?admin rdf:type topos:AdminstrativeUnit ;
        topos:hasName ?adminName ;
        geo:hasGeometry ?ag .

 ?forest rdf:type topos:Forest ;
         topos:hasName ?forestName ;
         geo:hasGeometry ?fg .

 ?ag geo:asWKT ?awkt .
 ?fg geo:asWKT ?fwkt .

 FILTER( geof:sfWithin(?fwkt, ?awkt) )
}
LIMIT 5
```

Query 14: Wikidata-linked entities with geometry

```
SELECT ?feature ?name ?wikidata ?wkt
WHERE {
 ?feature rdf:type topos:ToposKGFeature ;
          topos:hasName ?name ;
          topos:hasWikidataLink ?wikidata ;
          geo:hasGeometry ?g .

 ?g geo:asWKT ?wkt .
}
LIMIT 100
```

Query 15: Administrative units containing forests near water bodies

```
SELECT ?admin ?adminName ?forest ?forestName ?water ?waterName ?distance
WHERE {
  ?admin rdf:type topos:AdminstrativeUnit ;
         topos:hasName ?adminName ;
         geo:hasGeometry ?adminGeom .

  ?forest rdf:type topos:Forest ;
          topos:hasName ?forestName ;
          geo:hasGeometry ?forestGeom .

  ?water rdf:type topos:Waterbody ;
         topos:hasName ?waterName ;
         geo:hasGeometry ?waterGeom .

  ?adminGeom geo:asWKT ?adminWKT .
  ?forestGeom geo:asWKT ?forestWKT .
  ?waterGeom geo:asWKT ?waterWKT .

  FILTER(geof:sfWithin(?forestWKT, ?adminWKT))
  FILTER(geof:sfIntersects(?waterWKT, ?adminWKT))

  BIND(geof:distance(?forestWKT, ?waterWKT, uom:metre) AS ?distance)

  FILTER(?distance <= 10000)
}
LIMIT 100
```

For more thematic information we provide some additional query suites that might be of interest to users.

2.4.2 Administrative governance and territorial analysis

Query 16: Retrieve the administrative hierarchy of a country

```
SELECT ?country ?countryName ?region ?regionName ?local ?localName
WHERE {
  VALUES ?targetCountryName { "Greece" }

  ?country rdf:type/rdfs:subClassOf* topos:AdminUnit_Level0 ;
           topos:hasName ?countryName .

  ?region rdf:type/rdfs:subClassOf* topos:AdminUnit_Level1 ;
          topos:hasName ?regionName ;
          topos:hasUpperAdminUnit ?country .

  ?local rdf:type/rdfs:subClassOf* topos:AdminUnit_Level2 ;
         topos:hasName ?localName ;
         topos:hasUpperAdminUnit ?region .

  FILTER(STR(?countryName) = ?targetCountryName)
}
ORDER BY ?regionName ?localName
```

Query 17: Find the largest administrative units inside a country

```
SELECT ?unit ?name ?area
WHERE {
  VALUES ?targetCountryName { "Greece" }

  ?country rdf:type/rdfs:subClassOf* topos:AdminUnit_Level0 ;
           topos:hasName ?countryName .

  ?unit rdf:type/rdfs:subClassOf* topos:AdministrativeUnit ;
        topos:hasUpperAdminUnit+ ?country ;
        topos:hasName ?name ;
        topos:hasArea ?area .

  FILTER(STR(?countryName) = ?targetCountryName)
}
ORDER BY DESC(xsd:double(?area))
LIMIT 20
```

Query 18: Retrieve country identifiers for data integration

```
SELECT ?country ?name ?iso3 ?wikidata
WHERE {
  ?country rdf:type/rdfs:subClassOf* topos:AdminUnit_Level0 ;
           topos:hasName ?name .

  OPTIONAL { ?country topos:hasISO3Code ?iso3 . }
  OPTIONAL { ?country topos:hasWikidataLink ?wikidata . }
}
ORDER BY ?name
```

2.4.3 Environmental and physical-geography analysis

Query 19: Find natural features inside a given administrative area

```
SELECT ?feature ?name ?class ?classLabel
WHERE {
  VALUES ?targetAreaName { "Thessaly" }

  ?area rdf:type/rdfs:subClassOf* topos:AdministrativeUnit ;
        topos:hasName ?areaName ;
        geo:hasGeometry/geo:asWKT ?areaWKT .

  ?feature rdf:type ?class ;
           topos:hasName ?name ;
           geo:hasGeometry/geo:asWKT ?featureWKT .

  ?class rdfs:subClassOf* topos:Natural .

  FILTER(STR(?areaName) = ?targetAreaName)
  FILTER(geof:sfWithin(?featureWKT, ?areaWKT))
}
ORDER BY ?classLabel ?name
```

Query 20: Retrieve mountains and their broader mountain ranges

```
SELECT ?mountain ?mountainName ?range ?rangeName ?country
            ?continent ?mountainLevel
WHERE {
  ?mountain rdf:type/rdfs:subClassOf* topos:Mountain ;
            topos:hasName ?mountainName .

  OPTIONAL { ?mountain topos:hasUpperMountain ?range . }
  OPTIONAL { ?range topos:hasName ?rangeName . }
  OPTIONAL { ?mountain topos:inCountry ?country . }
  OPTIONAL { ?mountain topos:inContinent ?continent . }
  OPTIONAL { ?mountain topos:hasMountainLevel ?mountainLevel . }
}
ORDER BY ?continent ?country ?rangeName ?mountainName
```

Query 21: Find water bodies that intersect forests

```
SELECT ?water ?waterName ?waterType ?forest ?forestName
WHERE {
  ?water rdf:type topos:Waterbody ;
         topos:hasName ?waterName ;
         geo:hasGeometry/geo:asWKT ?waterWKT .

  ?forest rdf:type topos:Forest ;
          topos:hasName ?forestName ;
          geo:hasGeometry/geo:asWKT ?forestWKT .

  FILTER(geof:sfIntersects(?waterWKT, ?forestWKT))
}
ORDER BY ?forestName ?waterName
LIMIT 10
```

2.4.4 3. Urban services, accessibility, tourism, and everyday POI discovery

Query 22: Find food and drink places with cuisine, diet, and opening-hour metadata

```
SELECT ?poi ?name ?typeLabel ?cuisine ?vegetarian ?vegan ?openingHours
WHERE {
  VALUES ?targetAreaName { "Albania" }
  VALUES ?targetType {
    topos:Restaurant
    topos:Fast_food
    topos:Cafe
    topos:Bar
    topos:Pub
  }

  ?area rdf:type/rdfs:subClassOf* topos:AdminstrativeUnit ;
        topos:hasName ?areaName ;
        geo:hasGeometry/geo:asWKT ?areaWKT .

  ?poi rdf:type/rdfs:subClassOf* ?targetType ;
       topos:hasName ?name ;
       geo:hasGeometry/geo:asWKT ?poiWKT .

  ?targetType skos:prefLabel ?typeLabel .

  OPTIONAL { ?poi topos:hasCuisine ?cuisine . }
  OPTIONAL { ?poi topos:hasDietVegetarian ?vegetarian . }
  OPTIONAL { ?poi topos:hasDietVegan ?vegan . }
  OPTIONAL { ?poi topos:hasOpeningHours ?openingHours . }

  FILTER(STR(?areaName) = ?targetAreaName)
  FILTER(geof:sfWithin(?poiWKT, ?areaWKT))
}
Limit 5
```

Query 23: Find food and drink places with cuisine, diet, and opening-hour metadata

```
SELECT ?admin ?adminName ?typeLabel (COUNT(DISTINCT ?poi) AS ?poiCount)
WHERE {
  VALUES ?targetType {
    topos:School
    topos:University
    topos:Library
    topos:Theatre
    topos:Cinema
    topos:Community_centre
  }

  ?admin rdf:type/rdfs:subClassOf* topos:AdminUnit_Level2 ;
         topos:hasName ?adminName ;
         geo:hasGeometry/geo:asWKT ?adminWKT .

