



# Parallel and scalable processing of spatio-temporal RDF queries using Spark

Panagiotis Nikitopoulos<sup>1</sup> · Akrivi Vlachou<sup>1</sup> · Christos Doulkeridis<sup>1</sup> · George A. Vouros<sup>1</sup>

Received: 2 August 2018 / Revised: 25 March 2019 / Accepted: 20 June 2019 /

Published online: 03 July 2019

© Springer Science+Business Media, LLC, part of Springer Nature 2019

## Abstract

The ever-increasing size of data emanating from mobile devices and sensors, dictates the use of distributed systems for storing and querying these data. Typically, such data sources provide some spatio-temporal information, alongside other useful data. The RDF data model can be used to interlink and exchange data originating from heterogeneous sources in a uniform manner. For example, consider the case where vessels report their spatio-temporal position, on a regular basis, by using various surveillance systems. In this scenario, a user might be interested to know which vessels were moving in a specific area for a given temporal range. In this paper, we address the problem of efficiently storing and querying spatio-temporal RDF data in parallel. We specifically study the case of SPARQL queries with spatio-temporal constraints, by proposing the *DiStRDF* system, which is comprised of a Storage and a Processing Layer. The *DiStRDF* Storage Layer is responsible for efficiently storing large amount of historical spatio-temporal RDF data of *moving objects*. On top of it, we devise our *DiStRDF* Processing Layer, which parses a SPARQL query and produces corresponding logical and physical execution plans. We use Spark, a well-known distributed in-memory processing framework, as the underlying processing engine. Our experimental evaluation, on real data from both aviation and maritime domains, demonstrates the efficiency of our *DiStRDF* system, when using various spatio-temporal range constraints.

**Keywords** Distributed query processing · Distributed spatio-temporal queries · SPARQL queries

---

✉ Panagiotis Nikitopoulos  
nikp@unipi.gr

Akrivi Vlachou  
avlachou@aueb.gr

Christos Doulkeridis  
cdouk@unipi.gr

George A. Vouros  
georgev@unipi.gr

<sup>1</sup> Department of Digital Systems, School of Information and Communication Technologies, University of Piraeus, Piraeus, Greece

# 1 Introduction

Nowadays, several applications in critical domains, such as maritime and aviation, acquire, manage and process spatio-temporal data related to the mobility of entities (vessels, aircraft, etc.). Besides such surveillance data, other sources include weather and contextual data. As a result, the underlying data to be managed is typically heterogeneous, thus calling for a unified representation that allows declarative querying. To this end, an appropriate solution is the Resource Description Framework (RDF), a specification recommended by W3C<sup>1</sup> for modeling and exchanging data over the web. By adopting RDF as representation format, a declarative query language (SPARQL) can also be used for querying.

The motivation of our work comes from our involvement in the H2020 datAcron project (<http://www.datacron-project.eu/>), which targets Big Data analytics for time-critical maritime and aerial mobility forecasting [30]. In datAcron, surveillance data from vessels and aircraft are collected in real-time from various data sources (i.e., radars, satellites) in heterogeneous formats, transformed to RDF format, integrated with other external data sets, thus producing massive, GB-sized *spatio-temporal RDF data* in a daily fashion [22]. Efficient and scalable querying of this data set of increasing size is of paramount importance, as the query results produce integrated data that feed higher-level data analysis tasks, such as trajectory clustering based on mobility and weather attributes.

Even though scalable processing of SPARQL queries has attracted the attention of research community [11, 13, 20, 21, 25], these approaches rely on MapReduce/Hadoop, which has performance limitations, as surveyed in [6]. Lately, some first approaches for parallel processing of in-memory RDF data [18, 24, 26] using Spark have appeared, but these are not designed either for handling spatial nor spatio-temporal data. Essentially, this means that the spatio-temporal processing cannot be integrated in RDF processing, but must be developed as a pre- or post-processing step. However, this “decoupled” approach misses opportunities for pruning unnecessary data early in the query processing pipeline, and inevitably leads to inferior performance.

In this paper, we focus on scalable processing of spatio-temporal SPARQL queries on Spark, focusing on combining parallel RDF processing with spatio-temporal querying, in order to increase the efficiency of query processing. We propose the *DiStRDF engine* (Distributed Spatio-temporal RDF engine) which comprises of two main modules: the *Storage Layer* and the *Processing Layer*. The *Storage Layer* provides distributed storage of spatio-temporal RDF data, with careful data organization on disk and storage layout. The *Processing Layer* is a query engine that provides both logical and physical operators for spatio-temporal RDF data, thereby offering the opportunity for different execution plans to increase performance gains.

RDF data in *DiStRDF* is actually stored encoded using unique integer identifiers. To produce these identifiers, we exploit an 1D encoding scheme which injects spatio-temporal information to the stored RDF data. This approach has a significant advantage: we can prune nodes based on spatio-temporal criteria, by simply checking their unique identifiers. The *Processing Layer* exploits this feature to enable efficient distributed spatio-temporal RDF query processing.

In summary, our contributions can be summarized as follows:

- We present the design and implementation of *DiStRDF*, which is a parallel in-memory and scalable spatio-temporal RDF processing engine based on Spark.

<sup>1</sup><https://www.w3.org/>

- We propose the *DiStRDF Storage Layer*, which stores encoded RDF triples and a dictionary of mappings between integer identifiers and RDF resources; it also uses Property tables and columnar storage layout for improved performance.
- We propose the *DiStRDF Processing Layer*, which is comprised by a query parsing component, a logical query builder and a physical query constructor in order to produce execution plans that efficiently handle spatio-temporal constraints along with SPARQL processing.
- We demonstrate the efficiency of *DiStRDF* by means of a variety of experiments using fairly big, real-life data, from two domains: maritime and aviation.

This paper is an extended version of [19]. The new contributions of this paper can be summarized as follows:

- We extend the 1D encoding scheme from 3D (2D space and time) to the 4D case, by implementing Z-order based encoding, apart from Hilbert based encoding (Section 4.1). By extending the encoding scheme, we are able to support queries for other domains (such as ATM) which contain 3 spatial dimensions.
- We present the final implementation of the storage layer, which uses Property tables and Parquet (Section 4.3). Our proposed storage layer relies on HDFS to provide high availability of the stored data, while enabling efficient query execution and data compression.
- In the processing layer, we show the separation between logical and physical operators, and we study how different execution plans impact the performance of query processing (Sections 5 and 6). By separating the logical representation of a query from its physical implementation, we are able to (a) conclude to a more efficient query plan based on its logical representation and (b) select the most efficient set of algorithms for implementing the query plan.
- We present a more in-depth experimental evaluation, studying the benefits of our techniques (namely encoding, approximate filtering, different physical operators, scalability), and also using larger data sets, including 4D data coming from the ATM domain (Section 6). By experimenting with 4D data sets, we are able to evaluate our extended 1D encoding scheme, and demonstrate support queries having 3 spatial dimensions.

The rest of the paper is organized as follows: Section 2 provides an overview of related work. Section 3 demonstrates an overview of our *DiStRDF* system. Section 4 explains the 1D encoding scheme used for storing spatio-temporal data and introduces the *DiStRDF Storage Layer*. Then, in Section 5 we describe the *DiStRDF Processing Layer*, and the logical query plans we have implemented. Section 6 presents the results of our experimental study, while Section 7 concludes the paper and sketches future research directions.

## 2 Related work

Even though the topic of parallel processing of large-scale RDF data has attracted much attention recently (cf. [1, 12] for related surveys), there is no work on parallel and distributed processing of spatio-temporal RDF data at scale. Approaches for in-memory, distributed processing of RDF data [18, 24, 26] are related to our work, yet they do not cater for the case of spatio-temporal data represented in RDF. In practice, this means that processing of spatio-temporal RDF queries is “decoupled”, leading to filtering the RDF data based on the RDF graph patterns (without taking into account the spatio-temporal constraints), followed

by a refinement step that would exclude from the candidate results, those that do not satisfy the spatio-temporal constraints. Unfortunately, this approach incurs higher processing costs, since a large number of candidate results are only pruned during the very last stages of query processing.

Scalable processing of big spatial [7, 9, 28, 31–33] and spatio-temporal [2, 10] data has been studied recently, however these approaches focus only on the spatial (or spatio-temporal) dimension of data, by enabling efficient retrieval based on spatio-temporal constraints. In case of spatio-temporal RDF data, such solutions would have to resolve the required RDF pattern matching *after* having identified the data that satisfy the spatio-temporal constraints (also called candidate results). Obviously, this approach leads to wasteful processing, since a high number of candidate results are computed in vain, since they will later be pruned by the RDF pattern matching. Clearly, a more efficient solution would resolve both the spatio-temporal part of the query and the graph patterns at the same time, in order to increase the effectiveness of filtering. This is the approach adopted by our work, and it is provided without the use of specialized distributed indexing structures.

Existing works on spatio-temporal RDF data [3, 8, 15, 16] propose RDF storage and processing solutions over centralized stores, therefore they cannot cope with the voluminous nature of big spatio-temporal RDF data that our approach must handle. Finally, the proposed encoding scheme for spatio-temporal RDF data has similarities to the approach adopted in [17] for spatial RDF data. The main difference is that the temporal dimension cannot be treated as yet another dimension, but requires special handling. In turn, this raises challenges relating to producing compact encoded values, an issue that is studied in detail in [29].

The need for effectively managing linked geospatial data from heterogeneous data sources has attracted much attention. Hence, the Open Geospatial Consortium (OGC)<sup>2</sup> has proposed standards for modeling and querying this data in a unified manner. OGC standardizes the representation of geometries by using a text representation, called Well Known Text (WKT).<sup>3</sup> WKT can represent various geometry types in 2D or 3D space (e.g. line, point, polygon, etc.), by referencing the respective coordinates. Moreover, OGC has proposed a standardized language for querying geospatial data, namely the GeoSPARQL language.<sup>4</sup> This is an extension to the standard SPARQL query language, which provides additional features (vocabulary) for querying this type of data. The study in [15] has proposed the use of stRDF model and stSPARQL language for modeling and querying spatio-temporal data respectively. The work of [14] describes the various models and query languages that exists in literature, in more details.

## 2.1 Comparison of DiStRDF to other systems

In this section, we conduct a qualitative comparison of our DiStRDF system, against similar state of the art systems found in literature. Specifically, we compare DiStRDF against (a) Staborn [16] which is the state of the art centralized system for stSPARQL queries and (b) Simba [31] which is the most efficient system for main memory distributed spatial query processing. We study several factors affecting both performance and supported features of these systems, namely their processing model and supported query language, the number of spatio-temporal dimensions that they can handle efficiently on query evaluation, their

<sup>2</sup><http://www.opengeospatial.org/>

<sup>3</sup><https://www.opengeospatial.org/standards/wkt-crs>

<sup>4</sup><https://www.opengeospatial.org/standards/geosparql>

partitioning and indexing mechanisms, the geometry types that they use and the types of spatial or spatio-temporal range queries that they support. The summary of the comparative analysis is depicted in Table 1.

