Multidimensional Enrichment of Spatial RDF Data for SOLAP

Gür, Nurefsan; Pedersen, Torben Bach; Hose, Katja; Midtgaard, Mikael

Published in:
Semantic Web – Interoperability, Usability, Applicability

DOI (link to publication from Publisher):
10.3233/SW-210423

Publication date:
2022

Document Version
Publisher's PDF, also known as Version of record

Link to publication from Aalborg University

Citation for published version (APA):
Gür, N., Pedersen, T. B., Hose, K., & Midtgaard, M. (2022). Multidimensional Enrichment of Spatial RDF Data for SOLAP. Semantic Web – Interoperability, Usability, Applicability, *13*(1), 5-39. Advance online publication. https://doi.org/10.3233/SW-210423

Semantic Web 13 (2022) 5–39
DOI 10.3233/SW-210423
IOS Press

Multidimensional enrichment of spatial RDF data for SOLAP

Nurefşan Gür [*], Torben Bach Pedersen, Katja Hose and Mikael Midtgaard
Center for Data Intensive Systems, Aalborg University, Selma Lagerlöfsvej 300, DK-9220 Aalborg Ø, Denmark
E-mails: nurefsan@cs.aau.dk, tbp@cs.aau.dk, khose@cs.aau.dk, mikaelmidt@gmail.com

Editor: Boyan Brodaric, Geological Survey of Canada, Canada
Solicited reviews: Alberto Abelló, Universitat Politècnica de Catalunya, Spain; two anonymous reviewers

Abstract. Large volumes of spatial data and multidimensional data are being published on the Semantic Web, which has led to new opportunities for advanced analysis, such as Spatial Online Analytical Processing (SOLAP). The RDF Data Cube (QB) and QB4OLAP vocabularies have been widely used for annotating and publishing statistical and multidimensional RDF data. Although such statistical data sets might have spatial information, such as coordinates, the lack of spatial semantics and spatial multidimensional concepts in QB4OLAP and QB prevents users from employing SOLAP queries over spatial data using SPARQL. The QB4SOLAP vocabulary, on the other hand, fully supports annotating spatial and multidimensional data on the Semantic Web and enables users to query endpoints with SOLAP operators in SPARQL. To bridge the gap between QB/QB4OLAP and QB4SOLAP, we propose an RDF2SOLAP enrichment model that automatically annotates spatial multidimensional concepts with QB4SOLAP and in doing so enables SOLAP on existing QB and QB4OLAP data on the Semantic Web. Furthermore, we present and evaluate a wide range of enrichment algorithms and apply them on a non-trivial real-world use case involving governmental open data with complex geometry types.

Keywords: Spatial data warehouses, SOLAP, spatial RDF data cubes, geospatial Semantic Web

1. Introduction

Data warehouses (DWs) and Online Analytical Processing (OLAP) tools and queries are widely used for interactive data analysis. DWs have multidimensional (MD) models and store large volumes of data. MD models locate data in an n-dimensional space and are usually referred to as *data cubes*. The *cells* of a cube represent the topic of the analysis and associate observation *facts* with (numerical) *measures* that can be aggregated. Spatial data cubes can also contain *spatial* measures, which can be aggregated with spatial functions. For example, a data cube for *farms* might have a numerical measure 'number of animals' as well as the 'farm's coordinates' as spatial measure. Facts are linked to *dimensions*, which provide contextual information, e.g., farm production, farm location, and farm livestock. Dimensions are organized into *hierarchies* with *levels*, e.g., parish of the farm or herd type of livestock, which allow users to analyze and aggregate measures at different levels of detail. Levels have a set of *attributes* describing the characteristics of the level members.

In traditional DWs, the location dimension is generally used as a conventional (non-spatial) dimension with alphanumeric data and thus provided with only a nominal reference to places and areas, e.g., parish name. This does not allow for applying spatial operations or truly deriving topological relations between hierarchy levels based on geometric information such as coordinates, which are essential for enabling spatial OLAP (SOLAP) analysis.

By including the geometric information of locations in MD models, we can significantly improve the analysis process (e.g., proximity analysis of locations) with

[*]Corresponding author. E-mail: nurefsan@cs.aau.dk.

additional perspectives by revealing dynamic spatial hierarchy levels and new spatial level members in SOLAP operations (details and examples in [14,15]). In addition, by using geometric attributes of level members, topological relations between the levels, and levels and facts can be specified implicitly. Such topological relations are essential to correctly aggregate measures between levels with many-to-many (N:M) cardinality relations, for instance.

The Semantic Web (SW) has evolved, from prominently focusing on data publishing to also supporting complex queries, such as interactive analytical queries. Simultaneously, the data available on the SW has evolved from being simple, mostly alphanumerical data, to include complex data types, such as geospatial data. There are many examples of governmental and statistical Linked Open Data (LOD) sets with geographical attributes. However, such datasets are typically not modeled with multidimensional concepts. Thus, they cannot be queried with interactive analytical queries (OLAP). Although in recent years several platforms and tools for Business Intelligence (BI) and data warehouses have emerged [50], there is still a lack of common standards to model and publish (geo)semantic cubes on the SW [15].

More and more statistical datasets using the RDF Data Cube Vocabulary (QB) [48], the current W3C standard, are published on the SW. These datasets have observations and measures, which are well-suited for analytical queries. However, QB lacks the underlying structural metadata for multidimensional models and OLAP operations (Section 6). Well-defined structural metadata is required to translate OLAP queries into SPARQL 1.1 [14,46]. QB4ST [3] is a recent attempt to define extensions for spatio-temporal components to QB. However, it inherits the limitations of multidimensional modeling from QB.

To address the MD modeling challenges of the QB vocabulary, QB4OLAP [7] has been proposed, which reuses QB definitions by adding the required MD schema semantics. A significant number of data sets have already been published using the QB vocabulary. QB4OLAP descriptions of a QB data cube can be generated semi-automatically by adding the necessary MD semantics (e.g., the hierarchical structure of the dimensions) and the corresponding instances to populate the dimension levels. However, existing QB4OLAP annotation techniques [44] only cover *non-spatial* MD data cube concepts and its operations. Even though such statistical data sets have spatial information, not annotating the spatial MD concepts (e.g., spatial hierarchy levels such as administrative regions) hinders querying the data with interesting spatial OLAP operations. To emerge this need the QB4SOLAP vocabulary was proposed [13], which allows modeling the data cubes fully with both multidimensional and spatial concepts on the SW.

Problem motivation and definition Spatial OLAP (SOLAP) queries are currently not well supported by existing spatial RDF stores and endpoints. Instead, the user would have to a) download the (maybe very large) RDF data, b) map it to a relational schema (e.g., a snowflake schema), c) import it into a traditional spatial data warehouse, d) make all queries and analyses within the traditional spatial DW, and finally e) import and map any results and knowledge back into the original RDF store. This obviously is a slow, labor-intensive, and error-prone process, which furthermore completely locks out the vast majority of users without advanced programming skills.

Luckily, there already exist tools and vocabularies for (spatial) data warehouses on the SW: the QB4SOLAP vocabulary [13], for instance, allows publishing data with spatial multidimensional concepts on the SW and provides high-level SOLAP operators that can be translated into SPARQL [15]. Based on these, GeoSemOLAP [14] enables users to issue SOLAP queries on geo-semantic RDF data without detailed knowledge of SPARQL or RDF.

GeoSemOLAP, however, is restricted to RDF data sets that are *already* annotated with QB4SOLAP.

Thus, there is a *great unmet need for an automated approach to enrich and annotate geo-semantic RDF data from existing endpoints with QB4SOLAP metadata*. This is exactly what our proposed *RDF2SOLAP enrichment module* does (Fig. 1).

Since on-the-fly annotations would require the corresponding heavy spatial operations to be executed repeatedly for each new query, making response times

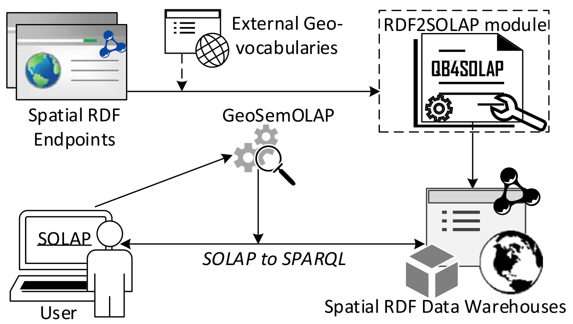

Fig. 1. Future vision of SOLAP on the SW.

too slow for interactive OLAP querying, we annotate the entire data set in a once-and-for-all fashion.

Contributions In summary, the main contributions of this paper are:

* An illustration of the need for QB4SOLAP, i.e., the need to enable fully-fledged data warehouse concepts for geo-semantic RDF data. We further introduce running examples from real-world governmental open data on environment and farming with complex geometry types.
* A detailed explanation and comparison of RDF data examples, which are depicted as graphs, and annotated both with QB4OLAP and QB4SOLAP vocabularies, then identifying the required spatial MD metadata and concepts (e.g., spatial hierarchies and topological relations) for SOLAP analysis based on the given comparison.
* Hierarchical enrichment algorithms for (1) detecting topological relations at hierarchy steps with direct links between the level members; and (2) discovering topological relations at hierarchy steps (which do not have direct links between the level members).
* Factual enrichment algorithms for fact-level relations between fact and level members.
* An automated way of re-defining a fact schema after factual enrichment, and association of spatial aggregate functions with spatial measures.
* General implementation of our approach for both hierarchical enrichment and factual enrichment processes.
* Evaluation of our approach in terms of accuracy and coverage in comparison to two standard environments (RDBMS and GIS tool).

Paper organization The remainder of this paper is organized as follows. Section 2 defines the preliminary concepts used throughout the paper with a running use case example. Section 3 presents the system architecture for the MD enrichment process. Section 4 defines the RDF2SOLAP enrichment algorithms with necessary helper functions and formalization of (spatial) RDF data. The Appendix presents the implementation details along with interesting examples and discusses the challenges and implemented solutions. Section 5 presents the qualitative and performance evaluation with comparison baselines. Finally, Section 6 discusses related work and Section 7 concludes the paper with an outlook to future work.

2. Preliminaries

In this section, we explain the preliminary concepts of spatial data warehouses and SOLAP (Section 2.1) and how to deploy them on the Semantic Web (Section 2.2) using the QB4SOLAP vocabulary.

2.1. Spatial data warehouses and SOLAP

Data cubes and spatially extended cube concepts Data warehouses (DW) are based on a multidimensional model that models data in an n-dimensional space – often referred to as a data cube. A cube *schema* defines the structure of a cube with MD concepts. The cells of the cube represent *(observation) facts* with a set of attributes called *measures*. Facts are linked to *dimensions*, which are the axes of an MD space and provide perspectives to analyze the data. Dimensions are organized into *hierarchies*, which allow users to aggregate measures at different granularities along the levels of a hierarchy. Hierarchies are composed of *levels*, which have a set of *attributes* describing the characteristics of the level members. Each *level member* is defined by its attributes and attribute values.

Cube members are MD concepts that are defined at the *instance* level and composed of level members, attributes of level members, *partial order* on level members, and fact members. A *hierarchy step* between levels (a child level and a parent level) defines a set of *roll-up* relations, where each relation relates a child level member to a parent level member. These roll-up relations define a partial order between level members with a *cardinality* relation. The cardinality (1:1, 1:N, N:1, N:M) describes the number of members in one level that can be related to a member in the other level for both child and parent levels.

Spatial data warehouses (SDW) extend a DW by storing geometries such as *point*, *line*, and *polygon* in the values of spatial measures and values of level attributes for spatial dimensions. The spatially extended MD schema of an SDW has spatial dimensions, spatial hierarchies, spatial levels [29], spatial hierarchy steps, and topological relations (in addition to cardinality relations) between spatial levels for each spatial hierarchy step [13]. Topological relations are Boolean spatial predicates that specify how two spatial objects are related to each other, e.g., *within, intersects, touches, crosses* and etc. [6]. Similar to conventional DWs, facts of an SDW can be associated with numeric measures, which are using aggregation functions such as `SUM`, `AVG`, etc. A fully extended spatial MD schema of an

SDW should also define spatial measures, which have geometries and spatial aggregate functions. Spatial aggregate functions aggregate two or more spatial objects and return a new spatial object. Union, Intersection, ConvexHull, and Minimum-BoundingRectangle (MBR) are example of spatial aggregate functions. For a detailed explanation of SDW concepts we refer the reader to [42].

OLAP and spatial OLAP operations DWs are commonly used to store large volumes of data for decision support with On-Line Analytical Processing (OLAP) operations. Spatial OLAP (SOLAP) integrates the features of OLAP tools and geographical information systems (GIS) [38]. SOLAP enables advanced analytical processing by taking the spatial information in the cube into account.

For example, a spatial data cube of livestock holdings in farms (referred to as *GeoFarmHerdState* in the rest of this paper) defines the farm location as a spatial measure, which is linked to the observation facts. In order to derive perspectives and relations on the state of the farms' livestock holdings (herds), spatial levels are defined: parishes and drainage areas. A sample set of the corresponding spatial data cube members are given in Fig. 2. The spatial MD concepts of the data cube are defined in the conceptual schema in Fig. 3, which depicts a simplified version of the GeoFarmHerdState spatial data cube without its non-spatial dimensions (see [12] for further details the GeoFarmHerdState cube). The cube has two spatial dimensions: *FarmDim* and *ParishDim*. The latter has a spatial hierarchy (*Geography*) with two spatial levels: *Parish* and *DrainageArea*. *FarmDim* on the other hand does not have a spatial hierarchy, despite its spatial (base) level: *Farm*.

The GeoFarmHerdState cube has spatial fact members for farms within a time frame and different kinds of measures, i.e., numeric measures: *NumberofAnimals* in the farm and *NitrogenReduction* potential of the farm land/soil, spatial measures: *FarmLocation* (Fig. 3).[1]

Fig. 2. GeoFarmHerdState – parish, farm, and drainage area instances.

To evaluate SOLAP operations, spatial levels such as *Parish* and *DrainageArea* are used to aggregate measures at different levels of detail. Due to the *polygon* geometry of the spatial level members, there are two different roll-up relations for the hierarchy step between the Parish and DrainageArea levels, where a parish can be completely contained *within* a drainage area or a parish and a drainage area can *intersect*.

For example, parish "Oue" is *within* drainage area "Mariager Inderfjord". Thus, all the farms that are *within* "Oue" are also *within* "Mariager Inderfjord". Whereas, parish "Astrup" *intersects* with drainage areas "Mariager Inderfjord" and "Langerak". Therefore, some farms that are *within* "Astrup" are *within* "Mariager Inderfjord", while the rest of the farms are *within* "Langerak". Figure 2 displays a sample set of Parish and DrainageArea level members.

The possible roll-up relations for the example above are depicted in Fig. 4 with black and red arrows representing the topological relations *within* and *intersects*. Blue arrows show the topological relation *contains*, which are drill-down (inverse operation of roll-up) relations from DrainageArea level to Farm level.

Topological relations between levels and facts can be implicitly specified through the geometry attributes of their instances (level members and fact members). The relations between spatial levels enable processing spatial roll-up and drill-down through range queries with spatial predicates [8]. In terms of cardinality, there is an N:M relationship between level members since a parish may intersect with more than one drainage area and vice versa. This induces the problem of computing measures incorrectly when a roll-up operation goes through an N:M relationship, which actually is the case between the Parish level and the DrainageArea level. For example, we would like to aggregate the measure NumberOfAnimals, from Parish level to the DrainageArea level with a roll-up query. In such a roll-up query, we might falsely aggregate the number of animals in farms that are contained

[1] Non-additive measures are also numeric measures, which are given in percentages or classified in numbers, therefore they cannot be meaningfully summarized by all aggregate functions i.e., `SUM`. However, depending on the semantics, other aggregate functions can be associated with them, e.g., `AVG` NitrogenReduction potential, `MAX` NitrateClass.

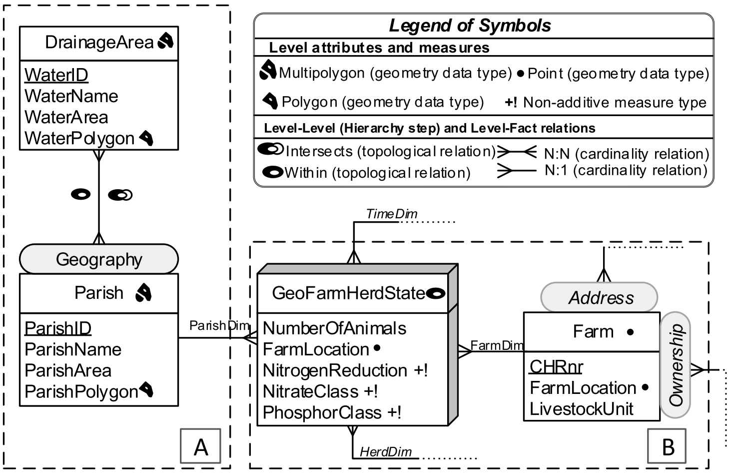

Fig. 3. GeoFarmHerdState – conceptual MD schema of livestock holdings data (spatial concepts).

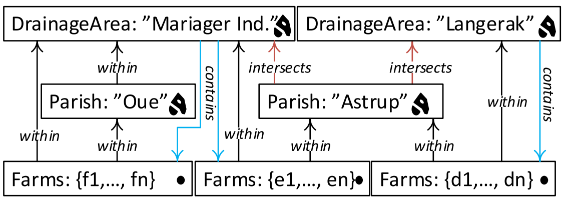

Fig. 4. Hierarchy example for SOLAP.

within the parish, but not contained *within* the drainage area, since the parish *intersects* with another drainage area. In order to refine such an analysis, SOLAP operations are required, where a (spatial) drill-down should be applied to the lowest granularity – from Parish level members to GeoFarmHerdState fact members, and then a spatial roll-up (with within predicate) can be applied from fact members (Farm instances) to DrainageArea level members. This would prevent falsely aggregating the number of animals from the farms that are (spatially) disjoint to the corresponding drainage area.

2.2. *QB4SOLAP: Spatial RDF data cube vocabulary for SOLAP operations*

There is an increasing amount of Linked Open Data (LOD) on the Semantic Web containing spatial information and numerical (statistical) data. This led to new opportunities for OLAP over spatial data using semantic web technologies and standards. Datasets on the SW use a standardized format: RDF (Resource Description Framework).[2]

In order to enable SOLAP operations on the Semantic Web, a comprehensive vocabulary is needed, i.e., annotation of the spatial hierarchy steps with topological relations. QB4SOLAP [15] is a vocabulary that allows the definition of *cube schemas* and *cube instances* in RDF. The QB4SOLAP vocabulary is an extension of QB4OLAP [7] capturing the semantics of spatial MD concepts (i.e., spatial hierarchy steps) that are essential for SOLAP operations. The QB4SOLAP vocabulary V1.3 is available on our project website[3] as well as via a persistent URL.[4]

A comprehensive foundation of spatial data warehouses on the Semantic Web can be found in [15], which includes detailed definitions with semantics of spatial MD concepts both at the schema level and instance level using QB4SOLAP.

