Strabo 2: Distributed Management of Massive Geospatial RDF Datasets

Dimitris Bilidas[1], Theofilos Ioannidis[1], Nikos Mamoulis[2], and Manolis Koubarakis[1]

[1] National and Kapodistrian University of Athens, Greece
{d.bilidas,tioannid,koubarak}@di.uoa.gr
[2] University of Ioannina, Greece
nikos@cs.uoi.gr

Abstract. We present Strabo 2, a distributed geospatial RDF store able to process GeoSPARQL queries over massive RDF datasets. Strabo 2 is based on robust technologies, able to scale on TBs of data distributed on hundreds of nodes. Specifically, we use the Spark framework, enhanced with the geospatial library SEDONA, for distributed in-memory processing on Hadoop clusters, and Hive for compact persistent storage of RDF data. Strabo 2 employs a flexible design that can store and partition thematic RDF data using different relational schemas, and spatial data in a separate Hive table, by taking into consideration the GeoSPARQL vocabulary. Strabo 2 is cluster friendly both memory and disk-wise, since it compresses triples using a partial encoding technique in addition to Parquet data file format compression schemes. GeoSPARQL queries are translated into the Spark SQL dialect, enhanced with the spatial functions and predicates offered by SEDONA. During this process the system takes into consideration SEDONA's capabilities for both spatial selections and spatial joins, in order to apply optimizations that result in efficient query processing. We experimentally test Strabo 2 on an award winning Hadoop based cluster environment and exhibit Strabo 2's excellent scalability while handling massive synthetic and real world datasets. We also show that Strabo 2 clearly outperforms state of the art centralized engines in a single server setup, once the dataset size increases beyond few GBs.

1 Introduction

As the spatial information in the web of linked data has been increasing steadily over the past decade, many systems that perform geospatial processing over RDF graphs have been developed, mainly targeting the GeoSPARQL vocabulary and query language, an OGC standard for representing and querying spatial information in RDF. At the same time, as large RDF datasets become available, the need for distributed processing of SPARQL queries has lead to the development of many RDF query engines that rely on big data tools and technologies for storing and processing massive RDF data. Some of the most prominent approaches

rely on distributed in-memory big data frameworks, mainly Apache Spark, like for example S2RDF [23] and PRoST [7].

However, despite the importance of the spatial dimension of these massive datasets, to the best of our knowledge none of the distributed RDF engines supports execution of spatial queries. This leads to a lack of spatial RDF engines able to scale to the continously increasing spatial information in the linked data cloud. For example, the state-of-the art geospatial RDF store Strabon can only handle up to 100GBs of point data and still be able to answer simple geospatial queries (selections over a rectangular area) efficiently (in a few seconds). Competitor systems like GraphDB perform similarly. If the complexity of geometries in the dataset increases (i.e., we have multi-polygons), not even the aforementioned performance can be achieved for both Strabon and GraphDB.

Reviewing benchmarks with big geospatial semantic datasets [15] for mature centralized RDF stores reveal the shortcomings of this category of systems handling large datasets (range of few GBs of size). In sum, the main shortcomings of such systems include: i) high bulk loading times, with mostly single threaded reading, usually one file at a time, followed by single-thread re-indexing. ii) only DBMS-based RDF stores seem to be able to marginally handle spatial selections and spatial joins against datasets of several GBs size and this depends very much on the DMBS tuning and iii) mostly single-threaded implementations of algorithms [18] and components, leaves unexploited the potential of these systems to vertically scale to the maximum of their potential on a regular multi-core server-grade single node. This is even more true for open source or free versions of these systems. Some commercial systems offer limited parallelization only in some of their components, i.e., bulk loaders in Ontotext GraphDB Free and offer full multi-threaded capabilities in their licensed product versions.

To address the above limitations, the main contributions of this work are:

- We present Strabo 2, the first distributed system that is able to process GeoSPARQL queries over massive geospatial RDF datasets on Spark clusters.
- We present a flexible design that can store thematic RDF data using different relational schemas, and spatial data in a separate Hive table, by taking into consideration the GeoSPARQL vocabulary. We use the query translation mechanism of Ontop-spatial in order to obtain the final set of spatial SQL queries from the initial GeoSPARQL query.
- We optimize the translation process based on spatial joins and also use the spatial partitioning and indexing capabilities of the Apache Sedona library in order to achieve efficient query execution.
- We present an extensive experimental evaluation in order to examine the scalability of the system with respect to different query characteristics, like the spatial and thematic selectivities. We also compare the system with state of the art centralized solutions for smaller datasets that can be ingested and processed by single-node installations.

2 Related Work

In this section we present related work. The examination of the capabilities and design choices of the systems we presented were taken into consideration in the definition of the architecture of the STRABO 2 distributed GeoSPARQL engine.

