The VLDB Journal manuscript No.
(will be inserted by the editor)

SRX: Efficient Management of Spatial RDF Data

Konstantinos Theocharidis · John Liagouris · Nikos Mamoulis · Panagiotis Bouros · Manolis Terrovitis

Received: date / Accepted: date

Abstract We present a general encoding scheme for the efficient management of spatial RDF data. The scheme approximates the geometries of the RDF entities inside their (integer) IDs and can be used, along with several operators and optimizations we introduce, to accelerate queries with spatial predicates and to re-encode entities dynamically in case of updates. We implement our ideas in SRX, a system built on top of the popular RDF-3X system. SRX extends RDF-3X with support for three types of spatial queries: range selections (e.g. find entities within a given polygon), spatial joins (e.g. find pairs of entities whose locations are close to each other), and spatial k nearest neighbors (e.g. find the three closest entities from a given location). We evaluate SRX on spatial queries and updates with real RDF data, and we also compare its performance with the latest versions of three popular RDF stores. The results show SRX's superior performance over the competitors; compared to RDF-3X, SRX improves its performance for queries with spatial predicates while incurring little overhead during updates.

Konstantinos Theocharidis
University of Peloponnese
E-mail: ktheocharid@uop.gr
IMSI 'Athena'
E-mail: kotheoha@imis.athena-innovation.gr

John Liagouris
ETH Zürich
E-mail: liagos@inf.ethz.ch

Nikos Mamoulis
University of Ioannina
E-mail: nikos@cs.uoi.gr

Panagiotis Bouros
Johannes Gutenberg University Mainz
E-mail: bouros@uni-mainz.de

Manolis Terrovitis
IMSI 'Athena'
E-mail: mter@imis.athena-innovation.gr

Keywords spatial RDF data · GeoSPARQL · bit encoding · Hilbert curve · RDF-3X · query evaluation · updates

1 Introduction

The Resource Description Framework (RDF) has become a standard for expressing information that does not conform to a crisp schema. Semantic-Web applications manage large knowledge bases and data ontologies in the form of RDF. RDF is a simple model, where all data are in the form of ⟨*subject, property, object*⟩ (SPO) triples, also known as *statements*. The subject of a statement models a *resource* (e.g., a Web resource) and the property (a.k.a. *predicate*) denotes the subject's relationship to the object, which can be another resource or a simple value (called *literal*). A resource is specified by a uniform resource identifier (URI) or by a *blank node* (denoting an unknown resource). An RDF knowledge base can be modeled as a graph, where nodes are resources or literals and edges are properties.

SPARQL is the standard query language for RDF data, used to express *query graph patterns* that have to be matched in the RDF data graph. The GeoSPARQL standard [10], defined by the Open Geospatial Consortium (OGC), extends RDF and SPARQL to represent geographic information and support spatial queries. Geospatial filter functions are used to express spatial predicates between entities in SPARQL queries. stSPARQL [19] has similar features.

Despite the large volume of work on indexing and querying large RDF knowledge bases [6,9,11,14,15,27,36,37,38,39,40,41], only a few works focus on the effective handling of spatial semantics in RDF data. In particular, the current spatial extensions of RDF stores (e.g., Virtuoso [4], GraphDB [1], Parliament [3], Strabon [20], and others [13,34,35]) focus mainly on supporting GeoSPARQL features, and less on performance optimization. The features and weak-

nesses of these systems are reviewed in Sect. 3. On the other hand, there is a large number of spatial entities (i.e., resources) in RDF knowledge bases (e.g., YAGO [5]). Thus, the power of the state-of-the-art RDF stores is limited by the inadequate handling of spatial semantics, given that it is not uncommon for user queries to include spatial predicates. At the same time, spatial data management systems [16] can only be used to index and search the spatial semantics of the entities, but do not support graph pattern search.

In this paper, we fill this gap by presenting SRX (Spatial RDF-3X), a system built on top of the open-source RDF-3X store [27] to efficiently support spatial queries and updates. SRX inherits the basic design principles of RDF-3X, which encodes all values that appear in SPO triples by identifiers with a help of a dictionary, and models the RDF knowledge base as a single long table of ID triples. A SPARQL query can then be modeled as a multi-way join on the triples table. The system creates a clustered B^+-tree for each of the six SPO permutations; the query optimizer identifies an appropriate join order, considering all the available permutations and advanced statistics [26]. RDF-3X is known to have robust performance in comparison studies on various RDF datasets and query benchmarks [11,27,39]. Although we have chosen RDF-3X as a basis for SRX, our techniques are also applicable to other RDF stores, e.g. [39]. In a nutshell, SRX includes the following extensions over RDF-3X:

Index Support for Spatial Queries. Similar to previous spatial extensions of RDF stores (e.g., [13]), SRX includes a spatial index (i.e., an R-tree [17]) for the geometries associated to the spatial entities. This facilitates the efficient evaluation of queries with very selective spatial components.

Spatial Encoding of Entities. The identifiers given to RDF resources in the dictionary of RDF-3X (and other RDF stores) do not carry any semantics. Taking advantage of this fact, we encode spatial approximations inside the IDs of entities (i.e., resources) associated to spatial locations and geometries. This mechanism has several benefits. First, for queries that include spatial components, the IDs of resources can be used as cheap filters and data can be pruned without having to access the exact geometries of the involved entities. Second, our encoding scheme does not affect the standard ordering (i.e., sorting) of triples used by the RDF-3X evaluation engine, therefore it does not conflict with the RDF-3X query optimizer; in other words, the original system's performance on non-spatial queries is not compromised. Finally, our encoding scheme adopts a flexible hierarchical space decomposition so that it can easily handle spatially skewed datasets and updates without the need to re-assign IDs for all entities.

Spatial Join Algorithms. We design spatial join algorithms tailored to our encoding scheme. Our *Spatial Merge Join* (SMJ) algorithm extends the traditional merge join algorithm to process the filter step of a spatial join at the approximation level of our encoding, while (i) preserving *interesting orders* of the qualifying triples that can be used by succeeding operators, and (ii) not breaking the pipeline within the operator tree. In typical SPARQL queries which usually involve a large number of joins, the last two aspects are crucial for the overall performance of the system. Our *Spatial Hash Join* (SHJ-ID) operates with unordered inputs, using their encodings to identify fast candidate join pairs.

Spatial kNN Algorithms. We design two k nearest neighbors (kNN) algorithms that make use of our encoding scheme. Both are based on previous work on grid-based kNN query evaluation. The first one operates on unordered input whereas the second exploits interesting orders and can be combined with other order-preserving operators to improve performance and further reduce the memory footprint.

Spatial Query Optimization. In addition to including standard selectivity estimation models and techniques for spatial queries, we extend the query optimizer of RDF-3X to consider spatial filtering operations that can be applied on the spatially encoded entities. For this purpose, we augment the original join query graph of a SPARQL expression to include binding of spatial variables via spatial join conditions.

Dynamic Spatial Re-encoding. Changes in real RDF datasets are the rule rather than the exception. Such changes occur as new triples are added and old ones are removed or updated, and the need for re-encoding spatial entities arises naturally. To tackle this problem with a low overhead in performance, we carefully integrate a dynamic re-encoding technique with the original update mechanism of RDF-3X.

An earlier version of SRX without support for kNN and dynamic re-encoding in case of updates has been presented in [21]. In this paper, we evaluate SRX by comparing it with the latest versions of two commercial spatial RDF management systems: Virtuoso [4] and Graph-DB [1], and a popular free RDF management system: Strabon [20]. For query evaluation, we use two real datasets: LinkedGeoData (LGD) [2] and YAGO [5]. To evaluate dynamic re-encoding, we generated a realistic update benchmark — the first one using real data — based on the deltas we collected between different versions of LGD and YAGO. The results demonstrate the superior performance and robustness of SRX over the competitors; SRX improves the performance of the original RDF-3X for queries with spatial predicates, while incurring insignificant overhead when performing updates.

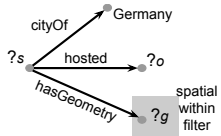
2 Preliminaries

The SPARQL queries we consider follow the format:

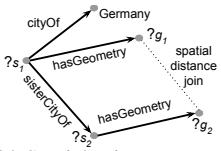
Select [projection clause]
Where [graph pattern]
Filter [condition]

subject	*property*	*object*
Dresden	cityOf	Germany
Prague	cityOf	CzechRepublic
Leipzig	cityOf	Germany
Wrocław	cityOf	Poland
Ostrava	cityOf	CzechRepublic
Hannover	cityOf	Germany
Dresden	sisterCityOf	Wrocław
Dresden	sisterCityOf	Ostrava
Leipzig	sisterCityOf	Hannover
Dresden	hosted	Wagner
Leipzig	hosted	Bach
Wagner	hasName	"Richard Wagner"
Wagner	performedIn	Leipzig
Wagner	performedIn	Prague
Wagner	performedIn	Ostrava
Bach	performedIn	Leipzig
Mozart	performedIn	Hannover
Mozart	performedIn	Dresden
Dresden	hasGeometry	"POINT (13.6, 51)"
Prague	hasGeometry	"POINT (14.3, 50)"
Leipzig	hasGeometry	"POINT (12.3, 51.3)"
Wrocław	hasGeometry	"POINT (16.9, 51.1)"
Ostrava	hasGeometry	"POINT (18.2, 49.8)"
Hannover	hasGeometry	"POINT (9.7, 52.4)"
...	...	...

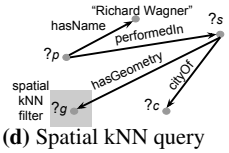
(a) RDF triples



(b) Spatial Within query



(c) Spatial Join query



(d) Spatial kNN query

Fig. 1: Example of RDF data and three spatial queries

The Select clause includes a set of variables that should be instantiated from the RDF knowledge base (variables in SPARQL are denoted by a ? prefix). A graph pattern in the Where clause consists of triple patterns in the form of *s p o* where any of the *s*, *p* and *o* can be either a constant or a variable. Finally, the Filter clause includes one or more *spatial predicates*. For the ease of presentation, in our discussion and examples, we consider only WITHIN range predicates (for spatial selections), DISTANCE predicates (for spatial joins), and kNN predicates (for *k* nearest neighbors). However, we emphasize that the results of our work are directly applicable to all spatial predicates defined in the GeoSPARQL standard [10]. In addition, we use a simplified syntax for expressing queries and not the one of the GeoSPARQL standard because the latter is verbose.

As an example, consider the RDF knowledge base partially listed in Fig. 1a. Literals and *spatial literals* (i.e., geometries) are in quotes. An exemplary query with a range predicate is:

Select ?*s* ?*o*
Where ?*s* cityOf Germany . ?*s* hosted ?*o* .
?*s* hasGeometry ?*g* .
Filter WITHIN(?*g*, "POLYGON(...)");

This query finds the cities of Germany within a specified polygonal range together with the persons they hosted. Note that there are three variables involved (?*s*, ?*o*, and ?*g*) connected via a set of triple patterns which also include constants, i.e., Germany. For example, if POLYGON(...) covers the area of East Germany, (Dresden, Wagner) and (Leipzig, Bach) are results of this query. The query is represented by the pattern graph of Fig. 1b. In general, queries can be represented as graphs with *chain* (e.g., ?s_1 hosted ?s_2. ?s_2 performedIn ?s_3.) and *star* (e.g., ?*s* cityOf ?*o*. ?*s* hosted Wagner.) components.