In the following, we depict an example of a hierarchy step from `gfs:Parish` child level to `gfs:drainageArea` parent level (Fig. 5). In the figure, we prefix the schema elements (attributes, levels, etc.) of the (GeoFarmHerdState) cube with `gfs:` and instance data from the cube with `gfsi:`. The left-center part of Fig. 5 shows the hierarchy structure `_:hs`, between `gfs:parish` and `gfs:drainageArea` levels at the schema level with the QB4OLAP vocabulary. QB4OLAP objects, classes, and properties are prefixed with `qb4o:`. The levels (`gfs:parish` and

[2] https://www.w3.org/TR/rdf11-primer/
[3] https://extbi.cs.aau.dk/QB4SOLAP
[4] https://w3id.org/qb4solap#

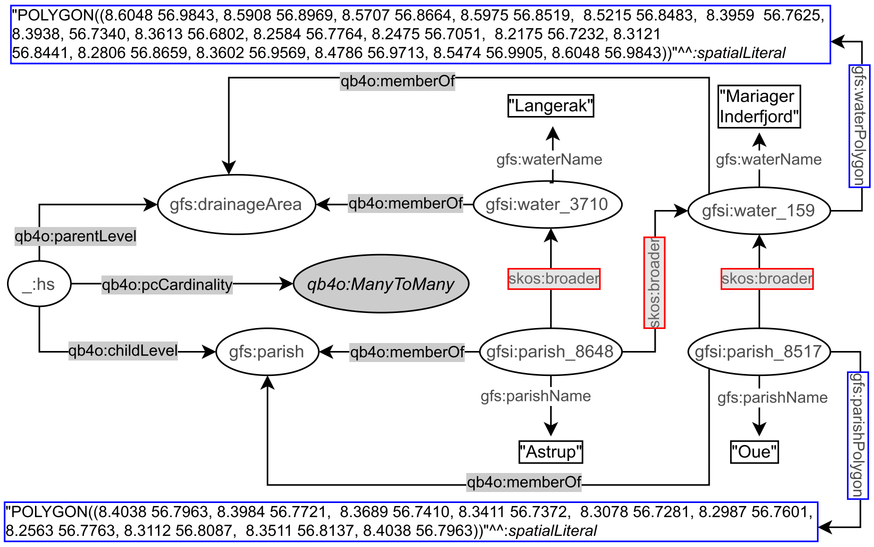

Fig. 5. Hierarchy steps in QB4OLAP before multidimensional enrichment.

`gfs:drainageArea`) are linked to the instances of level members (e.g., `gfsi:parish_8648`, `gfsi:water_3710` and etc.) by `qb4o:memberOf` property. The polygon geometry attributes are highlighted in blue boxes, on the top and the bottom of the figure. The coordinates recorded in the geometry attributes can be used to derive the topological relation between the level members by applying spatial boolean predicates (e.g., *insersects?*, *within?*) on the *polygon* geometries of the parish and drainage area level members.

However, QB4OLAP does not support annotating the topological relations that might exist between the level members at a hierarchy step. QB4OLAP uses only `skos:broader` property from SKOS (Simple Knowledge Organization System) [30] semantic relations for capturing the roll-up relations at hierarchy steps. The roll-up relations with `skos:broader` property are highlighted in red boxes in Fig. 5. The `skos:broader` property does not describe the nature of the roll-up relation with topological relations for spatial hierarchies. Therefore, QB4OLAP cannot capture the topological relations in a hierarchy step from Parish level to DrainageArea level or between these levels' members.

On the other hand, QB4SOLAP can define topological relations both at the schema level and the instance level. In Fig. 6, we prefix QB4SOLAP objects, classes, and properties with `qb4so:` and highlight them in green lines. The left-center part of the figure shows the spatial hierarchy structure `:_shs`, which has a QB4SOLAP property `qb4so:pcTopoRel` with two QB4SOLAP class instances `qb4so:Within` and `qb4so:Intersects`. This means that when we compare the geometry attributes of parish level members and drainage area level members, we discover two different topological relations (*within* and *intersects*) for all the (spatial) hierarchy steps between the parish and drainage area levels. And these relations are annotated at the schema level on the left-center part of Fig. 6.

Similarly, gfs:parish and `gfs:drainageArea` levels are linked to the instances of level members (e.g., `gfsi:parish_8648`) by `qb4o:memberOf` property. The explicit topological relations between each level member along a spatial hierarchy step are depicted in the figure with `qb4so:intersects` or `qb4so:within` predicates, which are highlighted in green boxes (e.g., `gfsi:parish_8648` *intersects* with `gfsi:water_159` and `gfsi:water_3170` etc.).

In conclusion, QB4SOLAP enables SOLAP operations by defining the semantics of spatial MD concepts both at the schema level and instance level. These semantics are essential for SOLAP operations, and they are defined as extensions to the QB4OLAP vocabulary.

3. System architecture

The importance of SOLAP to get accurate results in operations over spatial data warehouses is explained in

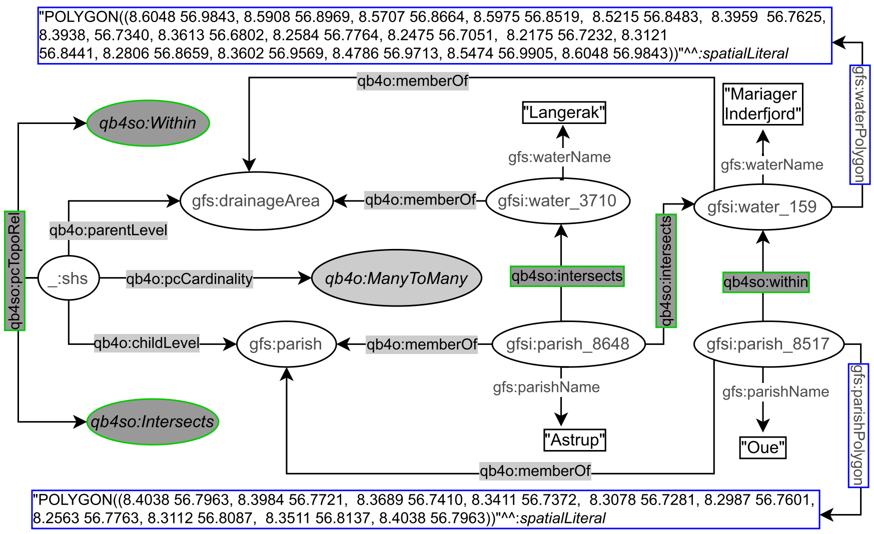

Fig. 6. Spatial hierarchy steps in QB4SOLAP after multidimensional enrichment.

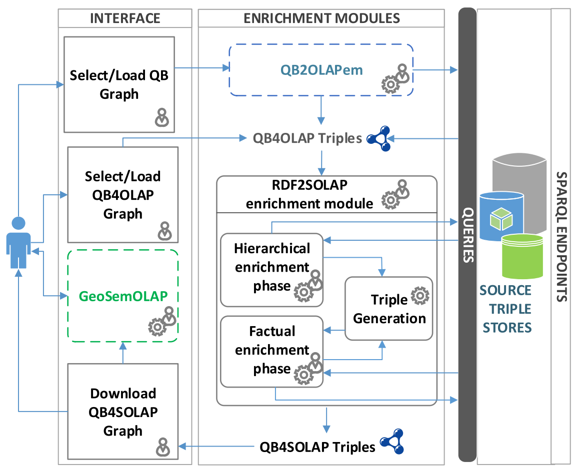

Fig. 7. Multidimensional enrichment process.

Section 2.1. However, the RDF data cubes (with spatial attributes) on the Semantic Web are not always annotated with vocabularies that allow users to formulate SOLAP queries. In this section we present an overview of the MD enrichment flow from RDF QB to QB4OLAP data cubes and QB4OLAP to QB4SOLAP data cubes. Thus, users can query the RDF data cubes with SOLAP queries.

A multidimensional enrichment process flow is illustrated in Fig. 7 with three main architectural layers: Interface, Enrichment Modules, and SPARQL Endpoints. The architectural layers in the figure are denoted in horizontal rectangles. The first layer facilitates user interaction with the enrichment modules (i.e., QB2OLAPem) and third party tools (i.e., GeoSemOLAP). In each layer, processes are given in right angle boxes, modules and tools are given in rounded corner boxes. Third party tools and modules are annotated in dashed lines. Arrows in the figure represents the interaction between processes and the modules.

Our main contribution in this paper is the RDF2SOLAP enrichment module, which is the core of the second layer. The RDF2SOLAP enrichment module operates on QB4OLAP triples that either already exist in the original data or have been generated by the QB2OLAPem enrichment module [44]. QB2OLAPem allows users to enrich an RDF QB dataset with QB4OLAP concepts and returns a graph of QB4OLAP triples.

The internal process flow of the RDF2SOLAP enrichment module consists of three phases: hierarchical enrichment, factual enrichment, and triple generation. The hierarchical and factual enrichment phases iteratively perform the enrichment algorithms explained in Section 4. Hierarchical enrichment phase and factual enrichment phase can run independently from each other in parallel. Factual enrichment phase additionally can suggest an enriched fact schema definition, which depends on the spatial relations found at the instance level enrichment for factual and hiearchical enrichment phases. Both of these enrichment phases allow interaction with external SPARQL endpoints to enhance the enrichment process via potential spatial and multidimensional concepts that could be retrieved externally. The third phase is the triple generation, which creates QB4SOLAP triples that can be used in third party tools such as GeoSemOLAP. GeoSemOLAP allows users without knowledge of RDF and SPARQL to query with SOLAP operations by interactively formulating the queries using a GUI with interactive maps [14].

The third layer (SPARQL endpoints) allows interaction between user and SPARQL endpoint for retrieving QB or QB4OLAP graphs as well as interaction between system and SPARQL endpoints, where the RDF2SOLAP enrichment module queries external triple stores for hierarchical enrichment and factual enrichment.

RDF2SOLAP is implemented in Javascript on the Node.js platform using the N3.js library for parsing the RDF triples in Javascript and the Turfjs library for spatial analysis.[5]

4. RDF2SOLAP enrichment algorithms

This section presents the core algorithms of our RDF2SOLAP enrichment module. Our MD enrichment approach builds upon QB4OLAP triples that either already exist in the original data or have been generated by the QB2OLAPem enrichment module [44] as depicted in Fig. 7. QB4OLAP defines only the non-spatial multidimensional semantics of RDF data, whereas QB4SOLAP enriches the MD semantics of RDF data with spatial concepts (formalizations and further details can be found in [15]). Nevertheless, in the following we briefly introduce basic notations.

The basic construct of RDF is a triple $t = (s, p, o)$ consisting of three components; s is the subject, p is the predicate, and o is the object. RDF triples are defined over $\mathcal{T} = (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$, where $\mathcal{I}$ is the set of *IRIs* (Internationalized Resource Identifiers), $\mathcal{B}$ is the set of *blank nodes*, and $\mathcal{L}$ is the set of *literals*. An object value can be a *literal* (i.e., string, spatial literal,[6] integer etc.). Subjects and objects can be represented by a *blank node* for anonymous resources. Predicates are always represented by IRIs. A set of RDF triples is referred to as an RDF *graph* $\mathcal{G}$. We use superscript notation to represent the type of a graph: schema graph $\mathcal{G}^S$ and instance graph $\mathcal{G}^I$. An instance graph has entities from a use-case dataset as a set of RDF triples. The schema graph describes the structure (schema) of the dataset recorded in the instance graph.

[5] **N3.js:** https://github.com/rdfjs/N3.js **Turfjs:** http://turfjs.org/.

[6] Spatial literals are represented as $\mathcal{L}_S$.

```
## Spatial hierarchies in QB4SOLAP with topological relations##
1 _:_shs rdf:type qb4o:HierarchyStep ; qb4o:inHierarchy gfs:geography ;
2     qb4o:childLevel gfs:parish ; qb4o:parentLevel gfs:drainageArea ;
3     qb4o:pcCardinality qb4o:ManyToMany ;
4     qb4so:pcTopoRel qb4so:Within , qb4so:Intersects .
```

Listing 1. Spatial hierarchy structure in QB4SOLAP

We use subscript notation to represent the MD concepts in RDF terms as a graph. For example, $\mathcal{G}^I_{A(lm)}$ is the RDF instance graph for attributes of level members – in the use case example this graph corresponds to the set of triples in Listing 2, Lines 3–6 or Lines 9–13 and Lines 17–22. $\mathcal{G}^S_{\mathrm{HS}(h)}$ is the RDF schema graph for hierarchy steps – in the use case example this graph corresponds to the set of triples in Listing 1.

We define function $\mathrm{id}(x) : \mathcal{G} \rightarrow \mathcal{I}$, which given an MD element x returns its identifier $\mathcal{I}$ from graph $\mathcal{G}$. Similarly, we use superscript notation to indicate the type of the identifier from the schema graph ($\mathcal{G}^S$) and instance graph ($\mathcal{G}^I$), e.g., $\mathrm{id}^S(a)$ for a schema identifier of a level (`gfs:parish` in Listing 2, Line 2 or in Listing 1, Line 2) and $\mathrm{id}^I(lm)$ for an instance identifier of a level member (`gfsi:parish_8648` in Listing 2, Line 1 or Line 8).

The MD enrichment process in RDF2SOLAP runs in two phases (*hierarchical enrichment phase* and *factual enrichment phase*), which are explained in the following.

4.1. Hierarchical enrichment phase

The hierarchical enrichment phase is built around spatial levels and their level members forming the spatial hierarchy of a dimension. Thus, by identifying the spatial relations between spatial levels and their level members, we can find the spatial hierarchy steps and the possible topological relations for these hierarchy steps.

Each spatial hierarchy corresponds to a path of roll-up relationships between the child level and parent level: each of these roll-up relationships corresponds to a *spatial hierarchy step* (Section 2.1). An example of a (spatial) hierarchy with QB4SOLAP is given in Listing 1. Line 4 extends the QB4OLAP schema definitions by enriching the hierarchy step with the possibility to annotate the spatial hierarchy steps with topological relations (see Section 2 for details and Section 2.2 for examples).

Listing 2 shows the GeoFarmHerdState spatial level members from Parish and Drainage Area levels. Lines 1–7 (Listing 2) represent the QB4OLAP annotation of a child level member from Parish level

```
## Parish (child) Level member before hierarchical enrichment##
1  gfsi:parish_8648 rdf:type qb4o:LevelMember ;
2      qb4o:memberOf gfs:parish ;
3      gfs:parishID 8648 ; gfs:parishName "Astrup" ;
4      gfs:parishArea 47,969 ; gfs:parishPolygon "POLYGON((8.438 56.796,
5      8.3984 56.7721, 8.3689 56.7410, 8.3411 56.7372, 8.3078 56.7281,
6      8.3112 56.8087, 8.3511 56.8137, 8.438 56.796))"^^geo:spatialLiteral ;
7      skos:broader gfsi:water_3710 , gfsi:water_159 .
## Parish (child) Level member after hierarchical enrichment##
8  gfsi:parish_8648 rdf:type qb4o:LevelMember ;
9      qb4o:memberOf gfs:parish ;
10     gfs:parishID 8648 ; gfs:parishName "Astrup" ;
11     gfs:parishArea 47,969 ; gfs:parishPolygon "POLYGON((8.438 56.796,
12     8.3984 56.7721, 8.3689 56.7410, 8.3411 56.7372, 8.3078 56.7281,
13     8.3112 56.8087, 8.3511 56.8137, 8.438 56.796))"^^geo:spatialLiteral ;
14     qb4so:intersects gfsi:water_3710 , gfsi:water_159 .
## DrainageArea (parent) Level member##
15 gfsi:water_159 rdf:type qb4o:LevelMember ;
16     qb4o:memberOf gfs:drainageArea ;
17     gfs:waterName "Mariager Inderfjord" ; gfs:waterArea 267,477 ;
18     gfs:waterPolygon "POLYGON((8.6048 56.9843, 8.5908 56.8969,
19     8.5707 56.8664, 8.5975 56.8519, 8.5215 56.8483,
20     8.3959 56.7625, 8.3938, 56.7340, 8.3613 56.6802,
21     8.2584 56.7764, 8.2475 56.7051, 8.2175 56.7232,
22     8.5474 56.9905, 8.6048 56.9843))"^^geo:spatialLiteral .
```

Listing 2. GeoFarmHerdState level members, attributes, and *spatial* roll-up relations

before multidimensional enrichment (with `skos:broader`), which is depicted in Fig. 5. Lines 8–14 represent the QB4SOLAP annotation of the same Parish level member after the multidimensional enrichment with topological relations (depicted in Fig. 6). Lines 15–22 represent the annotation of a parent level member from the Drainage area level, which remains the same before and after multidimensional enrichment since the hierarchy steps are defined with bottom-up relationships from child level to parent level and the roll-up relations and thus also the topological relations are annotated at the child level members of the hierarchy step.

We exploit QB4OLAP semantics, such as *non-spatial* hierarchy steps and levels as a starting point to find the *spatial* hierarchy steps. We distinguish two cases:

Case 1: Finding *explicit* spatial hierarchy steps for QB4OLAP levels, with `skos:broader` roll-up relations between their child-parent level members by *detecting spatial hierarchy steps* in Section 4.1.2. For this case we assume that level members have direct `skos:broader` relations as depicted in Fig. 5 and Listing 2, Line 7 with `skos:broader` property.

Case 2: Finding *implicit* spatial hierarchy steps from QB4OLAP levels without direct roll-up relations through the `skos:broader` property. In this case, we assume that the level members are only defined by the `qb4o:memberOf` property as shown in Listing 2, (Line 2) but *do not* have the `skos:broader` roll-up relation as given in Line 7. In this case, it is still possible to *discover spatial hierarchy steps* by finding spatial (topological) relations between level members through their attributes as explained in Section 4.1.3.

Algorithm 1: getSpatialValues($\mathcal{G}^I_{A(lm)}$): $V_{s(a)}$

Input: $\mathcal{G}^I_{A(lm)}$
Output: $V_{s(a)}$
1 **begin**
2 $V_{s(a)} = \emptyset$; /*initialize output set as empty set*/
3 **foreach** $(\mathrm{id}^I(lm)\ \mathrm{id}^S(a_i)\ v_{a_i}) \in \mathcal{G}^I_{A(lm)}$ **do**
4 **if** v_{a_i} *is a* `geo:spatialLiteral` **then**
5 $V_{s(a)} \cup = \{v_{a_i}\}$;
6 **return** $V_{s(a)}$

4.1.1. Spatial helper functions

To address the cases explained above, we need two spatial helper functions; for retrieving spatial attribute values (Algorithm 1, `getSpatialValues`), and for relating spatial attributes (Algorithm 2, `relateSpatialValues`).

Algorithm 1 (`getSpatialValues`) The first helper function gets an input graph of attributes of level members $\mathcal{G}^I_{A(lm)}$ and returns a set of spatial attribute values $V_{s(a)}$. For example, the function could receive Lines 3–6 from Listing 2 as input. In the algorithm, Lines 3 and 4 check the values v_{a_i} of each attribute $\mathrm{id}^S(a_i)$ (e.g., `gfs:parishName`, `gfs:ParishArea`, etc.) If the value is a type of `geo:SpatialLiteral` (e.g., the `POLYGON` geometry value linked to the `gfs:parishPolygon` attribute), then the value is incrementally added to the output set $V_{s(a)}$[7] in Line 5.

Algorithm 2: (`relateSpatialValues`) The next helper function is designed based on Table 1, w.r.t. the geometry values of the child-parent level members and based on the structure of a hierarchy step. We prepared Table 1 with topological relations based on DE-9IM.[8] We consider only the three simple geometry types, *point, line, and polygon* as the spatial

[7] Note that a level member might have the polygon geometry type for the parish borders and the point geometry type for the parish center, therefore a set of spatial values is required.

[8] DE-9IM (Dimensionally Extended Nine-Intersection Model) is a topological model that describes spatial relations of two geometries in two dimensions [6].

Algorithm 2: relateSpatialValues(v_{a_c}, v_{a_p}): topoRel$_i$

```
Input: v_ac, v_ap
Output: topoRel_i
begin
    topoRel_i = null; /*geoType(v_a) function returns the geometry type of a given attribute value*/
    switch (geoType(v_ac), geoType(v_ap)) do
        case (POINT, POINT)
            if equals?(v_ac, v_ap) then
                topoRel_i = qb4so:equals
        case (POINT, LINE)
            if intersects?(v_ac, v_ap) then
                topoRel_i = qb4so:intersects
        case (POINT, POLYGON)
            if within?(v_ac, v_ap) then
                topoRel_i = qb4so:within
            else if intersects?(v_ac, v_ap) then
                topoRel_i = qb4so:intersects
        case (LINE, LINE)
            if intersects?(v_ac, v_ap) then
                topoRel_i = qb4so:intersects
            else if overlaps?(v_ac, v_ap) then
                topoRel_i = qb4so:overlaps
        case (LINE, POLYGON)
            if within?(v_ac, v_ap) then
                topoRel_i = qb4so:within
            else if intersects?(v_ac, v_ap) then
                topoRel_i = qb4so:intersects
        case (POLYGON, POLYGON)
            if within?(v_ac, v_ap) then
                topoRel_i = qb4so:within
            else if intersects?(v_ac, v_ap) then
                topoRel_i = qb4so:intersects
    return topoRel_i
```

attribute values of child-parent level members in roll-up relations, excluding complex geometry types, such as multi-polygon, multi-point, etc. The possible topological relations that can occur in a spatial hierarchy step with a roll-up relation from child level to parent level are marked with check sign (✓) in the table. Topological relations, such as *contains* and *covers*, are not *hierarchically applicable* since a spatial child level member cannot contain or cover a spatial parent level member. For these relations, we mark the complete rows with minus sign (−) in the table, since they are not hierarchically applicable. Similarly, we mark the complete columns of *line-point*, *polygon-point*, and *polygon-line* roll-up relations with the minus sign (−) since these are also not hierarchically applicable. This is because we assume that in the instance data, a parent level member should always have a spatial attribute of a geometry type of the same or higher dimensionality of its child level member (a point is 0-dimensional, a line is 1-dimensional and a polygon is 2-dimensional).

