Efficient Spatio-temporal RDF Query Processing in Large Dynamic Knowledge Bases

Akrivi Vlachou
Department of Digital Systems
University of Piraeus
18534 Piraeus, Greece
avlachou@aueb.gr

Christos Doulkeridis
Department of Digital Systems
University of Piraeus
18534 Piraeus, Greece
cdoulk@unipi.gr

Apostolos Glenis
Department of Digital Systems
University of Piraeus
18534 Piraeus, Greece
apostglen46@gmail.com

Georgios M. Santipantakis
Department of Digital Systems
University of Piraeus
18534 Piraeus, Greece
gsant@unipi.gr

George A. Vouros
Department of Digital Systems
University of Piraeus
18534 Piraeus, Greece
georgev@unipi.gr

ABSTRACT

An ever-increasing number of real-life applications produce spatio-temporal data that record the position of moving objects (persons, cars, vessels, aircrafts, etc.). In order to provide integrated views with other relevant data sources (e.g., weather, vessel databases, etc.), this data is represented in RDF and stored in knowledge bases with the following notable features: (a) the data is dynamic, since new spatio-temporal data objects are recorded every second, and (b) the size of the data is vast and can easily lead to scalability issues. As a result, this raises the need for efficient management of large-scale, dynamic, spatio-temporal RDF data. In this paper, we propose boosting the performance of spatio-temporal RDF queries by compressing the spatio-temporal information of each RDF entity into a unique integer value. We exploit this encoding in a filter-and-refine framework for processing of spatio-temporal RDF data efficiently. By means of an extensive evaluation on real-life data sets, we demonstrate the merits of our framework.

CCS CONCEPTS

• **Information systems → Database query processing**;

KEYWORDS

Spatio-temporal RDF, query processing, encoding

ACM Reference Format:
Akrivi Vlachou, Christos Doulkeridis, Apostolos Glenis, Georgios M. Santipantakis, and George A. Vouros. 2019. Efficient Spatio-temporal RDF Query Processing in Large Dynamic Knowledge Bases. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3297280.3299732

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3299732

1 INTRODUCTION

A vast number of applications generate, collect, process, and analyze spatio-temporal data, thus imposing challenges for spatio-temporal data management. Typical applications include moving object databases, location-based systems, surveillance systems, but also social networks. Integrating spatio-temporal data from different sources, including linked open data, raises the need for an RDF-based representation. As an application example, consider a surveillance system that records the positions of moving entities (e.g., vessels or aircrafts), and transforms this information in RDF in order to enable linking to other external sources [2]. As moving entities are monitored, the vast majority of RDF triples represent the positions of entities in space and time, and SPARQL queries with spatio-temporal constraints on such triples is the most processing-intensive query type. A concrete example of a query from the maritime domain is: *find all fishing vessels in the vicinity of Cape Sounion in the last two hours*, where *Cape Sounion* is the spatial constraint, *last two hours* is the temporal constraint, and *fishing vessel* is the RDF predicate that implies that retrieved entities need to be of specific type. Thus, efficient processing of spatio-temporal SPARQL queries, such as the one described above, is of utmost importance. In such applications, data is dynamic in the sense that new positions of moving entities are recorded over time.

Unfortunately, existing RDF stores are not designed to manage spatio-temporal information efficiently (some exceptions [11, 16] are reviewed in the related work section). A common approach for storing and indexing RDF data used by many systems (e.g., RDF-3X [14], Hexastore [19], etc.) is to store all subject, property, object (SPO) statements in a single triples table. Typically, URIs (strings) are encoded as unique identifiers IDs (integer values) in a structure called *dictionary*, which provides a unique, bi-directional mapping between URI and ID. Then, the triples table contains integers, rather than strings (URIs), as storage and indexing of fixed-length records, and in particular integer values, is more efficient. For efficient access, clustered B^+-trees on all six (S, P, O) permutations of integer-encoded triples are built [14, 19].

Even though spatio-temporal data management is a well-studied topic, it is not trivial to find the best way to combine it with

RDF data processing. The straightforward solution of using off-the-shelf spatio-temporal indexes next to the necessary B^+-tree indexes for RDF processing has significant shortcomings. First, it induces increased processing cost due to separate processing of spatio-temporal and RDF filters, thus preventing any opportunity of pruning the search space jointly using both filtering criteria. Second, it increases the maintenance overhead of the system, since more indexes need to be kept up-to-date.

Motivated by such shortcomings of existing systems, in this paper, we propose a method for boosting the performance of spatio-temporal RDF queries. The gist of our approach is to produce a unique identifier for the positions of a spatio-temporal RDF entity in a deliberate way, so that the ID provides an approximation of the entity's spatio-temporal information instead of using a random number. The advantages of encoding spatial RDF data into integer values has been demonstrated in [12], showing that spatial RDF queries can be executed efficiently without any overhead to the original query engine. Nevertheless, the methods proposed in [12] are suitable only for static data; thus, several additional technical challenges need to be addressed: (a) handle the dynamic nature of the data, since new positions of moving objects arrive every second and need to be stored, (b) address the spatio-temporal skew in the data, which is typical for real-life applications, (c) keep the encoded identifiers as compact as possible in the 1D space of identifiers, since this affects the scalability of the approach.

In essence, our technique enables filtering of RDF triples based on the values of the identifiers, thus finding triples that match the RDF constraints and the spatio-temporal constraints at the same time. In consequence, there is no need for extra indexes; spatio-temporal filtering is performed using the existing B^+-tree indexes. In order to evaluate the proposed encoding approach we propose a generic framework for spatio-temporal RDF processing that relies on the proposed encoding technique. Therefore, our approach has the following salient features: (i) it can be readily integrated to any existing RDF engine that operates on encoded triples, (ii) it does not impose any overhead in the performance of non spatio-temporal RDF queries, while it boosts the performance of spatio-temporal RDF queries, (iii) it works on dynamic RDF data, thus does not require static data nor any a priori knowledge of the data distribution, (iv) it is designed to handle large-sized RDF data sets.

