ExtremeEarth

H2020 - 825258

Deliverable

D3.7

Software for querying and extreme analytics for big linked geospatial data - version II

Dimitris Bilidas, Dimitrios Chalatsis, Theofilos Ioannidis, Georgios Mandilaras, Dharmen Punjani, Manos Karvounis, Christos Lougiakis, Eleni Tsalapati, Despina-Athanasia Pantazi and Manolis Koubarakis

June 29, 2021

Status: FINAL
Scheduled Delivery Date: 30/06/2021

Executive Summary

This deliverable describes the activities of Task 3.3 of WP3 of the ExtremeEarth project during months M13-M30, about *querying and extreme analytics for big linked geospatial data*, aiming to develop a new version of the geospatial RDF store Strabon integrated in the Hopsworks platform. During the reported period, we developed the system Strabo2, that performs distributed GeoSPARQL query processing in Hopsworks. Strabo2 stores geospatial RDF data in a Hive database using the Vertical Partitioning RDF storage schema. Then, it performs GeoSPARQL to Spark SQL translation using the Ontop-spatial system. During these process, it uses a series of optimizations in order to come up with an efficient translation, that successfully scales to hundreds of worker nodes, as preliminary experiments show. We have also developed an endpoint running as a web service, that accepts GeoSPARQL queries. This endpoint communicates with the Hops-YARN resource manager through the Apache Livy framework, currently running in the Hopsworks installation in CREODIAS. The final and detailed experimental evaluation of Strabo2 will be reported in the Deliverable 3.5, as it depends on the evaluation framework and datasets currently under development in Task 3.5.

Document Information

Contract Number	H2020 - 825258	**Acronym**	ExtremeEarth
Full title	ExtremeEarth		
Project URL	`http://earthanalytics.eu/`		
EU Project Officer	Riku Leppänen		

Deliverable	**Number**	D3.7	**Name**	Software for querying and extreme analytics for big linked geospatial data - version II			
Task	**Number**	T3.3	**Name**	Querying and extreme analytics for big linked geospatial data			
Work package	**Number**			WP3			
Date of delivery	**Contract**	M30		**Actual**	30/06/2021		
Status	Draft ☐ Final ☑						
Nature	Prototype ☑ Report ☐						
Distribution Type	Public ☑ Restricted ☐						
Responsible Partner	UoA						
QA Partner	NCSR-Demokritos						
Contact Person	Prof. Manolis Koubarakis						
	Email	koubarak@di.uoa.gr		**Phone**	+30 210 7275213	**Fax**	+30 210 7275214

Project Information

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number H2020-825258. The beneficiaries in this project are the following:

Partner	Acronym	Contact
National and Kapodistrian University of Athens Department of Informatics and Telecommunications (Coordinator)	UoA	Prof. Manolis Koubarakis National and Kapodistrian University of Athens Dept. of Informatics and Telecommunications Panepistimiopolis, Ilissia, GR-15784 Athens, Greece Email: (koubarak@di.uoa.gr) Tel: +30 210 7275213, Fax: +30 210 7275214
VISTA Geowissenschaftliche Fernerkundung GmbH	VISTA	Heike Bach Email: (bach@vista-geo.de)
The Arctic University of Norway Deptartment of Physics and Technology	UiT	Torbjørn Eltoft Email: (torbjorn.eltoft@uit.no)
University of Trento Department of Information Engineering and Computer Science	UNITN	Lorenzo Bruzzone Email: (lorenzo.bruzzone@unitn.it)
Royal Institute of Technology	KTH	Seif Haridi Email: (haridi@kth.se)
National Center for Scientific Research - Demokritos	NCSR-D	Vangelis Karkaletsis Email: (vangelis@iit.demokritos.gr)
Deutsches Zentrum für Luft-und Raumfahrt e. V.	DLR	Corneliu Octavian Dumitru Email: (corneliu.dumitru@dlr.de)
Polar View Earth Observation Ltd.	PolarView	David Arthurs Email: (david.arthurs@polarview.org)
METEOROLOGISK INSTITUTT	METNO	Nick Hughes Email: (nick.hughes@met.no)
Logical Clocks AB	LC	Jim Dowling Email: (jim@logicalclocks.com)
United Kingdom Research and Innovation - British Antarctic Survey	UKRI-BAS	Andrew Fleming Email: (ahf@bas.ac.uk)

Contents

List of Figures

List of Tables

1. Introduction

This deliverable describes the progress of Task 3.3 of WP3 of the ExtremeEarth project up to the completion of the task in month M30 of the project. Task 3.3 develops the new highly scalable version of Strabon[1] [KKK12]. According to the task description, the result of this effort will be the first GeoSPARQL distributed engine for big geospatial data and extreme geospatial analytics, that will run on Hopsworks platform. The developments during the first year of the project were reported in Deliverable D3.3. There, we presented a detailed literature review regarding centralized systems for GeoSPARQL query processing, distributed systems that perform SPARQL query processing on large RDF graphs and distributed systems that perform spatial analytics. As a result, given the performance and scalability features of the systems and approaches that we investigated, we determined the fundamental design choices and the architecture of the the new distributed GeoSPARQL engine. Specifically, the decision to use the in-memory distributed processing engine Apache Spark [ZCF+10] was taken. Spark is widely used for distributed data processing and has a proven ability to scale to hundreds or thousands of worker nodes on Hadoop installations. Furthermore, in order to incorporate spatial functionality, we decided to use the GeoSpark [YWS15] libraries that provide rich spatial processing capabilities on top of Spark. GeoSpark is one of the most prominent solutions for spatial analytics at scale, and recently, it has been accepted as a project by Apache Incubator and has been renamed into Apache Sedona [2]. In Deliverable D3.3 we decided to extend the RDF data storage schema of systems that perform scalable SPARQL processing on Spark, like S2RDF [SPSL16] and PRoST [CFL18], with geospatial operators and data types. Finally we presented a first implementation, and based on experimental evaluation, we identified some weaknesses and limitations of this implementation.

In this deliverable we present the system *Strabo2*. Strabo2 achieves the task objective, as to the best of our knowledge, becomes the first GeoSPARQL distributed engine able to store and process massive geospatial RDF datasets. Strabo2 addresses the weaknesses and limitations of our first implementation, and currently, it has been experimentally evaluated with datasets of up to 450 GB size in NTRIPLES file format. Specifically, we have chosen to use the *Ontology-Based Data Access* (OBDA) system Ontop-spatial, as the module responsible for GeoSPARQL to Spark SQL enhanced with spatial functions. This choice addresses the limitations in translation of queries performed by systems like S2RDF, as it supports rich GeoSPARQL functionality.

The rest of the deliverable is organised as follows.

We present an overview of the system in Chapter 2, where we describe the two modules of Strabo2 and the main operations regarding the storage of geospatial RDF data in Hopsworks and the execution of GeoSPARQL queries. Then we proceed to describe improvements in the separate components of the system (data loader and query executor) in Chapters 3 and 4 respectively. These improvements address the shortcomings that we observed in experimental evaluation, that had been described in Deliverable D3.3, like for example the lack of compression, the need to access a permanent spatial index and the need for a query optimizer. Then in Chapter 5 we present Strabo2 SPARQL endpoint that communicates with Hops-YARN through Apache Livy and accepts requests in the form of GeoSPARQL queries. In Chapter 6 we discuss future steps, that mainly involve further experimental evaluation using the datasets and benchmarks developed in Task T3.5 of the project. Appendix A contains the paper [BK21] which optimizes the process of query translation as it is carried out by Ontop-Spatial. Finally, Appendix B presents the queries of the food security use-case that have been used in the evaluation. The final and detailed experimental evaluation of Strabo2 will be reported in the Deliverable 3.5, as it depends on the evaluation framework and datasets currently under development in Task 3.5.

[1] http://strabon.di.uoa.gr/
[2] https://sedona.apache.org/

2. An Overview of the Strabo2 System for Distributed GeoSPARQL Processing

In this chapter we present an overview of the Strabo2 system, starting with its architecture, which is shown in Figure 2.1.

The system comprises of two main modules: the data loader and the query executor. Data loader is shown in the bottom part of Figure 2.1 and is responsible for reading and importing into a HIVE database the RDF files from the file-system. We consider that RDF files are in the popular N-Triples file format, and that they have been saved in the distributed HopsFS file-system. Using the N-Triples format, an RDF graph can be saved in a set of files, such that each triple is represented in a separate line. This makes easy the distribution of N-Triples files accross the storage nodes of HopsFS. In case the initial RDF data are saved in a format that does permit convenient distribution across the file-system, then they should first be transformed into N-Triples. This is a simple task, as there are many efficient libraries for transforming RDF data from other formats into N-Triples. As the data loader reads each separate line form the input files, it saves the corresponding RDF triple in the HIVE database, using the *Vertical Partitioning* (VP) storage schema. According to VP, a separate two-column table is created in the database for each distinct predicate encountered in the RDF data. As an example, consider the following ten RDF triples from the ice maps dataset of the polar use case:

```
<http://earthanalytics.eu/polar/resource/norIceChart03_1>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://earthanalytics.eu/polar/ontology/IceObservation> .

<http://earthanalytics.eu/polar/resource/norIceChart03_1>
    <http://earthanalytics.eu/polar/ontology/hasRECDATE>
    "2018-03-01 00:00:00"^^<http://www.w3.org/2001/XMLSchema#dateTime> .

<http://earthanalytics.eu/polar/resource/norIceChart03_1>
    <http://earthanalytics.eu/polar/ontology/hasCT>
    "14"^^<http://www.w3.org/2001/XMLSchema#string> .

<http://earthanalytics.eu/polar/resource/norIceChart03_1>
    <http://earthanalytics.eu/polar/ontology/hasCTClassName>
    "Very Open Drift Ice"^^<http://www.w3.org/2001/XMLSchema#string> .

<http://earthanalytics.eu/polar/resource/norIceChart03_1>
    <http://www.opengis.net/ont/geosparql#hasGeometry>
    <http://earthanalytics.eu/polar/resource/Geometry_norIceChart03_1> .

<http://earthanalytics.eu/polar/resource/Geometry_norIceChart03_1>
    <http://www.opengis.net/ont/geosparql#asWKT>
    "<http://www.opengis.net/def/crs/EPSG/0/4326>
        POLYGON (...)"^^<http://www.opengis.net/ont/geosparql#wktLiteral> .

<http://earthanalytics.eu/polar/resource/image3f7488e1>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://earthanalytics.eu/polar/ontology/SatelliteImage> .

<http://earthanalytics.eu/polar/resource/image3f7488e1>
    <http://earthanalytics.eu/polar/ontology/hasURL>
    "https://finder.creodias.eu/resto/collections/Sentinel1/3f7488e1"
```

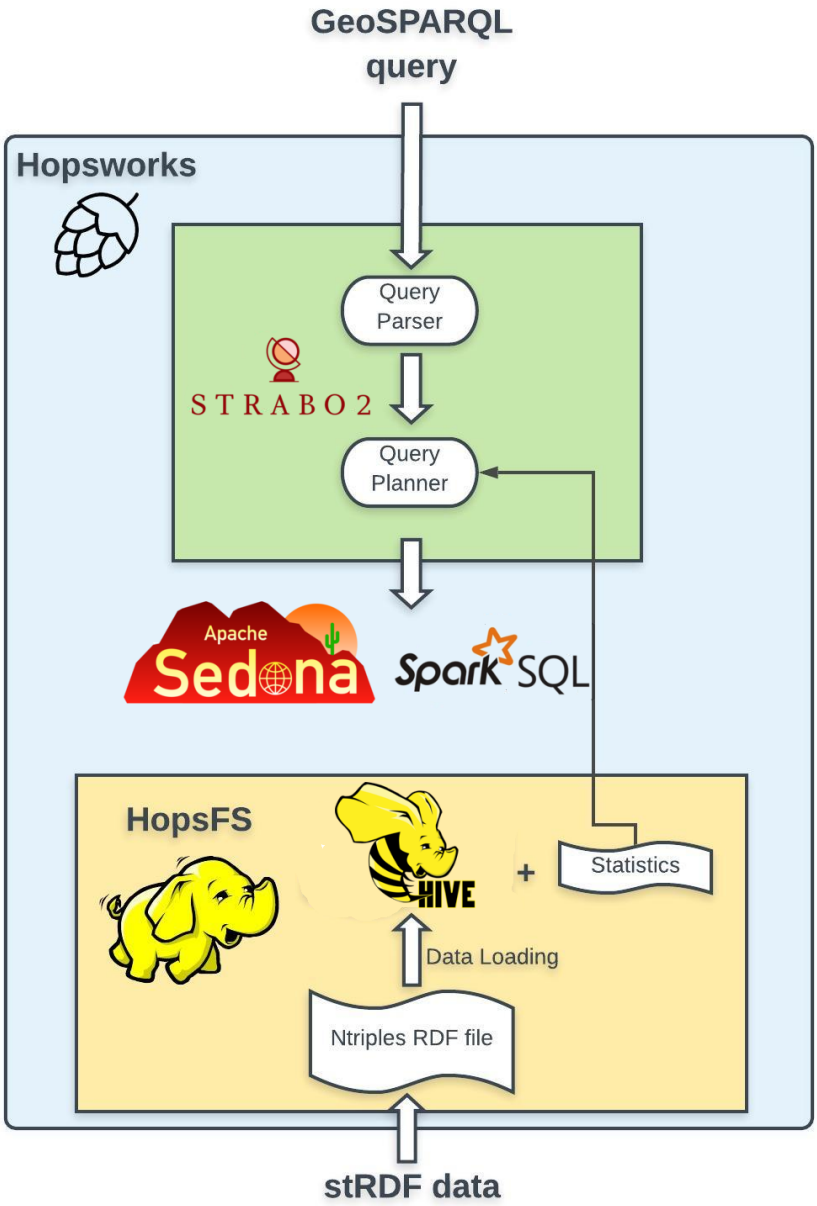


Figure 2.1: Architectural Overview of Strabo2

```
                                ^^<http://www.w3.org/2001/XMLSchema#string> .

<http://earthanalytics.eu/polar/resource/image3f7488e1>
    <http://www.opengis.net/ont/geosparql#hasGeometry>
    <http://earthanalytics.eu/polar/resource/Geometry_3f7488e1> .

<http://earthanalytics.eu/polar/resource/Geometry_3f7488e1>
    <http://www.opengis.net/ont/geosparql#asWKT> "<http://www.opengis.net/def/crs/EPSG/0/4326>
    MULTIPOLYGON (...)"^^<http://www.opengis.net/ont/geosparql#wktLiteral> .
```

For ease of presentation, we will use the following prefixes:

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX polaronto: <http://earthanalytics.eu/polar/ontology/>
PREFIX polarres: <http://earthanalytics.eu/polar/resource/>
```

For example, instead of the full IRI <http://earthanalytics.eu/polar/resource/norIceChart03_1> we will write polarres:norIceChart03_1. The specific triples, contain exactly seven different predicates: *rdf:type*, *polaronto:hasCT*, *polaronto:hasCTClassName*, *polaronto:hashasRECDATE*, *polaronto:hasURL*, *geo:hasGeometry* and *geo:asWKT*. For each of these predicates a table will be created in HIVE database. The name of the table will be in the form “propN", where N is an integer number different for each table. A dictionary that saves the mappings from the full property IRIs to these integers is also created. As an example, consider that data loader creates the following property dictionary in our example:

```
rdf:type -> prop1
polaronto:hasCT -> prop2
polaronto:hasCTClassName -> prop3
polaronto:hasRECDATE -> prop4
geo:hasGeometry -> prop5
geo:asWKT -> prop6
polaronto:hasURL -> prop7
```

In our example, the table prop2 will have two columns (subject and object), and it will contain the tuple: (polarres:norIceChart03_1, 14). Each of these tables will be saved in the highly efficient PARQUET file format, that uses columnar compression, which results in decreased size, especially for columns that exhibit a high degree of homogeneity.

The second module of Strabo2 is the query executor, and it is shown in the upper part of Figure 2.1. The query executor accepts GeoSPARQL queries from the user, and transforms them to a series of Spark SQL queries that access the HIVE tables created by the loader. The spatial operators of GeoSPARQL are translated to corresponding spatial functions offered by the Apache Sedona library, which operates on top of the Spark engine. The translation mechanism of the query executor depends on the Ontop-spatial system[BXK19]. Ontop-spatial is a system for GeoSPARQL-to-SQL query translation over arbitrary relational schemas. Ontop-spatial operates with data that exist in spatial relational database management systems, like the PostGIS spatial extension of PostgreSQL, and provides to the user access to a *virtual* RDF graph constructed from

the relational data, through the means of mappings defined in the W3C recommendation mapping language R2RML [1], that construct RDF terms from the database values.

In order to use Ontop-spatial for query translation in the query executor module of Strabo2, we had to perform several modifications and improvements in order to use Spark as a backend and work with the RDF data stored in HIVE. First of all, as in our case the data loader stores the data according to a specific storage schema tailored for RDF data, the VP schema, we had to provide mappings that reconstruct the original RDF triple for each tuple in the HIVE tables. In the normal setup of Ontop-spatial, a data engineer has to manually construct the mappings. This task is tedious and time-consuming, especially for complex relational schemas. In our case, as the HIVE schema is RDF specific, we can avoid this process and instead construct a trivial mappng for each table. Specifically, Strabo2 automatically constructs this mappings on system startup, transparently to the user, based on the contents of the property dictionary. As an example, consider the table prop2 constructed from the data loader. The following mapping is generated and provided as input to the Ontop-spatial translation mechanism:

```
{subject} polaronto:hasCT "{object}"^^<http://www.w3.org/2001/XMLSchema#string> <-
    SELECT subject, object FROM prop2
```

The right-hand side of the mapping is a SQL query that can be executed by Spark, whereas the left-hand side is a template that defines how triples should be generated, using the output columns of the SQL query within curly brackets. Ontop-spatial takes as input a set of such mappings and accesses the metadata of the database in order to gather necessary information that will guide the query translation. Again, as Spark is not compatible with the Ontop-spatial system, we provide the specific metadata automatically, during system start-up, using information from the property dictionary file. This information includes the tables that reside in the database, the data types of each column and information about primary keys. As in the case of the mappings, this information is constructed automatically by Strabo2.

Once Ontop-spatial has been provided with the set of mappings and the metadata, it is ready to accept GeoSPARQL queries. The initial GeoSPARQL query is initially parsed and transformed in an intermediate form. This form is based on logic programs. After this, a rewriting process is taking place, where the ontology axioms are taken into account, and finally the result of the rewriting is transformed, based on the mappings, into SQL queries on the dialect of Spark-SQL. During this procedure, the spatial operators of GeoSPARQL are transformed to spatial functions provided by Apache Sedona. In order to demonstrate query translation, consider the following initial GeoSPARQL query:

```
SELECT ?imgURL ?ctName
WHERE {
?observation rdf:type polar:IceObservation .
?observation polar:hasCTClassName ?ctName .
?observation geo:hasGeometry ?geo1 .
?geo1 geo:asWKT ?wkt1 .

?img rdf:type polar:SatelliteImage .
?img polar:hasURL ?imgURL .
?img geo:hasGeometry ?imgGeo .
?imgGeo geo:asWKT ?imgWKT .

FILTER (geof:sfIntersects(?wkt1, ?imgWKT)).
}
```

[1] https://www.w3.org/TR/r2rml/

This query asks for the name of the class assigned to specific ice observations and the URL of images, such that the geometry of each observation intersects with the geometry of the image. The query uses the GeoSPARQL function geof:sfIntersects with arguments the corresponding geometries. Given the property dictionary that we presented, query translation will produce the following query that will be sent for execution to the Spark engine, where function ST_Intersects is defined in the Apache Sedona library:

```
SELECT
   qview6."object" AS "imgURL",
   qview2."object" AS "ctName"
 FROM
"prop1" qview1,
"prop3" qview2,
"prop5" qview3,
"prop6" qview4,
"prop1" qview5,
"prop7" qview6,
"prop5" qview7,
"prop6" qview8
WHERE
(qview1."object" = 'http://earthanalytics.eu/polar/ontology/IceObservation') AND
(qview1."subject" = qview2."subject") AND
(qview1."subject" = qview3."subject") AND
(qview3."object" = qview4."subject") AND
(qview5."object" = 'http://earthanalytics.eu/polar/ontology/SatelliteImage') AND
(qview5."subject" = qview6."subject") AND
(qview5."subject" = qview7."subject") AND
(qview7."object" = qview8."subject") AND
(ST_Intersects(qview4."object",geom2."object"))
```

A drawback of the vertical partitioning schema is that we need to perform a join in the final SQL query for each triple pattern of the input SPARQL query. As RDF data do not follow a specific relational schema, this in general cannot be avoided. There are alternative storage schemas, such as the property tables, that try to reconstruct a tabular format from the RDF data, by storing multiple properties that correspond to a resource in a multi-column table, as for example was the case in the early RDF store Jena2 [WSK+03]. This approach has the drawback that we need to preprocess the data in order to decide which property tables should be created, and also if in the created tables for many resources some properties are not existent, tables with lot of NULL values will be created. On the other hand, this design is also complicated when a resource has multiple values for a specific property. For these reasons we chose to follow the vertical partitioning schema. But in this case, we do one exception, as we take advantage of the knowledge that according to the GeoSPARQL ontology, the propertes geo:hasGeometry and geo:asWKT are usually encountered together. As a result, we create a single table for these two properties, the table geometries, that has three columns: entity, geometry and wkt. Using this table, the previous query is finally rewritten in the following form, where the number of joins is reduced by two:

```
SELECT
   qview5."o" AS "imgURL",
   qview2."o" AS "ctName"
 FROM
"prop1" qview1,
"prop3" qview2,
"global_temp"."geometries2" qview3,
"prop1" qview4,
```