Another exemplary query, which includes a spatial join predicate, represented by the pattern graph of Fig. 1c, is:

Select ?s_1 ?s_2
Where ?s_1 cityOf Germany . ?s_1 sisterCityOf ?s_2 .
?s_1 hasGeometry ?g_1 . ?s_2 hasGeometry ?g_2 .
Filter DISTANCE(?g_1, ?g_2) < "300km";

This query asks for pairs of sister cities (i.e., ?s_1 and ?s_2) such that the first city (i.e., ?s_1) is in Germany and the distance between them does not exceed 300km. In the exemplary RDF base of Fig. 1a, (Dresden, Wrocław) and (Leipzig, Hannover) are results of this query while (Dresden, Ostrava) is not returned as the distance between Dresden and Ostrava is around 500km.

Finally, an examplary query with a kNN predicate, represented by the pattern graph of Fig. 1d, is the next:

Select ?*s* ?*c*
Where ?*p* hasName "Richard Wagner" . ?*p* performedIn ?*s* .
?*s* cityOf ?*c* . ?*s* hasGeometry ?*g* .
Filter kNN(?*g*, "POINT(...)", 2);

This query asks for the two closest to the specified point cities where Richard Wagner has performed, together with their respective countries. For example, if POINT(...) refers to the city of Chemnitz (12.8, 50.8), then the result of the query in the RDF base of Fig. 1a consists of the tuples (Leipzig, Germany) and (Prague, CzechRepublic).

Besides queries, we also consider delete, insert, and update operations on RDF data. Updates in SPARQL (and GeoSPARQL) are expressed via DELETE and INSERT statements following the format:

Delete|Insert [triples]

For example, to update the name of the entity Wagner in the RDF base of Fig. 1a, one can simply apply the following two statements:

Delete Wagner hasName "Richard Wagner"
Insert Wagner hasName "Wilhelm Richard Wagner"

3 Related Work

RDF Storage and Query Engines. There have been many efforts toward the efficient storage and indexing of RDF data. The most intuitive method is to store all ⟨*subject, property, object*⟩ (SPO) statements in a single, very large *triples* table. The RDF-3X system [27] is based on this simple architecture. RDF-3X (following an idea from previous work) uses a *dictionary* to encode URIs and literals as IDs. Indexing is then applied on the ID-encoded SPO triples. Fig. 2 illustrates a dictionary and the ID-encoded triples for the

ID	*URI/literal*
1	Dresden
2	cityOf
3	Germany
4	Prague
5	CzechRepublic
6	Leipzig
...	...

(a) Dictionary

subject	*property*	*object*
1	2	3
4	2	5
6	2	3
...	...	...

(b) ID-encoded SPO triples

Fig. 2: Use of Dictiorary

RDF base of Fig. 1a. RDF-3X creates a clustered B$^+$-tree index for each of the six SPO permutations (i.e., SPO, SOP, PSO, POS, OSP, OPS). A SPARQL query is transformed to a multi-way self-join query on the triples table; the query engine binds the query variables to SPO values and joins them (if the query contains literals or filter conditions, these are included as selection conditions). A query is first translated by replacing URIs or literals by the respective IDs and then evaluated using the six indices; finally, the query results (in the form of ID-triples) are translated back to their original form. The six indices offer different ways for accessing and joining the triples; RDF-3X includes a query optimizer to identify a good query evaluation plan. The system favors plans that produce *interesting orders*, where merge joins are pipelined without intermediate sorts. In addition, a run-time *sideways information passing* (SIP) mechanism [28] reduces the cost of long join chains. RDF-3X maintains nine additional aggregate indices, corresponding to the nine projections of the SPO table (i.e., SP, SO, PS, PO, OS, OP, S, P, O), which provide statistics to the query optimizer and are also useful for evaluating specialized queries. The query optimizer was extended in [26] to use more accurate statistics for star-pattern queries. RDF-3X employs a compression scheme to reduce the size of the indices by differential storage of consecutive triples in them. Hexastore [36] is a contemporary to RDF-3X proposal, which also indexes SPO permutations on top of a triples table. An earlier implementation of a triples table by Oracle [15] uses materialized join views to improve performance.

An alternative storage scheme is to decompose the RDF data into *property* tables: one binary table is defined per distinct property, storing the SO pairs that are linked via this property. In order to avoid the case of having a huge number of property tables, this extreme approach was refined to a *clustered-property* tables approach (used by early RDF stores, like Jena [37] and Sesame [14]), where correlated tables are clustered into the same table and triples with infrequent properties are placed into a *left-over* table. Abadi et al. [6] use a column-store database engine to manage one SO table for each property, sorted by subject and optionally indexed on object.

A common drawback of the column-store approach and RDF-3X is the potentially large number of joins that have to be evaluated, together with the potentially large intermediate results they generate. Atre et al. [9] alleviate this problem by introducing a 3D compressed bitmap index, which reduces the intermediate results before joining them. A similar idea was recently proposed in [39]; the participation of subjects and objects in property tables is represented as a sparse 3D matrix, which is compressed. Yet another storage architecture was proposed in [11]. The idea is to first cluster the triples by subject and then combine multiple triples about the same subject into a single row. Thus, the system saves join cost for star-pattern queries, however, it may suffer from redundancy due to repetitions and null values.

Trinity [40] is a distributed memory-based RDF data store, which focuses on graph query operations such as random walk distance, reachability, etc. RDF data are represented as a huge (distributed) graph and query evaluation is done in an exploration-based manner; starting from the most selective predicates, query variables are bound progressively, while the RDF graph is browsed. Trinity's power lies on the fact that memory storage eliminates the otherwise very high random access cost for graph exploration. gStore [41] is an earlier, graph-based approach, which models SPARQL queries as graph pattern matching queries on the RDF graph. More recently, EmptyHeaded [7, 8] employed novel worst-case optimal join algorithms to accelerate pattern matching queries on RDF graphs.

Spatial Extensions of RDF Stores. Parliament [10], built on top of Jena [37], implements most of the features of GeoSPARQL. Strabon [20], developed contemporarily with Parliament, extends Sesame [14] to manage spatial RDF data stored in PostGIS. Strabon adopts a column-store approach, implementing two SO and OS indices for each property table. Spatial literals (e.g., points, polygons) are given an identifier and are stored in a separate table, which is indexed by an R-tree [17]. Strabon extends the query optimizer of Sesame to consider spatial predicates and indices. The optimizer applies simple heuristics to push down (spatial) filters or literal binding expressions in order to minimize intermediate results. Strabon and Parliament are based on old RDF stores (i.e., Jena and Sesame) and lack sophisticated query optimization techniques.

Brodt et al. [13] extend RDF-3X [27] to support spatial data. The extension is limited, since range selection is the only supported spatial operation. Furthermore, query evaluation is restricted to either processing the non-spatial query components first and then verifying the spatial ones or the other way around. Finally, the opportunity of producing an interesting order from a spatial index (in order to facilitate subsequent joins) is not explored.

Geo-Store [34] is another spatial extension of RDF-3X. Geo-Store divides the space by a grid and orders the cells using a *Hilbert* space-filling curve. Each geometry literal g (e.g. "POINT (...)") is approximated by the Hilbert order

g.ID of the cell that includes it. Then, for all triples of the form *s* hasGeometry *g*, a triple *s* hasPos *g*.ID is added to the data. During query evaluation, an extra join with the hasPos triples is applied to perform the filter step of spatial queries. Geo-Store supports only spatial range and *k* nearest neighbor queries, but not spatial joins. In addition, it does not extend the query optimizer of RDF-3X to consider spatial query components. Finally, besides increasing the size of the original database with the introduction of hasPos triples, it is not clear how its encoding can handle complex spatial literals, such as "POLYGON (...)", which may span multiple cells of the grid.

S-Store [35] is a spatial extension of gStore [41]. Although S-Store was shown to outperform gStore for spatial queries, it handles spatial information only at a high level (i.e., the data are primarily indexed based on their structure). Spatial RDF queries are also supported by many commercial systems, such as Oracle, Virtuoso [4], and GraphDB [1], however, details about their internal design are not public.

Finally, [31] recently introduced DiStRDF that adapts our encoding scheme [21] to support RDF queries with spatio-temporal filters on top of Spark.

4 A Basic Spatial Extension

In the remainder of the paper, we present the steps of extending a standard query evaluation framework for triple stores (i.e., the framework of RDF-3X) to efficiently handle the spatial components of RDF queries. In RDF-3X, a query evaluation plan is a tree of operators applied on the base data (i.e., the set of RDF-triples). The leaves of the tree are any of the 6 SPO clustered indices. The operators apply either selections or joins. Each operator addresses a triple of the query pattern and instantiates the corresponding variables; the instantiated triples (or query subgraphs) are passed to the next operator, until they reach the root operator, which computes instances for the entire query graph.

This section outlines the basic (but essential) spatial extension to RDF-3X, which improves the spatial RDF-3X extension of Brodt et al. [13] to support spatial join and kNN query evaluation. We also discuss drawbacks of the basic extension that motivated us to design the spatial encoding scheme described in Sect. 5 and the query evaluation algorithms that use it in Sect. 6.

Spatial Indexing. Spatial entities i.e., resources associated to spatial literals like POINT and POLYGON, are indexed by an R-tree [17]. For each entity associated to a polygon, there is an entry at a leaf of the R-tree of the form (*mbr*, ID), where *mbr* is the minimum bounding rectangle (MBR) of the polygon. For each entry associated to a point *pt*, there is a (*pt*, ID) entry.

Spatial Selections. Given a query with a spatial selection Filter condition, the optimizer may opt to use the R-tree to evaluate this condition first and retrieve the IDs of all entities that satisfy it.[1] However, the output fed to the operators that follow (i.e., those that process non-spatial query components) is in a random order. Thus, query evaluation algorithms that rely on the input being in an *interesting order* (such as merge-join) are inapplicable. On the other hand, if the spatial selection is evaluated after another (i.e., non-spatial) operator, the R-tree cannot be used because the input is no longer indexed. Therefore, in this case, the system must look up the geometries of the entities that qualify the preceding operator at the dictionary, incurring significant cost. Fig. 3a and Fig. 3b illustrate two alternative plans for the spatial selection query of Fig. 1b. The plan of Fig. 3a uses the R-tree to perform the spatial selection and joins the result with the instances of triple ?*s* cityOf Germany. Finally, the join results are joined with the results of ?*s* hosted ?*o*. The plan of Fig. 3b first evaluates the non-spatial part of the query and then looks up and verifies the geometries of all ?*s* instances in it (i.e., the R-tree is not used here).

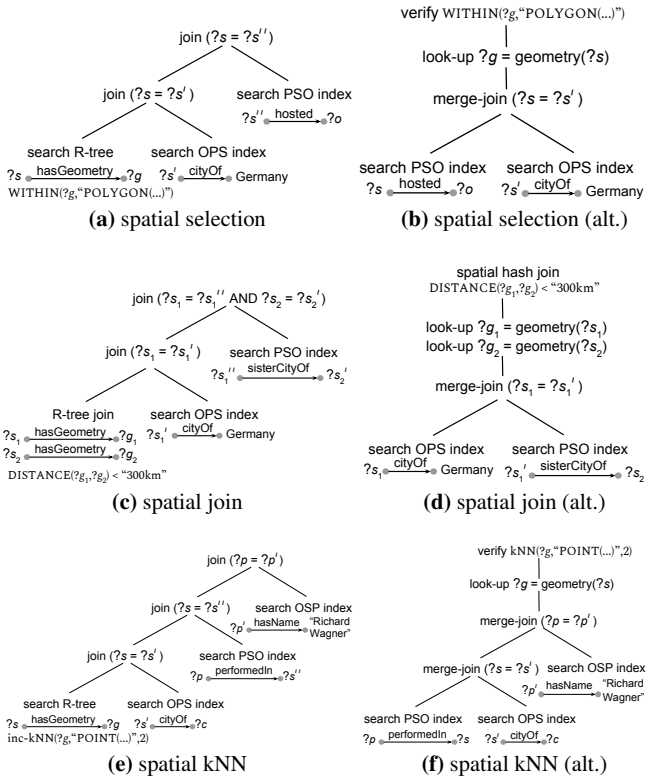


Fig. 3: Possible query plans in the basic extension

Spatial Joins. The R-tree can also be used to evaluate spatial join Filter conditions, by applying join algorithms based on

[1] For entities that have point geometries, the spatial selection can be evaluated using only the R-tree. If the entities have non-point geometries, the R-tree search may result in false positives, thus, the final results of the spatial filter are confirmed by retrieving the exact geometries from the dictionary.