Centralized GeoSPARQL query processing. Strabon is one of the first systems offering GeoSPARQL support. Strabon extends the well-known RDF store Sesame and uses the PostGIS spatially-enabled DBMS as the backend. GraphDB[3] is a semantic graph database enhanced with geospatial capabilities. For its geospatial capabilities, it relies on a uSeekM implementation and Lucene Spatial. The spatial index mechanism is controlled through an optional GeoSPARQL plugin. Other geospatial RDF stores include: Parliament [3] which uses a standard R-tree as its spatial index and concentrates on optimizing query patterns (using the Topology Vocabulary extension of GeoSPARQL) while it omits optimization for functions in the filter clause of a query, Oracle Spatial and Graph which supports the GeoSPARQL standard and also uses an R-Tree and Stardog, a popular knowledge graph platform that allows the use of custom connectors in order to enable geospatial support.

A detailed comparative study of centralized geospatial RDF stores is [14], where different systems were benchmarked and evaluated with datasets of up to 90GB size. Strabon [19] achieves the best overall score in most scenarios, such as the macro and scalability, whereas GraphDB[4] also performed very well on bulk loading and certain types of queries. These results motivated us to use Strabon and GraphDB as the baseline systems for the performance comparison with the new distributed implementation we are presenting in Section 3.

Apart from triple stores that store and query RDF graphs, GeoSPARQL querying is also supported in the context of Ontology-Based Data Access (OBDA), where data are stored in a spatially-enabled RDBMS, and GeoSPARQL to SQL translation is performed by the system in order to delegate query processing to the underlying database. Ontop-spatial [4], a geospatial extension of Ontop [6], was the first OBDA system able to answer GeoSPARQL queries on top of geospatial relational databases, performing on-the-fly GeoSPARQL-to-SQL translation using ontologies and mappings. The aim of Ontop-spatial is to allow integrating multiple geospatial sources, without converting, materializing and persisting original data as RDF. More recently, support for the GeoSPARQL query language has also been added to the main Ontop branch since version 4.1[5].

SPARQL query processing in the cloud. The increasing size of available RDF data has exceeded the capacity of single node systems. As a result, a large number of approaches for querying RDF graphs in the cloud rely on existing robust and widely used distributed data processing frameworks [17]. Among these systems, in-memory distributed data processing frameworks, and especially Spark,

[3] http://graphdb.ontotext.com/documentation/free/
[4] http://graphdb.ontotext.com/documentation/free/
[5] https://ontop-vkg.org/guide/releases.html#_4-1-0-february-28-2021

are amongst the most prominent and fast solutions for SPARQL processing. For example S2RDF [23] uses Spark to precompute specific semi-joins, PRoST [7] explores different storage strategies for RDF data as tabular data used by Spark, such as a single triples table, vertical partitioning and property tables. [2] extend the work of PRoST by examining several processing option in Spark. SPARQLGX [11] compiles triple patterns of a SPARQL query into operations over Spark's resilient distributed datasets (RDDs). S2RDF and PRoST are the more relevant systems to our approach, as they employ query translation in order to transform each SPARQL query into an SQL query that is executed using the corresponding API offered by Spark.

Parallel and distributed geospatial query processing. The first systems for distributed spatial query processing on the Hadoop ecosystem were implemented as extensions of the MapReduce paradigm, such as SpatialHadoop [9], Hadoop-GIS [1], and Parallel Secondo [20]. Hadoop provides a fault tolerant environment for parallel execution, but storing intermediate results to disk according to MapReduce increases the execution time for spatial operations. Hence, the in-memory execution model of Spark became very popular as it reduces the execution time drastically, compared to MapReduce jobs [12]. Following this trend, many Spark-based systems included geospatial support, most notable of which are the systems STARK [13], GeoSpark/Sedona [26, 27], Magellan [24] and Spatial-Spark [25]. In the context of this work, we will only consider the Spark-based systems as they reportedly achieve better performance [12, 13] than the Hadoop-based systems. Eldawy and Mokbel have presented a survey paper and tutorial on these systems [8, 10].

The above systems have also been compared regarding their functionality in [22]. GeoSpark/Sedona is found to be the most complete system, both in terms of functionality and performance, as it now offers support for spatial datatypes such as points, rectangles, polygons and lines, and spatial operations such as different kinds of spatial joins (e.g., contains, intersects, touches, overlaps) and distance-based joins. It supports several partitioning techniques such as Equal-grid, Hilbert, R-Tree, Voronoi and Quad-Tree. Spatial indexes like R-Tree or Quad-Tree are provided in the Spatial Query Processing layer. Sedona's index can be persisted either in memory or in disk for later use from the same program. It can be used via its Java or Scala API and also via an SQL interface that expands Spark SQL. Finally, Sedona is currently an Apache incubating project[6] and it is actively maintained and enhanced[7].

Finally, to the best of our knowledge, the only system that deals with a form of distributed spatial RDF processing is the DiStRDF system [21]. DistRDF accepts SPARQL queries, along with a set of spatial and temporal constraints for each query. DistRDDF does not support the GeoSPARQL language, it only considers point geometries and the user can only express a spatial range query for a given box or circle. In contrast, Strabo 2 accepts GeoSPARQL queries, sup-

[6] https://sedona.apache.org/
[7] https://github.com/apache/incubator-sedona/

ports different kinds of geometries, and besides spatial range queries, it also supports spatial joins and distance-based joins queries defined in the GeoSPARQL language.

3 The Strabo 2 System for Distributed GeoSPARQL Processing

In this section we present the technical details of the STRABO 2 system, starting with its architecture, which is shown in Figure 1.