```
"prop7" qview5,
"global_temp"."geometries2" qview6
WHERE
(qview1."o" = 'http://earthanalytics.eu/polar/ontology/IceObservation') AND
(qview1."s" = qview2."s") AND
(qview1."s" = qview3."entity") AND
(qview4."o" = 'http://earthanalytics.eu/polar/ontology/SatelliteImage') AND
(qview4."s" = qview5."s") AND
(qview4."s" = qview6."entity") AND
(ST_Intersects(qview3."wkt",qview6."wkt"))
```

3. Improvements in Semantic Data Storage and Loaders

In this chapter we present a mature, well researched and tested, set of techniques, packaged in the form of two semantic database storage schemes, each one offering solid ground for a compatible Query Executor variant.

Below we explain in detail both database schemes and discuss the logic behind their design alongside with how each one of the techniques is used in either scheme. All sample data presented in this chapter is related to the Synthetic dataset of Geographica 2 [1] [IGK+21].

3.1 Data Storage Design Schemes and Techniques

The main aim of this work is to achieve superior query performance for big geospatial semantic data on horizontally scalable computing platforms. The first step to achieving PB scaling, requires that input data to be ingested, take the least possible storage space and at the same time use data serialization formats that are splittable and support parallel ingestion.

Big data is a challenge in its own right, but it becomes an even bigger one in horizontally scalable computing platforms, such as clusters. The effective allocated storage of a dataset in a standard Hadoop cluster, is a multiple of the actual size. Replication factors are greater than x1.5, usually x3, and they are used in order to increase parallelism opportunities through data locality. A 100TB input dataset actually consumes between 150-300TB of the host cluster storage resources.

For semantic data, N-Triples is the serialization format which is splittable and allows fast parallel ingestion due to its simple parsing requirements, however it uses absolute IRIs and therefore these files have a large storage footprint. However, other serialization formats, such as Turtle, allow through the use of a common prefix dictionary in the file header, partial encoding of IRIs which effectively reduces the necessary storage requirements.

The second step for delivering superior query performance on big data is to design target database schemes that are efficient, flexible and robust. `Efficiency` is three fold: (i) minimizing the table storage requirements by using compression methods, (ii) using partitioning techniques to increase data locality and (iii) allow quick retrieval for common query patterns through various indexes and statistics bookkeeping. `Flexibility` means: (i) the database should be as self describing as possible so that various clients can be informed about the database configuration and make the most appropriate decisions for processing information, (ii) allow as many as possible target database options to be controlled by the user upon ingestion. `Robust` is a system that will perform consistently on medium, large and extreme data sizes, without having extreme dependencies on the availability of huge amounts of RAM and executor nodes.

3.2 Design Schemes - Loaders

There are two design schemes developed. Scheme 1 corresponds to Loader 1 and it is the early implementation developed mainly in the first 2 years of the project and its main focus is on performance and rapid application development. Scheme 2 is produced by Loader 2, which is a redesign of Loader 1, trying to improve on flexibility, robustness and scalability by minimizing storage, memory and processing requirements as possible.

[1] `http://geographica2.di.uoa.gr/`

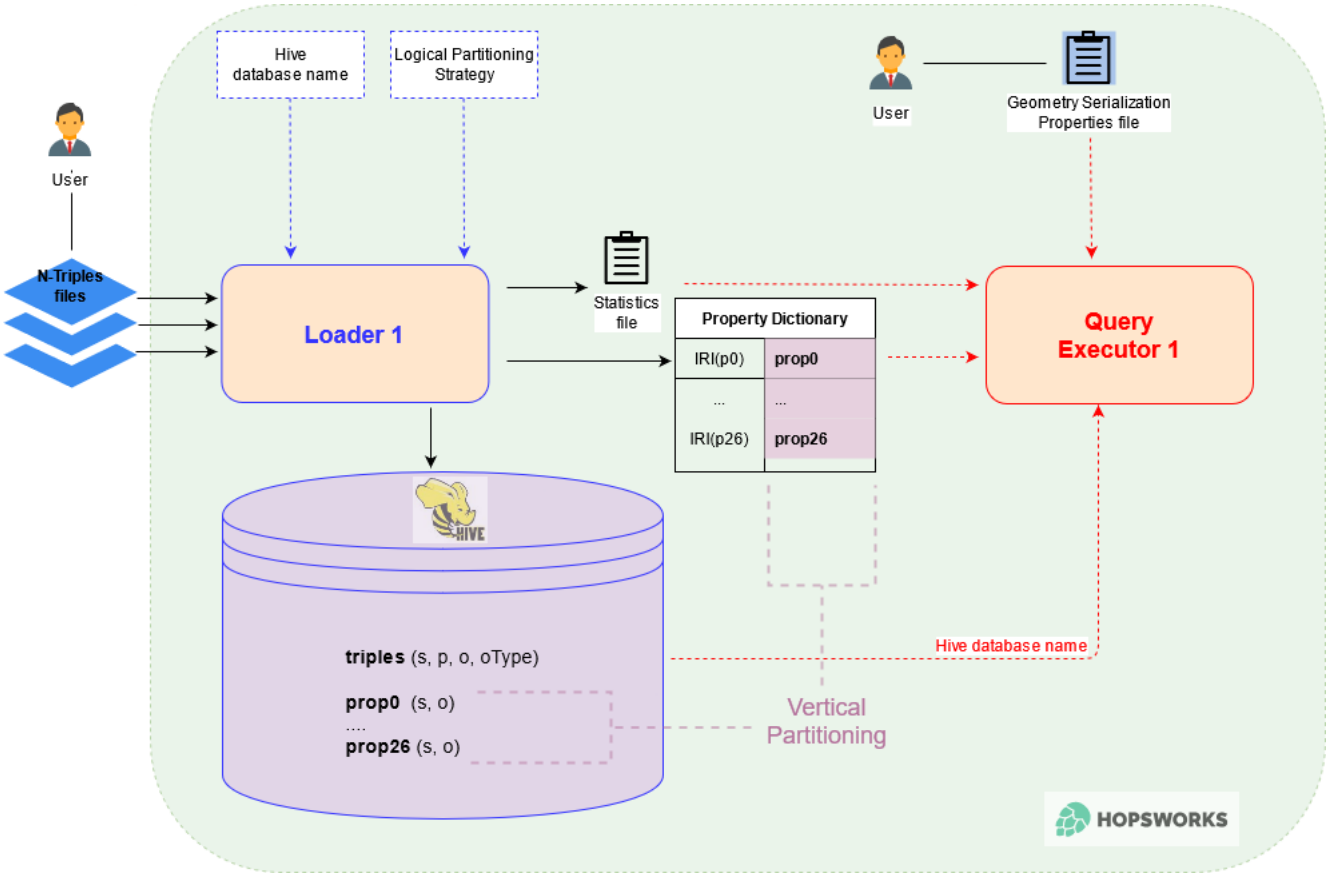


Figure 3.1: Overview of Loader 1 architecture

3.2.1 Scheme 1 - Loader 1

Scheme 1 is the layout of the Hive database and external files produced after Loader 1 loads a dataset. The N-Triples text files that form the dataset are expected to be found in a single folder. These are usually uploaded to the cluster or produced by some distributed application such as GeoTriples or the `DistSynthGen` (distributed synthetic generator). The output of the process is a set of tables in a Hive database and two external files. An overview of the Loader 1 architecture in the Hopsworks cluster environment is shown in Figure 3.1.

Apart from choosing the name of the Hive database, Loader 1 has the option of selecting combinations of two logical partitioning strategies, such as TripleTable and Vertical Partitioning, with the (TT+VP) combination being the most efficient setup. In the initial step of ingestion, the parsed triples are inserted to the TripleTable `triples(s, p, o, oType)` and categorized by the type of object. The additional calculated `oType` integer column differentiates triples with IRI objects (oType=2) from objects with literal values (oType=1).

After that, it calculates the set of distinct predicates from the TripleTable and maps each predicate IRI to a unique name, which follows the pattern `prop{X}`. This dictionary encoding of the predicates forms the `Property Dictionary` which is stored as an external CSV file. A sample extension of the Property Dictionary file produced for the synthetic dataset is presented in Figure 3.2

The encoded values prop{X} are used as the names of the VP tables to be created. Each VP table `propX(s, o)` corresponds to a predicate and contains the related subject-object pairs. The subject and object remain in their original form as absolute IRIs or literal values. VP allows for

```
http://geographica.di.uoa.gr/generator/stateCenter/asWKT,prop0
http://geographica.di.uoa.gr/generator/state/hasValue,prop1
http://geographica.di.uoa.gr/generator/landOwnership/hasTag,prop2
http://geographica.di.uoa.gr/generator/landOwnership/hasKey,prop3
http://geographica.di.uoa.gr/generator/road/hasGeometry,prop4
http://geographica.di.uoa.gr/generator/stateCenter/hasValue,prop5
```

Figure 3.2: Loader 1 - Property Dictionary sample

fast retrieval of triples matching the specific predicate. With the help of a Jupiter notebook a sample extension of a VP Hive table produced for the synthetic dataset is shown in Figure 3.3

An example run of the Loader 1 on a standard Hadoop+Spark v2.x cluster is shown below:

```
$SPARK_HOME/bin/spark-submit --class run.Main \
--master spark://localhost:7077 \
--conf spark.sql.hive.metastore.version=2.3.3 \
--conf spark.hadoop.hive.metastore.uris=thrift://localhost:9083 \
--conf spark.sql.hive.metastore.jars=$HIVE_HOME/lib/* \
--conf spark.hadoop.datanucleus.autoCreateSchema=true \
--conf spark.hadoop.datanucleus.fixedDatastore=false \
--conf spark.hadoop.hive.exec.dynamic.partition=true \
--conf spark.hadoop.hive.exec.dynamic.partition.mode=nonstrict \
target/PRoST-Loader-2.3.1-SNAPSHOT_hdfs.jar \
-i hdfs://localhost:9000/user/tioannid/Resources/Synthetic \
-o synthetic -lp TT,VP -dp false \
-sf hdfs://localhost:9000/user/tioannid/Results/statSynth.csv \
-df hdfs://localhost:9000/user/tioannid/Results/dictSynth.csv

-i : the input directory where all N-Triple files reside
-o : the desired name of the Hive database to store the dataset
-lp : logical partitioning options (TT: Triple Table, VP: Vertical Partitioning)
-dp : remove duplicates from all logical partitioning tables
-sf : statistics file
-df : dictionary file
```

As mentioned earlier, Loader 1 was built with rapid application development in mind in order to yield immediate results that allowed us to test and select the most appropriate techniques, libraries, file formats, encoding schemes, etc. However, it does not conserve storage space and memory and therefore does not scale in a cluster-friendly manner. A few points worth mentioning are listed below:

- The VP table format is decided by the default Spark or Hive cluster settings. As a result the Hive database size is not optimal and is bigger than the input dataset size.
- There is no single location for the ingested dataset and related metadata for easy consumption by distributed clients in the cluster. The property dictionary file which contains the mapping of predicate IRIs to VP table names and a very basic statistics file are stored as external HDFS CSV files. Therefore tools that operate on this knowledge graph, such as the Strabo2 Query Executor 1, have to be aware of all the locations of the database and the external files which have to be passed on as additional parameters.

```
sql("select s, o from prop2 limit 5").show(false)
```

```
+-----------------------------------------------------+-----------------------------------------------------------------+
|s                                                    |o                                                                |
+-----------------------------------------------------+-----------------------------------------------------------------+
|http://geographica.di.uoa.gr/generator/landOwnership/30844/|http://geographica.di.uoa.gr/generator/landOwnership/30844/tag/1/|
|http://geographica.di.uoa.gr/generator/landOwnership/30972/|http://geographica.di.uoa.gr/generator/landOwnership/30972/tag/1/|
|http://geographica.di.uoa.gr/generator/landOwnership/30785/|http://geographica.di.uoa.gr/generator/landOwnership/30785/tag/1/|
|http://geographica.di.uoa.gr/generator/landOwnership/31216/|http://geographica.di.uoa.gr/generator/landOwnership/31216/tag/1/|
|http://geographica.di.uoa.gr/generator/landOwnership/30973/|http://geographica.di.uoa.gr/generator/landOwnership/30973/tag/1/|
+-----------------------------------------------------+-----------------------------------------------------------------+
```

Figure 3.3: Loader 1 - Vertical Partitioned Table sample

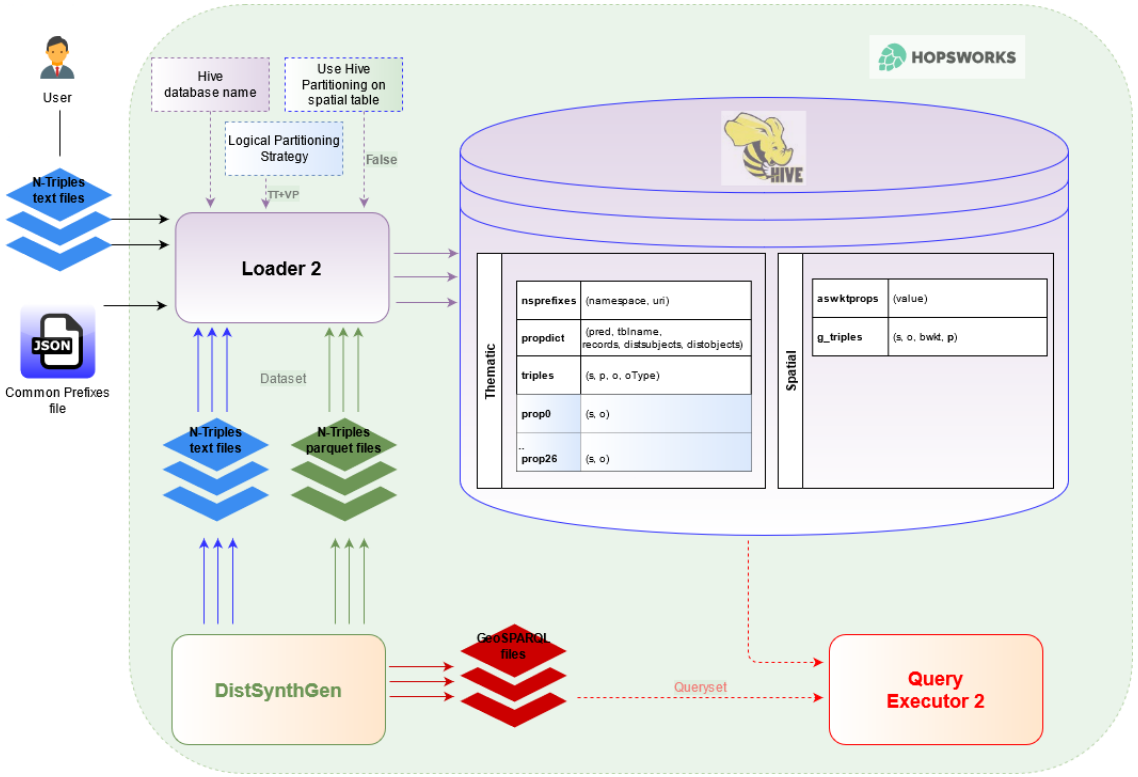


Figure 3.4: Overview of Loader 2 architecture

- For geospatial clients of the dataset the knowledge of which are the spatial serialization properties is of great importance. Loader 1 however behaves as a distributed generic RDF loader, does not discover or indicate them in any way, and basically places this burden on the client. As a result, tools such as the Executor 1, require that the user manually create an appropriate file with this information and provide it through the argument list.
- The statistics file is very basic, a list of VP table names and does not offer a lot of insight for query optimization opportunities.
- Loader 1, for simplicity reasons, is based entirely on the Spark SQL API for the bulk of the operations it performs and does not work with the Spark RDD interfaces which can provide finer access to optimizing the transformations performed.

3.2.2 Scheme 2 - Loader 2

Loader 2 is designed with versatility and stability for very large datasets in mind. It incorporates all the features of Loader 1, however it aims to be cluster friendly even in very large data scales, respecting the common storage and memory resources as much as possible.

It can import RDF graphs described with N-Triples serialization, in Text or Parquet files located in multiple folders. The tool works very well with partitioned files (Text or Parquet) which further speeds up ingestion. It shares similar resource-conscious design principles as the `DistSynthGen` (distributed synthetic generator) which provides Text or Parquet N-Triples partitioned files with user-defined number of partitions. The output of the loading process is a set of tables in a Hive database. An overview of the Loader 2 architecture in the Hopsworks cluster environment is shown in Figure 3.4.

Loader 2 parameters include:

```
sql("select namespace, uri from nsprefixes").show(false)
```

namespace	uri
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
xsd	http://www.w3.org/2001/XMLSchema#
dcterms	http://purl.org/dc/terms/
owl	http://www.w3.org/2002/07/owl#
geo	http://www.opengis.net/ont/geosparql#
geof	http://www.opengis.net/def/function/geosparql/
sgenroad	http://geographica.di.uoa.gr/generator/road/
sgenstate	http://geographica.di.uoa.gr/generator/state/
sgenstatcntr	http://geographica.di.uoa.gr/generator/stateCenter/
sgenlndown	http://geographica.di.uoa.gr/generator/landOwnership/
sgenpoi	http://geographica.di.uoa.gr/generator/pointOfInterest/
lgo	http://data.linkedeodata.eu/ontology#
lgd	http://data.linkedeodata.eu/osm/

Figure 3.5: Loader 2 - Common Prefixes table contents for Synthetic dataset

- the name of the output Hive database,
- selecting logical partitioning strategies, such as TripleTable and Vertical Partitioning, with the (TT+VP) combination being the most efficient setup,
- optional physical partitioning on the predicate of the spatial triples table,
- using HiveQL or Spark SQL dataframe API as the data definition language,
- choice of several Hive table format: Text, Parquet, Orc
- a JSON file with common IRI namespace prefixes related to the ingested dataset.

The `Common Prefixes` JSON file is constructed manually per imported dataset. Depending on the source of the geospatial dataset, e.g., shapefiles, and the intermediate processing by tools such GeoTriples, one can mine into the base ontology definitions (OSM, synthetic generators) and the possible transformation mappings to come up with a small set of the most common prefixes. This file is important because it guides the partial dictionary encoding of the IRIs at a later stage. At the very least the file should contain common namespace prefixes from XML, RDF, RDFS and GeoSPARQL vocabularies which are encountered in many datasets. The corresponding database `nsprefixes(namespace, uri)` table is shown in Figure 3.5 with the help of a Jupiter notebook and in the red rectangle we depict the prefixes subset related to the synthetic dataset ontology.

After the initial parsing to Spark RDD, Loader 2 proceeds with the inference of the geospatial WKT serialization predicates which are consequently persisted to the `aswktprops(value)` Hive table. The process involves searching for triples matching the triple pattern **(?s rdfs:subPropertyOf geo:asWKT)** and using the matching subject **?s** as a geospatial property. Based on this set of discovered spatial predicates and unlike Loader 1, it separates thematic from spatial triples into separate RDDs.

A major feature of Loader 2 is that, using the common namespace prefixes in `nsprefixes`, it applies partial dictionary encoding on all IRIs of thematic and spatial RDDs. This effectively simulates the main part of an N-Triples to Turtle conversion with the emphasis being on achieving a substantial first-level compression of the ingested dataset.

The encoded thematic and geospatial triples are stored in different triple tables. Similar to Loader 1, the Thematic TripleTable `triples(s, p, o, oType)` categorizes triples by the type of object, where the additional calculated `oType` integer column differentiates triples with IRI objects (oType=2) from objects with literal values (oType=1).

During VP table generation, the following important differences from Loader 1 are introduced:

```
sql("select pred, tblname, records, distSubjects, distObjects from propdict limit 5").show(false)
+------------------------+-------+-------+------------+-----------+
|pred                    |tblname|records|distSubjects|distObjects|
+------------------------+-------+-------+------------+-----------+
|sgenstatcntr:hasGeometry|prop15 |116281 |116281      |116281     |
|sgenlndown:hasGeometry  |prop1  |1048576|1048576     |1048576    |
|sgenstate:hasGeometry   |prop10 |116281 |116281      |116281     |
|sgenstatcntr:hasValue   |prop20 |116391 |116391      |116391     |
|sgenroad:hasGeometry    |prop11 |1024   |1024        |1024       |
+------------------------+-------+-------+------------+-----------+
```

Figure 3.6: Loader 2 - Property Dictionary Table sample

- VP concerns only the thematic triple predicates,
- the Loader 1 property dictionary, which was persisted as an external comma delimited file, is now persisted in the `Property Dictionary` **`propdict(pred, tblname, records, distsubjects, distobjects)`** table, as it stores the mapping of the partially encoded thematic predicates **pred** to VP table names **tblname**.
- the `Property Dictionary` plays also the role of a `Statistics File`, since it stores important statistics (number of records, distinct subjects and distinct objects) in the corresponding columns of **`propdict(pred, tblname, records, distsubjects, distobjects)`** for each VP table, calculated during VP table generation. A sample of the Property Dictionary and Statistics table in shown in Figure 3.6.

Triples identified as geospatial are persisted to a Spatial TripleTable `g_triples(s, p, o, bwkt)`. The additional column `bwkt` holds the geometry that corresponds to the WKT stored in the `o` column. This one-off calculation at ingestion time is performed through the Apache Sedona[2] geospatial library and is beneficial to any spatial client that is spared from repeating this calculation. It is also available an option of enabling physical Hive partitioning on the predicate `p` which provides similar performance benefits to VP logical partitioning, with the added benefit of hiding this physical detail from the SQL query language. A sample of the Spatial TripleTable for a point only feature class is shown in Figure 3.7, where the WKT and the calculated geometry are listed.

```
sql("select s, o, bwkt from g_triples where p='sgenpoi:asWKT' limit 3").show(false)
+--------------------+-----------------------------------------------------+-----------------------------------------------------------------------------------------------------+
|s                   |o                                                    |bwkt                                                                                                 |
+--------------------+-----------------------------------------------------+-----------------------------------------------------------------------------------------------------+
|sgenpoi:geometry/805/|POINT ( 62.99878120422363 61.24308492355548) |[0, 1, 0, 0, 0, 16, -40, 127, 79, 64, 37, 105, 34, 104, 29, -97, 78, 64, 1, 3, 1, -127]   |
|sgenpoi:geometry/82/ |POINT ( 62.93672561645508 6.238426041902546) |[0, 1, 0, 0, 0, -96, -26, 119, 79, 64, -44, -15, -47, -12, 37, -12, 24, 64, 1, 3, 1, -127]|
|sgenpoi:geometry/136/|POINT ( 62.94136047363281 10.346657825594466)|[0, 1, 0, 0, 0, -128, 126, 120, 79, 64, 82, -87, 111, 34, 125, -79, 36, 64, 1, 3, 1, -127]|
+--------------------+-----------------------------------------------------+-----------------------------------------------------------------------------------------------------+
```

Figure 3.7: Loader 2 - Spatial TripleTable sample

During all the previous steps and for each one of the created tables, Loader 2 forces manual calculation of Hive table, column and partition statistics and has them persisted in the Hive metastore. These statistics are more complete than the `propdict` table statistics presented earlier and are important in many occasions. One of the key use cases of statistics is query optimization and clients such as Executor 2 can use them as input to the optimizer cost functions in order to choose the best execution plan. Another useful case is that some aggregate statistics may be exactly what the users' queries request. Users can quickly get the answers for these queries by only retrieving stored statistics rather than firing long-running execution plans.

An example run of the Loader 2 on a standard Hadoop+Spark v2.x cluster is shown below:

```
$SPARK_HOME/bin/spark-submit --class run2.Main \
--master spark://localhost:7077 \
--conf spark.sql.parquet.compression.codec=snappy \
--conf spark.hadoop.hive.metastore.uris=thrift://localhost:9083 \
```

[2] `http://sedona.apache.org/`

```
--conf spark.sql.hive.metastore.version=2.3.3 \
--conf spark.sql.hive.metastore.jars=$HIVE_HOME/lib/*  \
--conf spark.hadoop.hive.exec.dynamic.partition=true \
--conf spark.hadoop.hive.exec.dynamic.partition.mode=nonstrict \
--files $HIVE_HOME/conf/hive-site.xml \
target/PRoST-Loader-2.3.1-SNAPSHOT_hdfs.jar \
-i hdfs://localhost:9000/user/tioannid/Resources/Synthetic/256/data \
-o ds256_hivepart -lp VP -hiveql -ttp -tblfrm parquet -df propdict \
-nsprf hdfs://localhost:9000/user/tioannid/Resources/commonprefixes.json

