DiStRDF: Distributed Spatio-temporal RDF Queries on Spark

Panagiotis Nikitopoulos
Department of Digital Systems
University of Piraeus
nikp@unipi.gr

Akrivi Vlachou
Department of Digital Systems
University of Piraeus
avlachou@aueb.gr

Christos Doulkeridis
Department of Digital Systems
University of Piraeus
cdoulk@unipi.gr

George A. Vouros
Department of Digital Systems
University of Piraeus
georgev@unipi.gr

ABSTRACT

The ever-increasing size of data emanating from mobile devices and sensors, dictates the use of distributed systems for storing and querying the data. Typically, these data sources provide some spatio-temporal information, alongside other useful data. The task of interlinking and interchanging this kind of information is challenging in the case of large heterogeneity of data sources. This issue can be addressed by adopting the RDF data model, proposed by W3C. Hence, with respect to an application scenario which analyzes vast amount of spatio-temporal heterogeneous data, the task of efficiently evaluating spatio-temporal queries on RDF is crucial. In this paper, we address the problem of efficiently processing SPARQL spatio-temporal queries in parallel, by proposing the *DiStRDF* system. We use Spark, a well-known distributed in-memory processing framework, as the underlying processing engine. On top of it, we devise a set of query execution plans which exploit an 1D encoding scheme for improving the performance of our system. Our experimental evaluation demonstrates the efficiency of *DiStRDF* system.

1 INTRODUCTION

The Resource Description Framework (RDF) is a specification recommended by W3C[1] for modeling and interchanging data over the web. RDF data is represented as a set of triples (*subject*, *property*, *object*) or $\langle s, p, o \rangle$, also known as *statements*. SPARQL is a declarative language for querying RDF data sets. It relies on graph pattern matching queries to extract relevant data. A *triple pattern tp* is an RDF triple where variables may occur in subject, predicate or object position.

Nowadays, RDF has become a popular model for linking data from heterogeneous data sources. In the era of ever increasing size of RDF repositories (e.g. Google Knowledge Vault), it is imperative to produce efficient and scalable distributed solutions for RDF processing. Challenging issues, such as high availability and fault tolerance, are common in Big Data systems and need to be thoroughly studied. Existing systems that address these issues, typically operate on a set of computing nodes, where data and processing workloads are distributed among them.

The Hadoop ecosystem is a popular solution for addressing Big Data issues. Efficient performance of SPARQL queries over *MapReduce*, is a challenging task, which has attracted the attention of research community [8, 10, 14, 15, 18]. However, in-memory frameworks, such as Spark, typically provide better performance and scalability. Therefore, Spark has arguably become the most popular platform for parallel, in-memory, data processing.

Even though there exist some first approaches for parallel processing of in-memory RDF data [13, 17, 19], 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.

As an application scenario consider the case of surveillance data from moving objects (e.g., vessels or aircrafts) collected in real time from various data sources (i.e., radars, satellites) in heterogeneous formats, transformed to RDF format, linked with other external data sets and stored in a distributed storage system. Then, this data need to be efficiently queried, in order to retrieve useful information, such as "which vessels were moving in a particular area during the past month?". Such data analysis tasks that require advanced spatio-temporal queries over RDF data, are common in the case of datAcron project[2].

In this paper, we focus on spatio-temporal SPARQL queries, which apply a user-defined spatio-temporal constraint, alongside other SPARQL operators, on RDF *mobility nodes*, i.e. RDF nodes that contain spatio-temporal information. Nodes like these, might be parts of an object's trajectory, or other events of interest. More specifically, given a spatio-temporal query over horizontally partitioned RDF data, our goal is to provide an efficient and scalable distributed processing engine. For simplicity, for the RDF part of the query, we focus on queries expressed as *sets of triple patterns*, e.g., $\{tp_1, tp_2, \ldots, tp_n\}$. Although such sets do not cover the entire SPARQL specification, they compose a significant subset that can be used to express several queries, while they constitute one of the most challenging processing tasks for an RDF processing engine.