In terms of query processing, Strabon is based on PostgreSQL for evaluating a given query. Thus, it is not designed to natively support distributed query evaluation by using a set of computing machines. Both Simba and DiStRDF use Spark as their query engine, hence they natively support distributed main memory query execution. Moreover, Simba supports SQL queries, by exploiting the Spark SQL API, Strabon supports stSPARQL queries by using a custom query parser and DiStRDF supports SPARQL queries by integrating Apache Jena in its Query Parsing component.

Simba is a spatial query engine, thus it can handle only spatial dimensions for query evaluations. Strabon and DiStRDF are spatio-temporal systems, thus both can handle a temporal dimension efficiently. Strabon supports 2 spatial dimensions through its stSPARQL query support, while DiStRDF supports 2 or 3 spatial dimensions by using the appropriate encoding scheme.

Strabon is a centralized engine, thus it does not need a partitioning mechanism to operate on the stored data. Simba constructs MBRs based on a sampled subset of the data, and then uses the bounds of these MBRs to partition the data, thus preserving their spatial locality. DiStRDF uses the 1D spatio-temporal encoding to partition the data, preserving their spatio-temporal locality. In terms of indexing, Strabon uses B+Trees for general RDF data, and an R-Tree-over-GiST structure as an index for the spatial information. Simba constructs first a local index on each of the computing machines, which is specified by a user and can be either a HashMap, a TreeMap or an R-Tree. Then, Simba aggregates the information from all local indexes to produce a global index. DiStRDF uses the Hilbert curve or the Z-order curve to produce an encoding (hash) value; these values are grouped in ranges to enable efficient pruning groups of data which do not satisfy query predicates.

Regarding geometry objects, Strabon supports several types of geometries: Point, Curve, LineString, Line, LinearRing, Polygon, and Triangle. Simba supports the main geometries, namely Circles, Lines, Segments, Rectangles, Points and Polygons. DiStRDF currently supports only Point geometries but is being extended to support also other types of geometries.

**Table 1** Feature comparison of related systems

Feature	Strabon [16]	Simba [31]	DiStRDF
Processing	Centralized	Distributed	Distributed
Query language	stSPARQL	SQL	SPARQL
Dimensions	2 spatial, 1 temporal	Multiple spatial	2 or 3 spatial, 1 temporal
Partitioning	—	Spatial range	Spatio-temporal range
Indexing	B+Tree, R-Tree-over-GiST	Multiple	Hilbert Hash, Z-order Hash
Geometries	Multiple	Circle, Line Segment, Rectangle, Point, Polygon	Point
Range Queries	Spatio-temporal Box, Spatio-temporal Circle	Spatial Box, Spatial Circle	Spatio-temporal Box, Spatio-temporal Circle

Lastly, both DiStRDF and Strabon support Circle and Box spatio-temporal range queries, while Simba supports natively only their spatial equivalents.

### 3 Overview of DiStRDF system

An overview of the DiStRDF system is depicted in Fig. 1. It consists of two layers: (a) the DiStRDF Processing Layer which is responsible for parsing and processing the SPARQL query and (b) the DiStRDF Storage Layer which is responsible for storing the RDF data. In the following, we briefly describe the main components of those Layers, by means of a SPARQL query processing workflow.

A SPARQL query and a separate spatio-temporal constraint are provided by a user to the DiStRDF Processing Layer. The SPARQL Query Parsing Component checks the correctness of the query syntax and transforms it to a machine readable representation. The Logical Plan Builder component creates the Logical Plan representation of the query, which incorporates the SPARQL query with the spatio-temporal constraints and consists of a set of Logical operators. Then, the Physical Plan Constructor Component transforms the Logical Plan to a Physical representation, by choosing an implementation algorithm for each Logical operator. The Processing Layer exploits Apache Spark to execute the query in parallel, on a set of computing machines. The components of DiStRDF Processing Layer are described in more details in Section 5.

Notice that we do not currently use a SPARQL extension specific for spatial or spatio-temporal data. However, our system is designed to easily support such extensions; one could easily extend the language used for query representation, which would then be translated to a logical query plan, and the underlying DiStRDF engine and its components would still operate on this query plan. This change, would affect mainly the SPARQL Query Parsing component, since it is responsible for transforming the input query in a machine readable representation.

Spark reads RDF triples data from HDFS. The triples are stored in a format suitable for efficient retrieval, by exploiting an 1D spatio-temporal encoding scheme. Since RDF data are encoded using this technique, a main memory distributed system is used (Redis) as a dictionary, mapping the corresponding stored values between their 1D representation

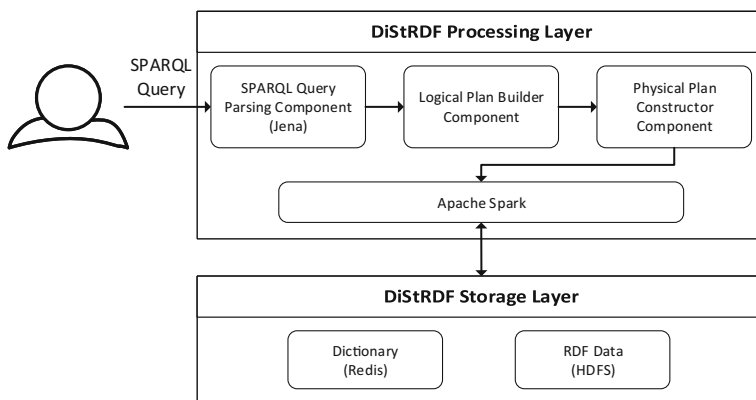


Fig. 1 Overview of the DiStRDF system

and their original value. The components of DiStRDF Storage Layer are described in more details in Section 4.

## 4 The DiStRDF storage layer

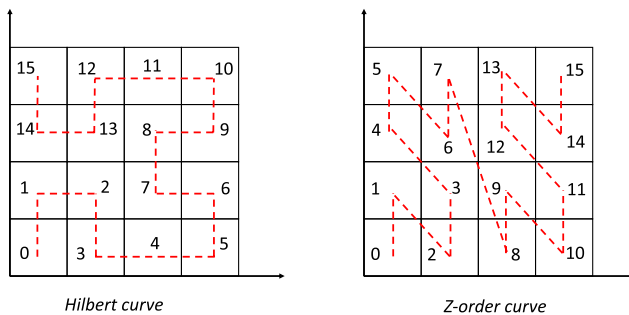
The storage layer is responsible for distributed storage of RDF data, represented as RDF triples. As typical in RDF storage systems, we employ a *dictionary encoding* technique [5] using a mapping table, in order to handle triples of integer values. This allows more efficient processing, since it is easier to index, compress, and process integer values, rather than strings.

However, as most of our RDF data have a spatio-temporal nature, we adopt the special-purpose encoding scheme described in [29]. As in any dictionary encoding scheme, an integer value corresponds to an RDF resource uniquely. In our case of RDF resources corresponding to spatio-temporal entities, we generate integer values in an intentional way, so that they provide an approximate position of the entity in space and time. In this way, we can filter RDF triples during scans using spatio-temporal constraints, as is shown in Section 5. More interestingly, this feature comes “for free”, without the need to build and maintain special-purpose (distributed) spatio-temporal indexes. Also, our solution is readily applicable to any distributed RDF processing system that utilizes dictionary encoding.

In the following, we first provide a short description of the 1D encoding scheme used for creating the dictionary (Section 4.1), so that the paper is self-contained. Then, we propose our solution for efficient and scalable storage and management of two pieces of data: (a) the dictionary that maps integer values to RDF resources and vice-versa (Section 4.2), (b) a large set of integer-encoded RDF triples (Section 4.3).

### 4.1 One-dimensional (1D) encoding scheme

To encode the spatio-temporal information of a moving entity, we first map its spatial location in a integer value. The spatial location can be in 2D (for cars, vessels, etc.) or 3D (for aircrafts), however for simplicity we focus on the 2D case in the following descriptions. The mapping of the spatial location using a grid partitioning of the space into  $2^m = (2^{m/2} * 2^{m/2})$  equi-sized cells, and a space-filling curve that provides an ordering for the spatial cells of the grid. Figure 2 shows an example for the 2D case for  $m = 4$ , where the Hilbert and Z-order curves are depicted, together with the IDs assigned to each cell. Space-filling curves aim to preserve the *spatial locality* by having nearby cells be also close on the curve. Obviously,



**Fig. 2** Space-filling curves (Hilbert and Z-order) used for ordering the spatial cells

some information loss is inevitable when going from 2D to 1D, however the spatial locality is mostly preserved. We have implemented both curves in our system, but Z-order is easier to extend for higher dimensions (e.g., 3D).

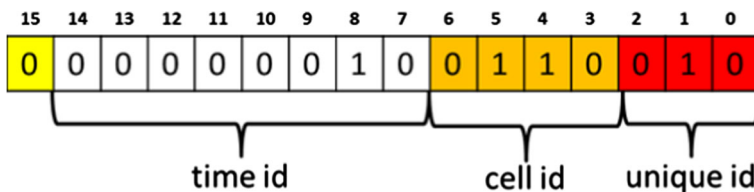
Any entity whose spatial location is enclosed in a spatial cell should be assigned with a unique identifier. Therefore, we reserve  $k$  bits for assigning unique IDs to different entities in the same spatial cell. As such, the maximum number of entities that fit in a spatial cell is  $2^k$ .

In order to handle the temporal dimension, consider a temporal partitioning  $\mathcal{T} = \{T_0, T_1, \dots\}$  of the time domain, where  $T_i$  represents a temporal interval. We make no assumptions on specific properties of the partitioning, i.e., the length (or duration) of temporal partitions can vary, apart from the fact that the partitions are disjoint, they cover the entire time domain ( $\bigcup T_i = \mathcal{T}$ ), and that  $T_i$  precedes  $T_{i+1}$  in the temporal order. Every temporal partition  $T_i$  is associated with a 2D spatial grid, as depicted in Fig. 2. The only restriction is that the identical grid structure (i.e.,  $2^m$  equi-sized cells) is used for all temporal partitions  $T_i$ .

Figure 3 shows how we put everything together and create a *unique* identifier for any spatio-temporal position of a moving entity. The figure depicts the binary representation of the identifier, consisting of  $b$  bits. The  $k$  least significant bits are used to encode the identifier of an entity in a specific spatio-temporal cell. The ID of the cell is recorded in the next  $m$  bits. The most-significant bit is reserved to discern between spatio-temporal RDF entities and other RDF entities. By convention, it is set to 0 for all IDs of spatio-temporal RDF entities, while it is set to 1 for IDs of all other RDF entities. The remaining  $b - (m + k + 1)$  bits are used for encoding the time, thus we can store  $2^{b-(m+k+1)}$  temporal partitions in total. The following example explains more clearly how the 1D identifier of an entity is generated.