-i : the input directory where all N-Triple files reside
-o : the desired name of the Hive database to store the dataset
-lp : logical partitioning option (VP: Vertical Partitioning implies TT)
-df : property dictionary table name for encoding IRIs
-nsprf : fixed dictionary JSON file with common RDF namespace prefixes
-hiveql : use HiveQL instead of Spark SQL Dataframe API for DDL
-ttp : use predicate partitioning for ‘g_triples’ table
-tblfrm : set the Hive default table format (parquet, orc, text)
```

Improvements over Scheme 1 - Loader 1

The choice for target Hive table formats provides the user with possibility to achieve optimal output storage size with minimal processing overhead. Combining the first level compression achieved through partial dictionary encoding with Parquet table format and Snappy compression, we achieve compression ratios less than 10%, a major difference from Loader 1.

The unification of all output data and metadata into the same flexible database, makes it easy for clients to find all relevant information to complete their tasks. There are extensive Hive metastore persisted statistics for all tables, columns and partitions alongside the Hive database included statistics for VP tables. This way, Loader 2 provides flexibility and necessary information for QueryExecutor like clients to make better query plans.

Critical metadata for the Query Executor such as WKT serialization properties and WKT-to-Geometry conversions are inferred, calculated and persisted in the Hive database, off-loading the QueryExecutor from having to do so.

Loader 2, uses both the Spark SQL API for many Hive related operations but does not miss the opportunity of tapping into the Spark RDD interfaces especially on early stages of the ingestion to perform the transformations in a more optimal way.

3.2.3 Loader Benchmarks

In this part we present comparative tests with Loader 1 and Loader 2 on a VM-hosted standard Hadoop cluster and in Hopsworks PolarTEP cluster.

Standard Hadoop on a VM

The virtual machine was hosted in a HP OMEN gaming laptop and the allocated resources were **4vCores** (Intel i7-7700HQ)and **21GB RAM** (DDR4 2400MHz). The cluster is a standard Hadoop `v2.10.0` with Spark `v2.4.5` and Hive `v2.3.7`. The Hive installation uses MariaDB as the metastore database. The HDFS **replication factor** was **1**. The cluster manager used for the jobs was **Spark Standalone** and cluster utilization before each run was **0%**.

The reference dataset used for import was the Geographica 2 Synthetic dataset (scaling factor **N=512**) and was served in three different modes with the corresponding statistics:

- Raw : 1 N-Triple text file, size 743.9MB, 3.880.328 triples

HDFS Output Storage Requirements (MB)

	Loader 1	Loader 2
Raw	752.3	63.5
GZip	752.3	63.5
Parts GZip	752.3	63.5

Bulk Load Time (min)

	Loader 1	Loader 2
Raw	4.1	4.7
GZip	5.2	8.9
Parts GZip	5.1	4.0

Figure 3.8: Loader 2 - Loaders Comparison - VM

- GZip : 1 Gzip file, size 32.2MB (4.3% of raw), 3.880.328 triples
- Parts GZip[3] : splitted Raw to 4 files and gzipped them, size 33MB (4.3% of raw), 970.082 triples GZip file

Both Loaders used TT+VP logical partitioning, dictionary encoding enabled and Parquet was the set as the default Spark table format. The comparison metrics are the size of the output data and the ingestion time.

From the results in Figure 3.8 it was verified that compressing the text N-Triples files did not work for either one of the Loaders, because GZip is not a splittable Hadoop input format and consequently push us to the splittable compressed alternatives such as Parquet with Snappy compression. The other obvious conclusion is that Loader 2 double compression, with partial dictionary encoding and Parquet+Snappy for database table format, requires much less storage than Loader 1 does. A final note was that although Loader 1 performed better on ingesting a single text or gzipped text file, Loader 2 outperformed Loader 1 when provided with multiple compressed text files (splittable input format) which is the expected input format when working in distributed file systems.

Hopsworks

The experiments for the Hopsworks platform were performed on the PolarTEP[4] infrastructure, which was setup with **replication factor 1**. The Hive installation uses MySql as the metastore database. The cluster manager used for the jobs is **Hadoop YARN**. The job profile comprised a **static Spark** allocation of **a driver (1vCPU, 4GB RAM)** and **10 executors (1vCPU, 4GB RAM)**. Before each run the **cluster utilization was below 13%**. The **maximum cluster utilization** during the job execution was **40%**.

The Geographica 2 Synthetic dataset was used again, this time with 4 increasing scaling factors **N=512, 1024, 2048, 4096**. The number of triples and the size of the input dataset quadruple for every successive scaling factor, as shown in Figure 3.9. The datasets were generated with the distributed synthetic generator `DistSynthGen` of Geographica benchmark series and each one of the 5 feature class N-Triples files was partitioned in 10 pieces using the automatic mode (value 0). Loader 2 run with both DDL statement options: `HiveQL` and `Spark SQL API`.

The results of the same Figure 3.9, assert most of the preliminary conclusions from the tests in the VM+HDFS platform:

- Loader 2 output database size is approximately an order of magnitude smaller than Loader 1 database size. The same holds for HiveQL and Spark SQL API.

[3]Manually partitioning and compressing, simulates a splittable input format for parallel ingestion under HFDS
[4]`https://hopsworks.polartep.io/hopsworks/`

Synthetic Dataset			Loader 1			Loader 2				
							Time		Output Size	
Scaling Factor (N)	Input Size	No Triples	Ingestion Format	Time	Output Size (MB)	Ingestion Format	HiveQL	Spark SQL API	HiveQL	Spark SQL API
512	778.5 MB	4,082,748	Text	13 min 22 sec	783.2 MB	Parquet	6 min 11 sec	*5 min 03 sec*	88.4 MB	*88.5 MB*
1024	3.0 GB	16,324,234	Text	20 min 27 sec	3.1 GB	Parquet	8 min 54 sec	*9 min 08 sec*	415.7 MB	*416.7 MB*
2048	12.2 GB	65,264,508	Text	40 min 20 sec	12.5 GB	Parquet	21 min 53 sec	*21 min 05 sec*	1.8 GB	*1.8 GB*
4096	49.2 GB	261,031,202	Text	86 min 12 sec	50.2 GB	Parquet	68 min 17sec	*78 min 14 sec*	7.3 GB	*7.3 GB*

Figure 3.9: Loader 2 - Loaders Comparison - PolarTEP Hopsworks

- Loader 2 is faster in all scaling factors and seems to scale more gracefully than Loader 1, as it requires less time for the same allocated cluster resources.
- The replication factor 1 is low and did not allow for increased parallelism opportunities.

3.2.4 Loader Source Code

Both Loaders are packaged in the same Maven Java POM project[5] which is available on Github. Each Loader has its own packages and different entry point class, which allowed for parallel development without interference on the logic. The latest version, as of this writing, is **v2.3.1**.

The Loaders were required to eventually run on the Hopsworks platform, but at the same time it was essential for development purposes, initially, to test the logic on a standard Hadoop cluster. This was also a logical step to take since the wide acceptance of the tool would benefit from availability on standard cluster platforms. Therefore two maven profiles were designed, each one targeting a different platform.

The **hdfs** profile, which is active by default, uses well known repositories and targets Hadoop `v2.10.0`, Spark `v2.4.5` and Hive `v2.3.7` which are the latest of the v2.x Hadoop ecosystem line. The **hops** profile, taps to additional Hopsworks repositories and targets the Hops customized Hadoop `v3.2.0.2`, Spark `v2.4.3.2` and Hive `v3.0.0.6`.

For parsing we used the RDF4J[6] `v3.3.1` and for the geospatial related transformations `v1.3.1` of the GeoSpark-Sedona library.

[5] `https://github.com/dbilid/PRoST`
[6] `https://rdf4j.org/`

4. Improvements in Query Executor

In this chapter we present some methods that aim to improve query execution times over the initial system design presented in Chapter 2. Specifically, we discuss how we have modified the system in order to use: i) a persistent spatial index and partitioning, ii) an optional cache that contains partitioned in-memory copies of the thematic tables of VP that originally reside in the HIVE database and iii) an optional cache that stores qualitative spatial relations based on the original geometries contained in the dataset. We also describe the details of the query optimizaton process in Strabo2. Finally we present some experimental results in order to evaluate the performance of Strabo2.

4.1 Using Persistent Spatial Indexing and Partitioning

According to the architecture that we described in Chapter 2, RDF data are stored in disk in HIVE database according to the VP schema and during query execution, the Spark execution engine loads the nessecary fragments in memory. Geometries have the same treatment. For example, if we want to apply a spatial selection, we have to read the geometries from the disk, build an in memory spatial index and/or partitioning during query execution time and discard this index/partitioning afterwards. If the next query is again a spatial selection this process has to be repeated. Unfortunately, this is an inherent issue of Apache Sedona when we access it from the SQL interface, due to the fact that clustered indexes cannot be defined in Spark-SQL [1]. In order to take advantage of persistent spatial indexes and partitioning we have implemented a hybrid translation to both the SQL and RDD/Scala interface.

To achieve this, we first create a persistent spatial structure usng the RDD/Scala interface of Sedona, and then for each query, we modify the intermediate translation that is in the form of a logic program rule, before the final translation into SQL, by identifying spatial FILTER clauses that can be evaluated efficiently using the persistent structure and then by replacing the atoms corresponding to thee specific spatial operation by temporary atoms that correspond to the intermediate result after accessing the persistent spatial structure. As an example, a persistent spatial index using a quad tree can be created in the distributed geometry dataset using the following code, which holds the result as a Spark RDD (spatialRDD):

```
var spatialDf = _sqlContext.sql(“SELECT entity, ST_GeomFromWKT(wkt) FROM geometries”)
spatialDf.registerTempTable(tableStat.tName)
spatialRDD = Adapter.toSpatialRdd(spatialDf, "wkt")
spatialRDD.buildIndex(IndexType.QUADTREE, false)
spatialRDD.indexedRawRDD.persist(StorageLevel.MEMORY_ONLY);
```

Then consider a query that asks for ice observations and the class name assigned to them, such that their geometries intersect a given polygon:

```
SELECT ?x ?ctName
WHERE {

?x rdf:type polaronto:IceObservation .
?x polaronto:hasCTClassName ?ctName .
```

[1] We had explicitly ask the developers of Apache Sedona (then named GeoSpark) about this issue and they had confirmed to us that we cannot achieve this from the SQL interface. (https://groups.google.com/g/geospark-discussion-board/c/p9y2kkQkYI4

```
?x geo:hasGeometry ?geo1 .
?geo1 geo:asWKT ?wkt .

FILTER(geof:sfIntersects(?wkt, "POLYGON((23.7 37.9,22.9 40.6,11.5 48.1,23.7 37.9))"
    ^^<http://www.opengis.net/ont/geosparql#wktLiteral>)).
}
```

The intermediate translation in terms of a logic program is given below:

```
ans1(URI("{}",t0_6),URI("{}",t1_7)) :-
    prop1(t0_6,"http://earthanalytics.eu/polar/ontology/IceObservation"),
    prop3(t0_6,t1_7),
    geometries(t0_6,t3_8,t4_8),
    SF-INTERSECTS(t4_8,GEOMFROMWKT(POLYGON((23.7 37.9,22.9 40.6,11.5 48.1,23.7 37.9))))
```

By identifying the spatial filter in the logic program we can see that we can use the spatialRDD for its evaluation. In order to do that, we are replacing the atoms $geometries(t0_6, t3_8, t4_8)$ and $SF-INTERSECTS(t4_8, GEOMFROMWKT(POLYGON((23.737.9, 22.940.6, 11.548.1, 23.737.9))))$ in this intermediate form with a new atom temp, that corresponds to the result of the access to the spatial index. The new logic program is the following:

```
ans1(URI("{}",t0_6),URI("{}",t1_7)) :-
    prop1(t0_6,"http://earthanalytics.eu/polar/ontology/IceObservation"),
    prop3(t0_6,t1_7),
    temp(t0_6)
```

Finally, we access the spatial index and take the result of the intersection with the given polygon, transform the result into a dataframe and save it in the temporary table with name *temp*.

```
val rangeQueryWindow = wktReader.read("POLYGON((23.7 37.9,22.9 40.6,11.5 48.1,23.7 37.9))")
                        .asInstanceOf[Polygon];
val considerBoundaryIntersection = true
val usingIndex = true
var queryResult = RangeQuery.SpatialRangeQuery(spatialRDD, rangeQueryWindow,
                considerBoundaryIntersection, usingIndex)
Adapter.toDf(queryResult).createGlobalTempView("temp")
```

4.2 Caching Partitioned Thematic Tables

In order to perform a join between two tables, Spark loads them into memory from HIVE and by default chooses between two different join strategies: distributed sort-merge join and broadcast join. Broadcast join is preferred when one of the two tables is very small in size (smaller than a given threshold), so it is replicated in every node in order to be joined with the larger table. In distributed sort-merge join the two tables are hash partitioned on the join key using the same hash method, so records from the first table that match records of the second are placed in the same node. After the hash partitioning, each partition of each table is sorted, and then a merge join is performed locally in every node.

When Ontop-spatial produces a query, given a specific join order of the tables and a specific threshold for the broadcast join, we can identify the partition and sorting operations that will take

place during the evaluation from Spark. Then, without any extra cost for the total evaluation of the query, we can first apply for each such table the partition and sorting, save the result in a temporary table, and then perform directly the joins in the query over these temporary tables. We then can keep the temporary tables in cache and use them for subsequent queries, if needed.

4.3 Caching Qualitative Spatial Relations Using JedAI-Spatial

JedAI-Spatial has been developed in the context of Task 3.2 of ExtremeEarth in order to perform spatial interlinking between different datasets. In our case, we can use JedAI-Spatial with the same dataset as both source and target dataset, in order to store the qualitative spatial relation between the entities of the dataset. As in the normal operation, we only need to store entities that present some spatial relation other than disjoint. After the execution of JedAI-Spatial, we create a table based on its results. This table contains 2 columns with the names of the entities that were found to have some spatial relation, and other nine columns with boolean values corresponding to the spatial relations:

- Contains
- CoveredBy
- Covers
- Crosses
- Equals
- Intersects
- Overlaps
- Touches
- Within

Of course, we do not have to store information that relates every entity with itself, as we can directly see which spatial relations hold in this case. Also, the disjoint relation can be tested by absence of the specific pair of geometry entities from this table. The construction of this table is an offline process, that can be optionally be employed in order to increase the efficiency of query evaluation in Strabo2, by avoiding quantitative computations with geometries during execution in case of spatial joins that involve an of the previously mentioned predicates, or the disjoint predcate. Unfortunately, distance join queries cannot take advantage of this table.

4.4 Query Optimization

In this chapter we describe the query optimizer that decides the execution plan of the query produced by Strabo2. The optimization works on two different levels. In the first level we decide the exact form of the produced query during query rewriting with respect to the ontology axioms, whereas in the second level we decide the exact join order for each subquery of the produced query that contains a series of thematic and spatial joins and filters. The Optimization in the first level is applicable to either Ontop-spatial or Ontop [RMKZ13] and is presented in detail in [BK21], which is also included in Appendix A.

4.4.1 Handling Redundancy During Query Rewriting

We now briefly describe the first level of optimization that takes place during query translation. By default, the final form of a query after rewriting in Ontop-spatial is a *union of conjunctive queries* (UCQ). This corresponds to an SQL query that is has as a top level operator a UNION with inputs different queries that have the form SELECT-FROM-WHERE. During query rewriting, the initial query over an ontology is rewritten in order to take into consideration the ontological axioms. The result of this process is a query, that when posed over the ABox alone (that is, by disregarding all the ontological axioms), will return the same answers as the initial query posed over the ontology. This is done using the notion of *certain answers*, that is answers present in every model of the ontology. During query unfolding the rewritten query is transformed into another query expressed in the query language of the underlying data sources (in our case Spark-SQL). During query rewriting, we use a cost-based method employed by the OBDA system during unfolding, for choosing the final form of the SQL query to be executed by the RDBMS. This method relies on heuristics that in turn rely only on factors known to the OBDA system, such as sizes of the relations, duplicates introduced by the mappings for each ontology term and selectivity estimation for simple CQs over the database, that are not affected by issues such as join ordering or access methods, and thus can be performed even from a system operating outside the RDBMS as long as some basic statistics about the tables have been obtained prior to the deployment of the system. Specifically, our method starts with the "fully" unfolded query produced by the method of [PLC+08] as the baseline, and uses the heuristics in order to "fold" back specific paths, when this is expected to be more effective. Each such fold corresponds to the creation of an intermediate table, as explained in the previous example. These heuristics are based on the notion of *redundant processing* between the union subqueries. We make a distinction between two kinds of redundant processing: i) duplicate answers and ii) repeated operations (disk reads and writes on the same data) from different union subqueries of the same query even in the absence of duplicate answers.

Regarding duplicates, using the standard set semantics for queries over ontologies, the final answer should be duplicate-free, but since RDBMSs (or Spark in our case) operate using the bag semantics, duplicates are often introduced during query evaluation. Duplicates can be introduced as different ways to obtain the same fact from the data, for example the same tuple may be produced from different mappings used for the same property or class assertion. Using the unfolding method from [PLC+08], this will result in duplicate answers coming from different union subqueries. But duplicates can be introduced even from a single mapping, in case the database relation already contains duplicate rows, or due to the projection operator in the SQL query in the body of the mapping. In this setting, duplicates are redundant answers whose impact can be detrimental for query evaluation, as the size of intermediate results can increase exponentially in the number of joins in the query. Even if the final SQL query produced by an OBDA system dictates that the result should be duplicate free using the SQL DISTINCT or UNION keyword, relational systems rarely consider early duplicate elimination in order to limit the size of intermediate results, but only perform the task on the final query result. This behavior is justified by the fact that duplicate elimination is a costly blocking operation [BD83] and also that the SQL queries are usually formulated by expert users who take into consideration the integrity constraints of normalized relational schemas. Under these assumptions, considering early duplicate elimination options during optimization is not usually regarded worthy. Contrary to this situation for SQL queries, it has been ascertained [KHJR+15] that in real world OBDA settings, duplicate answers frequently dominate query results and also that this appears as "noise" to end users that might be using a visual query formulation tool. For this reason we use a heuristic regarding early duplicate elimination, for duplicates introduced from a single mapping that we also extend for the case of duplicates that show up in different union subqueries, and use it to help us decide when to "fold" back specific branches of the unfolded query.

Regarding the second kind of redundant processing, this depends heavily on the exact execution plan that will be chosen by the RDBMS. As an example, consider the UCQ from Figure 4.1 and let us suppose that there are no duplicates (each fact for each ontology predicate can be obtained only from a single mapping). Also suppose that the RDBMS chooses to perform all the joins using

$$
\begin{aligned}
ans(f(x), g(y), h(w), k(z)) \leftarrow \\
A_1(x,y), A_3(x,v_1,w), C_1(y,z) \vee \\
A_2(x,y), A_3(x,v_1,w), C_1(y,z) \vee \\
A_3(x,y,v_2), A_3(x,v_1,w), C_1(y,z) \vee \\
A_1(x,y), A_3(x,v_1,w), C_2(y,z) \vee \\
A_2(x,y), A_3(x,v_1,w), C_2(y,z) \vee \\
A_3(x,y,v_2), A_3(x,v_1,w), C_2(y,z)
\end{aligned}
$$

Figure 4.1: UCQ over the database

index-based nested loops, using for the first three subqueries the table C_1 as the leftmost table and for the next three subqueries the table C_2 as the leftmost table. In this case, the redundant processing is equal to the two scans of table C_1 plus the two scans of table C_2 (ignoring the possible impact of the memory cache). If there was no redundant processing, then it is reasonable to assume that this form of the query would be the most efficient translation, as it consists of simple CQs which the RDBMS can efficiently optimize and probably evaluate in parallel. But since we have redundant processing, one would expect that it would be more efficient to first compute and save the temporary union table corresponding to the three mappings for P_1, if the RDBMS will again choose to perform index-based nested loops and the cost for creating and saving the temporary result is smaller than the cost of the initial redundant processing. As all these possible execution plans cannot be known to the OBDA system, for this case of redundant processing, we use a criterion according to which temporary tables are created in a “conservative" manner, only when it is almost certain that this decision will lead to smaller execution cost.

Our optimization method offers efficient solutions to the problem of handling redundancy in the execution of the queries produced by Ontop-spatial. Specifically, we use a heuristic regarding early duplicate elimination in duplicates introduced from a single mapping (that is for each union subquery of the final SQL query separately). This heuristic is evaluated over four different RDBMSs and have shown that its usage is justified and that for query mixes from two different used benchmarks, such that low selectivity queries do not dominate execution time, it can lead to overall improvement of up to 25% compared to the strategy of always performing duplicate elimination. Furthermore, we enhance the unfolding step previously described in the literature with cost-based decisions regarding the redundant processing, obtaining a full cost-based method for OBDA query translation and we extend the heuristic in order to deal with duplicate answers coming from different union subqueries. We also take into consideration other forms of redundant processing in the form of repeated operations. As mentioned, the detailed description is presented in Appendix A.

4.4.2 Execution Order of Thematic and Spatial Joins and Filters

Regarding the second level of optimization, this operates in the sequence of CQs that have been obtained after the query rewriting. We consider that each of these queries contains a series of thematic and spatial filters and joins. If the initial query contains more operators, for example nested sub-queries, these are optimized as a separate query. The query optimizer takes as input information about the existence of persistent spatial indexing and/or partitioning, the existence of cache that contains partitioned thematic tables on either the subject or object column, and the existence of qualitative spatial information that can be used to avoid computations with geometries as presented in Chapter 4. It also uses a selectivity and a cost estimator that return information about the size and the execution cost of a specific operator.

The query optimizer implemented in Strabo2 is based on the dynamic programming method proposed in [SAC$^+$89], where existence of partitioning and sorting in the cache is treated as a physical

Query	Execution Time (ms)	Number Of Results
Q1	30618	0
Q2	16595	0
Q3	246825	1
Q4	57768	1
Q5	648263	8
Q6	404554	0
Q7	761601	1
Q8	132831	0
Q9	378933	2
Q10	73941	0
AVG.	275192	

Table 4.1: Execution Times for Invekos and Lucas Datasets

data property, and we examine two different thematic join options for each join: broadcast join in case the size of one of the two tables in smaller than a given threshold, and sort-merge join otherwise. Also, thematic filters are always pushed down to be performed during the scan of the base table. On the other hand, spatial selections can be executed later in the execution plan, in case the original data are not spatially indexed.

4.5 Experimental Results

We now present some preliminary experimental results. These results present the execution times in Hopsworks for the Strabo2 executor, such that we are not using any kind of caching or persistent spatial indexing and partitioning as presented in Sections 4.1, 4.2 and 4.3. We will report the final experimental evaluation, which includes experiments evaluating the impact of these improvements, in deliverable D3.5 in month 36.

4.5.1 Experiments With Datasets From Food Security Use-Case

The Invekos and LUCAS datasets from the food security use-case was used for our first set of experiments. Invekos is published by Austrian administration's Land Parcel Identification System and it contains crop-types for different areas, as they are declared by the land owners. LUCAS is the 2018 Land use and cover area frame statistical survey. These datasets have been transformed into RDF and we have used 10 SPARQL queries that were formulated with help from partners VISTA, UNITN and NCSR-D. The exact SPARQL queries are presented in Appendix B. The transformed datasets contains 16 million triples, and its size in NTRIPLES format is about 4.9 GB.

We have loaded the dataset in Hopsworks and succesfully executed the queries using 8 workers, with 4GB memory per worker. The results of the execution are shown in Table 4.1. As seen, the average execution time is about 275 seconds. This is reasonable, as most of these queries are complex, involving nested subqueries and distance joins between multiple geometries. As most of these queries are searching for nearest points in the two datasets, the return few results, but their execution involves heavy spatial processing in the form of distance joins.

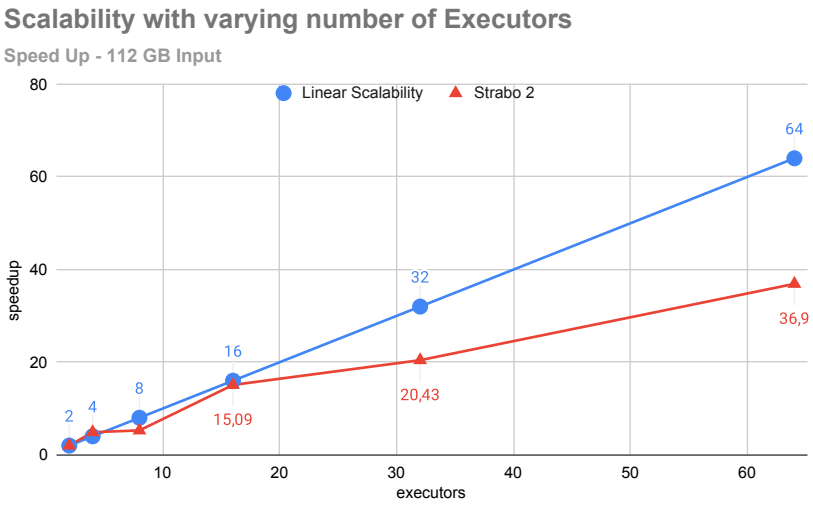


Figure 4.2: Execution with Varying Number of Executors

4.5.2 Experiments With Synthetic Dataset of Geographica2 Benchmark

We now present experimental evaluation with the synthetic dataset of the Geographica2 benchmark, which contains 36 queries of spatial selections and spatial joins, with different spatial predicates and selectivities. The dataset contains information about land ownerships, roads, points of interest and states in the form of small polygons, linestrings, points and large polygons.

Results for Scaling Factor 12228 We have generated datasets of different size. The largest dataset was generated for scaling factor 12228 which result in a size of 450 GB in NTRIPLES files. For this specific dataset we have successfully executed the 36 queries in a cluster with 128 executors, with 4 GB of memory pes executor, with execution times and results as shown in Table 4.2.

Scalability Experiments In order to evaluate the scalability of Strabo2 we have executed experiments with a varying number of worker nodes, and also with a varying dataset size. The results of the first set of experiments are shown in Figure 4.2, where we have executed the 36 queries of the benchmark in an inputa dataset of about 112 GB in NTRIPLES files, with 2, 4, 8, 16, 32 and 64 executors, while the results of the second set of experiments are shown in Figure 4.3, where we are using 64 executors in order to execute the 36 queries in datasets of increasing size, starting from 10 GB up to 450 GB. In both experiments the scalability of the system appears very promising, but further experimentation is needed in order to further evaluate the ability of the system to scale to larger datasets.

Query	Execution Time (ms)	Number Of Results
Q0	102017	150994944
Q1	52628	147456
Q2	51038	113289414
Q3	46428	106440
Q4	43519	75529136
Q5	31287	69528
Q6	32639	37761025
Q7	34392	36870
Q8	27823	15114600
Q9	22656	11664
Q10	31227	153077
Q11	22636	0
Q12	503528	268441611
Q13	554069	239596
Q14	70287	262081
Q15	137656	68844
Q16	182726	108284970
Q17	54156	99776
Q18	69911	99776
Q19	13994	45568
Q20	73043	150994944
Q21	35269	147456
Q22	41802	113249647
Q23	35725	110595
Q24	50883	75500911
Q25	36964	73731
Q26	35941	37752175
Q27	23851	36867
Q28	30834	15099247
Q29	22482	14745
Q30	29165	150895
Q31	34038	147
Q32	264832	150982657
Q33	381391	141305
Q34	314095	147443
Q35	45809	135
AVG.	98353	

Table 4.2: Execution Times for Geographica2 Synthetic Dataset 12228

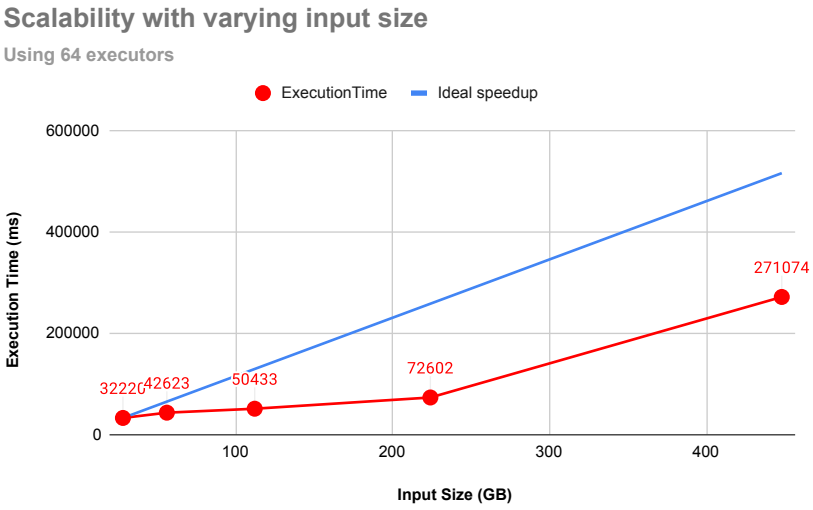


Figure 4.3: Execution with Varying Size of Input Dataset

5. Distributed Endpoint

Strabo2 executor operates as a Spark job, using the spark-submit command, with arguments that include the HIVE database name where Strabo2 loader has saved the dataset and the directory that contains a set of GeoSPARQL queries to be executed. Strabo2 Executor then translates each query and sends the resulted translation Spark, using Spark-SQL, as desribed in Chapter 2. The execution of the spark-submit command in Hospworks is monitored by Hops-YARN, which coordinates the deployment of resources with respect to the Hopsworks cluster. In this context, Strabo2 executor is submitted as a single jar file that contains all necessary libraries, including Sedona. During system initialization, the geospatial functions of Sedona are registered as UDFs in the Spark engine, so that they can be used in subsequent queries. In this chapter we discuss the implementation of a SPARQL endpoint for Strabo2 in Hopsworks, in order to use the query executor as a service that listens to requests for query execution.

5.1 Implementation of A SPARQL Endpoint in Hopsworks

The mode of execution of Strabo2 is prohibitive for interactive communication with Strabo2, where the system operates as a service that first is initialized and then it is ready to accept user queries that may come from a data visualization module like Sextant, from a federation engine like Semagrow, or from any other compatible client that communicates with the Strabo2 service. In order to be able to allow interactive communication, we have developed an HTTP SPARQL endpoint implementing the SPARQL Protocol[1], executed as a web application. This web application needs to execute spark jobs, but obviously cannot be executed through the spark-submit command. Spark jobs can be executed in this setting when operating in the Spark client mode, but in order to use the Hopsworks Spark installation and also the Hops-YARN resource manager it is necessary to use the Spark cluster mode.

As a solution in this problem, in order to communicate with Hops-YARN and the cluster installation of Spark from the web application that contains the Strabo2 endpoint, we use communication through a REST interface using Apache Livy[2]. Strabo2 endpoint implements all communication with Spark through Livy. Upon requesting a new session from Livy, the user can define the number and kind of resources needed like the number of executors, the amount of memory, the number of CPU cores, etc.

5.2 GeoSpark Function Registration using Apache Livy

In order to be able to use Sedona, the appropriate jars were placed in the installation of the endpoint and were passed to Spark through the extra jars setting, which dictates the Spark master to ship the mentioned jars to the worker nodes. Furthermore, it is necessary to register the spatial user data types (UDTs) defined by Sedona. This must be done during the initialization of the Spark application. For this reason, we implemented a class that extends the class SparkListener. This class listens to events from the Spark scheduler, and defines code that will be executed during the lifecycle of the application. The following code is included in the implemented listener, that upon application startup registers the spatial UDTs:

```
class GeoSparkRegistratorListener extends SparkListener
```

[1] https://www.w3.org/TR/sparql11-protocol/
[2] https://livy.apache.org/

```
{
    override def onApplicationStart(ev: SparkListenerApplicationStart): Unit = {
    UDTRegistration.register("com.vividsolutions.jts.geom.Geometry",
        "org.apache.spark.sql.geosparksql.UDT.GeometryUDT")
    UDTRegistration.register("com.vividsolutions.jts.index.SpatialIndex",
        "org.apache.spark.sql.geosparksql.UDT.IndexUDT")
}
```

We have included this implementation to the extra jars setting in order to be available from the application. After initializing a Spark session through LIVY, we also register all the spatial user defined functions of Sedona using specific Spark-SQL requests. For example, for the ST_Intersection UDF the followng command is sent to Livy:

```
{"code":
"spark.sessionState.functionRegistry.createOrReplaceTempFunction(\"ST_Intersection\",
            org.apache.spark.sql.geosparksql.expressions.ST_Intersection);"}
```

Once the endpoint has been deployed, the Hopsworks user responsible for the deployment can use the platform's user interface to monitor and manage the submitted Spark jobs, as with normal Spark applications. Furthermore, a docker module of the endpoint has been implemented in order to facilitate and automate the deployment in the platform.

6. Summary and Future Work

In this deliverable we presented the system Strabo2 that was developed during the Task T3.3 of the ExtremeEarth project. Strabo2 efficiently stores compressed RDF data in a HIVE database using the vertical partitioning schema, and uses the query translation mechanism of Ontop-spatial in order to produce a Spark SQL query from an initial GeoSPARQL query. The produced query is executed by Spark, extended with spatial functionality from the Sedona framework. Strabo2 also offers the opportunity to take advantage of persistent spatial indexing and partitioning offered by the RDD API of Sedona. Caching of partitioned thematic tables and qualitative spatial relations is also optionally offered by the system in order to improve query execution. Strabo2 has been succesfully deployed in the Hopsworks platform running on the CREODIAS [1] premises, and an endpoint based on Apache Livy has been developed, in order to communicate with the Hops-YARN resource manager, and execute on demand jobs by accepting requests for specific GeoSPARQL queries. Strabo2 has been tested in a cluster with up to 128 worker nodes, successfully storing and querying RDF datasets of up to 450 GB. Also, scalability experiments show that it is expected the system to efficiently scale to larger datasets and clusters.

In the remaining time of the project (M31-M36), we will concentrate on the extensive evaluation of Strabo2 using the latest version of the benchmark Geographica to be developed in Task 3.5 of WP3 containing both synthetic and real-world datasets. We plan to import and process even larger datasets and more thoroughly investigate the scalability of the system. We also want to evaluate each of the improvements that have been described in this deliverable and quantify their impact in the overall query execution time. This work will be reported in Deliverable D3.5 (it has been originally planned for the present deliverable but we had to delay it so that the latest version of the benchmark Geographica would be ready).

[1] https://creodias.eu/

Bibliography

[BD83] Dina Bitton and David J DeWitt. Duplicate record elimination in large data files. *ACM Transactions on database systems (TODS)*, 8(2):255–265, 1983.

[BK21] Dimitris Bilidas and Manolis Koubarakis. Handling redundant processing in obda query execution over relational sources. *Journal of Web Semantics*, 68:100639, 2021.

[BXK19] Konstantina Bereta, Guohui Xiao, and Manolis Koubarakis. Ontop-spatial: Ontop of geospatial databases. *Journal of Web Semantics*, 58:100514, 2019.

[CFL18] Matteo Cossu, Michael Färber, and Georg Lausen. PRoST: distributed execution of SPARQL queries using mixed partitioning strategies. *arXiv preprint arXiv:1802.05898*, 2018.

[IGK+21] Theofilos Ioannidis, George Garbis, Kostis Kyzirakos, Konstantina Bereta, and Manolis Koubarakis. Evaluating geospatial RDF stores using the benchmark geographica 2. *Journal on Data Semantics*, pages 1–40, 2021.

[KHJR+15] Evgeny Kharlamov, Dag Hovland, Ernesto Jiménez-Ruiz, Davide Lanti, Hallstein Lie, Christoph Pinkel, Martin Rezk, Martin G Skjæveland, Evgenij Thorstensen, Guohui Xiao, Dmitriy Zheleznyakov, and Ian Horrocks. Ontology based access to exploration data at Statoil. In *International Semantic Web Conference*, pages 93–112. Springer, 2015.

[KKK12] Kostis Kyzirakos, Manos Karpathiotakis, and Manolis Koubarakis. Strabon: a semantic geospatial dbms. In *International Semantic Web Conference*, pages 295–311. Springer, 2012.

[PLC+08] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. In *Journal on data semantics X*, pages 133–173. Springer, 2008.

[RMKZ13] Mariano Rodriguez-Muro, Roman Kontchakov, and Michael Zakharyaschev. Ontology-based data access: Ontop of databases. In *International Semantic Web Conference*, pages 558–573. Springer, 2013.

[SAC+89] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Readings in Artificial Intelligence and Databases*, pages 511–522. Elsevier, 1989.

[SPSL16] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF querying with SPARQL on spark. *PVLDB*, 9(10):804–815, 2016.

[WSK+03] Kevin Wilkinson, Craig Sayers, Harumi A Kuno, Dave Reynolds, et al. Efficient rdf storage and retrieval in jena2. In *SWDB*, volume 3, pages 131–150. Citeseer, 2003.

[YWS15] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 70. ACM, 2015.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

Appendix A

This appendix contains the paper:

- Bilidas, D., & Koubarakis, M. (2021). Handling redundant processing in OBDA query execution over relational sources. *Journal of Web Semantics, 68, 100639.*

Handling Redundant Processing in OBDA Query Execution Over Relational Sources

This is a pre-print of an article published in Journal of Web Semantics.
The final authenticated version is available online at:
https://doi.org/10.1016/j.websem.2021.100639

Handling Redundant Processing in OBDA Query Execution Over Relational Sources

Dimitris Bilidas[a,*], Manolis Koubarakis[a]

[a]Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Panepistimiopolis, Ilissia, Athens 15784, Greece

Abstract

Redundant processing is a key problem in the translation of initial queries posed over an ontology into SQL queries, through mappings, as it is performed by ontology-based data access systems. Examples of such processing are duplicate answers obtained during query evaluation, which must finally be discarded, or common expressions evaluated multiple times from different parts of the same complex query. Many optimizations that aim to minimize this problem have been proposed and implemented, mostly based on semantic query optimization techniques, by exploiting ontological axioms and constraints defined in the database schema. However, data operations that introduce redundant processing are still generated in many practical settings, and this is a factor that impacts query execution. In this work we propose a cost-based method for query translation, which starts from an initial result and uses information about redundant processing in order to come up with an equivalent, more efficient translation. The method operates in a number of steps, by relying on certain heuristics indicating that we obtain a more efficient query in each step. Through experimental evaluation using the Ontop system for ontology-based data access, we exhibit the benefits of our method.

Keywords: Query Translation, Data Integration, Ontology-Based Data Access, Ontop

1. Introduction and Outline

Ontology Based Data Access (OBDA) is a database technique in which an ontology is linked to underlying data sources through mappings. An end user can pose queries over the ontology, which we assume to represent a familiar vocabulary and conceptualization of the user domain. The OBDA system automatically translates the query and sends it for execution to the underlying data sources. This approach provides the end user with a convenient abstraction over possibly complex schemas and details about the data storage and query processing. The query translation involves query rewriting and query unfolding. During query rewriting, an initial query over an ontology is rewritten in order to take into consideration the ontological axioms. The result of this process is a query that when posed over property and class assertions only (that is, by disregarding all the ontological axioms), will return the same answers as the initial query posed over the ontology. This is done using the notion of certain answers, that is, answers present in every model of the ontology. During query unfolding the rewritten query is transformed into another query expressed in the query language of the underlying data sources. In what follows we consider an OBDA setting, where an OWL 2 QL ontology is linked through mappings to data stored in a relational database management system (RDBMS). This method provides the user with access to a virtual RDF graph. The original query is a conjunctive query expressed over the vocabulary of the virtual RDF graph, and the result of rewriting and unfolding is a SQL query.

Example 1. As an example of OBDA setting consider a relational schema that contains the relational tables A_1, A_2, A_3, C_1 and C_2 and the mappings from Figure 1. In these mappings P_1, Q_1, R_1, P_2, P_3

*Corresponding author

Email addresses: d.bilidas@di.uoa.gr (Dimitris Bilidas), koubarak@di.uoa.gr (Manolis Koubarakis)

and Q_3 are properties defined in the ontology, whereas f, g, h and k are functions that construct ontology objects from database values. These functions are responsible for constructing an object that acts as an ontology individual out of values occurring in the database. In our setting, they construct an RDF term. A query posed over the ontology can be the following: $ans(x, y, w, z) \leftarrow P_1(x, y), P_2(x, w), P_3(y, z)$.

The notion of OBDA as we describe it, was presented in [21]. There, the result of query rewriting of an initial conjunctive query (CQ) over the ontology is a union of conjunctive queries (UCQ) over the vocabulary of the ontology. Then, the authors define a faithful representation of this UCQ, along with the mappings and database instance in terms of a logic program. Query unfolding is based on partial evaluation of such logic programs, and as final result it produces a query which can be viewed as an SQL query. More details about this process are given in Section 3. Subsequent research was focused on more efficient rewritings in the form of UCQs over the ontology [14, 6, 20]. The main aim of these approaches was to produce a UCQ with as few subqueries as possible, as it was observed that the number of union subqueries in the result of query rewriting could be very large. A different approach was followed in [3], where a cost-based comparison of different reformulations is carried out, considering that no mappings are used and the ABox is directly stored in the external database. In general, the final query in this case will be an SQL query that contains joins over UCQs (JUCQs). An extension of this work for arbitrary relational schemas, so that it also takes into consideration the unfolding step with arbitrary mappings, is presented in [16].

Regarding the implementation of OBDA systems, it has been observed that in practice it is more efficient to compile ontological knowledge regarding class and property hierarchies into the mappings, and ignore such axioms during query rewriting. For this reason, Ultrawrap-OBDA[26] uses the notion of saturated mappings and Ontop[4] uses the so called $\mathcal{T}$-Mappings [22]. For example, consider the setting of Example 1 and an ontology that contains the following axioms: $Q_1 \sqsubseteq P_1, R_1 \sqsubseteq P_1$ and $Q_3 \sqsubseteq P_3$. We can ignore the axiom $Q_1 \sqsubseteq P_1$ during rewriting if we add to the original mappings the mapping $m1'$ from Figure 2, and similar for the other two axioms.

In [22] three main reasons are specified for the presence of a large number of union subqueries in the result of query translation: i) ontological queries with existentially quantified variables that can lead to rewritings of exponential size, ii) large ontological hierarchies and iii) multiple mappings for each ontology term. Also, the authors notice that the first reason is rarely observed in real-world ontologies and queries. As a result, when compiling ontological information about hierarchies into the mappings, as for example in the Ontop $\mathcal{T}$-mappings, the last two important reasons that lead to a large number of subqueries are encountered during query unfolding. As an example, consider the query from Example 1 posed over the previously specified OBDA setting and $\mathcal{T}$-mappings. The unfolding method from [21] will produce a UCQ over the database that contains six union subqueries as shown in Figure 3. Each subquery corresponds to a different combination of the three mappings defined for P_1 with the two mappings defined for P_3. One can easily see that in case of queries with many atoms posed over large hierarchies, the final UCQ can contain hundreds or thousands of subqueries. On the other hand, a different unfolding method could choose to first compute as intermediate results the queries that correspond exactly to the first and third atoms of the initial query. In the specific example, the first temporary result would be a union query over tables A_1, A_2 and A_3 and the second temporary result would be a union query over tables C_1 and C_2. The final result would be a join of UCQs. Finally, one could choose an intermediate strategy, that would compute only one of these two intermediate results. Clearly, a cost-based decision should be made by the OBDA system regarding which exactly of these intermediate results should be computed, and if the overhead from computing and saving these results is counterbalanced from the gain in the final query.

Unfortunately, uncertainty about query execution costs is an inherent problem in data integration, where the mediator system (in our case the OBDA system) operates outside the database engine[11], as knowing all the factors that affect query execution is difficult or even impossible. For example, these factors include the exact execution plan that will be chosen by the RDBMS, including the access methods for each base relation and the join order in a join query, hardware characteristics like the amount of available memory and disk throughput, the disk block size, the exact details of the database physical design, like the existing indexes and the kind of each index and several other

$$
\begin{aligned}
m1 &: A_1(v_1^{m1}, v_2^{m1}) \rightarrow P_1(f(v_1^{m1}), g(v_2^{m1})) \\
m2 &: A_2(v_1^{m2}, v_2^{m2}) \rightarrow Q_1(f(v_1^{m2}), g(v_2^{m2})) \\
m3 &: A_3(v_1^{m3}, v_2^{m3}, v_3^{m3}) \rightarrow R_1(f(v_1^{m3}), g(v_2^{m3})) \\
m4 &: A_3(v_1^{m4}, v_2^{m4}, v_3^{m4}) \rightarrow P_2(f(v_1^{m4}), h(v_3^{m4})) \\
m5 &: C_1(v_1^{m5}, v_2^{m5}) \rightarrow P_3(g(v_1^{m5}), k(v_2^{m5})) \\
m6 &: C_2(v_1^{m6}, v_2^{m6}) \rightarrow Q_3(g(v_1^{m6}), k(v_2^{m6}))
\end{aligned}
$$

Figure 1: Example Mappings

$$
\begin{aligned}
m1 &: A_1(v_1^{m1}, v_2^{m1}) \rightarrow P_1(f(v_1^{m1}), g(v_2^{m1})) \\
m1' &: A_2(v_1^{m1'}, v_2^{m1'}) \rightarrow P_1(f(v_1^{m1'}), g(v_2^{m1'})) \\
m1'' &: A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}) \rightarrow P_1(f(v_1^{m1''}), g(v_2^{m1''})) \\
m2 &: A_2(v_1^{m2}, v_2^{m2}) \rightarrow Q_1(f(v_1^{m2}), g(v_2^{m2})) \\
m3 &: A_3(v_1^{m3}, v_2^{m3}, v_3^{m3}) \rightarrow R_1(f(v_1^{m3}), g(v_2^{m3})) \\
m4 &: A_3(v_1^{m4}, v_2^{m4}, v_3^{m4}) \rightarrow P_2(f(v_1^{m4}), h(v_3^{m4})) \\
m5 &: C_1(v_1^{m5}, v_2^{m5}) \rightarrow P_3(g(v_1^{m5}), k(v_2^{m5})) \\
m5' &: C_2(v_1^{m5'}, v_2^{m5'}) \rightarrow P_3(g(v_1^{m5'}), k(v_2^{m5'})) \\
m6 &: C_2(v_1^{m6}, v_2^{m6}) \rightarrow Q_3(g(v_1^{m6}), k(v_2^{m6}))
\end{aligned}
$$

Figure 2: Example $\mathcal{T}$-Mappings

factors.

On the other hand, one could expect that the RDBMS is capable of optimizing the produced query, since it performs query planning and optimization by taking into consideration the aforementioned parameters. Unfortunately, database engines focus on optimization of certain aspects of queries, including join ordering of multi-join queries, optimization of aggregate functions, access methods for each relation, etc. Queries produced by OBDA systems have some characteristics that are not regularly encountered on human-written queries for database applications. One such characteristic is the occurrence of common subexpressions in different parts of the query, for example in different subqueries of a union query. As we saw, the number of these subexpressions and subqueries can be very large. Although common subexpression identification (and in the case of multiple queries the related multi-query optimization area) have long been investigated in database research and implemented in database prototypes [25, 19, 24], to the best of our knowledge these methods have not become integral part of commercial RDBMSs, due to the increase in optimization time and the complexity introduced to the query optimizer. But since these common subexpressions are created during query translation, the OBDA system has the knowledge about them that can be taken into consideration to produce the final SQL query. Furthermore, it has been observed [16] that by using knowledge from the mappings, we can compute during system setup some parameters that will help us obtain more accurate selectivity estimations. For example, in our approach, a crucial factor that must be used when deciding about the exact form of the final SQL query, is the number of duplicates contained in the mappings used during unfolding for each ontology predicate. For example, for predicate P_1 of the query given in the previous example, it is crucial to know the number of duplicates rows in tables A_1, A_2 and the table obtained by selecting the first two columns of table A_3. The OBDA system knows from the mappings for which such

$$
\begin{aligned}
ans(f(x), g(y), h(w), k(z)) \leftarrow \\
A_1(x, y), A_3(x, v_1, w), C_1(y, z) \vee \\
A_2(x, y), A_3(x, v_1, w), C_1(y, z) \vee \\
A_3(x, y, v_2), A_3(x, v_1, w), C_1(y, z) \vee \\
A_1(x, y), A_3(x, v_1, w), C_2(y, z) \vee \\
A_2(x, y), A_3(x, v_1, w), C_2(y, z) \vee \\
A_3(x, y, v_2), A_3(x, v_1, w), C_2(y, z)
\end{aligned}
$$

Figure 3: UCQ over the database

columns and tables it should collect such information as an one-time task prior to query execution. On the other hand, an RDBMS cannot accurately estimate the number of duplicates in seemingly unrelated tables and columns during query execution.

Given the previous observations, in this work we propose a cost-based method employed by the OBDA system during unfolding for choosing the final form of the SQL query to be executed by the RDBMS. This method relies on heuristics that in turn rely only on factors known to the OBDA system, such as sizes of the relations, duplicates introduced by the mappings for each ontology term and selectivity estimation for simple CQs over the database, that are not affected by issues such as join ordering or access methods, and thus can be performed even from a system operating outside the RDBMS as long as some basic statistics about the tables have been obtained prior to the deployment of the system. Specifically, our method starts with the "fully" unfolded query produced by the method of [21] as the baseline, and uses the heuristics in order to "fold" back specific paths, when this is expected to be more effective. Each such fold corresponds to the creation of an intermediate table, as explained in the previous example. These heuristics are based on the notion of redundant processing between the union subqueries. We make a distinction between two kinds of redundant processing: i) duplicate answers and ii) repeated operations (disk access regarding the same data) from different union subqueries of the same query even in the absence of duplicate answers.

Regarding duplicates, using the standard set semantics for queries over ontologies, the final answer should be duplicate-free, but since RDBMSs operate using the bag semantics, duplicates are often introduced during query evaluation. Duplicates can be introduced as different ways to obtain the same fact from the data, for example the same tuple may be produced from different mappings used for the same property or class assertion. Using the unfolding method from [21], this will result in duplicate answers coming from different union subqueries. But duplicates can be introduced even from a single mapping due to the projection operator in the SQL query in the body of the mapping. In this setting, duplicates are redundant answers whose impact can be detrimental for query evaluation, as the size of intermediate results can increase exponentially in the number of joins in the query. Even if the final SQL query produced by an OBDA system dictates that the result should be duplicate-free using the SQL DISTINCT or UNION keyword, relational systems rarely consider early duplicate elimination in order to limit the size of intermediate results, but only perform the task on the final query result. This behavior is justified by the fact that duplicate elimination is a costly blocking operation [2] and also that the SQL queries are usually formulated by expert users who take into consideration the integrity constraints of normalized relational schemas. Under these assumptions, considering early duplicate elimination options during optimization is not usually regarded worthy. Contrary to this situation for SQL queries, it has been ascertained [13] that in real-world OBDA settings, duplicate answers frequently dominate query results and also that this appears as "noise" to end users that might be using a visual query formulation tool. In the previous version of this work [1], we introduced a heuristic regarding early duplicate elimination, for duplicates introduced from a single mapping. In this version we extend this heuristic for the case of duplicates that show up in different union subqueries, and use it to help us decide when to "fold" back specific

branches of the unfolded query.

Regarding the second kind of redundant processing, this depends heavily on the exact execution plan that will be chosen by the RDBMS. As an example, consider the UCQ from Figure 3 and let us suppose that there are no duplicates (each fact for each ontology predicate can be obtained only once from a single mapping). Also suppose that the RDBMS chooses to perform all the joins using index-based nested loops, using for the first three subqueries the table C_1 as the leftmost table and for the next three subqueries the table C_2 as the leftmost table. In this case, the redundant processing is equal to the two scans of table C_1 plus the two scans of table C_2 (ignoring the possible impact of the memory cache). If there was no redundant processing, then it is reasonable to assume that this form of the query would be the most efficient translation, as it consists of simple CQs which the RDBMS can efficiently optimize and probably evaluate in parallel. But since we have redundant processing, one would expect that it would be more efficient to first compute and save the temporary union table corresponding to the three mappings for P_1, if the RDBMS will again choose to perform index-based nested loops and the cost for creating and saving the temporary result is smaller than the cost of the initial redundant processing. As all these possible execution plans cannot be known to the OBDA system, for this case of redundant processing, we use a criterion according to which temporary tables are created in a "conservative" manner, only when it is almost certain that this decision will lead to smaller execution cost.