To support scalable and efficient management and querying of spatio-temporal RDF data, we propose the *DiStRDF system* (<u>Di</u>stributed <u>S</u>patio-<u>t</u>emporal <u>RDF</u> system) which comprises of two main modules: the *DiStRDF Storage Layer* and the *DiStRDF Processing Layer*. The *Storage Layer* performs the necessary management of data on a persistent storage, in order to enable fast data retrieval and high data availability, even in the case of hardware failures. Naturally, the design of the *Storage Layer* determines to a great extent the efficiency of the corresponding *Processing Layer*. Hence, the *DiStRDF Storage Layer* provides several

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

[2] http://www.datacron-project.eu/

options for accessing the stored data, to facilitate the selection of the best option at the time of query execution.

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 *DiStRDF 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.
- We propose the *DiStRDF Storage Layer* which stores encoded RDF triples and a dictionary of mappings between integer identifiers and RDF resources. Moreover, it provides various options for storing and accessing the encoded RDF data.
- We propose the *DiStRDF Processing Layer* which exploits an 1D encoding scheme for executing efficient spatio-temporal RDF queries, while providing the ability to choose between different query execution plans.
- We implement both *Processing* and *Storage* layers and demonstrate the efficiency of the proposed *DiStRDF* system.

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

2 RELATED WORK

Even though the topic of parallel processing of large-scale RDF data has attracted much attention recently (cf. [1, 9] 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 [13, 17, 19] 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 at very last stages of query processing.

Scalable processing of big spatial [5, 21, 23, 24] and spatio-temporal [2, 7] 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.

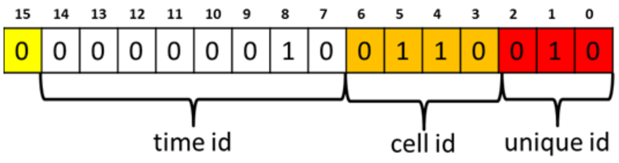


Figure 1: IDs encoding using bits: b total bits, m bits for spatial part (cell id), k bits for uniqueness, $b - (m + k + 1)$ bits for temporal part.

Existing works on spatio-temporal RDF data [3, 6, 11] 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 [12] 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 [22].

3 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 [4] 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 [22]. 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 4. 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 3.1), 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 3.2), (b) a large set of integer-encoded RDF triples (Section 3.3).

3.1 1D Encoding Scheme

Consider a regular spatial grid that partitions the 2D spatial domain into $2^m = (2^{m/2} * 2^{m/2})$ equi-sized cells. Also, 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. The only restriction is that the identical grid structure (i.e., 2^m equi-sized cells) is used for all temporal partitions T_i.

To encode the spatio-temporal information into an integer (ID), we consider its binary representation consisting of b bits (Figure 1). We set the most-significant bit to 0 for all IDs of spatio-temporal RDF entities, while it is set to 1 for IDs of all other RDF entities. We also keep m bits to represent the different 2^m spatial grid cells. Each spatial cell is assigned an m-bit identifier using a space-filling curve (Hilbert curve), in order to produce identifiers that respect the spatial locality of cells. Furthermore, we reserve k bits for assigning unique IDs to different entities in the same spatial cell for the same time partition. As such, the maximum number of entities that fit in a spatio-temporal (3D) cell is 2^k. This part of the ID is auto-incremented, and is encoded in the rightmost bits. Thus, $m + k$ bits are used for representing the identifiers of spatio-temporal entities of a single temporal partition. 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.

Example 3.1. In Figure 1, 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.

Given an ID of a spatio-temporal entity, the 3D grid 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 $s(\tau)$ 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 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 [22].

3.2 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.

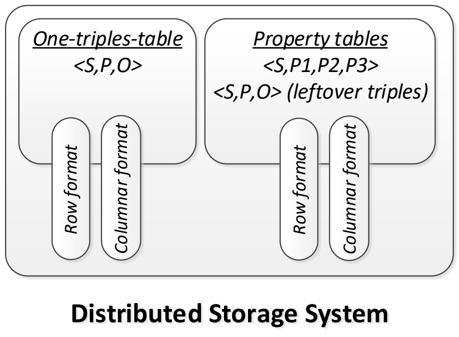


Figure 2: Design of distributed RDF storage.

- 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 popular REDIS[3], 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.