**Example 1** In Fig. 3, we consider the case of  $b=16$ ,  $m=4$ , and  $k=3$ , and the depicted identifier is  $2^8 + 2^5 + 2^4 + 2 = 306$ . The spatial cell in which it belongs is 6 (=0110), and the spatial grid contains  $2^4 = 16$  cells in total. This encoding can accommodate  $2^{b-(m+k+1)} = 2^8 = 256$  temporal partitions. We also observe that this ID corresponds to the second entity found in the given spatio-temporal cell and that the entity is located in the second temporal partition.

Given an ID of a spatio-temporal entity, the 3D spatio-temporal cell enclosing the entity can be retrieved. Also, given a 3D cell, a range of IDs can be computed that correspond to any entity belonging to the cell. The proposed encoding ensures that entities with similar spatio-temporal representations are assigned IDs that belong to small ranges, thus preserving data locality. For example, given a time partition  $T_i$ , all entities in  $T_i$  belong to the interval  $[2^i * (2^{m+k}), 2^{i+1} * (2^{m+k})]$ , where  $2^m$  is the number of spatial cells, and  $2^k$  is the maximum number of objects within each spatial cell. Essentially,  $2^i$  is used to shift the intervals, thus we can map the different temporal partitions to different 1D intervals of identifiers. In summary, our encoding: (a) allows to retrieve a spatio-temporal approximation



**Fig. 3** IDs encoding using bits:  $b$  total bits,  $m$  bits for spatial part (cell id),  $k$  bits for uniqueness,  $b - (m + k + 1)$  bits for temporal part



given an ID, and (b) achieves to reflect the spatio-temporal locality in the 1D integer domain, by assigning nearby integer values to entities which are close to each other in the spatio-temporal space. Details on the computation of the identifier as well as various strategies for partitioning the temporal domain dynamically are provided in [29].

## 4.2 Storing the dictionary

An efficient storage solution needs to be selected for storing the dictionary, by considering the following requirements:

- The data model of the dictionary is a plain *key-value model*, where a bi-directional mapping is required between strings and integer values.
- Regarding access patterns, the dictionary is used for *lookups (random access) based on some key*, and we need to support very efficient retrieval of the respective values, in order to avoid delaying the actual query processing.
- The dictionary should be *stored in main-memory* to enable fast retrieval.
- The size of the dictionary is expected to be large (related works have reported that its size can be comparable to the size of the RDF triples), therefore we need a *distributed storage* scheme that scales gracefully with increased data size.
- The dictionary should provide high availability, even in the case of hardware failures.

After putting all the above requirements together, we turn our attention to *distributed NoSQL key-value stores* that satisfy such requirements. In particular, we opt for the REDIS,<sup>5</sup> an open-source, in-memory, distributed key-value store, which fits our purposes.

Redis provides a key-value storage and access model that fits our requirements. In order to support efficient bi-directional lookups, we need to maintain two separate Redis databases: the integer to string mapping and vice-versa. Redis keeps all data and indices in main memory, to enable fast data retrieval. Also, it supports data partitioning and replication, to enhance the capacity and availability of the system.

Even though other NoSQL key-value stores could also be used for storing the dictionary (e.g., Aerospike, RocksDB, Memcached, etc.), it is out of the scope of this paper to identify the best performing NoSQL store among the many available options. Obviously, using a more efficient key-value store than Redis would also improve the performance of the part of query processing in DiStRDF related to decoding RDF triples.

## 4.3 Storage of RDF triples

Distributed storage of RDF triples has been well-studied recently (cf. [1, 12]), due to the ever-increasing number and size of publicly available RDF data sets. The storage layer of DiStRDF relies on HDFS, a generic distributed file system which is optimized for storing large sets of data. For the design of the DiStRDF storage layer, different parameters have been taken into account, such as file layout, compression, data organization, partitioning and indexing. DiStRDF supports different options regarding these parameters; for example, the administrator can select between row-based storage or column-based storage. However, in the following, we focus on the solutions that we found out that work better, with respect to the specific real-world data and operational query requirements in the targeted application area.

<sup>5</sup><https://redis.io/>

In terms of *data organization*, we opt for an approach that relies on *Property tables* together with a table for storing “leftover” triples. Property tables show good performance when a group of properties always exists for a given resource, thereby avoiding the need of costly joins to reassemble this information. In our case, the majority of the query workload targets nodes representing the position of moving objects (a.k.a. *semantic nodes*), therefore we build a property table that maintains information related to the positions of moving objects. This is exemplified in Fig. 4, where the property table stores instances of Semantic Node, which is used to represent the position of a moving object, together with related attributes, such as its heading, and spatio-temporal position. The leftover triples are stored in a table with three columns, corresponding to subject, predicate, and object.

As regards the *file layout*, we use Parquet,<sup>6</sup> which provides a columnar format that achieves better compression and performs better for queries that retrieve few columns of a table only. Most importantly, Parquet supports “push-down” of operations, such as selections and projections, thereby avoiding the cost associated with retrieval of data that are not useful for the query at hand.

Last, but not least, we use a deliberate *data partitioning* mechanism that respects the spatio-temporal nature of the underlying data. Instead of random or hash-based partitioning of triples to nodes, we partition data based on spatio-temporal criteria. To this end, we exploit the spatio-temporal ID used to encode resources, and range-partition triples based on the spatio-temporal information injected in the encoded value of *semantic nodes*.

## 5 The DiStRDF processing layer

The *DiStRDF Processing Layer* is a SPARQL query engine that supports scalable and efficient batch query processing over vast-sized, spatio-temporal RDF data. To implement the *DiStRDF Processing Layer*, a parallel in-memory data processing engine is required. For this purpose, we select Apache Spark [34], a popular data processing engine with the widest set of contributors implementing the MapReduce model in main memory, thereby achieving significant performance gains [27] to competitor systems, such as Hadoop.

The DiStRDF Processing Layer is comprised of three components, namely (a) the SPARQL query parsing component, (b) the logical plan builder component and (c) the physical plan constructor component. In this section, we describe the design and implementation of these components.

### 5.1 SPARQL query parsing component

Given a SPARQL query, the first task is to perform *query parsing*, in order to:

- check the correctness of syntax and ensure that the query is specified correctly, and
- transform the query into an internal representation that will be used by the remaining modules of the processing engine.

Assuming that the syntax is correct, the SPARQL query is translated into a set of *basic graph patterns* (BGPs). These basic graph patterns are provided to the *logical planner* in order to construct a *logical query plan*.

<sup>6</sup><https://parquet.apache.org/>

Property Table

Semantic Node	ofMovingObject	hasHeading	hasGeometry	hasTemporalFeature
node15	ves376609000	15	15.3W 47.8N	2017-01-03 02:20:05
node22	ves369715600	0	19.4E 35.9N	2017-01-30 17:29:00
node58	ves376609000	3	23.2E 35.7N	2017-02-05 10:42:08

Leftover Triples

Subject	Predicate	Object
node15	hasSpeed	0
StoppedInit	occurs	node15
node58	hasSpeed	0

Fig. 4 Example of Property table used in DiStRDF storage layer

In DiStRDF, the query parsing component is built using off-the-shelf software, namely it is based on functionality provided by Apache Jena.<sup>7</sup>

5.2 Logical plan builder component

Given a SPARQL query that has passed the query parsing step successfully, the *logical plan builder* is assigned with the task of constructing a logical query plan, that consists of a hierarchy of logical operators. In essence, the logical query plan represents a way to execute the respective SPARQL query, by determining the hierarchical order of a set of logical operators.

We use five types of logical operators, which are described below. These operators are needed for processing the BGP part of a SPARQL query. Obviously, these operators do not cover the complete SPARQL specification (e.g., grouping, sorting, etc.), however they cover a wide variety of SPARQL queries, and constitute a fundamental and challenging part of a distributed RDF processing engine.

- *Join operator* ( $\bowtie$ ): This is a binary operator that takes as input two sets of data and joins them based on the following triple pattern fields: Subject-Object, Object-Object, Object-Subject and Subject-Subject.
- *Projection operator* ( $\pi$ ): A unary operator that keeps only a subset of the available fields of the input set of data.
- *Selection operator* ( $\sigma$ ): A unary operator that takes as input a set of data and a triple pattern and returns all data that match the pattern.
- *Spatio-temporal filter estimation operator* ( $\sigma_\epsilon$ ): Provided with a spatio-temporal box constraint, this unary operator applies *approximate filtering*, by exploiting the 1D encoding scheme introduced in Section 4.1. It inspects only the subject field to extract the embedded approximate spatio-temporal information, and keeps only the data which satisfy the range constraint. Since the information is approximate, a refinement is also needed to discard the false positives from the result set. To this end, we also introduce a spatio-temporal refinement operator.

<sup>7</sup><https://jena.apache.org/>

- *Spatio-temporal filter refinement operator* ( $\sigma_\rho$ ): Provided with a spatio-temporal box constraint, this unary operator performs *exact filtering*, based on the exact spatio-temporal information provided in the data (i.e. in the corresponding spatial and temporal fields).

A query plan is typically represented as a tree, which consists of a set of connected nodes corresponding to *logical operators*. Each node can have up to two children nodes and zero or more parent nodes. The nodes take as input the results of their children nodes and calculate the result of the operation described by their Logical Operator; this result is then given to their parent nodes to continue query execution. If a node has zero children (i.e. leaf nodes), then its input is a physical source of data. Moreover, if a node has no parent (i.e., root node), then its output is considered to be the query result.

---

**Algorithm 1** Logical plan builder.
 

---

**Input:**  $BGP$   
**Output:** Logical Plan Tree

```

1   $LP \leftarrow \{\}$ 
2  foreach triple pattern  $t_i \in BGP$  do
3       $so_i \leftarrow \text{new SelectOperator}(t_i)$ 
4      foreach field  $f_{ij} \in t_i$  do                                //  $j = \{Subject|Predicate|Object\}$ 
5          if  $f_{ij}$  is value then
6               $so_i.\text{addFilter}(f_{ij})$ 
7          end
8      end
9       $LP.\text{add}(so_i)$ 
10 end
11 while  $\exists (LP_\alpha, LP_\beta) : \alpha \neq \beta, f_{\alpha x} = f_{\beta y}$  do
12      $jo_{\alpha\beta} \leftarrow \text{new JoinOperator}(LP_\alpha, LP_\beta, x, y)$ 
13      $LP.\text{remove}(LP_\alpha, LP_\beta)$ 
14      $LP.\text{add}(jo_{\alpha\beta})$ 
15 end
16 while  $LP.\text{size} > 1$  do
17      $jo \leftarrow \text{new JoinOperator}(LP_1, LP_2)$ 
18      $LP.\text{remove}(LP_1, LP_2)$ 
19      $LP.\text{add}(jo)$ 
20 end
21  $po \leftarrow \text{new ProjectOperator}(LP_1)$ 
22 return  $po$ 
    
```

---

The pseudocode for building the logical plan is depicted in Algorithm 22. It takes as input the  $BGP$  provided by the Query Parsing Component and outputs the root node of the Logical Plan Tree. Initially, it iterates through the triple patterns of the  $BGP$  (lines 2-10) and creates a Selection operator for every triple pattern (line 3). For every value specified in the three fields of the triple pattern (subject, predicate or object), the operator is assigned a filter value (lines 4-8). The Selection operator is added to the  $LP$  list (line 9), to be used later. Notice that by exploiting this approach, the Selection operators are pushed as low as possible in the plan, thus minimizing the amount of data provided to the parent nodes. Hence, the leaf nodes of a query plan are typically Selection operators.