  ?poi rdf:type/rdfs:subClassOf* ?targetType ;
       geo:hasGeometry/geo:asWKT ?poiWKT .

  ?targetType skos:prefLabel ?typeLabel .

  FILTER(geof:sfWithin(?poiWKT, ?adminWKT))
}
GROUP BY ?admin ?adminName ?typeLabel
ORDER BY ?adminName DESC(?poiCount)
```

3 The Topos Website and GUI

3.1 Website

In this section, we will introduce the Topos website, as well as the GUI, and how a novice user can navigate them.

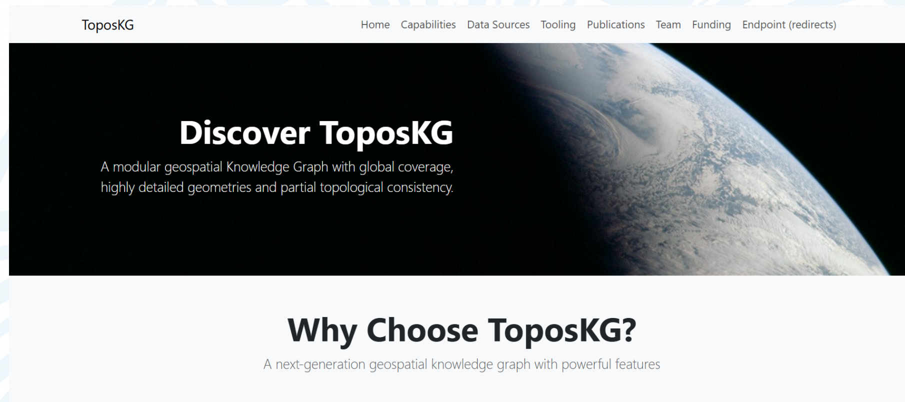


Figure 14: Home page of the Topos website.

3.1.1 Capabilities

In the capabilities section of the website, we present our frameworks main strengths. Under the `Python Integration` infobox we provide links to our toll-chain's pip library, python notebook tutorials for the various functionalities of our tool-chain (discussed in Section 4). Users can also find the link to the Topos Github repository under the `Free and Open Source` infobox.

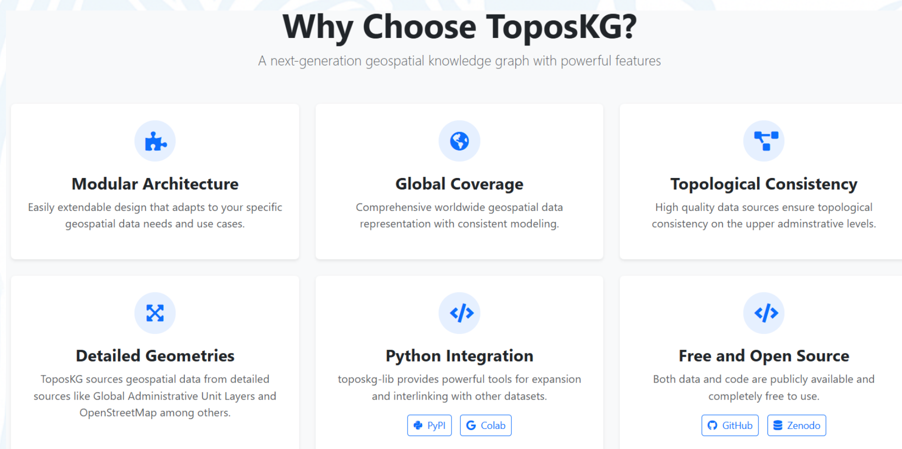


Figure 15: Capabilities section of the Topos website.

3.1.2 Data Sources

In the data sources section of the website, we provide links and information so a user can download and work with the ToposKG data. Currently, users can find the download as a combined large NTriples file, alongside the Topos ontology in the Zenodo repository of our project. Users can also download and work with ToposKG source files through our python package, which will be discussed later in Section 4. Finally, users can download parts of the ToposKG, in a modular fashion through our website's Data Sources Tree API.

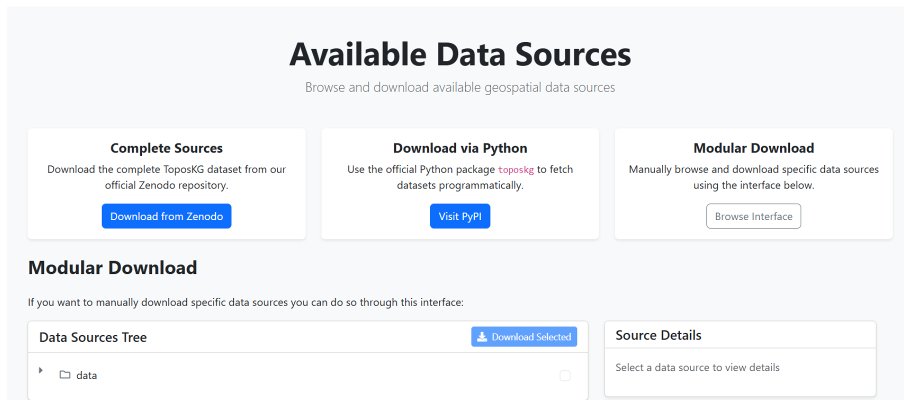


Figure 16: Data Sources section of the Topos website.

Our modular download interface allows users to select files or directories containing the geospatial information they require from ToposKG and download them, building their own geospatial KGs in a modular fashion.

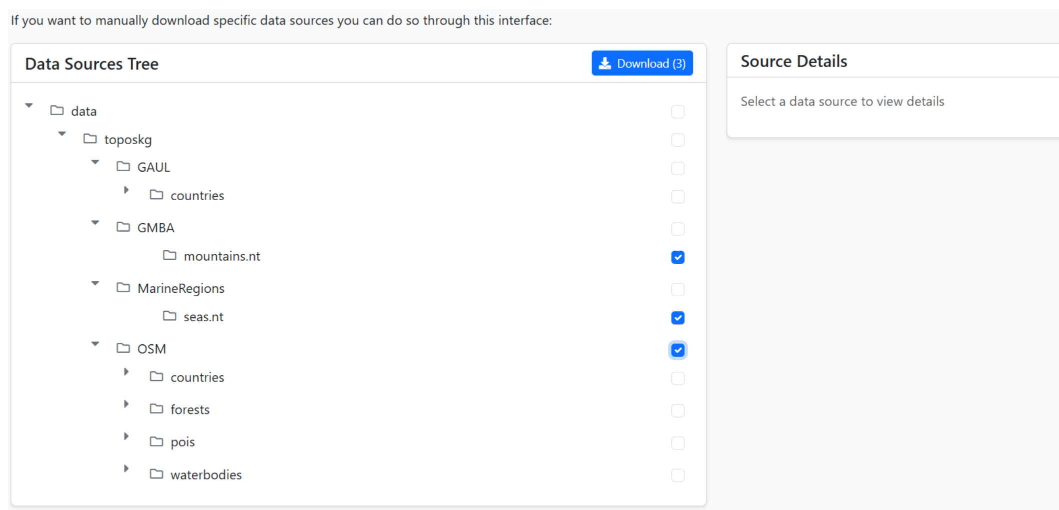


Figure 17: Modular Download interface of the Topos website.

3.1.3 Tooling

In this section of the website, we provide links for all tools available to the Topos framework, including a python notebook tutorial for the LLM powered chat-bot.

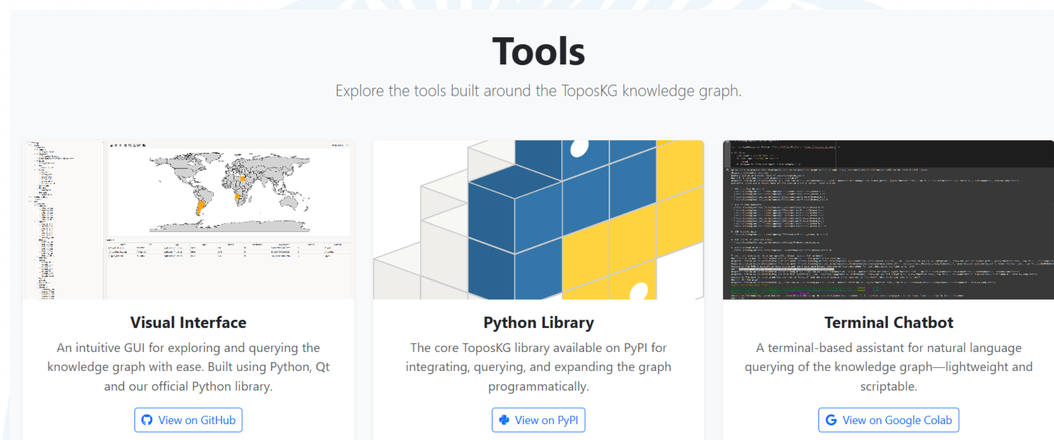


Figure 18: Tooling section of the Topos website.

3.2 GUI for the Topos tool-chain

This subsection is dedicated to the KGUI desktop application of the toposkg-lib. All the functionalities presented here will discussed and analyzed in Section 4. This GUI is intended for less experienced users, that still want to create and utilize GeoKGs. As of the current version of Topos v1.1.0, the GUI is only available as a desktop application. Our team is currently working on developing and maintaining a webapp and site for this functionalities.

To run K-GUI, first create a new virtual environment using your package manager of choice. In this example, we use `conda`.