3.3 Storing RDF Triples

Distributed storage of RDF triples has been well-studied recently, due to the ever-increasing number and size of publicly available RDF data sets. The design of our distributed RDF store is generic and supports different options for storing and accessing data, as shown in Figure 2. Our premise is to make available several different options, and provide the ability for the administrator to select the desired query execution plan. Based on the literature, the following aspects guide the design of our RDF triples storage layer:

File Layout: Typical file layouts include row-based storage (e.g., the case of CSV files) and column-based storage (e.g., Parquet[4]). Both formats have advantages and disadvantages. For instance, it is well-established in the field of databases that columnar format achieves better compression and performs better for queries that retrieve few columns of a table only, while row format is better for queries that retrieve many columns. Our storage layer stores data in both layouts (CSV and Parquet); the desired layout can be picked at the time of query execution.

[3]https://redis.io/
[4]https://parquet.apache.org/

Data Organization: Encoded RDF data can be organized into an *one triples table*, where each row corresponds to a single encoded RDF statement. *Property tables* is another approach for organizing RDF data, where the row is expanded to include multiple statements. A single row of the table stores a set of property values which share a common subject. The number of the values that are stored together, can be specified at the design time, while the rest of them are stored as simple leftover triples, in an *one triples table. 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. Our storage layer currently supports the storage and handling of both *one triples table* and *property tables*.

Data Partitioning: The distribution of triples to storage nodes is also important and typically has a major impact on query performance. As SPARQL queries typically involve many joins, a distribution of triples that does not take into account the access patterns based on the query workload is going to hinder data locality. Inevitably, this will result to large data transfers over the network. Thus, a good data partitioning scheme for RDF data is one that processes large parts of the query locally at a node, avoiding the need of exchanging large intermediate results. With respect to the goal of this study, we expect that all queries are going to have a spatio-temporal constraint. Therefore, we consider as a good practice to 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 *mobility nodes*.

Indexing: With respect to indexed access to disk when loading an RDD in memory, we exploit the predicate pushdown mechanism offered by Spark in combination with Parquet storage. Essentially, this mechanism enables selective access to the stored data, by exploiting filters present in the query, in order to restrict access from disk explicitly on those blocks that contain data matching the existing filters.

In order to satisfy the above requirements, we opt to HDFS for storing RDF triples. HDFS is a generic distributed file system which is optimized for storing large sets of data. It can be used as a data source for Spark applications while supporting several file layouts, such as text and Parquet. HDFS supports partitioning by splitting files into blocks of fixed size (usually 128 MBs). File blocks are stored internally into different cluster nodes and can be processed locally when needed. Furthermore, blocks can be replicated among cluster nodes to support high data availability.

4 THE DISTRDF PROCESSING LAYER

The *DiStRDF Processing Layer* is a SPARQL query engine that supports scalable and efficient batch RDF 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 [25], since it is the most popular data processing engine, with the widest set of contributors, implementing the MapReduce model in main memory, thereby achieving significant performance gains to competitor systems [20], such as Hadoop.

In the following, we explain the basic Spark query operators which can be used for processing queries expressed as sets of triple patterns. Then, we pick a query example to be studied as a typical use case of the proposed *DiStRDF* system, in order to demonstrate the overall processing approach and its merits in terms of performance gains. We devise a set of logical plans based on this query, which will be examined later in the experimental evaluation.

4.1 Basic Query Operators

The very basic operators needed for querying RDF data include *selection*, *projection* and *join*. 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 the fundamental and challenging part of a parallel RDF processing engine.

4.1.1 Selection.

The *Selection* operator (σ) takes as input a triple pattern and returns all RDF triples that match the pattern. In the absence of an index, the *Selection* operator scans the RDF triples to identify matching triples. In the case of an index or sorted access to data, the *Selection* operator can be implemented more efficiently and avoid the complete scan.

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

4.1.2 Projection.

The *Projection* operator (π) takes as input a subset of Subject, Predicate, and Object, and returns only this subset for all RDF triples. In practice, it is useful for keeping only the part of RDF triples necessary for performing a subsequent processing step. As an example, consider selecting only the Subjects of all triples having predicate p_1 and object o_1. It should be mentioned that *projection pushdown* is also supported by Parquet.