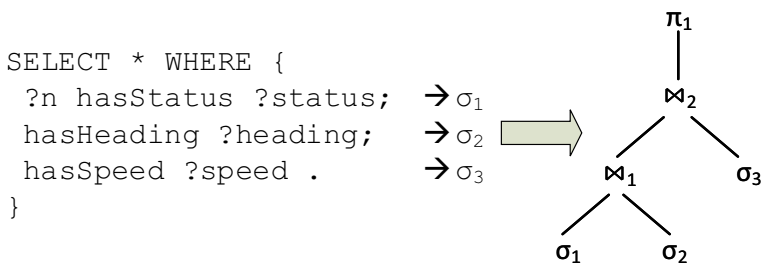
During the second step (lines 11-15), the logical plan builder iterates through the operators contained in *LP* list, trying to find common variable names between their corresponding fields. The matching operators are combined together by a Join operator (line 12), along with their matching fields which will be used as the join condition. Then these operators are removed from the *LP* list (line 13) before adding the new Join operator in it (line 14). Notice that, according to the SPARQL specification, this Join operator corresponds to an inner join between the underlying data sets. The loop ends when no other matching operators can be found in *LP* list.

In the third step (lines 16-20), the remaining operators that contain no common variable names, are combined together with a Join operator indicating a cross join between them. Since a cross join does not have a join condition, this Join operator stores only its children operators (line 17). As previously, the children are removed from *LP* (line 18), where the new Join operator is added (line 19). This loop ends, when the *LP* list is left with just a single element.

In the fourth step, the logical plan builder applies a Projection operator on the element of *LP* list (line 21), in order to keep only the fields which are specified in the *SELECT* statement of the SPARQL query. The Projection operator also specifies the field display names that will be assigned to the results, to match the ones provided by the user. The Projection operator is the output of the Logical Plan Builder component, since it is always the root node of our Logical Plan Tree.

An example of a logical plan produced by the aforementioned procedure is demonstrated in Fig. 5. The SPARQL query is depicted in the left part of the Figure, and consists of 3 triple patterns. During the first step, the triple patterns are mapped to  $\sigma_1, \sigma_2, \sigma_3$  Selection operators. These operators specify the filtering that should be applied to the result set, e.g.  $\sigma_1 : predicate = hasStatus$ . During the second step, the Selection operators are combined by Join operators on their matching variable names. The operators  $\sigma_1, \sigma_2$  and  $\sigma_3$  are all combined with the same variable name  $?n$ . In practice, since the underlying data are formatted as property tables, the join operation between two sets of data might be already computed. However, the Logical Plan Builder is designed to be a generic component, letting the Physical Plan Constructor to decide if a join operation is actually needed. In the example depicted in Fig. 5, although all Selection operators are combined with the same variable name  $?n$ , only the two of them are satisfied by the property table data source, namely the  $\sigma_1$  and  $\sigma_2$ . The remaining operator  $\sigma_3$ , is satisfied by the leftovers data source, thus requiring an actual join operation between  $\bowtie_1$  and  $\sigma_3$ . In the last step, a Projection operator is applied, to return the values  $?n, ?status, ?heading$  and  $?speed$  to the user.

The handling of spatio-temporal range constraints, and their corresponding operators, will be discussed later in Section 5.4.



**Fig. 5** Example of SPARQL query and corresponding Logical Plan

### 5.3 Physical plan constructor component

After the logical query plan has been built, the physical plan constructor transforms it to a physical execution plan by assigning a *physical operator* (i.e., a specific implementation of the corresponding task) to each logical node. Thus, in this section, we introduce the physical operators which are implemented in DiStRDF query engine.

These physical operators are designed to exploit the features of Spark SQL API, which distributes the workload among the available computing nodes. Hence, our implementations operate on data sets which are modeled as tables in main memory, typically represented by Spark SQL DataFrames. DataFrames are data structures which represent a set of distributed tuples, enabling parallel processing of the contained data.

#### 5.3.1 Selection operators

A physical Selection operator takes as input a table and selects a subset of its tuples which satisfy a set of filtering conditions. A Selection operator may define multiple filtering conditions which should be jointly satisfied, in order for a tuple to be included in the results. Thus, the physical Selection operators are designed to apply all specified filters in a single pass over the input data.

In DiStRDF, we have implemented two physical selection operators, namely the *IndexScan* operator and *TableScan* operator. *IndexScan* is used in the case an index is provided by the data source for the column(s) of the filter operation. Basically, it exploits the meta-data information included in *Parquet* formatted files to efficiently prune partitions of data that contain no matching tuples. If an index is not provided (e.g., the case of text formatted files), we opt to the *TableScan* operator which scans the input data in memory to identify matching tuples.

In Spark, a selection of data based on some filtering condition requires in principle a parallel scan of the input data; the resulting DataFrame contains only the tuples that match the filtering condition. Spark supports *predicate push-down*, namely avoiding reading all tuples and filtering them in memory, but rather reading only the tuples that match with the filtering condition. Essentially, Spark informs the storage system which records are necessary, and lets the storage system filter them without loading in memory. This is a very powerful feature when combined with a storage format, such as *Parquet*. We exploit predicate push-down when reading data from disk to memory, in order to achieve better performance.

Since the storage layer maintains the underlying data in two formats, namely property tables and one-triples tables (for leftover triples), special handling is needed to effectively operate on both formats in DiStRDF processing layer. To this end, we opt to transform the data obtained by one-triples formatted data sources to the equivalent property tables format. An example of such transformation is depicted in Fig. 6. The transformation occurs immediately after applying a Selection operator on the one-triples data source. In the remaining of this paper, this transformation procedure is implied whenever a Selection operator is applied on an one-triples data source.


#### 5.3.2 Join operators

The physical Join operator takes as input two tables and combines their tuples based on a related column between them. Notice that in some cases, the join operation is pre-computed

## Leftover Triples

Subject	Predicate	Object
node15	hasSpeed	52
node23	hasSpeed	43
node58	hasSpeed	22

## Property Table



Subject	hasSpeed
node15	52
node23	43
node58	22

**Fig. 6** Example of transforming an one-triples table to the equivalent property table

by the stored property tables. The DiStRDF processing layer is designed to exploit such information and avoids evaluating a pre-computed join operation.

Given the fact that data is distributed, processing a join operator requires distributed join processing. This is a challenging operation because it usually results in transferring large amounts of data from one node to another, and this cost can dominate the entire execution cost. As a result, optimizing the processing of joins is a critical factor for a distributed RDF processing engine.

By exploiting the Spark SQL API, we can utilize two physical join operators: Broadcast Hash Join and Sort-merge Join. The details of these algorithms can be found in [4]. However, they are briefly described in the following, assuming that *datasetA* and *datasetB* are joined together, when the size of *datasetB* is estimated to be smaller than the size of *datasetA*, based on the available statistics:

- **Broadcast Hash Join.** This algorithm is typically more efficient for smaller sizes of *datasetB*. It broadcasts *datasetB* to all nodes available in the cluster. Then, each node performs a join operation, using the portion of *datasetA* available locally. The execution steps of this algorithm are described below:
  1. *datasetB* is collected at a single node of the cluster (also called driver node).
  2. A hashed structure of *datasetB* is built locally on the driver node.
  3. Hashed *datasetB* is broadcast to all the nodes.
  4. The broadcast *datasetB* is joined with local portions of *datasetA* in parallel, using the hash join algorithm.
- **Sort-merge Join.** This algorithm performs better for larger sizes of *datasetB*. It can also be used when the actual size of *datasetB* is unknown and cannot be estimated accurately. It performs a shuffling (i.e. repartitioning) of both *datasetA* and *datasetB* on all nodes of the cluster and then joins together the local subsets. Sort-merge Join

is a more decentralized algorithm compared to Broadcast Hash Join, at the cost of potentially higher network bandwidth consumption.

1. *datasetA* and *datasetB* are repartitioned (shuffled) using the same *partitioner*<sup>8</sup> on their respective join keys. Thus, records from both datasets will reside on the same node, if and only if these records share the same join keys.
2. Each local subset of *datasetA* is sorted in parallel on all nodes.
3. The Sort-merge Join algorithm is applied on the subsets of sorted *datasetA* and *datasetB*.

### 5.3.3 Projection operator

The physical Projection operator takes as input a table and keeps only a subset of its columns. Additionally, by specifying new columns names, the resulting set of columns may also be renamed accordingly. In DiStRDF processing layer, the Projection operation is typically applied on the resulting set of records, in order to keep only the columns that the user has specified. It should be mentioned that Projection push-down is also supported by Parquet, since the underlying data organization is column-based. This means that whenever only few columns need to be retrieved from a table, the data in the remaining columns is not fetched from disk.

### 5.3.4 Decode operator

Since the underlying data is stored encoded in the storage layer, the processing engine handles it internally as integers. However, an encoded result set has little value for a typical user. Hence, the result set needs to be decoded prior to being delivered to her. To this end, we use the physical Decode operator, which takes as input a table and decodes the values of a subset of its columns in a distributed way, by using the Dictionary. The decision to use an in-memory distributed store for keeping the Dictionary, is largely due to the requirement that the Decode operator should have minimal impact on the overall performance of query evaluation.

## 5.4 Spatio-temporal logical query plans

Consider the case of a query, which is defined by a non spatio-temporal SPARQL query  $Q$ , and a spatio-temporal range constraint  $q$ . In abstract terms, the evaluation process of such a query, consists of two parts: (a) processing the triple patterns  $Q$  to find qualifying triples, and (b) processing the spatio-temporal query  $q$  to find matching tuples in spatio-temporal terms. The final result is the intersection of these intermediate results.

The gist of our approach is that given the spatio-temporal one-dimensional encoding introduced in Section 4.1, *we can efficiently prune data by exploiting the approximate spatio-temporal information embedded in encoded subject fields*. This approach has the advantage that it applies spatio-temporal filtering by considering just a single field (the subject field), avoiding the need for an early-stage filtering on the exact spatio-temporal information. This approximate filtering can produce false positives, i.e., tuples that do not actually match the spatio-temporal constraint  $q$ . Therefore, the produced result set needs to

<sup>8</sup>A partitioner is a mechanism that determines the location (i.e. node) of each record, on the repartitioning process.



go through a refinement phase, in order to discard false positives from the final result set. Notice that since the stored data are organized as encoded property tables, the spatial and temporal fields are available to the refinement phase without needing any additional join operations. However, a decoding operation is mandatory during the refinement, since the range constraint cannot be applied directly on the encoded spatial and temporal values. In Section 6 we empirically evaluate the performance improvements gained by adopting the approximate filtering in real-world data.

