PLATO: A SEMANTIC DATA CUBE IMPLEMENTATION USING ONTOLOGY-BASED DATA ACCESS TECHNOLOGIES

Dimitris Bilidas, Anastasios Mantas, Filippos Yfantis, George Stamoulis and Manolis Koubarakis

Dept. of Informatics and Telecommunications
National and Kapodistrian University of Athens, Greece

ABSTRACT

We present Plato, the first semantic data cube implementation which uses ontology-based data access technologies and, in particular, the system Ontop. We present the architecture of Plato and an evaluation of its performance using datasets from the use cases of the Horizon 2020 project DeepCube.

Index Terms— semantic data cubes, ontologies, ontology based data access, Ontop

1. INTRODUCTION

A *data cube* is a multidimensional array of values. It is a natural data structure for storing analysis-ready Earth observation (EO) data and other kinds of multidimensional data. Due to this, a number of data cube infrastructures targeting EO data have been developed recently (e.g., the Open Data Cube infrastructure in Australia, the Euro Data Cube and Earth System Data Cube funded by ESA etc.). These data cube infrastructures offer libraries and APIs (e.g., xarray, YAXArrays) to store and query multidimensional data. However, before data cube infrastructures became a trend, there had already been lots of research and development on *array data base management systems (DBMS)* (e.g., Rasdaman [7], SciDB [6] and MonetDB SciQL [9]) that offer *declarative query languages* for modeling and querying multidimensional data.

The concept of *semantic EO data cubes* (or *semantic data cubes* for simplicity) was first presented by Augustin et al. in [4]. The term *semantic* was used in order to distinguish them from regular EO data cubes, like the ones that can be constructed using the above infrastructures, that contain numbers without high-level meaning for the user (e.g., reflectance values). In semantic data cubes these values are related to *symbolic high-level concepts* based on an interpretation. We can define low-level semantic concepts (e.g., the colour of a pixel) or high-level ones (e.g., the land cover/use of an area). Users can then obtain information about these concepts, but also relate them to the original values. Apart from providing knowledge based on interpretations, a semantic data cube could also facilitate the use of external knowledge (datasets) and allow interlinking with the original values for carrying out a combined analysis. For example, demographic data published by some governmental organization can be used to identify big cities located within some distance from areas that have been classified as pine forests. In their paper [4], Augustin et al. use the Rasdaman array DBMS to implement the idea of a semantic EO data cube. In this paper we go beyond the work of Augustin et al. and make the following original contributions.

We bring the concept of semantic data cubes to its full realization by designing a semantic data cube system using techniques from the area of *geospatial ontology-based data access* [5]. Geospatial ontology-based data access techniques allow one to model geospatial data using *ontologies* and *mappings* from the ontology concepts to the underlying data sources. They also offer the *declarative query language GeoSPARQL* for querying the underlying data sources using concepts of the ontology (e.g., surface temperature), that provide a semantic layer over data cube data. We believe that this is an important extension of the work of Augustin et al. since ontologies, mappings and query languages such as GeoSPARQL are more appropriate for modeling/querying at the level of semantics, than modeling/querying the corresponding concepts of an array DBMS. At the same time, due to the *impedance mismatch* between the concepts of an ontology language (directed graphs of classes, instances, properties and values) and the concepts of data cubes (multidimensional arrays of values) make our task very challenging.

We show how to face the "impedance mismatch" challenges by implementing the semantic data cube system Plato using the well-known ontology-based data access system Ontop [3], Python xarray scripts and PostgreSQL *Foreign Data Wrappers (FDWs)*. During initialization of the Ontop engine, we provide an ontology in the OWL2 QL language and a set of mappings. The mappings define the way that ontology terms are related to the data residing in the backend. After initialization, Ontop is ready to accept GeoSPARQL queries and translate them into SQL enhanced with spa-

This work was supported by the first call for H.F.R.I. Research Projects to support faculty members and researchers and the procurement of high-cost research equipment grant (HFRI-FM17-2351). It was also partially supported by the ESA project DA4DTE (subcontract 202320239), the Horizon 2020 projects AI4Copernicus (GA No. 101016798) and DeepCube (GA No. 101004188), and Horizon Europe project STELAR (GA No. 101070122).

tial operators. The PostGIS backend contains virtual tables based on FDWs. The data cubes are stored in ".zarr" directory format or as `.nc` (netCDF) files, and we utilize Python scripts for efficient access. This is possible through the use of the Python xarray package that allows us to work with labelled multi-dimensional arrays conveniently. To handle access to the local or remote data cubes we use Multicorn, a PostgreSQL 9.1+ extension meant to make FDW development easy though the use of Python. Finally, to reduce execution time of queries over large cubes, we implemented a caching mechanism to minimize on-the-fly transformation from GeoSPARQL to SQL, and Raptor Join to efficiently compute spatial join operations.