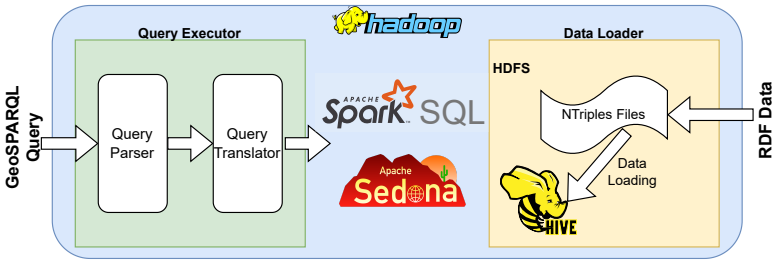

Fig. 1: Architectural overview of STRABO 2

The system consists of two main modules: the data loader and the query executor. The data loader is shown in the right part of Figure 1 and is responsible for reading and importing into a HIVE database the RDF files from the file system. The query executor module accepts the input GeoSPARQL queries, and it performs query translation. The result of this process is a series of Spark SQL queries that also contain spatial functions provided by the Sedona library.

3.1 Data Loader

The STRABO 2 data loader imports RDF graphs encoded 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. The output of the loading process is a set of tables in a Hive database.

The parameters of the data loader are the following: (i) The name of the output Hive database. (ii) Selecting the relational schema for the thematic data. Currently only vertical partitioning and a single triples table are supported. (iii) Optional physical partitioning on the columns of the created tables. (iv) Using HiveQL or Spark SQL dataframe API as the data definition language. (v) Hive table format: Parquet is the default file format, as it is highly efficient and also uses columnar compression, which results in decreased size. (vi) A JSON file with common IRI namespace prefixes related to the ingested dataset.

The `Common Prefixes` JSON file is constructed manually per imported dataset. This file guides the partial dictionary encoding of the IRIs at a later stage and, at the very least, it should contain common namespace prefixes from XML, RDF,

RDFS and GeoSPARQL vocabularies which are encountered in many datasets. The data loader uses 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. After the initial parsing to Spark RDD, the data loader 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. Finally, using the common namespace prefixes in `nsprefixes`, the data loader applies partial dictionary encoding on all IRIs.

After the initial loading, the data loader creates the *geometry linking tables*, aiming to achieve efficient spatial processing during query execution. These tables take into consideration the GeoSPARQL vocabulary in order to store in the same table information about the entities, the corresponding geometries and the serialization of the geometries, so that during query execution joins between the corresponding tables of the VP schema (or the corresponding self joins on the single triples table) can be avoided. Also, during this step, for these tables, the loader creates the binary geometry column from the serializations, using the `ST_GeomFromText` function of Sedona. The geometry linking tables are created as follows. For each VP table that corresponds to some subproperty of the GeoSPARQL `hasGeometry` property, we compute the object-subject join with any other table that corresponds to some subroperty of the GeoSPARQL `hasSerialization` property.

As an example consider the following triples, where we have omitted the full IRIs for ease of presentation. The example comes from a sea ice mapping using satellite images application that we have implemented using Strabo 2 in the context of European project ExtremeEarth[8]. *Drift ice* is sea ice that is not attached to the shoreline or any other fixed object (shoals, grounded icebergs, etc.). Unlike fast ice, which is "fastened" to a fixed object, drift ice is carried along by winds and sea currents, hence its name.[9].

```
Ice1 type IceObservation .     Ice1 hasCT "Drift Ice" .
Ice1 observationGeom Geo1 .    Geo1 asWKT "POINT (10 10)" .
Img1 type SatelliteImage .     Img1 imageGeom Geo2 .
Geo2 asWKT "POLYGON (8 8, 12 8, 12 12, 8 12, 8 8 )" .
```

According to the VP schema, a separate table corresponding to each distinct predicate will be created in Hive. These are the first five tables shown in Figure 2. Also, in this example, the properties `imageGeom` and `observationGeom` are subproperties of the GeoSPARQL `hasGeometry` property, and the property `asWKT` is a GeoSPARQL property that is subproperty of `hasSerialization`. As a result, the geometry linking tables `observationGeom-asWKT` and `imageGeom-asWKT` will also be created.

[8] https://earthanalytics.eu/
[9] https://en.wikipedia.org/wiki/Drift_ice

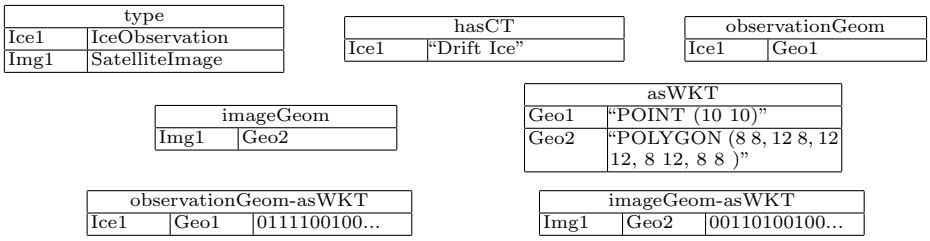

Fig. 2: Tables created in Hive

After data loading, during system startup, we also create in-memory spatial indexes on the geometry columns of the geometry linking tables. Due to the fact that clustered indexes cannot be defined when accessing Sedona from the SQL interface, we use the Scala/RDD interface. The following code is executed for each geometry linking table:

```
var spatialDf = _sqlContext.sql("SELECT entity,
    geometry, binary_geometry FROM observationGeom-asWKT")
spatialDf.registerTempTable(tableStat.tName)
spatialRDD = Adapter.toSpatialRdd(spatialDf, "binary_geometry")
spatialRDD.buildIndex(IndexType.QUADTREE, false)
spatialRDD.indexedRawRDD.persist(StorageLevel.MEMORY_AND_DISK);
```