In a nutshell, we make the following contributions:

- We study the problem of spatio-temporal querying of large and dynamic RDF data, which is ubiquitous in real-life applications that manage data of moving entities.
- We propose an encoding scheme for dynamic spatio-temporal data, which offers fast, approximate filtering of RDF data. We address the challenges raised by the dynamic nature of data, in particular with respect to the generation of compact encodings in the 1D space.
- We introduce a query processing framework for spatio-temporal RDF data that adheres to the filter-and-refine methodology, and introduce efficient algorithms that exploit the encoding to improve the effectiveness of filtering.
- We evaluate our proposal experimentally using real, large-scale data sets, which record the movement of vessels as well as other contextual data.

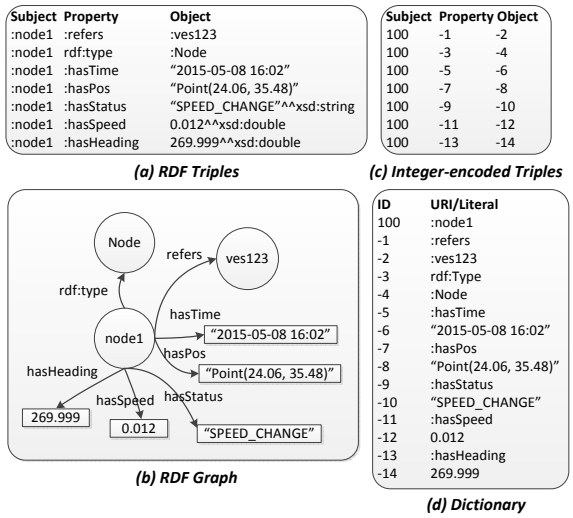


Figure 1: RDF triples, RDF graph, and encoded triples.

The remainder of this paper is structured as follows: Section 2 describes background knowledge and the problem statement. Section 3 describes the computation of the spatio-temporal identifier. Section 4 describes the proposed encoding techniques that handle spatio-temporal skew in the data. Then, in Section 5, we describe the query processing framework. The experimental evaluation is described in Section 6, while Section 7 reviews the related work. Finally, Section 8 concludes the paper.

2 PROBLEM STATEMENT

2.1 Preliminaries

Consider an RDF knowledge base $\mathcal{B}$ that stores data in the form of triples (*subject*, *property*, *object*) or $\langle s, p, o \rangle$, also known as *statements*, thus $\mathcal{B} = \{(s_i, p_i, o_i)\}$ with $i = 1, \ldots, |\mathcal{B}|$. Subjects, properties and objects are *resources* identified by a Uniform Resource Identifier (URI). The object can also be identified by a *literal* for the representation of simple values, such as strings, dates, numbers, etc. The property (or *predicate*) captures the relationship of the subject to an object. Another way to represent $\mathcal{B}$ is as a graph having as vertices the subjects and objects, and as edges the respective properties.

Example 1. *Consider a set of RDF triples and the corresponding RDF graph depicted in Figures 1(a) and (b) respectively, which represent one spatio-temporal position (x=24.06, y=35.48, t=2015-05-08 16:02) of a vessel (ves123) as well as its speed, heading and status. Figure 1(c) shows the equivalent ID-encoded triples, given the dictionary depicted in Figure 1(d), which maps unique IDs to URIs/literals and vice-versa.*

Processing a SPARQL query is essentially equivalent to a multi-way self-join query on the triples table. Since the result set consists of integer identifiers, the corresponding URI is retrieved using the dictionary, before returned to the user.

Symbols	Description
$\mathcal{B}^{st}$	Spatio-temporal RDF knowledge base
(s_i, p_i, o_i)	A triple in knowledge base $\mathcal{B}^{st}$
τ	A spatio-temporal data point $(\tau.x, \tau.y, \tau.t)$
$s(\tau)$ or $\tau(s)$	Association between point τ and entity s
$s(\tau).id$	The identifier of entity s
b	Total number of bits used for mapping
m	Number of bits used for spatial encoding
k	Number of bits used for unique ID

Table 1: Overview of main symbols.

2.2 Problem Formulation

We focus on a specific category of RDF knowledge bases $\mathcal{B}^{st}$ that contain information about spatio-temporal entities. As such, some vertices of the RDF graph (subjects or objects) are associated with spatio-temporal information. For simplicity, we assume in the following that only subjects in RDF triples are associated with spatio-temporal information, but this is not restrictive in any way. Moreover, we target large-scale RDF data sets, and we focus on scenarios where data is dynamic and arrives ordered by time, as in the case of a surveillance system for moving entities described in Section 1.

Given the 3-dimensional space defined by geographical coordinates and time, we consider data points τ located at a particular location $\tau.x, \tau.y$ at a given time $\tau.t$. A spatio-temporal RDF entity (subject s) is associated with a point τ in space-time, and we denote this by $s(\tau)$ or $\tau(s)$. In Figure 1, *node1* is $s(\tau)$, and $\tau.x$=24.06, $\tau.y$=35.48, $\tau.t$=2015-05-08 16:02.

In this paper, we turn our attention to a specific class of SPARQL queries over $\mathcal{B}^{st}$, namely those that retrieve spatio-temporal entities that satisfy both a spatio-temporal constraint and an RDF graph pattern. This class of queries is quite important, since they are most frequent in a typical query workload over $\mathcal{B}^{st}$, and the vast majority of resources are spatio-temporal. For a quick reference to the symbols used in this paper, we refer to Table 1.

Definition 1. ***Spatio-Temporal Window (StW) Queries**: Given a spatio-temporal RDF knowledge base $\mathcal{B}^{st}$, a non spatio-temporal SPARQL query Q, and a spatio-temporal constraint q, retrieve the set StW of spatio-temporal RDF resources $s(\tau) \in StW \subseteq \mathcal{B}^{st}$, such that: (a) s satisfies Q, and (b) $q.x_{min} \leq \tau.x \leq q.x_{max} \wedge q.y_{min} \leq \tau.y \leq q.y_{max} \wedge q.t_{min} \leq \tau.t \leq q.t_{max}$.*

Notice that other, more complex, spatio-temporal constraints (e.g., circle, polygon, etc.) can also be supported by transforming them into the corresponding minimum bounding rectangle. Then, an extra step is needed to refine and return those results that truly satisfy the given spatio-temporal constraint.