For example, a child level member with a spatial attribute of line geometry can only have parent level member(s) with spatial attributes of line or polygon geometries but not point geometry. We mark the *topologically not applicable* relations with cross sign (×) according to the DE-9IM model (e.g, a line cannot overlap a polygon).

In Fig. 8, we depict the hierarchically and topologically applicable topological relations from Table 1. We simplified them by generalizing the possible relations, e.g., if a line *touches* or *crosses* another line at one point, they are both classified as *intersects* in Fig. 8(d). The most general relations are underlined in Fig. 8 for

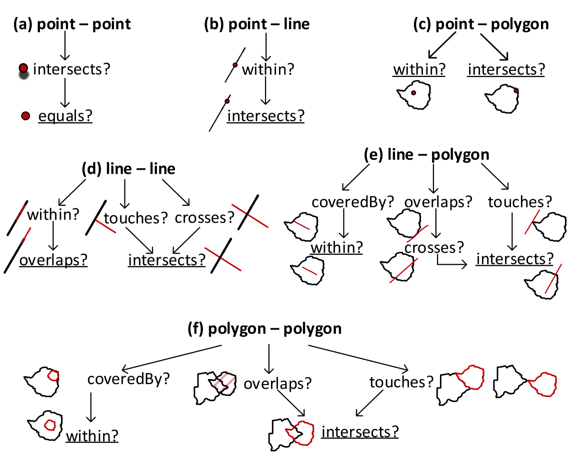

Fig. 8. Simplifying topological relations.

Table 1

Topological relations for hierarchy steps (✓: hierarchically and topologically applicable, ×: topologically not applicable, –: hierarchically not applicable)

Topological relations		Roll-up relations								
	Child level	Point (pt.)			Line (ln.)			Polygon (po.)		
	Parent level	pt.	ln.	po.	pt.	ln.	po.	pt.	ln.	po.
within		×	✓	✓	–	✓	✓	–	–	✓
contains		–	–	–	–	–	–	–	–	–
intersects		✓	✓	✓	–	✓	✓	–	–	✓
touches		×	×	×	–	✓	✓	–	–	✓
overlaps		×	×	×	–	✓	✓	–	–	✓
crosses		×	×	×	–	✓	✓	–	–	×
coveredBy		×	×	×	–	×	✓	–	–	✓
covers		–	–	–	–	–	–	–	–	–
equals		✓	×	×	–	✓	×	–	–	✓

each pair of geometry types (Fig. 8(a), (b), (c), (d), (e), and (f)).

In Algorithm 2 `relateSpatialValues`, we only consider these general topological relations that have a higher probability to satisfy the corresponding spatial predicates. For example, the topological relation *intersects* has the highest probability to satisfy from the DE-9IM matrix [6]. We generalize similar spatial predicates to ones that have higher probability to occur in a 2-dimensional space. For example, relations, such as a line *overlaps* (along the border of) a polygon, can be generalized to the relation – a line *crosses* a polygon at a minimum two points, which can later be generalized to the relation – a line *intersects* a polygon at a (minimum) single point as in Fig. 8(e). Similarly, a line *touches* a polygon at a single point can be generalized to the relation – a line *intersects* a polygon at a (minimum) single point.

The topological relation *coveredBy* requires an area of a geometry, therefore it is applicable only in line-polygon and polygon-polygon relations (Fig. 8(e) and 8(f)). For reasons of simplicity, we choose to generalize them as the *within* topological relation. In the algorithm, we also prioritize to check the topological relations based on the compared geometry types. If the spatial attribute values to relate are *point* and *polygon* geometry types, as in Fig. 8(c), it is more likely that a *point* is *within* a *polygon* than a *point intersects* a *polygon* in the instance data.

Therefore, we initially check for a more probable relation in the algorithm. For example, for the point-polygon relations case in Algorithm 2, Line 10: initially, the *within* spatial predicate is checked in the if statement (Line 11), then the *intersects* spatial predicate is checked in the else if statement (Line 13). After checking all the possible combinations of spatial attribute values in a `switch case`, a topological relation is returned from the algorithm (Line 30).

Now that we have introduced spatial helper functions, we present the main algorithms for finding the spatial hierarchy steps in the following.

4.1.2. *Detecting spatial hierarchy steps*

Algorithm 3 (`detectSpatialHS`) corresponds to case 1 (see the beginning of Section 4.1) and finds the explicit spatial hierarchy steps for QB4OLAP levels with `skos:broader` roll-up relations between child-parent level members. Intuitively, Algorithm 3 works as follows. Given instance graphs of attributes of level members $\mathcal{G}^I_{A(lm)}$ and roll-up relations of the hierarchy steps $\mathcal{G}^I_{\mathrm{RU}(hs)}$ between level members (using `skos:broader`), the key principle is to first retrieve pairs of child-parent level members based on the given input relationships $\mathcal{G}^I_{\mathrm{RU}(hs)}$. Then, spatial values are extracted (`getSpatialValues`) and finally the spatial relationship is verified (`relateSpatialValues`). The output $\mathcal{G}^I_{\mathrm{RU}(shs)}$ is then a graph of detected and verified hierarchy steps.

As an example, let us consider Listing 2: given Lines 1–6 ($G^I_A(lm)$), Line 7 ($G^I_R U(hs)$), and Lines 15–22 ($G^I_A(lm)$) as input, Algorithm 3 produces Line 14 ($G^I_R U(shs)$) as output.

Formally, Algorithm 3 works as follows:

Algorithm 3 (`detectspatialHS`) The input variables for Algorithm 3 are the instance graphs of attributes of level members $\mathcal{G}^I_{A(lm)}$ and roll-up relations of the hierarchy steps $\mathcal{G}^I_{\mathrm{RU}(hs)}$ between the level members. The RDF graph formulation of the attributes of the level members $A(lm)$ is: $\mathcal{G}^I_{A(lm)} =$

Algorithm 3: detectSpatialHS($\mathcal{G}^I_{\mathrm{RU}(hs)}$, $\mathcal{G}^I_{A(lm)}$): $\mathcal{G}^I_{\mathrm{RU}(shs)}$

Input: $\mathcal{G}^I_{A(lm)}$, $\mathcal{G}^I_{\mathrm{RU}(hs)}$
Output: $\mathcal{G}^I_{\mathrm{RU}(shs)}$

1 **begin**
2 $\mathcal{G}^I_{\mathrm{RU}(shs)} = \emptyset$; /*initialize output graph as emptyset*/
3 $\mathcal{G}^I_{A(lm_c)} = \emptyset$; $\mathcal{G}^I_{A(lm_p)} = \emptyset$; $V_{s(a_c)} = \emptyset$; $V_{s(a_p)} = \emptyset$; `topoRel`$_i$ $= null$; /*temporary variable and sets*/
4 **foreach** $((\mathrm{id}^I(lm_c)\ \mathrm{id}^S(a_c)\ v_{a_c}), (\mathrm{id}^I(lm_p)\ \mathrm{id}^S(a_p)\ v_{a_p})) \mid (\mathrm{id}^I(lm_c)\ \mathrm{id}^S(a_c)\ v_{a_c}), (\mathrm{id}^I(lm_p)\ \mathrm{id}^S(a_p)\ v_{a_p}) \in \mathcal{G}^I_{A(lm)} \wedge (\mathrm{id}^I(lm_c)$ `skos:broader` $\mathrm{id}^I(lm_p)) \in \mathcal{G}^I_{\mathrm{RU}(hs)} \wedge lm_c \rightsquigarrow v_{a_c} \wedge lm_p \rightsquigarrow v_{a_p} \wedge lm_c \sqsubseteq lm_p$ **do**
5   $\mathcal{G}^I_{A(lm_c)} = \{(\mathrm{id}^I(lm_c)\ \mathrm{id}^S(a_c)\ v_{a_c})\}$;
6   $V_{s(a_c)}$ =`getSpatialValues(`$\mathcal{G}^I_{A(lm_c)}$`);`
7   **if** $V_{s(a_c)} \neq \emptyset$ **then**
8     $\mathcal{G}^I_{A(lm_p)} = \{(\mathrm{id}^I(lm_p)\ \mathrm{id}^S(a_p)\ v_{a_p})\}$;
9     $V_{s(a_p)} =$ `getSpatialValues(`$\mathcal{G}^I_{A(lm_p)}$`);`
10   **if** $V_{s(a_p)} \neq \emptyset$ **then**
11   **foreach** $(v_{a_c}, v_{a_p}) \in V_{s(a_c)} \times V_{s(a_p)}$ **do**
12   `topoRel`$_i$ = `relateSpatialValues(`v_{a_c}, v_{a_p}`);`
13   **if** `topoRel`$_i \neq null$ **then**
14           $\mathcal{G}^I_{\mathrm{RU}(shs)} \cup = \{(\mathrm{id}^I(lm_c)$ `topoRel`$_i$ $\mathrm{id}^I(lm_p))\}$;
15 **return** $\mathcal{G}^I_{\mathrm{RU}(shs)}$

$\bigcup_{i=1}^{p}\{(\mathrm{id}^I(lm)\ \ \mathrm{id}^S(a_i)\ \ v_{a_i}) \mid lm \rightsquigarrow v_{a_i}\}$. Here, we denote by $lm \rightsquigarrow v_{a_i}$ that a level member lm has value v_{a_i} for attribute a_i (e.g., Listing 2, Lines 3–6, Lines 9–13, and Lines 17–22). The RDF graph formulation of the roll-up relations RU(hs) is: $\mathcal{G}^I_{\mathrm{RU}(hs)} = \bigcup_{i=1}^{k}\{(\mathrm{id}^I(lm_c)$ `skos:broader` $\mathrm{id}^I(lm_p)) \mid lm_{c_i} \sqsubseteq lm_{p_i}\}$. Here, we denote by $lm_{c_i} \sqsubseteq lm_{p_i}$ the partial order between level members, where a child level member lm_{c_i} rolls up to a parent level member lm_{p_i}[9] (e.g., Listing 2, Line 7).

The output of Algorithm 3 is the instance graph of roll-up relations for the *detected* spatial hierarchy steps $\mathcal{G}^I_{\mathrm{RU}(shs)}$ (e.g., Listing 2, Line 14). In Line 2, initially the output graph is initialized as an empty set. Next, in Line 3 we create two temporary graphs: $\mathcal{G}^I_{A(lm_c)}$ and $\mathcal{G}^I_{A(lm_p)}$ as empty sets,[10] to keep triple patterns separately in two graphs for attributes of child and parent level members. We also create two temporary sets: $V_{s(a_c)}$ and $V_{s(a_p)}$ for keeping the spatial attribute values from the child and parent level members, and initialize them as empty sets in Line 3. A set of spatial attribute values is defined over spatial literals $\mathcal{L}_s$ as $V_{s(a)} = \{v_{a_1}, \ldots, v_{a_i}, \ldots, v_{a_n} \mid 1 \leqslant i \leqslant n \wedge v_{a_i} \in \mathcal{L}_s\}$.

In the `foreach` loop in Line 4, we go through the elements of the input graphs $\mathcal{G}^I_{A(lm)}$ and $\mathcal{G}^I_{\mathrm{RU}(hs)}$ that are fulfilling a specific criteria, which is having an *explicit* `skos:broader` relation between child and parent level members.

In Line 5, while iterating through the `foreach` loop, we assign the set of triples of child level members and their attributes to the temporary graph $\mathcal{G}^I_{A(lm_c)}$. This temporary graph is given in Line 6 as an input to the helper function `getSpatialValues` (Algorithm 1), which finds the spatial attribute values from the given graph, and returns a set of spatial attribute values (i.e., $V_{s(a_c)}$) that are found in the input graph. The output of the helper function ($V_{s(a_c)}$) keeps the spatial attribute values of the child level member $\mathrm{id}^I(lm_c)$.

Next in Line 7, if $V_{s(a_c)}$ is not empty and has some spatial values of $\mathrm{id}^I(lm_c)$, we populate the next temporary graph $\mathcal{G}^I_{A(lm_p)}$ with its parent level $\mathrm{id}^I(lm_p)$ and attributes of the parent level in Line 8.

Similar to Line 6, Line 9 calls the helper function `getSpatialValues` with the input graph $\mathcal{G}^I_{A(lm_p)}$ and the output of the function is assigned to the temporary set $V_{s(a_p)}$. If this set is also not empty (Line 10), we go through the pairs of values (v_{a_c}, v_{a_p}) of the child-parent level members (Line 11), which are selected from the temporary graphs $\mathcal{G}^I_{A(lm_c)}$ and $\mathcal{G}^I_{A(lm_p)}$.

In this loop, we call the next helper function `relateSpatialValues` (Algorithm 2), where the input is the spatial value pairs. The output value of this function is the topological relation between the cor-

[9] We use subscript c and p to distinguish values for child and parent level members.
[10] Remark: a set of RDF triples is referred to as an RDF graph.

responding child and parent level members, and it is assigned to the initially created temporary variable $\texttt{topoRel}_i$ (Line 12). If this value is not null (checked in Line 13), `relateSpatialValues` function returns a topological relation (Line 12) that is satisfied as shown with a check-mark (✓) from Table 1.

Finally, the output graph for *spatial* hierarchy steps $\mathcal{G}^I_{\mathrm{RU}(shs)}$ is incrementally generated by adding the triple pattern with the topological relation (Line 14) and the output graph for the detected spatial hierarchy steps is returned (Line 15).

4.1.3. Discovering spatial hierarchy steps

Algorithm 4 (`discoverSpatialHS`) corresponds to case 2 (see the beginning of Section 4.1) and finds the implicit spatial hierarchy steps for QB4OLAP levels that do not have direct (`skos:broader`) roll-up relations. In this algorithm, we have to handle the situation where there are no explicit hierarchy steps between level members. Therefore, we benefit from schema graphs capturing dimensions, hierarchies, and levels by iterating through the RDF triples and compare the spatial attribute values of the level members to find the $\mathcal{G}^I_{\mathrm{RU}(shs)}$ topological relations within the same dimension.

Intuitively, Algorithm 4 works very much like Algorithm 3, the main difference being that in the absence of a direct link between the members, we need to find it first. Hence, we find pairs of level members exploiting information about dimensions, hierarchies in dimensions, and levels in hierarchies, which is provided by QB4OLAP. The detected pairs are then treated in a similar way as the child-parent level member pairs in Algorithm 3.

As an example, let us consider Listing 2: given Lines 1–6 ($G^I_A(lm)$) and Lines 15–22 ($G^I_A(lm)$) as input, Algorithm 4 produces Line 14 ($G^I_R U(shs)$) as output.

Formally, Algorithm 4 works as follows:

Algorithm 4 (`discoverSpatialHS`) The input variables for Algorithm 4 are the schema graphs of dimensions $\mathcal{G}^S_D$, hierarchies of the dimensions $\mathcal{G}^S_{H(d)}$, levels of the hierarchies $\mathcal{G}^S_{L(h)}$, the instance graphs of level members of levels $\mathcal{G}^I_{\mathrm{LM}(l)}$, and attributes of level members $\mathcal{G}^I_{A(lm)}$. Each dimension $d \in D$ has a set of hierarchies $H(d)$, which is shown in the RDF graph formulation for a dimension $d \in D$ as: $\mathcal{G}^S_d = \bigcup_{h \in H(d)} \{(\mathrm{id}^S(d)\ \texttt{qb4o:hasHierarchy}\ \mathrm{id}^S(h))\}$. Each hierarchy $h \in H(d)$ belongs to a dimension d and has a set of levels $L(h)$, which is shown in the RDF graph formulation for a hierarchy $h \in H(d)$ as: $\mathcal{G}^S_h = \{(\mathrm{id}^S(h)\ \texttt{qb4o:inDimension}\ \mathrm{id}^S(d))\} \cup \bigcup_{l \in L(h)} \{(\mathrm{id}^S(h)\ \texttt{qb4o:hasLevel}\ \mathrm{id}^S(l))\}$. Each level l has a set of level members $\mathrm{LM}(l) = \{lm_1, \ldots, lm_y\}$, which is shown in the RDF graph formulation for a level member $lm \in \mathrm{LM}(l)$ as:

$$\mathcal{G}^I_{lm} = \left\{(\mathrm{id}^I(lm)\ \texttt{qb4o:memberOf}\ \mathrm{id}^S(l))\right\}.$$

Each level member lm has a set of attributes $A(lm)$. The RDF graph formulation of attributes of level members $\mathcal{G}^I_{A(lm)}$ is already given in Section 4.1.2. In Listing 2, examples of a triple pattern for level members and attributes of level members are given in Lines 1–6, Lines 8–13 and Lines 15–22, without explicit roll-up relations (Line 7).

The output of Algorithm 4 is the instance graph of roll-up relations for the *discovered* spatial hierarchy steps $\mathcal{G}^I_{\mathrm{RU}(shs)}$ (e.g., Listing 2, Line 14). In Line 2, the output graph is initialized as an empty set. And a temporary variable ($\texttt{topoRel}_i$) for keeping the discovered topological relations is initialized as *null*. In Line 4, we create two temporary graphs: $\mathcal{G}^I_{A(lm_n)}$ and $\mathcal{G}^I_{A(lm_k)}$ as empty sets similar to Algorithm 3. We also create two temporary sets: $V_{s(a_n)}$ and $V_{s(a_k)}$ for storing spatial attribute values and initialize them as empty sets in Line 3.

To discover the spatial hierarchy steps, we need to get the attributes of all the level members from the instance graph ($\mathcal{G}^I_{A(lm)}$) and compare their spatial attribute values in pairs, where the pairs of level member attributes should be coming from two different levels in the same dimension hierarchy. Therefore, before getting the attributes of the level members, we need to classify the level members as they are grouped in different levels of a dimension hierarchy.

To achieve that, we use the schema definitions readily available in QB4OLAP, by looping through in Algorithm 4, in nested loops of dimensions in Line 5, hierarchies in the dimension (Line 6), levels in the hierarchy (Line 7). This helps us to determine the levels in a dimension hierarchy, where we can get level pairs from the same hierarchy (Line 8).

Now, while looping through the level pairs, we can identify the level members via the `qb4o:memberOf` property (Line 9). We get a pair of level members, where each level member should come from a different level, then we iterate through that pair of level members (Line 10).

Then, we get the triple patterns for the attributes of the level members from the each of the level mem-

Algorithm 4: discoverSpatialHS($\mathcal{G}^S_D$, $\mathcal{G}^S_{H(d)}$, $\mathcal{G}^S_{L(h)}$, $\mathcal{G}^I_{\mathrm{LM}(l)}$, $\mathcal{G}^I_{A(lm)}$): $\mathcal{G}^I_{\mathrm{RU}(shs)}$.

