Virtual Knowledge Graphs over Earth Observation Data

Albulen Pano, Davide Lanti(✉), and Diego Calvanese

Free University of Bozen-Bolzano, Bolzano, Italy
albulen.pano@student.unibz.it, lanti@inf.unibz.it, diego.calvanese@unibz.it

Abstract. Earth Observation (EO) data publication and dissemination continues to grow driven by the increase in satellite launches. openEO is one of the most well known platforms to query EO data (in the form of datacubes) using web APIs. To be truly valuable, EO data often needs to be combined with other data sources, notably relational databases. Knowledge graphs offer a way to bridge the semantic gap between EO data and these additional sources. However, the execution of integrated queries can run into scalability issues due to the enormous volume of EO data. In this paper, we propose addressing this challenge through the use of Virtual Knowledge Graphs — a paradigm that presents data to end-users as a knowledge graph, while keeping the data in its original sources rather than materializing it in graph form. We show the feasibility of our approach by implementing a prototype solution and test it over real world openEO examples.

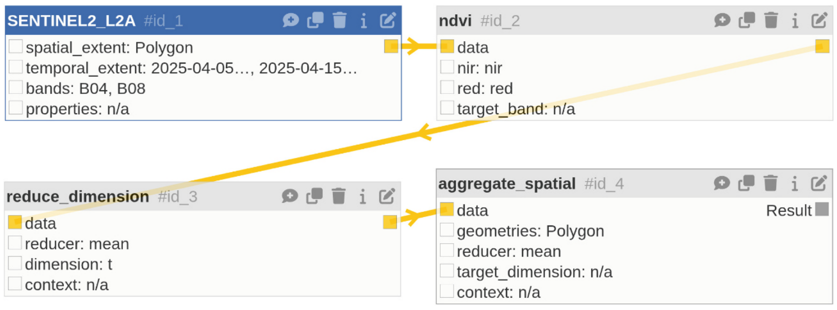
Keywords: Virtual knowledge graph · Earth observation · Ontology-based data access · Raster data · openEO

1 Introduction

The steady growth of satellite launches for Earth Observation (EO) has increased the range and volume of Earth imagery coverage. Satellite imagery is often represented as *raster datacubes* (often called simply *raster data*[1]) with multiple dimensions including space and time, as well as the respective light bands they measure. While the variety of data collected is limited by the number of bands that satellites support (which are a small number) the temporal dimension applied over the whole globe can easily accrue. Moreover, not all entities producing the data need to abide to the same dissemination strategies, hence different datasets and dataset formats could be a result not only of direct satellite observations, but also of interpolation exercises. For example, for weather analysis an entity might retrieve EO data from openEO and generate data in a new netCDF format.

[1] According to openEO, *raster data* is a single 2D grid of spatial values, while a *raster datacube* extends this to multiple dimensions (e.g., time, bands) to represent a series of rasters.

D. Garijo et al. (Eds.): ISWC 2025, LNCS 16141, pp. 149–166, 2026.
https://doi.org/10.1007/978-3-032-09530-5_9



SENTINEL2_L2A #id_1
spatial_extent: Polygon
temporal_extent: 2025-04-05..., 2025-04-15...
bands: B04, B08
properties: n/a

ndvi #id_2
data
nir: nir
red: red
target_band: n/a

reduce_dimension #id_3
data
reducer: mean
dimension: t
context: n/a

aggregate_spatial #id_4
data
geometries: Polygon
reducer: mean
target_dimension: n/a
context: n/a
Result

Fig. 1. Visual representation of a process graph in the openEO Web Editor.

To address these challenges, a prominent solution that has emerged is openEO [26], a web API designed to provide a standardized interface for querying EO data backends, regardless of their underlying datacube storage formats. Central to openEO is the concept of a *process graph*—a JSON-based, directed acyclic graph that represents a sequence of processing steps to be applied to EO data. Each node in the graph corresponds to an openEO process (e.g., loading data, filtering by time, applying a function), and edges define data dependencies between these processes. This structure enables users to build flexible, modular, and backend-independent EO analysis workflows. An example of such a process graph is shown in Fig. 1, demonstrating how complex analyses can be constructed by chaining simple operations.

The openEO specification was submitted to the Open Geospatial Consortium to become a standard in December 2023 [25]. While using openEO resolves the issue of heterogeneity across EO data storage, it does not provide a simple solution to integrate EO data with other domains.

Knowledge Graphs (KGs) provide a simple mechanism to connect concepts between different domains using the ubiquitous graph data model. Integrating additional domains simply involves specifying rules for how to connect two graphs. Unlike a relational database, it is not necessary to reformat entire tables. The most well known interchange standard for graph data is the *Resource Description Framework* (RDF) which represents data as a set of triples. However, despite its conceptual simplicity, RDF requires to re-materialize existing data as triples, which would negate any benefit from efficiently storing datacubes in existing raster data formats.

Virtual Knowledge Graphs (VKGs) provide a lightweight approach to integrate heterogeneous data across a variety of data sources and data formats, without the need to generate new graph datasets. A prominent platform that generates VKGs using relational data is Ontop [10,33], which already supports most well-known relational databases. Relational databases such as PostgreSQL support non-SQL extensions that make it feasible to query also non-relational data. In our work, we have developed the first method to query EO data via the openEO specification together with relational data, by relying on the VKG paradigm. To do so, we provide access to arbitrary openEO processes through corresponding SPARQL functions that can be freely used in a SPARQL query. We then exploit the PL/Python PostgreSQL extension to translate each (first-level) SPARQL function into an openEO process graph that carries out the evaluation of the corresponding openEO function. Conversely, our framework makes it possible to emulate any openEO process graph through suitably nested SPARQL functions. The resource we have constructed builds on the Ontop VKG platform, and is able to operate with a minimal overhead in query response time with respect to directly querying openEO, but provides the important benefit of allowing for integrated cross-domain queries.