3.2 Query Executor

The second module of STRABO 2 is the query executor shown in the left part of Figure 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 (and in some cases the spatial RDD indexes) created by the loader. The spatial operators of GeoSPARQL are translated to corresponding spatial functions and predicates 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 [5]. Ontop-spatial is a system for GeoSPARQL-to-SQL query translation over arbitrary relational schemas, through the means of mappings defined in the W3C recommendation mapping language R2RML[10], that construct RDF terms from the database values.

In order to use Ontop-spatial for query translation in the query executor module of STRABO 2, 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, 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, the user has to manually construct the mappings. In our case, as the Hive schema is predetermined from the loader, we can avoid this process and instead, during system startup, automatically construct the mappings for the thematic and the

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

geometry linking tables. As an example, consider the table `hasCT` of Figure 2 constructed from the data loader. The following mapping is generated and provided as input to the Ontop-spatial translation mechanism:

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

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 Hive created tables. 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 Strabo 2.

Once Ontop-spatial has been provided with the set of mappings and the metadata, it is ready to accept GeoSPARQL queries. The input GeoSPARQL query is initially parsed and transformed in an intermediate form, based on logic programs, and finally into SQL queries on the dialect of Spark SQL. During this procedure, the spatial operators of GeoSPARQL are transformed to spatial functions and predicates provided by Apache Sedona. Currently, we support the translation of all simple features relations of GeoSPARQL, and also of the GeoSPARQL functions, including the distance function, that corresponds to distance based joins in Sedona. In order to demonstrate query translation, consider the following initial GeoSPARQL query:

```
SELECT ?img WHERE {
?observation type IceObservation .
?observation hasCT "Drift Ice"^^<http://www.w3.org/2001/XMLSchema#string> .
?observation observationGeom ?obsGeo .  ?g1 geo:asWKT ?obsWKT .
?img type SatelliteImage .  ?img imageGeom ?imgGeo .  ?imgGeo asWKT ?imgWKT .
FILTER (geof:sfIntersects(?obsWKT, ?imgWKT)). }
```

This query asks for satellite images, such that the geometry of the image intersects with the geometry of an observation that has class type drift ice. The query uses the GeoSPARQL topological relation `geof:sfIntersects` with arguments the corresponding geometries. Default 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 qview5.subject AS img
FROM  type qview1, hasCT qview2, observationGeom qview3, asWKT qview4,
      type qview5, imageGeom qview6, asWKT qview7
WHERE
 qview1.object = 'IceObservation' AND qview2.object = 'Drift Ice' AND
```

```
qview1.subject = qview2.subject AND qview1.subject = qview3.subject AND
qview3.object = qview4.subject AND qview5.object = 'SatelliteImage' AND
qview5.subject = qview6.subject AND qview6.object = qview7.subject AND
ST_Intersects(ST_GeoFromText(qview4.object),ST_GeoFromText(qview7.object))
```

Using Geometry Linking Tables. The default translation only uses the tables of the VP schema. In order to obtain a more efficient query, during translation we identify joins between subproperties of `hasGeometry` and `hasSerialization`. According to the GeoSPARQL vocabulary, in order to access the geometry serialization of an entity, the query needs to contain two triple patterns. The first pattern relates the entity with its geometry though the `hasGeometry` property (or a subproperty), and the second pattern relates the geometry with a serialization through the `hasSerialization` property (or a subproperty). By taking advantage of the fact that such joins between triple patterns usually occur in GeoSPARQL queries, and having computed the corresponding geometry linking tables during import, we can save one join if we replace the access to the two tables, with access to the corresponding geometry linking table. In our example query, we identify two such cases, one for the join between `observationGeom` and `asWKT`, and the second one for the join between `imageGeom` and `asWKT`. The optimized SQL query is shown below, and it contains two less joins from the default translation.