Input: $\mathcal{G}^S_D$, $\mathcal{G}^S_{H(d)}$, $\mathcal{G}^S_{L(h)}$, $\mathcal{G}^I_{\mathrm{LM}(l)}$, $\mathcal{G}^I_{A(lm)}$
Output: $\mathcal{G}^I_{\mathrm{RU}(shs)}$

1 **begin**
2 $\mathcal{G}^I_{\mathrm{RU}(shs)} = \emptyset$; `topoRel`$_i$ $= null$ /*initialize the output graph as an empty set and a temporary variable as null*/
3 $V_{s(a_n)} = \emptyset$; $V_{s(a_k)} = \emptyset$; /*initialize temporary sets as empty sets for keeping spatial attribute values*/
4 $\mathcal{G}^I_{A(lm_n)} = \emptyset$; $\mathcal{G}^I_{A(lm_k)} = \emptyset$; /*initialize empty sets to keep triple patterns for attributes of level members*/
5 **foreach** ($\mathrm{id}^S(d)$ `qb4o:hasHierarchy` $\mathrm{id}^S(h)) \in \mathcal{G}^S_D$ /*iterate through the dimensions*/ **do**
6 **foreach** ($\mathrm{id}^S(h)$ `qb4o:inDimension` $\mathrm{id}^S(d)) \in \mathcal{G}^S_H(d)$ /*iterate through the hierarchies*/ **do**
7 **foreach** ($\mathrm{id}^S(h)$ `qb4o:hasLevel` $\mathrm{id}^S(l)) \in \mathcal{G}^S_H(d)$ /*while iterating through the levels in the hiearchy*/ **do**
8 **foreach** $(\mathrm{id}^S(l_i), \mathrm{id}^S(l_j)) \in \mathcal{G}^S_{L(h)} \times \mathcal{G}^S_{L(h)} \mid \mathrm{id}^S(l_i) \neq \mathrm{id}^S(l_j) \wedge$ /*. . . get level pairs $(\mathrm{id}^S(l_i), \mathrm{id}^S(l_j)) * /$
9 $\bigcup_{lm \in \mathrm{LM}(l)}$((id$^I(lm)$ `qb4o:memberOf` id$^S(l_i)$), (id$^I(lm)$ `qb4o:memberOf` id$^S(l_j)$)) $\in \mathcal{G}^I_{\mathrm{LM}(l)}$ /*in each level pair, while iterating through their level members, get a pair of level members $(id^I(lm_n), id^I(lm_k))$, where each level member comes from different levels*/ **do**
10 **foreach** $(\mathrm{id}^I(lm_n), \mathrm{id}^I(lm_k)) \in \mathcal{G}^I_{\mathrm{LM}(l)} \times \mathcal{G}^I_{\mathrm{LM}(l)} \mid \mathrm{id}^I(lm_n) \neq \mathrm{id}^I(lm_k) \wedge \mathrm{id}^I(lm_n) \in \mathcal{G}^I_{\mathrm{LM}(l_i)} \implies \mathrm{id}^I(lm_k) \in \mathcal{G}^I_{\mathrm{LM}(l_j)} \mid \mathcal{G}^I_{\mathrm{LM}(l_i)} \subset \mathcal{G}^I_{\mathrm{LM}(l)} \wedge \mathcal{G}^I_{\mathrm{LM}(l_j)} \subset \mathcal{G}^I_{\mathrm{LM}(l)} \wedge \mathcal{G}^I_{\mathrm{LM}(l_i)} \neq \mathcal{G}^I_{\mathrm{LM}(l_j)}$ /*iterate through the pairs of level members*/ **do**
11 **foreach** $((\mathrm{id}^I(lm_n)\ \mathrm{id}^S(a_i)\ v_{a_i}), (\mathrm{id}^I(lm_k)\ \mathrm{id}^S(a_j)\ v_{a_j})) \in \mathcal{G}^I_{A(lm)} \times \mathcal{G}^I_{A(lm)}$ /*iterate through the pairs of level members' attributes*/ **do**
12 $\mathcal{G}^I_{A(lm_n)} = \{(\mathrm{id}^I(lm_n)\ \mathrm{id}^S(a_i)\ v_{a_i})\}$; $\mathcal{G}^I_{A(lm_k)} = \{(\mathrm{id}^I(lm_k)\ \mathrm{id}^S(a_j)\ v_{a_j})\}$;
13 $V_{s(a_n)} =$ `getSpatialValues`($\mathcal{G}^I_{A(lm_n)}$); $V_{s(a_k)} =$ `getSpatialValues`($\mathcal{G}^I_{A(lm_k)}$);
14 **if** $V_{s(a_n)} \neq \emptyset \wedge V_{s(a_k)} \neq \emptyset$ /*make sure there are spatial values in the temporary sets*/ **then**
15 **foreach** $(v_{a_i}, v_{a_j}) \in V_{s(a_n)} \times V_{s(a_k)}$ **do**
16 `topoRel`$_i$ = `relateSpatialValues`(v_{a_i}, v_{a_j});
17 **if** `topoRel`$_i$ $\neq null$ /*make sure there is a topological relation assigned to the variable*/ **then**
18 $\mathcal{G}^I_{\mathrm{RU}(shs)} \cup= \{(\mathrm{id}^I(lm_n)$ `topoRel`$_i$ $\mathrm{id}^I(lm_k))\}$;
19 **return** $\mathcal{G}^I_{\mathrm{RU}(shs)}$

ber in the pair, and iterate through those pairs of the triple patterns (Line 11). While iterating through the triple patterns, we insert them to the temporary graphs $\mathcal{G}^I_{A(lm_n)}$ and $\mathcal{G}^I_{A(lm_k)}$ (Line 12), which are created earlier as empty sets in Line 4. So, we can filter the spatial values from the triple patterns kept in the temporary graphs by calling the helper function `getSpatialValues` (Algorithm 1), with those input graphs $\mathcal{G}^I_{A(lm_n)}$ and $\mathcal{G}^I_{A(lm_k)}$ (Line 13).

Next, we call the helper function `getSpatialValues` (Algorithm 1) twice, with the input graphs $\mathcal{G}^I_{A(lm_n)}$ and $\mathcal{G}^I_{A(lm_k)}$. The outputs of the each (helper) function call are assigned to the temporary sets $V_{s(a_n)}$ and $V_{s(a_k)}$ correspondingly (Line 13). If these sets are not empty (Line 14), it means that `getSpatialValues` identified spatial values in the triple patterns of the input graphs.

Then, we iterate through the spatial value pairs retrieved from the each of the sets (Line 15). In this loop, we call the next helper function `relateSpatialValues` (Algorithm 2), where the input is the spatial value pairs. The output value of this function is the topological relation between the corresponding level members, and it is assigned to the initially created temporary variable $\texttt{topoRel}_i$ (Line 16).

Finally, if this $\texttt{topoRel}_i$ value is not null (Line 17), the output graph for the *spatial* hierarchy steps $\mathcal{G}^I_{\mathrm{RU}(shs)}$ is incrementally generated by adding the triple pattern with the topological relation (Line 18) and the output graph for the *discovered* spatial hierarchy steps is returned in Line 19.

4.2. Factual enrichment phase

The factual enrichment phase is built around the observation facts and their spatial attributes a.k.a spatial measures and fact-dimension relations (Section 2.1).

In QB4OLAP facts are linked to the dimensions at the lowest granularity level, which is the base level of the dimensions. For example, the GeoFarmHerdState cube has two spatial base levels linked to the cube: Parish level and Farm level. The GeoFarmHerdState cube also has a spatial measure listed in the cube: FarmLocation (Fig. 3). In QB4OLAP, a fact schema defines the structure of a cube with the `qb:DataStructureDefinition` property (Listing 3, Line 1). Base levels (Lines 2 and 4) and measures (Line 6) are given as `qb:components` of the fact (Listing 3). The cardinality relationship between the base level and the fact can also be represented with `qb4o:cardinality` in QB4OLAP as given in Lines 2 and 4 in Listing 3.

On the other hand, with QB4SOLAP we can also represent fact-level topological relations that are similar to the topological relations between the child-

```
  ##Spatial Fact Schema in QB4SOLAP##
1 gfs:GeoFarmHerdState a qb:DataStructureDefinition ;
  #Lowest spatial level for each dimension in the cube#
2 qb:component [qb4o:level gfs:farm ; qb4o:cardinality qb4o:ManyToOne ;
3     qb4so:topologicalRelation qb4so:Equals] ;
4 qb:component [qb4o:level gfs:parish ; qb4o:cardinality qb4o:ManyToOne ;
5     qb4so:topologicalRelation qb4so:Within] ;
  #Example of a spatial measure in the cube#
6 qb:component [qb:measure gfs:farmLocation ;
7     qb4o:aggregateFunction qb4so:ConvexHull] .
```

Listing 3. GeoFarmHerdState fact schema definition in QB4SOLAP

```
  ##GeoFarmHerdState cube: observation fact example##
1 gfsi:farmState_103850_12_2015 a qb:Observation ;
2     gfs:farm gfsi:farm_103850 ; gfs:parish gfsi:parish_8648 ;
3     gfs:livestockUnit "4.2699999999999996"^^xsd:double ;
4     gfs:farmLocation "POINT (8.31941 56.75822)"^^geo:spatialLiteral ;
5     qb4so:equals gfsi:farm_103850 ; qb4so:within gfsi:parish_8648 ;
6     qb4so:within gfsi:water_3770 .
```

Listing 4. GeoFarmHerdState fact member with base levels and measures

parent levels at the hierarchy steps. Fact-level topological relations are given in spatial fact schema with blue in Lines 3 and 5 (Listing 3). QB4SOLAP also extends the (cube) schema with spatial aggregate functions, which are defined over spatial measures as highlighted in blue (Listing 3, Line 7).

An example of an observation fact (fact member) at the instance level is given in Listing 4. A fact member is a `qb:Observation` (Line 1), which is related to the base levels (Line 2) with respect to the data structure definition (DSD) of the fact schema, and has a set of measures (Lines 3, 4) where some measures (Line 4) might have spatial values (Listing 4). To define a QB4OLAP fact schema, first, we need to enrich the fact members by annotating with topological relations as highlighted with blue in Line 5. We can derive topological relations between fact members and the (base) level members by comparing the spatial measures of the fact members and spatial attributes of the (base) level members with Boolean spatial predicates. The links between fact members and base level members are already given explicitly in Line 2 (Listing 4). However, these links are simple references between the fact and base level members, which do not describe the nature of the topological relation. By applying Boolean spatial predicates on fact and level members, we can find the exact topological relations, i.e., if a fact member *intersects* with the level member or if a fact member is *within* the level member. We explain how to detect these *explicit* fact-level (topological) relations in Section 4.2.1.

Moreover, there might also be some missing links between the (observartions) fact members and the corresponding base level members. For this case we need to find all the base level members that are spatial and derive the links between the spatial measure values and spatial attribute values (of the base level members) by using Boolean spatial predicates. We explain how to discover fact-level (topological) relations, which are not explicitly linked between observation fact and base level members in Section 4.2.2.

There are also cases where we would like to establish a direct (topological) relation between the fact members and higher granularity (parent) level members, which are not at the base level of the dimension. Using the example depicted in Fig. 4 we explained that wrongly aggregating the measures (i.e., double counting) becomes a problem when we roll-up between the levels that have many-to-many (N:M) cardinality relations (as in Parish and Drainage Area levels). Therefore, it is necessary to drill-down to the lowest granularity (fact members) and find the direct relation between the observation fact members and the corresponding level members of the higher level in many-to-many cardinality relations.

In order to prevent this problem, we address the issue in our algorithm to discover and annotate the fact-level (topological) relations that are between the observation fact members and level members of a higher level in an N:M cardinality relation in Section 4.2.2. For example, such a relation is given in green in Line 6 (Listing 4) that shows a topological relation between an observation fact member (farm state) and a higher level – not a base level – member (drainage area).

Finally, in Section 4.2.3 we explain how to define a data structure definition (DSD) of spatial fact schema using a QB4OLAP fact schema and the spatial fact member instances derived in the previous two algorithms.

4.2.1. Detecting explicit fact-level relations

In this section, we present an algorithm for detecting explicit fact-level topological relations between observation fact members and base level members where there is a direct reference between the fact member and the base level member. To derive these topological relations we need to get the spatial attributes of fact members (spatial measures) and base level members.

Algorithm 5 (`detectFactLevelRelations`) The input variables for Algorithm 5 are the instance graphs of fact members $\mathcal{G}^I_{\mathrm{FM}(F)}$, level members $\mathcal{G}^I_{\mathrm{LM}(l)}$, and attributes of level members $\mathcal{G}^I_{A(lm)}$.

Every fact member $f_i \in$ FM has an IRI $\mathrm{id}^I(f_i)$ and defined as a `qb:Observation`. The RDF graph formulation of a fact member f_i is:

$$\mathcal{G}^I_{f_i} = \bigcup_{l_j \in L(f_i)} \{(\mathrm{id}^I(f_i)\ \mathrm{id}^S(l_j)\ \mathrm{id}^I(lm_j) \mid f_i \rightsquigarrow lm_j\} \cup \bigcup_{m_k \in M(f_i)} \{(\mathrm{id}^I(f_i)\ \mathrm{id}^S(m_k)\ v_{m_k} \mid f_i \rightsquigarrow v_{m_k}\}.$$

Algorithm 5: detectFactLevelRelations($\mathcal{G}^I_{\mathrm{FM}(F)}$, $\mathcal{G}^I_{A(lm)}$) : $\mathcal{G}^I_{\mathrm{FM}(F_s)}$

Input: $\mathcal{G}^I_{\mathrm{FM}(F)}$, $\mathcal{G}^I_{A(lm)}$
Output: $\mathcal{G}^I_{\mathrm{FM}(F_s)}$

begin
  $\mathcal{G}^I_{\mathrm{FM}(F_s)} = \mathcal{G}^I_{\mathrm{FM}(F)}$; `topoRel`$_i$ = *null*; $\mathcal{G}^I_{A(f_i m_k)} = \emptyset$;
  $\mathcal{G}^I_{A(lm_j)} = \emptyset$; $V_{s(m_k)} = \emptyset$; $V_{s(a_i)} = \emptyset$;
  /*initialize the ouput graph, temporary variable and sets*/
  foreach
  /*get each observation fact (fact member)*/
  ($\mathrm{id}^I(f_i)$ `rdf:type qb:Observation`) $\in \mathcal{G}^I_{\mathrm{FM}(F)}$ **do**
    foreach
    /*get measure-level member pairs*/
    $((\mathrm{id}^I(f_i)\ \mathrm{id}^S(m_k)\ v_{m_k}), (\mathrm{id}^I(f_i)\ \mathrm{id}^S(l_j)\ \mathrm{id}^I(lm_j)))$
    $\in \mathcal{G}^I_{\mathrm{FM}(F)} \times \mathcal{G}^I_{\mathrm{FM}(F)} \mid f_i \rightsquigarrow v_{m_k} \wedge lm_j \rightsquigarrow v_{a_i} \wedge$
    $(\mathrm{id}^I(lm_j)\ \mathrm{id}^S(a_i)\ v_{a_i}) \in \mathcal{G}^I_{A(lm)}$
    /*get measure and attribute values of level members*/ **do**
      $\mathcal{G}^I_{A(f_i m_k)} = \{(\mathrm{id}^I(f_i)\ \mathrm{id}^S(m_k)\ v_{m_k})\}$;
      $V_{s(m_k)} =$ `getSpatialValues`($\mathcal{G}^I_{A(f_i m_k)}$);
      if $V_{s(m_k)} \neq \emptyset$ **then**
        $\mathcal{G}^I_{A(lm_j)} = \{(\mathrm{id}^I(lm_j)\ \mathrm{id}^S(a_i)\ v_{a_i})\}$;
        $V_{s(a_i)} =$ `getSpatialValues`($\mathcal{G}^I_{A(lm_j)}$);
        if $V_{s(a_i)} \neq \emptyset$ **then**
          foreach $(v_{m_k}, v_{a_i}) \in V_{s(m_k)} \times V_{s(a_i)}$ /*foreach spatial value pairs*/ **do**
            `topoRel`$_i$ = `relateSpatialValues`(v_{m_k}, v_{a_i});
            if `topoRel`$_i \neq$ *null* **then**
              $\mathcal{G}^I_{\mathrm{FM}(F_s)} \cup= \{(\mathrm{id}^I(f_i)$ `topoRel`$_i$ $\mathrm{id}^I(lm_j))\}$;
  return $\mathcal{G}^I_{\mathrm{FM}(F_s)}$

Here, we denote by $f_i \rightsquigarrow lm_j$ that a fact member f_i has an explicit link to a level member lm_j (e.g., Listing 4, Line 3). Note that we denote by $lm \rightsquigarrow v_{a_i}$ that a level member lm has value v_{a_i} for attribute a_i (Section 4.1.2), which is used in Algorithm 5 (Line 12) to get the attribute values of the linked level members. Moreover, we denote here by $f_i \rightsquigarrow v_{m_k}$ that a fact member lm has value v_{m_k} for measure m_k (e.g., Listing 4, Lines 5 and 6). The RDF graph formulation of the other input variables are: attributes of level members $\mathcal{G}^I_{A(lm)}$ and level members $\mathcal{G}^I_{\mathrm{LM}(l)}$ are already given, respectively, in Sections 4.1.2 and 4.1.3.

The output of Algorithm 5 is the enriched instance graph of fact members with topological relations $\mathcal{G}^I_{\mathrm{FM}(F_s)}$. In Line 2, we initialize the output graph as the input graph of fact members (without topological relations) so that we can gradually enrich it with the detected topological relations (Line 22). Initially, the topological relation variable $\texttt{topoRel}_i$ is set to *null*. We also create two temporary graphs: $\mathcal{G}^I_{A(lm_j)}$ and $\mathcal{G}^I_{A(f_i m_k)}$ as empty sets to keep triple patterns separately in two graphs for attributes of level members and (measures of) fact members. We also create two temporary sets: $V_{s(m_k)}$ and $V_{s(a_i)}$ for keeping the spatial values from the fact and level members, and initialize them also as empty sets in Line 3.

In the first `foreach` loop (Line 4 and 5) we retrieve the observation fact members from the input graph of fact members, which corresponds to Line 1 in Listing 4. Getting the fact members allows us to access each of their measures in Line 6 and level members in Line 7 (Algorithm 5). In the next `foreach` loop (Line 9) we match each measure-level member pair, where we can already retrieve the measure values from the input graph of fact members $\mathcal{G}^I_{\mathrm{FM}(F)}$ (Line 10) and through the input graph for attributes of the level members $\mathcal{G}^I_{A(lm)}$ (Line 11 and 12), we can retrieve the attribute values. In Line 13, we assign the set of triples for measure attributes of fact members to a temporary graph $\mathcal{G}^I_{A(f_i m_k)}$ created earlier in Line 2. This temporary graph is given as an input to the helper function `getSpatialValues` (Algorithm 1) in Line 14 (Algorithm 5). The helper function returns the spatial attribute (measure) values of the fact members, which are kept in the temporary set $V_{s(m_k)}$. If this set is not empty (checked in Line 15) and has some spatial measures of fact member $\mathrm{id}^I(f_i)$, we repeat the same procedure for retrieving the spatial attribute values of level member $\mathrm{id}^I(lm_j)$ in Lines 16 and 17. If the output set for spatial attribute values $V_{s(a_i)}$ is also not empty (Line 18), then we go through the pairs of spatial values (v_{m_k}, v_{a_i}) in Line 19. In this loop, we call the next helper function `relateSpatialValues` (Algorithm 2), where the input is the spatial value pairs. The output value of this function is the topological relation between the corresponding fact and level members, which is assigned to the variable $\texttt{topoRel}_i$ (Line 20).

4.2.2. Discovering implicit fact-level relations

In this section, we present an algorithm for discovering fact-level (topological) relations, where there are no direct links between the fact and level members. This algorithm handles the following situations: 1) Finding the topological relations between observation facts and base level members; 2) Finding the topological relations between observation facts and parent level members in an N:M cardinality relation. In both cases there are no direct links between the observation facts and level members. Therefore, we benefit from (QB4OLAP) schema graphs of dimensions, hierarchies, and levels for iterating through the RDF triples to distinguish the base level members, and find the parent level members, when there is an N:M cardinality relation between the levels of a hierarchy at a hierarchy step.