The rest of the paper is structured as follows: Sect. 2 discusses related work, Sect. 3 introduces the resource architecture, Sect. 4 elaborates on the implementation inside Ontop, Sect. 5 presents an experimental evaluation, and finally Sect. 6 discusses some limitations and future work.

2 Related Work

EO data is typically represented as raster datacubes derived from satellite observations, such as those from the SLSTR instrument [12] aboard Sentinel-3 [18]. Once the datacubes are retrieved, they can be manipulated according to user requirements, which often involve filtering along one or more dimensions—such as spatial, temporal, or spectral—by specifying the desired *extents* (i.e., ranges of values within those dimensions).

A variety of software solutions for analyzing EO data have been developed, including relational database extensions (e.g., PostGIS, an extension of PostgreSQL), Python packages (e.g., rasterio and xarray), and array databases like Rasdaman [7]. However, none of these approaches leverages the capabilities of the graph data model. Conversely, while KGs have shown potential for EO analysis, their use in this context has not yet been standardized.

Efforts towards such standardization are currently underway through the *Spatial Data on the Web Working Group* [31,32]. One of the key challenges lies in determining the appropriate level of granularity for representing raster data. At the two extremes of the so-called *spectrum of linkiness*, one can either publish every EO data point as RDF triples — which is highly resource-intensive — or restrict the publication to the metadata about the sources, which offers limited informational value. An intermediate approach, called *dynamically-generated RDF*, focuses instead on generating only the portion of the data cube that is necessary to answer users' requests. However, even this approach may result in materializing a large volume of RDF when queries are complex, impacting performance.

Other proposals leverage on GeoSPARQL [11], the current standard for representing and querying *vector data*[2] in KGs. GeoSPARQL+ [22] implements via Apache Jena [2] an extension of GeoSPARQL that can handle raster operations

[2] Vector data represents geographic features using points, lines, and polygons, while raster data represents features as a grid of cells or pixels, each with a specific value.

and map individual EO observations to IRIs. GeoLD [1] introduces its own syntax extensions and employs a SPARQL optimization engine built on the Apache Jena ARQ engine, in conjunction with a Web Coverage Processing Service (WCPS) node backed by a Rasdaman array database. Unlike GeoSPARQL+, GeoLD supports a dynamically generated RDF strategy, which allows it to achieve better performance in terms of query response times.

Another approach that has been investigated in the literature proposes leveraging the VKG paradigm to eliminate the need to transform and physically store raster data as RDF. This allows users to retrieve the EO data at the desired level of granularity, without the overhead of materializing or duplicating the underlying raster as RDF triples. One such proposal [21] relies on PL/pgSQL, the PostgreSQL Procedural Language, to first transform the data into a tabular format and then expose it as a VKG. The approach involves an expensive computational step of calculating all needed indicators for each datacube cell, significantly increasing both the storage requirements and processing time. Moreover, in case a user requires a new indicator, the database must be re-created. So, while technically this is an approach based on the VKG paradigm, it still requires substantial (tabular) materialization and introduces data duplication.

OntoRaster [19] leverages the VKG paradigm to access EO data by using stored procedures written in PL/Python, the Python procedural language for PostgreSQL, to access EO data stored in a Rasdaman array database. In OntoRaster, PL/Python is used to transform EO data from Rasdaman into a tabular format, which is then mapped to a VKG. Compared to [21], this approach offers two main advantages: it eliminates the need to pre-compute all required indicators, and it allows for faster data ingestion due to Rasdaman's specialization in handling raster data. However, a key limitation is that data must first be downloaded from an external backend (e.g., openEO) and then imported into Rasdaman, resulting in data duplication when the source is not a Rasdaman database. Additionally, supported aggregation operations are limited to those available in RasQL (i.e., `min`, `max`, `count`, `sum`, `avg`) and Rasdaman's native clipping functionality, which limits the system's ability to leverage the full capabilities of more expressive APIs like openEO, which supports over 100 functions.

The Plato Semantic Data Cube System [8] demonstrates yet another approach of VKG over raster data, which leverages the Python xArray package to directly retrieve data from raster files in NetCDF of ZARR format. Hence, contrarily to the other two solutions, Plato does not require the data to be preloaded into a database. Plato adopts a caching mechanism over spatial indexes to improve query answering performance over vector and raster data. A custom *Raptor Join* [30], which efficiently filters raster data that overlap with vector data, avoids any conversion costs between the data. The approach provides an original solution to joining vector-raster data in the context of VKGs. However, a new prototype solution for geospatial joins is unlikely to be as efficient as mature geospatial solutions, such as the raster extension in PostGIS or Rasdaman, and Plato has not been benchmarked with traditional geospatial systems.

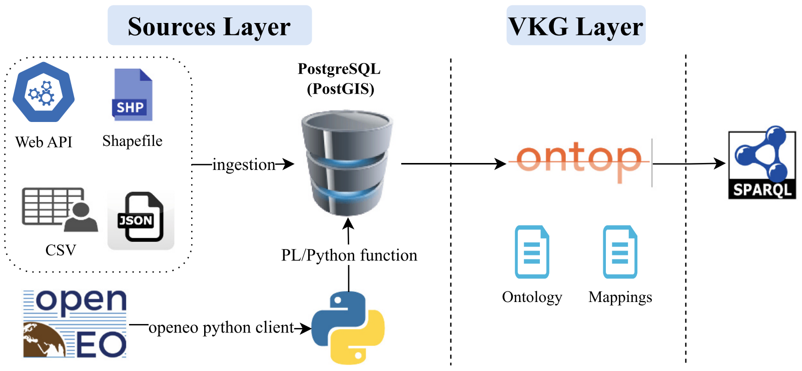


Fig. 2. Architecture of ontopEO.