2.3 Anatomy of the Identifier

The gist of our approach is the following. Consider a space partitioning that divides the spatio-temporal domain into partitions, in such a way that any spatio-temporal point belongs to a single partition based on its coordinates. An (integer) identifier is assigned to each spatio-temporal entity $s(\tau)$, such that: (a) the identifier serves as a *spatio-temporal approximation* for the exact coordinates of τ, and (b) the identifier is *unique* in order to be used for encoding the entity. Non-spatio-temporal entities are encoded as well, with the only requirement that their identifiers take values from a different range of integer values. For example, by convention, in Figure 1(d), spatio-temporal entities use positive integer values, whereas other entities take negative values.

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. [1] 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.

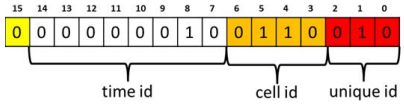


Figure 2: IDs encoding using bits (b: total bits, m: bits for spatial part, k: bits for uniqueness, $b-(m+k+1)$: bits for temporal part.

To encode the spatio-temporal information into an integer (ID), we consider its binary representation consisting of b bits (Figure 2). 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. 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 2. *In Figure 2, 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.*

Obviously, given such an identifier, we can determine the enclosing spatial cell and temporal partition. In this way, we inject useful spatio-temporal information in the identifier of an entity, without losing any of the benefits of having a dictionary and triples encoded as integers. Also, given a spatio-temporal partition (cell), a range of IDs can be computed that correspond to any entity belonging to the respective 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,

[1] As a starting point, 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.

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 that the spatio-temporal locality is reflected in the 1D domain of integers.

3 COMPUTATION OF THE IDENTIFIER

Recall that spatio-temporal RDF entities arrive streamed based on time, i.e., the spatio-temporal entities are encoded in sorted temporal order. In order to compute the next available unique identifier for a new spatio-temporal RDF entity, we maintain (in memory) 2^m counters $C[i]$ ($0 \leq i < 2^m$), one for each spatial cell, that count the number of objects assigned to each cell. Initially, all counters are set to zero. Then, the cell of the 2D spatial grid in which spatio-temporal RDF entity falls in is determined. To derive 1D values for the spatial grid cells, a space-filling curve is used, so as to preserve locality. We employ the Hilbert curve for acquiring a 1D ordering of cells of the spatial grid, which has been shown to have nice clustering properties [13]. Assuming that the spatio-temporal entity belongs to the first time interval, then the integer values of the time interval, the spatial cell, and the corresponding counter are combined to create the unique identifier. Finally, the counter of the cell is increased by one. In case any counter reaches the value of 2^k, this indicates that the cell has no more available IDs to assign. This is the case of *overflow*, and will be addressed subsequently in the paper (Section 4). Thus, identifiers are computed by a function $nextID(h, t, C)$ takes as input the Hilbert value h of the cell, time partition t, and the counters C, and returns the spatio-temporal identifier.

Related to the time partitions, consider the simple case of having a *static*, pre-defined, equi-sized temporal partitioning of the time domain $\mathcal{T}$ (similar to the spatial grid). In other words, any pair of temporal partitions T_i, T_j ($i \neq j$) are of the same duration, i.e., it holds that $|T_i| = |T_j|$. Each time an entity $s(\tau)$ needs to be encoded, we check if the time $\tau.t$ of the point τ belongs to the current time partition or the next one. In the latter case, all cell counters maintained need to be reset to zero. We call this approach *static temporal partitioning*.

This plain approach is not applicable for dynamic data since we cannot guarantee that a unique identifier is available, since (in worst case) more than 2^k points may be located in a spatial cell for a single temporal partition. Thus, this approach cannot handle temporal skew in the data. Setting the duration of time partitions is a difficult task, since the data arrival rate may vary. Even if the above problem is set aside, in case of skewed spatial distribution, we may end up with spatial grids having too many empty cells. This means that we have retained identifiers that are never used, thus reducing the overall storage utilization dramatically.

4 DYNAMIC TEMPORAL PARTITIONING

The main properties that need to by ensured by the novel proposed encoding scheme are: (a) handle the dynamic nature of the data, since new positions of moving objects arrive every second and need to be stored, (b) address the spatio-temporal skew in the data, which is typical for real-life applications, (c) keep the encoded identifiers as compact as possible in the 1D space of identifiers, since this affects the scalability of the approach. Our work focuses on applications with high volume of dynamic data and therefore it is important to ensure that the system can store as many entities as possible. Notice that in each time interval the unique integer will be larger than the integer of the previous interval, while integers related to empty spatial cells (for a given time interval) will be unused. Many spatial cells may contain only few or no spatio-temporal entities for a given time partition, due to spatio-temporal skewness in the data distribution, thus leading to wasted (unused) unique identifiers. The encoding scheme should accommodate as many entities as possible, i.e., it should minimize the number of unused identifiers.

To quantify the effectiveness of the encoding, we define the *storage utilization* U at a given time as the number of encoded spatio-temporal entities divided by the maximum number of spatio-temporal entities that could have been stored thus far. The value of U should be as high as possible, so that a large number of spatio-temporal entities can be accommodated. We also define a *lower bound on storage utilization* $\widehat{U}$, as the minimum number of entities that can be encoded divided by the maximum number of stored entities. This corresponds to a worst-case scenario for a given encoding scheme.

In the following, we describe two alternative overflow management strategies using *dynamic temporal partitioning*: *Basic* and *Chain*. *Basic* handles the temporal skewness in the data and computes a unique identifier for any spatio-temporal entity in contrast to static temporal partitioning. *Chain* provides a better packing of identifiers in the 1D domain of integer values, by providing a lower bound on the storage utilization. More importantly, *Chain* takes into account the properties of the encoding to retain spatial locality despite the higher storage utilization.