4.1.3 Join.

The *Join* operator ($\bowtie$) takes as input two instances of RDF sets of triples and associates triples from both sets using some common part of the triples (e.g., Subject, Predicate, or Object), which can also be a variable. Join operators usually result to large amounts of data being exchanged over the network, thus optimizing its processing cost is often crucial. Notice that it is quite common to join one set of triples with itself, such as the case of searching *graph patterns*, i.e., triples that are linked together.

Given the fact that data is distributed, processing a join operator often 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.

The *DiStRDF Processing Layer* supports all the aforementioned operators at the logical level. These operators may be physically implemented using various algorithms. Typically, query engines support more than one physical implementations on each of the operators available. As such, it is crucial for a query engine to be able to choose effectively at runtime the most efficient physical implementation available in the system. In *DiStRDF Processing*

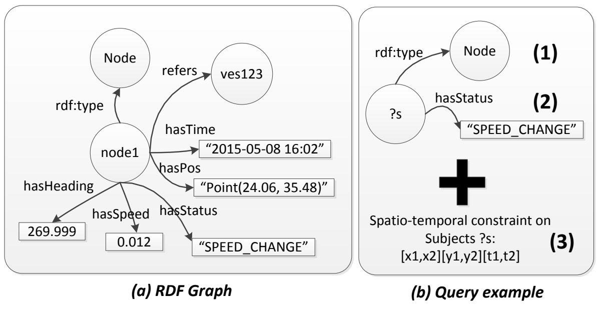


Figure 3: Example of RDF graph and spatio-temporal range query.

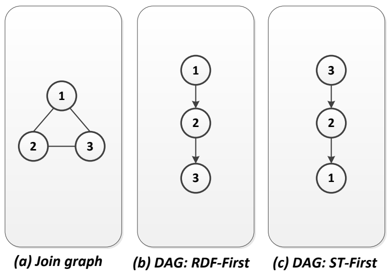


Figure 4: Example of join graph for query of Figure 3, and different DAGs corresponding to execution plans.

Layer the choice of a physical implementation, depends on a static set of rules (rule-based optimization).

The Spark SQL API, implements two main physical join operators: Broadcast Hash Join and Sort-merge Join. These algorithms are described in the following, assuming that *datasetA* and *datasetB* are joined together, when the size of *datasetB* is estimated[5] to be smaller than the size of *datasetA*:

- **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*[6] 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]An estimator is built in Spark SQL Catalyst optimizer.

4.2 Spatio-temporal RDF Processing

Consider the case of a spatio-temporal query *StW*, which is defined by a non spatio-temporal SPARQL query Q, and a spatio-temporal constraint q. Moreover, for ease of presentation, let us restrict the SPARQL query Q to consist of a set of triple patterns, i.e., $Q = \{tp_1, tp_2, \ldots, tp_n\}$. In abstract terms, processing an *StW* 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 entities 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 3.1, *we can transform the spatio-temporal part of the query q into a range filter for encoded Subjects in triples*, i.e., similar to having an additional triple pattern tp_{n+1} in Q. This approach has the advantage that it essentially replaces the need for spatio-temporal processing (and any specialized index structure that would be required for efficient processing) with a plain additional filtering constraint that needs to be imposed on resulting triples. The caveat is that this approach can produce false positives, i.e., triples that do not actually match the spatio-temporal constraint q. Therefore, the produced result set needs to go through a refinement phase, in order to discard false positives from the final result set.