3 Resource Architecture

In this section, we describe the architecture of the resource we have developed, called ontopEO, as illustrated in Fig. 2. The architecture is organized in two layers: the *Sources Layer*, which interfaces with the underlying data sources, and the *VKG Layer*, which provides integrated SPARQL access to these sources using the VKG paradigm via the Ontop system.

3.1 Sources Layer

Data Sources. The openEO ecosystem consists of several data and infrastructure providers (commonly referred to as *cloud providers*), each hosting its own instance of the openEO Web API. These providers offer computing resources, storage capacity, and the services necessary to support large-scale EO data processing. Two prominent providers are *Copernicus* [16], hosted by *Copernicus Data and Information Access Services* (DIAS), and *openEO EURAC* [15], managed by the *Institute for Earth Observation* at *EURAC Research*. The EURAC instance provides access to Sentinel satellite data, with a particular focus on the South Tyrol region in Italy, as well as other regional datasets. For the development, testing, and evaluation of our ontopEO system, we use the Copernicus backend, which is part of the Copernicus Data Space Ecosystem. Copernicus offers free and open access to a wide range of EO datasets, including global coverage from the Sentinel satellite constellation. We selected Copernicus for its scalability, reliability, and strong institutional backing as a long-term initiative supported by the European Union.

As illustrated in Fig. 2, our approach supports not only raster datacubes from openEO, but also geospatial and generic tabular data, such as datasets regularly published by government agencies. For our experiments, we use municipal boundary *Shapefiles* obtained from the regional geocatalogue [6] of South Tyrol.

PostgreSQL and PL/Python. At the core of our architecture is PostgreSQL 17, a robust relational database, extended with PostGIS 3.5.0 to support geospatial capabilities. PostGIS is the widely adopted geospatial extension for PostgreSQL, enabling the storage, querying, and processing of both vector and raster geospatial data. It is one of the most popular[3] open-source geospatial extensions for relational databases.

Vector geometries can be imported from standard formats such as Shapefiles (`.shp`)—using the `shp2pgsql` loader. Moreover, the *Geospatial Data Abstraction Library (GDAL)* and its `ogr2ogr` utility facilitate reliable import of a wide range of geospatial data formats, including *GeoJSON* and *GPKG*. These capabilities make PostgreSQL combined with PostGIS a natural choice for federating non-raster geospatial and tabular data within our system.

In our architecture, we use PL/Python to interact with the openEO API via the official openEO Python client [27], a well-maintained library for accessing openEO services. This integration offers enhanced flexibility by enabling the use of any Python package (or combination of packages) within the database backend. Our implementation is based on Python 3.11 and openEO version 0.33.0 to connect to selected openEO cloud backends. Additionally, we employ Shapely 2.0.2 to convert geometries from the well-known text (wkt) format into representations compatible with openEO.

3.2 VKG Layer

The platform used to construct the VKG is *Ontop v5.2.1*, which provides query answering capabilities over VKGs by relying on relational databases such as PostgreSQL as a data source. Ontop fully supports all W3C standards relevant to VKGs, including the RDF data model, the SPARQL query language, the OWL 2 QL [24] profile of the OWL ontology language, and the R2RML [13] mapping language. Ontop operates by taking as input an ontology that defines the domain knowledge, along with a mapping file that links the relational tables in PostgreSQL/PostGIS to the ontology. Table 1 shows a database table excerpt containing data about municipalities in South Tyrol, while the listing below shows a corresponding Ontop mapping that relates such data to suitable classes and properties at the ontology level.

```
mappingId   south_tyrol_vector_data
target      :region/{gid} a :Municipality ;
              geo:asWKT {geom1}^^geo:wktLiteral ;
              rdfs:label {name_it}@it .
source      SELECT gid, name_it, ST_AsText(geom) AS geom1
            FROM region_south_tyrol
```

More specifically, the table `region_south_tyrol` contains vector data for all municipalities in South Tyrol, including their administrative boundaries. The attributes used in the mapping include: `gid`, which serves as a unique identifier

[3] PostGIS has consistently ranked as the top spatial relational database in DB-Engines https://db-engines.com/en/ranking/spatial+dbms.

Table 1. South Tyrol table excerpt, with geometries represented in wkbs.

gid	name_de	name_it	geom
1	Bozen	Bolzano	010300FF...
2	Meran	Merano	F0BF0000...
3	Brixen	Bressanone	AA0DF0E0...

for each municipality; `name_it`, which contains the municipality's name in Italian; and `geom`, which stores the geometry in well-known binary (wkb) format. The PostGIS function `ST_AsText` is used to convert geometries from wkb to the more widely adopted wkt format, which is better suited for use in RDF representations. The example assumes the existence of a class `:Municipality` in the ontology, along with the data properties `geo:asWKT` and `rdfs:label`.

To fully implement our architecture, we extended the Ontop VKG system to accommodate the specific characteristics of the openEO API — particularly its support for complex processing pipelines expressed as so-called *process graphs.* The next section provides a detailed explanation of this extension, which constitutes the core of our contribution.

4 Ontop Implementation

4.1 Implementation of openEO Processes as SPARQL Functions

We extended Ontop by implementing a set of SPARQL functions mapped to their corresponding openEO processes, along with a rewriting mechanism that composes complex function chains into a single openEO process graph. This enables SPARQL queries to emulate process graphs — typically authored through the openEO Python client or web interfaces — by composing SPARQL functions directly within the query. Our implementation reformulates such queries into a single JSON-based process graph that encapsulates all openEO-related operations before dispatching it to the openEO backend for execution. This approach significantly reduces execution latency by minimizing round-trip interactions with the backend, and constitutes a key distinguishing feature of our system compared to existing proposals in the literature.

Ontop offers the flexibility to define any number of custom SPARQL functions with arbitrary arity, argument types, and return datatypes. SPARQL also supports function overloading, enabling multiple versions of a function to vary by input type or count. This is useful for mapping openEO processes, such as overloading `load_collection` to optionally include cloud cover filtering.