```
SELECT qview4.subject AS img
FROM type qview1, hasCT qview2, observationGeom-asWKT qview3,
     type qview4, imageGeom-asWKT qview5
WHERE
qview1.object = 'IceObservation' AND qview2.object = 'Drift Ice' AND
qview1.subject = qview2.subject AND qview1.subject = qview3.entity AND
qview4.object = 'SatelliteImage' AND qview4.subject = qview5.entity AND
ST_Intersects(qview3.binary,qview5.binary)
```

Pushing Thematic Processing Before Spatial Joins. The query produced so far is optimized in the sense that it avoids extra thematic joins between the geometry related tables, but it still contains a spatial join that poses a potentially heavy burden on the execution engine. The reason for that is that in order to perform the spatial join, Sedona will either perform a distributed GSJoin algorithm, where it will spatially partition the two input operands of the join, and also create a local spatial index at each partition, or, if the datasets are small, it will perform a broadcast join algorithm, where it will partition the larger input, and it will replicate the smaller [27]. In any case, the spatial join will lead to data shuffling and computationally heavy processing. Also, the Spark catalyst optimizer treats the spatial UDF as a black box, as it does not take into consideration the cost of the spatial partitioning and indexing, and in many cases it will not optimally optimize the produced query with respect to the join order of the operators. For this reason, in STRABO 2 query translator we follow a heuristic that aims at minimizing the size of input operands of the spatial join. Specifically, we push thematic processing before the spatial join operators in the final produced query.

The rationale is that thematic processing on each side of the spatial join input will limit the size of the intermediate result that has to be spatially partitioned and indexed. In our example query, for the input that corresponds to ice observations, we will first apply the filter that ensures that we only need resources about ice observations, we will perform the thematic join corresponding to the `hasCT` predicate and also the filter that ensures that the classification result is drift ice. This will limit the number of geometries that need to be processed, in contrast with a bad execution plan, where, for example, we first partition all geometries and then perform the join corresponding to the `hasCT` predicate and filter out the observations that correspond to drift ice. In order to ensure the execution plan according to our heuristic, we identify spatial joins during the translation process, and then we decompose the result in different subqueries, that are sequentially sent for execution. In our example, we will first produce two subqueries that create temporary views corresponding to the two inputs of the spatial join, and one final query that performs the spatial join between these two intermediate results:

```
CREATE TEMPORARY VIEW TEMP1 AS
SELECT qview3.binary as qview3_binary
FROM type qview1, hasCT qview2, observationGeom-asWKT qview3,
WHERE
qview1.object = 'IceObservation' AND qview2.object = 'Drift Ice' AND
qview1.subject = qview2.subject AND qview1.subject = qview3.entity

CREATE TEMPORARY VIEW TEMP2 AS
SELECT qview5.binary as qview5_binary, qview4.subject as qview4_subject
FROM type qview4, imageGeom-asWKT qview5
WHERE
qview4.object = 'SatelliteImage' AND qview4.subject = qview5.entity

SELECT TEMP2.qview4_subject AS img
FROM TEMP1, TEMP2
WHERE ST_Intersects(TEMP1.qview3_binary,TEMP2.qview5_binary)
```

Using Persistent Spatial Indexing and Partitioning. As described in Section 3.1, both thematic and spatial RDF data are stored in disk in a Hive database according to the specified schema and the geometry linking tables of the dataset. During query execution, the Spark execution engine loads the necessary fragments of thematic data in memory. Geometries have the same treatment. In case of 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. 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, that accesses the cached spatial RDDs that have been created during import. 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 spatial index, and then by replacing the atoms corresponding to the specific spatial operation by temporary atoms that correspond to the intermediate result after accessing the persistent spatial structure. As an example, 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 type IceObservation . ?x hasCT ?ctName .
        ?x hasGeometry ?geo1 .   ?geo1 asWKT ?wkt .
FILTER(geof:sfIntersects(?wkt,"POLYGON((1 0,3 0,4 4,1 0))"^^geo:wktLiteral)).}
```

The translation result without using the spatial RDD that has been created during import, is the following:

```
SELECT qview1.subject AS x, qview2.object AS ctName
FROM type qview1, hasCT qview2, observationGeom-asWKT qview3,
WHERE
qview1.object = 'IceObservation' AND qview1.subject = qview2.subject AND
qview1.subject = qview3.entity AND
ST_Intersects(qview3.binary,POLYGON((1 0,3 0,4 4,1 0)))
```

By identifying the spatial filter during the transaltion, we can see that we can use the spatialRDD for its evaluation. In order to do that, we are replacing access to table `observationGeom-asWKT` with a new temporary table, that corresponds to the result of the access to the spatial index. First, 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((1 0,3 0,4 4,1 0))")
val considerBoundaryIntersection = true
val usingIndex = true
var queryResult = RangeQuery.SpatialRangeQuery(spatialRDD,
        rangeQueryWindow,considerBoundaryIntersection, usingIndex)
Adapter.toDf(queryResult).createGlobalTempView("temp")
```

Finally, we issue the following SQL query, that accesses the temporary result instead of the table `observationGeom-asWKT`:

```
SELECT qview1.subject AS x, qview2.object AS ctName
FROM type qview1, hasCT qview2, temp qview3,
WHERE qview1.object = 'IceObservation'
AND qview1.subject = qview2.subject AND qview1.subject = qview3.entity
```

4 Experimental Results

In this section we present the experimental evaluation of STRABO 2, with three main objectives. First, we evaluate the system as a whole, including the

ability to scale with respect to the cluster and dataset size. Second, we evaluate specific aspects of the system, and importantly the impact of the improvements and optimizations. Last, as Strabo 2 is the first distributed system able to handle geospatial queries on RDF graphs, we compare the performance of Strabo 2 with that of existing centralized GeoSPARQL processing systems in a single server environment.

4.1 Datasets and Queries

We have used the following datasets and queries: (i) *Scalability Workload.* This is a real-world dataset based on Open Street Maps from the Geographica 2 benchmark, which features a set of increasingly larger datasets (up tom 500M triples) and a queryset of 3 queries: 1 spatial selection (SC1) and 2 spatial joins (SC2, SC3). (ii) *PregenSynthetic Workload.* This is based on the synthetic workload of the Geographica 2 benchmark, but it has been modified so that it now uses a distributed Spark-based generator. (iii) *ExtremeEarth Workload.* This is a real-world dataset accompanied by 12 GeoSPARQL queries that were produced after analyzing end-user needs from the use cases of the project. The queries use a combination of spatial selections and spatial joins (including distance joins). The dataset has a size of 32 GB in N-Triples format.

4.2 Results in Distributed Environment

The experiments were carried out in a cluster provided by CREODIAS[11] consisting of 53 virtual processing cores and 164 GB of RAM. The Hopsworks platform v. 2.1.0 was used as the execution environment[12] providing access to an underlying Hadoop v. 3.2.0 installation with Spark v. 2.4.3 and Hive v. 3.0.0. Hopsworks is an open-source platform that provides an execution environment for distributed data science and data engineering tasks and extends Hadoop with an optimized distributed metadata architecture [16]. In the experiments, we set the number of shuffle partitions in Strabo 2 (Spark parameter `spark.sql.shuffle.partitions`) to be 5x the number of virtual cores in each setting.

As a first experiment, we determined the largest possible PregenSynthetic dataset that can be generated and imported in Strabo 2 using the aforementioned cluster, in order to stress the system given the available resources. As a result, we have generated the dataset for scaling factor 16384 and we have generated all the thematic tags. The size of the dataset is 156 GB in compressed Parquet format, which corresponds to an initial size of **1.16 TB** in N-Triples text format. We have also generated 72 queries with spatial selectivities of 1%, 0.1% and 0.01% and thematic selectivities corresponding to values 4096, 8192 and 16384 (a thematic selectivity with value M means that one every M entitites is annotated with a thematic tag that has the corresponding value). We have

[11] https://creodias.eu/
[12] https://www.hopsworks.ai/

used 22 executors with 6120 MB of memory and 2 virtual cores per executor. The data loader finished import in **4.5** hours. It is worth mentioning that, according to our experiments, this size of input datasets is much larger from what centralized geospatial RDF stores can ingest using a setup similar to the one described in Section 4.3. The total execution time for the 72 queries is **7711** seconds, which gives an average execution time of **107** seconds per query. The queries and the execution time for each query can be found in the supplemental material. We have also used 12 executors with 2 virtual cores with 4096 MB memory each, in order to execute the 12 queries from the ExtremeEarth dataset. The data loader in this case executed the import in **34** minutes. The average execution time was **122** seconds.

In further experiments, and in order to evaluate the specific aspects and improvements in query execution we have used the PregenSynthetic generator to generate a dataset with scale factor 1024 and queries for spatial selectivities of 1%, 0.1% and 0.01% and thematic selectivities corresponding to values 256, 512 and 1024. As before, we generate all thematic tags. In total we have generated 72 queries. We use 4 worker nodes, each one with 2 virtual cores and 4096 MB of memory. First, in order to evaluate the impact of the hybrid translation with persistent spatial index and partitioning, as described in Section 3.2, we have executed the 18 queries that contain a spatial selection with and without the spatial index. The total execution time when using the persistent spatial index and partitioning, drops from **111 seconds to 54 seconds**, leading to a reduction in execution time of more than 50%. The exact execution times are included in the supplemental material.