In this work we present efficient solutions to the problem of handling redundancy, considering ontologies belonging to the OWL 2 QL language[1], as the W3C recommendation for query answering against datasets stored in relational back-ends. Nevertheless, several aspects of this work can be considered for other ontology languages as well. As mentioned, an early version of this work was presented in [1], where a heuristic was presented for early duplicate elimination in duplicates introduced from a single mapping (that is for each union subquery of the final SQL query separately). This heuristic was evaluated over four different RDBMSs and it was shown that its usage is justified and that for query mixes from two different used benchmarks, such that low selectivity queries do not dominate execution time, it can lead to overall improvement of up to 25% compared to the strategy of always performing duplicate elimination. The main contributions of the present work, extending this previous version in several aspects, are as follows:

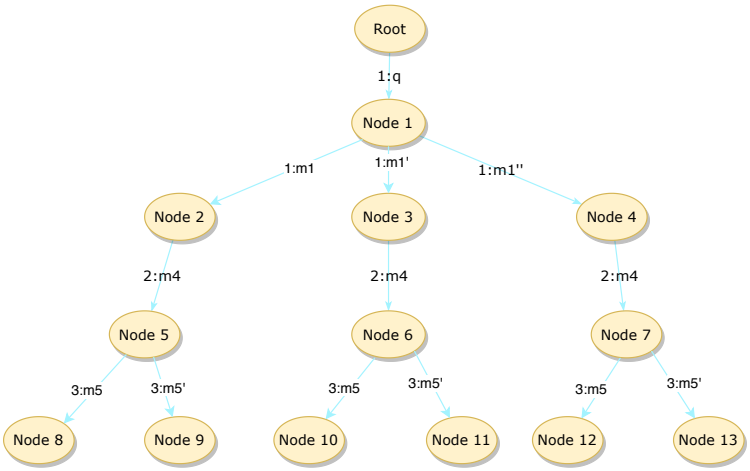
- We enhance the unfolding step previously described in the literature with cost-based decisions regarding the redundant processing, obtaining a full cost-based method for OBDA query translation (Section 3).
- We extend the heuristic in order to deal with duplicate answers coming from different union subqueries (Section 4).
- We take into consideration other forms of redundant processing in the form of repeated operations (Section 4).
- We implement our method for cost-based translation by modifying the state of the art OBDA system Ontop [22] and we perform extended experimental evaluation (Section 5).

The organization of this paper is as follows. We start by providing some preliminaries regarding ontologies, mappings, relational databases and logic programs (Section 2). In Section 3 we modify the unfolding method from [21], which is based on partial evaluation of logic programs, in order to explore equivalent results given that certain mappings have been replaced by a combined mapping which we define. In Section 4 we describe the cost-based decisions and we present the algorithm that incorporates them in the unfolding process. In Section 5 we present experimental evaluation of our implementation using the Ontop OBDA framework and the NPD and LUBM benchmarks. We also use the Wisconsin benchmark to compare our results with the results of [16]. In Section 6 we present relevant work and conclusions.

2. Preliminaries

We consider the following pairwise disjoint alphabets: Σ_O of ontology predicates, Σ_R of database relation predicates, Const of constants, Var of variables and Λ of function symbols, where each function symbol has an associated arity. We also consider that Const is partitioned into $\mathit{DB}_{\mathit{Const}}$ of database constants and $\mathit{O}_{\mathit{Const}}$ of ontology constants.

[1] https://www.w3.org/TR/owl2-profiles/



$$Root : ans(x,y,z)\theta_0 \leftarrow ans(x,y,z)$$
$$\theta_0 = \{\}$$
$$Node1 : ans(x,y,z)\theta_0\theta_1 \leftarrow P_1(x,y), P_2(x,h(A)), P_3(y,z)$$
$$\theta_1 = \{\}$$
$$Node2 : ans(x,y,z)\theta_0\theta_1\theta_2 \leftarrow A_1(v_1^{m1}, v_2^{m1}), P_2(f(v_1^{m1}), h(A)), P_3(g(v_2^{m1}), z)$$
$$\theta_2 = \{x/f(v_1^{m1}), y/g(v_2^{m1})\}$$
$$Node3 : ans(x,y,z)\theta_0\theta_1\theta_3 \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), P_2(f(v_1^{m1'}), h(A)), P_3(g(v_2^{m1'}), z)$$
$$\theta_3 = \{x/f(v_1^{m1'}), y/g(v_2^{m1'})\}$$
$$Node4 : ans(x,y,z)\theta_0\theta_1\theta_4 \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), P_2(f(v_1^{m1''}), h(A)), P_3(g(v_2^{m1''}), z)$$
$$\theta_4 = \{x/f(v_1^{m1''}), y/g(v_2^{m1''})\}$$
$$Node5 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), P_3(g(v_2^{m1}), z)$$
$$\theta_5 = \{v_1^{m_4}/v_1^{m1}, v_3^{m_4}/A\}$$
$$Node6 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6 \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), P_3(g(v_2^{m1'}), z)$$
$$\theta_6 = \{v_1^{m_4}/v_1^{m1'}, v_3^{m_4}/A\}$$
$$Node7 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7 \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), P_3(g(v_2^{m1''}), z)$$
$$\theta_7 = \{v_1^{m_4}/v_1^{m1''}, v_3^{m_4}/A\}$$
$$Node8 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5\theta_8 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), C_1(v_2^{m1}, v_2^{m5})$$
$$\theta_8 = \{v_1^{m5}/v_2^{m1}, z/k(v_2^{m5})\}$$
$$Node9 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5\theta_9 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), C_2(v_2^{m1}, v_2^{m5'})$$
$$\theta_9 = \{v_1^{m5'}/v_2^{m1}, z/k(v_2^{m5'})\}$$
$$Node10 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6\theta_{10} \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), C_1(v_2^{m1'}, v_2^{m5})$$
$$\theta_{10} = \{v_1^{m5}/v_2^{m1'}, z/k(v_2^{m5})\}$$
$$Node11 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6\theta_{11} \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), C_2(v_2^{m1'}, v_2^{m5'})$$
$$\theta_{11} = \{v_1^{m5'}/v_2^{m1'}, z/k(v_2^{m5'})\}$$
$$Node12 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7\theta_{12} \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), C_1(v_2^{m1''}, v_2^{m5})$$
$$\theta_{12} = \{v_1^{m5}/v_2^{m1''}, z/k(v_2^{m5})\}$$
$$Node13 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7\theta_{13} \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), C_2(v_2^{m1''}, v_2^{m5'})$$
$$\theta_{13} = \{v_1^{m5'}/v_2^{m1''}, z/k(v_2^{m5'})\}$$

Figure 4: SLD Tree

As in [21], we use functions with symbols from Λ in order to solve the so called impedance mismatch problem of constructing ontology objects from values occurring in the database. We assume that for $\lambda_1, \lambda_2 \in \Lambda$, where $\lambda_1 \neq \lambda_2$, the range of function with symbol λ_1 and the range of function with symbol λ_2 are disjoint. That is, the same ontology object cannot be produced from different functions.

2.1. Databases.

We start by giving definitions for database instances and queries over them, following the bag semantics from [5]. A bag B is a pair (US_B, μ), where US_B is a set called the underlying set of B and μ is a function from elements of US_B to the positive integers, which gives the multiplicities of elements of US_B in B. A relation instance is a bag of tuples of fixed arity using constants from DB_{Const}. A source schema S is a set of relation names from Σ_R. A database instance D for a source schema S is a mapping from relation names in S to relation instances.

2.2. Queries.

We define queries following the bag semantics of [5]. In our definitions we use the term "SQL query" although the syntax of our formulas is that of first-order logic. Similarly, relation instances are viewed as bags of ground atoms (i.e., with no variables) of first-order logic.

A SQL query over a relational schema S is an expression that has the form: $SQL(\vec{x}) \leftarrow \alpha$, where α is a first order expression containing predicates from Σ_R, which are among the relations that belong to S, $SQL \in \Sigma_R$, $SQL \notin S$ and $\vec{x}$ is a vector of constants from DB_{Const} and variables from Var that appear in α.

A conjunctive query Q over a relational schema S is a SQL query, where α has the form $R_1(\vec{x_1}) \wedge ... \wedge R_n(\vec{x_n})$, where $\vec{x_1}, ..., \vec{x_n}$ are vectors of constants from DB_{Const} and variables from Var, and $R_1, ..., R_n \in S$. Variables from $\vec{x_1}, ..., \vec{x_n}$ that do not appear in $\vec{x}$ are existentially quantified, but we omit the quantifiers in order to simplify the reading. CQs roughly correspond to SQL Select-From-Where queries.

An assignment mapping of a conjunctive query Q into a database instance D is an assignment of values from DB_{Const} belonging to D to the variables of Q such that every atom in the body of Q is mapped to a ground atom in D. Let θ be an assignment mapping of Q into database instance D and let X be a variable in Q. We denote by $\theta(X)$ the constant in DB_{Const} to which θ maps X and we denote by $\theta(R_i(\vec{x_i}))$ the ground atom to which $R_i(\vec{x_i})$ is mapped.

Let μ_i denote the multiplicities $\mu(\theta(R_i(\vec{x_i}))), i = 1, ..., n$. The result due to θ of a conjunctive query Q over D is the tuple $(\theta(\vec{x}), \mu_\theta)$ with the multiplicity $\mu_\theta = \mu_1 \mu_2 \cdots \mu_n$. The result of a conjunctive query Q over a database instance D denoted by $Q(D)$ is given by $\uplus_\theta r_\theta$, where θ is any assignment mapping of Q into D, r_θ is the result due to θ and $\uplus$ denotes bag union.

2.3. Ontology and Mappings.

A TBox is a finite set of ontology axioms. An ABox is a finite set of membership assertions $A(\rho)$ or role assertions $P(\rho, \rho')$, where $\rho, \rho' \in O_{Const}$ and $A, P \in \Sigma_O$ denote a concept name and role (or property) name respectively. A DL ontology O is a pair $\langle \mathcal{T}, \mathcal{A} \rangle$ where $\mathcal{T}$ is a TBox and $\mathcal{A}$ an ABox.

A mapping assertion (or simply a mapping) m from a source schema S to a TBox $\mathcal{T}$ has the form: $\phi(\vec{x}) \rightarrow \psi$, where $\phi(\vec{x})$ will be denoted by body(m) and it is the right-hand side of an SQL query over a database schema S, ψ has the form $P(f^1(\vec{x^1}), f^2(\vec{x^2}))$ or $C(f^1(\vec{x^1}))$ with P (respectively C) $\in \Sigma_O$ a property (respectively concept) name, all variables in ψ also appear in $\vec{x}$, and each $f^j \in \Lambda$ is a function with arity equal to the length of $\vec{x^j}$ and range a subset of O_{Const}. The right-hand side will be denoted by head(m). A mapping collection $\mathcal{M}$ is a finite set of such mapping assertions. In this setting, having a conjunction of atoms in the head of the mapping assertion does not add to the expressivity of the mapping language [21].

Let $\mathcal{M}$ be a mapping collection, we will use the symbol $\mathcal{M}_{CQ}$ to denote the assertions from $\mathcal{M}$ whose body is a CQ over the database schema. In correspondence with CQs over a relational schema, we define a CQ over an ontology O as an expression of the form: $Query(\vec{x}) \leftarrow P_1(\vec{x_1}) \wedge ... \wedge P_n(\vec{x_n})$ where $\vec{x_1}, ..., \vec{x_n}$ are vectors of constants from O_{Const} and variables from Var, $\vec{x}$ is a vector of constants from O_{Const} and variables from Var that appear in $\vec{x_1}, ..., \vec{x_n}$, and $P_1, ..., P_n \in \Sigma_O$ are ontology predicates that appear in O. A union of conjunctive queries UCQ over an ontology O is an expression of the form $Query(\vec{x}) \leftarrow CQ_1(\vec{x}) \vee ... \vee CQ_n(\vec{x})$, where each CQ_i for $i = 1, ..., n$ is an expression of the form $P_1^i(\vec{x_1^i}) \wedge ... \wedge P_n^i(\vec{x_n^i})$ as in the previous definition.

2.4. Logic Programs

Following [21], we use partial evaluation of logic programs in order to translate a UCQ over the vocabulary of the ontology into a UCQ over the data sources. In this section we present basic notions from logic programs[17] regarding partial evaluation [18]. As we are interested in the translation of UCQs, we do not deal with negation, and as a result we only present notions related to definite logic programs. As a result, in what follows we are referring to definite programs, clauses and rules.

A logic program is a set of statements that have the following form: $\forall \vec{x}(A \leftarrow A_1 \wedge ... \wedge A_n)$, where $A, A_1, ..., A_n$ are atoms as in standard first order logic definitions and $\vec{x}$ are all the variables occurring in $A, A_1, ..., A_n$. Each such statement is also called a program clause, or a rule, with A being the head of the rule, and $A_1 \wedge ... \wedge A_n$ the body of the rule. A goal is a clause such that the head is empty. Following the standard convention in logic programming, we omit the existential quantifiers and use the syntactic form $A_1, ..., A_n$ for the body, instead of $A_1 \wedge ... \wedge A_n$, both in clauses and goals.

A substitution θ is a finite set of the form: $\{x_1/t_1, .., x_n/t_n\}$, where each x_i is a variable, each t_i is a term distinct from x_i, variables $x_1, ..., x_n$ are pairwise distinct and no variable x_i occurs in some term t_i. Let Exp be an expression. The application of a substitution θ on Exp is denoted $Exp\theta$ and is the expression obtained by Exp after replacing each occurrence of x_i with t_i for $i = 1, ..., n$. Let Exp_1 and Exp_2 be expressions. A unifier for Exp_1 and Exp_2 is a substitution θ such that $Exp_1\theta = Exp_2\theta$. Let $\theta_1 = \{x_1/s_1, ..., x_m/s_m\}$ and $\theta_2 = \{y_1/t_1, ..., y_n/t_n\}$ be substitutions such that no variable from $x_1, ..., x_m$ occurs in θ_2. The composition of θ_1 with θ_2 is the following substitution: $\{x_1/s_1\theta_2, ..., x_m/s_m\theta_2, y_1/t_1, ..., y_n/t_n\}$. The most general unifier (mgu) of two expressions Exp_1 and Exp_2, is a unifier ξ such that for every unifier ν of Exp_1 and Exp_2 there exists a substitution θ such that ν is the composition of ξ with θ.

A computation rule is a function from a set of goals to a set of atoms, such that the value of the function for a goal is always an atom, called the selected atom, in that goal.

Let G be $\leftarrow A_1, ..., A_m, ..., A_k$, C be $A \leftarrow B_1, ..., B_q$ and R be a computation rule. Then, the goal G' is derived from G and C using the mgu θ via R if the following conditions hold:

- A_m is the selected atom in G given by R,
- θ is an mgu of A_m and A,
- G' is the goal $\leftarrow (A_1, ..., A_{m-1}, B_1, ..., B_q, A_{m+1}, ..., A_k)\theta$.

A resultant is a first order formula of the form $Q_1 \leftarrow Q_2$, where each of Q_1, Q_2 is either absent or a conjunction of atoms. Any variables in Q_1 or Q_2 are assumed to be universally quantified at the front of the resultant.

Let P be a program, G' be a goal with body G and R a computation rule. Then, the SLD-tree of $P \cup \{G'\}$ via R is the tree defined as follows:

- Each node is a resultant (possibly with an empty body)
- The root node is $G_0\{\} \leftarrow G_0$, where $G_0 = G$.
- Let $G\theta_0...\theta_i \leftarrow A_1, ..., A_m, ..., A_k$ be a node in the tree with $k \geq 1$ and suppose that A_m is the selected atom of the derivation given by R. Then, this node has a descendant for each input clause of $A \leftarrow B_1, ..., B_q$ of P such that A_m and A are unifiable. The descendant is $G\theta_0...\theta_{i+1} \leftarrow (A_1, ..., B_1, ..., B_q, ..., A_k)\theta_{i+1}$, where θ_{i+1} is an mgu of A and A_m.
- Nodes which are resultants with empty bodies have no descendants.

Each branch of the SLD-tree is a derivation of G'. A branch which ends in a node such that the selected atom does not unify with the head of any program clause is called a failure branch. A branch which ends in the empty clause is called a success branch. An SLD-tree is complete if all of its branches are either failure or success branches. An SLD-tree that is not complete is called partial.

In general an SLD-tree can contain branches that correspond to infinite derivations, but we will not deal with this case, as the logic programs that we will construct do not contain recursion.

The computed answer θ for a node $Q\theta_0, ..., \theta_i \leftarrow Q_i$ of an SLD-tree is the restriction of $Q\theta_0, ..., \theta_i$ to the variables in the goal G'.

Let P be a program, A an atom and R a computation rule and T an SLD-tree for $P \cup \{\leftarrow A\}$ via R. Then:

- any set of nodes such that each non-failing branch of T contains exactly one of them is a Partial Evaluation (PE) of A in P;

- the logic program obtained from P by replacing the set of clauses in P whose head contains A with a PE of A in P is a PE of P with respect to A.

The semantics of a logic program P can be defined by two different ways, proved to be equivalent. The first one is the declarative, that uses the model-theoretic semantics of first-order logic, where the semantics are given by the least Herbrand model, which contains the facts that are true in every model of P. The second way is the procedural, where the SLD-tree is used, and the semantics are given by the success set of P, that is all the facts A such that the SLD-tree of $P \cup \{\leftarrow A\}$ has a success branch. Also, it is known that the semantics of a program P coincide with the semantics of any partial evaluation of P[17].

3. Unfolding Queries Through Partial Evaluation

In this section we describe the process of unfolding queries over the ontology, into queries over the external relational database using mappings. We are following the approach of [21], with the following modifications:

- We enforce that during each step of the SLD-Derive process, the algorithm employs the computation rule that chooses for unification the leftmost possible atom in the right-hand side of the resultant.
- We make a distinction between mapping assertions whose body is a CQ over the database and the rest of the mapping assertions.
- We define a step that "folds" back specific branches of the PE tree based on the notion of combined mapping, and we show that the SQL query that is obtained based on this form of the PE tree has exactly the same answers with the SQL query obtained using the initial form of the tree.

The logic program for a UCQ $Q(\vec{x}) \leftarrow CQ_1(\vec{x}) \vee ... \vee CQ_n(\vec{x})$ over: (i) an ontology $\mathcal{O}$ (ii) a database instance D over a database schema S and (iii) a mapping collection $\mathcal{M}$ from source schema S to the vocabulary of $\mathcal{O}$ is defined in [21]. As it is shown that the result of the unfolding process is independent of the database instance D, here we omit the second component and we directly define the logic program with respect to $\mathcal{O}$ and $\mathcal{M}$. Also, we modify the process by using auxiliary predicates only for mapping assertions in $\mathcal{M} \setminus \mathcal{M}_{CQ}$.

The program for Q and $\mathcal{M}$, denoted $P(Q, \mathcal{M})$ is the logic program defined as follows:

- $P(Q, \mathcal{M})$ contains the clause $Q(\vec{x}) \leftarrow CQ_i(\vec{x})$ for each CQ_i in the right-hand side of Q.
- $P(Q, \mathcal{M})$ contains each mapping assertion $m \in \mathcal{M}_{CQ}$.
- For each mapping assertion $m \in \mathcal{M} \setminus \mathcal{M}_{CQ}$, $P(Q, \mathcal{M})$ contains the clause $head(m) \leftarrow Aux_m(\vec{x})$, where Aux_m is an auxiliary predicate associated to m, whose arity is the same as $head(m)$.

We now present the function SLD-Derive defined in [21], with the extra condition that we enforce use of the computation rule that chooses for unification the leftmost possible atom. The SLD-Derive($P(Q, \mathcal{M})$) takes as input $P(Q, \mathcal{M})$, where Q has the form $q(x) \leftarrow \beta$, and returns a set Res of resultants constituting a PE of $q(\vec{x})$ in $P(Q, \mathcal{M})$, by constructing an SLD-tree for $P(Q, \mathcal{M}) \cup \{\leftarrow q(\vec{x})\}$ as follows:

- it starts by selecting the atom $q(\vec{x})$,
- it continues by selecting the atoms whose predicates belong to the alphabet of $\mathcal{T}$, as long as possible, using the computation rule R which selects each time the leftmost such atom
- it stops the construction of a branch when no atom with predicate in the alphabet of $\mathcal{T}$ can be selected.

The partial evaluation $PE(Q, \mathcal{M})$ of $P(Q, \mathcal{M})$ with respect to $q(\vec{x})$ is obtained by dropping the clauses for q in $P(Q, \mathcal{M})$ and replacing them with the result of SLD-Derive($P(Q, \mathcal{M})$).

Example 2. Consider the query $ans(x, y, z) \leftarrow P_1(x, y), P_2(x, h(A)), P_3(y, z)$, with $h \in \Lambda$ and $A \in DB_{Const}$ the mapping collection ($\mathcal{T}$-mappings) shown in Figure 2 and a database instance over a schema that contains the relation names A_1, A_2, A_3, C_1 and C_2 with tuples of appropriate arities according to Figure 2. The SLD-tree for $P(Q, \mathcal{M}) \cup \{\leftarrow ans(x, y, z)\}$ is shown in Figure 4.

In [21] the virtual ABox given by a mapping collection $\mathcal{M}$ over a database instance D for a database

schema S is defined as the set of ABox assertions generated by applying each mapping assertion in $\mathcal{M}$ over D and it is shown that for each tuple of constants $\vec{t}$, $P(Q, \mathcal{M}) \cup \{\leftarrow q(\vec{t})\}$ is unsatisfiable if and only if $\vec{t}$ belongs to the result of executing Q over the database instance that stores exactly the assertions contained in the virtual ABox. Here we omit the formal definitions and the proof, but we note that it is straightforward to see that the specific result carries over to our modified definition of $P(Q, \mathcal{M})$. Also, the algorithm UnfoldDB is defined, which, given an UCQ Q over an ontology $\mathcal{O}$ with a mapping collection $\mathcal{M}$, translates the set of resultants returned by SLD-Derive($P(Q, \mathcal{M})$) into queries over the database instance D. Again, we omit the details and we note that in our case the resulted query will be a UCQ over S that has the form

$$Query(\vec{x}) \leftarrow Q_1(\vec{x}) \vee ... \vee Q_n(\vec{x}) \tag{1}$$

where each Q_i for $i = 1, ..., n$ is the translation given by UnfoldDB that corresponds to a resultant returned by SLD-Derive($P(Q, \mathcal{M})$), and it is an expression of the form

$$Q_i(\vec{f_i}(\vec{x_i})) \leftarrow$$
$$Aux_{i_1}(\vec{x_{i_1}}) \wedge ... \wedge Aux_{i_l}(\vec{x_{i_l}}) \wedge R_{i_{l+1}}(\vec{x_{i_{l+1}}}) \wedge ... \wedge R_{i_m}(\vec{x_{i_m}}) \tag{2}$$

where each $f_i^j \in \vec{f_i}$ is a function whose function name belongs in Λ and whose variable arguments are among the variables of $\vec{x_{i_1}}, ..., \vec{x_{i_m}}$, each Aux_{i_j} for $j = 1, ..., l$ corresponds to $body(m)$ for some $m \in \mathcal{M} \setminus \mathcal{M}_{CQ}$ and each R_{i_k} for $k = l + 1, ..., m$ is a relation name from the database schema. Note that on the original definition of UnfoldDB semantic query optimization (SQO) with respect to the database schema S is not performed. Nevertheless, in subsequent research, the role of SQO with respect to this context was proved crucial [27, 23]. In this work we consider that SQO, like self-join elimination, is performed in the result of UnfoldDB, that is in each Q_i for $i = 1, ..., n$ in (1). Furthermore, by overloading the definition of UnfoldDB, we consider a version of the function that takes as input an SLD-tree resulted from the application of the SLD-Derive($P(Q, \mathcal{M})$), and operates as described to produce a query that has the aforementioned form.

We now proceed with some definitions that will be used when we "fold back" the SLD-tree produced by SLD-Derive. For each edge e of the SLD-tree we define $source(e)$ to be the node at the beginning of e, $target(e)$ to be the node at the end of e, $TM(e)$ to be the predicate symbol of the atom selected by computation rule R at $source(e)$, $M(e)$ to be the clause (mapping assertion) used in the specific derivation, $sub(e)$ to be the substitution used in the specific derivation and $pos(e)$ to be the set of integers corresponding to the positions of atoms affected by the derivation in the right-hand side of the resultant in $target(e)$.

Let $m_1, ..., m_n$ be mapping assertions of the form $\phi_i \rightarrow \psi_i$ where no variable is repeated in ψ_i, for $i = 1, ..., n$. Also let θ be a unifier such that the atoms $\psi_1\theta, ..., \psi_n\theta$ are all equal (obviously the predicate symbol at the head of each assertion is the same). Then, the combined mapping of $m_1, ..., m_n$ is the following expression:

$\phi_1\theta \vee ... \vee \phi_n\theta \rightarrow \psi_1\theta$

If a variable z is repeated in some ψ_i, we modify ψ_i by keeping only the first occurrence and we replace all other occurrences with fresh variables $z_1, ..., z_k \in Var$. Then, we add the conditions $z = z_1, ..., z = z_n$ as conjuncts in the body of ψ_i.

Essentially, the combined mapping introduces a mapping assertion whose body is the union the input mappings, with the appropriate renaming. Two examples of combined mappings for the example mappings shown in Figure 2 are presented in Figures 5 and 6.

Proposition 1. Let T be an SLD-tree resulted from SLD-Derive with input $P(Q, \mathcal{M})$, m_c be the combined mapping of mappings $m_1, ..., m_n \in \mathcal{M}$ and $\mathcal{M}_c = (\mathcal{M} \setminus \{m_1, ..., m_n\}) \cup \{m_c\}$. The semantics of $PE(Q, \mathcal{M})$ coincide with the semantics of $PE(Q, \mathcal{M}_c)$.