Ontop requires that the RDF datatype of every function's output is explicitly declared in advance. However, since the result of an openEO process graph may vary, returning either a datacube or a scalar value depending on the operations performed, the output type cannot always be determined at query time. To address this, all results are published with the generic datatype `xsd:string`.

Table 2. Implemented openEO SPARQL functions.

`apply`	`mask`
`apply_dimension`	`ndvi`
`apply_kernel`	`merge_cubes`
`band_math`	`oneof`
`filter_bbox`	`filter_bands`
`linear_scale_range`	`aggregate_temporal_period`
`load_collection`	`aggregate_spatial`
`reduce_dimension`	`rename_labels`
`sar_backscatter`	`to_scl_dilation_mask`

Users can then apply standard SPARQL cast functions, such as `xsd:double`, to convert the result into the appropriate datatype as needed. Note that, as discussed in Sect. 2, there is still no standardized approach for encoding rasters within the RDF framework. Consequently, also all raster or array outputs in our system are represented using the generic RDF datatype `xsd:string`, where the structure and dimensionality are indicated by the nesting of square brackets.

Table 2 lists all openEO functions implemented in our Ontop extension. For readability, the shared prefix `openeo:` (corresponding to the namespace `http://www.openeo-ontop.org#`) is omitted from each function name.

Most of the Ontop functions closely align with their corresponding openEO processes in a one-to-one manner. Exceptions include `band_math` and `oneof`, which encapsulate band operations and logical `OR` conditions, respectively. These were introduced to simplify the syntax of queries that involve the openEO `apply` process. Mathematical operations and boolean comparisons are expressed using standard SPARQL syntax and are translated into the corresponding openEO functions during query rewriting.

4.2 SQL Translation

In VKGs, query answering is typically achieved through query reformulation techniques. In Ontop, this means translating SPARQL queries over the VKG into equivalent SQL queries over the underlying relational data sources. In our case, however, reformulation must go beyond SQL translation: it must also include generating calls to the openEO API. Moreover, multiple SPARQL function calls related to openEO within a single query need to be grouped and composed into a unified openEO process graph, so as to take advantage of the expressivity of the openEO API and reduce round-trip interactions.

To explain our solution, we rely on a running example inspired by Copernicus. Consider the following SPARQL query, asking for the average *normal-*

ized difference vegetation index[4] (NDVI) for a specific time period and over the municipality of Bolzano:

```
PREFIX openeo: <http://www.openeo-ontop.org#>
SELECT ?average_NDVI {
  ?g a :Municipality;
    rdfs:label "Bolzano"@it ;
    geo:asWKT ?geom.
  BIND ("SENTINEL2_L2A" AS ?satellite_instrument) .
  BIND ("2025-04-05T00:00:00Z"^^xsd:dateTime AS ?start_time) .
  BIND ("2025-04-15T00:00:00Z"^^xsd:dateTime AS ?end_time) .
  BIND ("[B04, B08]" AS ?band) .
  BIND (openeo:load_collection(?satellite_instrument, ?geom,
        ?start_time, ?end_time, ?band) AS ?cube1) .
  BIND (openeo:ndvi(?cube1) AS ?cube2) .
  BIND (openeo:reduce_dimension(?cube2, "t", "mean") AS ?cube3).
  BIND (openeo:aggregate_spatial(?cube3, ?geom, "mean") AS
        ?average_NDVI) . }
```

Although simple, the query above already illustrates the benefit of using an ontology that operates at a higher level of abstraction than raw raster datacubes in openEO. In a standard openEO workflow, the user must manually specify the area of interest as a polygon when loading a data collection. In contrast, our integrated approach leverages the database to automatically retrieve the relevant geometry, allowing users to express queries in terms of semantic entities, such as municipalities, instead of low-level spatial coordinates.

Starting from this query, our tool produces the following translation to SQL:

```
SELECT ontop_openeo.process_graph_function(TO_JSONB(ARRAY['id_4
    ', 'aggregate_spatial', 'from_node_id_3', ST_ASTEXT(v1.geom)
    , 'mean', 'id_3', 'reduce_dimension', 'from_node_id_2', 't',
     'mean', 'id_2', 'ndvi', 'from_node_id_1', 'id_1', '
    load_collection', 'SENTINEL2_L2A', ST_ASTEXT(v1.geom),
    '2025-04-05T00:00:00Z', '2025-04-15T00:00:00Z', '[B04, B08
    ]']))
FROM region_south_tyrol v1
WHERE 'Bolzano' = v1.name_it;
```

Function `process_graph_function` (which is part of the `ontop_openeo` schema) is a PL/Python function responsible for generating an openEO process graph from a structured JSON representation. This JSON object is constructed using a SQL array that encodes the sequence and hierarchy of SPARQL openEO function calls issued in the query. Each segment of the array contains identifiers, function names, arguments, and dependency links (via `from_node_id`) that together define the topology and logic of the full process graph. The function interprets this linearized representation and builds a valid openEO process graph in JSON format, which is then sent to the openEO backend for execution.

In the following, we detail the generation of the process graph in PL/Python.

4.3 Generation of the Process Graph

The execution of the PL/Python function `process_graph_function` is triggered by Ontop, which provides as input a JSON array encoding the openEO process graph in a linearized form.

[4] Commonly used vegetation index that measures the health and density of vegetation using the difference between near-infrared and red light reflected by the surface.