```
conda create -n kgui python=3.10
conda activate kgui
```

Install the required packages.

```
pip install -r requirements.txt
```

Run the application from the `src/` directory.

```
cd src
python kgui_window.py
```

To function, the desktop application needs a link or a path to the directory holding the ToposKG graph in it's modular representation. If the user is not sure what version or which data to use, they can download the complete modular GeoKG in our project's Zenodo repository. After inserting the link/path the users will be presented with the windows presented in figure 19

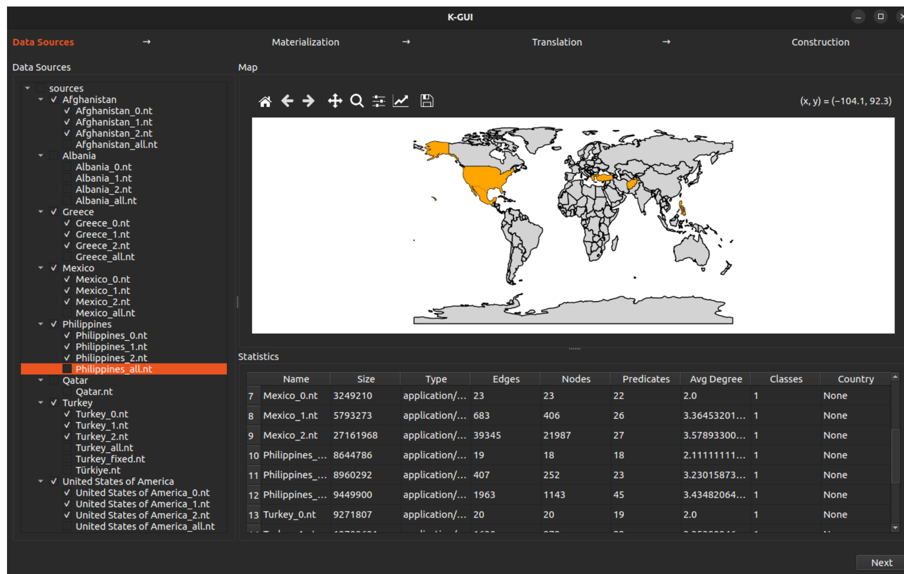
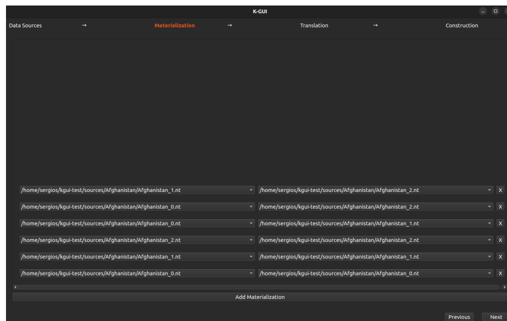


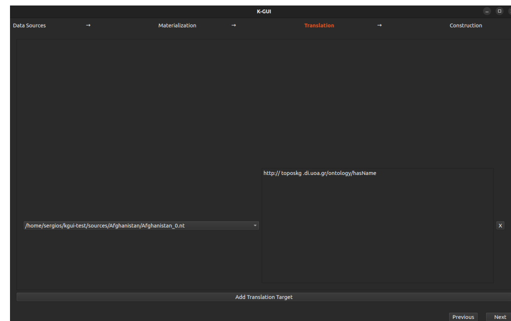
Figure 19: Initial step of the GeoKG creation process

From here, users can select the data sources they want to include in their knowledge graph. When a data source is selected, relevant statistics are displayed for convenience. Additionally, if the selected data is geographically associated with a specific country, that country is highlighted on the map.

After selecting the required data sources for their GeoKG, the user can move on to the next step as shown in figure 20a. This step involves the materialization of geospatial relationships such as within, crosses, intersects, etc. between pairs of data sources. The users here can add as many pairs as they require for the materialization step. More information on the materialization process is available in Section 4.2.



(a) Materialization step of the GeoKG creation process



(b) Translation step of the GeoKG creation process

Figure 20: Materialization and translation steps of the GeoKG creation process

The next step in the construction pipeline, is the translation. This step takes as input a selected predicated and tries to translate all strings of that predicate into their english counterparts. Currently, this approach has more favorable results when targeting toponyms, but simple translations are still valid. The translation step is shown in figure 20b. More information on the translation process is available in Section 4.3

Finally, the users can select the output directory and filename of their constructed GeoKG. A showcase of this is shown in figure 21.

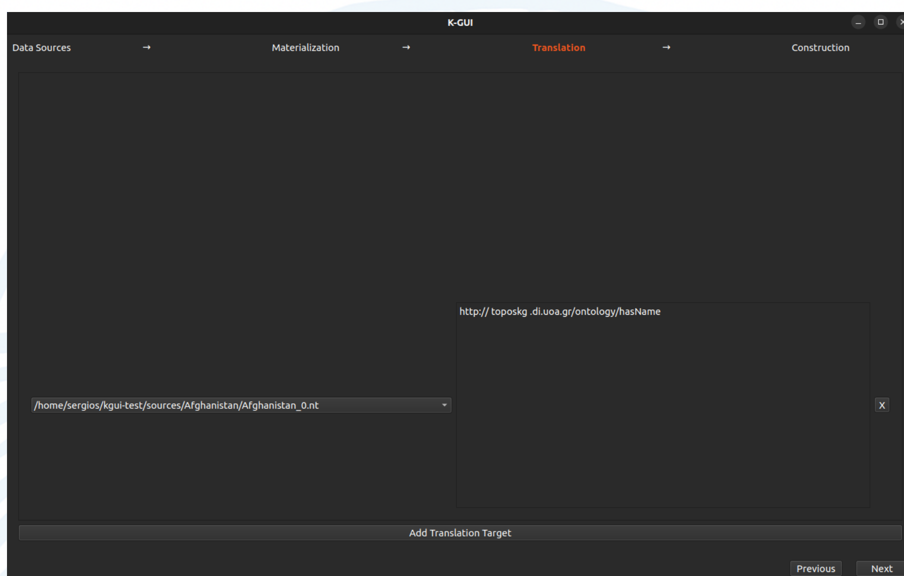


Figure 21: Final step of the GeoKG creation process

4 The `toposkg-lib` Toolchain

`toposkg-lib` is a Python library developed as part of the Topos framework. It provides easy access to powerful functionality for customizing and extending ToposKG but is also compatible with arbitrary source files. Highlights of our effort are the following:

- **Powerful features.** Customize and expand ToposKG using powerful tools for geospatial interlinking, toponym translation and entity linking.
- **Natural Language Interface.** toposkg-lib can be used with a textual interface, powered by LLM function calling.
- **Active development.** toposkg-lib will keep getting updates as we work on our projects.

We recommend using toposkg-lib through pip.