R-trees. We implemented three algorithms for this purpose. First, the R-tree join algorithm [12] can be used in the case where both spatially joined variables involved in the Filter condition are instantiated directly from the base data and do not come as outputs of other query operators. Second, we use the SISJ algorithm [24] for the case where the R-tree can be used only for one variable. Finally, we implemented a spatial hash join (SHJ) algorithm [22] for the case where both inputs of the spatial join filter condition are output by other operators.[2] As in the case of spatial selections, spatial join algorithms do not produce interesting orders and for spatial join inputs that are instantiated by preceding query operators, the system has to perform dictionary look-ups in order to retrieve the geometries of the entities before the join. Fig. 3c and Fig. 3d illustrate two alternative plans for the spatial join query of Fig. 1c. The plan of Fig. 3c applies an R-tree self-join [12] to retrieve nearby ($?s_1$, $?s_2$) pairs and then binds $?s_1$ with the result of $?s_1$ cityOf Germany. The output is then joined with the result of $?s_1$ sisterCityOf $?s_2$. The plan of Fig. 3d first evaluates the non-spatial part of the query and then looks up the geometries of all ($?s_1$, $?s_2$) pairs, and joins them using SHJ. In the following, we briefly describe SISJ and SHJ for completeness.

SISJ joins a spatial input A which is not indexed, with an R-tree B. Assuming that we want to use H hash buckets, SISJ first divides the entries at the uppermost level of B that contains at least H entries into H groups based on their spatial proximity. The i-th group has as spatial extent the MBR of all entries in group i. Bucket B_i contains all objects in the subtrees of B pointed by the entries in the i-th group. The objects from A are hashed to buckets such that bucket A_i contains all objects that intersect the spatial extent of the i-th group. Finally, each A_i is spatially joined in memory with B_i (e.g., using plane sweep). Our SHJ implementation pulls the smallest of the two join inputs (based on the query optimizer's estimation) and constructs from it a spatial hash table in memory. Each hash bucket corresponds to a cell in a 2D grid with side equal to the distance join threshold ε. Each entity from the hashed join input is assigned to all buckets (cells) that it spatially overlaps. Then, SHJ pulls the records from the other input one by one and, for each spatial entity e, (i) it retrieves e's geometry from the dictionary, (ii) identifies the cell c whereto e belongs, and (iii) accesses the buckets that correspond to c and its neighboring cells to find candidate entities that can match with e based on their spatial approximations. For each such candidate entity e', the operator computes the exact distance between e and e', and outputs the join pair (e, e') if the distance is at most ε.

Spatial kNN. The R-tree can also be used to evaluate a spatial kNN predicate in the Filter clause. In this case, the nearest entities are fetched from the R-tree and fed to the operators that follow. Since some of these entities might be filtered out by subsequent operators, we should use an *incremental NN algorithm* for R-trees [23] (an operation often referred to as *distance browsing*). As in the case of spatial selections, the drawback of using this algorithm is that the IDs of the fetched entities are in random order, preventing the use of efficient operators that rely on interesting orders. On the other hand, when the R-tree is not used, the kNN evaluation needs to perform dictionary lookups to fetch the geometries of all entities that qualify the RDF part of the query and keep track using a heap, the k nearest entities. Fig. 3e and Fig. 3f depict two possible plans that correspond to the two options above for the query pattern of Fig. 1d.

[2] If the spatial join inputs are very small, we simply fetch the geometries of the input entity sets and do a nested-loops spatial join.

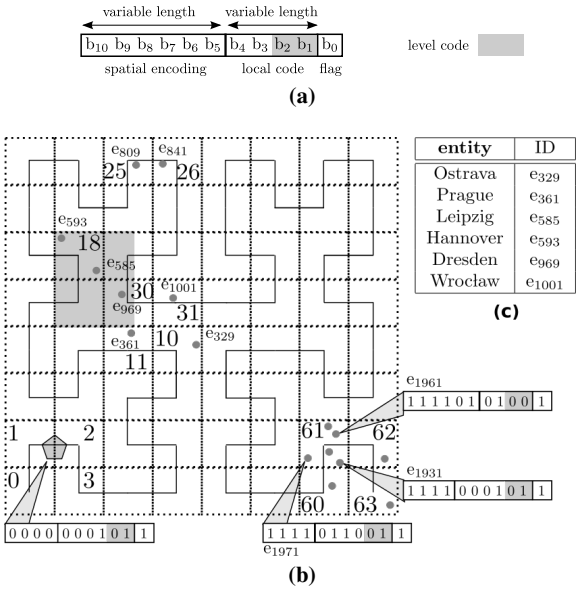


Fig. 4: Spatial encoding of entity IDs

5 Encoding the Spatial Dimension

We observe that in most RDF engines, the IDs given to resources or literals at the dictionary mapping do not carry any semantics. Instead of assigning random IDs to resources, we propose to *encode* into the ID of a resource an approximation of the resource's location and geometry that can be used to (i) apply spatial Filter conditions on-the-fly in a query evaluation plan, and (ii) define spatial operators that apply on the approximations.

Fig. 4b illustrates the *Hilbert* space filling curve, a classic encoding scheme of spatial locations into one-dimensional values. We partition the space using a grid, and order the cells based on the curve. We then divide the ID given to a spatial resource r into two components: (i) the Hilbert order of the cell where r spatially resides occupies the m most significant bits (where $2^{m/2} \times 2^{m/2}$ is the resolution of the

grid), and (ii) a *local* identifier which distinguishes r from other resources that reside in the same cell as r. Since the RDF data may also contain resources or literals, which are not spatial, we use a different range of ID values for non-spatial resources with the help of the least significant bit as a flag. In the toy example of Fig. 4a, the least significant bit (b_0) indicates whether the entity modeled by the ID is spatial ($b_0 = 1$) or non-spatial ($b_0 = 0$), the next 4 bits are used for the local identifier, and the 6 most significant bits encode the Hilbert order of the cell. For example, in Fig. 4b, entity e_{1961} is spatial (b_0 is set) and it is located in the cell with Hilbert order 111101 (cell with ID 61), having local code 0100. For a non-spatial resource, bit b_0 would be 0 and the remaining ones would not have any spatial interpretation. Fig. 4c illustrates which IDs encode the cities of Fig. 1a.

In the case of a skewed dataset, a cell may overflow, i.e., there could be too many entities falling inside it rendering the available bits for the local codes of entities in it insufficient. In this case, entities that do not fit in a full cell are assigned to the parent of the cell in the hierarchical space decomposition. For instance, consider the data in Fig. 4b and assume that the cell with ID 61 is full and that the entity e_{1931} cannot be assigned to it. e_{1931} will be assigned to the parent cell, i.e., the square that consists of the cells 60, 61, 62, and 63. This cell's encoding has 4 bits, that is, 2 bits less than its children cells. These 2 bits are now used for the local encoding of entities in it. Intuitively, as we go up in the hierarchy of the grid, each cell can accommodate more entities. An entity that must be assigned to an overflown cell ends in the first non-full ancestor of that cell as we go up in the hierarchy. The $\lceil log_2(m/2) \rceil$ least significant bits of the local code area are reserved to encode the level of the spatially-encoded cell in the ID (the most detailed level being 0). In our example, $m = 6$, hence, 2 bits of the local code are used to denote the level of the cell that approximates each entity.

The encoding we described is also used for arbitrary geometries that may overlap with more than one cells of the bottom level. For example, the polygon at the lower left corner of the grid of Fig. 4b spans across cells with IDs 1 and 2, thus, it will be assigned to their parent cell, which has a spatial encoding 0000. Due to the variable number of bits given to the spatial approximations, the encoding is also suitable for dynamic data (i.e., inserted entities that fall into overflown cells are given less accurate approximations).

The most important benefit of the spatial encoding is that the (approximate) evaluation of spatial predicates can be seamlessly combined with the evaluation of non-spatial patterns in SPARQL. For example, spatial Filter conditions included in a query which are bound to entity variables (for example, ?s hasGeometry ?g, Filter WITHIN (?g, "POLYGON(...)") can be evaluated on-the-fly at any place in the evaluation plan where the entity variable (e.g., ?s) has been instantiated, by decoding the IDs of the instances. Note that the spatial mapping is only approximate (based on the conservative grid approximation of the spatial locations); by applying a spatial predicate on the approximations (i.e., cells) of the entities, false hits may be included in the results, which need to be verified. Still, for many entities, the spatial approximation suffices to confirm that they are definitely included (or not) in the query result. This way, random accesses for retrieving their exact geometries are avoided.

A side-benefit of using a Hilbert-encoded grid to approximate the object geometries is that by counting the number of resources in each cell (counting is already performed by the mapping scheme), we can have a spatial histogram to be used for selectivity estimation in query optimization (this issue will be discussed in detail in Sect. 7). SRX uses the encoding we described to accelerate queries with spatial predicates as shown in the next section.

6 Query Evaluation

We now show how the encoding scheme of SRX further extends the basic framework presented in Sect. 4 to apply efficient spatial filters directly on the entities IDs and reduce the number of dictionary lookups as well as the number of expensive spatial operations on the actual geometries.

All operators we describe in this section evaluate the spatial predicates in two phases: first, by applying the spatial predicate on the IDs of the entities (filtering phase) and, second, by fetching the actual geometries only for the results that could not be verified in the first phase. In general, the sooner we apply the on-the-fly filtering the better because it does not incur any I/O cost and its CPU cost is negligible[3]. For spatial range and join predicates, the on-the-fly filtering can be done early: after each non-spatial operator that instantiates entity variables, which also appear in a WITHIN or DISTANCE predicate, the condition is applied to the spatially encoded IDs of the entities. In such cases, after applying the filter, we also append a *verification bit* (or vbit) to the tuples that pass the filter. This bit is used in the second phase as follows: if, for a tuple, the verification bit is 1, the tuple is guaranteed to qualify the corresponding spatial predicate (no verification is required). On the other hand, if the bit is 0, this means that it is unknown at this point whether the exact geometries of the entities in the tuple qualify the spatial predicate (however, they cannot be pruned based on their spatial approximations encoded in their IDs). By the end of processing all non-spatial query components, for tuples having their vbits 0, the system fetches the exact geometries of the involved entities and perform verification of the spatial Filter conditions.

[3] Most spatial predicates, when translated to the grid-based approximations of the encoding, involve distance computations and/or cheap geometry intersection tests.