The function interprets this flat array by scanning for process identifiers prefixed with `id_`*n*, which are considered as delimiters of segments in the array. Each such segment encodes a distinct step in the process graph and is mapped to a predefined template associated with the corresponding openEO process (whose name is given by the element immediately following `id_`*n*). For each segment, the function creates a portion of the process graph by instantiating the template corresponding to that segment. For example, the `load_collection` process is matched to the following template:

```
arg1: {
  "process_id": arg2,
  "arguments": {
    "id": arg3,
    "spatial_extent": { arg4 },
    "temporal_extent": [ arg5, arg6 ],
    "bands": [ arg7 ]
  }
}
```

Placeholders `arg1`, ..., `arg7` are replaced with the seven arguments for `load_collection` (elements from `id_1` onwards). During this process, geometry inputs are converted from wkt to the GeoJSON format required by openEO using the `shapely` Python library. Once all templates are instantiated, the individual components are assembled into a single[5], coherent process graph, which is then submitted to the openEO backend for execution. For our running example, the resulting process graph is:

```
{
  "process_graph": {
    "id_1": {
      "process_id": "load_collection",
      "arguments": {
        "id": "SENTINEL2_L2A",
        "spatial_extent": {
          "type": "Polygon",
          "coordinates": [ [ [ 11.9, 46.6 ], ... ] ]
        },
        "temporal_extent": [ "2025-04-05T00:00:00Z", "2025-04-15
            T00:00:00Z"],
        "bands": ["B04", "B08"]
      }
    },
    "id_2": {
      "process_id": "ndvi",
      "arguments": {
        "data": {
          "from_node": "id_1"
        }
      }
    },
  ...
}
```

Directly translating every function into a fixed template is not always feasible. Some openEO processes, such as `apply`, involve more complex logic — including

[5] Naturally, if the function is invoked multiple times — e.g., to retrieve data for several municipalities with different geometries — multiple process graphs will be constructed and corresponding requests will be issued to the openEO backend.

nested comparisons and mathematical operations — where the structure and order of operations can vary significantly. For example, `openeo:apply(?c > 300 || ?c < 280)` requires the use of a combination of 3 subprocesses, the comparison operators `gt` and `lt`, and the logical operator `or`. To handle such cases, we use specialized templates that are dynamically selected based on the operations involved. Specifically for `apply`, we delegate its construction to a dedicated PL/Python helper function, invoked by the main process graph generator, which encapsulates the logic required to correctly assemble the corresponding subgraph.

Custom Operations. In real-world openEO workflows (especially those implemented in Python) users often incorporate additional computations via user-defined functions (UDFs). These custom functions allow for the execution of specialized logic that goes beyond standard openEO processes. Although the use of UDFs is generally recommended as a “last resort”, our experience indicates that they are widely adopted in practice.

Our approach naturally accommodates UDFs: they can be embedded directly within a SPARQL query using the `BIND` construct, treating them as plain strings. However, embedding UDFs inline can quickly become cumbersome, especially for frequently reused or more complex custom transformations. In such cases, defining the UDF as a dedicated PL/Python function proves to be a more scalable and maintainable solution. This allows users to extend the system by integrating new operations or enhancing existing process templates with additional logic encapsulated in separate PL/Python functions.

For example, in our experiments, we implemented a custom Gaussian kernel[6] generator, that can be used, e.g., with the `apply_kernel` openEO process. This user-defined function constructs a fixed-size kernel with a predefined standard deviation, which is then injected into the process graph at runtime.

5 Experiments and Evaluation

We demonstrate the applicability of our solution by running a set of experiments over a variety of real-world queries.

5.1 Setup and Data

The experiments were conducted using a Lenovo V15 G4 IRU machine running Ubuntu 24.04.2 LTS with a 13th Gen Intel(R) Core(TM) i5-13420H CPU, 16GB of RAM and 512GB disk capacity. To comprehensively evaluate openEO queries using real world phenomena like heatwaves and wildfires, we executed openEO queries across diverse geographic regions, specifically selecting study areas in South Tyrol and Campania, Italy, and the Netherlands. Italian geospatial vector data with administrative boundaries for South Tyrol and Campania are

[6] A Gaussian kernel is a matrix used to reduce noise or detail in raster data.

retrieved respectively from the local GeoBrowser MapView [5] and the Geoportale Regione Campania [29]. We also retrieve population data for the South Tyrol municipalities from the regional statistical unit ASTAT [4]. The Dutch National Georegister [23] provides administrative data on the Netherlands including its municipalities, districts, and neighborhoods. In cases where direct connections to other datasets are unavailable, such as queries regarding oil spills, we demonstrate the adaptability of openEO by employing the geographic coordinates of the affected region as part of the query. This exemplifies that spatial information does not need to be exclusively sourced from a database.

5.2 Queries

Copernicus provides a set of Python notebooks [17] that showcase the capabilities of openEO to execute queries for real-world use cases. We use these examples to design 8 queries to test the expressivity and performance of our resource. In order to demonstrate the value of integrated queries over a VKG using openEO, our VKG integrates the raster datacubes from openEO with the relational datasets mentioned in Sect. 5.1. Our queries, written in SPARQL, are described below and are part of the online resource.

Q1. Find the NDVI for the municipalities in the Val Venosta district of South Tyrol between 15 April 2025 and 25 April 2025.
Q2. Find heatwave data in neighborhoods of the municipality of Krimpen aan den IJssel in the Netherlands between 1 June 2023 and 30 October 2023.
Q3. Find wildfire data for the municipalities of Val Venosta in South Tyrol and impacted population between 15 April 2025 and 25 April 2025.
Q4. Find landslide data by municipality for the island of Ischia, Campania, Italy by comparing data from 25 August 2022 to 25 November 2022 with data from 26 November 2022 to 25 December 2022.
Q5. Find the radar vegetation index (RVI) by municipality of South Tyrol between 15 April 2025 and 25 April 2025.
Q6. Find oil spill data in southern coast of Kuwait near the resort community of Al Khiran (reported in 2017).
Q7. Retrieve NO2 emissions over Bolzano, Italy, for periods June 2020–June 2021 vs. June 2022–June 2023 (to assess COVID impact).
Q8. Find surface soil moisture values in Bolzano from 1 to 3 September 2023.