To support such spatio-temporal operations in DiStRDF processing layer, we have introduced, in Section 5.2, two additional logical filtering operators, namely the Spatio-temporal filter estimation operator ( $\sigma_\epsilon$ ) and Spatio-temporal filter refinement operator ( $\sigma_\rho$ ). Essentially, the first operator applies the approximate spatio-temporal filtering on subject fields, while the second operator applies exact spatio-temporal filtering by examining the spatial and temporal fields. As already mentioned, a  $\sigma_\rho$  operator implies internally a decoding operation also, for the spatial and temporal fields.

To exemplify, Fig. 7 demonstrates the case of a SPARQL query, where the left part shows (part of) an RDF graph and the right part depicts a query on that graph, consisting of three triple patterns and a spatio-temporal range constraint. Essentially the RDF graph is composed of a *semantic node* (node1) which has a specific `rdf:type` property and a set of observation values, such as its speed, its status, its spatial and temporal information, etc. The query's goal is to retrieve the *semantic nodes* along with their status, heading direction and movement speed, while satisfying a spatio-temporal range constraint. The SPARQL query containing these triple patterns has already been introduced in Fig. 5.

While building the logical plan of the example query in DiStRDF processing layer, the triple patterns are mapped to the Selection operators  $\sigma_1$ ,  $\sigma_2$  and  $\sigma_3$ , as already discussed in Section 5.2. Moreover, due to having a spatio-temporal range constraint  $q$  in the query, we also include two additional operators to it, namely  $\sigma_\epsilon$  and  $\sigma_\rho$ . The  $\sigma_\epsilon$  operator is always pushed as low as possible in the logical plan. This enables early filtering of the result set, thus reducing the amount of data forwarded to the parent operators. However, the refinement phase ( $\sigma_\rho$ ) may either be appended at the end of the query plan, or pushed down, just above the  $\sigma_\epsilon$  operator. The logical query plans produced by following these two alternatives are depicted in Fig. 8a and b respectively.

More specifically, the query plan depicted in Fig. 8a, first evaluates the approximate spatio-temporal filtering operator  $\sigma_\epsilon$  and the Selection operator  $\sigma_1$  and joins them

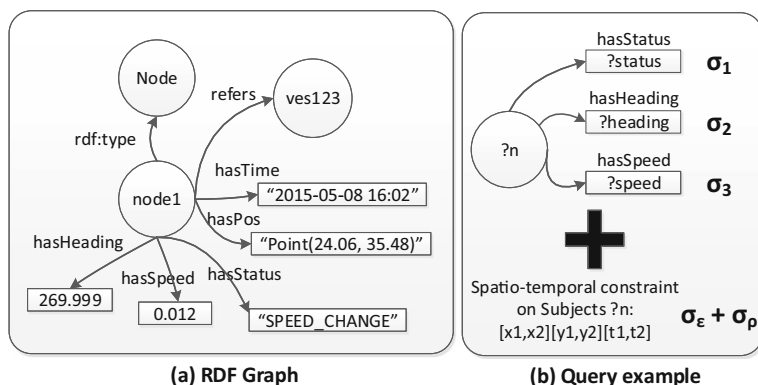
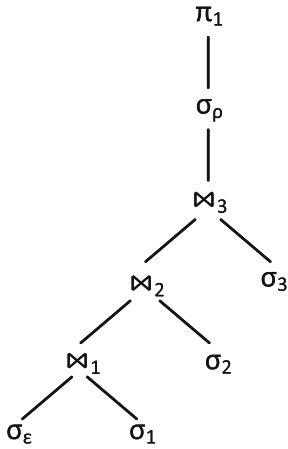
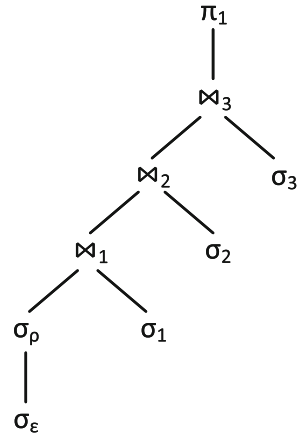


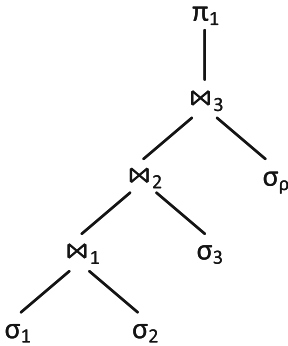
Fig. 7 Example of RDF graph and SPARQL query with spatio-temporal constraints



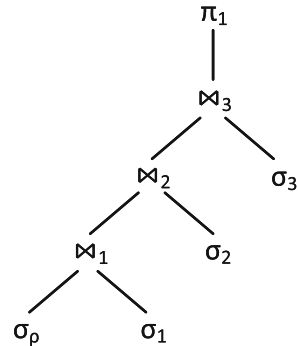
(a) Late refinement with 1D filtering



(b) Early refinement with 1D filtering



(c) Late refinement without 1D filtering



(d) Early refinement without 1D filtering

**Fig. 8** Logical query plans with spatio-temporal operators

on their respective subject fields. This ensures that the output of  $\bowtie_1$  operator satisfies both  $\sigma_\epsilon$  and  $\sigma_1$  filters. Then,  $\sigma_2$  and  $\sigma_3$  are joined to the result set, and the output of  $\bowtie_3$  operator is provided to  $\sigma_\rho$ . This operator first decodes the spatial and temporal columns and then applies the exact spatio-temporal filtering, based on those fields. Finally, the  $\pi_1$  operator keeps only the fields requested by the user. Letting the refinement phase to take place at the end of the query plan, is expected to improve performance on queries which produce small result sets, i.e. its triple patterns part, has small output size.

The second query plan, depicted in Fig. 8b, applies the exact spatio-temporal filtering, immediately on the results of  $\sigma_\epsilon$ . In the rest of the plan, the operators  $\sigma_1$ ,  $\sigma_2$  and  $\sigma_3$  are joined with  $\sigma_\rho$ , and their output is forwarded to  $\pi_1$ . This approach is expected to benefit by

queries which have a tighter spatio-temporal range, while their triple patterns part produces large output size.

We demonstrate the performance benefit of the proposed approximate filtering, by providing two additional logical plans depicted in Fig. 8c and d. These two plans do not perform an approximate filtering, letting the refinement operator to filter out all data which do not satisfy the spatio-temporal range constraint. The plan depicted in Fig. 8c applies spatio-temporal filtering at the end of the query evaluation, while the fourth plan in Fig. 8d, pushes the spatio-temporal filter as low as possible.

In the rest of this paper, the first alternative of performing late refinement with approximate filtering (Fig. 8a) will be referred as LP1, while the second alternative (Fig. 8b) which pushes down the refinement operation, while keeping the approximate filtering, will be cited as LP2. The remaining alternatives (Fig. 8c) and d, which drop the use of the approximate filtering, will be quoted as LP3, and LP4 respectively.

## 6 Experimental evaluation

In this section, we present the results of our experimental study. Our algorithms are implemented using Scala 2.11 and Apache Spark 2.1. We deployed our code on a proprietary cluster of 10 physical nodes, each having 128GB RAM and a 6-core 1.7GHz processor. All nodes are running Ubuntu 16.04.

### 6.1 Experimental setup

**Data sets** We used surveillance information from the maritime and aviation domains. The maritime surveillance data concern the entire month of January 2016, covering the Mediterranean Sea and part of the Atlantic Ocean. The aviation surveillance data refer to a week of April 2016, covering the entire space of Europe. Furthermore, we used a data set which contains flight plan data covering the entire area of Europe for the month of April 2016. The datAcron ontology [23] was used to represent all data in RDF format.

The total size of the surveillance data set is roughly 400 million triples: 300 million triples from the maritime and 100 million triples from the aviation domain. The flight plans data set consists of approximately 1.2 billion triples. All triples were encoded to integer values using the method described in Section 4.1 to form the one-triples tables; the size of the resulting data set is approximately 45GB in text format or 18GB in Parquet format using snappy compression. We also built property tables for the *Semantic Node* and *Vessel* entities of datAcron Ontology to enable efficient access to their corresponding properties during query execution. In this section, we experiment with data stored in HDFS using Parquet file format, to enable efficient access for our Spark applications and benefit by columnar storage, compression and predicate push-down.

A dictionary containing the mapping between encoded and decoded values was also created and stored in a Redis cluster instance, running on our cluster, with no replication enabled. The total number of records (key-value pairs) stored in the Redis dictionary is approximately 400 million.

**Configuration** We configured Spark on YARN, using Hadoop 2.7.2. One node was set to be the driver node, while the others contain the Spark Executors. All experiments conducted, use executors with 5 CPU cores and 8GB memory. We experiment with various number of

executors. HDFS is configured with replication factor of 3. We also used the Jedis<sup>9</sup> library, to communicate with the Redis cluster.

**Type of queries** We conducted experiments using 9 real-world SPARQL queries for surveillance maritime and aviation domains and also for the flight plan domain. Queries 1-3 refer to the maritime surveillance domain, 4-6 to the aviation surveillance domain and 7-9 to the flight plan data. Queries 1, 4 and 7 are comprised of three triple patterns each and require no join operation to be evaluated, since their triple patterns are already joined in the property tables formatted data source. Queries 2, 5 and 8 are composed of four triple patterns each, requiring a single join operation to be actually evaluated. The rest queries 3, 6 and 9 contain five triple patterns, and require 2 join operations to be evaluated during query execution. All queries are presented in SPARQL format in Appendix A.

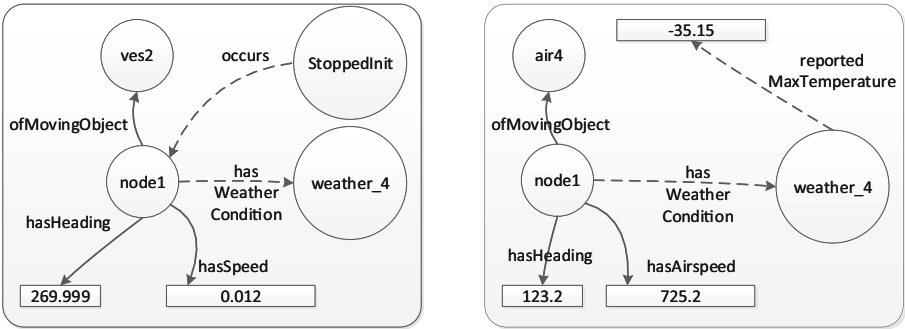
Figure 9 shows the parts of the RDF graphs which correspond to the evaluated queries. The first graph (Fig. 9a), depicts the RDF graph for the first three queries, which refer to the maritime surveillance domain, the second graph (Fig. 9b), demonstrates the RDF graph for queries 4 to 6, on the aviation surveillance domain and the third graph (Fig. 9c) shows the graph for queries 7 to 9, concerning the flight plans data. For convenience, the relationships between RDF resources which are pre-computed in the property tables, are depicted as solid lines. Hence, the dashed lines represent the join operations, which should be computed during execution time. For example, the property *hasWeatherCondition* is stored in the leftover triples data source, while the *hasHeading* property is stored in the property table; the first property requires a join to compute its relationship with *node1*, while for the second property, the join is pre-computed in the data source. Interestingly although queries 3, 6 and 9 require two join operations each, they actually perform different types of queries: Query 3 performs two star join operations, query 6 performs chain join operations and query 9 performs a star join and a chain join operation.