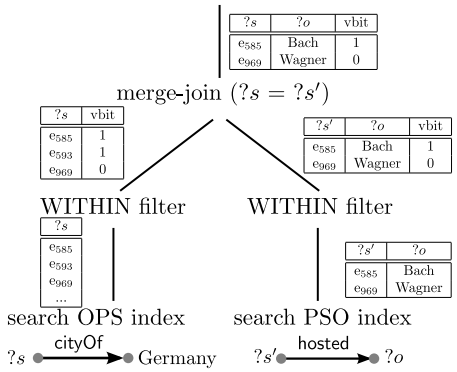


Fig. 5: Plan for the query of Fig. 1b

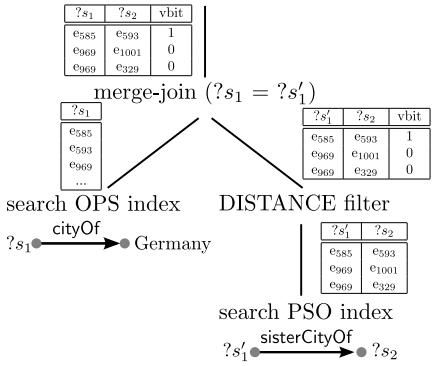


Fig. 6: Plan for the query of Fig. 1c

6.1 Spatial Range Filtering

Spatial range queries bind a pattern variable to geometries that are spatially restricted by a range. As an example, consider again the query depicted in Fig. 1b. Our encoding scheme allows the filtering phase of the spatial range query to be performed on-the-fly while scanning the indices, as illustrated by the evaluation plan of Fig. 5. The plan searches the OPS and PSO indexes in order to fetch and merge-join ($?s = ?s'$) the two lists that qualify patterns $?s$ cityOf Germany, $?s'$ hosted $?o$, i.e., the plan follows the logic of the plan shown in Fig. 3b. Taking advantage of the spatial encoding, before the merge-join, the plan of Fig. 5 applies the spatial filter for ($?s$ hasGeometry $?g$, WITHIN($?g$,"POLYGON (...)")) on the instances of $?s$ that arrive from scanning the OPS and PSO indexes; a vbit is appended to each survived tuple, to be used by the next operators. In this example, assume that the spatial entities and the spatial range (i.e., "POLYGON (...)") are the points and the shadowed range, respectively, shown in Fig. 4b. Entities e_{809} and e_{841} are filtered out from the left scan, because they are not within the cells that intersect the query spatial range. Entity e_{969} survives spatial filtering, but we cannot ensure that it qualifies the spatial range predicate either, because its cell-ID is not completely covered by the spatial query range; therefore the vbit for the tuples that involve e_{969} is 0. On the other hand, the vbit for tuples containing e_{585} or e_{593} is 1 as their cell-ID is completely covered by the spatial range. Therefore, after the merge-join, we only have to fetch and verify the geometry of e_{969}. Range filtering is applied at the bottom of query plans, after each index scan that contains a respective spatial variable.

6.2 Spatial Join Filtering

Similar to spatial range selections, the filtering phase for binary spatial join predicates can also be applied on-the-fly, as soon as the IDs of candidate entity pairs are available. As an example, consider the join query depicted in Fig. 1c. A possible query evaluation subplan is given in Fig. 6, which follows the flow of the plan shown in Fig. 3d; however, the plan of Fig. 6 applies the spatial join filter (i.e., the distance filter) early. By the time the candidate pairs ($?s'_1$, $?s_2$) are fetched by the index scan on PSO, the filter is applied so that only the pairs of entities that cannot be spatially pruned are passed to the next operator. Assume that the pairs that qualify $?s'_1$ sisterCityOf $?s_2$ are as shown at the right-bottom side of Fig. 6, above the search PSO index operator. Assume that the distance threshold (i.e., 300km) corresponds to the length of the diagonal of each cell in Fig. 4. After applying the distance spatial filter on all ($?s'_1$, $?s_2$) pairs produced by the PSO index scan, the pairs that survive are (e_{585}, e_{593}), (e_{969}, e_{1001}) and (e_{969}, e_{329}). However, only entities e_{585} and e_{593} are guaranteed to be within ε distance as they belong to same cell; thus, the vbit for pair (e_{585}, e_{593}) is 1. When the pairs are merge-joined ($?s_1 = ?s'_1$) with the results of the OPS index-scan on the left (for $?s_1$ cityOf Germany), the vbits of qualifying tuples are carried forward.

In contrast to the range filter that always appears at the bottom level of the operator tree, distance join filtering can be applied on any intermediate relation that contains two joined spatial variables. This case is possible when two relations are first joined on attributes other than the spatial entities. In Sect. 7.1, we show how the query optimizer can identify all pairs of spatially joined variables in a query, for which distance join filtering can be applied; here, we only gave an example with a pair coming from an index scan.

6.3 Spatial Merge Join on Encoded Entities

In this section, we propose a *spatial merge join* (SMJ) operator that applies directly on the spatial encodings (i.e., the IDs) of the entities from the two join inputs. SMJ assumes that both its inputs are sorted by the IDs of the spatial entities to be joined. Like the spatial filters discussed above, this algorithm only produces pairs of entities for which the exact geometries are likely to qualify the spatial join predicate (typically, a DISTANCE filter). Again, a verification bit is used to indicate whether the join condition is definitely

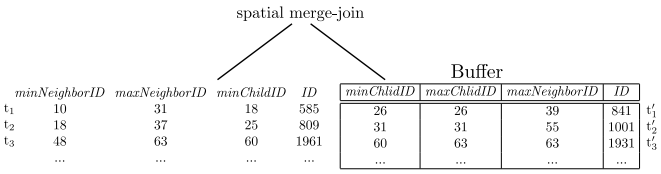


Fig. 7: Example of SMJ

qualified by a pair. Besides using the spatially encoded IDs of the entities, SMJ takes advantage of and preserves the ID-based sorting of its inputs. Thus, the algorithm does not break the pipeline within the operator tree, as any other spatial join algorithm would. Note that SMJ is a binary join algorithm that takes two inputs, while the filtering technique discussed in Sect. 6.2 takes a single input of candidate join pairs and merely applies the join condition on the entity-ID pairs on-the-fly.

Similarly to a classic merge join algorithm, SMJ uses a buffer B_R to cache the streaming tuples from its right input R. For each entity e_l read from the left input L, SMJ uses the ID of e_l to compute the minimum and maximum cell-IDs that could include entities e_r from R, which could possibly pair with e_l in the join result, based on the given DISTANCE filter. SMJ then keeps reading tuples from input R and buffering them into B_R, as long as they are likely to join with e_l. As soon as B_R is guaranteed to contain all possible entities that may pair with e_l, SMJ computes all join results for e_l and discards e_l (and potentially tuples from B_R).

We now provide the details of SMJ. The algorithm is based on the (on-the-fly and on-demand) computation of four cell IDs for each entity e based on e's ID. First, *minNeighborID* and *maxNeighborID* are the minimum and maximum cell-IDs that could include entities that pair with e in the join result, respectively. To compute these cells, we have to expand e's cell based on the distance join threshold and find the minimum and maximum cell-ID that intersects the resulting range. For example, consider entity e_{841} contained in cell with ID 26 in Fig. 4b and assume that the join distance threshold equals the diagonal length of a cell. For this entity, *minNeighborID*=18 and *maxNeighborID*=39. Second, *minChildID* and *maxChildID* correspond to the minimum and maximum cell-IDs that have a common non-empty ancestor (in the hierarchical Hilbert space decomposition) with the cell of e. For entity e_{841} which has only empty ancestors, the *minChildID* and *maxChildID* are both 26, that is, the cell ID of e_{841}. For e_{1931}, the *minChildID* and *maxChildID* are 60 and 63 respectively because e_{1931} is assigned to a cell at the first level of the grid.

At each step, the distance join is performed between the current entity e_l from the left input and all entries in B_R. After reading e_l, SMJ reads entries e_r and buffers them into B_R and stops as soon as e_r's *minChildID* is greater than the *maxNeighborID* of e_l; then we know that we can join e_l and all entities in B_R and then discard e_l, because any unseen tuples from R cannot be included within the required distance from e_l.[4] For example, consider the buffered inputs of Fig. 7 that have to be joined. The *maxNeighborID* of the first entity e_{585} on the left is smaller than the *minChildID* of entry e_{1931}, therefore e_{585} cannot be paired with entries after e_{1931} (that are guaranteed to have *minChildID* greater than the *maxNeighborID* of e_{585}).[5] Thus, for any e_l, we only need to consider all entities in R before the first entity having *minChildID* greater than the *maxNeighborID* of e_l.

After e_l has been joined, it is discarded. At that point we also check if buffered tuples in B_R can also be removed. In order to decide this, we use *maxNeighborID* of each entity on the right. In case this is smaller than the *minChildID* of the next entity in L, then the right entry can be safely removed from the buffer without losing any qualifying pairs. Below, we give a pseudocode for SMJ.

Algorithm: SMJ
Input : Two join inputs L and R; a distance threshold ε
Output : Grid-based spatial distance join of L and R

```
1 Initialize (empty) buffer B_R;
2 e_r = R.get_next(); add e_r to B_R;
3 while e_l = L.get_next() do
4     Prune from B_R all tuples e_r such that e_r.maxNeighborID <
      e_l.minChildID;
5     while e_l.maxNeighborID ≥ e_r.maxChildID do
6         e_r = R.get_next(); add e_r to B_R;
7     join e_l with all tuples in B_R and output results to the next
      operator;
```

We now discuss some implementation details. First, the required *min/maxNeighborID* and *min/maxChildID* for the entries are computed fast on-the-fly by simple operations. In particular, *min/maxChildID*s are computed by shifting (or masking) bits to keep only those most significant bits that encode the cell ID at a particular level of the grid (cf. Sect.5). For the neighbor IDs, we rely on the id-to-offset and offset-to-id Hilbert transformations as follows. First, we use e's ID and the (normalized) distance threshold ε to identify the offsets of the bottom-level cells that must be examined. Then, we transform the offsets back to the corresponding cell IDs, and we use the latter to compute the minimum and maximum entity IDs by masking bits in the local code (i.e., the least significant bits - cf. Sect.5). Second, for joining an entity e_l from L, we scan through the qualifying entities of B_R and compute their grid-based distances to e_l, but only for entities whose *minChildID-maxChildID* range overlaps

[4] Recall that the inputs are sorted by ID and that entities may be encoded at different granularities due to data skew or geometry extents. Therefore, using the cell-ID of e_r alone is not sufficient and we have to use the *minChildID* of e_r.

[5] The fact that the entities arrive from the inputs sorted by their IDs guarantees that they are also sorted based on their *minChildID*s.

with the *minNeighborID-maxNeighborID* range of e_l; this is a cheap filter used to avoid grid-based distance computations. Finally, we buffer all tuples that have the same entity ID (in either input). For such a buffer, we perform the join only once but generate all join pairs.

6.4 Spatial Hash Join on Encoded Entities

If either of the two inputs of a spatial join is not ordered with respect to the joined entities, SMJ is not applicable. In this case we can still use the IDs of the joined entities to perform the filter step of the spatial join. The idea is to apply a *spatial hash join* (SHJ-ID) algorithm (similar to that proposed in [22]) using the approximate geometries of the entities taken from their IDs.[6] SHJ-ID simply uses the existing assignment of the entities to the cells of the grid (as encoded in their IDs) and considers each such cell as a distinct bucket. The only difference from a typical spatial hash join algorithm is that in the bucket-to-bucket join phase, we have to consider all levels of the encoding scheme. Therefore, each bucket from the left input, corresponding to a cell c, is joined with all buckets from the right input which correspond to all cells that satisfy the DISTANCE filter with c. The output of SHJ-ID is verified as soon as the geometries of the candidate pairs are retrieved from disk.