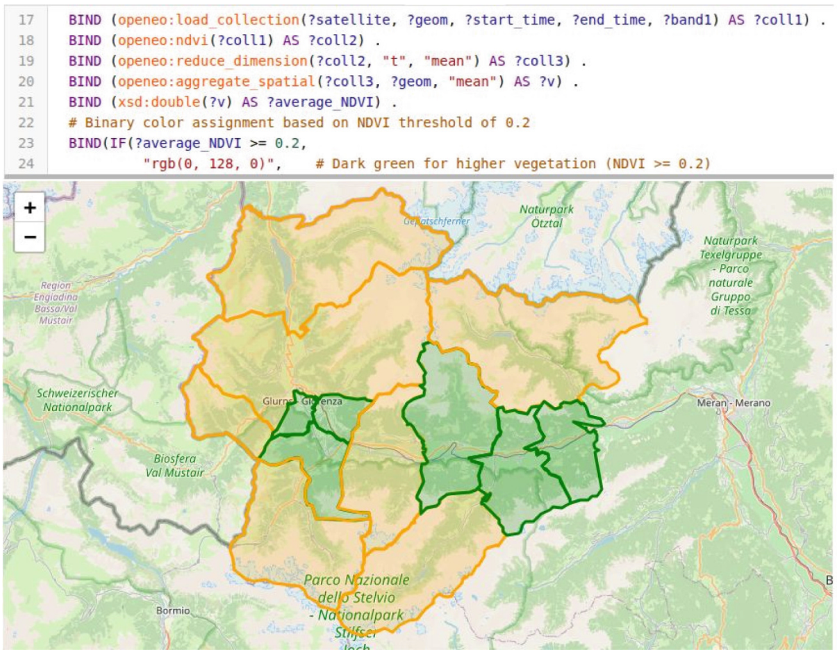
Figure 3 provides a visual representation of the results of query **Q1** in ontopEO.

5.3 Results and Discussion

Table 3 reports the response times for ontopEO over the eight queries from our use-case. We remark that ours is the first solution that enables interrogation of openEO Web APIs integrated with relational data via a VKG. Therefore, it was not possible to compare other approaches with our solution while exploiting its full expressiveness.

Table 3. Query response time (seconds) / number of records.

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
ontopEO	8760/13	2130/6	2036/13	995/5	1271/6	152/1	859/1	174/1



```
BIND (openeo:load_collection(?satellite, ?geom, ?start_time, ?end_time, ?band1) AS ?coll1) .
BIND (openeo:ndvi(?coll1) AS ?coll2) .
BIND (openeo:reduce_dimension(?coll2, "t", "mean") AS ?coll3) .
BIND (openeo:aggregate_spatial(?coll3, ?geom, "mean") AS ?v) .
BIND (xsd:double(?v) AS ?average_NDVI) .
# Binary color assignment based on NDVI threshold of 0.2
BIND(IF(?average_NDVI >= 0.2,
        "rgb(0, 128, 0)",    # Dark green for higher vegetation (NDVI >= 0.2)
```

Gepatschferner
Naturpark Ötztal
Naturpark Texelgruppe - Parco naturale Gruppo di Tessa
Region Engiadina Bassa/Val Müstair
Schweizerischer Nationalpark
Glurns Glorenza
Meran - Merano
Biosfera Val Müstair
Parco Nazionale dello Stelvio - Nationalpark Stilfser Joch
Bormio

Fig. 3. Result of **Q1**. Municipalities with NDI > 0.2 are displayed in green.

Execution times show that ontopEO successfully answers all queries, although durations vary significantly — from approximately 2 min for **Q6** to about 2.5 h for **Q1**. The relatively short execution time of **Q6** is due to the simplicity of the query, which returns only a single record defined by a bounding box. In contrast, **Q1** involves 13 municipalities, each described by complex polygon geometries, which increases processing time considerably.

The number of records is, of course, not the sole determinant of query complexity. For example, query **Q2**, which involves the Dutch National Georegister, exhibits execution times comparable to query **Q3**, despite involving fewer municipalities. This is primarily due to the fact that Dutch municipality polygons in our dataset are significantly more detailed (because the municipalities are bigger), typically containing about three times as many points as their Italian counterparts, thereby resulting in similar computational overhead. Another example is **Q7**, which involves a single but very complex polygon (2867 points) and gathers data over a long time period (2 years).

A further remark concerns **Q1** and the visualization shown in Fig. 3. When the query is the explicit cast and the binary color assignment, response time drops significantly — from 8760 s to 1172 s. This performance difference is not inherent to our approach, but rather stems from the way Ontop translates the cast and the IF-statement into SQL, relying on a costly nested `CASE` expression.

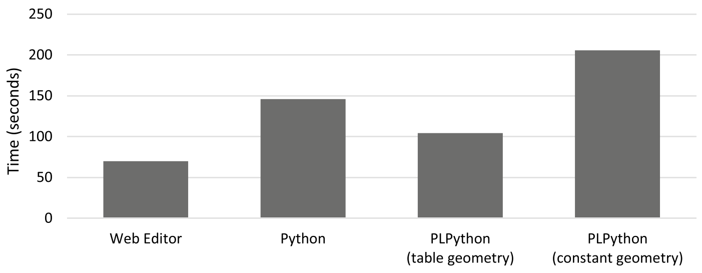


Fig. 4. Comparison on **Q1**, restricted to the area of Bolzano.

We conducted an additional experiment to assess the overhead of our approach compared to direct execution via the openEO API. Since openEO lacks multi-source integration, we simplified our queries to operate on a single hard-coded geometry. Figure 4 reports response times for this streamlined version of **Q1**. As expected, the Copernicus Web Editor delivers the fastest execution, followed by the Python API. Our PL/Python-based approach adds further overhead. However, when geometries are accessed directly from database tables (as intended in the typical usage of our tool) PL/Python can outperform standalone Python. This appears due to reduced data transfer and in-memory access to binary-stored geometries. Notably, clearing the cache yields similar performance to using constant geometries, suggesting in-memory access is the key factor.