Algorithm 1 Basic(s,t,C[]).

```
1: Input: Spatio-temporal entity s(τ), t-th time partition, counters C
2: Output: Unique ID id
3: h ← HilbertValue(τ.x, τ.y)
4: id ← nextID(h, t, C)
5: if (id = −1) then
6:     t ← t + 1
7:     reset counters C
8:     id ← nextID(h, t, C)
9: return id
```

4.1 Basic Strategy

Algorithm 1 describes the *Basic* strategy for overflow management. Whenever a spatial cell in the current temporal partition T_i is filled with 2^k spatio-temporal entities and a new entity is enclosed in this spatial cell, a new time partition T_{i+1} is created. Obviously, all cell counters C maintained need to be reset to zero. Since the temporal partitions are not necessary equi-sized, we need to additionally keep the timestamps when new time partitions are created.

To calculate the lower-bound on storage utilization $\widehat{U}$ for *Basic*, we note that in worst case all entities belong to the same spatial cell. Then, the minimum number of entities for each time partition is 2^k. In general, the maximum number of time partitions is $2^{b-(m+k+1)}$. Thus, the minimum number of entities for the entire 3D grid is

$2^{b-(m+k+1)} * 2^k = 2^{b-(m+1)}$. Notice that this number depends only on the number of total bits (b) and the bits (m) used for the spatial grid, and is independent of the k parameter, because smaller values of k will lead to more temporal partitions. Since the maximum possible number of IDs using $b-1$ bits is 2^{b-1}, we derive that $\widehat{U} = 2^{-m}$. For example, for $m = 8$, $\widehat{U} \approx 0.0039$, which, in turn, means that for b=32, approximately 8.4M spatio-temporal entities can be stored in the worst case. The main problem of *Basic* is the low utilization $\widehat{U}$, if the data is spatially skewed.

Algorithm 2 Chain(s,t,C[],V[]).

```
1:  Input: Spatio-temporal entity s(τ), t-th time partition, counters C,
    chain bit vector V, fill factor ff
2:  Output: Unique ID id
3:  h ← HilbertValue(τ.x, τ.y)
4:  while (V[h] = 1) do
5:      h ← (h + 1) mod 2^m
6:  id ← −1
7:  while (id = −1) do
8:      id ← nextID(h, t, C)
9:      if (id = −1) then
10:         if (∑ C[i] ≥ ff * 2^(m+k)) then
11:             t ← t + 1
12:             reset counters C
13:         else
14:             V[h] ← V[h] + 1
15:             h ← (h + 1) mod 2^m
16: return id
```

4.2 Chain Strategy

This strategy sets a minimum storage utilization that must hold before initializing a new time partition. To achieve this goal, each time a spatial cell overflows, we assign the object that causes the overflow to the next spatial cell based on the ordering produced by the Hilbert curve. In this way, *chains* of cells are created, which contain objects belonging to the first cell of the chain. An advantage of this approach is that it maintains the spatial locality of the IDs since the next cell based on the Hilbert value is expected to be close in the spatial space, but also its entities are assigned to the next interval in the domain of the IDs.

Algorithm 2 describes the *Chain* strategy. To implement this strategy, we additionally need to keep track of the chain. This can be solved by maintaining in memory 2^m bits V (one bit for each spatial cell) that are used to capture if the cell belong to a chain. For instance, consider a grid with 8 cells and the cell with ID 4, whose objects are also stored to the cells with ID 5 and 6 due to overflows. Then, V = 00011000 indicates the chain.

In addition, we define a parameter called *fill factor* ff, which sets the minimum number of points that are stored in a 2D spatial grid. Such a grid can accommodate 2^{m+k} points in total. Essentially, the use of ff is to delay the start of a new temporal partition, until the number of points stored in the spatial grid is at least equal to $ff * 2^{m+k}$. In this case, ff determines the minimum storage utilization. In practice, we want to keep the fill factor ff below a limit (for example 60%), because otherwise chains of large length can be formed, leading to less accurate spatial approximation of the position of an entity.

Algorithm 3 QueryST().

```
1: Input: A spatio-temporal unique identifier s(τ).id, spatio-temporal
   query q
2: Output: True if point τ satisfies q, otherwise False
3: {ID_s, ID_t} ← QueryRDF((s(τ).id,hasPos,?o1),
                            (s(τ).id,hasTime,?o2))
4: {τ.x, τ.y} ← DictionaryLookup(ID_s)
5: {τ.t} ← DictionaryLookup(ID_t)
6: if τ satisfies q then
7:     return True
8: else
9:     return False
```

In Algorithm 2 the main difference compare to *Basic* is the overflow management (lines 9–15). Instead of immediately starting a new time partition, the current number of data objects in the grid is compared to the minimum number of data objects that is needed for the desired storage utilization (line 10). If the goal is achieved, a new time partition is initiated, otherwise the data object is inserted in the cell that is next based on the Hilbert ordering and a chain is created.

5 QUERY PROCESSING FRAMEWORK

5.1 Query Primitives

In this section, we define primitive query operations that are applied to the encoded triples and dictionary, and are used by our query processing algorithms.

5.1.1 RDF Querying. Given an StW-Query $StW(Q, q)$, the RDF part Q of the query is defined by a graph pattern that needs to be evaluated on the stored triples. We define a function: **QueryRDF(*Q*)** that performs this task, namely it returns the identifiers of all resources that match a given graph pattern Q, by examining the stored triples using the most appropriate index. For example, if Q is (?s, *hasStatus*, "SPEED_CHANGE"), the POS index can be used for efficient access. Obviously, this function incurs I/O cost, as it needs to examine the triples stored on disk, in order to produce the result set.

5.1.2 Spatio-temporal Filtering. Given a spatio-temporal entity $s(\tau)$ with known identifier $s(\tau).id$, we discover if this spatio-temporal entity is a candidate for a given StW-Query $StW(Q, q)$. Essentially, for the spatio-temporal constraint q, we distinguish two sets of cells: (a) those cells that are enclosed in q, and (b) those cells that intersect (overlap) with q, but are not contained in q. Any entity that belongs to the former set is a query result, whereas entities that belong to the latter set need to be further examined.