Proof. We need to show that for every tuple $\vec{t}$ of constants, $q(\vec{t})$ is true in $PE(Q, \mathcal{M})$ if and only if $q(\vec{t})$ is true in $PE(Q, \mathcal{M}_c)$, which follows directly from the construction of m_c. □

Let T be the tree resulted from SLD-Derive($P(Q, \mathcal{M})$) and e_0 an edge in T. Also, let $e_1, ..., e_n$ be edges in T with $source(e_0) = source(e_1) = ... = source(e_n)$ and $TM(e_0) = TM(e_1) = ... = TM(e_n)$ such that there exists a combined mapping $m_c : \phi_0\theta \vee ... \vee \phi_n\theta \rightarrow \psi_1\theta$, with $\theta = sub(e_0)$ and $TM(e_0)$ be equal to the predicate at the head of m_c. A fold of T into e_0 is the tree T_1 that is resulted from T by replacing in each descendant node of $target(e_0)$ (including $target(e_0)$) the atoms at positions $pos(e_0)$ with the

$$cm_1 : A_1(v_1^{m1}, v_2^{m1}) \vee A_2(v_1^{m1}, v_2^{m1}) \vee A_3(v_1^{m1}, v_2^{m1}, v_3^{m1''}) \rightarrow P_1(f(v_1^{m1}), g(v_2^{m1}))$$
$$\theta = \{v_1^{m1'}/v_1^{m1}, v_1^{m1''}/v_1^{m1}, v_2^{m1'}/v_2^{m1}, v_2^{m1''}/v_2^{m1}\}$$

Figure 5: Combined Mapping for Mapping Assertions $m1$, $m1'$ and $m1''$

$$cm_2 : C_1(v_1^{m5}, v_2^{m5}) \vee C_2(v_1^{m5}, v_2^{m5}) \rightarrow P_3(g(v_1^{m5}), k(v_2^{m5}))$$
$$\theta = \{v_1^{m5'}/v_1^{m5}, v_2^{m5'}/v_2^{m5}\}$$

Figure 6: Combined Mapping for Mapping Assertions $m5$ and $m5'$

atom $\psi_1\theta$, and deleting all the sub-trees starting from $target(e_1), ..., target(e_n)$. Moreover, let $f_1, ..., f_m$ be all the edges in T (including e_0) such that $M(f_1) = ... = M(f_m) = M(e_0)$. Then, the fold of T based on m_c is the tree that is obtained if we sequentially apply the process of obtaining the fold of T into f_i for $i = 1, ..., m$ ensuring that for each f_k, f_l with k, l in $1, ...m$, if the depth of $target(f_k)$ in T is smaller than the depth of $target(f_l)$, then the fold of T into f_k is obtained after obtaining the fold of T into f_l.

Figure 7 shows the fold of the SLD-tree of example 2 based on the combined mapping from Figure 6, where Aux_{cm_2} is an auxiliary predicate used for mappings in $\mathcal{M} \setminus \mathcal{M}_{CQ}$ according to the construction of $P(Q, \mathcal{M})$, that correspond to combined mapping cm_2. Note that the same combined mapping is recognized and used in three different nodes of the initial tree.

Proposition 2. Let T be an SLD-tree resulted from SLD-Derive with input $P(Q, \mathcal{M})$, m_c be the combined mapping of mappings $m_1, ..., m_n \in \mathcal{M}$ and $\mathcal{M}_c = (\mathcal{M} \setminus \{m_1, ..., m_n\}) \cup \{m_c\}$. The fold of T based on m_c is exactly the tree returned by SLD-Derive with input $P(Q, \mathcal{M}_c)$.

Proof. Let T_{fold} be the fold of T based on m_c and T_c be the SLD-tree resulted from SLD-Derive with input $P(Q, \mathcal{M}_c)$. We need to show that T_{fold} and T_c consist of the same resultants. Clearly the two trees have the same root. Then, given a resultant $ans(\vec{x})\theta_0...\theta_k \leftarrow A_1(\vec{x_1}), ..., A_i(\vec{x_i}), ..., A_w(\vec{x_w})$ at depth k which is the same for the two trees, it is sufficient to show that the children of this resultant are the also the same for the two trees. Let $node_{T_c}$ and $node_{T_{fold}}$ be the nodes in T_c and T_{fold} respectively that contain the specific resultant. Suppose that A_i is the leftmost atom in the body of the resultant with predicate that belongs to the alphabet of $\mathcal{T}$ and will be chosen by the computation rule R. Also, for now, let us suppose that there is only one node $node_T$ in the initial tree T such that for every edge e in T with $TM(e)$ equal to the predicate symbol at the head of m_c, then $source(e) = node_T$. According to the construction of the fold of T into e_0, if $node_T$ is different from $node_{T_{fold}}$, then the children of $node_{T_{fold}}$ are the same with the children of $node_{T_c}$, as they are not affected by the combined mapping. If $node_T$ is equal to $node_{T_{fold}}$, then $node_T$ has n children affected by the combined mapping, plus a number of children not affected (possibly 0). The second kind of children are also children of $node_{T_c}$, whereas the first kind have been replaced in T_{fold} with the child $ans(\vec{x})\theta_0...\theta_k\theta_{k+1} \leftarrow A_1(\vec{x_1}), ..., Aux_c m(\vec{z}), ..., A_w(\vec{x_w})$, which is also a child of $node_{T_c}$, and these are the only children of both $node_{T_c}$ and $node_{T_{fold}}$. Now, if there are more nodes in T affected by the fold of T based on m_c, then from the construction of the fold, where descendant nodes are always modified prior to their predecessors, and from the fact that R chooses always the leftmost possible atom, it is straightforward to see that the result of the case where only one node is affected by the combined mapping is carried over to this case.

□

A direct consequence of Propositions 1 and 2 is that if we consider the SLD-tree tree T resulted from SLD-Derive with input $P(Q, \mathcal{M})$ and we apply the UnfoldDB algorithm on the resultants contained in the the fold of T based on the combined mapping m_c, then the SQL query that will be produced has exactly the same answers with the SQL

query produced by applying the UnfoldDB algorithm on the resultants of the original tree T. This gives us the ability to choose a sequence of folds, in order to obtain an equivalent translation that can be more efficient, by using a cost-based search in the initial SLD-tree, which we describe in detail in the Section 4.2.

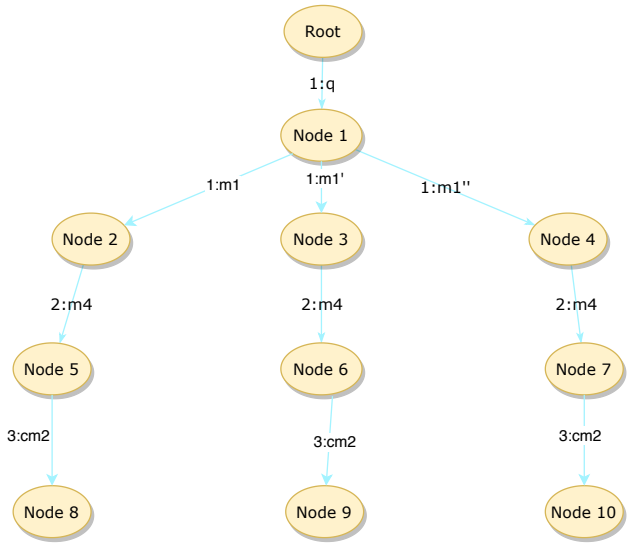
An issue that arises in these translations has to do with the way the combined mapping is treated. One way is to be treated as a regular mapping assertion, as the body of this mapping is simply a union query over the original mapping assertions. This union query will be computed as many times as the combined mapping is used in the produced query. Obviously a better choice would be to create a temporary table that holds the specific result as an intermediate result of the main query, in the same database connection. This is the solution that we follow in this work, as it also avoids the overheads of creating permanent materialized views in the database as in [26]. A second issue that has to be handled is the decision regarding which folds should be used, if any, for a specific query. As we will describe in the following section, the process of taking the specific decision heavily depends on the size of the SQL query, the size of each combined mapping in comparison to the size of the final SQL query and the number of duplicate answers contained in them.

4. Cost-Based Selection of Query Translation

In this section we consider a cost-based algorithm in order to choose a specific sequence of folds and obtain the SQL translation of the initial query. During this process we take into consideration the two kinds of redundant processing that we described in Section 1. Regarding the first kind (redundancy due to duplicates), we will employ a heuristic about early duplicate elimination of intermediate results during query evaluation that we first described in [1]. In order to describe the heuristic, we first consider a single subquery that has the form shown in formula (2) of Section 3. After that, in Section 4.2 we describe our algorithm operating on the complete query that has the form shown in formula (1). Our method relies on an estimation of the final result size of each union subquery. To obtain this estimation we should gather some statistics from the database in the form of data summarization for all the columns that can be possibly referenced from a query, that is all the columns in the SQL queries of some mapping assertion. As making an estimation for an arbitrary FOL query is an involved process, we make a distinction between assertions in $\mathcal{M}_{CQ}$ ($R_{i+l+1}, ..., R_{i+m}$ in formula 2) and assertions in $\mathcal{M} \setminus \mathcal{M}_{CQ}$ ($Aux_{i_1}, ..., Aux_{i_n}$ in formula 2). We consider that the latter are primitive tables as if they were virtual views, and we collect statistics only for the output columns, whereas the former are parsed and we collect statistics for all the referenced columns. We will refer to each conjunct in the right-hand side of (2) as an input table of query $Q_i(\vec{f}_i(\vec{x}_i))$.

Let q be a query as in (2) and $I_i(\vec{x}_i)$ be an input table of q. The query $ans(\vec{x_{c_i}}) \leftarrow I_i(\vec{x}_i)$, where $\vec{x_{c_i}}$ contains exactly the variables of $\vec{x}_i$ that appear at least two times in q, will be called the projection query of input table $I_i(\vec{x}_i)$ from q. Additionally, let D be a database instance (which will be implied). Intuitively the projection query selects all the columns of an input table that are mentioned elsewhere in q. In Section 4.1, we decide if we will save each projection query as an intermediate result with respect to duplicates.

Analyzing External Tables. As we operate outside the RDBMS engine, in order to extract the needed information we should import all the corresponding data, which is clearly not practical. Luckily we have several other options. One such option is to only import a random sample and extract the needed information from that, as most database vendors support ordering the results by a random function. Another option is to obtain the data summarization directly from the RDBMS, if it provides a way to access this information. This option is likely to give the most accurate results, but it is highly dependant on the specificities of each database vendor. One third option is to build a simple single-bucket histogram for each column, by sending for execution queries that ask for the number of values, number of distinct values, minimum and maximum value. Simple histograms like this are known to give imprecise selectivity estimations for filter and join results of attributes that exhibit skewness [10], but on the other hand their construction and usage is faster in comparison to more elaborate kinds of histograms. For our experiments we have chosen the last option, as it is fast and simple and can be applied to any underlying RDBMS. This is a one-time offline process that needs to be done before query execution, similar to an analyze command in a database schema, as it only depends on the re-



$$Root : ans(x,y,z)\theta_0 \leftarrow ans(x,y,z)$$
$$\theta_0 = \{\}$$
$$Node1 : ans(x,y,z)\theta_0\theta_1 \leftarrow P_1(x,y), P_2(x,h(A)), P_3(y,z)$$
$$\theta_1 = \{\}$$
$$Node2 : ans(x,y,z)\theta_0\theta_1\theta_2 \leftarrow A_1(v_1^{m1}, v_2^{m1}), P_2(f(v_1^{m1}), h(A)), P_3(g(v_2^{m1}), z)$$
$$\theta_2 = \{x/f(v_1^{m1}), y/g(v_2^{m1})\}$$
$$Node3 : ans(x,y,z)\theta_0\theta_1\theta_3 \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), P_2(f(v_1^{m1'}), h(A)), P_3(g(v_2^{m1'}), z)$$
$$\theta_3 = \{x/f(v_1^{m1'}), y/g(v_2^{m1'})\}$$
$$Node4 : ans(x,y,z)\theta_0\theta_1\theta_4 \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), P_2(f(v_1^{m1''}), h(A)), P_3(g(v_2^{m1''}), z)$$
$$\theta_4 = \{x/f(v_1^{m1''}), y/g(v_2^{m1''})\}$$
$$Node5 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), P_3(g(v_2^{m1}), z)$$
$$\theta_5 = \{v_1^{m_4}/v_1^{m1}, v_3^{m_4}/A\}$$
$$Node6 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6 \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), P_3(g(v_2^{m1'}), z)$$
$$\theta_6 = \{v_1^{m_4}/v_1^{m1'}, v_3^{m_4}/A\}$$
$$Node7 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7 \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), P_3(g(v_2^{m1''}), z)$$
$$\theta_7 = \{v_1^{m_4}/v_1^{m1''}, v_3^{m_4}/A\}$$
$$Node8 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5\theta_8 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), Aux_{cm_2}(v_2^{m1}, v_2^{m5})$$
$$\theta_8 = \{v_1^{m5}/v_2^{m1}, z/k(v_2^{m5})\}$$
$$Node9 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6\theta_{10} \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), Aux_{cm_2}(v_2^{m1'}, v_2^{m5})$$
$$\theta_{10} = \{v_1^{m5}/v_2^{m1'}, z/k(v_2^{m5})\}$$
$$Node10 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7\theta_{12} \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), Aux_{cm_2}(v_2^{m1''}, v_2^{m5})$$
$$\theta_{12} = \{v_1^{m5}/v_2^{m1''}, z/k(v_2^{m5})\}$$

Figure 7: SLD Tree 2

lations occurring in mappings and data. Also, as it is crucial to have an accurate estimation of the number of duplicate answers that come from different mappings for the same predicate, we execute queries counting exactly the distinct number of answers for queries in bodies of mappings that can possibly formulate a combined mapping assertion. These mapping assertions can simply be identified offline as the subsets of mappings whose heads can be unified during the partial evaluation. Regarding duplicates coming from a single mapping, adopting the commonly used value independence assumption between the result attributes and the uniformity of values in an attribute [28], we estimate the distinct tuples of the relation to be the product of the distinct values of its attributes. In case this value is larger than the number of tuples in the relation, we assume that there are no duplicate tuples in the relation.

4.1. Early Duplicate Elimination of Intermediate Results

First, we define the duplicate-tuple ratio DTR_R of a relation instance R to be equal to $\frac{\sum_{t \in US_R} \mu(t)}{|US_R|}$. A relation instance with DTR equal to 1 will be called a duplicate-free relation instance. Now, let us suppose that we have a single SQL subquery coming from the unfolding step and we have to take the decision regarding a single input table (either "real" primitive table or virtual view) used in this subquery; we will take into consideration different union subqueries in Section 4.2. In this case, it may be advantageous to dictate the RDBMS to perform the duplicate elimination on projection query of the specific input table at the beginning of query execution, store the duplicate-free intermediate result in a temporary table and use it for the specific query. This can be done in several ways depending on the exact SQL dialect and capabilities of the underlying system. For example, one can use (non-recursive) common table expressions or temporary table definitions. Of course, the exact decisions as to when this should happen depend on several factors, including the exact query, the DTR of the projection query of the input table, the number of uses of the specific input table in the query, the choice to save the temporary table in disk or keep it in memory and several other factors that depend on the database physical design, database tuning parameters, the exact query execution plan and the evaluation methods chosen by the optimizer of the RDBMS. As mentioned, it is difficult for all these factors to be estimated outside the database engine. For this reason, in what follows, we propose to take this decision according to a heuristic that depends only on the size of the data and the DTR of the input table, whose estimation can be obtained using data summarization.

The main assumption that we make regarding duplicate elimination, states that the impact of an input table with DTR equal to a constant number n in the number of tuples of the final query result is proportional to n. As a result of this assumption, the selectivity of the query plays the most important role in duplicate elimination decisions. Intuitively, a query whose result size is much larger than the size of the intermediate result for which we examine the duplicate elimination option, it is expected to be faster to first perform the elimination, as each tuple of the intermediate result has as impact the creation of a large number of tuples in the final result. On the other hand, a query with few results is expected to be evaluated faster if duplicates are eliminated directly from the final result. In this case one would expect that each tuple of the intermediate result does not add that much to the total cost of the query in order to counterbalance the cost of a duplicate elimination, especially when expecting the optimizer to limit the sizes of intermediate query results as soon as possible.

A Heuristic Regarding Duplicate Elimination. Given a database instance D, a query q of the form (2) whose result over D is the relation instance Q and an input table $I_i(\vec{x_i})$ of q, perform duplicate elimination on input table $I_i(\vec{x_i})$ prior to execution of q if

$$Size_Q - \frac{Size_Q}{DTR_{Ans}} > \frac{Size_{Ans}}{DTR_{Ans}}$$

where relation instance Ans is the result of the projection query of $I_i(\vec{x_i})$ from q on D and $Size_Q$ and $Size_{Ans}$ are the estimated sizes (in bytes) of relation instances Q and Ans respectively. That is, duplicate elimination should be performed if it is expected that the reduction on the size of the final result will be bigger than the size of the intermediate result with duplicate elimination.

4.2. Cost-based Translation

In this section we present the algorithm *GetTranslation* (Algorithm 1), which, given a UCQ Q over an ontology $\mathcal{O}$ and a mapping collection $\mathcal{M}$

from O to a database instance D over a database schema S, it returns a SQL query over D and provides a set $CM_{temporary}$ of temporary views to be created. Each of these temporary views corresponds to a SQL query on the body of a combined mapping that exists in the SDL-tree produced by SLD-Derive($P(Q, \mathcal{M})$). In other words, the algorithm chooses a sequence of folds based on one of these combined mappings each time, that are performed repeatedly in a corresponding sequence of trees, starting from the initial SLD-tree. The $T_{current}$ variable holds the current tree at each point of execution. In each step, the fold that is expected to provide the largest gain is chosen, and this process is continued until no fold that provides gain exists. In this sense, the algorithm proceeds in a greedy way, in order to avoid examining all the combinations. The gain for each possible combined mapping is estimated based in the redundant processing that we avoid by materializing and using the specific mapping with respect to i) duplicate answers and ii) repeated operations even in the absence of duplicate answers.

Regarding duplicate answers, in correspondence with the observations made in Section 4.1, here the main factors that determine the behavior of the algorithm are the query selectivity and the size of the result of the SQL query in the body of each combined mapping. The difference here is that we consider the final query that is the result of UnfoldDB, instead of a single union subquery, and a combined mapping that contains many input mappings which can produce duplicate results between them, instead of a single input table of one subquery. Let cm be the combined mapping $\phi_1 \vee ... \vee \phi_n \rightarrow \psi$ in this context, for simplicity we will denote by $Size_{cm}$ and DTR_{cm} the size and DTR of the relation instance that is the result of executing the query $\phi_1 \vee ... \vee \phi_n$ over the database instance D, given that duplicate elimination is not performed. Computing and saving the combined mapping is expected to be more efficient, if the reduction on the size of the final SQL query will be bigger than the size of the temporary table resulting from the materialization of the combined mapping with duplicate elimination ($Size_{cm}/DTR_{cm}$). Using the quantity $Size_{SQL_{cm}}$ to denote the size of the result of the final SQL query when the combined mapping cm has been chosen for materialization with the duplicates eliminated, which is equal to $Size_{SQL_{current}}/DTR_{cm}$, we have that the result of UnfoldDB with input the fold of T into cm (SQL_{cm}) is preferred over the result of UnfoldDB with input T ($SQL_{current}$) if:

$$Size_{SQL_{current}} - Size_{SQL_{cm}} > \frac{Size_{cm}}{DTR_{cm}} \tag{3}$$

Regarding repeated operations even in the absence of duplicate answers, as discussed in Section 1, in order to obtain an exact cost model we should be aware of the exact execution plan and the choice of access methods for each relation in order to estimate the amount of data read and written to disk for each CQ. As this is not viable for the OBDA system that operates outside the database engine, we base our estimation on the sizes of the input relations and the size of the result. Specifically, we consider that the smaller table in each CQ is fully scanned once, and all other tables are either probed using an index as many times as the number of final query results or are fully scanned once, depending on which of the two options has the lowest cost. In order to find the smaller table, table sizes in this context are compared by taking into consideration the filters that appear in each table in the CQ, that is tables are compared according to the size of each corresponding projection query. Also, as we do not want to take into consideration duplicates introduced from the combined mapping under consideration, for each input table that participates in the combined mapping, we take its size after we divide it by DTR_{cm}.

Let SQL be an SQL query of the form 1 that is the result of UnfoldDB, we will denote by RR_{SQL} the estimation for the size in bytes of redundant reads in the absence of duplicates as described. In other words, RR_{SQL} holds the sum of redundant reads for every disjunct (CQ) in the right-hand side of (1). Then, the result of UnfoldDB with input the fold of T into cm (SQL_{cm}) is preferred over the result of UnfoldDB with input T ($SQL_{current}$) if the estimated reduction in redundant reads from $SQL_{current}$ to SQL_{cm} is larger than the size of the temporary table resulting from the materialization of the combined mapping with duplicate elimination ($Size_{cm}/DTR_{cm}$):

$$RR_{SQL_{current}} - RR_{SQL_{cm}} > \frac{Size_{cm}}{DTR_{cm}} \tag{4}$$

If we want to take both kinds of redundant processing into consideration concurrently, we simply have to add the left-hand side parts of (3) and (4):

$$Size_{SQL_{current}} - Size_{SQL_{cm}} + RR_{SQL_{current}} - RR_{SQL_{cm}} > \frac{Size_{cm}}{DTR_{cm}} \quad (5)$$

In Algorithm 1 we are considering the heuristic as a quantity giving the expected gain, with negative values meaning that we have loss instead of gain, as shown in Line 11 of the algorithm, since we want to compare the different options and choose the one that gives the biggest gain at each step. So the final formula used is:

$$Size_{SQL_{current}} - Size_{SQL_{cm}} + RR_{SQL_{current}} - RR_{SQL_{cm}} - \frac{Size_{cm}}{DTR_{cm}} \quad (6)$$

Regarding some implementation issues, we should note that we do not need to make selectivity estimation for all the results each time, but only for those that are affected by the combined mapping, that is, the disjuncts in the result of UnfoldDB that correspond to resultants in the SLD-tree which are descendants of nodes which use some of the input mappings of the combined mapping examined each time. As a matter of fact, we can modify the gain formula so that only these disjuncts are taken into consideration in the computation of $RR_{SQL_{current}}, RR_{SQL_{cm}}, SQL_{current}$ and SQL_{cm}.

5. Implementation and Experimental Evaluation

We have implemented our translation in an prototype extension of Ontop version 1.18.1. This version of Ontop normally uses the default unfolding method of [21] over the $\mathcal{T}$-Mappings in order to emulate H-complete ABoxes [22], as we mentioned in Section 1, and employs the tree-witness query rewriting [14] on such ABoxes. We follow the same architecture, using the tree-witness approach for query rewriting and we modify the unfolding step over the $\mathcal{T}$-Mappings as described here.

Newer versions of Ontop use a different query unfolding method that employs the notion of intermediate query (IQ) [30]. We discuss the relevance of our method to this in Section 6. For this reason, we compare our method with both the default translation based in partial evaluation of logic progrmams obtained by version 1.18.1, but also with the new

Algorithm 1: Translation Process

