Distributed storage of large knowledge graphs with mobility data

Panagiotis Nikitopoulos, Nikolaos Koutroumanis, Akrivi Vlachou, Christos Doulkeridis and George A. Vouros

This chapter presents novel solutions for storage and querying of large knowledge graphs, represented in RDF, which consist of mobility data. Such knowledge graphs are generated and updated daily based on incoming positional information of moving entities, possibly linked with contextual information and weather data. Efficient management of knowledge graphs is critical, as it enables advanced data analysis of integrated data and the discovery of hidden patterns of movement. Despite that the scalable management and querying of large RDF graphs has been extensively studied recently, most of these works do not deal with spatial or spatio-temporal query processing. One significant factor that differentiates the knowledge graphs of mobility data is their spatio-temporal dimension which requires special treatment.

Querying knowledge graphs of mobility data typically entail both spatial and temporal constraints on some entities. Existing RDF engines would have to evaluate the RDF part of the query first, and then process the spatio-temporal constraint (in a post-processing step). However, this imposes a significant overhead, since large portions of data could have been filtered by the spatio-temporal constraint, and results in inferior performance of query processing. Motivated by these shortcomings, in this chapter, the design and implementation of a parallel/distributed RDF engine is presented, which targets to store and process efficiently spatio-temporal RDF data. The main constituent parts of the engine include:

- the one-dimensional encoding scheme for mapping spatio-temporal data to one-dimensional values,
- the storage layer of the engine that stores RDF data, but exploits the proposed encoding for efficient indexing of spatio-temporal entities, and
- the processing layer of the engine that supports queries with spatio-temporal and RDF constraints.

Panagiotis Nikitopoulos, Nikolaos Koutroumanis, Akrivi Vlachou, Christos Doulkeridis and George A. Vouros
University of Piraeus, Karaoli & Dimitriou St. 80

e-mail: {nikp,koutroumanis,avlachou,cdoulk,georgev}@unipi.gr

Encoding scheme

RDF engines typically store resources and literals in the form of integers (rather than strings), for efficient indexing and management. This is achieved by means of an integer-encoding technique that provides a one-to-one mapping from strings to integers (and vice-versa). As a result, the produced integer values do not carry any semantic information, instead they just allow the retrieval of the string representing the respective RDF resource or literal.

The intuition behind our proposed encoding scheme is to produce integer values for spatio-temporal entities in such a way that the integer value provides some information about the spatial and temporal information of the entity. In turn, this allows *joint*]filtering of such entities based on spatial and temporal constraints at the same time as the RDF filtering is performed.

Thus, prior to the storage of data, the encoding step takes place for each entity. The adopted encoding scheme is applied on the spatio-temporal information of the moving entities. The discrete spatial and temporal information is mapped into an integer value which represents a part of the 3D space with specific spatial and temporal bounds. The spatial location can be in 2D (for cars, vessels, etc.) or 3D (for aircraft). Only the 2D case is presented in the following for simplicity. The mapping of the spatial location is based on the grid partitioning method, where space is split into $2^m = (2^{m/2} \cdot 2^{m/2})$ equi-sized cells. Also, a space-filling curve is exploited, providing an ordering for the spatial cells of the grid. In Figure 1, Hilbert and Z-order curves are depicted, with an integer value (ID) assigned to each cell. The integer value is determined by the ordering defined by the employed space-filling curve, in order to achieve *spatial locality*, i.e., entities that are located nearby in the 2D space are assigned similar integer values.

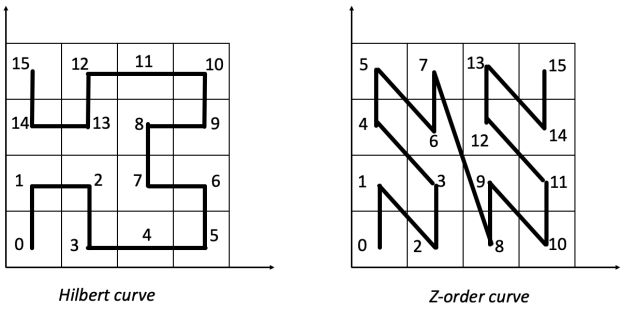

Fig. 1: Space-filling curves (Hilbert and Z-order) used for ordering the spatial cells.

In order to add the temporal dimension, the temporal domain is partitioned into temporal intervals $\mathscr{T} = \{T_0, T_1, \ldots\}$ where T_i represents a time span. Temporal par-

titions are disjoint and can vary, covering the entire time domain ($\bigcup T_i = \mathscr{T}$). Each temporal partition T_i is associated with a spatial grid, as depicted in the figure. The only restriction is that the identical grid structure (i.e., 2^m equi-sized cells) is used for all temporal partitions T_i.

Both spatial and temporal partitions are combined together, resulting to a unique identifier for any spatio-temporal position of a moving entity. The representation of the identifier is depicted in Figure 2, where it consists of b bits. The k least significant bits are used as unique identifier of any spatio-temporal entity in a specific spatio-temporal cell. The ID of the cell which represents its spatial part, is recorded in the next m bits. The most-significant bit is reserved so as to discern between spatio-temporal RDF entities and other RDF entities. Specifically, the most-significant bit is set to 0 for all IDs of spatio-temporal RDF entities, while for IDs of all other RDF entities is set to 1. The remaining $b-(m+k+1)$ bits are used for the temporal representation, thus $2^{b-(m+k+1)}$ temporal partitions can be stored in total.