The function **FilterByID(**$s(\tau).id, q$**)** returns -1, if the spatio-temporal entity $s(\tau)$ belongs to a cell that does not overlap with q. If $s(\tau)$ belongs to a cell that is contained in the box q specified in the query, it returns 1. In all other cases, it returns 0. Put differently, in the first case the spatio-temporal entity is not a candidate result, in the second case it definitely belongs to the result set (as far as the spatio-temporal query is concerned), while in the last case refinement is necessary to assess whether it belongs to the result set or not. Notice that this function incurs only computational cost, no I/O cost.

5.1.3 Spatio-temporal Refinement. For any candidate spatio-temporal entity $s(\tau)$ that cannot be filtered out based on its identifier, its exact spatio-temporal values $\tau.x, \tau.y, \tau.t$ need to be retrieved, in order to decide if $s(\tau)$ belongs to the results. In fact, retrieving the spatio-temporal information of a given RDF entity $s(\tau)$ based on its identifier $s(\tau).id$ defines two graph patterns ($s(\tau).id$,*hasPos*,?o1) and ($s(\tau).id$,*hasTime*,?o2) that need to be evaluated on the stored triples. For efficient access, the SPO-index can be used. Algorithm 3 presents the pseudocode of **QueryST**($s(\tau)$,q), a query primitive that returns true if $s(\tau)$ satisfies q, otherwise it returns false. Function **DictionaryLookup**(ID), that retrieves the spatial or temporal values of the entity with identifier ID.

5.1.4 Spatio-temporal Bounding. Given an StW-Query $StW(Q, q)$, we can exploit the encoding, in order to quickly discover the intervals of unique identifiers that the candidate entities belong to. Thus, it is feasible to bound the search space of any RDF query based on these intervals. We define the function **BoundByID(q)** that takes as a parameter a spatio-temporal constraint q, and returns a set of intervals $\{\mathcal{I}_1, \mathcal{I}_2, \dots\}$ of identifiers of entities.

Put differently, any spatio-temporal entity with identifier that does not belong to $\bigcup \mathcal{I}_i$ does not satisfy the spatio-temporal constraint q. In practice, this function exploits the proposed encoding, in order to perform a cost-free filter on available identifiers, thus restricting the search space considerably. Notice that not all returned identifiers correspond to actual objects, as the function returns identifiers that comply with the constraint, but this does not guarantee that every such identifier has been used and assigned to a real entity.

The search space of any RDF query Q can be bounded by $\mathcal{I}$ (denoted as $Q(\mathcal{I})$), which means that only entities with identifiers in $\mathcal{I}$ will be retrieved. By using an SPO-index the execution cost of such a query can be drastically reduced compared to the unbounded query. As a side-note, the intervals are returned annotated, indicating whether the entities in that interval definitely belong to the spatio-temporal constraint, or need to be refined. For simplicity, we omit such implementation details in the depicted pseudocode.

5.2 Query Processing Algorithms

In this section, we present filter-and-refine algorithms (*RDF-First* and *BRDF-First*) that exploit the proposed encoding scheme.

5.2.1 Baseline Algorithm. In the absence of a spatio-temporal index or a filtering mechanism, the only alternative to evaluate a spatio-temporal RDF query, is first to evaluate the RDF graph pattern, and then employ spatio-temporal refinement for all retrieved entities. This *Baseline* approach is not efficient especially in the case of RDF queries that have results sets of high cardinality, where the result set of the spatio-temporal query may be much smaller due to the spatio-temporal constraint.

In the following we present query processing algorithms that will definitely refine fewer spatio-temporal objects than *Baseline*, and adhere to the *filter-and-refine* framework. Since our data have both a spatio-temporal and a semantic (RDF) dimension, the proposed algorithms exploit that spatio-temporal encoding, in order to prune spatio-temporal entities that do not satisfy the spatio-temporal constraint, thereby reducing the number of objects that need to be refined.

Algorithm 4 RDF-First.

```
1:  Input: Spatio-temporal Window query StW(Q, q)
2:  Output: Result set res from B^st
3:  res ← ∅
4:  candIDs ← QueryRDF(Q)
5:  for all s(τ).id ∈ candIDs do
6:      f ← FilterByID(s(τ).id, q)
7:      if f = 0 then
8:          if QueryST(s(τ).id,q)=True then
9:              res ← res ∪ s(τ)
10:     if f = 1 then
11:         res ← res ∪ s(τ)
12: return res
```

Algorithm 5 BRDF-First.

```
1:  Input: Spatio-temporal Window query StW(Q, q)
2:  Output: Result set res from B^st
3:  res ← ∅
4:  I ← BoundByID(q)
5:  for all I_i ∈ I do
6:      candIDs ← QueryRDF(Q(I_i))
7:      for all s(τ).id ∈ candIDs do
8:          f ← FilterByID(s(τ).id, q)
9:          if f = 0 then
10:             if QueryST(s(τ).id,q)=True then
11:                 res ← res ∪ s(τ)
12:         if f = 1 then
13:             res ← res ∪ s(τ)
14: return res
```

5.2.2 RDF-First Algorithm. Algorithm 4 starts by processing the RDF part of the query (line 4) by calling **QueryRDF()**, in order to identify candidate results, which are additionally filtered solely based on their identifier values against the spatio-temporal query (line 6). All remaining candidate results need to be verified (refined) based on their spatio-temporal values (line 8).

Essentially in this algorithm, termed *RDF-First*, filtering is performed when checking the unique identifier values against the query. The refinement takes place after the actual spatio-temporal values have been retrieved. This latter part requires I/Os and is more demanding than the filtering step, which only decodes the identifiers.

5.2.3 BRDF-First Algorithm. Algorithm 5 describes the pseudocode of our *BRDF-First* algorithm that improves *RDF-First*. To achieve this, *BRDF-First* bounds the search space of the RDF query based on the intervals of the spatio-temporal identifiers of the entities. First, **BoundByID(q)** is executed (line 4), and then for each interval of identifiers $\mathcal{I}_i$, a bounded RDF query $Q(\mathcal{I}_i)$ is executed (line 6). The remaining algorithm is similar to *RDF-First*. Thus, in this algorithm, the identifiers are used both for bounding the RDF query and for filtering the results.