```
1  GetTranslation (M, Q, D);
   Input  : Mapping Collection M, Query Q,
            Database D
   Output: SQL query over D
2  CM_temporary = ∅;
   // The combined mappings that should be
      used as temporary tables
3  T_current = SLD-Derive(P(Q, M)); // The
      SLD-tree at each step. Initially equal to
      the result of SLD-Derive(P(Q, M))
4  SQL_current = UnfoldDB(T_current);
5  Add to CM_used all the combined mappings
     that exist in T_current;
6  MaxGain = 0;
7  do
8      foreach cm ∈ CM_used do
9          T_cm: the fold of T_current based on cm;
10         SQL_cm = UnfoldDB(T_cm);
11         Compute Gain from SQL_current to
             SQL_cm according to Formula 6 ;
12         if Gain > MaxGain then
13             MaxGain = Gain;
14             T_best = T_cm;
15             SQL_best = SQL_cm;
16             BestCm = cm;
17         end
18     end
19     if MaxGain > 0 then
20         SQL_current = SQL_best;
21         T_current = T_best;
22         Remove BestCm from CM_used;
23         Add BestCm to CM_temporary;
24     end
25 while MaxGain > 0;
26 return SQL_current;
```

translation method obtained from the latest Ontop versions 3.0.1 and 4.0.2. In general, version 3.0.1 outperforms version 4.0.2, so we only report times for version 3.0.1 here, but all the execution times for version 4.0.2 are also available along with all other material[2].

Our aim in this section is to perform an experimental comparison of our approach with other methods using well-known benchmarks. For this reason, we present experiments using the NPD and LUBM benchmarks in Section 5.1, comparing our approach with the translation performed by the two aforementioned Ontop versions. Then, in Section 5.2, we compare our approach with the JUCQ approach using the datasets and queries from [16] and, in Section 5.3, we study the performance of our method in comparison to the default translation, for different query characteristics. Finally, in order to obtain an empirical analysis of our heuristic regarding duplicate elimination, in Section 5.4 we perform an experimental evaluation using a micro benchmark with specific query fragments coming from queries used in the general evaluation.

5.1. Experiments with NPD and LUBM Benchmarks

We have performed an experimental evaluation of our techniques using the LUBM [9] and NPD [15] benchmarks, with the ontology and mappings that are publicly available at the Ontop repository on github[3] and with existential reasoning enabled. Both datasets were generated for scale 100.

The experiments in this section were carried out on a machine with an Intel Core i7-3770K processor with 8 cores and 32 GB of RAM running UBUNTU 18.04, using PostgreSQL version 11.3 as a backend. PostgreSQL was setup and tuned for usage in a machine with 32GB RAM. The schema and data in all systems were identical and all the proposed indexes were created. The database size was about 1.1 GB for LUBM and about 5.8 GB for NPD.

Queries and Mappings. For LUBM benchmark in total 84 mapping assertions were produced as $\mathcal{T}$-Mappings from Ontop. For LUBM we used the original 14 queries. For NPD we used a subset of 19 out of the original 30 queries: queries 1-12, 22-25 and 28-30, excluding the queries that use GROUP BY, as it is not supported by the used Ontop version, queries that contain OPTIONAL and queries with empty translation due to incompatible IRIs. To these queries we added four more, in order to showcase the advantage of duplicate elimination coming from a single mapping. The reason for this addition is that despite the fact that many mappings introduce duplicates, the existing queries are only using a small subset of the mappings that mostly avoid this problem. We believe that the four added queries are sensible and simple, yet their evaluation proved very hard. This showcases that the problem we are dealing with is also present in the NPD benchmark. These new queries are numbered 31 to 34 and presented in Appendix A. All SPARQL queries were executed using the DISTINCT modifier.

Overhead in Setup and Optimization. The time needed to gather all the necessary statistics and analyze tables prior to the first deployment of the system as described in Section 4 was 48 seconds for LUBM and 3 minutes and 10 seconds for NPD. Total optimization time for the 14 LUBM queries total time increased from 325 ms to 360 ms, whereas for the 23 NPD queries the increase was from 1115 ms to 1380 ms. The given times include the total time from parsing each SPARQL query to outputting the corresponding SQL query. The first time is the time needed by the original Ontop version 1.18.1, whereas the second time is the time needed by our modified version.

Results. For each query we used a timeout of 1000 seconds. For each setting, all queries were executed sequentially according to their numbering, after a full system reboot. The given times measure the total time needed for each query including the optimization time in Ontop, the execution time in the relational back-end and the time to obtain the results in Ontop. All the results were obtained, but they were not saved or processed otherwise. The combined mappings chosen by our method were materialized as temporary tables during execution in the same session as the main query and unique indexes were created on those tables. All times are in milliseconds. All results and the produced SQL queries, as well as all the necessary material to reproduce the experiments are available in the link given in the beginning of this section.

[2] http://cgi.di.uoa.gr/~dbilid/experiments-obda/

[3] https://github.com/ontop/iswc2014-benchmark/tree/master/LUBM and https://github.com/ontop/npd-benchmark

Query	v1 Default	v1 Opt.	v3	#Results
NPD 1	4899	5258	13696	1627744
NPD 2	4189	4142	5015	172751
NPD 3	1155	1119	1535	83737
NPD 4	20542	20899	27159	1627744
NPD 5	54	66	128	193
NPD 6	33234	23533	36128	1231564
NPD 7	1438	1377	1489	180
NPD 8	307	303	ERROR[1]	5974
NPD 9	2354	2222	1537	12750
NPD 10	4243	3649	3800	79512
NPD 11	86773	7650	8523	418056
NPD 12	122712	14376	16824	838430
NPD 22	6373	3247	8003	1113200
NPD 23	6565	3304	44340	763400
NPD 24	2437	498	ERROR[1]	147400
NPD 25	10055	9324	12106	1725400
NPD 28	32343	22815	167362	2141968
NPD 29	90271	17212	26400	419834
NPD 30	163276	26661	58143	705984
NPD 31	TIMEOUT	29771	54641	2979400
NPD 32	1085	318	746	8000
NPD 33	77139	19545	24509	148037
NPD 34	5443	3329	18678	486000
Avg.	30768[2]	9592	25274[2]	

[1] Error during unfolding
[2] Excluding timeouts and errors

Table 1: Results for NPD scale 100 (Times in ms)

Query	v1 Default	v1 Opt.	v3	#Results
LUBM 01	543	587	685	4
LUBM 02	1283	1272	1377	264
LUBM 03	129	87	101	6
LUBM 04	149	125	438	34
LUBM 05	69	98	71	719
LUBM 06	17086	8868	29419	1048532
LUBM 07	259	306	334	67
LUBM 08	393	301	1079	7790
LUBM 09	47126	33518	16539	27247
LUBM 10	16	16	13	4
LUBM 11	191	187	192	224
LUBM 12	132	134	245	15
LUBM 13	112	111	138	472
LUBM 14	3096	2826	4406	795970
Avg.	5042	3460	3931	

Table 2: Results for LUBM scale 100 (Times in ms)

Results are presented in Table 1 for NPD queries and in Table 2 for LUBM queries. Results in column v1 Default contains the execution times obtained by the Ontop version 1.18.1, column v1 Opt. contains the times obtained by the modified Ontop version according to our approach and column v3 contains the times obtained by Ontop version 3, the latest stable Ontop release. The average execution times for each case are also shown in the bottom of each table, excluding errors and timeouts. For the case of NPD queries, there was 1 timeout from v1 Default for query 31, and two errors during unfolding from Ontop v3. The exact error message for each error can be found at our result repository. According to the results, our approach outperforms on average both Ontop version 1.18.1 and version 3. For the NPD benchmark the decrease in average execution time obtained by our method is 69% and 62% in comparison to version 1.18.1 and version 3 respectively, while for the LUBM benchmark the decrease is 31% and 12% respectively. Also, with very few exceptions, our method outperforms the other two approaches on every single query.

5.2. Comparison with the JUCQ Approach

In this section we compare our method with the approach from [16]. As this implementation is not part of the Ontop release, we directly use the queries produced by this approach, which are available at the Ontop examples github repository [4]. For this reason, in all the experiments presented in this section we only report the time for executing the SQL queries in PostgreSQL, omitting the time for query unfolding. For measuring the execution times of the JUCQ approach, we used the scripts provided in the aforementioned github repository. As in the previous section, we also include the times obtained using the versions 1.18.1 and 3 of Ontop. The execution environment is the same as in the previous section.

We use the exact benchmark and queries that were also used in [16]. Specifically, we use the OBDA version of the Wisconsin benchmark [7], with the same ontology and mappings, for which we have created 24 instances of the base relational table, each one with 1 million tuples. This is the exact setting used in [16]. The results of the Wisconsin benchmark are presented in Table 3, where

[4] https://github.com/ontop/ontop-examples/tree/master/iswc-2017-cost/

there are two different query sets, one that contains queries consisting of 3 atoms, and the other with queries consisting of 4 atoms. Each query set contains 84 queries, and the average execution time for each approach is shown. Our approach outperforms all other translations, followed by the JUCQ approach, whereas the worst performance is obtained from the default translation of version 1, which is the only approach such that timeouts occur. One other observation has to do with the execution times for the UCQ (default translation of Ontop 1.18.1) and JUCQ translations reported in [16]. Specifically, our execution times for these two sets of approaches seem to be much better. For example, in their reported times, timeouts of 20 minutes occurred in every setting, and the average execution time for the JUCQ approach was 160 seconds for the 3 atoms query set, whereas in our experiments the corresponding time is only 30.5 seconds. These differences can possibly be attributed to different versions of the PostgreSQL database (they used version 9.6) and different tuning parameters of the database engine. Other than that, our findings are consistent with theirs. Specifically, we observed the largest improvement of JUCQ with respect to the default UCQ translation for queries with more mappings and redundancy. The behavior of our approach is similar, exhibiting large improvement for these queries in comparison to all other three approaches.

Finally, we use the same modified NPD queries NPD 6*, NPD 11*, NPD 12* and NPD 31* as in [16], executed over the scale 100 of the NPD benchmark. This is different from [16], where these queries were executed only over the original NPD dataset (scale 1). The results are presented in Table 4. Again our approach outperforms all other approaches. Also, regarding the JUCQ translation, the results here show a different situation in comparison to the Wisconsin benchmark, as it exhibits the worst performance and also a timeout occurs for query NPD 31*. The queries produced by the JUCQ approach seem in general more complicated from the ones produced from the other three approaches.

5.3. Performance gain

In this section, we study the performance gain of our optimized method over the default translation which is obtained by partial evaluation of logic programs and leads to generation of UCQs. Following the setup of [16], we use the Wisconsin benchmark,

Query Set	v1 Default	v1 Opt.	v3	JUCQ
3 atoms	73133	22589	53624	30528
4 atoms	223684[1]	30922	64048	43926

[1] Excluding 24 timeouts

Table 3: Average execution time (ms) for Wisconsin Benchmark (24 tables with 1 million tuples per table)

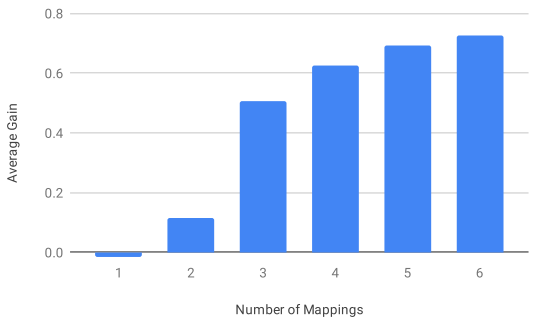


Figure 8: Performance gain for varying number of mappings per predicate

generating 24 tables with 1 million tuples per table, executing 84 queries with 3 atoms each, with a varying number of mappings used for each query (from 1 to 6) and we compute the performance gain using the formula 1−(Opt. Time/Default Time). In Figure 8 we present results for each number of mappings per predicate. The figure presents the average gain for all the queries per case (1 to 6 mappings). As expected, when there is only 1 mapping per predicate, our method does not generate any temporary table, and as a result, it performs roughly the same as the default translation. Starting from two predicates, our methods begins to outperform the default translation, reaching an average gain of more than 0.7.

In Figure 9, we present a scatter chart with the performance gain with respect to the number of results for the 84 queries of the benchmark. As shown, the queries are partitioned in visually distinct groups with respect to the number of their results. The effect of query selectivity is evident in this chart, with our method becoming increasingly efficient compared to the default, as the number of results grows larger, achieving a gain of 0.75 for the queries with about 1.7 million results. On the contrary, for the first group of queries, with low number of results, we cannot see a consistent behavior in comparison with the default.

Query	v1 Default	v1 Opt.	v3	JUCQ [16]	#Results
NPD 6*	91083	21700	132787	295445	2150854
NPD 11*	169833	9189	20070	204426	734214
NPD 12*	74001	5415	11471	14699	734214
NPD 31*	224925	18201	3980	ERROR[1]	1718
AVG	139960	13626	42077	171523[2]	

[1] Error during execution after 221 seconds
[2] Excluding Errors

Table 4: Results for NPD queries from [16] (scale 100-Times in ms)

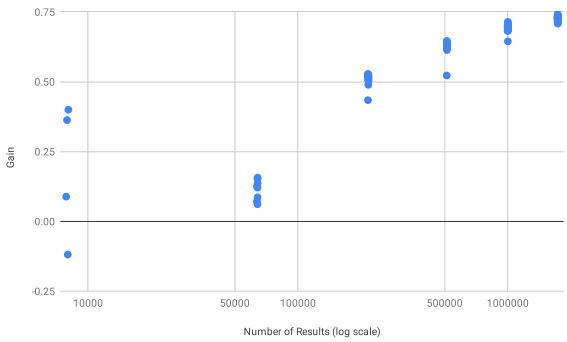


Figure 9: Performance gain with respect to number of results

5.4. Evaluating the Duplicate Elimination Heuristic

In this section we present experimental justification for the use of our heuristic regarding duplicate elimination. For this purpose, we have chosen four query fragments from the LUBM benchmark and four from NPD, such that duplicate elimination is applicable on them, as it was found during the previously described experiments. The experiments of this section were carried out on a machine with an Intel Core i7-3770K processor with 8 cores and 16 GB of RAM running UBUNTU 16.04. As our intention was to examine how our optimizations perform in different underlying systems, we used four different back-ends: PostgreSQL (version 9.3), MySQL (version 5.7) and two of the most widely used proprietary RDBMSs, which due to their license we will call System I and System X. All systems were setup and tuned for usage in a machine with 16GB RAM.

Each query fragment consists of a single select-from-where subquery. The fragments were chosen such that they have varying characteristics regarding the execution time, the number of results and the DTR of the mapping assertion under consideration. In order to test these queries with different selectivities, we applied to them extra filters. As LUBM100 contains information about exactly 100 universities, we used a filter on the university ID attribute in direct correspondence to the percentage of selectivity, whereas for NPD we used different filters for each fragment. We used filters that result in selectivity percentage of 1, 5, 10, 30 and 60, resulting in a total of 40 queries per system. We executed each of these 40 queries with and without duplicate elimination performed, resulting in a total of 240 runs for all systems. The results were obtained with warm caches.

In the upper part of Table 5 (one-time) we present the total execution times for these queries per system, depending on the duplicate elimination strategy. The titles of the first three columns are self-explanatory. The fifth column gives the total time, if always the best strategy was chosen for each system. The fourth column gives the best time, if for each query and each selectivity, the best common strategy was chosen for all systems. This way, the difference between the fourth and fifth column can give an indication of how similar the behaviors of the systems are, whereas comparison of third and fourth columns can give a measure of how well our heuristic takes advantage of this common behavior.

One can observe that the strategy of always performing duplicate elimination is much better than never performing, and that even the strategy of always choosing the best approach is not extremely better. The reason for this result is that for queries with low selectivity, the execution time is much larger and dominates the total time. For these queries, performing duplicate elimination is preferable and sometimes gives up to two orders of magnitude better results. In order to simulate a query mix such that low selectivity queries do not dominate execution time, we also computed results where we give very selective queries a weight, such

that queries with 1% selectivity have been executed 60 times, queries with 5% selectivity have been executed 12 times, etc. We present the total execution time under this setting in the lower part of Table 5. As before, exact times and queries are available at the same location [5].

6. Related Work and Conclusions

Regarding related work, the research of Lanti et al. [16] constitutes the most relevant to ours, as it also deals with cost-based translation. The authors extend the cover-based translation of Bursztyn et al. [3], in order to take into consideration the mappings to arbitrary relational schemas. The authors analyze the database as a preprocessing step, in order to extract useful statistics, such as the cardinality of join results between queries in bodies of mapping assertions whose heads can be joined. Using these statistics, the authors can obtain accurate selectivity estimations for the produced queries. Unfortunately, despite the accurate selectivity estimations, the cost model used to compare the different cover-based reformulations is not realistic, as it assumes that all joins in a CQ are performed using hash joins, which is highly unlikely, and also it is assumed that every input relation is completely scanned. Also, the join order is not taken into consideration at all, something that can have a huge impact in the cost of the query. As we have discussed, this is an inherent problem of a system that operates outside the database engine. The difference with our method is that we use heuristics that apply to different execution plans and database engines, and also, at each step of our method, we compare highly relevant queries, where apart from the relations affected by the combined mapping under consideration, all other input relations and joins between them are the same, such that query selectivity plays the most important role in our decision. Also, we avoid running the query translation process multiple times, whereas in [16] for each different query cover, the rewriting, unfolding and estimation process has to be performed independently. Finally, the authors only consider mappings whose the body is always a CQ over the relational schema.

Since version 3, the Ontop system has departed from the usage of partial evaluation of logic programs for query unfolding. Specifically, it now relies on a query representation which is called intermediate query [30], in order to represent both SPARQL and SQL queries, facilitating the translation of SPARQL query operators like OPTIONAL [29] and GROUP BY. Instead, in this work we concentrate only on CQs over the ontology. We have experimentally shown that our method performs better on average for CQs in comparison with the latest Ontop versions. We believe that it is an interesting topic for future research to also apply cost-based methods to other operators present in SPARQL, possibly combining our results with the line of research carried out in [30, 29].

The work presented by Sequeda et al. [26] is also relevant, as it uses a cost model in order to materialize specific views prior to query execution. This solution in many cases provides efficient query execution, but incurs expensive preprocessing and also, using materialized views in the database increases the database maintenance load, especially for frequently updated tables, as well as the the database size. Also, it is not in line with the overall OBDA approach of providing the end user with access to several underlying data sources, without the need to modify data, and on a practical level, such access may not be even possible. In contrast, we compute specific temporary views during query execution, when we estimate that this will result in lower execution cost, without affecting the original database schema.

Jacques et al. [12] adopt a logic which enables them to avoid mappings when using an object-relational back-end and a combination of data completion and query rewriting. During this process primary keys are used for object identification, removing the need for duplicate elimination. Also, the authors use disjointness axioms in the ontology to further remove the need of duplicate elimination between unions. Gottlob et al. [8] present query rewriting and optimization techniques that eliminate redundant atoms during the application of a resolution based algorithm. To do so, they employ a method that takes into consideration the tuple-generating dependencies (TGDs) of the ontological language they consider, which unlike the DL-Lite languages, considers atoms of arbitrary arity, thus it is conceptually closer to the relational model and does not need separate mappings, so a separate unfolding phase is not needed.

We have identified redundant processing as a bottleneck in OBDA query processing and we have proposed solutions to overcome this problem. We

[5] http://cgi.di.uoa.gr/~dbilid/experiments-obda/

	System	Always	Never	Heuristic	Best (common)	Best(Separate)
one-time	PostgreSQL	13345	168785	12854	12638	12353
	MySQL	281598	-	281685	279522	279265
	SystemI	10733	143616	9906	9693	9502
	SystemX	20558	27479	8588	8803	7280
query-mix	PostgreSQL	167116	618328	144984	146406	143191
	MySQL	1129311	-	1066499	1056659	1056145
	SystemI	135790	520408	102724	101984	99989
	SystemX	167761	220408	93660	90557	83045

Table 5: Query Results for Different Duplicate Elimination Strategies (Times in sec.)

believe that using cost-based planning is a prominent direction towards OBDA query optimization, that has not been fully explored yet. In future work, we plan to incorporate decisions about physical database design by analyzing the mapping assertions. One more direction regarding future research has to do with duplicate elimination in case the OBDA system is equipped with query processing capabilities, in other words when it acts as a mediator. In this setting, along with decisions regarding which query fragments should be evaluated in external databases, one should decide when duplicate elimination should be "pushed" to endpoints or performed by the OBDA processing engine during data import.

Acknowledgments

The present work was co-funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No 825258, and by the Seventh Framework Program (FP7) of the European Commission under Grant Agreement 318338.

References

[1] Bilidas, D., Koubarakis, M.: Efficient duplicate elimination in SPARQL to SQL translation. In: Description Logics (2018)

[2] Bitton, D., DeWitt, D.J.: Duplicate record elimination in large data files. ACM Transactions on database systems (TODS) 8(2), 255–265 (1983)

[3] Bursztyn, D., Goasdoué, F., Manolescu, I.: Teaching an RDBMS about ontological constraints. Proceedings of the VLDB Endowment 9(12), 1161–1172 (2016)

[4] Calvanese, D., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., Xiao, G.: Ontop: Answering sparql queries over relational databases. Semantic Web 8(3), 471–487 (2017)

[5] Chaudhuri, S., Vardi, M.Y.: Optimization of real conjunctive queries. In: Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. pp. 59–70. ACM (1993)

[6] Chortaras, A., Trivela, D., Stamou, G.B.: Optimized query rewriting for OWL 2 QL. In: CADE. vol. 11, pp. 192–206. Springer (2011)

[7] DeWitt, D.J.: The wisconsin benchmark: Past, present, and future. In: Gray, J. (ed.) The Benchmark Handbook for Database and Transaction Systems (2nd Edition). Morgan Kaufmann (1993)

[8] Gottlob, G., Orsi, G., Pieris, A.: Query rewriting and optimization for ontological databases. ACM Transactions on Database Systems (TODS) 39(3), 25 (2014)

[9] Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web 3(2), 158–182 (2005)

[10] Ioannidis, Y.: The history of histograms (abridged). In: Proceedings of the 29th international conference on Very large data bases-Volume 29. pp. 19–30. VLDB Endowment (2003)

[11] Ives, Z.G., Florescu, D., Friedman, M., Levy, A., Weld, D.S.: An adaptive query execution system for data integration. ACM SIGMOD Record 28(2), 299–310 (1999)

[12] Jacques, J.S., Toman, D., Weddell, G.E.: Object-relational queries over $CFD^{\forall-}_{nc}$ knowledge bases: OBDA for the SQL-literate. In: Description Logics (2016)

[13] Kharlamov, E., Hovland, D., Jiménez-Ruiz, E., Lanti, D., Lie, H., Pinkel, C., Rezk, M., Skjæveland, M.G., Thorstensen, E., Xiao, G., Zheleznyakov, D., Horrocks, I.: Ontology based access to exploration data at Statoil. In: International Semantic Web Conference. pp. 93–112. Springer (2015)

[14] Kikot, S., Kontchakov, R., Zakharyaschev, M.: Conjunctive query answering with OWL 2 QL. In: Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning (2012)

[15] Lanti, D., Rezk, M., Xiao, G., Calvanese, D.: The NPD benchmark: Reality check for OBDA systems. In: Proc. of the 18th Int. Conf. on Extending Database Technology (EDBT) (2015)

[16] Lanti, D., Xiao, G., Calvanese, D.: Cost-driven ontology-based data access. In: The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10587, pp. 452–470. Springer (2017)

[17] Lloyd, J.W.: Foundations of logic programming.

Springer Science & Business Media (2012)
[18] Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. The Journal of Logic Programming 11(3-4), 217–242 (1991)
[19] Park, J., Segev, A.: Using common subexpressions to optimize multiple queries. In: Data Engineering, 1988. Proceedings. Fourth International Conference on. pp. 311–319. IEEE (1988)
[20] Pérez-Urbina, H., Horrocks, I., Motik, B.: Efficient query answering for owl 2. The Semantic Web-ISWC 2009 pp. 489–504 (2009)
[21] Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. In: Journal on data semantics, pp. 133–173. Springer (2008)
[22] Rodriguez-Muro, M., Kontchakov, R., Zakharyaschev, M.: Ontology-based data access: Ontop of databases. In: International Semantic Web Conference. pp. 558–573. Springer (2013)
[23] Rodríguez-Muro, M., Rezk, M.: Efficient SPARQL-to-SQL with R2RML mappings. Web Semantics: Science, Services and Agents on the World Wide Web 33, 141–169 (2015)
[24] Roy, P., Seshadri, S., Sudarshan, S., Bhobe, S.: Efficient and extensible algorithms for multi query optimization. ACM SIGMOD Record 29(2), 249–260 (2000)
[25] Sellis, T.K.: Multiple-query optimization. ACM Transactions on Database Systems (TODS) 13(1), 23–52 (1988)
[26] Sequeda, J.F., Arenas, M., Miranker, D.P.: OBDA: query rewriting or materialization? in practice, both! In: International Semantic Web Conference. pp. 535–551. Springer (2014)
[27] Sequeda, J.F., Miranker, D.P.: Ultrawrap: SPARQL execution on relational data. Web Semantics: Science, Services and Agents on the World Wide Web 22, 19–39 (2013)
[28] Swami, A., Schiefer, K.B.: On the estimation of join result sizes. In: International Conference on Extending Database Technology. pp. 287–300. Springer (1994)
[29] Xiao, G., Kontchakov, R., Cogrel, B., Calvanese, D., Botoeva, E.: Efficient handling of sparql optional for obda. In: International Semantic Web Conference. pp. 354–373. Springer (2018)
[30] Xiao, G., Lanti, D., Kontchakov, R., Komla-Ebri, S., Güzel-Kalaycı, E., Ding, L., Corman, J., Cogrel, B., Calvanese, D., Botoeva, E.: The virtual knowledge graph system Ontop. ISWC 2020 - 19th International Semantic Web Conference (2020)

Appendix A. NPD Queries 31-34