Algorithm 6 (`discoverFactLevelRelations`) The input variables at the schema level for Algorithm 6 are the schema graphs of dimensions $\mathcal{G}^S_D$, hierarchies of the dimensions $\mathcal{G}^S_{H(d)}$, levels of the hierarchies $\mathcal{G}^S_{L(h)}$, and hierarchy steps of the hierarchies $\mathcal{G}^S_{\mathrm{HS}(h)}$. The RDF graph formulations of the schema level input variables (dimensions $\mathcal{G}^S_{H(d)}$, hierarchies $\mathcal{G}^S_{H(d)}$, and levels $\mathcal{G}^S_{L(h)}$) are already given in Section 4.1.3. Therefore, we only explain the structure of a hierarchy step in the schema graph. Each hierarchy step hs_i is defined in the schema graph $\mathcal{G}^S_{\mathrm{HS}(h)}$ as a blank node $\texttt{\_:hs}_i \in \mathcal{B}$. Each hierarchy step is linked to a hierarchy $\mathrm{id}^S(h)$ with the `qb4o:inHierarchy` predicate and has a child level $\mathrm{id}^S(l_c)$, a parent level $\mathrm{id}^S(l_p)$, and a cardinality relation $\mathrm{id}^S(card)$, which are provided with `qb4o:childLevel`, `qb4o:parentLevel`, and `qb4o:pcCardinality` predicates in Line 6.

The input variables at the instance level are the instance graphs of fact members $\mathcal{G}^I_{\mathrm{FM}(F)}$, level members of levels $\mathcal{G}^I_{\mathrm{LM}(l)}$, and attributes of level members $\mathcal{G}^I_{A(lm)}$. We have already explained the RDF graph formulations of the instance level input variables (fact members $\mathcal{G}^I_{\mathrm{FM}(F)}$, level members $\mathcal{G}^I_{\mathrm{LM}(l)}$, and attributes of level members $\mathcal{G}^I_{A(lm)}$) in Section 4.2.1.

Algorithm 6: discoverFactLevelRelations($\mathcal{G}^I_{\text{FM}(F)}, \mathcal{G}^I_{\text{LM}(l)}, \mathcal{G}^I_{A(lm)}, \mathcal{G}^S_D, \mathcal{G}^S_{H(d)}, \mathcal{G}^S_{\text{HS}(h)}$) : $\mathcal{G}^I_{\text{FM}(F_s)}$

Input: $\mathcal{G}^I_{\text{FM}(F)}, \mathcal{G}^I_{\text{LM}(l)}, \mathcal{G}^I_{A(lm)}, \mathcal{G}^S_D, \mathcal{G}^S_{H(d)}, \mathcal{G}^S_{\text{HS}(h)}$
Output: $\mathcal{G}^I_{\text{FM}(F_s)}$

begin
  $\mathcal{G}^I_{\text{FM}(F_s)} = \mathcal{G}^I_{\text{FM}(F)}$; `topoRel`$_i$ = *null*; /*initialize the output graph and temporary variable*/
  $\mathcal{G}^I_{A(f_i m_k)} = \emptyset$; $\mathcal{G}^I_{A(lm_j)} = \emptyset$; $V_{s(m_k)} = \emptyset$; $V_{s(a_i)} = \emptyset$; /*initialize temporary graphs and sets as empty set*/
  foreach ($\text{id}^S(d)$ `qb4o:hasHierarchy` $\text{id}^S(h)) \in \mathcal{G}^S_D$ /*iterate through the dimensions*/ **do**
    foreach ($\text{id}^S(h)$ `qb4o:inDimension` $\text{id}^S(d)) \in \mathcal{G}^S_H(d)$ /*iterate through the hierarchies*/ **do**
      foreach ($\text{id}^S(h)$ `qb4o:hasLevel` $\text{id}^S(l_n)) \in \mathcal{G}^S_H(d)$ /*iterate through the levels in the hierarchy*/ **do**
        foreach (`_:hs`$_i$ `qb4o:inHierarchy` $\text{id}^S(h)) \in \mathcal{G}^S_{\text{HS}(h)}$ | (`_:hs`$_i$ `qb4o:childLevel` $\text{id}^S(l_c)) \in \mathcal{G}^S_{\text{HS}(h)} \wedge$ (`_:hs`$_i$ `qb4o:parentLevel` $\text{id}^S(l_p)) \in \mathcal{G}^S_{\text{HS}(h)} \wedge$ (`_:hs`$_i$ `qb4o:pcCardinality` $\text{id}^S(card)) \in \mathcal{G}^S_{\text{HS}(h)}$ /*each hierarchy step has a child level (l_c), a parent level (l_p), and a cardinality relation between these levels*/ **do**
          if $(\text{id}^S(l_n) \neq \text{id}^S(l_p)) \vee (\text{id}^S(l_n) = \text{id}^S(l_p) \wedge \text{id}^S(card) =$ `qb4o:ManyToMany`) /*check in each hierarchy step that level l_n should not be annotated as a parent level l_p, thus it is a base level OR if it is a parent level, there should be also a N:M cardinality relation in the hierarchy step*/ **then**
            foreach ($\text{id}^I(lm_j)$ `qb4o:memberOf` $\text{id}^S(l_n)) \in \mathcal{G}^I_{\text{LM}(l)}$ /*get level members of the level l_n*/ **do**
              foreach (($\text{id}^I(lm_j)$ `qb4o:memberOf` $\text{id}^S(l_n)$), ($\text{id}^I(f_i)$ `rdf:type qb:Observation`)) $\in \mathcal{G}^I_{\text{LM}(l)} \times \mathcal{G}^I_{\text{FM}(F)} \mid \bigcup_{m_k \in M(f_i)} (\text{id}^I(f_i)\ \text{id}^S(m_k)\ v_{m_k}) \in \mathcal{G}^I_{\text{FM}(F)} \wedge \bigcup_{a_i \in A(lm)} (\text{id}^I(lm_j)\ \text{id}^S(a_i)\ v_{a_i}) \in \mathcal{G}^I_{A(lm)}$ /*get level member-fact member pairs, where each fact member has some measure values v_{m_k}, and each level member has some attribute values v_{a_i} */ **do**
                foreach (($\text{id}^I(f_i)\ \text{id}^S(m_k)\ v_{m_k}$), ($\text{id}^I(lm_j)\ \text{id}^S(a_i)\ v_{a_i}$)) $\in \mathcal{G}^I_{\text{FM}(F)} \times \mathcal{G}^I_{A(lm)}$ **do**
                  $\mathcal{G}^I_{A(f_i m_k)} = \{(\text{id}^I(f_i)\ \text{id}^S(m_k)\ v_{m_k})\}$; $\mathcal{G}^I_{A(lm_j)} = \{(\text{id}^I(lm_j)\ \text{id}^S(a_i)\ v_{a_i})\}$;
                  $V_{s(m_k)} =$ `getSpatialValues(`$\mathcal{G}^I_{A(f_i m_k)}$`)`; $V_{s(a_i)} =$ `getSpatialValues(`$\mathcal{G}^I_{A(lm_j)}$`)`;
                  if $V_{s(m_k)} \neq \emptyset \wedge V_{s(a_i)} \neq \emptyset$ **then**
                    foreach $(v_{m_k}, v_{a_i}) \in V_{s(m_k)} \times V_{s(a_i)}$ **do**
                      `topoRel`$_i$ = `relateSpatialValues(`v_{m_k}, v_{a_i}`)`;
                      if `topoRel`$_i \neq$ *null* **then**
                        $\mathcal{G}^I_{\text{FM}(F_s)} \cup= \{(\text{id}^I(f_i)$ `topoRel`$_i$ $\text{id}^I(lm_j))\}$;
  return $\mathcal{G}^I_{\text{FM}(F_s)}$

The output of Algorithm 6 is the enriched instance graph of fact members with the topological relations $\mathcal{G}^I_{\mathrm{FM}(F_s)}$. In Line 2, we initialize the output graph as the input graph of fact members (without topological relations) so that we can gradually enrich it with the detected topological relations (Line 22). Initially, the topological relation variable $\texttt{topoRel}_i$ is set to *null*. We also create two temporary graphs: $\mathcal{G}^I_{A(lm_j)}$ and $\mathcal{G}^I_{A(f_i m_k)}$ as empty sets to keep triple patterns separately in two graphs for attributes of level members and (measures of) fact members. We also create two temporary sets: $V_{s(m_k)}$ and $V_{s(a_i)}$ for keeping the spatial values from the fact and level members and initialize them also as empty sets in Line 3.

To find the topological relations between observation facts (with spatial measures) and base level members (with spatial attributes), first, we need to find all the base levels since there is no direct link between the fact and level members. To achieve this in Algorithm 6, we use the schema definitions readily available in QB4OLAP. In Line 4, we iterate through the nested loops of dimensions to get the hierarchies and in Line 5 we iterate the nested loops of hierarchies to get the hierarchy levels. To find the base level of a hierarchy, we have to iterate through the hierarchy steps, where each hierarchy step describes a child level, a parent level and a cardinality relation between the levels (Line 6). If a level $\mathrm{id}^S(l_n)$ has never been assigned as a parent level with `qb4o:parentLevel` predicate in any of the hierarchy steps in a hierarchy h from the schema graph $\mathcal{G}^S_{\mathrm{HS}(h)}$, then l_n is the base level of a hierarchy h (Line 7).

Thus, we can retrieve the level members of level l_n from the instance graph level members $\mathcal{G}^I_{\mathrm{LM}(l)}$ (Line 8). In the next `foreach` loop we can pair the level members from the instance graph $\mathcal{G}^I_{\mathrm{LM}(l)}$, and observation facts from the instance graph of fact members $\mathcal{G}^I_{\mathrm{FM}(F)}$ (Line 9). We can retrieve a set of attributes (measures) for fact members from the fact members graph (Line 10), and a set of attributes for level members from the instance graph $\mathcal{G}^I_{A(lm)}$ (Line 11).

Then, in the next `foreach` loop in Line 12, we get the triple patterns with each measure values of the fact member and attribute values of the level member in pairs. While iterating through the (pair of) triple patterns, we insert each member of the pair to the temporary graphs for measures of fact members $\mathcal{G}^I_{A(f_i m_k)}$ and attributes of level members $\mathcal{G}^I_{A(lm_j)}$ (Line 13), which are created earlier as empty sets in Line 3. Then, we can filter the spatial values from the triple patterns kept in the temporary graphs by calling the helper function `getSpatialValues` (Algorithm 1), with those input graphs $\mathcal{G}^I_{A(f_i m_k)}$ and $\mathcal{G}^I_{A(lm_j)}$ (Line 14). We call the helper function `getSpatialValues` (Algorithm 1) twice, with the input graphs $\mathcal{G}^I_{A(f_i m_k)}$ and $\mathcal{G}^I_{A(lm_j)}$, where the outputs of the each (helper) function call are assigned to the temporary sets $V_{s(m_k)}$ and $V_{s(a_i)}$ correspondingly (Line 14). If these sets are not empty (Line 15), it means that `getSpatialValues` identified spatial values in the triple patterns of the input graphs.

Then, we iterate through the spatial value pairs retrieved from the each of the sets (Line 16). In this loop, we call the next helper function `relateSpatialValues` (Algorithm 2), where the input is a spatial value pair. The output value of this function is the topological relation between the corresponding level members, and it is assigned to the initially created temporary variable $\texttt{topoRel}_i$ (Line 17). If this $\texttt{topoRel}_i$ value is not null (Line 18), the output graph for the spatial fact members is incrementally enriched by adding the triple pattern with the topological relation (Line 19).

To find the topological relations between the observation facts and parent level members in an N:M cardinality relation, we check in Line 20 that if level $\mathrm{id}^S(l_n)$ is assigned as a parent level in a hierarchy step with `qb4o:parentLevel` predicate and the hierarchy step entails an N:M relation with `qb4o:ManyToMany` predicate. If that is the case, we repeat the same steps from Lines 8 to 19.

Finally, the output graph for the spatial fact members with *discovered* fact-level (topological) relations is returned in Line 22.

4.2.3. *Defining spatial fact DSD*

In this section, we present an algorithm for redefining the fact schema data structure definition (DSD) by enriching the DSD with fact-level topological relations. An example of a fact schema in QB4OLAP is given in the black-colored lines of Listing 3 (for now please ignore Lines 3, 5 and 7). We re-define the spatial fact schema to QB4SOLAP (Listing 3 Lines 1–7) by using the enriched fact members that are generated via Algorithms 5 and 6.

Algorithm 7 (`defineSpatialFactDSD`) The input variables for Algorithm 7 are the instance graph of spatial fact members $\mathcal{G}^I_{\mathrm{FM}(F_s)}$ and schema graph of QB4OLAP fact schema $\mathcal{G}^S_F$. Spatial fact members in the instance graph $\mathcal{G}^I_{\mathrm{FM}(F_s)}$ must be anno-

Algorithm 7: defineSpatialFactDSD($\mathcal{G}^I_{\mathrm{FM}(F_s)}, \mathcal{G}^S_F$) : $\mathcal{G}^S_{F_s}$

Input: $\mathcal{G}^I_{\mathrm{FM}(F_s)}, \mathcal{G}^S_F$
Output: $\mathcal{G}^S_{F_s}$

1 **begin**
2 $\mathcal{G}^S_{F_s} = \mathcal{G}^S_F$; `aggFunc`$_i$ = *null*; /*initalize the output graph and temporary variable*/
3 **foreach** ($\mathrm{id}^I(f_i)$ `rdf:type qb:Observation`) $\in \mathcal{G}^I_{\mathrm{FM}(F_s)}$ **do**
4 **foreach** ($\mathrm{id}^I(f_i)$ `topoRel`$_i$ $\mathrm{id}^I(lm_j)$) $\in \mathcal{G}^I_{\mathrm{FM}(F_s)} \mid \bigcup_{l_n \in L(f_i)} (\mathrm{id}^I(f_i)\ \mathrm{id}^S(l_n)\ \mathrm{id}^I(lm_j)) \in \mathcal{G}^I_{\mathrm{FM}(F_s)}$
/*each `topoRel`$_i$ in the fact member triples goes into the DSD with its corresponding level l_n*/ **do**
5 $\mathcal{G}^S_{F(F_s)} \cup =$
{($\mathrm{id}^S(F)$ `qb:component [qb4o:level` $\mathrm{id}^S(l_n)$, `qb4so:topologicalRelation` id^S(`topoRel`$_i$)])};
6 **foreach** $v_{m_k} \in (\mathrm{id}^I(f_i)\ \mathrm{id}^S(m_k)\ v_{m_k})$ /*find the spatial measures from the fact triples*/ **do**
7 **if** v_{m_k} *is a* `geo:spatialLiteral` **then**
8 **switch** *(geoType(v_{m_k}))*
/*$geoType(v_a)$ function returns the geometry type of a given attribute value*/ **do**
9 **case** (`POINT`)
/*point geometry measures are supported to be aggregated with ConvexHull function*/
10 `aggFunc`$_i$ = `qb4so:ConvexHull`
11 **case** (`LINE`)
/*line geometry measures are supported to be aggregated with Union function*/
12 `aggFunc`$_i$ = `qb4so:Union`
13 **case** (`POLYGON`)
/*polygon geometry measures are supported to be aggregated with Union, Centroid,*/
14 `aggFunc`$_i$ = `qb4so:Union` $\vee$ `qb4so:Centroid` $\vee$ `qb4so:MBR`
/*or MBR functions*/
15 $\mathcal{G}^S_{F(F_s)} \cup =$
{($\mathrm{id}^S(F)$ `qb:component [qb:measure` $\mathrm{id}^S(m_k)$, `qb4o:aggregateFunction` id^S(`aggFunc`$_i$)])};
16 **return** $\mathcal{G}^S_{F_s}$

tated with QB4SOLAP or can be generated by using Algorithms 5 and 6 from QB4OLAP fact members. A QB4OLAP fact schema $\mathcal{G}^S_F$ has (base) levels and measures of the cube as `qb:components` and defines the fact-level cardinality relation with `qb4o:cardinality` predicate, aggregate functions on (numerical) measures with `qb4o:aggregateFunction` predicate.[11]

[11] In QB4OLAP, `qb4o:AggregateFunction` class has only instances (e.g., `qb4o:Avg`, `qb4o:Sum` functions) for numerical measures. QB4SOLAP extends this class with a subclass `qb4so:SpatialAggregateFunction`, which has instances of spatial aggregate functions (e.g., `qb4so:ConvexHull`, `qb4so:Union`) for spatial measures [13,15].

The output of Algorithm 7 is the enriched fact schema graph $\mathcal{G}^S_F$ annotating the fact-level relations with QB4SOLAP topological relations and measures with spatial aggregate functions.

In Line 2, we initialize the output graph as the input schema graph so that we can gradually enrich it with QB4SOLAP schema annotations (Lines 5 and 15). Initially, an aggregate function variable `aggFunc`$_i$ is created and set to *null* (Line 2).

The first `foreach` loop iterates through the fact members graph $\mathcal{G}^I_{\mathrm{FM}(F_s)}$ and finds each fact member f_i by using the triple pattern ($\mathrm{id}^I(f_i)$ `rdf:type qb:Observation`). The second `foreach` loop gets every *distinct* topological relation `topoRel`$_i$ of the fact member f_i (Line 4). Then the output schema is annotated with the identifier of these topological relations (Line 5). Next, we get every measure v_{m_k} of the fact member f_i (Line 6), and check if it is a spatial measure (Line 7). If it is a spatial measure, we find the geometry type with *geoType* function (Line 8). We have appointed the corresponding spatial aggregate functions (Lines 10, 12, and 14) with regard to the geometry type of the spatial measure (Lines 9, 11, and 13). Finally, the output schema $\mathcal{G}^S_{F_s}$ is annotated with the identifier of these spatial aggregate functions (Line 15) and returned (Line 16).

4.3. Implementation choices

When implementing the algorithms presented in this section, we had to make some implementation choices both in technical as well as strategical aspects that we would like to briefly comment on. Further details regarding the implementation of the algorithms themselves are available in the Appendix. As mentioned earlier, RDF2SOLAP is implemented in Javascript on the Node.js platform using the N3.js library for parsing the RDF triples in Javascript and the Turfjs library for spatial analysis. Details of our approach, endpoints, and datasets can be found on our project page.[12] The code repository for the whole implementation can be found on GitHub.[13]

To answer the question: "*Can this approach be reasonably implemented on top of triple stores by directly using Web and Semantic Web technologies?*", we have come across a number of challenges, where specific choices had to be made.

For example, we chose to store RDF data in a well-established triple store (Virtuoso Open Source) that supports many geometry data types (i.e., POLYGON, MULTIPOLYGON). Even though Virtuoso supports several shape types (e.g., POLYGON, MULTIPOLYGON, etc.), it has a limited number of spatial Boolean functions available as built-in functions from the DE9DIM model (see Table 1). Therefore, we have also decided to use a third party *Javascript library* for spatial analysis, which is called *Turfjs*[6]. This way, we can ensure that RDF2SOLAP can be used on top of any triple store since the Javascript library provides us with the spatial analysis capabilities and a flexible development environment, independent from the choice of the triple store.

[12]Project Page: http://extbi.cs.aau.dk/RDF2SOLAP.

[13]RDF2SOLAP Repository: https://github.com/lopno/rdf2solap.

We are working with multi-part POLYGON data (for drainage areas and parishes), which means that, when several polygons are grouped by unique (parish or water) URIs they can compose a MULTIPOLYGON for a single parish or drainage area instance. From the implementation point of view, we had to implement a bounding box function for multi-part POLYGON data, in order to call the spatial Boolean functions (within and intersects) between the correct parish and drainage area instances, then annotate the topological relations between their unique URIs. If triple stores already provided overall support of complex spatial data types, spatial indices, and a complete support of built-in spatial functions, decoupling the triple stores during development of RDF2SOLAP would not have been necessary. We could then directly have used the spatial capabilities of the triple stores that were required for developing RDF2SOLAP. However, to the best of our knowledge, a third party spatial analysis library was needed to fully implement our RDF2SOLAP (spatial) multi-dimensional enrichment algorithms described in Section 4.

5. Experimental evaluation

The section is structured as follows. We describe experimental settings in Section 5.1. Then we compare development time between our approach and the baseline, followed by a comparison of the runtimes and the annotation quality. Finally, we summarize our findings in Section 5.5.