BRDF-First refines at most as many entities as *RDF-First*. However, its main gain is the retrieval of fewer candidate entities than *RDF-First*. In other words, the candidate entities of *BRDF-First* are at most

as many as in *RDF-First*, but usually they are much fewer, since *RDF-First* ignores the spatio-temporal constraint when processing the RDF graph pattern Q. The trade-off is that *BRDF-First* will issue multiple queries that scan multiple disjoint intervals of identifiers, but the total number of retrieved entities will be smaller.

6 EXPERIMENTAL STUDY

In this section, we present the results of our evaluation. We implemented all algorithms in Java 8. All experiments run on a machine equipped with an Intel Core processor i7 at 3.6GHz, 16GB RAM.

6.1 Experimental Setup

Data sets. We denote the real data sets used in our empirical study as *DS1* and *DS2* with 2.5M and 20M triples respectively, which differ in size by one order of magnitude. Both data sets contain surveillance information, in the form of Automatic Identification System (AIS) messages for vessels, collected in January 2016. The *DS1* data set covers the Adriatic Sea, while *DS2* the Mediterranean Sea and part of the Atlantic Ocean. We have cleaned, annotated, and transformed these raw data to RDF. The encoded RDF triples are stored on disk, indexed by clustered B^+-trees for efficient access (using XXL: http://www.xxl-library.de/), while the dictionary is kept in main memory.

Query Generation. We create synthetic queries $StW(Q, q)$ as follows. First, we create the spatio-temporal constraint q by selecting a random point in the 3D spatio-temporal space. Then, we create a spatio-temporal box around this point by setting the spatial and temporal size, and vary the spatio-temporal selectivity by setting the size of the box as a percentage (e.g., 25%–100%) of the spatio-temporal grid cell. We ensure that the query returns results, i.e., it does not correspond to an empty spatio-temporal region. Then, we create the RDF graph pattern Q as a *star query*, having as center a spatio-temporal entity. The RDF graph pattern is of varying selectivity, namely we use three classes of constraints Q, with decreasing selectivity. In particular, the graph patterns are ($?s$, $hasStatus$, "STOP"), ($?s$, $hasStatus$, "SPEED_CHANGE"), and ($?s$, $hasStatus$, "TURN"), where "STOP" and "TURN" return the fewest and most results respectively. Following this methodology, for a given setup, we create 100 queries, and we report the average in all cases.

Algorithms. For comparison purposes, apart from the proposed algorithms, we also developed *Baseline*, an algorithm similar to RDF-First, but without using a deliberate encoding scheme. *Baseline* practically corresponds to the approach that would be followed by any existing RDF store. By contrasting the performance of our algorithms to *Baseline*, we can clearly show the net gain achieved by the proposed encoding scheme.

Metrics. Our main metric to evaluate the different encoding strategies is *storage utilization* U at a given time, defined as the number of encoded spatio-temporal entities divided by the maximum number of spatio-temporal entities that could have been encoded at the given time. We also report other auxiliary statistics of the encoding whenever necessary. In addition, we evaluate the performance of all algorithms in terms of the *query execution time*. Moreover, we report the size of the set of objects that needed refinement, as this is a costly step for all algorithms, and typically smaller "refinement sets" determine the performance of an algorithm.

	Encoding Strategy	k	Storage utilization (U)	Temporal partitions	Total bits
DS1	*Basic*	8	2.58%	115	23
	Chain	8	20%	15	20
DS2	*Basic*	4	1.45%	33,572	28
	Chain	4	20.03%	2,436	24
DS2	*Basic*	8	1.46%	2,092	28
	Chain	8	20%	152	24

Table 2: Encoding evaluation (b=32, m=8, and k=$\{4, 8\}$) for both data sets.

6.2 Evaluating the Encoding

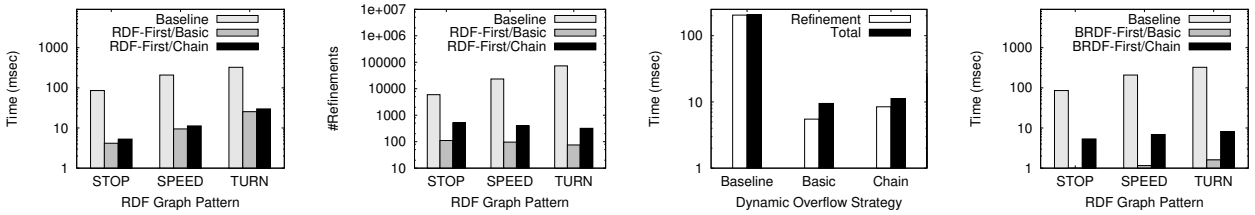
In Table 2, we evaluate the different strategies for encoding and dynamic overflow management, using both data sets and also varying k. Our first observation is that k does not significantly affect the storage utilization. We also observe that *Basic* only achieves low values of storage utilization, 2.58% and 1.45% for the *DS1* and *DS2* data sets respectively. This is due to its eager strategy of initiating a new temporal partition as soon as any spatial cell is filled. Instead, the *Chain* strategy fixes this problem, as it achieves much higher storage utilization (about one order of magnitude improvement).

Chain achieves exactly the storage utilization determined by the fill factor ff, which in this experiment is set to 20%. For the *Chain* strategy, in the *DS1* data set, it holds that for approximate 11.5% of the cells an overflow occurred, the length of the maximum chain is 24, yet the average length of chain is 0.675. This shows that most chains are of small length, thus do not impose excessive overhead during query processing.

The number of created temporal partitions shown in the table, also follows the storage utilization trend, since low utilization is associated with many temporal partitions. Lastly, the column *Total bits* shows the number of bits required for representing the highest identifier for each encoded value in the data set at hand. The lower values of *Chain* in this column show that it can represent the same data with much fewer bits than *Basic*. The *processing time* needed for encoding the RDF data without saving anything to disk is approximately 0.03 – 0.035 msec/entity.