```
pip install toposkg
```

If you want to include the translation functionality.

```
pip install toposkg[tl]
```

If you want to include the function calling functionality.

```
pip install toposkg[fc]
```

We recommend that you install this custom version of RDF-lib before using toposkg-lib.

```
pip install git+https://github.com/SKefalidis/rdflib-speed@main
```

Otherwise you can use the original rdflib.

4.1 GeoKG blueprint builder

In this and the following two subsections, we will showcase the main functionalities of our `toposkg-lib`. We will construct a small geospatial knowledge graph around Greece, using a running example that starts from source discovery and culminates in a reusable GeoKG blueprint. The central idea in `Toposkg` is that we do not begin by loading all available data, but by defining a *blueprint*: a declarative specification of which sources should participate in the graph.

We begin by importing the two core abstractions:

- `KnowledgeGraphSourcesManager`: used to inspect available ToposKG sources.
- `KnowledgeGraphBlueprintBuilder`: used to assemble a custom GeoKG.

Using the source manager:

```python
from toposkg.toposkg_lib_core import (
    KnowledgeGraphBlueprintBuilder,
    KnowledgeGraphSourcesManager
)

sources_manager = KnowledgeGraphSourcesManager(
    sources_repositories='https://toposkg.di.uoa.gr'
)
```

Suppose our running example concerns environmental and administrative data for Greece. We may first inspect available sources:

```python
sources_manager.print_available_data_sources(
    tree=False,
    filter="Greece"
)
```

Illustratively, the first five rows of returned source paths might look like:

Row	Source Path
1	GAUL/countries/Greece/Greece_all.nt
2	OSM/forests/Greece/greece_forest.nt
3	OSM/countries/Greece/Greece_1.nt
4	OSM/rivers/Greece/greece_rivers.nt
5	OSM/lakes/Greece/greece_lakes.nt

Now we define a first blueprint containing administrative boundaries and forests:

```python
builder = KnowledgeGraphBlueprintBuilder()

builder.set_name("ToposKG.nt")
builder.set_output_dir("/content/")

builder.add_source_path(
"/root/.toposkg/sources_cache/toposkg/GAUL/countries/Greece/Greece_all.nt"
)

builder.add_source_path(
"/root/.toposkg/sources_cache/toposkg/OSM/forests/Greece/greece_forest.nt"
)

blueprint = builder.build()
blueprint.construct(validate=False)
```

Conceptually, this materializes a graph combining:

- Administrative entities (country, regions, municipalities)
- Forest entities from OpenStreetMap

- Spatial relations and ontology predicates connecting them

A major strength of the builder is that source selection can scale beyond manual additions.

Selection by regex. Suppose we want all Greece-level sources except fine-grained individual level files:

```
builder.clear_source_paths()

builder.add_source_paths_with_regex(
 sources_manager.get_source_paths(),
 r"(?i).*Greece_(?!\\d).*\\.nt"
)
```

First five matching sources (illustrative):

Row	Regex Match
1	Greece_all.nt
2	Greece_hydrology.nt
3	Greece_transport.nt
4	Greece_landuse.nt
5	Greece_protected_areas.nt

Selection by substring filtering. We may instead restrict ourselves to OSM sources related to Greece:

```
builder.clear_source_paths()

builder.add_source_paths_with_strings(
    sources_manager.get_source_paths(),
    ["Greece", "OSM"]
)
```

This is often easier than regular expressions for exploratory work.

Selection by folders. Whole folders can also act as source bundles:

```
builder.clear_source_paths()

builder.add_source_path(
"/root/.toposkg/sources_cache/toposkg/GAUL/countries/Greece"
)
```

This makes the blueprint abstraction especially convenient: once a collection strategy is found, it can be reused and versioned.

At this stage we have a reusable GeoKG blueprint capable of generating domain-specific geospatial knowledge graphs.

4.2 Materialization

Once a blueprint has selected sources, the next stage is *materialization*: explicitly deriving or expanding implicit knowledge. In most geospatial knowledge graph pipelines, source files only provide the geometries of geospatial entities, or some very basic geospatial interactions between them. Materialization enriches these by generating inferred triples. For our Greece example, imagine a forest polygon intersects a protected region. To calculate this relationship, the user would require an additional. Materialization may derive additional statements such as:

- ForestA within Greece
- ForestA intersects ProtectedArea

This step can dramatically improve downstream querying. For example, without materialization, a query utilizing GeoSPARQL functions would be necessary:

```
SELECT ?forest WHERE {
 ?forest :hasGeometry ;
    :asWKT ?fwkt .
 ?region :hasGeometry ;
    :asWKT ?rwkt .
 :Greece :hasGeometry ;
    :asWKT ?gwkt .
  FILTER(:sfWithin(?fwkt,?gwkt) && :sfIntersects(?fwkt,?rwkt))
}
```

After materialization:

```
SELECT ?forest WHERE {
 ?forest :within :Greece .
 ?forest :intersects ?region .
}
```

This illustrates a recurring philosophy in `toposkg`: preprocessing complexity can simplify reasoning later. In practice, when utilizing our library materialization is enabled, by simply providing file pairs to the KnowledgeGraphBlueprintBuilder class. In our running example this would look something like this:

```
mat_candidates =
↪  [("/root/.toposkg/sources_cache/toposkg/GAUL/countries/Greece/Greece_all.nt",
"/root/.toposkg/sources_cache/toposkg/OSM/forests/Greece/greece_forest.nt")]

builder.set_materialization_pairs(mat_candidates)
blueprint = builder.build()
blueprint.construct(validate=False)
```

This code snippet, will make sure that "difficult" geospatial calculations like within, intersect, crosses, etc., that exist between the pairs of files given as input, will be calculated and stored into the constructed output knowledge graph.

4.3 Translation

Our final stage enriches the GeoKG with multilingual labels. Suppose many entities arrive only with Greek labels. For international interoperability, we may generate English labels automatically.