We evaluate the performance of Plato with data cubes from the use cases of European Horizon 2020 project DeepCube (https://deepcube-h2020.eu/). Our experiments demonstrate that the optimizations we have developed allow us to process complicated GeoSPARQL queries targeting GBs of data very efficiently.

2. RELATED WORK

Plato is the first implementation of a semantic data cube system based on OBDA technologies. Since background on semantic data cubes was covered in the introduction, now we introduce only OBDA. OBDA is a method for linking an ontology that encodes knowledge about the classes and properties of entities for a given application domain, to underlying data sources. The linking is accomplished through declarative mappings, which are used to generate ontology terms from information in the data sources. Instead of materializing all the ontology terms, users can pose a query over the ontology, and then a process of query transformation is carried out to produce a new query in the native language of the underlying data sources. This query is then executed and the results are transformed as ontology terms to be presented to the user. This approach, also known as virtual knowledge graph approach, has the advantage of providing a familiar vocabulary to the user to pose queries, concealing details about the underlying data sources, such as complex schemas and storage particularities. On the other hand, the process of transforming the initial query over the ontology into a query over the underlying sources, may lead to complex and large queries.

Ontop [3] (https://ontop-vkg.org/) is one of the first OBDA systems able to perform SPARQL to SQL query translation. The inputs for this process are: (i) an ontology in the subset OWL2 QL of the ontology language OWL2 [11], (ii) a database schema, (iii) a set of mapping assertions that generate virtual RDF triples from database values and (iv) an initial SPARQL query over the ontology. The result is an SQL query that is executed on any database instance that follows the input schema, and provides the complete answers with respect to the ontology axioms. Ontop has been successfully deployed in several demanding use cases and has an active community of users and developers, with 154 forks and 549 stars in Github. It has also been commercialized by the Italian company Ontopic (https://ontopic.ai/en/).

Ontop-spatial was developed by our group in 2016 [1, 5] (https://ontop-spatial.di.uoa.gr/). It is the first geospatial OBDA system and it was implemented as a geospatial extension of Ontop. In Ontop-spatial, the input GeoSPARQL query is transformed into an intermediate form based on Datalog, and this query is rewritten by taking into consideration the ontology and the mappings from the ontology-concepts to the data sources. The final result is an SQL query that uses spatial SQL functions, which correspond to the GeoSPARQL functions and operators of the initial query. This SQL query can be executed in a spatially-enabled relational system, like PostGIS (the spatial extension of PostgreSQL) or Spatial-Lite (the spatial extension of SQLite). The functionalities of Ontop-spatial has been integrated fully into Ontop as of version 4.1.0.

3. THE ARCHITECTURE OF PLATO

The architecture of Plato is shown in Figure 1. To facilitate our discussion, we assume that a data cube consists of analysis ready data in four dimensions: latitude, longitude, time and variable of interest. Further dimensions can be added as a result of an analysis.

The two main components of Plato are the OBDA system Ontop and the PostGIS backend. During initialization of the Ontop engine, an ontology in the OWL2 ontology language is defined, alongside a specified set of mappings.

The PostGIS backend contains virtual tables used to communicate with data cubes (stored locally or remotely). This communication is achieved through Python scripts utilizing the Xarray [10] library and the Multicorn package [2], to implement FDWs.

In Plato, data cubes are stored either in a `.zarr` directory format or as `.nc` (netCDF) files. Even with a compressed format like these two, many data cubes are very large to unpack and materialize in a PostgreSQL database. For that matter, we utilize FDWs and the Xarray package in order to work with labelled multi-dimensional arrays conveniently.

Using FDWs and Xarray can turn out to be an intensive operation, both time and memory-wise, and is by no means a realistic approach for large data cubes. For that reason, we decided to also look into parallelization modules of Python. As the FDW applications are not IO bound, multi-threading was not of much benefit, which was confirmed from brief experimentation. Multiprocessing, on the other hand, resulted in massive speedups, through the exploitation of data chunking and dispatched reading by multiple spawned processes, wherever possible.