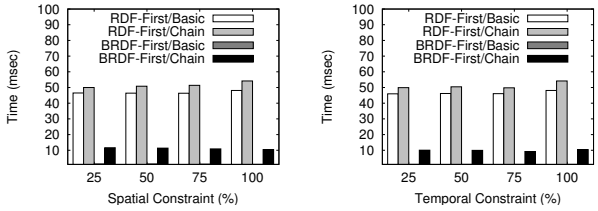
6.3 Query Processing

6.3.1 Encoding Algorithms vs. Baseline. Figure 3 depicts the results obtained when using the *DS1* data set, with b=32, m=8, k=8, for the *RDF-First* algorithm. We evaluate the different encoding strategies and compare the performance of *RDF-First* against *Baseline*, which uses no special encoding. We vary the selectivity of the RDF graph pattern Q, since this is the dominant factor that influences any algorithm (such as *RDF-First* and *Baseline*) that starts by evaluating the RDF graph pattern. The spatio-temporal box (or constraint) q is set to have the size of a spatio-temporal grid cell, but the position of the box is random at the 3D space. We compare the performance of *Baseline* to *RDF-First*, since *RDF-First* processes the query in a



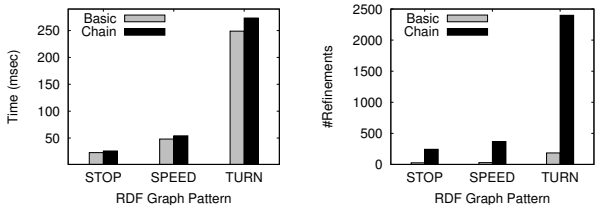
(a) Execution time. (b) Number of entities for refinement. (c) Refinement vs. execution time. (d) Execution time.

Figure 3: Performance of *RDF-First* (3(a)–3(c)) and *BRDF-First* (3(d)) when varying the selectivity of Q (*DS1* data set).



(a) *Varying spatial selectivity.* (b) *Varying temporal selectivity.*

Figure 4: Varying the selectivity of q (*DS2* data set).



(a) *RDF-First.* (b) Number of entities for refinement.

Figure 5: Varying the selectivity of Q (*DS2* data set).

similar way to *Baseline*, but refines fewer entities due to the use of filtering based on the encoding. Notice the log-scale on the y-axis.

In Figure 3(a), in all setups, both encoding methods outperform the *Baseline* by 1-2 orders of magnitude. This is our first strong result that clearly demonstrates the gain and the usefulness of the encoding. As expected, for less selective RDF predicates (e.g., "TURN") that return many results, the execution time of *Baseline* increases rapidly, while this effect on the *RDF-First* algorithm is smoother. Figure 3(b) shows the number of entities that need to be refined: their spatial and temporal information has to be retrieved by accessing the stored RDF triples. *Baseline* has to refine all triples that match the graph pattern Q, since it does not use spatio-temporal encoding in the identifiers. *RDF-First* using the *Basic* encoding strategy needs to refine the fewest entities. Figure 3(c) depicts the time needed for refinement vs. the total time, when the query Q is the RDF graph pattern $(?s, hasStatus,$ "SPEED_CHANGE"). As shown in the chart, the number of entities to be refined is the dominant factor for the performance of this family of algorithms.

Figure 3(d) shows the results obtained with *BRDF-First* for the different encoding strategies, again using log-scale on the y-axis. The chart shows that for the same encoding strategy *BRDF-First* performs better than *RDF-First* for any of the RDF graph patterns used. This is due to *BRDF-First* using the cheap spatio-temporal filtering by means of the encoding, which results in fewer RDF triples that are fetched from disk.

6.3.2 Varying the Query Selectivity. We evaluate the scalability of our algorithms and encoding using the *DS2* data set. We do not use *Baseline* in this set of experiments, as it performs much worse than our algorithms for large data sets. We vary the query selectivity separately for the spatial (Figure 4(a)), temporal (Figure 4(b)) and graph pattern (Figure 5) part of the query.

Figure 4(a) shows the results of all our algorithms when increasing the size of the spatial box in the query q, from 25% to 100% of the spatial grid cell size. We set the temporal interval in q equal to 100%, and we use the graph pattern $(?s, hasStatus,$ "SPEED_CHANGE") as Q. Figure 4(b) shows how increasing the temporal constraint of q affects the performance of our algorithms for the different encoding schemes. We set the spatial query constraint equal to 100% of a spatial cell size. In general, the *BRDF-First* algorithm performs better. Between the encodings, *Basic* is better, followed by *Chain*. However, *Chain* presents a nice solution that trades performance for managing larger data sets, since it addresses the problem of low storage utilization of *Basic*.

In Figure 5, we vary the selectivity of the RDF graph pattern Q, while fixing the spatial and temporal constraints to 100% of a cell's size. We show the performance of *RDF-First* which is most affected by the selectivity of Q. Comparing these results of the *DS2* data set to *DS1* (Figure 3), more time is required and more entities need to be refined. For queries with low selectivity (that return many results), such as "TURN", this increase in time is higher.

7 RELATED WORK

Efficient RDF data management at scale is a hot research topic lately [1, 3–6, 15, 20, 21], due to the increasing size of RDF data. However, only a handful of works address the case of spatio-temporal RDF data, which presents new challenges.

g^{st}-Store [18] is a system for spatio-temporal RDF data. A statement is denoted by a five-tuple $\langle s, p, o, l, t \rangle$, where s, p, o, l, t stand for subject, predicate, object, location and time interval respectively.

Each RDF entity is denoted by a bit string (signature), which is computed based on its neighborhood, thus converting the RDF graph into a signature graph [17]. The system uses a specialized index called ST-tree, which is a balanced tree, and each layer of the ST-tree forms a signature graph with spatiotemporal features. In contrast, our approach does not rely on a specialized index structure, and can be integrated in any RDF system.