We also executed the spatial join queries using the default translation, in order to compare it with the optimized translation that pushes the thematic processing before spatial joins. In all cases the optimized translation was much faster, in some cases, especially for queries with few results, more than **10x**. The reason for that is that the Spark catalyst optimizer, when it takes as input the default translation query, it chooses to partition and index all the geometries on the geometry linking tables for the left side of the spatial join. We have also executed queries with thematic selectivity equal to 1. In this case the thematic processing does not filter out any values. In this extreme scenario, the default translation performs similarly with the optimized one.

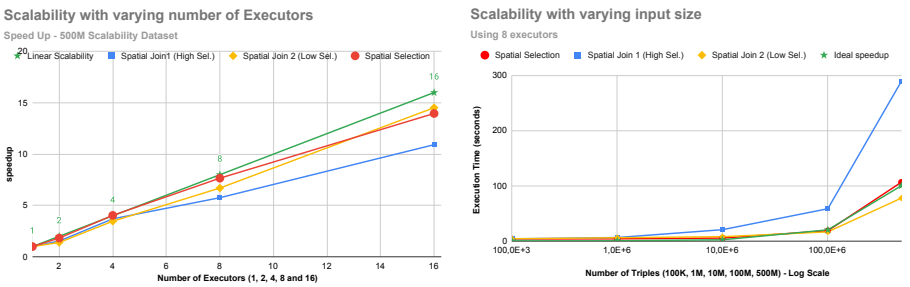

Fig. 3: Scalability of STRABO 2 with varying number of executors (left) and varying dataset size (right)

Regarding the ability of the system to scale in the distributed setting, we used the Scalability Workload and the results are presented in Figure 3. In the left side of the picture we have used the 500M dataset and executed the three queries with 1, 2, 4, 8 and 16 executors. In the right side we use 8 executors with the datasets from 100K to 500M triples. In both cases each executor had two cores and 6600 MB of memory. In the plots we also show the ideal speedup/linear scalability. In both experiments STRABO 2 exhibits very good behavior for both spatial selection and spatial join queries as in both scenarios, the spatial selection and the low selectivity join exhibit scaling very close to linear. The most difficult query to scale is the high selectivity join where large intermediate results needs to be saved, but even in this case the improvement as we add more executors is substantial.

4.3 Results in Centralized Environment

In order to perform a comparison between centralized RDF stores and STRABO 2, we selected the two most competitive systems from the Geographica 2 benchmark, namely Strabon and GraphDB. For Strabon we used v3.3.2-SNAPSHOT and for GraphDB v9.10.3. The test server was a Dell Inc PowerEdge R820 with 128 GB, with an Intel(R) Xeon(R) CPU E5-4603 v2 @ 2.20GHz with 32 execution threads, running Ubuntu 18.04.6 LTS. The system also features a PostgreSQL v12.10 installation as it is required by Strabon and was appropriately tuned. For disk-based centralized systems we report both warm cache and cold cache times. For GraphDB the `Preload` loading tool was used for all repositories. For STRABO 2 we used Spark 2.4.5 and Hive 2.3.6. We also used all available processing threads and we set 116 GB of memory available to Spark, although even for the 500M dataset half of this amount was enough.

In this set of experiments we have used the Scalability Workload and the PregenSynthetic workload. For the latter, four increasingly bigger datasets were generated, with scaling factor N in {512, 768, 1024, 2048}. For each dataset the corresponding query set had two thematic tags to help achieve the least and maximum thematic selectivity and the spatial selectivity list was fixed to (100%, 10%, 1%). The execution results for the scalability workload are shown in Figure 4. From the two centralized systems, Strabon exhibits better performance. In comparison with Strabon, STRABO 2 performs worse for the small datasets, but once the datasets size is increased, especially for the 100M and 500M, STRABO 2 in most cases outperforms the centralized solutions. An important point is that both centralized systems scale poorly when we increase the dataset size beyond 10M triples. As an example, even in the warm cache setting for spatial join 1, where Strabon performs faster than STRABO 2 with execution time of 133.29 seconds, we have an increase of 10x in execution time from the 100M case, whereas the corresponding increase for STRABO 2 is 6x. Due to space limitation we omit the full results for the PregenSynthetic workload, which are available at the supplemental material, but the systems exhibit similar behaviour. Especially for the dataset with scaling factor of 2048, from the total 28 queries, GraphDB had **12 timeouts** and an average time of **82** and **45** seconds (cold

and warm cache) for the 16 succeeded queries, whereas Strabon and STRABO 2 had no timeouts with average execution time of **134** and **105** seconds (cold and warm cache) from Strabon and only **28** seconds for STRABO 2.

		GraphDB		Strabon		Strabo 2
Dataset	Query	Cold (sec)	Warm (sec)	Cold (sec)	Warm (sec)	
100K	Spatial Selection	2.44	0.93	1.54	0.39	0.78
1M		10.60	7.62	5.84	3.83	1.08
10M		68.07	55.32	28.10	22.11	2.78
100M		553.37	556.03	246.82	165.70	15.54
500M		2,876.22	2,895.70	983.50	753.41	99.46
100K	Spatial Join 1 (High Selectivity)	52.39	53.58	0.67	0.08	2.35
1M		454.36	440.52	1.64	0.45	5.21
10M		3,941.93	3,972.95	7.26	2.34	12.54
100M		> 2h	-	50.30	13.75	28.44
500M		> 2h	-	575.94	133.29	172.80
100K	Spatial Join 2 (Low Selectivity)	21.41	20.45	0.59	0.08	2.15
1M		20.26	19.17	1.70	0.48	4.47
10M		87.21	84.80	10.67	2.57	6.54
100M		312.70	311.64	95.60	22.63	14.22
500M		> 2h	-	962.29	457.16	75.12

Fig. 4: Execution Results for Scalability Workload

5 Conclusions and Future Work

We presented STRABO 2, the first distributed geosaptial RDF store, able to handle massive datasets beyond the capabilities of centralized systems. Through experimental evaluation, we showed that STRABO 2 is faster even in a single server environment, once the dataset size reaches several GBs, by taking advantage the parallel multi-threaded execution carried out by Spark. For future work we plan to cover the GeoSPARQL RCC8 and Egenhofer topological relations and the Query Rewrite Extension.

Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under GA No 825258 (ExtremeEarth), GA No 101016798 (AI4Copernicus), EU Horizon Europe GA No 101070122 (STELAR), and from the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "First Call for H.F.R.I. Research Projects to support Faculty members and Researchers and the procurement of high-cost research equipment grant" (Project Number: HFRI-FM17-2351 GeoQA)

Supplemental Material Statement: STRABO 2 source code and exprimental results are publicly available[13]. Scalability and Synthetic workloads are available at the Geographica 2 website. ExtremeEarth datasets and queries can be found at the GitHub repository of the project[14].

[13] https://github.com/db-ee/Strabo-2
[14] https://github.com/ExtremeEarth-Project

References

1. Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., Saltz, J.: Hadoop gis: a high performance spatial data warehousing system over mapreduce. Proceedings of the VLDB Endowment **6**(11), 1009–1020 (2013)
2. Arrascue Ayala, V.A., Koleva, P., Alzogbi, A., Cossu, M., Färber, M., Philipp, P., Schievelbein, G., Taxidou, I., Lausen, G.: Relational schemata for distributed sparql query processing. In: Proceedings of the International Workshop on Semantic Big Data. pp. 1–6 (2019)
3. Battle, R., Kolas, D.: Enabling the geospatial Semantic Web with Parliament and GeoSPARQL. Semantic Web **3**(4), 355–370 (2012)
4. Bereta, K., Koubarakis, M.: Ontop of geospatial databases. In: International Semantic Web Conference. pp. 37–52. Springer (2016)
5. Bereta, K., Xiao, G., Koubarakis, M.: Ontop-spatial: Ontop of geospatial databases. Journal of Web Semantics **58**, 100514 (2019)
6. 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)
7. Cossu, M., Färber, M., Lausen, G.: PRoST: distributed execution of SPARQL queries using mixed partitioning strategies. arXiv preprint arXiv:1802.05898 (2018)
8. Eldawy, A., Mokbel, M.F.: The era of big spatial data: a survey. Information and Media Technologies **10**(2), 305–316 (2015)
9. Eldawy, A., Mokbel, M.F.: SpatialHadoop: A MapReduce framework for spatial data. In: 2015 IEEE 31st international conference on Data Engineering. pp. 1352–1363. IEEE (2015)
10. Eldawy, A., Mokbel, M.F.: The era of big spatial data. In: Proceedings of the VLDB Endowment (2017)
11. Graux, D., Jachiet, L., Geneves, P., Layaïda, N.: SPARQLGX: Efficient distributed evaluation of SPARQL with apache spark. In: International Semantic Web Conference. pp. 80–87. Springer (2016)
12. Hagedorn, S., Götze, P., Sattler, K.U.: Big spatial data processing frameworks: Feature and performance evaluation. In: EDBT. pp. 490–493 (2017)
13. Hagedorn, S., Räth, T.: Efficient spatio-temporal event processing with STARK. In: Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017. pp. 570–573 (2017). https://doi.org/10.5441/002/edbt.2017.72, https://doi.org/10.5441/002/edbt.2017.72
14. Ioannidis, T., Garbis, G., Kyzirakos, K., Bereta, K., Koubarakis, M.: Evaluating geospatial RDF stores using the benchmark Geographica 2. arXiv preprint arXiv:1906.01933 (2019)
15. Ioannidis, T., Garbis, G., Kyzirakos, K., Bereta, K., Koubarakis, M.: Evaluating geospatial RDF stores using the benchmark geographica 2. Journal on Data Semantics pp. 1–40 (2021)
16. Ismail, M., Gebremeskel, E., Kakantousis, T., Berthou, G., Dowling, J.: Hopsworks: Improving user experience and development on hadoop with scalable, strongly consistent metadata. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). pp. 2525–2528. IEEE (2017)
17. Kaoudi, Z., Manolescu, I.: RDF in the clouds: a survey. VLDB J. **24**(1), 67–91 (2015)

18. Kyzirakos, K., Alvanaki, F., Kersten, M.: In memory processing of massive point clouds for multi-core systems. In: Proceedings of the 12th International Workshop on Data Management on New Hardware. pp. 1–10 (2016)
19. Kyzirakos, K., Karpathiotakis, M., Koubarakis, M.: Strabon: A Semantic Geospatial DBMS. In: The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I. pp. 295–311 (2012)
20. Lu, J., Güting, R.H.: Parallel secondo: boosting database engines with hadoop. In: 2012 IEEE 18th International Conference on Parallel and Distributed Systems. pp. 738–743. IEEE (2012)
21. Nikitopoulos, P., Vlachou, A., Doulkeridis, C., Vouros, G.A.: Parallel and scalable processing of spatio-temporal rdf queries using spark. GeoInformatica **25**(4), 623–653 (2021)
22. Pandey, V., Kipf, A., Neumann, T., Kemper, A.: How good are modern spatial analytics systems? Proceedings of the VLDB Endowment **11**(11), 1661–1673 (2018)
23. Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF querying with SPARQL on spark. PVLDB **9**(10), 804–815 (2016)
24. Sriharsha, R.: Magellan: Geospatial Analytics Using Spark. https://github.com/harsha2010/magellan, [Online; accessed 04-November-2019]
25. You, S., Zhang, J., Gruenwald, L.: Large-scale spatial join query processing in cloud. In: 2015 31st IEEE International Conference on Data Engineering Workshops. pp. 34–41. IEEE (2015)
26. Yu, J., Wu, J., Sarwat, M.: 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. p. 70. ACM (2015)
27. Yu, J., Zhang, Z., Sarwat, M.: Spatial data management in apache spark: the GeoSpark perspective and beyond. GeoInformatica **23**(1), 37–78 (2019)