6.5 Spatial kNN on Encoded Entities

kNN predicates are evaluated differently from WITHIN and DISTANCE predicates in that no early spatial filtering or verification bits are utilized. We introduce two kNN operators that make use of the encoding: one for handling entities whose IDs come from the previous operator in a random order (Sect. 6.5.1), and a second one that exploits ordering (Sect. 6.5.2) and, thus, can be used efficiently in combination with other order-preserving operators. Both operators are applied in a pipelined fashion at the root of the operator tree (i.e., on the output of the previous operators) and are inspired by the work in [25]. The difference compared to previous kNN operators is the integration with the multi-level encoding scheme of Sec. 5. This integration enables us to (i) compute approximate distances using arithmetic operations on the entity IDs, and (ii) leverage the interesting orders preserved by previous operators in the query plan to reduce random I/Os and improve performance. Random I/Os are common in index-based kNN algorithms from Sect. 4, which we compare with our approach in Sec. 9.2.

6.5.1 kNN on Unsorted Entity IDs

The logic of first kNN operator is given in Algorithm KNN-UNSORTED-INPUT. The operator takes as input a point p (the one specified in the Filter clause of the query) along with an iterator I on the tuples coming from the previous operator in the query plan. Let t be a tuple in I and e be the ID of the spatial entity in t that is used in the evaluation of the kNN predicate. The operator uses two priority queues Q_1 and Q_2 to keep tuples ordered in ascending Euclidean distance of e from p's actual geometry: in the former queue, the distance has been calculated based on e's cell whereas in the latter based on e's actual geometry.

The evaluation proceeds in two phases. First, the operator pulls all tuples from the previous operator in the query plan and populates Q_1 (*lines* 3-4). The function POPULATE_Q_1 uses the multi-level encoding scheme to compute the minimum distance between e's cell and p (*minDist*) and keeps entries in ascending *minDist*. Note that *minDist* is an approximation of the exact distance between the entity e and the point p; the latter is computed only in the second phase of the algorithm (*lines* 5-9) where the operator starts draining Q_1 to populate Q_2. Specifically, each time an entry is popped from Q_1, the exact geometry of e is retrieved via a

Algorithm: KNN-UNSORTED-INPUT
Input : Input I from the previous operator; a point p
Output : k tuples from I that satisfy the kNN predicate
Param. : The number k

```
1  let Q1, Q2 be two priority queues;
2  lastDist = ∞;
3  while t = I.get_next() do
4      POPULATE_Q1(t, p);
5  while Q1 is not empty do
6      (t, minDist) = Q1.pop();
7      if minDist ≥ lastDist then
8          break;
9      POPULATE_Q2(t, p, lastDist, k);
10 return all t tuples in Q2;
```

Function: POPULATE_Q_1(t: TUPLE, p: POINT)

```
1  let e be the ID of the spatial entity in tuple t;
2  let c be the grid cell e belongs to;        //extracted from e
3  if c is the last-level cell then
4      minDist = 0;
5  else if c is an upper-level cell then
6      find the bottom-level child cell of c, let c_b, which is the
       nearest to the given point p;
7      set minDist to the minimum distance of c_b from p;
8  else
       //c is a bottom-level cell
9      set minDist to the minimum distance of c from p;
10 Q1.push((t, minDist));          //keep in ascending minDist
```

Function: POPULATE_Q_2(t: TUPLE, p: POINT, *lastDist*: FLOAT, k: INTEGER)

```
1  let e be the ID of the spatial entity in tuple t;
2  retrieve e's geometry from dictionary and compute the exact
   distance between e and p, let exactDist;
3  Q2.push((t, exactDist));          //keep in ascending exactDist
4  if |Q2| = k then
5      set lastDist equal to exactDist of the k-th entry in Q2;
```

[6] Recall that the actual geometries of the entities have not been retrieved yet; otherwise, SHJ [22] would be used (see Sect. 4).

Algorithm: KNN-SORTED-INPUT
Input : Input I from the previous operator; a point p
Output : k tuples from I that satisfy the kNN predicate
Param. : The number k

```
1  let Q1, Q2 be two priority queues;
2  lastDist = ∞;
3  let c_p be the bottom-level cell that contains p;
4  limit = prevLimit = COMPUTE_LIMIT(c_p);
   //load first round of input data into Q1
5  READ_NEXT(I, limit, p);
6  for each rectangle r in the first zone around c_p do
7      compute the minimum distance of r from p, let minDist;
8      Q1.push((r, minDist));      //keep in ascending minDist
9  while Q1 is not empty do
10     (entry, minDist) = Q1.pop();
11     if minDist ≥ lastDist then
12         break;
13     if entry is a tuple t with a spatial entity then
14         POPULATE_Q2(t, p, lastDist, k);
15     else
           //entry is a rectangle r
16         find the maximum bottom-level cell ID c_m falling in r;
17         limit = COMPUTE_LIMIT(c_m);
18         if limit > prevLimit then
19             prevLimit = limit;
               //load next round of input data
20             READ_NEXT(I, limit, p);
21         let r' be the next zone rectangle in the direction of r;
22         set minDist to the minimum distance of r' from p;
23         Q1.push((r', minDist));      //in ascending minDist
24 return all t tuples in Q2;
```

dictionary lookup and the tuple t is pushed into Q_2 using now the exact distance between e and p (*exactDist* in function POPULATE_Q_2). The draining of Q_1 stops when the algorithm pops an entity e whose minimum possible distance from p is at least equal to the current exact distance of the k-th element in Q_2 (*lines* 7-8 in KNN-UNSORTED-INPUT).

In contrast to Q_2 that holds at most k tuples from the input I, Q_1 is populated with all tuples from I in the first phase of KNN-UNSORTED-INPUT. The intuition behind this strategy is to sort the entities based on their cells and use this ordering to minimize the expensive geometry lookups in the second phase. Since the IDs of the spatial entities come out of order and each next entity may fall anywhere in the grid, Q_1 must store all input tuples from I. This increases the memory footprint (and the latency) of KNN-UNSORTED-INPUT significantly when the RDF part of the query is not selective. When the spatial entities come in order, we can tackle this problem with the kNN operator we describe next.

6.5.2 kNN on Sorted Entity IDs

The second kNN operator we introduce uses an adaptation of the CPM technique from [25] and its logic is given in Algorithm KNN-SORTED-INPUT. The core idea here is to exploit the ordering of entities and avoid draining the iterator I, i.e., pulling the whole output from the pervious operator in the query plan. To do so, the evaluation proceeds in "zones" starting from the (bottom-level) cell of the point p in the Filter condition. Each such zone consists of four rectangles (*up*, *down*, *left*, *right*), which in turn consist of bottom-level grid cells and form a "circular" area around p's cell, as shown in Fig. 8a. The operator follows the same steps as in CPM and extends the original technique to (i) work with our multi-level encoding scheme, and (ii) pull tuples from the input gradually, as it examines the zones.

Function: READ_NEXT(I: ITERATOR, *limit*: INTEGER, p: POINT)

```
1  while t = I.peek() do
2      let e be the ID of the spatial entity in tuple t;
3      if e ⩽ limit then
4          t = I.get_next();      //Pull happens at this point
5          POPULATE_Q1(t, p);
6      else break;
```

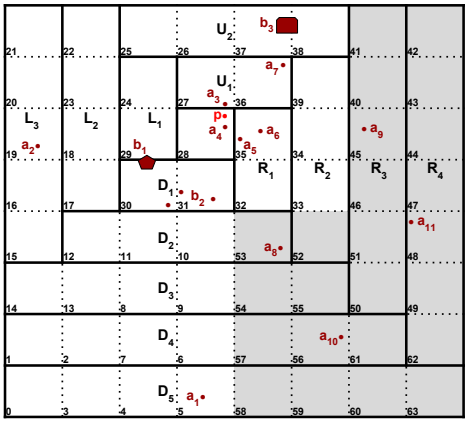
Function: COMPUTE_LIMIT(c_{id}: BOTTOM-LEVEL CELL)

```
1  let c_i be the parent cell of c_id at the i-th grid level;      //c_0 ≡ c_id
2  let m_i be the maximum encoded spatial entity ID in c_i;
3  return max_{0≤i≤13} m_i;      //14 grid levels with 32-bit IDs
```

First, the operator identifies the bottom-level cell c_p that contains the given point p (*line* 3). It then computes the maximum ID among all spatial entities that might fall in c_p (*line* 4). This is done in function COMPUTE_LIMIT, which simply returns the maximum spatially encoded ID that exists in the database and falls either in c_p or in a parent cell of c_p[7]. COMPUTE_LIMIT is a very cheap function that requires only a few lookups in the grid statistics kept in memory. The returned ID serves as an upper *limit* to bound the number of tuples pulled from I when populating Q_1 in function READ_NEXT. The intuition here is that, at each step of the algorithm, only the tuples of the current examined zone (initially p's cell) must be pulled from the input. To do so, the operator first peeks into I (*line* 1 in READ_NEXT) to check the entity ID e of the next tuple and decide if this ID is at most equal to the *limit*; if so, this means that e's actual geometry might fall in the examined zone, thus, the tuple is pulled from I (*line* 4 in READ_NEXT) and the algorithm continues with peeking the next tuple; otherwise none of the following entities fall in the examined zone, thus, the algorithm exits the loop (*line* 6 in READ_NEXT), computes the distance of each rectangle in the first zone from p, as in original CPM, and adds the respective entry to Q_1 (*lines* 6-8 in KNN-SORTED-INPUT).

Then, the operator continues similarly to KNN-UNSORTED-INPUT, i.e. it starts pulling from Q_1 (*line* 9) to populate Q_2 with the exact distances. The termination condition in

[7] In case there are no spatial entities in the database falling in c_p or one of its parent cells, then as *limit* we use the first free (i.e. the minimum) spatial ID for an entity in c_p.



entity	ID (binary) - (decimal)
a_1	000101|0000|1 - 161
a_2	010011|0000|1 - 609
a_3	011011|0000|1 - 865
a_4	011100|0000|1 - 897
a_5	100011|0000|1 - 1121
a_6	100011|0100|1 - 1129
a_7	100100|0000|1 - 1153
b_1	0111|000001|1 - 899
b_2	0111|000101|1 - 907
b_3	1001|000001|1 - 1155

Fig. 8: An example grid **(a)** with 64 cells at the bottom level (11-bit encoding) ordered according to the Hilbert curve and organized in CPM zones (L_i, R_i, U_i, D_i) around a query point p in cell 28. The entity IDs are shown on the right **(b)** in binary and decimal format

lines 11-12 is the same as in KNN-UNSORTED-INPUT. The only difference here is that, whenever the algorithm encounters a new rectangle r in Q_1, the latter is used to update (i.e. increase) the *limit* and pull the required additional tuples (if any) from the input I (*lines* 16-20). After that, the algorithm also expands the search space to the next zone (*lines* 21-23) by adding to Q_1 the rectangle of the next zone that is in the same direction (up, $down$, $left$, $right$) as r with respect to p's cell. This is CPM's actual control flow and the correctness of the computation relies on the correctness of the original method (cf. Lemma 3.1 in [25]). As a final comment, KNN-SORTED-INPUT is designed to pull as few tuples from I as possible (it exhausts I only in the worst case, i.e. when *limit* is greater than all spatial IDs in I) and, thus, tends to perform much better than KNN-UNSORTED-INPUT, as we show in Sect. 9.