Overall, these results demonstrate that `ontopEO` can effectively support a variety of real-world EO queries, as posed by domain experts, while highlighting the need for further optimizations if the query results require further post-processing.

5.4 Limitations

This section is divided into two parts: first, we discuss the limitations of the proposed solution as identified through the evaluation; second, we outline the limitations inherent in the evaluation process itself.

Limitations of the Proposed Solution. Our solution has currently a number of limitations:

- *SPARQL function return type.* The return type of openEO SPARQL functions is unknown at query time. For instance, the function `openeo:reduce_dimension` can return a raster or a scalar. Consequently, all openEO SPARQL functions return `xsd:string`, and specific types require an explicit cast.

- *Expressivity.* Our resource only implements the most common processes of openEO. Support can be extended by introducing new SPARQL openEO functions and templates, however this might require substantial effort depending on the complexity of the openEO processes being added (this holds true especially for complex *User Defined Processes* (UDPs) [28]).
- *PL/Python Overhead.* PL/Python introduces context-switching overhead when calling external APIs like openEO, due to interpreter switching, session resource management, and result serialization. It also lacks support for parallelism. As our experiment against Python and the openEO Web interface has shown, this overhead is mitigated when geometries are already serialized in a database, rather than provided as constants in the query.
- *Request rate API limits.* Providers of openEO data have their own API request limits that constrain the number of SPARQL queries that can be executed. For example, concurrent API requests via Copernicus are limited to 2.

Limitations of the Evaluation. The evaluation we have carried out has the following limitations:

- *Comparison with Other Systems.* To the best of our knowledge, our approach is the first to translate SPARQL queries into openEO API calls. As such, there is currently no directly comparable system. One potential point of comparison is Rasdaman; however, this would ultimately amount to comparing openEO and Rasdaman themselves — two fundamentally different paradigms (API-based orchestration versus database management). Moreover, as outlined in Sect. 2, integrating full-scale Copernicus raster datasets into Rasdaman presents significant technical challenges, further complicating any direct comparison.
- *Scalability Analysis.* Our evaluation includes a diverse set of SPARQL queries, varying in the number of municipalities involved and the complexity of the spatial geometries — from simple bounding boxes to intricate regional shapes. In the absence of standardized benchmarks tailored to this use case, we developed a custom evaluation grounded in realistic scenarios and informed by domain expert requirements. Unlike evaluations based on synthetic benchmarks (e.g., LUBM [20] or BSBM [9]), our approach prioritizes practical applicability through the use of real-world data. However, a limitation of this methodology is that it does not systematically assess how performance scales with increasing data volume — a key strength of synthetic benchmarks, which allow for controlled, repeatable scalability testing.

6 Conclusions and Future Work

We introduced `ontopEO`, a tool that leverages the VKGs paradigm to query EO data via openEO. Our approach reformulates SPARQL queries, enriched with openEO functions, into process graphs that can be executed by openEO providers. ontopEO can answer typical Copernicus-related questions over a higher-level conceptualization enabled by a domain ontology.

The tool has been developed and deployed within the context of the EFRE 1078 project CRIMA, which aims at the realization of an ontology-based decision support system for climate risk mitigation and adaptation. The initiative reflects a concrete demand from the EO community to augment satellite observation capabilities with semantic technologies. The project is coordinated by the Eurac Institute for Earth Observation, a primary maintainer of the openEO API.

Although the tool is in an early stage and has not yet established a user community, it is being developed in a dedicated branch of the widely adopted Ontop system. Integration into the main Ontop repository is planned as the tool matures, in line with established Ontop development practices.

For future work, we plan to support additional openEO processes, including provider-specific UDPs. Another promising direction is the use of advanced mapping languages like RML, which supports mappings over Web APIs. Notably, the RML-FNML [3] extension of RML allows for declarative function definitions using the *Function Ontology* (FnO) [14], offering a more flexible alternative to hard-coded functions. Embedding openEO functions directly in mappings, rather than only in SPARQL, could enable higher-level abstractions over raster data. For instance, concepts like "Wildfire" could be defined ontologically and instantiated through relevant observations.
Resource Availability Statement: The resource is publicly released under the CC-BY-SA-4.0 license, and it is made available together with the material and instructions for replication at https://github.com/apano-on/ontopEO.

Acknowledgments. This work has been carried out while Albulen Pano was enrolled in the Italian National Doctorate on Artificial Intelligence run by Sapienza University of Rome in collaboration with Free University of Bozen-Bolzano. This work was partially supported by the Agenzia per la cybersicurezza nazionale under the programme for promotion of XL cycle PhD research in cybersecurity – B83C24005640005. The views expressed are those of the authors and do not represent the funding institution. This work has also been partially supported by the HEU project CyclOps (GA n. 101135513), by the Province of Bolzano and FWF through project OnTeGra (DOI 10.55776/PIN8884924), by the Province of Bolzano and EU through projects ERDF-FESR 1078 CRIMA, and ERDF-FESR 1047 AI-Lab, by MUR through the PRIN project 2022XERWK9 S-PIC4CHU, and by the EU and MUR through the PNRR project PE0000013-FAIR.

Disclosure of Interests. The authors have no competing interests.