Furthermore, we added a spatio-temporal range constraint to all of our experiments. We use various ranges of different sizes, based on the volume of total space and time that they cover, namely 10%, 20% and 40%.

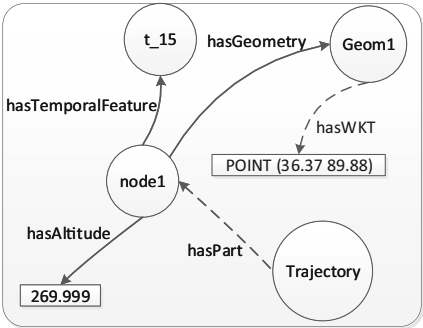
**Metrics** Our main evaluation metric is the execution time needed for each SPARQL query to be processed on the Spark cluster. We focus on measuring the actual execution time needed for our queries to be evaluated, thus omitting (a) any overhead caused by Spark initialization processes and (b) the time needed to store the result set in HDFS. In technical terms, we measure only the time needed to calculate the result set in main memory, by executing a call to the count method of the Spark DataFrame containing this result. Moreover, we ran each experiment 10 times, as a warm-up procedure, and report only the time needed for the 11th execution. This warm-up procedure, ensures that the cost of the Java JIT compiler and the overhead for establishing connections to the Redis cluster from all YARN containers is also omitted.

**Methodology** All of our experiments are executed on both data sets. Furthermore, for each experiment, we also vary the size of the spatio-temporal range constraint. First, we study the performance of each logical planning strategy, namely LP1, LP2, LP3 and LP4. Then, we opt to the best performing logical planning strategy (LP1) to evaluate the scalability of DiStRDF, by varying the number of Spark executors. Lastly we experiment with Sort-merge

<sup>9</sup><https://github.com/xetorthio/jedis>



(a) RDF graph for maritime surveillance data. (b) RDF graph for aviation surveillance data.



(c) RDF graph for flight plan data.

Fig. 9 RDF graphs for experimental queries

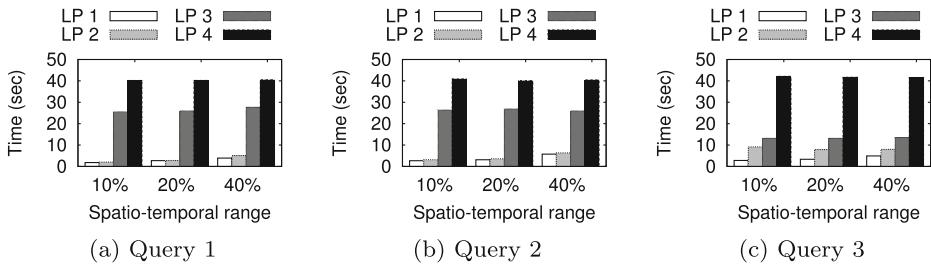
Join and Broadcast Hash Join physical operators. Our experimental setup is summarized in Table 2, which marks default values in bold.

6.2 Results

**Comparing the performance of logical plans** Figures 10, 11 and 12 demonstrate the execution times needed to evaluate queries from surveillance maritime, surveillance aviation

Table 2 Experimental setup parameters (default values in bold)

Parameter	Values
Spatio-temporal range sizes	10%, 20%, 40% (of the data set spatio-temporal size)
Queries	Surveillance Maritime: (Q1, Q2, Q3), Surveillance Aviation: (Q4, Q5, Q6), Flight Plans: (Q7, Q8, Q9)
Logical plans	<b>LP1</b> , LP2, LP3, LP4
Physical plans	<b>Sort-Merge Join</b> , Broadcast Hash Join
Number of Spark executors	2, 4, <b>9</b>



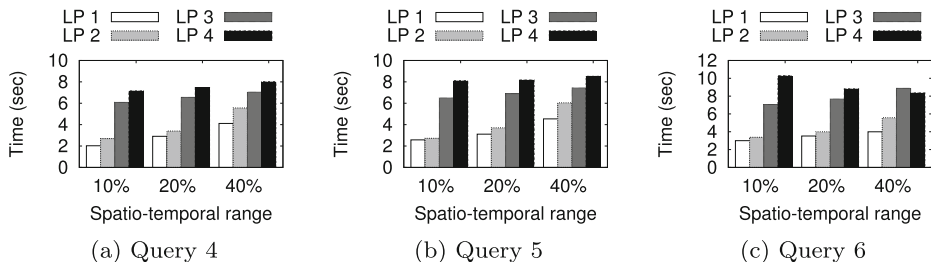
**Fig. 10** Performance of *DiStRDF* when using various logical plans on maritime surveillance data

and flight plan domains respectively, by using various logical query plans. Generally, as expected, queries from the surveillance aviation domain need less execution time, since the size of the aviation data is smaller. Also, all queries perform much better on plans LP1 and LP2, namely by including the approximate filtering selection  $\sigma_\epsilon$  operation in their execution plans. The benefit by applying the approximate spatio-temporal filtering, varies from an order of magnitude to 100%. These results, justify the efficiency of our approximate filtering, compared to the regular spatio-temporal filter.

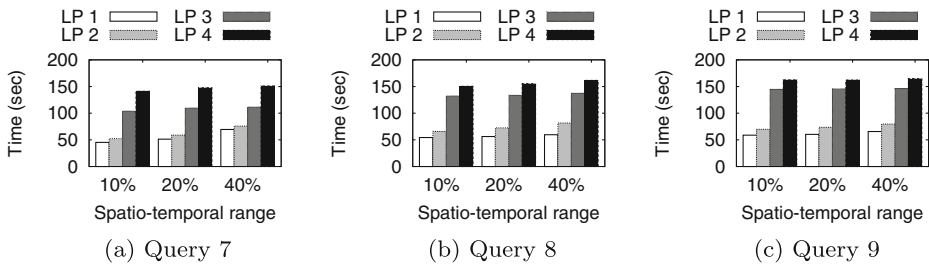
More specifically, Fig. 10a depicts the performance of Query 1, which performs no join operations. Logical plans LP3 and LP4, require roughly constant execution time to be evaluated, regardless of the size of the spatio-temporal constraint. In these logical plans, all the underlying tuples are checked against the exact spatio-temporal filter, thus requiring the same filtering cost for all range sizes. Pushing the refinement operation  $\sigma_\rho$  as low as possible (LP4), indicates the worst performance, since all tuples in store are checked against the costly spatio-temporal filter. Regular Selection operators  $\sigma_1$ ,  $\sigma_2$  and  $\sigma_3$  have lower processing cost, compared to the  $\sigma_\rho$ , thus improving the overall performance by executing them prior to  $\sigma_\rho$ . Plans LP1 and LP2 perform similarly when no join operations are included in the query plan. Their execution time increases slightly for larger spatio-temporal range sizes, because more tuples survive by the low-cost approximate filter. Notice that tuples which are not pruned by the approximate filtering, need to be processed by the exact filtering also.

Figure 10b depicts the execution times needed for Query 2, which performs a single join operation. The performance of all logical plans is not greatly affected by the single join operation. As in the previous experiment, plans LP1 and LP2 perform better than LP3 and LP4 for all spatio-temporal ranges examined. Plans LP3 and LP4 have approximately constant execution times, while the performance of LP1 and LP2 decreases slightly for larger spatio-temporal sizes.

The execution times of Query 3 are depicted in Fig. 10c. During its evaluation, this query performs 2 join operations, which cause its result size to be greatly reduced due to having



**Fig. 11** Performance of *DiStRDF* when using various logical plans on aviation surveillance data



**Fig. 12** Performance of *DiStRDF* when using various logical plans on flight plan data

small number of tuples satisfying the join conditions in the store. Hence, the logical plans which perform refinement at the end of the query execution (LP1, LP3) benefit by their reduced input size. As such, LP1 is better alternative than LP2 and LP3 is much better than LP4. However the efficiency of our approximate filtering poses LP2 to be still slightly better than LP3 for all range sizes.

Results in Fig. 11a, for the surveillance aviation domain, show that by using lower size of input data, the cost of Selection operations is considerably lower, thus improving the performance of plans LP3 and LP4. Interestingly, the LP3 needed just 1.5 seconds more than LP2 to evaluate the 40% range size query, due to high cost of  $\sigma_\rho$  operator, especially when pushed low in the logical plan. However, LP1 is still the best choice between the execution plans examined, due to the efficiency provided by our approximate spatio-temporal filtering.

Figure 11b depicts the results of Query 5, which performs a single join operation during its execution. As in the maritime domain, the effect of a single join operation negligible, leading to similar performance as Query 4.

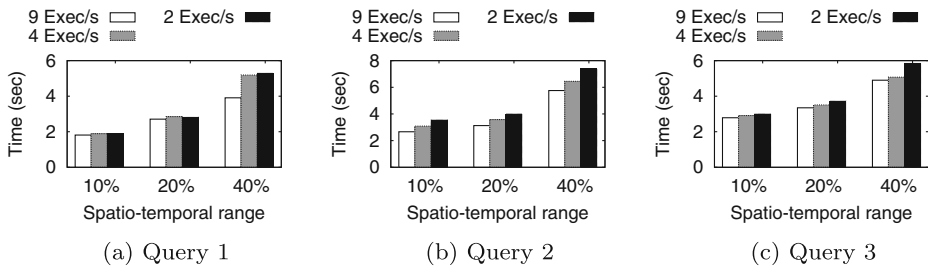
The results of Query 6, depicted in Fig. 11c, show that logical plans LP3 and LP4 continue to perform worse than LP1 and LP2. LP1 is again the most efficient plan. Hence, we opt to use LP1 on all of our remaining experiments.

Figure 12a depicts the execution times needed to evaluate Query 7 on flight plans data, for different logical plans. Generally, queries on flight plans data need more time to be evaluated, since the amount of the containing triples is much larger. However, the benefit by using our *DiStRDF* system, along with the spatio-temporal encoding, is evident by comparing the LP1 to LP3 and LP4. LP1 benefits by our spatio-temporal filtering to achieve much better efficiency. LP2 needs approximately the same execution time as LP1, since it also utilizes the same spatio-temporal filtering operations, but in different order.

The execution times of the logical plans of Query 8, are depicted in Fig. 12b. Naturally Query 8 needs slightly more execution time, when compared to Query 7, since the former requires a join operation. Logical plans LP1 and LP2 prove again to be more efficient than LP3 and LP4, showcasing the benefits of our employed spatio-temporal filtering.

Figure 12c depicts the results for Query 9 on flight plans data. It requires more time to be evaluated, compared to the other flight plan queries, since it evaluates two join operations. LP1 is the most efficient logical plan, while LP4 is the worst.

**Comparing the performance by varying the number of executors** Figures 13, 14 and 15 demonstrate the execution times needed to evaluate queries for surveillance maritime, surveillance aviation and flight plan data respectively, by varying the number of employed Spark Executors. Essentially, this set of experiments is an indication of the scalability of our approach, since they demonstrate the case of having fewer nodes to evaluate the result set.