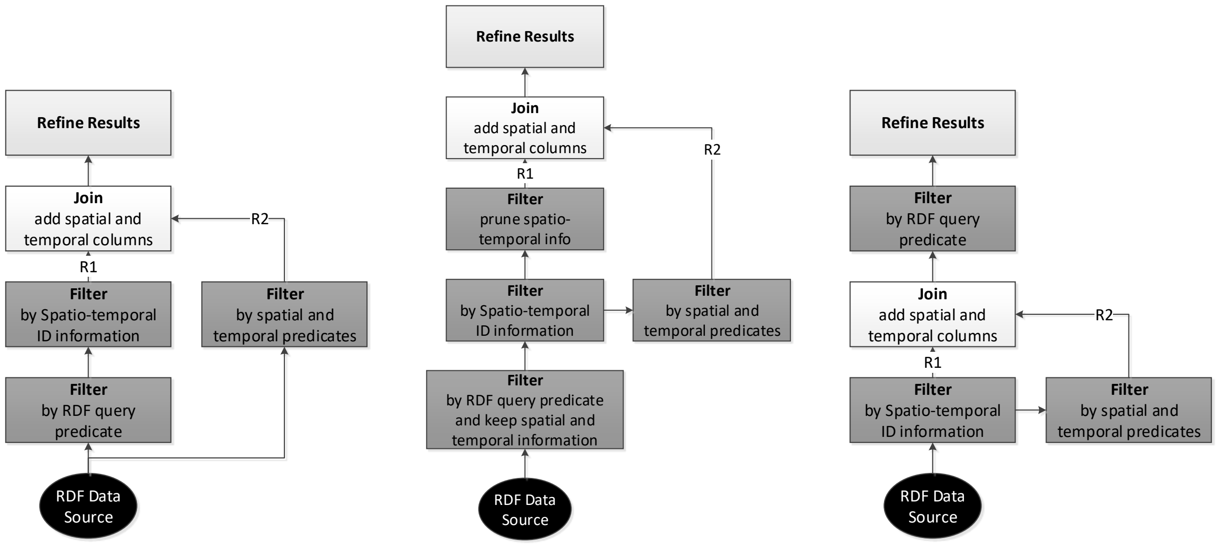
For a given *StW* query, we construct a *logical query plan* that practically determines one potential way to process the query, by specifying the execution order of the query operators. Obviously, several queries may be executed using various query plans, each leading to different performance. For instance, the *StW* query might be executed by either processing the spatio-temporal part of the query first (denoted *ST-First*), or by processing the RDF part of the query first (denoted *RDF-First*). Undoubtedly, other query plans can be produced by changing the order of the triple patterns in the RDF part of the query, but we assume a specific order of execution for the triple patterns to ease exposition and examine only these two options for query plans.

DiStRDF impements both ST-First and RDF-First logical query plans, which are described in detail in the following.

4.3 Logical Query Plans

To exemplify, Figure 3 demonstrates the the case of an *StW* query, where the left part shows (part of) an RDF graph and the right part depicts a query on that graph that consists of two triple patterns and a spatio-temporal constraint. Essentially the RDF graph is composed of a *mobility 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 *mobility nodes* which are of type

[6]A partitioner is a mechanism that determines the location (i.e. node) of each record, on the repartitioning process.


Refine Results
Join
add spatial and temporal columns
R1
R2
Filter
by Spatio-temporal ID information
Filter
by spatial and temporal predicates
Filter
by RDF query predicate
RDF Data Source
Refine Results
Join
add spatial and temporal columns
R1
R2
Filter
prune spatio-temporal info
Filter
by Spatio-temporal ID information
Filter
by spatial and temporal predicates
Filter
by RDF query predicate and keep spatial and temporal information
RDF Data Source
Refine Results
Filter
by RDF query predicate
Join
add spatial and temporal columns
R1
R2
Filter
by Spatio-temporal ID information
Filter
by spatial and temporal predicates
RDF Data Source

(a) RDF-First logical plan **(b) Improved RDF-First logical plan** **(c) ST-First logical plan**

Figure 5: Logical query plans.

"Node" and have status "SPEED_CHANGE", while satisfying a spatio-temporal box constraint.

The two triple patterns are marked with 1 and 2. Also, we mark the spatio-temporal constraint with 3, as if it were an additional triple pattern, since we have explained that our encoding permits us to deal with spatio-temporal constraints as a triple pattern, namely a *Selection* operator on triples.

Figure 4 shows the *join graph* corresponding to the query. Also, notice that the join graph consists of 3 nodes, which correspond to the 3 triple patterns of the query. The edges connect nodes that share a common variable, in this case: ?s. Essentially, the join graph is a representation of triples pattern joins.

This graph can be transformed to a directed acyclic graph (DAG) in different ways, as shown in Figures 4(b) and (c). In this way physical execution plans are produced, which correspond to alternative ways to execute the query. For example, the DAG in Figure 4(b) corresponds to an RDF-First approach, where the RDF part of the query is processed first, pruning unnecessary triples, and followed by the spatio-temporal part of the query. This plan should be selected when the RDF part of the query is very selective, thus eagerly pruning a great number of triples. In contrast, the DAG in Figure 4(c) corresponds to an ST-First approach, where the spatio-temporal filter is processed first, thus restricting the number of RDF triple by pruning those that do not comply with the spatio-temporal constraint, followed by processing the RDF part of the query. Again, this plan is preferred when the spatio-temporal constraint is very selective.