In order to facilitate testing and a successful deployment of the entire pipeline, we have developed a dockerfile to build an image that installs all the necessary components (Post-

greSQL, Python3, Multicorn, Xarray, Zarr) and exposes a port to access the database within the created container.

Figure 1 also shows two optimization techniques that we have implemented at the PostGIS level. These techniques optimize the handling of large volumes of data (caching) as well the joining of raster and vector data (Raptor Join).

The main idea behind caching raster data in PostGIS by modifying the Ontop plugin is to efficiently query large volumes of data cubes by having various parts of them be materialized and readily available. To achieve this, during Ontop's query translation from GeoSPARQL to SQL, Plato recognizes the portions of raster data that need to be accessed and transformed to geometries, and saves them in an intermediate cache table in the database. In order to check if the requested data is already in the cache table, we have implemented data structures as indices inside the Ontop plugin. Currently, a hash table is fully implemented for the time dimension of the datasets, with similar functionality for latitude and longitude dimensions being under development using R-trees. If the requested data is found to be available in the cache table, the query translation process into SQL is modified, in order to access the cache and not a FDW (virtual table). This ease of access to "hot" data is the main benefit of the implementation, as it allows more efficient joins and other operations between dense data cubes and other materialized (non-EO) data.

After testing various GeoSPARQL queries on large data cubes, we have discovered that accessing large portions of data cubes and transforming each pixel to a vector point (the obvious implementation) created a bottleneck in our system. The idea behind Raptor Join [8] alleviates this problem by reading only parts of the raster that overlap a set of vector geometries (using *scanlines*), without the need for conversions between the two forms in order to perform a join. The Raptor Join method is implemented in Plato as a Python FDW and calculates the result of a spatial operation as output. The requirements are: a set of vector geometries, an EO variable name (raster), an aggregate function name (e.g., sum, max, count, etc.), a spatial relation (intersection currently supported), and a specific time frame as input. Just by adding the properties reflecting these parameters to a given ontology, we are able to create a single mapping to connect Ontop with the FDW operator.

4. PERFORMANCE EVALUATION

The experiments were run using Ontop 4.2, on a 64-bit machine with 32 logical processors (2.20 GHz), and 128GB of RAM (DDR3, 1600 MHz). Five different data cubes are used in the experiments, with time, latitude and longitude as the primary dimensions, along with several data variables:

- **DC-GR-1 (4314×562×700)**. NDVI data for Greece for the period 2009-2020.

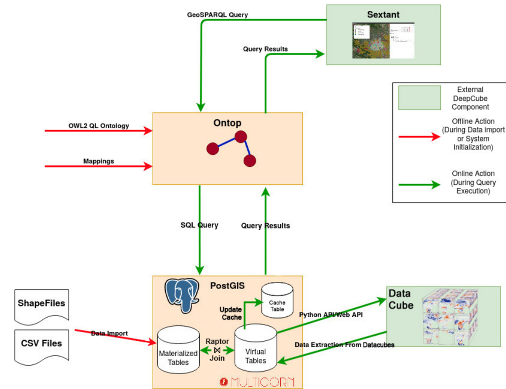

Fig. 1. The architecture of Plato

- **DC-GR-2 (1×940×1328)**. Daily relative humidity for Greece for the year 2022.
- **DC-BR (2160×200×200)**. NDVI data for Brazil for the year 2019.
- **DC-SI (8762×150×310)**. Hourly total precipitation data for Slovenia for the year 2021.
- **DC-FR (8760×334×636)**. Hourly total precipitation data for France for the year 2022.

Alongside those, we have imported vector data concerning fire prediction data for Greece (point geometries; 2022), Natura-protected areas for Europe (multipolygon geometries), and administrative data for Brazil (polygon geometries). The sizes for all of the datasets are displayed in table 1.

Table 1. Raster and Vector datasets

Raster (GB)	
Data cube	Size
DC-GR-1	10
DC-GR-2	0.5
DC-BR	5.9
DC-SI	15.5
DC-FR	35.5

Vector (MB)	
Dataset	Size
Fire Prediction Data	25
Natura Areas (GR)	7
Natura Areas (EU)	78
Brazil Admin. Data	0.5

Since Plato is the only existing semantic data cube system to our knowledge, the evaluation consists of experiments with different query types over the presented data sources, and the results of our optimization techniques. We start with the evaluation of the cache implementation. The different query types selected to evaluate the performance of our queries when using a cache table are as follows: (A): Requested variable for one day, (B): Requested variable for a range of dates, (C): Requested variable & join with Natura areas (historical) or predictions (daily), for one day, (D): Requested variable & join with Natura areas (historical) or predictions (daily), for a range of dates. The range of dates for our tests was 3-5 days. The results for the cache implementation on queries posed to

all the data cubes are shown in table 2. We can see that by utilizing a cache table we can overcome the overhead of retrieving the requested data from a foreign table through the use of FDWs. This is more apparent in queries that concern a range of dates (Types B and D), where the cache implementation shows by far the best results. In this way, it is clear that having readily available "hot" data for multiple requested dates by materializing them, improves the overall user experience.