5.1. Experimental setup

The rationale for developing RDF2SOLAP is to be able to *enrich and annotate existing spatial RDF data with spatial and multi-dimensional metadata while staying within the RDF environment*. This upgrades the spatial RDF data to allow SOLAP querying directly in SPARQL. The alternative would be to export the spatial RDF data to relational format, do the enrichment with relational/GIS tools and perform the SOLAP on the resulting relational data, thus loosing all the advantages of having the data in RDF in the first place. Doing the enrichment purely within the RDF environment is expected to come at a cost as support for spatial and multidimensional data in the RDF/SW stack is still less mature; this will however improve over time. Thus, our goal is just to demonstrate that we can do this in a pure RDF environment with *adequate performance* in terms of runtime and annotation quality (which may vary). In return, RDF2SOLAP provides a solution that is both flexible and *general* for all data sets. We then compare our general solution to the alternative baseline, which is spending long development times on hand-crafting specialized enrichment solutions using RDBMSes and GIS for each new data set.

We used the common *Virtuoso version 07.20.3217 on Linux (x86_64-ubuntu-linux-gnu), Single Server Edition* as triplestore. We implemented RDF2SOLAP on the *Node.js* platform, running on a Macbook Pro 14.3 with one Intel Core i7 2.8GHz 4-core CPU 256KB L2 cache, 6MB L3 cache, and 16GB RAM. All test cases are in the GitHub repository so the experiments can be repeated. Each experiment was run in a single process. For the baseline implementation, we used a leading GIS tool and a leading RDBMS (we cannot write the names due to license restrictions, but they can be supplied on demand). These were running on a Windows 10 Enterprise server with 4 Intel Core i7 2.9GHz CPUs and 32GB memory, i.e., *considerably more powerful hardware*. We use both the GeoFarmHerdState data set described above and the GeoNorthwind data set from [15].

We now describe how the RDBMS and GIS baselines were implemented. Since the GIS tool and the RDBMS cannot process RDF data in native format, we first have to extract and prepare the data for loading into them. The preparation time is part of the development time discussed below. Doing this preparation requires that the developer has basic knowledge of the domain, extraction of RDF data with SPARQL queries, writing SQL queries, and knows how to use the RDBMS and GIS tools. We extracted the spatial level members (farms states, parishes, and drainage areas) from our RDF endpoint in CSV format. To load the data into the GIS tool and RDBMS we use the relational schema defined by QB4SOLAP. In the GIS tool we saved CSV data layers (for each level; farm states, parishes and drainage areas) and converted these into native GIS format (shape files). Then, we run the *Join Attributes By Location* function, which is a built-in data management function. We run this function as a batch process, for parishes-drainage areas (Alg. 4), farm states-parishes, and farm states-drainage areas (Alg. 6). We load the WKT data (spatial attributes of level and fact members) in the CSV files into the GIS tool and a relational geo-database, with the same decimal precision for the coordinates. We extract topological relations between the child and parent members by using spatial joins in the GIS tool and built-in spatial functions in the RDBMS. Overall, most of the time was spent on data extraction, preparation, and load, caused by having to convert data from its existing RDF format. None of these tasks are needed if the enrichment is done entirely within an RDF environment, like in RDF2SOLAP.

5.2. *Development time comparison*

We now compare the time required to develop enrichment solutions for spatial RDF data with RDF2SOLAP to using the RDBMS and GIS baselines. Here, it is important to keep in mind that RDF2SOLAP has *general* algorithms that use existing metadata and annotations to work for *any* spatial RDF data set, requiring only *a few minutes of configuration*, while a new baseline enrichment solution has to be implemented for *each* new data set, requiring literally *(repeated) days of development time*. Of course, there was a *onetime development cost for RDF2SOLAP*. However, this cost is already paid and will not be repeated, unlike the case for the baselines.

The development times for RDF2SOLAP, and the RDBMS and GIS baselines for the one-time step of General Algorithms (only RDF2SOLAP) and the GeoFarmHerdState data set, are given in Table 2. One day corresponds to 8 hours. All development was carried out by the first and fourth author who both have significant experience in all the used tools and platforms, and also recorded the development times for RDF2SOLAP and GeoFarmHerdState. The RDF2SOLAP configuration times for each data set were also recorded by the first author. We find these development times realistic and comparable. As GeoNorthWind is only included to demonstrate that RDF2SOLAP can generalize to other data sets, we have not implemented the baseline enrichment solutions for GeoNorthWind. Thus, we have not reported RDBMS and GIS development times in the table. However, realistic estimates would be in the same range as for GeoFarmHerdState, i.e., one or more days.

From the table, we can see that RDF2SOLAP allows to enrich and annotate new spatial RDF data sets with just minutes of effort, since the enrichment process is

Table 2
Development times

	RDF2SOLAP	RDBMS	GIS
General algorithms	1.4 days (one-time)	N/A	N/A
GeoFarmHerdState	5 min (config)	1.3 days	2 days
GeoNorthWind	10 min (config)	N/A	N/A

Table 3
Input, output, and runtimes for RDF2SOLAP algorithms on GeoFarmHerdState

		INPUT			OUTPUT		
		NumberOf child members	NumberOf parent members	NumberOf explicit relations	NumberOf topological relations		Run times (in seconds)
Section 5.2	Alg. 3	parishes: 2,180	drainageAreas: 134	2,683	intersects	636	29 s
					within	2,046	
	Alg. 5	farmStates: 40,039	parishes: 2,180	39,800	within	39,334	7 s
Section 5.3	Alg. 4	parishes: 2,180	drainageAreas: 134	NONE	intersects	1,088	2,622 s
					within	3,392	
	Alg. 6	farmStates: 40,039	parishes: 2,180	NONE	within	39,998	1,920 s
		farmStates: 40,039	drainageAreas: 134	NONE	within	39,845	525 s

done automatically after retrieving the input parameters to the enrichment algorithms from the endpoint. In comparison, *each* new data set requires one or more days of (repeated) development effort for the baseline RDBMS and GIS enrichments.

5.3. *Runtime comparison*

Having established that RDF2SOLAP requires up to 2 orders of magnitude less development time for a new data set, we now investigate whether it can do the enrichment with reasonable runtimes, and compare its runtimes to those of the baseline implementations. Both the total runtimes (in minutes) and the subtotals (in seconds, for accuracy) for the different algorithms are reported. The GeoFarmHerdState runtimes are seen in Tables 3 and 4.

Alg. 3 and 5 (to detect explicit topological relations) are only implemented in the RDBMS since the GIS tool does not support the foreign key joins of explicit (skos:broader) relations which are needed for these two algorithms. In order to implement the needed topological relations in RDF2SOLAP, we had to use theturf.js library running on a node.js server, where the RDF data is parsed into JSON format, since these relations were not supported by the triplestore. This meant that we could not take advantage of spatial indexing in the triplestore.

To understand the size of the input and output of the algorithms for GeoFarmHerdState, we report these along with corresponding RDF2SOLAP runtimes in Table 3. The input parameters and numbers for each algorithm are shown in Table 3 under the INPUT column(s). The input datasets to the algorithms are 2,180 parish members, 40,039 farm state members, and 134 drainage area members. The OUTPUT columns show the number of topological relations found and run times of the algorithms. In this section, we only focus on the runtime, the annotation quality is evaluated later.

Table 4
GeoFarmHerdState runtimes (f.s. = farm states, p. = parishes, d.a. = drainage areas)

	RDF2SOLAP	RDBMS	GIS
Alg. 3 (p. – d.a.)	29 s	<1 s	N/A
Alg. 4 (p. – d.a.)	2,622 s	43 s	45 s
Alg. 5 (f.s. – p.)	7 s	<1 s	N/A
Alg. 6 (f.s. – p.)	1,920 s	95 s	72 s
Alg. 6 (f.s. – d.a.)	525 s	48 s	41 s
Total	**85 m**	**3 m**	**>2.5 m**

In Table 3, we can see that most expensive algorithm is Alg. 4 (`discoverSpatialHS`), which runs in 2,622 seconds. The algorithm takes parishes and drainage areas (POLYGON data type) as input instances, and not explicit relations as in Alg. 3 (`detecSpatialHS`). Alg. 3, given (2,683) distinct explicit relations, checks the corresponding spatial Boolean functions (*within* and *intersects*) 2,683 times each. In comparison, Alg. 4 calls (*within* and *intersects*) 134 $\times$ 2,180 = 292,120 times each. Similarly, Alg. 6 is slower than Alg. 5 since the former does not use explicit relations. However, it is much faster than Alg. 4 since this calls the spatial Boolean functions between farm states (POINT data type) and parishes and drainage areas (POLYGON data type).

From the GeoFarmHerdState runtimes, we see that Alg. 4 and 6 use the most time, in particular for RDF2SOLAP. However, the total RDF2SOLAP runtime of 85 minutes is very reasonable and well within the requirements for practical use: a user wanting to analyze a new data set (which usually does not happen several times a day) can simply spend a few minutes on configuration and then let RDF2SOLAP run in the background for the next 1.5 hours. Especially for non-developer RDF users, this is a much better value

Table 5
GeoNorthWind runtimes

Hierarchy step	Algorithm	RDF2SOLAP
Customer-city (point-polygon)	Alg. 3	2 s
	Alg. 4	7 s
Supplier-city (point-polygon)	Alg. 3	<1 s
	Alg. 4	<2 s
City-state (polygon-polygon)	Alg. 3	1 s
	Alg. 4	13 s
Total		**26 s**

Table 6
Number of topological relations found by each tool (f.s. = farm states, p. = parishes, d.a. = drainage areas)

		Tools		
		GIS	RDBMS	RDF2SOLAP
Alg. 3: (p. – d.a.)	*intersects*	N/A	1,897	636
	within	N/A	785	2,046
Alg. 4: (p. – d.a.)	*intersects*	2,556	2,802	1,088
	within	1,039	785	3,392
Alg. 5: (f.s. – p.)	*within*	N/A	39,334	39,334
Alg. 6: (f.s. – p.)	*within*	39,805	39,984	39,998
Alg. 6: (f.s. – d.a.)	*within*	39,441	39,845	39,845

proposition than first spending one or more days on technical development in the baseline tools.

For GeoNorthWind, the baselines were not implemented (see above) so only RDF2SOLAP runtimes are reported, see Table 5. For this smaller data set, RDF2SOLAP completes in just 26 seconds, making it usable even in interactive mode.

In summary, the runtime comparisons show that, even though RDF2SOLAP is slower than the handcrafted baselines, it still has a runtime performance that is more than adequate for its intended use case.

5.4. *Annotation quality comparison*

We now compare RDF2SOLAP to the RDBMS and GIS baseline tools in terms of annotation quality. Specifically, we report the number of the topological relations found by each algorithm/step in each tool, and relate these to accuracy and coverage. The numbers are given in Table 6.

As mentioned earlier, Alg. 3 and Alg. 5 were not implemented in the GIS tool due to its lack of support for explicit relations between parent-child members; thus these numbers are reported as N/A. We thus only tested the discovery of implicit topological relations (`discoverSpatialHS` and `discoverFactLevelRelations`) by utilizing its spatial join functionality to emulate the results for Alg. 4 and Alg. 6.

For the RDBMS tool, we tested both detect and discover topological relations, where we used joins on unique IDs if they were present (drainage area foreign key in parishes, parish foreign key in farm states), and with spatial joins by using the `STWithin`, `STIntersects`, and `STOverlaps` built-in functions.

We now compare results for each algorithm in the different tools. For Alg. 3, RDF2SOLAP reports only a third of the intersecs relations but almost three times as many within relations, as the RDBMS tool. This is due to generalizing the multi-part POLYGON data as bounding boxes in RDF2SOLAP (due to restrictions in our spatial library), in the spatial RDBMS, multi-part POLYGON data is processed in its original format, yielding better quality. Interestingly, the total number of intersects+within relations for the two tools are exactly the same, namely 2682. This suggest that RDF2SOLAP can get the same annotation quality as the RDBMS tool when better spatial support become available in the RDF environment.

Similar results are seen for Alg. 4. Here, the GIS and RDBMS results are similar, but not identical, showing that perfect annotation quality is not a given, even with traditional tools. Again, RDF2SOLAP finds fever intersects relations and more within relations (again due to the bounding box generalization), and again the total number of relations found by the 3 tools are very similar. This indicates that RDF2SOLAP can achieve the same annotation quality with better spatial RDF support.

For Alg. 5, the RDF2SOLAP and RDBMS results are identical. This perfect annotation quality can be achieved since the within relations are found between points and polygons which can be done exactly by the library.

For Alg. 6, the results found by the 3 tools for (farm states-parishes) are very close, with RDF2SOLAP differing 0.5% from the GIS tool and 0.04% from the RDBMS tool. For (farm states-drainage areas), the RDF2SOLAP and RDBMS results are identical, while the GIS result differs by 0.01%. Thus, the annotation quality for all 3 tools is near-perfect.

Similar results were found for GeoNorthWind. For Hierarchy Step 1, there are 89 correct within relations. Of these, Alg. 3 found 75 of them correctly, while Alg. 4 found 91 relations (the 89 correct and 2 extra incorrect). For Hierarchy Step 2 there are 28 correct within relations: Alg. 3 found 24 of them correctly, while Alg. 4 found all 28 of them correctly. For Hierarchy Step 3, we do not have the correct ground truth since this requires the GIS and RDBMS baselines to have been implemented.

RDF2SOLAP's problems with POLYGON-POLYGON relations could have been prevented if we had been able to use multi-part POLYGON and MULTIPOLYGON data in its original form instead of generalizing them to bounding boxes. However, We encountered both performance and formatting problems while loading MULTIPOLYGON data to Virtuoso, which led to missing data in the triplestore for drainage areas. Even if the MULTIPOLYGON data was could be successfully loaded, Turf.js is not able to handle POLYGON-MULTIPOLYGON *within* relations. We thus had to make a trade-off and implement the POLYGON-POLYGON relations with generalized bounding boxes.

In summary, the annotation quality of RDF2SOLAP is near-perfect for the spatial relations that are well supported in the RDF environment. There are some problems with polygon-to-polygon relations, but these are caused by limitations in our spatial library. When better spatial RDF support becomes available, we are confident that RDF2SOLAP will provide near-perfect annotation quality for all cases.

5.5. *Experimental summary*

In summary, we have seen that RDF2SOLAP provides orders of magnitude less development time for new data sets (minutes versus days), and, while slower than the RDBMS and GIS tools, has adequate runtimes for its intended use case. For some algorithms, its annotation quality is near-perfect, while for others, it will be when better spatial RDF support becomes available.

6. Related work

Utilizing DW/OLAP technologies on the Semantic Web with RDF data makes RDF data sources more easily available for interactive analysis. As summarized by Abelló et al. [1], related work has studied OLAP and data warehousing possibilities on the Semantic Web (SW) in general. Our work, however, is centered around spatial OLAP (SOLAP) and spatial data warehouses (SDW) on the Semantic Web, which is not yet a comprehensively studied research topic. We focus on performing spatial OLAP analysis directly over multi-dimensional data published on the Semantic Web. Therefore, we review the related work with relevant approaches classified under the following titles: (1) *data modeling and representation* (on the SW for multi-dimensional and spatial data), (2) *metadata enrichment and MD analysis* (OLAP-like analysis over RDF data).

Data modeling and representation The RDF Data Cube (QB) vocabulary [48] is the W3C recommendation to publish statistical data and its metadata in RDF. Thus, QB is commonly used to publish raw or already aggregated multidimensional data sets. However, QB lacks the underlying metadata for multidimensional models and OLAP operations. The set of MD concepts, such as, hierarchy levels along a cube dimension, semantics of the relationships between levels, semantics and definitions of aggregate functions are missing in QB vocabulary, are essential in an MD schema to enable OLAP analysis. Therefore, Kämpgen et al. define an OLAP data model on top of QB by using SKOS [30] extensions[14] to support multidimensional hierarchies [26,27]. However, the proposed model has some limitations on levels to exist only in one hierarchy. The OLAP operations are made available on the data cubes with the proposed model but restricting the cubes with only one hierarchy per dimension. Etcheverry et al. propose QB4OLAP [7] as an extension to the QB vocabulary, which supports modeling a complete MD data cube and querying the cube with OLAP operations on the Semantic Web. Modeling of MD data on the Semantic Web motivated the publication of datasets from several domains (e.g., statistical data sets from EuroStat and World Bank data, AirBase air quality data, and many other environmental and governmental open data) as RDF data cubes [47].

The need of fully multi-dimensional semantic data warehouses (where OLAP operations are enabled in SPARQL) made the QB4OLAP vocabulary prominent. Therefore, RDF data cubes from statistical and environmental domains [10,12,43] are published with an extended QB vocabulary. Moreover, semantic Extract-Transform-Load (ETL) tools automate and ease the process of annotating and publishing open data with QB4OLAP on the Semantic Web [5,31,32]. Therefore, we can see more and more multi-dimensional datasets annotated with QB4OLAP on the Semantic Web.

These multi-dimensional semantic modeling approaches and querying with OLAP on the Semantic Web lead us to find ways for modeling, publish-

[14] http://www.w3.org/2011/gld/wiki/ISO_Extensions_to_SKOS

ing, and querying *spatial* data warehouses in particular since modeling and querying *spatial* data bring new challenges. QB4SOLAP [13] – a spatial extension to a fully multi-dimensional QB4OLAP vocabulary emerges the need of modeling and publishing geo-semantic data warehouses on the Semantic Web.

Modeling and publishing (non multi-dimensional) spatial data on the Semantic Web has been a focus by many communities and research groups. Some of the efforts for standardizing and aligning vocabularies to describe spatial data (e.g., locations, geometries, etc.) are GeoSPARQL [34] by the Open Geospatial Consortium (OGC), Basic Geo (WGS84 lat/long) Vocabulary by W3C Semantic Web Interest Group [4], NeoGeo Vocabularies by GeoVocab working group [40], INSPIRE Directive metadata on the Semantic Web [35], and GeoNames Ontology [45] among many others.

These standards have been commonly used in a wide range of projects. Government Linked Data (GLD) working group listed some of these geo-vocabularies as standards to publish governmental linked data sets [20]. Andersen et al. re-use some of these vocabularies for publishing governmental and spatial data on the Semantic Web [2]. LinkedGeoData is a big contribution to the Semantic Web, which interactively transforms OpenStreetMap data to RDF data [41]. The GeoKnow project focuses on linking geospatial data from heterogeneous sources [39]. More recent works by Kyzirakos et al. to transform geospatial data into RDF graphs using R2RML mappings [28] and geo-semantic labelling of open data [33] by Neumaier et al. show that spatial data on the Semantic Web will keep growing. However, none of these standards considers the MD aspects of spatial data for geo-semantic data warehouses.

Large volumes of spatial data on the Semantic Web yield a need for advanced modeling and analysis of such data. As mentioned earlier, QB4SOLAP [13] remedies this need. Aggregate functions, cardinality relationships, and topological relations are rich sources of knowledge in spatial data cubes in order to query with spatial OLAP operations in SPARQL [15].

QB4ST [3] is a recent attempt to define extensions for spatio-temporal components to RDF Data Cube (QB). However, it has the inherent limitations of QB to support OLAP dimensions with hierarchies, levels, and aggregate functions. Lack of OLAP hierarchies and aggregate functions in QB4ST hinders to define and operate with topological relations at hierarchy steps or spatial aggregate functions on spatial measures, which are essential MD concepts for SOLAP operators. These spatial MD concepts in geo-semantic data warehouses are defined together with SOLAP to SPARQL query mappings in [15].

Metadata enrichment and MD analysis Increasing popularity of RDF data cubes and MD OLAP cubes on the Semantic Web raised interest in tools and frameworks that can ease the annotation and querying of MD data on the Semantic Web from existing RDF sources.

Ibragimov et al. present a framework for exploratory OLAP over Linked Open Data (LOD), where the MD schema of the data cube is annotated with QB4OLAP [22]. Based on this MD schema, they propose a system that is capable of querying data sources, extracting and aggregating data to build OLAP cubes in RDF [23] and querying in a federated setup [21]. Similarly, Gallinucci et al. propose an exploratory OLAP approach, namely iMOLD by interactively MD modeling of linked data [11]. Their approach allows users to enrich RDF cubes with aggregation hierarchies through a user-guided process. During this interactive process, the recurring modeling patterns that express roll-up relationships between RDF concepts are recognized in the LOD, then these patterns are translated into aggregation hierarchies to enrich the RDF cube. Varga et al. enables OLAP analysis with the QB2OLAP tool in [43] over statistical data published with QB vocabulary, by applying dimensional enrichment steps described thoroughly in [44]. The proposed enrichment steps allow users to enrich a QB dataset with QB4OLAP concepts such as fully-fledged dimension hierarchies. However, none of these frameworks and approaches supports spatial data warehouses and SOLAP operations.