**Fig. 13** Performance of *DiStRDF* when varying the number of Spark Executors on maritime surveillance data

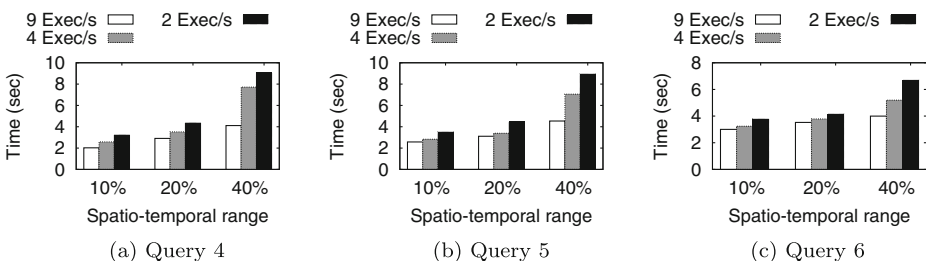
By using a spatio-temporal range query, Spark performs data processing only on the subset of executors which contain data satisfying that constraint. Since our data are partitioned by their spatio-temporal position, a range query might be satisfied by using only a small set of executors. As such, the case of the 10% range size query, clearly benefits by this partitioning scheme, since in Fig. 13a, b and c its performance is not affected by the number of executors.

However, in surveillance aviation queries (Fig. 14a, b and partitioning scheme, since in), the 10% range size query benefits slightly less by the partitioning scheme, since more nodes are involved to the query evaluation, especially in Queries 5 and 6. The results in the rest of this set of experiments, perform as expected: more executors lead to higher efficiency, especially for the 40% range size queries, which process the larger volume of data. Hence, our *DiStRDF* system is able to efficiently process higher volume of data, by providing more executors to the cluster.

The experiments with flight plans data, in Fig. 15a, b and c demonstrate the case, where data are large enough to not fit in memory. Since, our executors are configured to use 8GB of memory, by using 2 or 4 executors, Spark is not able to fit in memory the entire flight plan data set. However, the execution times measured in these experiments are close to linear, since our partitioning scheme is able to prune early in the execution stage the partitions that do not satisfy query constraints. As such, the remaining partitions are able to fit in main memory and there we avoid spilling the data into hard disk drives.

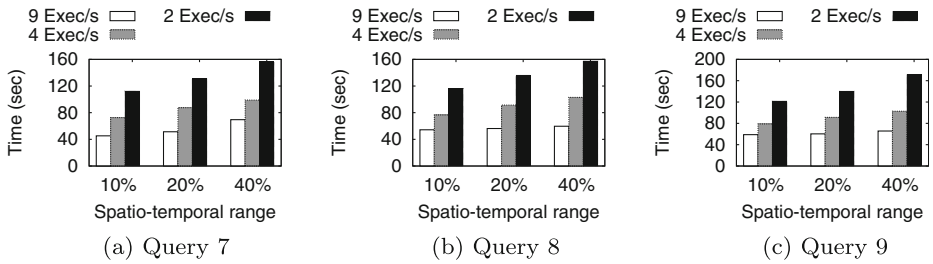
**Comparing the performance of physical join operators** Figures 16, 17 and 18 demonstrate the impact of query evaluation performance for surveillance maritime, surveillance aviation and flight plans data respectively, by selecting different physical join operators. Queries 1, 4 and 7, do not perform a join operation, thus are excluded from this set of experiments.

Generally, the Sort-merge join operator performs better than the Broadcast join operator. These results can be justified by the fact that a broadcast hash join performs better for small sets of input data, whereas the queries of our experiments provide large sets of data

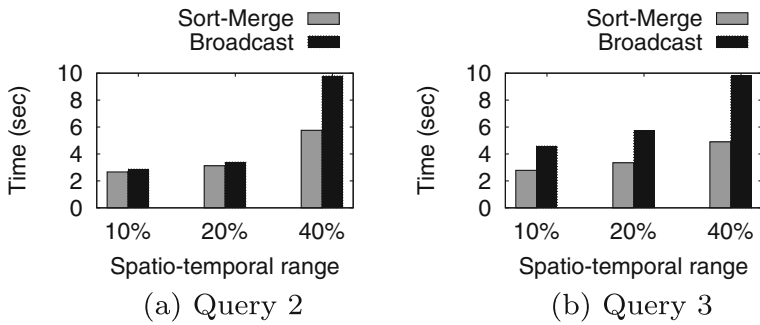


**Fig. 14** Performance of *DiStRDF* when varying the number of Spark Executors on aviation surveillance data



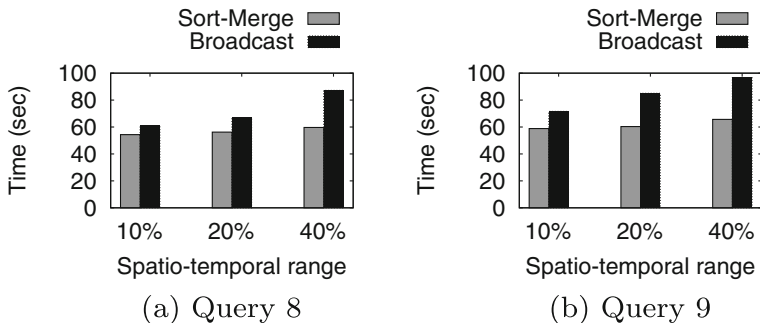
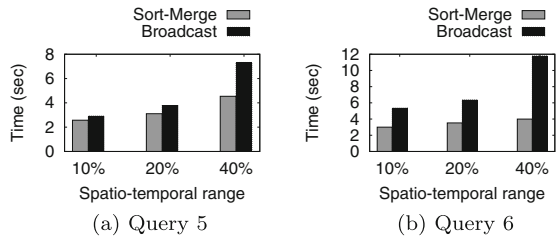


**Fig. 15** Performance of *DiStRDF* when varying the number of Spark Executors on flight plan data



**Fig. 16** Performance of *DiStRDF* when varying the physical join operator on maritime surveillance data

**Fig. 17** Performance of *DiStRDF* when varying the physical join operator on aviation surveillance data



**Fig. 18** Performance of *DiStRDF* when varying the physical join operator on flight plan data

to the join operators. Sort-merge operator also benefits by the the executor's shared memory between executor cores. By exploiting the executor's shared memory, Spark is able to exchange data between executor cores without transferring them over the network. Since we used 5 CPU cores per executor, our experiments had plenty of data being exchanged locally on the executor's shared memory. It is also worth noting that Sort-merge Join algorithm, as implemented by Spark SQL API, performs a re-partitioning of the entire data set, to a user configurable number of partitions. This number was set to be 20 during this set experiments.

As expected, the experiments having 40% range size, perform worse than the others, since the approximate filtering is able to prune less data early. This leads to more data being forwarded to the parent join operators, thus increasing their computational cost, especially for the case of Broadcast Hash Join operators. Also, the impact of selecting a physical join operator is less significant in Figs. 16a, 17a and 18a, when compared to Figs. 16b, 17b and 18b respectively, since in the former queries, only a single join is computed during query execution.

## 7 Conclusions

In this paper, we present the first parallel and scalable in-memory solution to the problem of spatio-temporal SPARQL query processing. Our proposed *DiStRDF* system, which comprises of a *Processing* and a *Storage* Layer, is designed to benefit by the tools and best practices for handling vast-sized data. Notable features of the proposed solution include the support for different query execution plans, as well as efficient storage of spatio-temporal data on different data organizations. Our experiments demonstrate the performance of our system, which is able to efficiently process RDF queries with spatio-temporal range constraints in a few seconds.

The current status of the implemented DiStRDF system, supports several real-world SPARQL queries, which are mostly defined by triple patterns and a spatio-temporal range constraint. In the future, we plan to extend our system to cover a larger part of the SPARQL specification, including support for filtering, grouping and sorting operations, as well as facilitate data transformations by enabling the use of SPARQL functions on the underlying data. Additionally, we plan to improve the performance of *DiStRDF Processing Layer* by implementing query optimization techniques, such as join reordering.

**Acknowledgements** This work is supported by the datAcron project, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 687591.

## Appendix

### A SPARQL queries used In experiments

#### SPARQL Query 1 (Maritime Domain)

```
Prefix : <http://www.datacron-project.eu/datAcron#>
SELECT * WHERE {
    ?n :ofMovingObject ?ves ;
    :hasHeading ?heading
    :hasSpeed ?speed .
}
```

**SPARQL Query 2 (Maritime Domain)**

Prefix : &lt;http://www.datacron-project.eu/datAcron#&gt;

```
SELECT * WHERE {  
  ?n :ofMovingObject ?ves ;  
  :hasHeading ?heading ;  
  :hasSpeed ?speed ;  
  :hasWeatherCondition ?w .  
}
```

**SPARQL Query 3 (Maritime Domain)**

Prefix : &lt;http://www.datacron-project.eu/datAcron#&gt;

```
SELECT * WHERE {  
  ?n :ofMovingObject ?ves ;  
  :hasHeading ?heading ;  
  :hasSpeed ?speed ;  
  :hasWeatherCondition ?w .  
  :StoppedInit :occurs ?n .  
}
```

**SPARQL Query 4 (Maritime Domain)**

Prefix : &lt;http://www.datacron-project.eu/datAcron#&gt;

```
SELECT * WHERE {  
  ?n :ofMovingObject ?aircraft ;  
  :hasHeading ?heading ;  
  :hasAirspeed ?speed .  
}
```

**SPARQL Query 5 (Maritime Domain)**

Prefix : &lt;http://www.datacron-project.eu/datAcron#&gt;

```
SELECT * WHERE {  
  ?n :ofMovingObject ?aircraft ;  
  :hasHeading ?heading ;  
  :hasAirspeed ?speed ;  
  :hasWeatherCondition ?w .  
}
```

**SPARQL Query 6 (Maritime Domain)**

Prefix : &lt;http://www.datacron-project.eu/datAcron#&gt;

```
SELECT * WHERE {  
  ?n :ofMovingObject ?aircraft ;  
  :hasHeading ?heading ;  
  :hasAirspeed ?speed ;  
  :hasWeatherCondition ?w .  
  :reportedMaxTemperature ?temp .  
}
```

### SPARQL Query 7 (Maritime Domain)

Prefix : <http://www.datacron-project.eu/datAcron#>  
 Prefix dul: <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#>  
 SELECT \* WHERE {  
     ?n dul:hasTemporalFeature ?temporal ;  
     :hasAltitude ?altitude ;  
     :hasGeometry ?geom .  
 }

### SPARQL Query 8 (Maritime Domain)

Prefix : <http://www.datacron-project.eu/datAcron#>  
 Prefix dul: <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#>  
 SELECT \* WHERE {  
     ?n dul:hasTemporalFeature ?temporal ;  
     :hasAltitude ?altitude ;  
     :hasGeometry ?geom .  
     ?geom :hasWKT ?wkt .  
 }