```
SELECT DISTINCT ?q ?u
WHERE {
?q :inLithostratigraphicUnit ?u .
?u rdf:type :LithostratigraphicUnit .
}
```

Listing 1: Query NPD 31

```
SELECT DISTINCT ?quadrant ?name
WHERE {
?quadrant rdf:type :Quadrant .
?quadrant :name ?name .
}
```

Listing 2: Query NPD 32

```
SELECT DISTINCT ?unit ?era
WHERE {
?unit :geochronologicEra ?era .
?unit rdf:type :LithostratigraphicUnit .
}
```

Listing 3: Query NPD 33

```
SELECT DISTINCT ?wellbore ?discovery ?year
WHERE {
?wellbore rdf:type :Wellbore .
?wellbore :wellboreForDiscovery ?discovery .
?discovery :discoveryYear ?year
}
```

Listing 4: Query NPD 34

Appendix B

Queries used in the Invekos and Lucas datasets

QUERY01

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geor: <http://www.opengis.net/def/rule/geosparql/>
PREFIX strdf: <http://strdf.di.uoa.gr/ontology#>
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#>
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>

select ?i ?conversion ?l_geom ?i_geom ?dist
where {
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1> ?l_lc1 .
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1_SPEC> ?l_lc1_sp .
?conversion <http://deg.iit.demokritos.gr/lucasLC1> ?l_lc1 .
?conversion <http://deg.iit.demokritos.gr/lucasLC1_spec> ?
l_lc1_sp .
?conversion <http://deg.iit.demokritos.gr/
invekosCropTypeNumber> ?cropNu .
?i <http://ai.di.uoa.gr/invekos/ontology/hasCropTypeNumber> ?
cropNu .
<https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l_geom_id .
?l_geom_id geo:asWKT ?l_geom .
?i geo:hasGeometry ?i_geom_id .
?i_geom_id geo:asWKT ?i_geom .
bind(geof:distance(?l_geom,?i_geom,uom:metre) as ?dist) .
filter(?dist < 10) .
}
```

QUERY 02

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geor: <http://www.opengis.net/def/rule/geosparql/>
PREFIX strdf: <http://strdf.di.uoa.gr/ontology#>
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#>
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>


select ?i ?conversion ?l_geom ?i_geom ?dist
where {
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
```

```
lucas/ontology/hasLC1> ?l_lc1 .
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1_SPEC> ?l_lc1_sp .
?conversion <http://deg.iit.demokritos.gr/lucasLC1> ?l_lc1 .
?conversion <http://deg.iit.demokritos.gr/lucasLC1_spec> ?
l_lc1_sp .
?conversion <http://deg.iit.demokritos.gr/
invekosCropTypeNumber> ?cropNu .
?i <http://ai.di.uoa.gr/invekos/ontology/hasCropTypeNumber> ?
cropNu .
<https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l_geom_id .
?l_geom_id geo:asWKT ?l_geom .
?i geo:hasGeometry ?i_geom_id .
?i_geom_id geo:asWKT ?i_geom .
bind(geof:distance(?l_geom,?i_geom,uom:metre) as ?dist) .
}
order by asc(?dist)
limit 1
```

QUERY 03

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geor: <http://www.opengis.net/def/rule/geosparql/>
PREFIX strdf: <http://strdf.di.uoa.gr/ontology#>
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#>
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>


select ?i ?conversion ?l_geom ?i_geom ?dist
where {
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1> ?l_lc1 .
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1_SPEC> ?l_lc1_sp .
?conversion <http://deg.iit.demokritos.gr/lucasLC1> ?l_lc1 .
?conversion <http://deg.iit.demokritos.gr/lucasLC1_spec> ?
l_lc1_sp .
<https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l_geom_id .
?l_geom_id geo:asWKT ?l_geom .
?i geo:hasGeometry ?i_geom_id .
?i_geom_id geo:asWKT ?i_geom .
bind(geof:distance(?l_geom,?i_geom,uom:metre) as ?dist) .
}
```

```
order by asc(?dist)
limit 1
```

QUERY 04

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geor: <http://www.opengis.net/def/rule/geosparql/>
PREFIX strdf: <http://strdf.di.uoa.gr/ontology#>
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#>
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>



select ?i ?conversion ?l_geom ?i_geom ?dist
where {
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1> ?l_lc1 .
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1_SPEC> ?l_lc1_sp .
?conversion <http://deg.iit.demokritos.gr/lucasLC1> ?l_lc1 .
?conversion <http://deg.iit.demokritos.gr/lucasLC1_spec> ?
l_lc1_sp .
<https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l_geom_id .
?i geo:hasGeometry ?i_geom_id .
?i_geom_id geo:asWKT ?i_geom .
?l_geom_id geo:asWKT ?l_geom .
bind(geof:distance(?l_geom,?i_geom,uom:metre) as ?dist) .
{
        select ?dist2
        where {
        <https://ai.di.uoa.gr/lucas/resource/1> <https://
ai.di.uoa.gr/lucas/ontology/hasLC1> ?l2_lc1 .
        <https://ai.di.uoa.gr/lucas/resource/1> <https://
ai.di.uoa.gr/lucas/ontology/hasLC1_SPEC> ?l2_lc1_sp .
        ?conversion2 <http://deg.iit.demokritos.gr/lucasLC1> ?
l2_lc1 .
        ?conversion2 <http://deg.iit.demokritos.gr/
lucasLC1_spec> ?l2_lc1_sp .
        ?conversion2 <http://deg.iit.demokritos.gr/
invekosCropTypeNumber> ?cropNu2 .
        ?i2 <http://ai.di.uoa.gr/invekos/ontology/
hasCropTypeNumber> ?cropNu2 .
        <https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l2_geom_id .
```

```
        ?l2_geom_id geo:asWKT ?l2_geom .
        ?i2 geo:hasGeometry ?i2_geom_id .
        ?i2_geom_id geo:asWKT ?i2_geom .
        bind(geof:distance(?l2_geom,?i2_geom,uom:metre) as ?
dist2) .
        }
        order by asc(?dist2)
        limit 1
}
filter(?dist <= ?dist2)
}
```

QUERY 05

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geor: <http://www.opengis.net/def/rule/geosparql/>
PREFIX strdf: <http://strdf.di.uoa.gr/ontology#>
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#>
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>



select ?conversion ?l_lc1 ?l_lc1_sp ?cropNu ?l_geom ?i_geom ?i ?
dist
where {
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1> ?l_lc1 .
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1_SPEC> ?l_lc1_sp .
?conversion <http://deg.iit.demokritos.gr/lucasLC1> ?l_lc1 .
?conversion <http://deg.iit.demokritos.gr/lucasLC1_spec> ?
l_lc1_sp .
?conversion <http://deg.iit.demokritos.gr/
invekosCropTypeNumber> ?cropNu .
?i <http://ai.di.uoa.gr/invekos/ontology/hasCropTypeNumber> ?
cropNu .
<https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l_geom_id .
?l_geom_id geo:asWKT ?l_geom .
?i geo:hasGeometry ?i_geom_id .
?i_geom_id geo:asWKT ?i_geom .
bind(geof:distance(?l_geom,?i_geom,uom:metre) as ?dist) .
{
        select ?dist2
        where {
```

```
        <https://ai.di.uoa.gr/lucas/resource/1> <https://
ai.di.uoa.gr/lucas/ontology/hasLC1> ?l2_lc1 .
        <https://ai.di.uoa.gr/lucas/resource/1> <https://
ai.di.uoa.gr/lucas/ontology/hasLC1_SPEC> ?l2_lc1_sp .
        ?conversion2 <http://deg.iit.demokritos.gr/lucasLC1> ?
l2_lc1 .
        ?conversion2 <http://deg.iit.demokritos.gr/
lucasLC1_spec> ?l2_lc1_sp .
        <https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l2_geom_id .
        ?l2_geom_id geo:asWKT ?l2_geom .
        ?i2 geo:hasGeometry ?i2_geom_id .
        ?i2_geom_id geo:asWKT ?i2_geom .
        bind(geof:distance(?l2_geom,?i2_geom,uom:metre) as ?
dist2) .
        filter(?dist2 < 10) .
        }
        order by asc(?dist2)
        limit 1
}

filter(?dist <= ?dist2)
}

QUERY 06

PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geor: <http://www.opengis.net/def/rule/geosparql/>
PREFIX strdf: <http://strdf.di.uoa.gr/ontology#>
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#>
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>



select ?i ?conversion ?l_geom ?i_geom ?dist
where {
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1> ?l_lc1 .
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1_SPEC> ?l_lc1_sp .
?conversion <http://deg.iit.demokritos.gr/lucasLC1> ?l_lc1 .
?conversion <http://deg.iit.demokritos.gr/lucasLC1_spec> ?
l_lc1_sp .
?conversion <http://deg.iit.demokritos.gr/
invekosCropTypeNumber> ?cropNu .
```

```
?i <http://ai.di.uoa.gr/invekos/ontology/hasCropTypeNumber> ?
cropNu .
<https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l_geom_id .
?l_geom_id geo:asWKT ?l_geom .
?i geo:hasGeometry ?i_geom_id .
?i_geom_id geo:asWKT ?i_geom .
bind(geof:distance(?l_geom,?i_geom,uom:metre) as ?dist) .
{
        select (geof:distance(?l2_geom,?i2_geom,uom:metre) as ?
dist2)
        where {
        <https://ai.di.uoa.gr/lucas/resource/1> <https://
ai.di.uoa.gr/lucas/ontology/hasLC1> ?l2_lc1 .
        <https://ai.di.uoa.gr/lucas/resource/1> <https://
ai.di.uoa.gr/lucas/ontology/hasLC1_SPEC> ?l2_lc1_sp .
        ?conversion2 <http://deg.iit.demokritos.gr/lucasLC1> ?
l2_lc1 .
        ?conversion2 <http://deg.iit.demokritos.gr/
lucasLC1_spec> ?l2_lc1_sp .
        ?conversion2 <http://deg.iit.demokritos.gr/
invekosCropTypeNumber> ?cropNu2 .
        ?i2 <http://ai.di.uoa.gr/invekos/ontology/
hasCropTypeNumber> ?cropNu2 .
        <https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l2_geom_id .
        ?l2_geom_id geo:asWKT ?l2_geom .
        ?i2 geo:hasGeometry ?i2_geom_id .
        ?i2_geom_id geo:asWKT ?i2_geom .
        }
        order by asc(?dist2)
        limit 1
}
filter(10 <= ?dist2)
}
order by asc(?dist)
limit 1
```

QUERY 07

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geor: <http://www.opengis.net/def/rule/geosparql/>
PREFIX strdf: <http://strdf.di.uoa.gr/ontology#>
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#>
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>
```

```
select ?i ?conversion ?l_geom ?i_geom ?dist
where {
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1> ?l_lc1 .
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1_SPEC> ?l_lc1_sp .
?conversion <http://deg.iit.demokritos.gr/lucasLC1> ?l_lc1 .
?conversion <http://deg.iit.demokritos.gr/lucasLC1_spec> ?
l_lc1_sp .
<https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l_geom_id .
?l_geom_id geo:asWKT ?l_geom .
?i geo:hasGeometry ?i_geom_id .
?i_geom_id geo:asWKT ?i_geom .
bind(geof:distance(?l_geom,?i_geom,uom:metre) as ?dist) .
{
        select (geof:distance(?l2_geom,?i2_geom,uom:metre) as ?
dist2)
        where {
        <https://ai.di.uoa.gr/lucas/resource/1> <https://
ai.di.uoa.gr/lucas/ontology/hasLC1> ?l2_lc1 .
        <https://ai.di.uoa.gr/lucas/resource/1> <https://
ai.di.uoa.gr/lucas/ontology/hasLC1_SPEC> ?l2_lc1_sp .
        ?conversion2 <http://deg.iit.demokritos.gr/lucasLC1> ?
l2_lc1 .
        ?conversion2 <http://deg.iit.demokritos.gr/
lucasLC1_spec> ?l2_lc1_sp .
        ?conversion2 <http://deg.iit.demokritos.gr/
invekosCropTypeNumber> ?cropNu2 .
        ?i2 <http://ai.di.uoa.gr/invekos/ontology/
hasCropTypeNumber> ?cropNu2 .
        <https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l2_geom_id .
        ?l2_geom_id geo:asWKT ?l2_geom .
        ?i2 geo:hasGeometry ?i2_geom_id .
        ?i2_geom_id geo:asWKT ?i2_geom .

        filter(?dist2 < 10)
        }
        order by asc(?dist2)
        limit 1
}
filter(?dist < ?dist2)
}
```

QUERY 08

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geor: <http://www.opengis.net/def/rule/geosparql/>
PREFIX strdf: <http://strdf.di.uoa.gr/ontology#>
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#>
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>

SELECT * WHERE {
  ?i <http://earthanalytics.eu/fs/ontology/hasCropTypeNumber> ?
nu.
  ?i geo:hasGeometry ?geomI.
  ?geomI geo:asWKT ?wktI.
  ?r <http://ai.di.uoa.gr/eu-hydro/ontology/hasRiver_Net_p> ?o.
  ?o geo:hasGeometry ?geom.
  ?geom geo:asWKT ?wkt.
  BIND(geof:distance(?wktI, ?wkt, uom:metre) as ?dist).
  FILTER(?dist < 100).
}
```

QUERY 09

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geor: <http://www.opengis.net/def/rule/geosparql/>
PREFIX strdf: <http://strdf.di.uoa.gr/ontology#>
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#>
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>



select ?conversion ?l_lc1 ?l_lc1_sp ?l_geom ?i_geom ?i ?dist
where {
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1> ?l_lc1 .
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1_SPEC> ?l_lc1_sp .
?conversion <http://deg.iit.demokritos.gr/lucasLC1> ?l_lc1 .
?conversion <http://deg.iit.demokritos.gr/lucasLC1_spec> ?
l_lc1_sp .
<https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l_geom_id .
?l_geom_id geo:asWKT ?l_geom .
```

```
?i geo:hasGeometry ?i_geom_id .
?i_geom_id geo:asWKT ?i_geom .
bind(geof:distance(?l_geom,?i_geom,uom:metre) as ?dist) .
filter(?dist < 10)
{
        select ?dist2
        where {
        <https://ai.di.uoa.gr/lucas/resource/1> <https://
ai.di.uoa.gr/lucas/ontology/hasLC1> ?l2_lc1 .
        <https://ai.di.uoa.gr/lucas/resource/1> <https://
ai.di.uoa.gr/lucas/ontology/hasLC1_SPEC> ?l2_lc1_sp .
        ?conversion2 <http://deg.iit.demokritos.gr/lucasLC1> ?
l2_lc1 .
        ?conversion2 <http://deg.iit.demokritos.gr/
lucasLC1_spec> ?l2_lc1_sp .
        ?conversion2 <http://deg.iit.demokritos.gr/
invekosCropTypeNumber> ?cropNu2 .
        ?i2 <http://ai.di.uoa.gr/invekos/ontology/
hasCropTypeNumber> ?cropNu2 .
        <https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l2_geom_id .
        ?l2_geom_id geo:asWKT ?l2_geom .
        ?i2 geo:hasGeometry ?i2_geom_id .
        ?i2_geom_id geo:asWKT ?i2_geom .
        bind(geof:distance(?l2_geom,?i2_geom,uom:metre) as ?
dist2) .
        }
        order by asc(?dist2)
        limit 1
}
filter(?dist < ?dist2)
}

QUERY 10

PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geor: <http://www.opengis.net/def/rule/geosparql/>
PREFIX strdf: <http://strdf.di.uoa.gr/ontology#>
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#>
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>



select ?conversion ?l_lc1 ?l_lc1_sp ?l_geom ?dist
where {
```

```
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1> ?l_lc1 .
<https://ai.di.uoa.gr/lucas/resource/1> <https://ai.di.uoa.gr/
lucas/ontology/hasLC1_SPEC> ?l_lc1_sp .
?conversion <http://deg.iit.demokritos.gr/lucasLC1> ?l_lc1 .
?conversion <http://deg.iit.demokritos.gr/lucasLC1_spec> ?
l_lc1_sp .
<https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l_geom_id .
?l_geom_id geo:asWKT ?l_geom .
?i geo:hasGeometry ?i_geom_id .
?i_geom_id geo:asWKT ?i_geom .
bind(geof:distance(?l_geom,?i_geom,uom:metre) as ?dist) .
{
        select (geof:distance(?l2_geom,?i2_geom,uom:metre) as ?
dist2)
        where {
        <https://ai.di.uoa.gr/lucas/resource/1> <https://
ai.di.uoa.gr/lucas/ontology/hasLC1> ?l2_lc1 .
        <https://ai.di.uoa.gr/lucas/resource/1> <https://
ai.di.uoa.gr/lucas/ontology/hasLC1_SPEC> ?l2_lc1_sp .
        ?conversion2 <http://deg.iit.demokritos.gr/lucasLC1> ?
l2_lc1 .
        ?conversion2 <http://deg.iit.demokritos.gr/
lucasLC1_spec> ?l2_lc1_sp .
        <https://ai.di.uoa.gr/lucas/resource/1> geo:hasGeometry ?
l2_geom_id .
        ?l2_geom_id geo:asWKT ?l2_geom .
        ?i2 geo:hasGeometry ?i2_geom_id .
        ?i2_geom_id geo:asWKT ?i2_geom .
        }
        order by asc(?dist2)
        limit 1
}
filter(10 <= ?dist2)
}
order by asc(?dist)
limit 1
```