In this paper, we propose a framework, where OLAP cubes in RDF can be enriched with spatial MD concepts from the *QB4SOLAP* vocabulary by employing RDF2SOLAP enrichment algorithms over QB4OLAP triples. This allows users to query MD cubes with SOLAP operators in SPARQL. Optionally, users can utilize GeoSemOLAP [14] tool on top of QB4SOLAP data sets, which helps users formulate SOLAP queries in SPARQL.

7. Conclusion and future work

Motivated by the need to conciliate MD/OLAP RDF data cubes and spatial data on the Semantic Web as geo-semantic data warehouses, we have presented a number of contributions in this paper. As a first attempt to enrich RDF data cubes with spatial concepts, we have shown that the QB4SOLAP vocabulary yields the need for fully-fledged spatial data warehouse concepts (that is built on top of non-spatial QB4OLAP and RDF Data Cube (QB) vocabularies), by demonstrating the running use case examples from real world governmental open data sets from various domains (i.e., environment, farming) with complex geometry types. We introduced running use case examples annotated both with QB4OLAP and QB4SOLAP vocabularies, in RDF triples and formalized the RDF triples as parameters to use in the enrichment algorithms. Second, we have built our conceptual architecture in relation to existing semantic (spatial) OLAP tools (e.g., on top of the QB2OLAPem enrichment module and at the back-end of GeoSemOLAP). Third, we have provided hierarchical enrichment algorithms for two cases that cover finding explicit hierarchy steps with direct links between the level members and finding implicit hierarchy steps (without direct links between the level members) by comparing geometry attributes of the level members. We have defined and deployed the necessary algorithms as spatial helper functions for finding spatial attributes and comparing these attributes to derive topological relations. Fourth, we have presented the factual enrichment phase for both implicit and explicit fact-level relations between the fact and level members. Moreover, we have presented how to re-define the fact schema after the factual enrichment phase in an automated manner. Re-defining the fact schema also includes finding the spatial measures and associating them with spatial aggregate functions. In the end, we have implemented all the algorithms that are designed for both hierarchical enrichment and factual enrichment processes, then we presented the details of our implementation.

Finally, we have evaluated our approach and its accuracy as well as the implementation with the underlying technologies by comparing the number of topological relations found in the RDF2SOLAP framework (between the level members in spatial hierarchies and between the level members and the fact members, respectively, during the hierarchical enrichment phase and the factual enrichment phase) against two different non-SW environments. We have presented the experimental set-up and our comparison baselines and concluded our evaluation with technical lessons learned.

In conclusion, RDF2SOLAP facilitates the spatial enrichment of RDF data cubes and fills an important gap in our vision of SOLAP on the Semantic Web despite of the challenges and restrictions in supporting complex spatial data types with the current state of the most common triple stores [18,19,24].

Several directions are interesting for future research: creating a comprehensive benchmark by implementing the RDF2SOLAP enrichment algorithms on different platforms and testing on different use cases, deriving spatial hierarchy levels and level member instances from external geo-vocabularies and extending our approach in QB4SOLAP, GeoSemOLAP and RDF2SOLAP to handle highly dynamic spatio-temporal data and multi-dimensional analytical queries [25]. Another line of future work would be runtime optimizations for scalable querying of spatial data warehouses [9] exploiting specifics of Linked Data Management [16,17]. Moreover, it is also important to develop query optimization techniques for OLAP queries on semantic DW/RDF data, similar to the ones developed for cubes and XML data [36,37,49]. Furthermore, to achieve scalable querying and runtime optimization, new research directions can be taken with binary serialization of the QB4SOLAP RDF data such as header dictionary triples (HDT), which is a compact data structure that can be compressed and kept in-memory, thus it enables high performance (and also concurrent) querying.

Acknowledgements

This research is partially funded by the European Commission through the Erasmus Mundus Joint Doctorate Information Technologies for Business Intelligence (EM IT4BI-DC), the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B, Aalborg University's Talent Programme, and the Poul Due Jensen Foundation.

Appendix. Implementation details

In this section, first we provide the details on how the algorithms from Section 4 are implemented to generate spatially enriched RDF triples with QB4SOLAP (Sections A.1, A.2, A.3, and A.4). Afterwards, we discuss our implementation choices in Section 4.3 and present the results of applying the algorithms on the use case data (GeoFarmHerdState) in Section 5 (Table 3).

A.1. QB4SOLAP triples generation

To implement the algorithms from Section 4, we have chosen a use case data set that can be annotated with multi-dimensional concepts in QB4OLAP and has the required spatial properties to be enriched as a fully spatial multidimensional cube with QB4SOLAP. The required spatial properties are: 1) Level members in a (spatial) hierarchy must have spatial attributes, where the geometry of the attributes should be different than only a simple *point* geometry type, e.g., *polygon, line*, etc. Thus we can implement the hierarchical enrichment (Section 4.1). 2) Fact members should have spatial measures, thus we can implement the factual enrichment (Section 4.2).

Therefore, we have chosen GeoFarmHerdState as use case, which we have already used as running example throughout the paper. In Section 2, we discussed the spatial multi-dimensional concepts of the GeoFarmHerdState data cube and in Section 4 we provided RDF triple snippet examples of those concepts: *(a)* spatial hierarchy structure with QB4SOLAP (Listing 1), *(b)* level members annotated with QB4OLAP and with QB4SOLAP after hierarchical enrichment (Listing 2), *(c)* spatial fact schema (Listing 3), and *(d)* spatial fact members with spatial measures (Listing 4). A full overview of the GeoFarmHerdState cube with spatial and non-spatial dimensions can be found in our previous work [12] and on our project website http://extbi.cs.aau.dk/GeoFarmHerdState/.

Note that we use the non-spatial annotation of the GeoFarmHerdState data cube with QB4OLAP as an input to our algorithms, which is publicly available from our SPARQL endpoint[15] with corresponding namespaces for schema data triples[16] and instance data triples.[17]

We query the endpoint and extract RDF data in JSON format as an input to our implementation of the four main enrichment algorithms; Algorithm 3 – `detectSpatialHS`, Algorithm 4 – `discoverSpatialHS`, Algorithm 5 – `detectFactLevel`, and Algorithm 6 – `discoverFactLevel`.

In the following, we show the implementation highlights of each algorithm and helper function along with code snippets.

A.2. Detecting explicit topological relations

Detecting explicit topological relations are addressed in the following algorithms: Algorithm 3 – `detectSpatialHS` and Algorithm 5 – `detectFactLevel`. In both cases the source data has explicitly defined roll-up relations, which means there is a direct relation between level members with `skos:broader` for hierarchy steps (e.g., Listing 2, Line 7) and there is a direct relation between a fact member and a base level member's foreign key URI (e.g., Listing 4, Line 2)

The input variables for Algorithm 3 – `detectSpatialHS` are the triples with roll-up relations of the hierarchy steps ($\mathcal{G}_R^I U(hs)$) and the attributes of level members ($\mathcal{G}_A^I(lm)$) from the instance data graph. Explicit `skos:broader` relations are annotated in the instance graph of hierarchy steps ($\mathcal{G}_R^I U(hs)$). Therefore, we query the endpoint by filtering with the explicit `skos:broader` relations between all the level members. We fetch the results of the query in Node.js in JSON format.

The input variables for Algorithm 5 – `detectFactLevel` are the triples with fact members ($\mathcal{G}_F^I M(F)$) and the attributes of level members ($\mathcal{G}_A^I(lm)$) from the instance data graph. Explicit fact-level relations (by referring to the foreign key URI of level members) are annotated in the instance graph of fact members ($\mathcal{G}_F^I M(F)$). Therefore, we query the endpoint with all the fact members and the corresponding attributes of level members. We fetch the results of the query in Node.js in JSON format.

Initially, we need to provide the explicit (roll-up) relations between the level members and fact-level members to implement Algorithms 3 and 5 for detecting the (explicit) topological relations. As mentioned above, we provide these relations from the data set by querying the endpoint and fetching the results of the query in Node.js in JSON format.

The next step is to retrieve the spatial attribute and measure values from the attributes of the level members and fact members.

Retrieving attribute and measure values In this step, we retrieve the (spatial) attribute values and measure values of level members and fact members by accessing object (o) of the each triple pattern $t = (s, p, o)$ from the instance graphs of attributes of level members ($\mathcal{G}_A^I(lm)$) and fact members ($\mathcal{G}_F^I M(F)$) (Listing 5). This is followed by passing the `getLevelMemberAttributes` and `getMeasures` constants to

[15] SPARQL Endpoint: http://lod.cs.aau.dk:8890/sparql.

[16] QB4OLAP schema: http://extbi.cs.aau.dk/geofarm/qb4olap/farm-qb4olap-schema.ttl.

[17] QB4OLAP instances: http://extbi.cs.aau.dk/geofarm/qb4olap/farm-qb4olap-input.tar.gz.

```
const getLevelMemberAttributes = val =>
      val.substring (val.indexOf("(") +1,
      val.indexOf(")"));
const getMeasures = mval =>
      mval.substring (val.indexOf("(") +1,
      mval.indexOf(")"));
```

Listing 5. Get level member attributes and fact member measures

```
1  const getSpatialValues = value => {
2  const locationString =
3        getLevelMemberAttributes (value);
4  if (value.startsWith("POLYGON")) {
5        const polygons =
6              generatePolygonPoints(locationString);
7        return turf.polygon(coordinates:[polygons]); }
8  if (value.startsWith("LINE")) {
9        const lines = locationString;
10       return turf.lineString(coordinates:[lines]); }
11 if (value.startsWith("POINT")){
12       const points = locationString;
13       return turf.point(coordinates:[points]); }
14 return null; };
```

Listing 6. Filtering spatial data types

`getSpatialValues` constant[18] as explained below (*filtering spatial values*) and given in Listing 6.

Filtering spatial values Before employing spatial analysis functions, we have to filter the spatial attributes of level members and spatial measures of fact members. Spatial values are always an object (o) value in a triple pattern $t = (s, p, o)$, which is defined as spatial literals $\mathcal{L}_s$ (Section 4). Therefore, we have retrieved the attribute and measure values as objects as mentioned above.

We have shown the helper function Algorithm 1 – `getSpatialValues`, which is used in the main algorithms. We have implemented this helper function on Node.js by filtering the WKT geometries from the input JSON data as exemplified in Listing 6. We create a `locationString` constant that takes a string value from `getLevelMemberAttributes` (Line 2). The string value is the last index location of a triple pattern constructed in `getLevelMemberAttributes`.[19]

Finding topological relations Each of the four main enrichment algorithms (Algorithms 3, 4, 5, and 6) returns an instance graph of level members or fact members with topological relations annotated in QB4SOLAP. To find these topological relations we have introduced a helper function in Algorithm 2 – `relateSpatialValues`. This algorithm is implemented by using *boolean* functions (spatial predicates) from the Turf.js library for relating spatial values and finding the appropriate topological relations. The library supports the following topological relations with corresponding predicates between certain spatial data types (Table 7). A complete list of functions and details can be found online at http://turfjs.org/docs.

We grouped the available Turf.js spatial boolean functions in Table 7 under three main topological relations (EQUALS, WITHIN, INTERSECTS), with respect to the simplification rules for grouping topological relations (Section 4.1.1) and explained along with Fig. 8 and Table 1. In Table 7, Turf.js built-in functions (predicates) are shown with #boolean prefix. In parentheses, we show how we have named them in our implementation by using the corresponding built-in functions.

Listing 7 provides an overview of the implementation of the boolean functions from Table 7 that are called in the main function for relating spatial values (relateSpatialValues) shown in Listing 8. We provide examples for each of the main topological relations (EQUALS, WITHIN, INTERSECTS).

This *first* spatial boolean function in Listing 7 is `equals` (Lines 1–8), which can be between any pair of the same spatial data type (Table 7). We have grouped child level spatial (attribute) values and parent level spatial (attribute) values by their unique id (URI) for each spatial level attribute. This allows us to use *javascript array prototype (instance) methods*, e.g., `every` or `some`, where we can create our own spatial predicate `equals` with condition to satisfy that `every` (grouped) child level attribute values should be equal to `every` (grouped) parent level attribute values. This ensures the multi-point, multi-line, and multi-polygon data types can be covered in our implementation.

[18] We differentiate measure and level attribute values in seperate constants since a measure is annotated as `qb:MeasureProperty` and a level attribute is annotated as `qb4o:LevelAttribute` in the schema graph.

[19] Similarly, we create a second `locationString(2)` for spatial measure values that takes the string value from `getMeasures`, which is not repeated in Listing 6.

Table 7
Turf.js spatial Boolean functions

EQUALS	WITHIN	INTERSECTS
#booleanEqual: (`equals`) *between* POINT-POINT LINE-LINE POLYGON-POLYGON	#booleanWithin: (`within`) *between* LINE-POLYGON POLYGON-POLYGON #booleanPointInPolygon: (`within`) *between* POINT-POLYGON	#booleanCrosses: (`crosses`) *between* LINE-POLYGON #booleanOverlap: (`overlaps`) *between* POLYGON-POLYGON #booleanPointOnLine: (`intersects`) *between* POINT-POLYGON

For example, in the source data, we had multi-polygons for drainage areas, where each unique drainage areas is a multi-polygon that is composed of several polygons. To simplify we did not store multi-polygon data in RDF. Instead, we have annotated each unique drainage area as several polygons (of the multi-polygon), where each polygon of the drainage area is bound to its drainage area via unique id – URI of the drainage area. This means in the instance graph of parent level members $\mathcal{G}^I_{A(lm_p)}$ (drainage areas), there will be triple patterns $t = (s, p, o)$, where many different polygons – objects (o) have the same subject (s) – URI of a unique drainage area to represent the multi-polygon.

To handle these multi-polygons, we gather them in a bounding box by using `turf.bboxPolygon` and `turf.bbox` functions in Listing 7 (Lines 13–14). In Listing 7 (Lines 10–18) depicts how several polygons of the same parent level can be put into a bounding box, which is passed as a parameter to our *second* spatial boolean function `within`. Finally, the function returns in Lines 19–23 with condition to satisfy that `every` (grouped) child level attribute value should be within the simplified parent level polygon – `parentLevelMultipolygonBoundinxBox` (Line 23).

The *third* spatial boolean function in Listing 7 is `crosses` (Lines 24–31), where we re-use the Turf.js spatial predicate `booleanCrosses`. This function is very similar to `overlaps` in implementation. The only difference is `crosses` occurs between LINE-POLYGON, `overlaps` occurs between POLYGON-POLYGON. For both cases, the condition to satisfy is that `some` of the (grouped) child level attribute values should cross/overlap `some` of the (grouped) parent level attribute values.

```
   // equals function
1  const equals = (childLevelSpatialValues,
2  parentLevelSpatialValues) =>
3        childLevelSpatialValues.every(
4        childLevelSpatialValue =>
5              parentLevelSpatialValues.every(
6              parentLevelSpatialValue =>
7                turf.booleanEqual(childLevelSpatialValue,
8                parentLevelSpatialValue)));
   // within function (POLYGON-POLYGON)
9  const within = (childLevelSpatialValues,
10 parentLevelSpatialValues) => {
11       const parentLevelMultipolygonBoundingBox
12       = turf.bboxPolygon(
13        turf.bbox(
14         turf.multiPolygon(coordinates: [
15          parentLevelSpatialValues.map(
16          parentLevelSpatialValue =>
17           pathOr([], [0], turf.getCoords(
18           parentLevelSpatialValue)))])));
   // all child level values are within the parent level
   // polygon (simplified with bounding box)
19 return childLevelSpatialValues.every(
20 childLevelSpatialValue => {
21       return turf.booleanWithin(
22        childLevelSpatialValue,
23        parentLevelMultipolygonBoundingBox);});};
   // crosses function (LINE-POLYGON)
24 const crosses = (childLevelSpatialValues,
25 parentLevelSpatialValues) =>
26       childLevelSpatialValues.some(
27       childLevelSpatialValue =>
28             parentLevelSpatialValues.some(
29             parentLevelSpatialValue =>
30               turf.booleanCrosses(childLevelSpatialValue,
31               parentLevelSpatialValue)));
```

Listing 7. Spatial Boolean functions

Listing 8 uses our own spatial predicates (explained above) to implement the helper function Algorithm 2 – `relateSpatialValues`. Note that we have followed the simplification rules for grouping topological

```
1  const relateSpatialValues = (childLevelSpatialValues,
2          parentLevelSpatialValues) => {
3        const childLevelGeoType = pathOr(
4          null, [0, "geometry", "type"],
5          childLevelSpatialValues);
6        const parentLevelGeoType = pathOr(
7          null, [0, "geometry", "type"],
8          parentLevelSpatialValues);
9        if (childLevelGeoType === "Point" &&
10       parentLevelGeoType === "Point") {
11         if (equals(childLevelSpatialValues,
12         parentLevelSpatialValues)) {
13            return "qb4so:equals";}
14       } else if (childLevelGeoType === "Point" &&
15         parentLevelGeoType === "LineString") {
16         if (intersects(childLevelSpatialValues,
17         parentLevelSpatialValues)) {
18            return "qb4so:intersects";}
19       } else if (childLevelGeoType === "Point" &&
20         parentLevelGeoType === "Polygon") {
21         if (pointWithin(childLevelSpatialValues,
22         parentLevelSpatialValues)) {
23            return "qb4so:within";}
24       } else if (childLevelGeoType === "LineString"
25         && parentLevelGeoType === "LineString") {
26         if (crosses(childLevelSpatialValues,
27         parentLevelSpatialValues)) {
28            return "qb4so:intersects";}
29         if (overlaps(childLevelSpatialValues,
30         parentLevelSpatialValues)) {
31            return "qb4so:overlaps";}
32       } else if (childLevelGeoType === "LineString"
33         && parentLevelGeoType === "Polygon") {
34         if (within(childLevelSpatialValues,
35         parentLevelSpatialValues)) {
36            return "qb4so:within";}
37         if (crosses(childLevelSpatialValues,
38         parentLevelSpatialValues)) {
39            return "qb4so:overlaps";}
40       } else if (childLevelGeoType === "Polygon"
41         && parentLevelGeoType === "Polygon") {
42        const isWithin = within(
43         childLevelSpatialValues,
44         parentLevelSpatialValues);
45        const isOverlaps = overlaps(
46         childLevelSpatialValues,
47         parentLevelSpatialValues);
48          if (isWithin) {
49            return "qb4so:within";}
50          if (isOverlaps) {
51            return "qb4so:overlaps";}}
52 return null;};
```

Listing 8. Relating spatial values

relations (Fig. 8), aligned with switch cases for spatial data type pairs from Algorithm 2 in our implementation.

In our implementation illustrated in Listing 8, we create two functions `childLevelGeoType` (Line 3) and `parentLevelGeoType` (Line 6), which returns the geometry type of a given attribute value. This way we can implement `switch`(geoType(v_{a_c}), geoType(v_{a_p})) `cases` from Algorithm 2 – `relateSpatialValues`.

```
const detectSpatialHierarchySteps = (
  parentLevelMembers,
  childLevelMembers,
  explicitRelations) => {
 const spatialHierarchySteps =
  explicitRelations.results.bindings.map(
  binding => {
      const childLevelMemberId = binding.s.value;
      const parentLevelMemberId = binding.o.value;
      const childLevelSpatialValues = pathOr([ ],
        [childLevelMemberId],childLevelMembers
        ).map(childLevelMember =>
           utils.getSpatialValues(
           childLevelMember.value));
      const parentLevelSpatialValues = pathOr([],
        [parentLevelMemberId], parentLevelMembers
        ).map(parentLevelMember =>
           utils.getSpatialValues(
           parentLevelMember.value));
      const topoRel = utils.relateSpatialValues(
        childLevelSpatialValues,
        parentLevelSpatialValues);
      return {
           ...binding,
           p: {type: "uri", value: topoRel ||
           "skos:broader"}};
});
      return {
           ...explicitRelations,
           results: { ...explicitRelations.results,
           bindings: spatialHierarchySteps}
      };
};
```

Listing 9. Detecting topological relations (between level members)

Detecting topological relations Finally, we have implemented detecting topological relations algorithms (Algorithms 3 and 5) with a bottom-up approach after implementing the core helper functions. In the following, we show the function implemented on Node.js for detecting topological relations (Listing 9) between level members, which is covered in Algorithm 3. The same approach with minor differences (in parameter passing) is used in our implementation for detecting topological relations between fact-level members (Algorithm 5).