Example. Consider the grid of Fig. 8 where a_i denotes a spatial entity encoded at the bottom level and b_i denotes a spatial entity encoded at the exact next level. For simplicity, assume that there are no entities at higher levels. Assume also that each bottom-level cell has a side of 1 metric unit. Consider a query point p falling in cell 28 and let $k = 2$. Algorithm KNN-SORTED-INPUT first pulls from the input I and inserts into Q_1 all tuples with spatial entities that may fall in p's bottom-level cell, i.e. all tuples from I before a tuple with a spatial entity ID greater than $b_2 = 907$ (recall that tuples in I are in ascending spatial entity ID order). Then, the algorithm proceeds with the insertion of the first zone rectangles L_1, R_1, U_1, D_1 resulting in a priority queue $Q_1 = \{(a_4,0),(b_1,0),(b_2,0),(a_3,0.1),(U_1,0.1),(R_1,0.2),(L_1,0.8),(D_1,0.9),(a_2,2.8),(a_1,4.9)\}$. Numbers in Q_1 depict the Euclidean distance of the respective entry (grid cell or zone rectangle) from p's geometry. At the next step, the algorithm starts pulling entries from Q_1 to populate Q_2. When it reaches the first rectangle entry U_1, it computes the new $limit = b_3 = 1155$. At that point, we have $Q_1 = \{(R_1,0.2),(a_5,0.2),(a_6,0.2),(a_7,\sqrt{0.05}),(b_3,\sqrt{0.05}),(L_1,0.8),(D_1,0.9),(U_2,1.1),(a_2,2.8),(a_1,4.9)\}$ and $Q_2 = \{(a_3,0.12),(a_4,0.21)\}$. Distances in Q_2 have now been computed using the Euclidean distance between the entry's actual geometry and the point p. The algorithm then pops entry R_1, which results in updating Q_1 only with $(R_2,1.2)$, and terminates when it pops a_7 whose $minDist = \sqrt{0.05}$ is less than the $lastDist = 0.21$ of the previous entry a_4 popped from Q_1. Eventually, $Q_2 = \{(a_3,0.12),(a_4,0.21)\}$ and the entries a_3, a_4 are returned.

7 Query Optimization

In this section we describe our extensions to the query optimizer of RDF-3X, in order to take into consideration (i) the R-tree index and the query evaluation plans that involve it (see Sect. 4) and (ii) the query evaluation techniques described in Sect. 6 for spatial range and join queries. The encoding-based kNN operators (Sect. 6) do not affect query optimization as they are always applied after the RDF part.

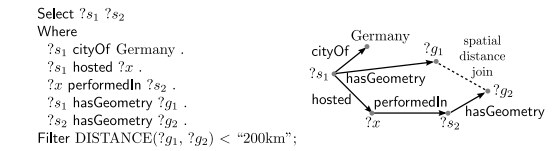
7.1 Augmenting the Query Graph

Consider the query depicted in Fig. 9a. This query includes a spatial distance join between the geometries $?g_1$ and $?g_2$. The filtering phase of the spatial distance join can also be applied on the variables $?s_1$ and $?s_2$, using their IDs, as explained in Sect. 6.3. We call such variables *spatial variables*:

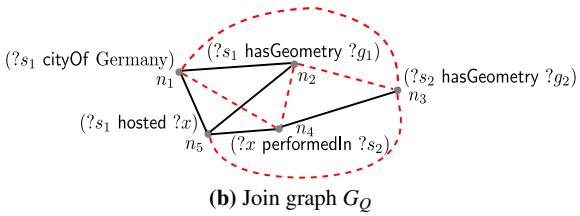
Definition 1 (SPATIAL VARIABLE) A variable $?s_i$ at the subject position of a triple pattern $?s_i$ hasGeometry $?g_i$ that appears in the Where clause of a query Q is called a spatial variable. We say that two spatial variables $?s_i$, $?s_j$ $(i \neq j)$ are joined iff $?g_i$ and $?g_j$ appear in the same DISTANCE predicate in the Filter clause of Q.

Spatial variables are identified in the beginning of the optimization process and they are used to augment the initial join query graph G_Q with additional join edges that correspond to the filtering step of the spatial operation. For example, the initial G_Q for the RDF query of Fig. 9a is the graph shown in Fig. 9b, considering solid lines only as edges; the nodes of G_Q are the triples of the RDF query graph and there is an edge between every pair of nodes that have at least one common variable. An ordering of the edges of G_Q corresponds to a join order evaluation plan.

The procedure of augmenting G_Q is given in Algorithm AUGMENT. First, we identify all spatial variables in the query Q; in our example, $?s_1$ and $?s_2$. Note that a spatial variable $?s_i$ may also appear either as subject or object in triple patterns, other than $?s_i$ hasGeometry $?g_i$. The second step is to



(a) RDF query



(b) Join graph G_Q

Fig. 9: Augmenting a query graph

collect all pairs of nodes in G_Q that include at least one spatial variable. In the example of Fig. 9b, all nodes include one of $?s_1$ and $?s_2$. Then, for each pair of nodes (n_i, n_j), where $n_i \neq n_j$, such that n_i includes $?s_1$ and n_j includes $?s_2$, we either add a new edge (if no edge exists between n_i and n_j) or we add the spatial join predicate (e.g., DISTANCE($n_i.s_i$, $n_j.s_j$) < "200km") in the set of predicates modeled by the edge between these two nodes (these are equality predicates for their common variables). For instance, n_4 and n_5 in the initial G_Q are connected by an edge with predicate $n_4.x = n_5.x$, but after the augmentation the predicates on this edge are $n_4.x = n_5.x$ *and* DISTANCE($n_4.s_2, n_5.s_1$) < "200km". This implies that the optimizer will consider two possible subplans for joining n_4 with n_5. The first one will first perform the equality join on x and then evaluate the distance predicate whereas the second subplan will first perform the filtering phase of the spatial join on (s_1, s_2) and then apply the equality on x. In the augmented G_Q for our example (Fig. 9b) the additional edges are denoted with dashed lines.

If a query Q also includes WITHIN predicates, in the end of the augmentation procedure and for each spatial variable $?s$ whose geometry $?g$ participates in a WITHIN predicate, we add a condition of the form WITHIN($?s$, GEOMETRY) to the set of filters of Q, so that this filter can be applied in any (intermediate) relation that contains the spatial variable $?s$. Note that this condition differs from the existing spatial condition WITHIN($?g$, GEOMETRY) in that it includes the spatial variable $?s$ and not the geometry variable $?g$. Similarly, for each pair (s_i, s_j) of joined spatial variables, we add the corresponding spatial join condition to Q's existing filters, so that this filter can be applied on every (intermediate) relation that includes both the spatial variables s_i and s_j. Overall, the final augmented G_Q may include more edges than the initial G_Q, additional predicates in the edges, and a set of general spatial filters for variables or pairs of variables that can be applied on intermediate results of subplans.

Algorithm: AUGMENT
Input : A query Q and its initial join query graph G_Q
Output : An augmented query graph G_Q for Q

```
1  Identify all triples in Q that include at least one spatial variable
   in as subject or object. Each such triple corresponds to a node
   of G_Q;
2  for each pair ?s_i, ?s_j of joined spatial variables do
3      for each pair of nodes (n_i, n_j) ∈ G_Q, such that n_i includes
       ?s_i and n_2 includes ?s_j do
4          if there is no edge in G_Q between n_i and n_j then
5              Add a new edge denoting the filtering phase of
               the spatial join of ?s_i and ?s_j;
6          else
7              Add the filtering phase of the spatial join
               predicate of ?s_i and ?s_j in the predicate list of
               edge between n_i and n_j;
8  For each spatial variable ?s appearing in a WITHIN predicate,
   add WITHIN(?s, GEOMETRY) to filtering conditions of Q;
9  For each pair of spatial variables ?s_i, ?s_j (i ≠ j) joined in Q,
   add DISTANCE(?s_i, ?s_j) Op ε to filtering conditions of Q;
10 return G_Q;
```

7.2 Spatial Join Operators

Our plan generator can place a spatial join operation at every level of the operator tree. Table 1 summarizes all possible cases of the L and R inputs of a spatial join (if L and R are swapped there is no difference because the join is symmetric). The right column includes the join algorithms, which the plan generator of the optimizer considers in each case.

Depending on whether the inputs of the join are indexed, sorted, or unsorted, there are different algorithms to be considered. If both join inputs come ordered by the IDs of the spatial entities to be joined, then SMJ (Sect. 6.3) is the algorithm of choice. In the special case where both inputs are the results of $?s_i$ hasGeometry $?g_i$ patterns applied on the entire set of triples, besides of applying SMJ on the SPO (or SOP) index, we can apply an R-tree self-join [12] on the R-tree index. When just one of the inputs, e.g., R, is a result of a $?s_i$ hasGeometry $?g_i$ pattern, besides SMJ, we can also apply the SISJ algorithm [24]. In this case, we also consider Index Nested Loops join using the R-tree, by applying one spatial range query for each tuple of the other input, e.g., L. This is expected to be cheap only when L is very small. Finally, when either L or R are unsorted, SMJ is not applicable and we can use SHJ-ID on the entity IDs (Sect. 6.4), or either SISJ or SHJ depending on whether one of the inputs is a direct result of a $?s_i$ hasGeometry $?g_i$ pattern or not. We also consider Index Nested Loops or Nested Loops, if one of the inputs is too small.

7.3 Spatial Query Optimization

We extend the query optimizer of RDF-3X to consider all possible spatial join cases and algorithms outlined in Sect. 7.2.

Table 1: Spatial Join Scenarios in Optimal Plan Build

Case	Algorithm(s) to Consider
L and R sorted on entity IDs	SMJ (Sect. 6.3)
L and R results of (?s_i hasGeometry ?g_i)	SMJ or R-tree Join [12]
L sorted on entity IDs R result of a pattern (?s_2 hasGeometry ?g_2)	SMJ (Sect. 6.3), SISJ [24], or Index Nested Loops
L unsorted R result of a pattern (?s_2 hasGeometry ?g_2)	SHJ-ID (Sect. 6.4), SISJ [24] or Index Nested Loops
L and R unsorted	SHJ-ID, SHJ [22] or Nested Loops

In addition, the optimizer considers the case of performing a spatial selection Filter using the R-tree (see Sect. 4). The optimizer also considers any spatial selection and join filter conditions that are applied on-the-fly; i.e., in plans where the non-spatial query pattern components are evaluated first, our optimizer uses spatial query selectivity statistics to estimate the output size of these components *after* the spatial filter is applied on them. Consider for example, the plan of Fig. 5. The estimated output of the ?s hosted ?o pattern is further refined to consider the spatial WITHIN filter that follows. In other words, the cardinality of the right input to the merge-join algorithm that follows is estimated using both RDF-3X statistics on the selectivity of ?s hosted ?o and spatial statistics for the selectivity of WITHIN(?g,"POLYGON (...)").

7.4 Selectivity Estimation

For estimating the selectivity of spatial query components, we use grid-based statistics, similar to previous work on spatial query optimization (e.g., see [24]). Specifically, we take advantage of statistics that are obtained by the spatial encoding phase of the entity IDs. For each cell of the grid, defined by the Hilbert order, we keep track of the number of spatial entities that fall inside. The spatial join or selection is then applied at the level of the grid, based on uniformity assumptions about the spatial distributions inside the cells. In addition, we assume independence with respect to the other query components. For example, for estimating the input cardinality of the right merge-join input at the plan of Fig. 5, we multiply the selectivity of the ?s hosted ?o pattern with that of the WITHIN(?g,"POLYGON (...)") filter. In practice, this gives good estimates if the spatial distribution of the entities that instantiate ?s is independent to the spatial distribution of all entities.