4.3.1 RDF-First Logical Plan. Figure 5a depicts the RDF-First logical plan. This query plan aims to minimize the size of R_1.

First, the RDF data source (set of RDF triples) is filtered by the RDF query predicate, and then by the spatio-temporal information present in the 1D encoding. These filters produce R_1. Notice the benefit of the proposed encoding scheme: in the absence of this encoding, it would not be possible to apply the second filter, thus R_1 would contain data filtered only by the RDF constraint. Thus, we can reduce the size of R_1 at an early stage of processing.

As described earlier, filtering by spatio-temporal ID information, produces false positives, which need to be refined. For this purpose, we need the spatio-temporal exact information of each entity. Therefore, R_2 is produced by applying the spatial and temporal predicates on the data source. R_2 contains all the spatio-temporal information contained in the data source.

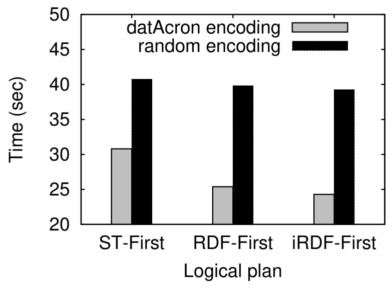
Then, a join operator is used between R_1 and R_2, in order to add the encoded spatio-temporal information to the intermediate result set.

After the join, the refinement phase takes place, which is the same for all logical query plans: spatio-temporal information is decoded in actual spatial and temporal value, and false positives are eliminated to produce the result set. Practically, only the records that satisfy the spatio-temporal query range predicate are kept. Finally, a projection of the final result set is applied to select only the columns that the user has requested. Obviously, the result needs to be decoded prior to being reported to the user.

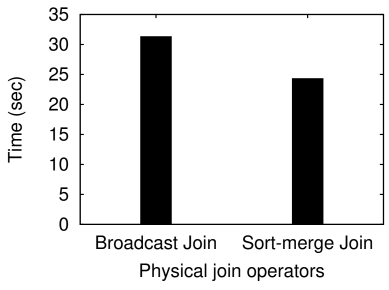
4.3.2 Improved RDF-First Logical Plan. Figure 5b depicts the improved RDF-First logical plan. This plan aims to minimize both R_1 and R_2 at the cost of an extra filtering step. This is achieved by changing the first operator in the plan, which besides filtering based on the RDF constraint, it also keeps the spatial and temporal information. In this way, the filter operator that produces R_2 is able to avoid accessing the data source, rather it is produced from the in-memory result (RDD) produced by the previous filter operator.

4.3.3 ST-First Logical Plan. Figure 5c depicts the ST-First logical plan. In this plan, the spatio-temporal filtering is applied first. This query plan aims to minimize the size of R_2.

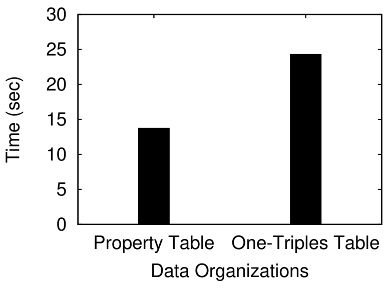
As already mentioned, the plan performs a spatio-temporal filter first, based on the information encoded in spatio-temporal ID. R_1 is produced directly by this filter, while R_2 is produced by applying a filter to R_1 to keep only the spatio-temporal information.



(a) 1D encoding vs. random encoding



(b) Broadcast Hash Join vs. Sort-merge



(c) Property Table vs. One-Triples Table

Figure 6: Performance of *DiStRDF* system when using various execution plans and join operators.

Then, R_1 and R_2 are joined together to enrich the intermediate result set with spatio-temporal information and the rest of the refinement process is identical to the aforementioned RDF-First logical query plan.

5 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 64GB RAM and a 6-core 1.7GHz processor. All nodes are running Ubuntu 16.04.

5.1 Experimental Setup

Parameter	Values
Encoding scheme	**1D encoding**, random encoding
Logical plans	**Improved RDF-First**, RDF-First, ST-First
Physical plans	**Sort-merge Join**, Broadcast Hash Join
Data organizations	**One-triples table**, Property table

Table 1: Experimental setup parameters (default values in bold).