```
builder = KnowledgeGraphBlueprintBuilder()

builder.set_name("ToposKG_translation.nt")
builder.set_output_dir("/content/")

builder.add_source_path(
"/root/.toposkg/sources_cache/toposkg/OSM/countries/Greece/Greece_1.nt"
)

builder.add_translation_target(
(
 "/root/.toposkg/sources_cache/toposkg/OSM/countries/Greece/Greece_1.nt",
 ["<http://toposkg.di.uoa.gr/ontology/hasName>"]
)
)

blueprint = builder.build()
blueprint.construct(debug=True)
```

The key idea is that translations are not requested globally, but targeted at specific predicates—here:

`<http://toposkg.di.uoa.gr/ontology/hasName>`

Suppose some entities originally contain:

Entity	Original Label	Language
E1	Αθήνα	el
E2	Θεσσαλονίκη	el
E3	Πάτρα	el
E4	Ηράκλειο	el
E5	Ρόδος	el

After translation, keeping only the first five rows:

Entity	Original	English Translation	Predicate
E1	Αθήνα	Athens	hasEnglishName
E2	Θεσσαλονίκη	Thessaloniki	hasEnglishName
E3	Πάτρα	Patras	hasEnglishName
E4	Ηράκλειο	Heraklion	hasEnglishName
E5	Ρόδος	Rhodes	hasEnglishName

This can be viewed as adding triples such as:

```
<entity/Athens> hasEnglishName "Athens"
```

Once translated, multilingual querying becomes easier. A user can ask for:

- all rivers near Athens
- forests near Thessaloniki
- municipalities in Rhodes

without needing original Greek spellings.

Running example conclusion. Our running example has now followed three stages:

1. Build a GeoKG blueprint from selected Greece sources.
2. Materialize expensive geospatial calculations.
3. Translate labels to enrich multilingual accessibility.

Together these illustrate the core philosophy of `toposkg`: begin from modular source selection, enrich through reasoning, and expose the resulting graph through interoperable semantic annotations. For larger applications, the same pattern scales from a two-source toy graph to national or continental-scale geospatial knowledge graphs.

4.4 From Raw Data to RDF Graphs

Our library, provides one additional very powerful functionality: It can transformed semistructured data sources into geospatial data. This is achieved through either Naive mappings designed by our team for novice users that do not need to edit their underlying ontologies, or RML mappings. This section is based on a tutorial presented on BIDS 2025 and can be accessed in the tutorial's repository. The input files are represented in `GeoJSON`, an extension of JSON for modeling geospatial information. The desired output of this part of the tutorial is RDF data in NTriples form.

This tutorial section demonstrates how raw Earth Observation (EO) and geospatial data can be transformed into RDF graphs using `toposkg`. We begin with semi-structured GeoJSON files containing information about the city of Hamburg together with Sentinel-1 and Sentinel-2 derived data, and convert them into RDF in NTriples form.

4.4.1 Parsing GeoJSON Naively

The parsing library of `toposkg` allows the user to parse a wide array of semi-structured data (CSV, JSON, XML, KML, GeoJSON, etc.) and transform them into RDF data. One of the two approaches the library uses is the naive converter classes. These classes take as input a semi-structured data source and provide a simple representation of the data source into RDF data. To achieve this we build a simple Python script.

- Create a directory called output/

```
mkdir output
```

- Import the libraries

```
from toposkg.converter.toposkg_lib_geojson_converter import GeoJSONConverter
import os
```

- Select URIs for our ontology

RDF data and knowledge graphs operate on internet resources. A URI is a string that identifies a resource, while a URL is a type of URI that also tells you how and where to access that resource on the web.

An example URI from Wikidata for Hamburg is:

```
https://www.wikidata.org/wiki/Q1055
```

For our part, we will base our URIs on the tutorial site URL:

```
# Ontology URIs for classes and properties
ontology_uri = "https://ai-team-uoa.github.io/LTEOI-BIDS-2025/ontology/"
resource_uri = "https://ai-team-uoa.github.io/LTEOI-BIDS-2025/resource/"
```

Keep in mind that the RDF data we generate are not actual dereferenceable URIs.

- Iterate through our GeoJSON data in geojsons/

```
output_dir = "./output/"

def change_to_nt(file_path, ext):
    base, _ = os.path.splitext(os.path.basename(file_path))
    return os.path.join(output_dir, base + f".{ext}")

directory_in_str = "./geojsons/"
directory = os.fsencode(directory_in_str)

for file in os.listdir(directory):
    filename = os.fsdecode(file)

    geojson_file = os.path.join(
        directory_in_str,
        filename
    )

    output_nt_file = change_to_nt(
        geojson_file,
        "nt"
    )
```

- Create the converter and generate RDF output

Inside the loop:

```
converter = GeoJSONConverter(
    geojson_file,
    output_nt_file,
    ontology_uri,
    resource_uri
)

converter.parse()
converter.export()
```

Complete script:

```
from toposkg.converter.toposkg_lib_geojson_converter import GeoJSONConverter
import os

ontology_uri = "https://ai-team-uoa.github.io/LTEOI-BIDS-2025/ontology/"
resource_uri = "https://ai-team-uoa.github.io/LTEOI-BIDS-2025/resource/"

def change_to_nt(file_path, ext):
    base, _ = os.path.splitext(file_path)
    return base + ".{}".format(ext)

directory_in_str = "./geojsons/"
directory = os.fsencode(directory_in_str)

for file in os.listdir(directory):

    filename = os.fsdecode(file)
    geojson_file = os.path.join(
        directory_in_str,
        filename
    )

    output_nt_file = change_to_nt(
        geojson_file,
        "nt"
    )

    converter = GeoJSONConverter(
        geojson_file,
        output_nt_file,
        ontology_uri,
        resource_uri
    )

    converter.parse()
    converter.export()
```

This script generates a `.nt` file for each input GeoJSON file. These are valid RDF data that can be loaded into an RDF store.

4.4.2 Parsing GeoJSON Files using RML Mappings

The previous method is simple, but it does not let you change how the RDF data looks. For more control over the ontology and generated graph, we use RML mappings. RML (RDF Mapping Language) is a way to transform data from different formats into RDF so that data can be linked and shared on the web. The `toposkg` library includes a default mapping generator that creates a basic RML mapping file for the input data.

Import libraries

```
from toposkg.converter.rml.toposkg_lib_default_mapping_generator import (
    DefaultMappingGenerator
)
import os
```

Select ontology URIs

```
ontology_uri = "https://ai-team-uoa.github.io/LTEOI-BIDS-2025/ontology/"
resource_uri = "https://ai-team-uoa.github.io/LTEOI-BIDS-2025/resource/"
```

Create mapping generator

```
generator = DefaultMappingGenerator()
```

Iterate through geojsons/

```
output_dir = "./output/"

def change_to_nt(file_path, ext):
    base,_ = os.path.splitext(
        os.path.basename(file_path)
    )
    return os.path.join(
        output_dir,
        base + f".{ext}"
    )

directory_in_str="./geojsons/"
directory=os.fsencode(directory_in_str)

for file in os.listdir(directory):

    filename=os.fsdecode(file)
```

```
    geojson_file=os.path.join(
        directory_in_str,
        filename
    )

    output_mapping_file=change_to_nt(
        geojson_file,
        "ttl"
    )
```

Generate mappings

```
generator.generate_mappings(
    "GeoJSON",
    geojson_file,
    output_mapping_file
)
```

Complete mapping generation script:

```
from toposkg.converter.rml.toposkg_lib_default_mapping_generator import (
    DefaultMappingGenerator
)
import os

generator = DefaultMappingGenerator()

directory_in_str="./geojsons/"
directory=os.fsencode(directory_in_str)

for file in os.listdir(directory):

    filename=os.fsdecode(file)

    geojson_file=os.path.join(
        directory_in_str,
        filename
    )

    output_mapping_file=change_to_nt(
        geojson_file,
        "ttl"
    )

    generator.generate_mappings(
        "GeoJSON", geojson_file, output_mapping_file
    )
```

Editing mappings Generated mappings can be edited for customization.
Original mapping:

```
<#FeatureMap> a rr:TriplesMap;
...
rr:subjectMap [
 rr:template
 "https://example.org/resource/{_pyrml_mapper_generated_id}";
];
```

Modified mapping with explicit class:

```
<#FeatureMap> a rr:TriplesMap;
...
rr:subjectMap [
 rr:template
 "https://example.org/resource/{_pyrml_mapper_generated_id}";
 rr:class onto:poi
];
```

Adding explicit classes is important for later question answering components.

Generate RDF triples from mappings

```
from toposkg.converter.rml.toposkg_lib_default_mapping_generator import (
    DefaultMappingGenerator
)
import os

generator = DefaultMappingGenerator()

directory_in_str="./geojsons/"
directory=os.fsencode(directory_in_str)

for file in os.listdir(directory):

    filename=os.fsdecode(file)

    geojson_file=os.path.join(
        directory_in_str,
        filename
    )

    output_mapping_file=change_to_nt(
        geojson_file,
        "ttl"
    )

    output_nt_file=change_to_nt(
        geojson_file,
```

```
        "nt"
    )

    generator.generate_triples(
        output_mapping_file,
        output_nt_file
    )
```

The generated RDF files are available in `output/`.
All scripts are present inside the current directory and can be run directly:

- `geojson_to_ntriples_naive.py` — Naive method
- `geojson_to_mapping.py` — RML mapping generation
- `mapping_to_triples.py` — Mappings to RDF triples

5 Building GeoKGs for GIS Applications using ToposKG and the Topos Framework

In this final section of our manual we will be showcasing how to create a GeoKG for a real life application. For our running example, we will be using the PnyQA system. The description of the PnyQA, as presented in the Topos paper is the following:

> Accessible and accurate electoral analysis can be a boon for researchers, reporters and the general public to better understand how people vote, which factors dominate voting behavior and how elections are managed. With this in mind, we developed PnyQA [?], a tool for visualizing election results using natural language.

At the core of PnyQA lies once again a GeoKG, which enables queries that include spatial relations between administrative units. This GeoKG has once again been enhanced with additional use-case specific information. Specifically, data about the 2020 U.S. Presidential Election and demographic data for the United States, both at a County and State level.

The first consideration we needed to make when creating the GeoKG for our application was the kind of natural language questions, that would be meaningful and interesting for an analyst to ask, based on geographical and statistical information. Thus the process was split into two distinct tasks: 1) Create a geospatial knowledge database that we would use as our base and 2) enrich this knowledge base with appropriate thematic information to cover a statistical analysis of the 2020 US elections.

5.1 Building the GeoKG

The information that we would use as our geospatial base, would be the administrative divisions of the US up to county level (2nd-level division), forests in the US, water bodies in the US and finally mountains/mountain ranges in the US. Here the modular nature of ToposKG, makes the construction of this GeoKG very simple. Using the scripts from Section 4 we can easily construct the appropriate GeoKG.

```
from toposkg.toposkg_lib_core import (
    KnowledgeGraphBlueprintBuilder,
    KnowledgeGraphSourcesManager
)

sources_manager = KnowledgeGraphSourcesManager(
    sources_repositories='https://toposkg.di.uoa.gr'
)

builder = KnowledgeGraphBlueprintBuilder()

builder.set_name("PnyQA.nt")
builder.set_output_dir("/content/")

# Administrative divisions of US (GAUL offers up to 2nd-level)
builder.add_source_path(
"/root/.toposkg/sources_cache/toposkg/GAUL/countries/United States of
↪  America/United States of America_all.nt"
)
```

```
# Forest spatial data in the US
builder.add_source_path(
"/root/.toposkg/sources_cache/toposkg/OSM/forests/United States of
↪  America/United States of America.nt"
)

# Water body spatial data in the US
builder.add_source_path(
"/root/.toposkg/sources_cache/toposkg/OSM/waterbodies/United States of
↪  America/United States of America.nt"
)

# Mountain spatial data
builder.add_source_path(
"/root/.toposkg/sources_cache/toposkg/GMBA/mountains.nt"
)
```

In the real-life application of PnyQA, response time was crucial to the user experience. For this reason, we materialized the spatial relationships between administrative units and the rest of the data. This is illustrated in the code snippet below:

```
us_admin = "/root/.toposkg/sources_cache/toposkg/GAUL/countries/United States of
↪  America/United States of America_all.nt"
us_forest = "/root/.toposkg/sources_cache/toposkg/OSM/forests/United States of
↪  America/United States of America.nt"
us_water = "/root/.toposkg/sources_cache/toposkg/OSM/waterbodies/United States
↪  of America/United States of America.nt"
mountains = "/root/.toposkg/sources_cache/toposkg/GMBA/mountains.nt"

# Create the materialization pairs
# Admin Units -> Forests
# Admin Units -> Water
# Admin Units -> Mountains
# Admin Units -> Admin Units (self discovery for queries like bordering states
↪  or counties)
mat_candidates = [(us_admin,us_forest),(us_admin,us_water),(us_admin,mountains),
(us_admin,us_admin)]

builder.set_materialization_pairs(mat_candidates)
blueprint = builder.build()
blueprint.construct(validate=False)
```

With these snippets, the spatial part of the GeoKG is complete under the file PnyQA.nt.

5.2 Introducing thematic information

The thematic data we collected concerned the 2020 U.S. Presidential Election as well as additional demographic data for the United States, both at a County and State level. These data was access from `https://www.census.gov/`. The data originally, had the column data:

Column Name	Label
GEO_ID	Geography
NAME	Geographic Area Name
DP1_0001C	Count!!SEX AND AGE!!Total population
DP1_0002C	Count!!SEX AND AGE!!Total population!!Under 5 years
DP1_0003C	Count!!SEX AND AGE!!Total population!!5 to 9 years
DP1_0004C	Count!!SEX AND AGE!!Total population!!10 to 14 years
DP1_0005C	Count!!SEX AND AGE!!Total population!!15 to 19 years
DP1_0006C	Count!!SEX AND AGE!!Total population!!20 to 24 years

Table 6: Census data columns and their corresponding labels.

GEO_ID	NAME	Total Pop.	Under 5	5–9	10–14	15–19	20–24
0100000US	United States	331449281	18400235	20130423	21627830	22036076	22166199

Table 7: Sample of the demographic data used in PnyQA.

Data about economical statistics and election data had the same format. For the purposes of our application we transformed this data into the following .csv files. This process was very simple and is omitted from this part of the pipeline. For anyone interested on how the parsing process is available in this python notebook. The final input .csv files have the following format:

education.csv							
Geography	**Geographic Area Name**	**Total**	**Associate**	**Bachelor**	**Master**	**Prof.**	**Doctorate**
01001	Autauga County, Alabama	37860	3323	6320	3399	504	498
01003	Baldwin County, Alabama	155563	14357	31444	13331	2845	2016
01005	Barbour County, Alabama	17797	1385	1296	540	118	113
01007	Bibb County, Alabama	15987	1087	1183	481	109	41
01009	Blount County, Alabama	39814	5220	3540	1382	222	132

Table 8: Sample rows from the education dataset.

income.csv			
Geography	**Geographic Area Name**	**Income 1**	**Income 2**
01001	Autauga County, Alabama	57982	75614
01003	Baldwin County, Alabama	61756	83626
01005	Barbour County, Alabama	34990	51557
01007	Bibb County, Alabama	51721	61655
01009	Blount County, Alabama	48922	66360
...	...	...	...

Table 9: Sample rows from the income dataset.

census.csv								
FIPS	**Name**	**Total Pop.**	**Children**	**Young Adults**	**Middle Aged**	**Seniors**	**Male**	**Female**
01001	Autauga County, Alabama	58805	11646	26306	7986	12867	28390	30415
01003	Baldwin County, Alabama	231767	40268	92612	31293	67594	112627	119140
01005	Barbour County, Alabama	25223	4221	10644	3479	6879	13167	12056
01007	Bibb County, Alabama	22293	3933	10179	3139	5042	11798	10495
01009	Blount County, Alabama	59134	11318	24957	8003	14856	29197	29937
...	...	...	...	...	...	...	...	...

Table 10: Sample rows from the census dataset. Additional columns are omitted for compactness.

election.csv						
FIPS	**State**	**County**	**Party**	**Cand. Votes**	**Total Votes**	**Mode**
21181	KENTUCKY	NICHOLAS	Other	6	3396	TOTAL
21181	KENTUCKY	NICHOLAS	Libertarian	27	3396	TOTAL
21181	KENTUCKY	NICHOLAS	Republican	2408	3396	TOTAL
21181	KENTUCKY	NICHOLAS	Green	0	3396	TOTAL
21181	KENTUCKY	NICHOLAS	Democrat	955	3396	TOTAL

Table 11: Sample rows from the election dataset.

Our first step here is to transform this data into RDF data. To do that we will use the naive mappings introduced in Section 4.4. Let's say that the election data is store in the file election.csv. Then the script we need to create is the following:

```
from toposkg.converter.toposkg_lib_csv_converter.py import CSVConverter
import os

converter = CSVConverter(
    input_file="final.csv",
    out_file="election.nt",
    delimeter=",",
    ontology_uri="http://toposkg.di.uoa.gr/ontology/",
    resource_uri="http://toposkg.di.uoa.gr/resource"
)

converter.parse()
converter.export()
```

The election.nt file will have rdf data of the following form:

Listing 1: Sample RDF triples describing election results.

```
@prefix toposr: <http://toposkg.di.uoa.gr/resource/> .
@prefix topos:  <http://toposkg.di.uoa.gr/ontology/> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#> .

toposr:713e668b141bcde3_0
    topos:county_fips     "21181"^^xsd:string ;
    topos:state           "KENTUCKY"^^xsd:string ;
    topos:county_name     "NICHOLAS"^^xsd:string ;
    topos:party           "Other"^^xsd:string ;
    topos:candidatevotes "6"^^xsd:string ;
    topos:totalvotes      "3396"^^xsd:string ;
    topos:mode            "TOTAL"^^xsd:string .

toposr:713e668b141bcde3_1
    topos:county_fips     "21181"^^xsd:string ;
    topos:state           "KENTUCKY"^^xsd:string ;
    topos:county_name     "NICHOLAS"^^xsd:string ;
    topos:party           "Libertarian"^^xsd:string ;
    topos:candidatevotes "27"^^xsd:string ;
    topos:totalvotes      "3396"^^xsd:string ;
    topos:mode            "TOTAL"^^xsd:string .

toposr:713e668b141bcde3_2
    topos:county_fips     "21181"^^xsd:string ;
    topos:state           "KENTUCKY"^^xsd:string ;
    topos:county_name     "NICHOLAS"^^xsd:string ;
    topos:party           "Republican"^^xsd:string ;
    topos:candidatevotes "2408"^^xsd:string .
```

The following process can be applied to all other csv files. At this current step we have two separate KGs. The GeoKG that we created with spatial data and the KG containing the thematic information. Our final step will be to interlink these two KGs and have our final PnyQA GeoKG.

5.3 Interlinking the GeoKG with thematic information

For this step we will need to install a linking tool named pyJedAI developed and maintained by our team:

```
pip install pyjedai
```

The next step will be to isolate the things that we need for the linking process. To do that we can run the following script:

```
    import csv
from collections import defaultdict
from pathlib import Path

from rdflib import Graph, URIRef, Literal


def short_predicate_name(predicate_uri: str) -> str:
    """
    Convert a full predicate URI into a compact CSV column name.

    Example:
      http://toposkg.di.uoa.gr/ontology/county_name
    becomes:
      county_name
    """
    uri = str(predicate_uri)

    if "#" in uri:
        return uri.rsplit("#", 1)[1]

    return uri.rstrip("/").rsplit("/", 1)[-1]


def clean_object_value(obj) -> str:
    """
    Convert RDF objects into CSV-friendly values.

    Literal:
      "21181"^^xsd:string -> 21181

    URI:
      http://example.org/resource/x -> <http://example.org/resource/x>
    """
    if isinstance(obj, Literal):
        return str(obj)

    if isinstance(obj, URIRef):
        return f"<{obj}>"

    return str(obj)
```

```python
def nt_predicates_to_csv(
    input_nt: str | Path,
    output_csv: str | Path,
    predicates: list[str],
    multi_value_separator: str = " | ",
) -> None:
    """
    Extract selected predicates from an N-Triples file and write them to CSV.

    Parameters
    ----------
    input_nt:
        Path to the input .nt file.

    output_csv:
        Path to the output .csv file.

    predicates:
        List of predicate URIs to extract. They may be given with or without
        angle brackets.

    multi_value_separator:
        Separator used when the same entity has multiple values for the same
        predicate.
    """

    input_path = Path(input_nt)
    output_path = Path(output_csv)

    predicate_refs = [
        URIRef(predicate.strip().strip("<>"))
        for predicate in predicates
    ]

    predicate_to_column = {
        predicate: short_predicate_name(predicate)
        for predicate in predicate_refs
    }

    rows = defaultdict(lambda: {
        column_name: ""
        for column_name in predicate_to_column.values()
    })

    graph = Graph()
    graph.parse(input_path, format="nt")

    for subject, predicate, obj in graph:
        if predicate not in predicate_to_column:
            continue
```

```
        entity = f"<{subject}>"
        column_name = predicate_to_column[predicate]
        value = clean_object_value(obj)

        if rows[entity][column_name]:
            rows[entity][column_name] += f"{multi_value_separator}{value}"
        else:
            rows[entity][column_name] = value

    fieldnames = ["entity"] + [
        predicate_to_column[predicate]
        for predicate in predicate_refs
    ]

    output_path.parent.mkdir(parents=True, exist_ok=True)

    with output_path.open("w", encoding="utf-8", newline="") as outfile:
        writer = csv.DictWriter(outfile, fieldnames=fieldnames)
        writer.writeheader()

        for entity in sorted(rows):
            writer.writerow({
                "entity": entity,
                **rows[entity],
            })
```

This script is also available in toposkg lib. To call this snippet simply run:

```
nt_predicates_to_csv(
    input_nt="election.nt",
    output_csv="collected_election.csv",
    predicates=[
        "http://toposkg.di.uoa.gr/ontology/county_fips",
        "http://toposkg.di.uoa.gr/ontology/state",
        "http://toposkg.di.uoa.gr/ontology/county_name",
    ],
)
```

The output of this script for the election.csv file will look as follows:

entity	county_fips	state	county_name
toposr:713e668b141bcde3_0	21181	KENTUCKY	NICHOLAS

Table 12: Example CSV output row

To collect the necessary data for the linking of the GeoKG, we will call the same method but for specific predicates from our Topos ontology, specifically predicates containg information about FIPS codes and names of each entity. These predicates are the following:

```
nt_predicates_to_csv(
    input_nt="PnyQA.nt",
    output_csv="collected_PnyQA.csv",
    predicates=[
        "http://toposkg.di.uoa.gr/ontology/hasFIPS",
        "http://toposkg.di.uoa.gr/ontology/hasNistFips_code",
        "http://toposkg.di.uoa.gr/ontology/hasName",
    ],
)
```

The output of this script for the election.csv file will look as follows:

entity	hasFIPS	hasNistFips_code	hasName
toposr:usa_121830_23240	36001	36001	Albany

Table 13: Example CSV output row

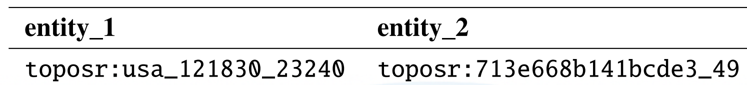
Now we are able to utilize the tool pyJedAI to link these two data files. The following script does just that. These two parsing scripts are available in our Topos GitHub repository:

```
    import pandas as pd

from pyjedai.datamodel import Data
from pyjedai.vector_based_blocking import EmbeddingsNNBlockBuilding


# ------------------------------------------------------------------
# Hard-coded input/output paths
# ------------------------------------------------------------------

D1_PATH = "collected_election.csv"
D2_PATH = "collected_PnyQA.csv"
OUTPUT_PATH = "linked_output.csv"


# ------------------------------------------------------------------
# Load data
# ------------------------------------------------------------------

d1 = pd.read_csv(D1_PATH, sep=",", engine="python", na_filter=False).astype(str)
d2 = pd.read_csv(D2_PATH, sep=",", engine="python", na_filter=False).astype(str)

# Clean column names
d1.columns = d1.columns.str.strip().str.replace("\ufeff", "", regex=False)
d2.columns = d2.columns.str.strip().str.replace("\ufeff", "", regex=False)


# ------------------------------------------------------------------
# Select attributes used for linking
# ------------------------------------------------------------------
```

```
d1 = d1[["entity", "county_fips", "state", "county_name"]]
d2_columns = [["entity", "hasFIPS" , "hasNistFips\_code", "hasName"]]

if "hasNameEn" in d2.columns:
    has_name_en_clean = (
        d2["hasNameEn"]
        .astype(str)
        .str.strip()
        .replace({"nan": "", "None": "", "null": "", "NaN": ""})
    )

    if has_name_en_clean.ne("").any():
        d2["hasNameEn"] = has_name_en_clean
        d2_columns.append("hasNameEn")
    else:
        print("hasNameEn is empty for all rows - excluding it.")

d2 = d2[d2_columns]


# -------------------------------------------------------------------
# Print preview
# -------------------------------------------------------------------

print("Dataset 1 preview:")
print(d1.head())

print("\nDataset 2 preview:")
print(d2.head())

print("\nUsing d1 attributes:", d1.columns[1:].to_list())
print("Using d2 attributes:", d2.columns[1:].to_list())


# -------------------------------------------------------------------
# Create PyJedAI data object
# -------------------------------------------------------------------

data = Data(
    dataset_1=d1,
    attributes_1=d1.columns[1:].to_list(),
    id_column_name_1="entity",
    dataset_2=d2,
    attributes_2=d2.columns[1:].to_list(),
    id_column_name_2="entity",
)


# -------------------------------------------------------------------
# Run embeddings-based blocking / matching
# -------------------------------------------------------------------
```

```python
emb = EmbeddingsNNBlockBuilding(
    vectorizer="sminilm",
    similarity_search="faiss",
)

blocks, g = emb.build_blocks(
    data,
    top_k=1,
    similarity_distance="cosine",
    load_embeddings_if_exist=False,
    save_embeddings=False,
    with_entity_matching=True,
)

mapping_df = emb.export_to_df(blocks)


# -------------------------------------------------------------------
# Merge matches back with the original data
# -------------------------------------------------------------------

merged1 = pd.merge(
    d1,
    mapping_df,
    left_on="entity",
    right_on="id1",
    how="left",
)

final_df = pd.merge(
    merged1,
    d2,
    left_on="id2",
    right_on="entity",
    how="left",
    suffixes=("_gaul", "_osm"),
)

final_df = final_df.drop(columns=["id1", "id2"], errors="ignore")


# -------------------------------------------------------------------
# Save output
# -------------------------------------------------------------------

final_df.to_csv(OUTPUT_PATH, index=False)

print(f"\nSaved output to: {OUTPUT_PATH}")
```

The result of this script will be the following:
This `.csv` file provides the links between the two datasets. We can then transform it into RDF



entity_1	entity_2
toposr:usa_121830_23240	toposr:713e668b141bcde3_49

Table 14: Example linked CSV output row

either with a simple Python script that introduces direct links between the corresponding entities, or by using our RML-based mapping implementation for CSV files.

To summarize the process:

- We create a GeoKG by selecting the necessary spatial data from Topos.
- We select and parse the thematic data, a process supported by Topos' libraries.
- Using a simple linking framework, we interlink the two KGs, creating a compact, ready-to-use GeoKG for our PnyQA application.

5.4 Analysis Table for TerraQ and PnyQA usecases

In Table 15, we provide a brief summary of the tooling functionality necessary to recreate the knowledge graphs used by the TerraQ and PnyQA usecases.

Use case	ToposKG components used	Toolchain components required	Role in the application
TerraQ	Administrative units, natural features, marine regions, and satellite-image metadata.	Modular KG construction; RDF construction and validation; translation of non-English labels; geospatial relation materialization.	TerraQ requires a custom GeoKG for retrieving satellite images through spatial, temporal, and metadata constraints. The translation component was needed because several geospatial features lacked English labels, complicating named-entity disambiguation. Materialization was needed to precompute expensive spatial relations and avoid slow runtime GeoSPARQL evaluation over large geospatial volumes.
PnyQA	U.S. administrative units at state and county level, together with election and demographic data.	Modular KG construction; thematic-data integration; entity/geospatial interlinking; materialization of administrative and neighboring relations; validation.	PnyQA requires a compact application-specific GeoKG for natural-language electoral visualization. Administrative polygons provide the spatial units over which election and demographic data are queried. Topologically consistent geometries and materialized spatial relations are necessary for queries involving neighboring counties, such as retrieving counties adjacent to Texas that voted for a given party.

Table 15: Topos toolchain components required by the two application use cases.

6 Conclusion

This white paper has presented ToposKG and the broader Topos tooling framework as a practical entry point for constructing, exploring, and extending geospatial knowledge graphs. Through the Topos ontology, the Topos website, and the accompanying tool-chain, users are provided with a modular environment for working with geospatially enriched RDF data, integrating thematic information, and developing applications on top of the resulting knowledge graph.

The document serves both as a user manual and as a technical reference for the current release of the Topos framework. It introduced the main design principles behind ToposKG, described the available interaction and construction tools, and demonstrated how these components can be combined in end-to-end workflows, including the creation of a GIS application inspired by our geospatial question-answering system, PnyQA.

While the current version of ToposKG and its tool-chain already provides a complete and usable framework, it should also be viewed as an evolving platform. Future versions will continue to expand the available functionality, incorporate user feedback, and support additional requirements related to GeoKG construction, enrichment, querying, and application development. In this sense, ToposKG is not only a knowledge graph resource, but also a foundation for further experimentation, customization, and community-driven geospatial knowledge graph development. The website storing all relevant info for ToposKG is available here: `https://toposkg.di.uoa.gr/`.

7 Acknowledgments

This project is funded in part by the following scholarships: "GD.402. ARCHI-YPPhD-0824" PhD scholarship, "NIK. D. XRYSOVERGI" PhD scholarship.

AIteam

NATIONAL & KAPODISTRIAN UNIVERSITY OF ATHENS

ToposKG White Paper

Thank you for reading!

Artificial Intelligence Research Group

Department of Informatics and Telecommunications

National and Kapodistrian University of Athens