SPARQLst [16] is an extension of SPARQL for spatio-temporal queries. SPARQLst extends a commercial relational database that supports spatial objects. Its storage scheme consists of encoded RDF triples in a single triples table, and additional tables for storing the spatial and temporal information separately. Query evaluation is performed by translating a SPARQLst query into a SQL query over the above tables. Strabon [11] is a spatio-temporal RDF store that supports stSPARQL [10] and GeoSPARQL, which are semantic geospatial query languages. Strabon extends the RDF store Sesame that uses PostGIS as backend, thus exploiting its spatial functionality. In contrast, our algorithms do not rely on a relational system, and are readily applicable to any native RDF store as well.

The closest work to ours is the approach followed in [12], where a spatial encoding for RDF data is proposed. However that work is applicable to *static spatial* data, whereas our encoding scheme is designed for *dynamic, spatio-temporal* RDF data. Applying the encoding of [12] to the 3D spatio-temporal space does not work, since it leads to the same problems that static partitioning has, as described earlier in this paper. Instead, our approach takes into account the temporal dimension, which takes values from an ever-increasing domain, and raises technical challenges related to overflow management and keeping the identifiers packed.

Last, but not least, our work is related to space-filling curves (SFCs) that have been extensively used in the spatial data management literature for efficient indexing [9]. In the case of spatial data, the problem is simpler because the data is of static nature. Instead, the case of spatio-temporal data and moving objects, where data is dynamic, is more similar to our case, and SFCs have also been used in this context. For instance, we refer to B^+-tree based indexes for moving objects, such as B^x-tree [7] or B^x_r-tree [8], that also use SFCs. One differentiating factor is that in our case we must use the SFC in order to assign unique identifiers to objects, so that they can be used to encode RDF entities. This raises some technical challenges related to overflow management, namely how to achieve the combination of high storage utilization and efficient processing, which are addressed by our dynamic temporal partitioning schemes. In particular, *Chain* presents a nice trade-off that balances these two objectives, based on our experimental evaluation.

8 CONCLUSIONS

In this paper, we proposed an encoding scheme for spatio-temporal RDF data, coupled with efficient query processing algorithms. Our algorithms belong to the filter-and-refine framework, and achieve 1-2 orders of magnitude improvement over a typical filter-and-refine algorithm for RDF processing that does not use a deliberate encoding. Our approach is designed to be applicable in setups where the data is dynamic, which raises challenges with respect to the management of the temporal dimension. Experiments on real data sets demonstrate the usefulness of the proposed encoding scheme.

ACKNOWLEDGMENTS

This work is supported by project datAcron, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 687591, and from the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under grant agreement No 1667.

REFERENCES

[1] Ibrahim Abdelaziz, Razen Harbi, Semih Salihoglu, Panos Kalnis, and Nikos Mamoulis. 2015. SPARTex: A Vertex-Centric Framework for RDF Data Analytics. *PVLDB* 8, 12 (2015), 1880–1891.

[2] Christophe Claramunt, Cyril Ray, Elena Camossi, Anne-Laure Jousselme, Melita Hadzagic, Gennady L. Andrienko, Natalia V. Andrienko, Yannis Theodoridis, George A. Vouros, and Loïc Salmon. 2017. Maritime data integration and analysis: recent progress and research challenges. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*. 192–197.

[3] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. In *Proceedings of SIGMOD*. 289–300.

[4] Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, Seyed-Mehdi-Reza Beheshti, and Sherif Sakr. 2015. DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *PVLDB* 8, 6 (2015), 654–665.

[5] Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis, and Nikos Mamoulis. 2015. Evaluating SPARQL Queries on Massive RDF Datasets. *PVLDB* 8, 12 (2015), 1848–1851.

[6] Jiewen Huang, Daniel J. Abadi, and Kun Ren. 2011. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB* 4, 11 (2011), 1123–1134.

[7] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. 2004. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*. 768–779.

[8] Christian S. Jensen, Dalia Tiesyte, and Nerius Tradisauskas. 2006. Robust B+-Tree-Based Indexing of Moving Objects. In *7th International Conference on Mobile Data Management (MDM 2006), Nara, Japan, May 9-13, 2006*. 12.

[9] Ibrahim Kamel and Christos Faloutsos. 1994. Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*. 500–509.

[10] Manolis Koubarakis and Kostis Kyzirakos. 2010. Modeling and Querying Metadata in the Semantic Sensor Web: The Model stRDF and the Query Language stSPARQL. In *Proceedings of ESWC*. 425–439.

[11] Kostis Kyzirakos, Manos Karpathiotakis, Konstantina Bereta, George Garbis, Charalampos Nikolaou, Panayiotis Smeros, Stella Giannakopoulou, Kallirroi Dogani, and Manolis Koubarakis. 2013. The Spatiotemporal RDF Store Strabon. In *Proceedings of SSTD*. 496–500.

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

[13] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. 2001. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE Trans. Knowl. Data Eng.* 13, 1 (2001), 124–141.

[14] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *VLDB J.* 19, 1 (2010), 91–113.

[15] Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. 2016. Processing SPARQL queries over distributed RDF graphs. *VLDB J.* 25, 2 (2016), 243–268.

[16] Matthew Perry, Prateek Jain, and Amit P. Sheth. 2011. SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries. In *Geospatial Semantics and the Semantic Web*. 61–86.

[17] Dong Wang, Lei Zou, Yansong Feng, Xuchuan Shen, Jilei Tian, and Dongyan Zhao. 2013. S-store: An Engine for Large RDF Graph Integrating Spatial Information. In *Database Systems for Advanced Applications, 18th International Conference, DASFAA 2013, Wuhan, China, April 22-25, 2013. Proceedings, Part II*. 31–47.

[18] Dong Wang, Lei Zou, and Dongyan Zhao. 2014. g^{st}-Store: An Engine for Large RDF Graph Integrating Spatio-temporal Information. In *Proceedings of EDBT*. 652–655.

[19] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: sextuple indexing for semantic web data management. *PVLDB* 1, 1 (2008), 1008–1019.

[20] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB* 6, 4 (2013), 265–276.

[21] Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min Wang. 2013. EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *Proceedings of ICDE*. 565–576.