### SPARQL Query 9 (Maritime Domain)

Prefix : <http://www.datacron-project.eu/datAcron#>  
 Prefix dul: <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#>  
 SELECT \* WHERE {  
     ?n dul:hasTemporalFeature ?temporal ;  
     :hasAltitude ?altitude ;  
     :hasGeometry ?geom .  
     ?geom :hasWKT ?wkt .  
     ?traj dul:hasPart ?n .  
 }

## References

1. Abdelaziz I, Harbi R, Khayyat Z, Kalnis P (2017) A survey and experimental comparison of distributed SPARQL engines for very large RDF data. *PVLDB* 10(13):2049–2060
2. Alarabi L, Mokbel MF, Musleh M (2017) St-hadoop: a mapreduce framework for spatio-temporal data. In: *Advances in spatial and temporal databases - 15th international symposium, SSTD 2017, Arlington, VA, USA, August 21-23, 2017, Proceedings*, pp 84–104
3. Bereta K, Smeros P, Koubarakis M (2013) Representation and querying of valid time of triples in linked geospatial data. In: *The Semantic web: semantics and big data, 10th international conference, ESWC 2013, Montpellier, France, May 26-30, 2013. Proceedings*, pp 259–274
4. Blanas S, Patel JM, Ercegovac V, Rao J, Shekita EJ, Tian Y (2010) A comparison of join algorithms for log processing in mapreduce. In: *Proceedings of the ACM SIGMOD international conference on management of data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pp 975–986. <https://doi.org/10.1145/1807167.1807273>
5. Curé O, Blin G (2014) *RDF database systems: triples storage and SPARQL query processing*. Elsevier
6. Doukeridis C, Nørsvåg K (2014) A survey of large-scale analytical query processing in mapreduce. *VLDB J* 23(3):355–380
7. Eldawy A, Mokbel MF (2015) Spatialhadoop: a mapreduce framework for spatial data. In: *31st IEEE international conference on data engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pp 1352–1363

8. Garbis G, Kyzirakos K, Koubarakis M (2013) Geographica: a benchmark for geospatial rdf stores (long version). In: International semantic web conference, pp 343–359. Springer
9. Giannousis K, Bereta K, Karalis N, Koubarakis M (2018) Distributed execution of spatial SQL queries. In: IEEE international conference on big data, big data 2018, Seattle, WA, USA, December 10–13, 2018, pp 528–533. <https://doi.org/10.1109/BigData.2018.8621908>
10. Hagedorn S, R  th T. (2017) Efficient spatio-temporal event processing with STARK. In: Proceedings of the 20th international conference on extending database technology, EDBT 2017, Venice, Italy, March 21–24, 2017, pp 570–573
11. Husain MF, Doshi P, Khan L, Thuraisingham BM (2009) Storage and retrieval of large rdf graph using hadoop and mapreduce. *CloudCom* 9:680–686
12. Kaoudi Z, Manolescu I (2015) RDF in the clouds: a survey. *VLDB J* 24(1):67–91
13. Kim H, Ravindra P, Anyanwu K (2011) From SPARQL to mapreduce: the journey using a nested triplegroup algebra. *PVLDB* 4(12):1426–1429
14. Koubarakis M, Karpathiotakis M, Kyzirakos K, Nikolaou C, Sioutis M (2012) Data models and query languages for linked geospatial data. In: Reasoning web. Semantic technologies for advanced query answering - 8th international summer school 2012, Vienna, Austria, September 3–8, 2012. Proceedings, pp. 290–328. [https://doi.org/10.1007/978-3-642-33158-9\\_8](https://doi.org/10.1007/978-3-642-33158-9_8)
15. Koubarakis M, Kyzirakos K (2010) Modeling and querying metadata in the semantic sensor web: the model strdf and the query language stsparql. In: The Semantic web: research and applications, 7th extended semantic web conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part I, pp 425–439
16. Kyzirakos K, Karpathiotakis M, Bereta K, Garbis G, Nikolaou C, Smeros P, Giannakopoulou S, Dogani K, Koubarakis M (2013) The spatiotemporal RDF store Strabon. In: Proceedings of SSTP, pp 496–500
17. Liagouris J, Mamoulis N, Bouras P, Terrovitis M (2014) An effective encoding scheme for spatial RDF data. *PVLDB* 7(12):1271–1282
18. Naacke H, Amann B, Cur   O (2017) SPARQL graph pattern processing with apache spark. In: Proceedings of the 5th international workshop on graph data-management experiences & systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017, pp 1:1–1:7
19. Nikitopoulos P, Vlachou A, Doulkeridis C, Vouros GA (2018) Distrdf: distributed spatio-temporal RDF queries on spark. In: Proceedings of the workshops of the EDBT/ICDT 2018 joint conference (EDBT/ICDT 2018), Vienna, Austria, March 26, 2018, pp. 125–132. <http://ceur-ws.org/Vol-2083/paper-19.pdf>
20. Ravindra P, Kim H, Anyanwu K (2011) An intermediate algebra for optimizing rdf graph pattern matching on mapreduce. In: Extended semantic web conference, pp 46–61. Springer
21. Rohloff K, Schantz RE (2011) Clause-iteration with mapreduce to scalably query datagraphs in the SHARD graph-store. In: DDC'11, Proceedings of the 4th international workshop on data-intensive distributed computing, San Jose, CA, USA, June 8, 2011, pp 35–44
22. Santipantakis GM, Glenis A, Patroumpas K, Vlachou A, Doulkeridis C, Vouros GA, Pelekis N, Theodoridis Y (2018) Spartan: semantic integration of big spatio-temporal data from streaming and archival sources. *Future Generation Comp Syst*
23. Santipantakis GM, Vouros GA, Doulkeridis C, Vlachou A, Andrienko GL, Andrienko NV, Fuchs G, Garcia JMC, Martinez MG (2017) Specification of semantic trajectories supporting data transformations for analytics: the datacron ontology. In: Proceedings of the 13th international conference on semantic systems, SEMANTICS 2017, Amsterdam, The Netherlands, September 11–14, 2017, pp 17–24
24. Sch  tzle A, Przyjaci  l-Zablocki M, Berberich T, Lausen G (2015) S2X: graph-parallel querying of RDF with graphx. In: Biomedical data management and graph online querying - VLDB 2015 workshops, Big-O(Q) and DMAH, Waikoloa, HI, USA, August 31 - September 4, 2015, Revised Selected Papers, pp 155–168
25. Sch  tzle A, Przyjaci  l-Zablocki M, Hornung T, Lausen G (2013) Pigsparql: a SPARQL query processing baseline for big data. In: Proceedings of the ISWC 2013 posters & demonstrations track, Sydney, Australia, October 23, 2013, pp. 241–244
26. Sch  tzle A, Przyjaci  l-Zablocki M, Skilevic S, Lausen G (2016) S2RDF: RDF querying with SPARQL on Spark. *PVLDB* 9(10):804–815
27. Shi J, Qiu Y, Minhas UF, Jiao L, Wang C, Reinwald B,   zcan F (2015) Clash of the Titans: MapReduce vs. Spark for large scale data analytics. *PVLDB* 8(13):2110–2121
28. Tang M, Yu Y, Malluhi QM, Ouzzani M, Aref WG (2016) LocationSpark: a distributed in-memory data management system for big spatial data. *PVLDB* 9(13):1565–1568

29. Vlachou A, Doukeridis C, Glenis A, Santipantakis GM, Vouros GA (2019) Efficient spatio-temporal RDF query processing in large dynamic knowledge bases. In: Proceedings of the 34th annual ACM symposium on applied computing, SAC 2019, Limassol, Cyprus, April 08-12, 2019
30. Vouros GA, Vlachou A, Santipantakis GM, Doukeridis C, Pelekis N, Georgiou HV, Theodoridis Y, Patrourmpas K, Alevizos E, Artikis A, Claramunt C, Ray C, Scarlatti D, Fuchs G, Andrienko GL, Andrienko NV, Mock M, Camossi E, Joussemme A, Garcia JMC (2018) Big data analytics for time critical mobility forecasting: recent progress and research challenges. In: Proceedings of the 21th international conference on extending database technology, EDBT 2018, Vienna, Austria, March 26-29, 2018., pp 612–623
31. Xie D, Li F, Yao B, Li G, Zhou L, Guo M (2016) Simba: efficient in-memory spatial analytics. In: Proceedings of the 2016 international conference on management of data, SIGMOD conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, pp 1071–1085
32. You S, Zhang J, Gruenwald L (2015) Large-scale spatial join query processing in cloud. In: 31st IEEE international conference on data engineering workshops, ICDE workshops 2015, Seoul, South Korea, April 13-17, 2015, pp 34–41. <https://doi.org/10.1109/ICDEW.2015.7129541>
33. Yu J, Wu J, Sarwat M (2015) GeoSpark: a cluster computing framework for processing large-scale spatial data. In: Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems, pp 70:1–70:4
34. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I (2012) Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the USENIX conference on networked systems design and implementation (NSDI), pp 2–2

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



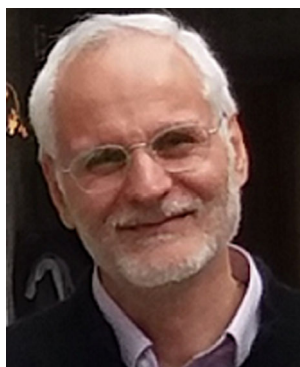
**Panagiotis Nikitopoulos** received his BSc degree in 2012 from the Department of Informatics of the Technical Educational Institute of Athens and his MSc degree in 2014 from the Department of Digital Systems of University of Piraeus. He is currently a PhD student at the Department of Digital Systems of University of Piraeus. His research interests lie in the area of Big Data, including distributed data management and query processing.



**Akrivi Vlachou** received the BSc and MSc degrees from the Department of Computer Science and Telecommunications of University of Athens in 2001 and 2003, respectively, and the PhD degree in 2008 from the Athens University of Economics and Business (AUEB). She is currently a post-doctoral researcher at the University of Piraeus. Her research interests include query processing and data management in large-scale distributed systems.



**Christos Doulkeridis** received the BSc degree in electrical engineering and computer science from the National Technical University of Athens and the MSc and PhD degrees in Information Systems from the Department of Informatics of Athens University of Economics and Business. He is currently an assistant professor in the Department of Digital Systems of the University of Piraeus. His research interests include parallel and distributed data management, and data analytics. Further details concerning his work can be found in <https://www.ds.unipi.gr/prof/cdoulk/>.



**George A. Vouross** holds a BSc in Mathematics, and a PhD in Artificial Intelligence all from the University of Athens, Greece. Currently he is a Professor in the Department of Digital Systems in the University of Piraeus. He has done research in the areas of Expert Systems, Knowledge Management, Ontologies, and Agents and Multi-Agent Systems. He has extensively served as program chair and chair and member of organizing committees of national and international conferences on related topics, member of steering committees and chair of the Hellenic A.I. Society board. Further details concerning his work can be found in <http://ai-group.ds.unipi.gr/georgev/>.