Data sets. We used surveillance and static information from the maritime domain, collected in January 2016, covering the Mediterranean Sea and part of the Atlantic Ocean. We used the RDF ontology described in [16]. The size of the data set is 269,357,225 triples, which translates to approximately 6GB in text format. These triples were encoded to integer values using the method described in Section 3.1 to form the encoded triples data set. Data is stored in HDFS using Parquet file format, to enable efficient access for Spark applications and benefit by columnar storage, compression and predicate pushdown. The Parquet input data set is partitioned to 10 HDFS blocks which results to roughly 3GB size of compressed data.

A dictionary containing the mapping between encoded and decoded values was also created and stored in a Redis cluster instance, running on all 10 nodes of the cluster, with no replication enabled.

Configuration. We configured Spark on YARN, where each executor is set to use one virtual core and 2GB of RAM. One node was set to be the driver node, while the others contain the Spark Executors. All experiments conducted, use 10 Spark Executors, with 5 executor cores each. We also used the Jedis [7] library, to communicate with the Redis cluster instances.

Type of query. We focused our experiments on star spatio-temporal queries, i.e., StW queries having a star RDF predicate on a fixed spatio-temporal constraint. All of our experiments were conducted using the same query parameters, producing a result set of 21 triples.

Algorithms. We have implemented the aforementioned logical and physical plans as described in the previous sections for star spatio-temporal queries. More specifically, we experimented with (a) RDF-First, ST-First, Improved RDF-First logical plans, (b) Broadcast Hash Join, Sort-merge Join physical plans and (c) one-triples table, property tables data organizations. Table 1 summarizes the algorithms used during the experimental evaluation process.

Metrics. Our main evaluation metric was the total execution time of each experiment on the Spark cluster. The actual execution time of our algorithms is presented here, omitting any overhead caused by Spark initialization procedures. Each experiment was run 3 times, and the average execution time is depicted in the charts.

Methodology. First, we study the benefits of the 1D encoding scheme, by conducting experiments using our 1D encoding against random encoding, which is typically used by RDF engines. Then, we evaluate each of the logical plans, using a fixed physical plan for the join operator (Sort Merge Join). In the following, we experiment with the two join physical plans, using only the Improved RDF-First logical plan. Finally, we examine the feature of storing RDF data in property tables, against the one-triples table. Our experimental setup is summarized in Table 1, having in bold the default value of each parameter, unless specified otherwise.

5.2 Results

Benefit of 1D Encoding. Figure 6a depicts the execution time comparing our 1D encoding against a random encoding, for all three logical plans. Clearly, by using the spatio-temporal 1D encoding we are able to prune early a set of triples which do not satisfy the query spatio-temporal constraint. This improves performance by at least 10 seconds. It is expected that this gain will increase for larger data sets. This demonstrates the advantage offered by our deliberate encoding scheme. It is also important to note that this early pruning also exploits the predicate pushdown

[7] https://github.com/xetorthio/jedis

feature of Parquet, resulting in smaller I/O cost, since fewer data will be accessed from HDFS.

Comparing the Performance of Logical Plans. Figure 6a demonstrates the performance comparison of logical plans, when using the 1D encoding scheme. ST-First logical plan performs worst, due to the increased size of input to the join operator. Evidently, the RDF predicate is able to prune many triples, resulting to better performance for the RDF-First alternatives. RDF-First achieves to reduce the input size to the join operator, performing better than ST-RDF. However, the Improved RDF-First algorithm combines the benefits of both RDF-First and ST-First, providing to the join operator the smallest input size. These benefits correspond to the execution time needed by Improved RDF-First, which performs better than all other alternatives.

Comparing the Performance of Physical Join Operators. Figure 6b demonstrates the impact on performance by the selection of a physical join operator. The Sort-merge join operator performs better than the Broadcast join operator due to the large size of input data. Notice that we used 5 CPU cores per executor, which results to 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 equal to the number of executors (10) during the above experiments.

Figure 6c shows that using the property tables data organization results to much better performance, due to not needing a join operation to evaluate the query results.