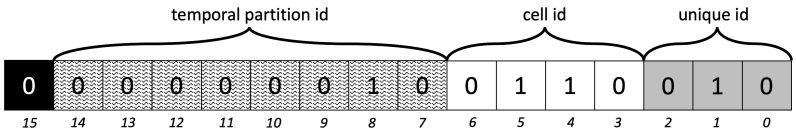

Fig. 2: IDs encoding using bits: b total bits, m bits for spatial part (cell id), k bits for uniqueness, $b-(m+k+1)$ bits for the temporal partition.

Given the ID of a spatio-temporal entity, the 3D spatio-temporal cell where the entity belongs to can be identified. The same applies reversely; given a 3D cell, a range of IDs that correspond to any entity belonging to the cell can be computed. The encoding scheme guarantees data locality, meaning that entities with similar spatio-temporal representations are assigned nearby IDs (that belong to small ranges of values).

DiStRDF Storage Layer

Concerning the storage layer, the proposed solution was designed for operating on distributed environment. This scalable storage solution supports replication, in-memory lookups, indexing, compression and efficient query execution. Even in the case of hardware failures on some nodes of the cluster, the availability of the stored data will be unaffected. The storage layer consists of two data stores: (a) a dictionary which stores the mappings between RDF triples and their encoded values, and (b) the RDF data store which contains the encoded RDF triples stored in HDFS.

For the DiStRDF dictionary, it is essential to support efficient lookups and horizontal scalability. Thus, the attention is given to distributed key-value NoSQL stores which are designed for storing, retrieving and managing dictionaries or hash tables.

A key-value NoSQL store is a distributed data structure that contains sets of key-value pairs. The store is responsible for storing those pairs and retrieving the corresponding value when a key is provided. DiStRDF leverages Redis (https://redis.io/) for storing the dictionary, but any other key-value store can be used.

The design of the distributed RDF data storage covers several aspects, such as compression, file layout, data organization, indexing and partitioning. The DiStRDF store organizes the RDF triples in encoded form. RDF triples are transformed to encoded integer values and are stored in an HDFS cluster. Each node of the cluster, maintains the spatio-temporal information in *Property tables* as shown in Table 1. 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. Along with the *Property tables* that are stored across the nodes, another table is used for storing RDF data (*leftover triples*) that is not associated with spatio-temporal information. This table is used as a regular triples data store, which stores subject, predicate and object in three different columns. Table 2 depicts an example of a leftover triples table, where non spatio-temporal information is stored, such as event occurrence and static information about vessels.

Both tables are stored in Parquet formatted files, which is a column-based data layout with native support for several compression codecs (lzo, gzip, snappy). Parquet formatted data sets are usually split into several files where every file contains a set of metadata such as ranges of values for every column of the file. This metadata can significantly boost query performance, e.g., in case a query retrieves few columns of a property table. Predicate and projection pushdown are also features supported by Parquet, which avoids the cost of reading all data from disk at query time.

The partitioning mechanism used in DiStRDF storage layer partitions the data according their spatio-temporal similarity. More specifically, the 1D encoded integers computed for the spatio-temporal data are used for their distribution across the available nodes based on range partitioning.

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 17:20:00
node58	ves376609000	3	23.2E 35.7N	2017-02-05 10:42:08

Table 1: RDF property table example.

DiStRDF Processing Layer

The DiStRDF processing layer interacts with the storage layer, so as to efficiently retrieve the query-relevant data. The processing layer is a SPARQL query engine that processes batch queries over huge amounts of spatio-temporal RDF data. It

Subject	Predicate	Object
turnInit	occurs	node22
stoppedInit	occurs	node15
ves376609000	hasBuildDate	2009-05-31

Table 2: RDF leftover triples example.

is implemented on Apache Spark, a popular engine for parallel in-memory data processing based on the MapReduce model. The processing layer is comprised of the following components: (a) the SPARQL Query Parsing, (b) the Logical Plan Builder, (c) the Logical Plan Optimizer and (d) the Physical Plan Constructor.

The SPARQL Query Parsing component is in charge of performing the parsing task of the query. It checks the correctness its syntax and transforms it into an internal representation, used by the other modules of the processing engine. Assuming that the syntax is correct, the SPARQL query is translated into a set of basic graph patterns (BGP). These graph patterns are given to the logical planner so as to construct a logical query plan. The query parsing component is built using the functionality of the Apache Jena software so as to obtain the BGPs from the SPARQL query.

The Logical Plan Builder constructs a logical plan of the query, after having been parsed by SPARQL Query Parsing component. The logical query plan consists of logical operators ordered in hierarchical form (tree). Specifically, the logical plan represents a way to execute the respective SPARQL query.

The Logical Plan Optimizer, performs optimizations in the logical plan of a query. Optimizations are classified as *rule-based* or *cost-based*. In *rule-based optimization*, a set of rules is applied on join operators in order to be exploited during physical for performing the query in a more efficient way. In *cost-based optimization*, the formation and the ordering of join operators are determined based on statistics (histograms).

The Physical Plan Constructor component takes as input the optimized logical plan and transforms it to a physical query plan. A physical query plan is comprised of a set of physical operators, which represent an algorithm for every stage (operation) of the query execution. Hence, the physical plan constructor aims to choose the best performing algorithm for every stage of the query execution. All operators are designed to operate by utilizing the features of Apache SparkSQL API. They process a distributed set of data (i.e., a Spark DataFrame) by performing parallel operations on it. Therefore, the workload of any DiStRDF physical operator is in fact distributed among the available computing nodes to be computed in parallel.