References

1. Almobydeen, S.B., Viqueira, J.R., Lama, M.: GeoSPARQL query support for scientific raster array data. Comput. Geosci. **159**, 105023 (2022). https://doi.org/10.1016/j.cageo.2021.105023
2. Apache Software Foundation: Apache Jena. https://jena.apache.org. Accessed 14 May 2025
3. Arenas-Guerrero, J., et al.: An RML-FNML module for python user-defined functions in Morph-KGC. SoftwareX **26**, 101709 (2024). https://doi.org/10.1016/j.softx.2024.101709
4. ASTAT – Provincial Institute of Statistics of South Tyrol: ASTAT – Statistics Portal. https://astat.provincia.bz.it. Accessed 14 May 2025
5. Autonomous Province of Bolzano – South Tyrol: Civis GeoBrowser MapView. https://maps.civis.bz.it, accessed: 14-05-2025
6. Autonomous Province of Bolzano – South Tyrol: GeoNetwork South Tyrol. https://geonetwork1.civis.bz.it/geonetwork/geonetwork/ita/catalog.search#/home. Accessed 14 May 2025
7. Baumann, P., Furtado, P., Ritsch, R., Widmann, N.: The RasDaMan approach to multidimensional database management. In: Proceeding of the 12th ACM Symposium on Applied Computing (SAC), pp. 166–173 (1997). https://doi.org/10.1145/331697.331732
8. Bilidas, D., et al.: The semantic data cube system Plato and its applications. In: Proceedings of the 44th IEEE International Symposium Geoscience and Remote Sensing (IGARSS), pp. 2514–2518. IEEE Computer Society (2024). https://doi.org/10.1109/IGARSS53475.2024.10640737
9. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. Int. J. Semantic Web Inform. Syst. **5**(2), 1–24 (2009). https://doi.org/10.4018/JSWIS.2009040101
10. Calvanese, D., et al.: Ontop: answering SPARQL queries over relational databases. Semantic Web J. **8**(3), 471–487 (2017). https://doi.org/10.3233/SW-160217
11. Car, N.J., et al.: OGC GeoSPARQL - a geographic query language for RDF data. OGC Implementation Standard OGC 22-047, Open Geospatial Consortium (2023). http://www.opengis.net/doc/IS/geosparql/1.1
12. Copernicus: Sentinel-3 SLSTR Instrument Overview. https://sentiwiki.copernicus.eu/web/s3-slstr-instrument. Accessed 14 May 2025
13. Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF mapping language. W3C Recommendation, World Wide Web Consortium (Sept 2012). https://www.w3.org/TR/r2rml/
14. De Meester, B., Dimou, A.: The Function Ontology. https://fno.io/. Accessed 14 May 2025
15. Eurac Research: openEO Project at Eurac Research. https://www.eurac.edu/en/institutes-centers/institute-for-earth-observation/projects/openeo. Accessed 14 May 2025
16. European Space Agency (ESA): Copernicus Data Space - openEO Interface. https://dataspace.copernicus.eu/analyse/openeo. Accessed 14 May 2025
17. European Space Agency (ESA): Copernicus Data Space – Use Case Documentation. https://documentation.dataspace.copernicus.eu/Usecase.html. Accessed 14 May 2025
18. European Space Agency (ESA): Sentinels - Copernicus. https://sentinels.copernicus.eu. Accessed 14 May 2025

19. Ghosh, A., Pano, A., Xiao, G., Calvanese, D.: OntoRaster: Extending VKGs with raster data. In: Proc. of the 8th Int. Joint Conf. on Rules and Reasoning (RuleML+RR). Lecture Notes in Computer Science, vol. 15183, pp. 108–123. Springer (2024). https://doi.org/10.1007/978-3-031-72407-7_9
20. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for OWL knowledge base systems. J. Web Semantics **3**(2–3), 158–182 (2005)
21. Hamdani, Y., Xiao, G., Ding, L., Calvanese, D.: An ontology-based framework for geospatial integration and querying of raster data cube using virtual knowledge graphs. ISPRS Int. J. Geo-Inform. **12**(9), 375 (2023). https://doi.org/10.3390/ijgi12090375
22. Homburg, T., Staab, S., Janke, D.: GeoSPARQL+: Syntax, semantics and system for integrated querying of graph, raster and vector data. In: Pan, J.Z., Tamma, V., d'Amato, C., Janowicz, K., Fu, B., Polleres, A., Seneviratne, O., Kagal, L. (eds.) Proc. of the 19th Int. Semantic Web Conf. (ISWC), pp. 258–275. Springer (2020)
23. Kadaster (Netherlands): Nationaal Georegister. https://nationaalgeoregister.nl. Accessed 14 May 2025
24. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 Web Ontology Language profiles (second edition). W3C Recommendation, World Wide Web Consortium (Dec 2012). https://www.w3.org/TR/owl2-profiles/
25. Open Geospatial Consortium: OGC API – Processes: Part 1: Core. https://portal.ogc.org/files/106981 (2024), draft Standard. Accessed 10 Dec 2024
26. OpenEO: OpenEO: An Open Earth Observation Processing Platform. https://openeo.org/ (2024). Accessed 10 Dec 20244-12-10
27. openEO Consortium: openEO Python Client. https://open-eo.github.io/openeo-python-client/. Accessed 14 May 2025
28. openEO Consortium: User-Defined Processes – openEO Python Client. https://open-eo.github.io/openeo-python-client/udp.html. Accessed 14 May 2025
29. Regione Campania: Campania Regional Geoportal (SIT). https://sit2.regione.campania.it. Accessed 14 May 2025
30. Singla, S., Eldawy, A., Diao, T., Mukhopadhyay, A., Scudiero, E.: The raptor join operator for processing big raster + vector data. In: Proc. of the 29th International Conference on Advances in Geographic Information Systems, pp. 324–335. SIGSPATIAL '21, Association for Computing Machinery (2021). https://doi.org/10.1145/3474717.3483971
31. W3C Education and Outreach Working Group: How to Make Your Data Easier to Use with Quality Metadata. https://www.w3.org/TR/eo-qb/ (2023), w3C Working Group Note. Accessed 10 Dec 2024
32. W3C/OGC Spatial Data on the Web Working Group: Spatial Data on the Web Working Group. https://www.w3.org/2021/sdw/ (2021). Accessed 10 Dec 2024
33. Xiao, G., et al.: The virtual knowledge graph system Ontop. In: Proceedings of the 19th International Semantic Web Conference (ISWC). Lecture Notes in Computer Science, vol. 12507, pp. 259–277. Springer (2020). https://doi.org/10.1007/978-3-030-62466-8_17