7.5 Runtime Optimizations

RDF-3X uses a lightweight Sideways Information Passing (SIP) mechanism for skipping redundant values when scanning the indexes [28]. Consider a merge join, which binds the values of a variable ?s coming from two inputs. If the join result is fed to another (upper) merge join operator that binds ?s, then the upper operator can use the next value v of its other input to notify the lower operator that ?s values less than v need not be computed.

In the case of spatial joins where at least one side comes from a scan in the R-tree (e.g., consider the plan shown in Fig. 3a), SIP is not applicable since there is no global order for the geometries in the 2D space. On the other hand, the SMJ algorithm proposed in Sect. 6.3 can use SIP to notify the operators below its left input which is the minimum ID value for the next entity e_l to pair with any entity buffered in B_R. For the spatial hash join, we can also use SIP, by creating a bloom filter for one input, similar to the one RDF-3X constructs for the traditional hash join, and use it to prune tuples from its other input, while scanning the B$^+$-tree index. A value is pruned if it is not included in the bloom filter.

8 Updates

The proposed encoding scheme requires significant changes in the update mechanism of RDF-3X [27,29]. Inserting a new spatial entity is straight-forward and requires generating the appropriate ID based on the entity's geometry and the occupancy of the grid. On the other hand, removing or updating the geometry of a spatial entity requires additional care as it might trigger the re-encoding of other spatial entities besides the one being updated. Such re-encodings tend to improve the latency of spatial queries, since entities are "moved" to lower levels of the grid, but incur an overhead during updates because they result in additional triples to be removed and re-inserted with new IDs.

Insert and delete commands in RDF-3X (cf. Sect. 2) are given in batches and are processed in two phases. In the first phase, the triples to insert or delete are resolved via lookups in the dictionary, i.e. they are translated into triples of integer IDs used internally by the system. Updates are not applied directly to the database; instead, the affected triples are first resolved in memory for all updates in the batch using *differential indexes*, which are then synchronized with the base indexes in the second phase. This is a common technique in bulk update processing that aims to minimize I/Os and increase the system throughput. RDF-3X maintains six differential indexes (SPO, SOP, OPS, OSP, PSO, POS), one for each full base index, which are synchronized with both the full and the aggregated base indexes. For example, the SPO differential index is synchronized with the base SPO index, the binary aggregated index SP, and the unary aggregated index S.

SRX integrates the original update mechanism of RDF-3X with the encoding scheme of Section 5. For this purpose, it suffices to change only the first update phase, whereas the index synchronization can be used as is. To simplify the presentation, we distinguish two cases: a) only inserts of new triples, i.e., the subject entity s of the input triple $\langle s,p,o\rangle$ does not exist in the dictionary, b) inserts and deletes of triples whose subject entity already exists in the data. The update process takes as input a batch B of triples annotated

with *insert* or *delete*, and updates two in-memory sets of triples tI (to insert) and tD (to delete), which are used to build the differential indexes.

Inserts of new entities. The insertion to the triples set tI is performed similarly to the original RDF-3X update process, but IDs are generated using the modified function GENERATE_ID. GENERATE_ID takes as input the entity's URI and a boolean value, which indicates whether the new ID should be spatial (*true*) or not ($false$), i.e., whether the input triple introduces a geometry for the subject entity. GENERATE_ID is also responsible for updating the dictionary and for tracking the set of *new* IDs for the current batch B. The set *new* (initially empty for a batch) contains all IDs that do not exist in the database and is used in the second part of the update algorithm to avoid expensive lookups in the base indexes, as we explain later on. Since, the insertion of new triples only differs from the original RDF-3X process in the creation of the ID, we omit the pseudocode for the sake of brevity.

Updates on existing entities. The pseudocode for the updates on existing entities is given in Algorithm UPDATES ON EXISTING ENTITIES. This part handles triples with existing spatial and non-spatial subject entities, and is further split into three sub-parts: one for inserting triples that introduce a geometry for the non-spatial subject (*lines* 5-16), one for inserting triples that do not introduce a geometry (*lines* 17-23), and a last one for deleting triples (*lines* 24-34).

In the first sub-part, when a geometry is introduced for a non-spatial entity, the algorithm generates a new spatial ID s_{new} (*line* 8) and proceeds with updating the in-memory sets tI and tD accordingly. To do so, it first checks if the old subject ID (s_{id}) exists in the set of *new* IDs for the current batch; if yes, it simply updates the set *new* along with tI (*lines* 15-16), otherwise it retrieves all affected triples from the database and updates both tI and tD (*lines* 11-14 and 16). The update algorithm also ensures in this case that each entity is associated with *at most one* geometry but these additional checks are omitted here for the sake of brevity.

The last two sub-parts of UPDATES ON EXISTING ENTITIES follow the original RDF-3X update logic and differ only in the use of *new* in *lines* 20 and 28 to avoid expensive lookups in the base indexes; these lookups are only performed as last steps in *lines* 22 and 30.

Spatial re-encoding is the task of re-assigning spatial IDs that get released (after geometry deletions) to spatial entities encoded at higher levels of the grid due to overflow. It is an iterative bottom-up process, from lower to higher levels of the grid, which takes place in *line* 34 of UPDATES ON EXISTING ENTITIES. The re-encoding function receives the spatial ID of an entity whose geometry is being deleted, replaces this ID with a non-spatial one (the next free even ID), and checks if there is a spatial entity from a higher level that can be re-encoded using the recently released spatial ID. If so, the re-encoding of the spatial entity releases another spatial ID, and the process cascades until no more re-encodings are possible.

Algorithm: UPDATES ON EXISTING ENTITIES
Input : Batch B of triples annotated with $op = \{insert, delete\}$
Output : Sets of triples tI (to insert) and tD (to delete)

```
1  let new be the set of new entity IDs for the current batch B;
2  while t = ⟨s,p,o,op⟩ = B.get_next() do
3      if s ∈ dictionary then
4          let s_id be the ID of s as found in the dictionary;
5          if s_id is not spatial and op = insert and p =
           "hasGeometry" then
               //A geometry is given for s
6              let p_id = FETCH_OR_GEN_ID(p, false);
7              let o_id = FETCH_OR_GEN_ID(o, false);
8              let s_new = GENERATE_ID(s, true);
9              tI = tI ∪ {⟨s_new, p_id, o_id⟩};
10             if s_id ∉ new then
11                 let affected be the set of triples in the
                   database containing s_id as subject or object;
12                 let replace = affected \ tD;
13                 tI = tI ∪ replace;
14                 tD = tD ∪ replace;
15             else new = new \ {s_id};
16             change s_id to s_new in all triples of tI;
17         else if op = insert and p ≠ "hasGeometry" then
               //Both for spatial and non spatial s_id
18             let p_id = FETCH_OR_GEN_ID(p, false);
19             let o_id = FETCH_OR_GEN_ID(o, false);
20             if {s_id, p_id, o_id} ∩ new ≠ ∅ then
21                 tI = tI ∪ {⟨s_id, p_id, o_id⟩};
22             else if ⟨s_id, p_id, o_id⟩ ∉ database then
23                 tI = tI ∪ {⟨s_id, p_id, o_id⟩};
24         else if op = delete and (s_id is spatial or p ≠
           "hasGeometry") then
               //For any valid delete operation
25             if p ∈ dictionary and o ∈ dictionary then
26                 let p_id be the ID of p as found in the
                   dictionary;
27                 let o_id be the ID of o as found in the
                   dictionary;
28                 if {s_id, p_id, o_id} ∩ new ≠ ∅ then
29                     tI = tI \ {⟨s_id, p_id, o_id⟩};
30                 else if ⟨s_id, p_id, o_id⟩ ∈ database then
31                     tD = tD ∪ {⟨s_id, p_id, o_id⟩};
32                 else continue;
33                 if s_id is spatial and p = "hasGeometry"
                   then
34                     re-encode spatial entities;
```

The overall process relies on two thresholds h_1, $h_2 \in [0,1]$, which define a range $[h_1, h_2]$ on the ratio of assigned to total spatial IDs that a cell can accommodate (fill factor). In particular, a re-encoding process starts when the fill factor of a cell c drops below h_1, and continues as long as (i) there are entities from higher levels to re-encode in c, and (ii) c's fill factor remains below h_2. By the time at least one of

these two conditions does not hold, the algorithm continues with examining the parent cells of c (at the next level), and so forth, until it reaches the top level. In the end, all spatial entities have been re-encoded at the lowest possible level such that each cell's fill factor is smaller than h_2. The two thresholds h_1 and h_2 are used to trade off the frequency of re-encodings with the overhead in processing updates, and we discuss the choice of their values in Sect. 9.

9 Experimental Evaluation

We compare SRX with the original RDF-3X system [27], the latest version of Strabon [20]; version `3.3.2`, and the latest versions of two commercial triple stores with spatial data support, Virtuoso `7.2.5-rc1.3217-pthreads` [4] and GraphDB Free `8.6` [1] (the successor of OWLIM-SE that we used in [21]). We implemented SRX in C++ (g++ 8.2.0) and conducted all experiments on a machine with an i7-4930K CPU at 3.40 GHz, a 3.6Tb 7.2K rpm SATA-3 hard disk, and 64GB RAM running Linux Debian (4.18.0-1-amd64). For the Strabon database, we used PostgreSQL `11.2` and PostGIS `2.5` versions, while for the R-tree implementation, we used the open-source SaIL library [18], which we also extended with support for incremental kNN computation (cf. Sect. 4).

9.1 Queries Setup

Datasets. We experimentally evaluate our system using two real datasets: LinkedGeoData[8] (LGD) and YAGO2s[9] (YAGO). LGD contains user-contributed content from the OpenStreetMap project. YAGO is an RDF knowledge base, derived from Wikipedia, WordNet and Geonames. Table 2 shows statistics about the sizes of the datasets (including the dictionary and indexes) and the number of entities and geometries in them. The sizes of the input triple files are 2.4GB (LGD) and 18.1GB (YAGO). The R-trees (using 4KB nodes) occupy 160MB and 221MB, respectively. The size of the grid in both datasets is 1.5GB (89M cells in total for all levels). Note that, despite the aggressive indexing, in both cases, we end up with a database size having around double the size of the input files. Regarding the spatial distribution of the entities they include, both datasets are highly skewed (i.e., the density of the data is high in populated areas), as shown in the supplementary material of this paper (Online Resource 1); this is also reflected by the percentage of geometries that reside at the different levels of our encoding scheme (see Table 3). YAGO includes a significant percentage of geometries (multipoints) which have not been cleaned and span a very large area; this explains the increased percentage of geometries encoded at high levels of the grid hierarchy.

[8] `https://tinyurl.com/yc4lxqdv`
[9] `https://tinyurl.com/y7ukhge3`

Table 2: Characteristics of the real datasets

Dataset	Triples	Entities	Points	Polygons	Linestrings	Multipoints
LGD (5.1 GB)	15.4M	10.6M	590K	264K	2.6M	0
YAGO (29.8 GB)	205.3M	108.5M	4M	0	0	780K

Table 3: Percentage (%) of geometries per grid level

Level	0 (bottom)	1	2	3	4	5	6	≥ 7
LGD	28.5	21.6	16.9	12.7	8.7	5.4	3.0	3.2
YAGO	50.3	19.2	8.1	4.5	3.0	2.4	1.9	10.6

Encoding. We used a grid of 8,192 $\times$ 8,192 cells at the bottom level, hence, the maximum number of bits used in an entity's ID to encode its cell-ID is 26. This means that we can have up to 14 levels of spatial approximation. This is the maximum granularity we can achieve when the IDs of the entities are 32-bit integers. Using 64-bit IDs for better spatial approximation is possible, but significantly increases the size of the triple indexes, thus, one should do this only when the total number of entities is greater than 2^{32}. Besides, the grid must be relatively small so that it resides in memory (for selectivity estimation purposes). In our case, the grid size is less than 2GB for both datasets. As shown in Table 3, all levels of the grid are used in the encoding, because some of the extended geometries (polygons, linestrings, multipoints) span the borders of quadrants at high levels of the grid and some multipoints in YAGO have very large MBRs.