Listing 9 is constructed with the main function `detectSpatialHierarchySteps` with parameters of `parentLevelMembers`, `childLevelMem-`

bers, and explicitRelations.[20] In Line 5, the constant spatialHierachySteps takes the explicitRelations between child level and parent level members, and creates constants for those in Lines 8 and 9. The next step is to get the spatial values of the level members (child level members Lines 10–14 and parent level members Lines 15–19), where we utilize the helper function getSpatialValues, which is described in Listing 6. In Line 20, we create a constant topoRel, which takes the helper function relateSpatialValues (Listing 8) with two parameters childLevelSpatialValues and parentLevelSpatialValues that are created, in Lines 10 and 15, respectively. Next, we return the topological relations (topoRel) as predicates (p) between Lines 24–26. If a topological relation is not found, we keep the explicit relation as skos:broader (Line 26). Finally, we return the new results by replacing the explicitRelations with spatialHierarchySteps (Lines 28–32).

We now discuss our results in Table 3, for both cases covered in Algorithms 3 and 5, together with a number of input level members and fact members.

A.3. Discovering implicit topological relations

Discovering implicit topological relations is addressed in the following algorithms: Algorithm 4 – discoverSpatialHS and Algorithm 6 – discoverFactLevel. In both cases the source data has not any defined roll-up relations (with skos:broader), or has missing spatial hierarchy steps between level members. Similarly, a fact level member has no defined relation link to any spatial level member of its dimensions.

The input variables for Algorithm 4 – discoverSpatialHS are the triples with dimensions ($\mathcal{G}^S_D$), hierarchies in dimensions ($\mathcal{G}^S_H(d)$), levels in hierarchies ($\mathcal{G}^S_L(h)$) from the schema graph, and level members of levels ($\mathcal{G}^I_L M(l)$) and the attributes of level members ($\mathcal{G}^I_A(lm)$) from the instance data graph. Therefore, we query the endpoint by filtering with the schema elements qb4o:hasHierarchy, qb4o:inDimension, and qb4o:hasLevel. We fetch the results of the query in Node.js JSON format.

The input variables for Algorithm 6 – discoverFactLevel are the triples with dimensions ($\mathcal{G}^S_D$), hierarchies in dimensions ($\mathcal{G}^S_H(d)$), levels in hierarchies ($\mathcal{G}^S_L(h)$) from the schema graph, and fact members ($\mathcal{G}^I_F M(F)$), level members of levels ($\mathcal{G}^I_L M(l)$) and the attributes of level members ($\mathcal{G}^I_A(lm)$) from the instance data graph. Therefore, we query the endpoint by filtering with the schema elements qb4o:hasHierarchy, qb4o:inDimension, and qb4o:hasLevel. We fetch the results of the query in Node.js JSON format.

The following listing (Listing 10) shows how we implement a schema wrapper by filtering the schema graph at our endpoint with predicates for schema elements (Lines 3, 7, 11, and 14). Once we get to the levels, we filter the level members in each level with qb4o:memberOf predicate (Line 11). Afterwards, we group level members by level that are in the same hierarchy and pass these grouped level members as inputs to a similar function as in Listing 9, which is called detectSpatialHierarchyStepsExpensive. This function takes only two parameters without explicit relations (two sets of level members grouped by level: parentLevelMembers and childLevelMembers). We run this algorithm several times for each pair of grouped level members (by level) within the same hierarchy as our approach is discovering implicit relations between level members and fact-level members. For fact members we similarly use one parameter (i.e., parentLevelMembers) as the grouped level members

```
1  const discoverSpatialHierarchySteps = schema =>
2      schema.results.bindings.filter(binding =>
3        binding.p.value ==="qb4o:hasHierarchy")
4      .map(hierachyBinding =>
5        schema.results.bindings.filter(binding =>
6        hierarchyBinding.o.value === binding.s.value
7        && binding.p.value ==="qb4o:hasLevel"));
8      .map(levelBinding =>
9        schema.results.bindings.filter(binding =>
10       levelBinding.o.value === binding.s.value
11       && binding.p.value ==="qb4o:memberOf"));
12 const inDimension =
13     schema.results.bindings.filter(binding =>
14     binding.p.value === "qb4o:inDimension");
15 module.exports ={
16     wrapper: discoverSpatialHierarchySteps};
```

Listing 10. Discovering topological relations (schema wrapper)

[20]We do not repeat a similar listing in the paper for detecting topological relations between fact-level members (Algorithm 5) where the parameter childLevelMembers from Listing 9 corresponds to fact members and parentLevelMembers corresponds to base level members in the implementation of detecting topological relations between fact-level members.

(by level), and the other parameter is fact members (i.e., `childLevelMembers`), which are annotated as `qb:Observation`. In the `detectSpatialHierarchyStepsExpensive` function we utilize the same helper functions that are implemented with child-parent topological relations and simplification rules defined in Section 4.1.1 along with Fig. 8 and Table 1. This ensures to apply spatial boolean predicates (on geometries of level members and fact members) with `relateSpatialValues` helper function only between the appropriate spatial data types given in Tables 1 and 7. Since there are no explicit relations in `detectSpatialHierarchyStepsExpensive` function, `relateSpatialValues` helper function is called $NumberOf_{childLevelMembers} \times NumberOf_{parentLevelMembers}$ in one iteration, where with `detectSpatialHierarchySteps` function, the helper function is called only $NumberOf_{explicitRelations}$ times.

We now discuss our results in Table 3 for both cases covered in Algorithms 4 and 6, together with a number of input level members and fact members.

A.4. Generating the fact schema

Finally, we implement the enrichment of the fact schema based on spatially enriched fact instances (members). To extract the input variables for Algorithm 7 – `defineSpatialFactDSD`, we use the spatially enriched fact members (by Algorithms 5 and 6) and non-spatial fact schema.

The first step of generating the fact schema is to look for detected and discovered topological relations between the fact and level members and then annotate each of them with `qb4so:topologicalRelation` in the fact schema as given in Listing 3. The next step is to identify the spatial data types with helper functions `getMeasures` and `getSpatialValues` (Listings 5 and 6). Finally, for each of the identified spatial data types we annotate the fact schema with the corresponding spatial aggregate function, e.g., spatial data type POINT can have *ConvexHull* aggregate function, LINE can have *Union* etc.

In our implementation of detecting and discovering topological relations between fact members and level members, we have only encountered the `qb4so:within` topological relation. Thus, the fact schema enrichment implementation generates Lines 4 and 5 as exemplified in Listing 3. As spatial measures in fact members, we have found the POINT spatial data type. Therefore, the fact schema enrichment implementation generates Lines 6 and 7, annotating that the spatial measure has `qb4so:ConvexHull` aggregate function, as exemplified in Listing 3.

After the spatial enrichment is fully completed, both schema[21] and instance[22] data has been published via the same SPARQL endpoint with QB4SOLAP.

References

[1] A. Abelló, O. Romero, T.B. Pedersen, R. Berlanga, V. Nebot, M.J. Aramburu and A. Simitsis, Using semantic web technologies for exploratory olap: A survey, *IEEE Transactions on Knowledge and Data Engineering* **27**(2) (2015), 571–588. doi:10.1109/TKDE.2014.2330822.

[2] A.B. Andersen, N. Gür, K. Hose, K.A. Jakobsen and T.B. Pedersen, Publishing Danish Agricultural Government data as semantic web data, *Semantic Technology* **8943** (2014), 178–186. doi:10.1007/978-3-319-15615-6_13.

[3] R. Atkinson, QB4ST: RDF data cube extensions for spatio-temporal components, W3C Working Group, 2017, https://www.w3.org/TR/qb4st/.

[4] D. Brickley, Basic geo (WGS84 lat/long) vocabulary, W3C Semantic Web Interest Group, 2003, https://www.w3.org/2003/01/geo/.

[5] R.P. Deb Nath, K. Hose and T.B. Pedersen, Towards a programmable semantic extract-transform-load framework for semantic data warehouses, in: *Proceedings of the 18th International Workshop on Data Warehousing and OLAP*, 2015, pp. 15–24. doi:10.1145/2811222.2811229.

[6] M.J. Egenhofer and J. Herring, A mathematical framework for the definition of topological relationships, in: *Fourth International Symposium on Spatial Data Handling*, 1990, pp. 803–813. doi:10.1080/02693799108927841.

[7] L. Etcheverry, A. Vaisman and E. Zimányi, Modeling and querying data warehouses on the semantic web using QB4OLAP, in: *Data Warehousing and Knowledge Discovery*, Vol. 8646, Springer, 2014, pp. 45–56. doi:10.1007/978-3-319-10160-6_5.

[8] V. Gaede and O. Günther, Multidimensional access methods, *ACM Computing Surveys (CSUR)* **30**(2) (1998), 170–231. doi:10.1145/280277.280279.

[9] L. Galárraga, K. Ahlstrøm, K. Hose and T.B. Pedersen, Answering provenance-aware queries on RDF data cubes under memory budgets, in: *The Semantic Web – ISWC 2018*, Springer International Publishing, Cham, 2018, pp. 547–565. doi:10.1007/978-3-030-00671-6_32.

[10] L. Galárraga, K.A.M. Mathiassen and K. Hose, QBOAirbase: The European air quality database as an rdf cube, in: *International Semantic Web Conference (Posters, Demos & Industry Tracks)*, 2017.

[21] http://extbi.cs.aau.dk/geofarm/qb4solap/geofarm-qb4solap-schema.ttl

[22] http://extbi.cs.aau.dk/geofarm/qb4solap/geofarm-qb4solap-output.tar.gz

[11] E. Gallinucci, M. Golfarelli, S. Rizzi, A. Abelló and O. Romero, Interactive multidimensional modeling of linked data for exploratory olap, *Information Systems* (2018). doi:10.1016/j.is.2018.06.004.

[12] N. Gür, K. Hose, T.B. Pedersen and E. Zimányi, Enabling spatial OLAP over environmental and farming data with QB4SOLAP, in: *Semantic Technology*, Vol. 10055, Springer, 2016, pp. 287–304. doi:10.1007/978-3-319-50112-3_22.

[13] N. Gür, K. Hose, E. Zimányi and T.B. Pedersen, Modeling and querying spatial data warehouses on the semantic web, in: *Semantic Technology*, Vol. 9544, Springer, 2015, pp. 1–20. doi:10.1007/978-3-319-31676-5_1.

[14] N. Gür, J. Nielsen, K. Hose and T.B. Pedersen, GeoSemOLAP: SOLAP on the semantic web made easy, in: *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 213–217. doi:10.1145/3041021.3054731.

[15] N. Gür, T.B. Pedersen, E. Zimányi and K. Hose, A foundation for spatial data warehouses on the semantic web, *Semantic Web Journal* **9**(5) (2018), 557–587. doi:10.3233/SW-170281.

[16] A. Harth, K. Hose and R. Schenkel (eds), *Linked Data Management*, Chapman and Hall/CRC, 2014. doi:10.1201/b16859.

[17] O. Hartig, K. Hose and J.F. Sequeda, Linked data management, in: *Encyclopedia of Big Data Technologies*, S. Sakr and A.Y. Zomaya, eds, Springer, 2019. doi:10.1007/978-3-319-63962-8_76-1.

[18] K. Hose and R. Schenkel, RDF stores, in: *Encyclopedia of Database Systems*, L. Liu and M.T. Özsu, eds, 2nd edn, Springer, 2018. doi:10.1007/978-1-4614-8265-9_80676.

[19] W. Huang, S.A. Raza, O. Mirzov and L. Harrie, Assessment and benchmarking of spatially enabled RDF stores for the next generation of spatial data infrastructure, *ISPRS International Journal of Geo-Information* **8**(7) (2019), 310. doi:10.3390/ijgi8070310.

[20] B. Hyland and B.V. Terrazas, Cookbook for open government linked data, W3C Government Linked Data Working Group, 2011, https://www.w3.org/2011/gld/wiki/Linked_Data_Cookbook.

[21] D. Ibragimov, K. Hose, T.B. Pedersen and E. Zimányi, Processing aggregate queries in a federation of SPARQL endpoints, in: *The Semantic Web. Latest Advances and New Domains – 12th European Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, May 31–June 4, 2015. Proceedings*, 2015, pp. 269–285. doi:10.1007/978-3-319-18818-8_17.

[22] D. Ibragimov, K. Hose, T.B. Pedersen and E. Zimányi, Towards exploratory OLAP over linked open data – a case study, in: *Enabling Real-Time Business Intelligence*, Springer, 2015, pp. 114–132. doi:10.1007/978-3-662-46839-5_8.

[23] D. Ibragimov, K. Hose, T.B. Pedersen and E. Zimányi, Optimizing aggregate SPARQL queries using materialized RDF views, in: *The Semantic Web: 15th International Semantic Web Conference (ISWC'16)*, Springer, 2016, pp. 341–359. doi:10.1007/978-3-319-46523-4_21.

[24] T. Ioannidis, G. Garbis, K. Kyzirakos, K. Bereta and M. Koubarakis, Evaluating geospatial RDF stores using the benchmark Geographica 2, 2019, arXiv preprint, https://arxiv.org/abs/1906.01933 arXiv:1906.01933.

[25] K.A. Jakobsen, A.B. Andersen, K. Hose and T.B. Pedersen, Optimizing RDF data cubes for efficient processing of analytical queries, in: *COLD*, 2015, http://ceur-ws.org/Vol-1426/paper-02.pdf.

[26] B. Kämpgen and A. Harth, No size fits all – running the star schema benchmark with SPARQL and RDF aggregate views, in: *Extended Semantic Web Conference*, Springer, 2013, pp. 290–304. doi:10.1007/978-3-642-38288-8_20.

[27] B. Kämpgen, S. O'Riain and A. Harth, Interacting with statistical linked data via OLAP operations, in: *The Semantic Web: ESWC 2012 Satellite Events*, Vol. 7540, Springer, 2012, pp. 87–101. doi:10.1007/978-3-662-46641-4_7.

[28] K. Kyzirakos, D. Savva, I. Vlachopoulos, A. Vasileiou, N. Karalis, M. Koubarakis and S. Manegold, GeoTriples: Transforming geospatial data into RDF graphs using R2RML and RML mappings, *Journal of Web Semantics* **52** (2018), 16–32. doi:10.1016/j.websem.2018.08.003.

[29] E. Malinowski and E. Zimányi, Representing spatiality in a conceptual multidimensional model, in: *Proceedings of the 12th Annual ACM International Workshop on Geographic Information Systems, GIS'04*, ACM, New York, NY, USA, 2004, pp. 12–22, http://doi.acm.org/10.1145/1032222.1032226. doi:10.1145/1032222.1032226.

[30] A. Miles and S. Bechhofer, SKOS Simple Knowledge Organization System namespace document, W3C Recommendation, 2009, https://www.w3.org/TR/skos-reference/.

[31] R.P.D. Nath, K. Hose, T.B. Pedersen and O. Romero, SETL: A programmable semantic extract-transform-load framework for semantic data warehouses, *Information Systems* **68** (2017), 17–43. doi:10.1016/j.is.2017.01.005.

[32] R.P.D. Nath, K. Hose, T.B. Pedersen, O. Romero and A. Bhattacharjee, SETLBI: an integrated platform for semantic business intelligence, in: *Companion of the 2020 Web Conference 2020*, Taipei, Taiwan, April 20–24, 2020, 2020, pp. 167–171. doi:10.1145/3366424.3383533.

[33] S. Neumaier and A. Polleres, Geo-semantic labelling of open data, SEMANTiCS 2018 – 14th International Conference on Semantic Systems, *Procedia Computer Science* (2018). doi:10.1016/j.procs.2018.09.002.

[34] Open Geospatial Consortium, GeoSPARQL: A geographic query language for RDF data, W3C Recommendation, 2014.

[35] K. Patroumpas, N. Georgomanolis, T. Stratiotis, M. Alexakis and S. Athanasiou, Exposing INSPIRE on the semantic web, *Web Semantics* **35**(P1) (2015), 53–62. doi:10.1016/j.websem.2015.09.003.

[36] D. Pedersen, J. Pedersen and T.B. Pedersen, Integrating XML data in the TARGIT OLAP system, in: *Proceedings. 20th International Conference on Data Engineering*, IEEE, 2004, pp. 778–781. doi:10.1109/ICDE.2004.1320045.

[37] D. Pedersen, K. Riis and T.B. Pedersen, Query optimization for OLAP-XML federations, in: *Proceedings of the 5th International Workshop on Data Warehousing and OLAP (DOLAP'02)*, ACM, 2002, pp. 57–64. doi:10.1145/583890.583899.

[38] S. Rivest, Y. Bédard, M.-J. Proulx, M. Nadeau, F. Hubert and J. Pastor, SOLAP technology: Merging business intelligence with geospatial technology for interactive spatio-temporal exploration and analysis of data, *ISPRS Journal of Photogrammetry and Remote Sensing* **60**(1) (2005), 17–33. doi:10.1016/j.isprsjprs.2005.10.002.

[39] G. Rojas, G. Giannopoulos and J.J.L. Daniel Hladky, Managing geospatial linked data in the GeoKnow project, in: *The Semantic Web in Earth and Space Science. Current Status and Future Directions*, Vol. 20, IOS Press, 2015, p. 51. doi:10.3233/978-1-61499-501-2-51.

[40] J.M. Salas and A. Harth, NeoGeo vocabulary specification, GeoVocab Working Group, 2012, http://geovocab.org/doc/neogeo/.

[41] C. Stadler, J. Lehmann, K. Höffner and S.A. Linkedgeodata, A core for a web of spatial open data, *Semantic Web* **3**(4) (2012), 333–354, https://dl.acm.org/doi/10.5555/2590208.2590210. doi:10.3233/SW-2011-0052.

[42] A. Vaisman and E. Zimányi, Spatial data warehouses, in: *Data Warehouse Systems*, Springer, 2014, pp. 427–473. doi:10.1007/978-3-642-54655-6.

[43] J. Varga, L. Etcheverry, A.A. Vaisman, O. Romero, T.B. Pedersen and C. Thomsen, QB2OLAP: enabling OLAP on statistical linked open data, in: *32nd IEEE International Conference on Data Engineering*, 2016, pp. 1346–1349. doi:10.1109/ICDE.2016.7498341.

[44] J. Varga, A.A. Vaisman, O. Romero, L. Etcheverry, T.B. Pedersen and C. Thomsen, Dimensional enrichment of statistical linked open data, in: *Web Semantics: Science, Services and Agents on the World Wide Web*, Vol. 40, Elsevier, 2016, pp. 22–51. doi:10.1016/j.websem.2016.07.003.

[45] M. Wick and B. Vatant, Geo names ontology, GeoNames, 2012, http://www.geonames.org/ontology.

[46] World Wide Web Consortium, SPARQL query language for RDF, W3C Recommendation, 2008, https://www.w3.org/TR/sparql11-query/.

[47] World Wide Web Consortium, Data cube implementations, 2011, https://www.w3.org/2011/gld/wiki/Data_Cube_Implementations.

[48] World Wide Web Consortium, The RDF data cube vocabulary, W3C Recommendation, 2014, https://www.w3.org/TR/vocab-data-cube/.

[49] X. Yin and T.B. Pedersen, Evaluating XML-extended OLAP queries based on physical algebra, *Journal of Database Management (JDM)* **17**(2) (2006), 85–116. doi:10.4018/jdm.2006040105.

[50] E. Zimanyi, *Advanced Data Warehouse Design: From Conventional to Spatial and Temporal Applications. Data-Centric Systems and Applications*, Springer, 2008. doi:10.1007/978-3-540-74405-4.