6 CONCLUSIONS

In this paper, we present the first parallel and scalable in-memory solution to the problem of spatio-temporal RDF 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 sizes of data. Notable features of the proposed solution include the support for various query execution plans, as well as different storage file types and data organizations. Our experiments demonstrate the performance of our system, which is able to efficiently process simple RDF spatio-temporal queries, in a few seconds.

In the future, we plan to extend our system, to cover a larger part of the SPARQL specification. Furthermore, we plan to improve the performance of *DiStRDF Processing Layer* by implementing more sophisticated execution plans, based on statistics of the data.

ACKNOWLEDGMENTS

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.

REFERENCES

[1] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. *PVLDB* 10, 13 (2017), 2049–2060.

[2] Louai Alarabi, Mohamed F. Mokbel, and Mashaal Musleh. 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.* 84–104.

[3] Konstantina Bereta, Panayiotis Smeros, and Manolis Koubarakis. 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.* 259–274.

[4] Olivier Curé and Guillaume Blin. 2014. *RDF Database Systems: Triples Storage and SPARQL Query Processing.* Elsevier.

[5] Ahmed Eldawy and Mohamed F. Mokbel. 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.* 1352–1363.

[6] George Garbis, Kostis Kyzirakos, and Manolis Koubarakis. 2013. Geographica: A benchmark for geospatial rdf stores (long version). In *International Semantic Web Conference.* Springer, 343–359.

[7] Stefan Hagedorn and Timo Räth. 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.* 570–573.

[8] Mohammad Farhan Husain, Pankil Doshi, Latifur Khan, and Bhavani M Thuraisingham. 2009. Storage and Retrieval of Large RDF Graph Using Hadoop and MapReduce. *CloudCom* 9 (2009), 680–686.

[9] Zoi Kaoudi and Ioana Manolescu. 2015. RDF in the clouds: a survey. *VLDB J.* 24, 1 (2015), 67–91.

[10] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. 2011. From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra. *PVLDB* 4, 12 (2011), 1426–1429.

[11] Manolis Koubarakis and Kostis Kyzirakos. 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.* 425–439.

[12] John Liagouris, Nikos Mamoulis, Panagiotis Bouros, and Manolis Terrovitis. 2014. An Effective Encoding Scheme for Spatial RDF Data. *PVLDB* 7, 12 (2014), 1271–1282.

[13] Hubert Naacke, Bernd Amann, and Olivier Curé. 2017. SPARQL Graph Pattern Processing with Apache Spark. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017.* 1:1–1:7.

[14] Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. 2011. An intermediate algebra for optimizing RDF graph pattern matching on MapReduce. In *Extended Semantic Web Conference.* Springer, 46–61.

[15] Kurt Rohloff and Richard E. Schantz. 2011. Clause-iteration with MapReduce to scalably query datagraphs in the SHARD graph-store. In *DIDC'11, Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing, San Jose, CA, USA, June 8, 2011.* 35–44.

[16] Georgios M. Santipantakis, George A. Vouros, Apostolos Glenis, Christos Doulkeridis, and Akrivi Vlachou. 2017. The datAcron Ontology for Semantic Trajectories. In *The Semantic Web: ESWC 2017 Satellite Events - ESWC 2017 Satellite Events, Portorož, Slovenia, May 28 - June 1, 2017, Revised Selected Papers.* 26–30.

[17] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thorsten Berberich, and Georg Lausen. 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.* 155–168.

[18] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thomas Hornung, and Georg Lausen. 2013. PigSPARQL: A SPARQL Query Processing Baseline for Big Data. In *Proceedings of the ISWC 2013 Posters & Demonstrations Track, Sydney, Australia, October 23, 2013.* 241–244.

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

[20] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. 2015. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. *PVLDB* 8, 13 (2015), 2110–2121.

[21] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. 2016. LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data. *PVLDB* 9, 13 (2016), 1565–1568.

[22] Akrivi Vlachou, Christos Doulkeridis, Apostolos Glenis, Georgios M. Santipantakis, and George A. Vouros. [n. d.]. Efficient Spatio-temporal RDF Query Processing in Large Dynamic Knowledge Bases. *Submitted for publication* ([n. d.]).

[23] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 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.* 1071–1085.

[24] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 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.* 70:1–70:4.

[25] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 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).* 2–2.