Table 2. Query execution times with cache implementation

Cache (sec)			
Data Cube	Type	Default	Cache
DC-GR-1	A	72.3	20.2
DC-GR-1	B	243	60.6
DC-GR-1	C	1038	953
DC-GR-1	D	>18000	2221
DC-GR-2	A	28.6	1.06
DC-GR-2	B	96.8	7.65
DC-GR-2	C	29.7	1.26
DC-GR-2	D	90.2	2.38
DC-BR	A	6.83	2.22
DC-BR	C	13.6	10.8
DC-SI	A	5.61	2.51
DC-SI	C	72.9	8.31
DC-FR	A	26.5	11.1
DC-FR	C	5121	168

We continue with the evaluation of the join queries. For benchmarking needs, a method simpler than Raptor Join is also implemented, which checks if the pixels within the Minimum Bounding Rectangle (MBR) of input vectors coincide with the actual geometries (requires pixel-to-point transformations). We can also evaluate the performance of our entire pipeline by allowing the Raptor Join implementation to utilize the available data found in the cache table for each of the data cubes and posing the same queries as before. The performance results for those queries for all the different techniques are shown in table 3.

Table 3. Query execution times with Raptor join and Raptor-Cache combined

Join Results (sec)				
Data Cube	Default	MBR	Raptor	Raptor–Cache
DC-GR-1	1038	139	54	40.2
DC-BR	15.1	21.6	20.9	25.8
DC-SI	72.3	90.4	46.5	17.9
DC-FR	5121	299	137	75.2

The table shows the benefits of using the various join optimizations that we implemented instead of using the default FDWs for data retrieval and letting PostGIS handle the joins by making the necessary pixel-to-point transformations. First of all, the available data (both raster and vector) for Brazil is not very big, so the extra parsing/preprocessing in such cases seems to have an inverse effect on the total time efficiency. Furthermore, the vector data for France has geometries spread around the area of the entire country, while DC-FR is limited to the region of Occitanie. Hence, MBRs are calculated for many non-overlapping geometries, in which case Cache availability allows for the better handling of transformations by PostGIS rather than the use of the MBR technique's FDW. Finally, it is clear that both the simpler MBR method as well as the Raptor Join method generally offer a more efficient approach than handling join queries using standard PostGIS and the default FDWs. Caching data for specific observations and timeframes, when combined with Raptor Join, provide the best speedup (for data of substantial size).

5. SUMMARY

We presented Plato, the first semantic data cube system implementation using OBDA technologies. We evaluated Plato using data from the use cases of project DeepCube and showed that the optimizations of caching and Raptor Join allowed us to process GeoSPARQL queries very efficiently.

REFERENCES

[1] K. Bereta and M. Koubarakis. Ontop of geospatial databases. In *ISWC*, 2016.

[2] R. Dunklau and F. Mounier. Multicorn - PostgreSQL extension, 2015. URL https://multicorn.org/.

[3] G. Xiao et al. The virtual knowledge graph system ontop. In *ISWC*, 2020.

[4] H. Augustin et al. Semantic Earth observation data cubes. *Data*, 4(3), 2019.

[5] K. Bereta et al. Ontop-spatial: Ontop of geospatial databases. *J. Web Semant.*, 58, 2019.

[6] M. Stonebraker et al. The architecture of SciDB. In *SSDBM*, 2011.

[7] P. Baumann et al. The RasDaMan approach to multidimensional database management. In *ACM SAC*, 1997.

[8] S. Singla et al. The Raptor Join operator for processing big raster + vector data. In *ACM SIGSPATIAL*, 2021.

[9] Y. Zhang et al. SciQL: array data processing inside an RDBMS. In *ACM SIGMOD*, 2013.

[10] S. Hoyer and J. Hamman. xarray: N-D labeled arrays and datasets in Python. *Open Research Software*, 5(1), 2017.

[11] W3C. OWL 2 web ontology language profiles (second edition), 2012. URL http://www.w3.org/TR/owl2-profiles/.