Queries. All queries used in our experiments have two parts: (i) an RDF part that can be evaluated by a traditional SPARQL engine and (ii) a spatial part, i.e., a FILTER condition that includes a WITHIN predicate (for spatial range queries), a DISTANCE predicate (for spatial distance joins) or a kNN predicate (for spatial nearest neighbors). The range queries have similar structure to those depicted in Fig. 1b; we divide them into four classes based on the selectivities of the two parts. Queries belonging to class SL have their RDF part more selective compared to their spatial part and the opposite holds for queries in class LS (S stands for small result, L for large). For queries in classes SS and LL, both parts roughly have the same selectivity. The characteristics of the spatial join queries (denoted by J) and the spatial kNN queries will be discussed in Sect. 9.2. All query expressions can be found in the supplementary material (Online Resource 1).

Comparison Measures. We evaluated each query 5 times (both with cold and warm cache) and report their average response times. The reported runtimes include the query optimization cost (i.e., the time spent by the optimizer to apply the techniques of Sect. 7) and the time spent in the ID-to-string dictionary lookups for the output variables.

Table 4: Spatial range queries on LGD (total response time in *ms* - optimizer time in parentheses)

Query	Number of results			Strabon		GraphDB		Virtuoso		SRX					
										Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding	
	RDF	Spatial	Combined	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
LGD.SL1	524	2,537,757	411	64,581	12,791	6,960	136	74,645	83	3,880 (99)	35 (1)	3,333 (148)	54 (20)	2,334 (107)	78 (1)
LGD.SL2	215,355	2,943,209	186,302	168,099	20,129	20,957	14,045	78,568	76	4,495 (97)	272 (1)	3,644 (178)	330 (58)	3,009 (100)	323 (1)
LGD.SL3*	13,090	2,537,757	9,814	160,591	18,081	8,496	822	72,624	36	11,554 (97)	82 (1)	9,891 (147)	104 (20)	3,983 (100)	177 (1)
LGD.LS1	25,617	9,002	86	63,614	12,866	9,568	2,338	58,757	81	1,913 (95)	84 (1)	856 (124)	62 (15)	222 (109)	14 (1)
LGD.LS2	191,976	908	3	63,065	12,785	24,702	17,159	46,066	167	2,257 (97)	176 (2)	429 (117)	21 (15)	183 (91)	12 (2)
LGD.LS3*	5,791	908	9	63,251	12,694	16,065	410	45,317	183	14,305 (107)	52 (1)	484 (127)	20 (14)	174 (89)	2 (1)
LGD.SS1	8,621	9,002	69	63,403	13,271	8,118	829	58,658	51	1,696 (98)	65 (1)	932 (131)	63 (16)	211 (92)	14 (1)
LGD.SS2*	13,090	9,002	120	66,370	12,708	8,488	949	57,745	35	8,181 (97)	75 (1)	1,167 (137)	61 (15)	450 (100)	5 (1)
LGD.SS3*	5,791	9,002	7	66,275	12,750	16,479	410	57,251	22	14,308 (107)	53 (1)	934 (142)	53 (16)	350 (100)	3 (1)
LGD.LL1	191,976	350,405	13,416	89,097	13,900	24,868	17,178	53,714	120	2,694 (95)	183 (1)	2,562 (142)	200 (17)	815 (117)	55 (2)

Table 5: Spatial range queries on YAGO (total response time in *ms* - optimizer time in parentheses)

Query	Number of results			GraphDB		Virtuoso		SRX					
								Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding	
	RDF	Spatial	Combined	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
YAGO.SL1*	11,547	364,992	891	23,689	2,823	66,017	9,053	10,610 (61)	61 (1)	10,342 (62)	61 (1)	6,119 (46)	48 (1)
YAGO.SL2*	6,030	31,260	69	32,091	1,816	59,802	1,318	8,984 (173)	70 (1)	8,654 (175)	70 (1)	3,783 (168)	43 (1)
YAGO.LS1*	2,226	138	0	4,817	232	19,276	19	2,769 (60)	61 (1)	2,661 (79)	64 (15)	772 (56)	37 (1)
YAGO.LS2*	285,613	41,945	4,471	182,118	9,657	77,759	1,274	17,590 (183)	696 (1)	16,471 (178)	697 (1)	8,692 (181)	161 (1)
YAGO.SS1*	6,030	8,440	3	30,249	1,853	51,155	459	8,375 (175)	70 (1)	12,019 (173)	275 (17)	3,201 (170)	41 (1)
YAGO.SS2*	7,074	7,042	2	11,131	547	36,008	325	4,641 (61)	62 (1)	11,286 (85)	245 (15)	2,319 (56)	46 (1)
YAGO.LL1*	285,613	184,743	10,454	188,015	13,465	66,381	5,040	19,882 (184)	701 (1)	18,688 (182)	699 (1)	10,978 (179)	173 (1)
YAGO.LL2*	152,693	107,625	88	82,420	10,880	55,230	3,123	6,262 (61)	2,971 (1)	5,999 (62)	2,993 (1)	5,302 (56)	1,921 (1)

System Parameters. RDF-3X does not have its own data cache for the query results; instead, it relies entirely on the OS caching mechanism. The same architectural principle is also adopted in our implementation.[10] When a query is executed for a second time, its optimization and evaluation is performed from scratch, since there are no logs or cached results as in a typical database system. To illustrate the effect of OS caching in the overall response time of the system, we report query evaluation times on warm and cold caches separately.

9.2 Queries Comparison

Results on Range Queries. Table 4 shows response times for range queries on the LGD dataset. The first three columns of the table show the number of results of the RDF query component only, the spatial component only and the complete query (combined). We first focus on comparing our approach (Encoding) with the basic extension presented in Sect. 4 (Basic) and the original RDF-3X system (Baseline). Basic uses the R-tree to retrieve the entities falling in the given range only for queries where the spatial component is selective enough (LS, SS); in all other cases, it applies the same plan as Baseline, i.e. it evaluates the RDF part first and then applies the WITHIN filter to the tuples that qualify it. On the other hand, Encoding always chooses to evaluate the RDF part of the queries first and uses the spatial range filtering technique (see Sect. 6.1) to reduce the number of entities that have to be spatially verified. Encoding is superior in all queries. In specific, we avoid fetching a large percentage of exact geometries (96% on average for all range queries in both datasets), which Baseline obtains by random accesses to the dictionary. The cost differences between Encoding and Baseline is small only for SL queries, where the spatial filtering has little effect. In all other cases, Encoding is significantly faster than Baseline and Basic, especially in LS and SS queries and in queries involving entities with non-point geometries, denoted by a star (*), where the difference is up to one order of magnitude. In the case of warm caches, all runtimes are very low, so the cost of Encoding may exceed the cost of Baseline sometimes (e.g., see SL queries) due to the overhead of applying the spatial filter on all accessed entities in the evaluation of the RDF part of the query.

The difference in the optimization times (in parentheses) between warm and cold caches in all alternatives is because the reported numbers include the time spent for parsing the query, resolving the IDs of the URIs/strings in it, and finally building the optimal plan. Hence, when a query is issued for the first time, it requires some dictionary lookups for resolving the IDs of the entities. With warm caches, the respective dictionary pages are already cached by the OS, thus, query optimization is always cheaper. Note that, in most cases, the time spent for query optimization by Encoding is similar to that of Baseline, meaning that the overhead of augmenting the query graph and using spatial statistics is negligible compared to the query optimization overhead of the original RDF-3X system. With warm caches, the overhead in query optimization by Encoding and Basic compared to Baseline (due to the use of spatial statistics) is more profound.

Similar results are observed for range queries on the YAGO dataset (see Table 5). All queries in this case involve entities that may have multipoint geometries (therefore they are marked by a star). Encoding always chooses to evaluate

[10] We only included a small separate cache of 40Kb for the R-tree. Since the OS caches R-tree pages, we used a small cache size in order to reduce the effect of double caching by the SaIL library.

the RDF part of the queries first, as in LGD. Basic chooses the same plan as Baseline in all cases, except for LS1 and SS queries, where it opts to evaluate the spatial selection using the R-tree; on the other hand, the spatial part of LS2 is not considered very selective by the optimizer, preventing the use of R-tree for that query. Note that for YAGO the cost of Basic is high enough (even higher than Baseline for SS queries). After analysis, we found that this is due to the bad performance of the R-tree on this dataset; the range queries access roughly half of the R-tree nodes. The reason is that many multipoints in YAGO are dirty and have huge MBRs that cover most of the data space. Thus, the non-leaf R-tree entries have extremely large MBRs, causing a random query to access a large percentage of tree nodes.

Overall, with cold caches, the median speedup of Encoding over Baseline (resp. Basic) across all queries is 8.3× (resp. 2.6×) for LGD, and 2× (resp. 2×) for YAGO. When warm caches are used, the median speedups are 5.3× (resp. 4×) for LGD, and 1.6× (resp. 2.8×) for YAGO.

Results on Spatial Joins. Table 6 and Table 7 show the costs of spatial distance join queries on LGD and YAGO, respectively. The threshold 0.1 shown in the tables corresponds to a distance around 10km. In LGD, all queries have thresholds greater than the diagonal of a cell in our encoding except queries LGD.J6.1 and LGD.J6.2. In YAGO, threshold 0.1 is greater than the cell diagonal, but 0.01 is not. After performing experiments with various types of queries, we found that the SMJ and SHJ-ID algorithms should only be used when the spatial distance threshold is greater than the diagonal of the grid cell at the bottom level. Otherwise, they do not produce any verified results and, hence, they have similar or slightly worse performance compared to directly applying SHJ (as Basic would). We have added this simple rule of thumb in the optimizer of our system, hence, in all spatial join queries that have a distance threshold less than the cell diagonal, Encoding applies the same plans as Basic using the SHJ operator. In LGD, these queries are J6.1 and J6.2, while in YAGO, the respective queries are J8.1 and J8.2. For this reason, we focus mostly on queries where the distance threshold is greater than the cell diagonal.

All spatial join queries on the LGD dataset (Table 6) have a similar pattern: they include two disjoint RDF star-shaped parts with a spatial distance predicate between the geometries of their center nodes. This is the only type of queries we could define here since the LGD dataset includes a rather poor RDF part; besides the POI type, there are very few properties such as “label” and “name” which link the POIs with text attributes. For this type of queries, Baseline can only execute a bushy plan where the two stars are evaluated separately and then joined in a nested-loop fashion, applying the spatial distance filter. On the other hand, Basic may choose to apply an R-tree join first for retrieving the candidate pairs within distance ε or to first evaluate the RDF part of the query and follow-up with a spatial hash join (SHJ) in the end (e.g., see the plans of Fig. 3c and Fig. 3d). In all queries we tested, Basic chose the SHJ algorithm and this is quite reasonable; in large datasets, the optimizer would prefer not to perform an expensive spatial self-join over the whole set of points. Encoding can choose between one of the previously mentioned methods and also try the algorithms of Sect. 6.3 and Sect. 6.4 on the augmented query graph. Since we have star-shaped queries and the IDs of the center nodes are coming sorted, SMJ was favored in all queries we present. Although Encoding is much faster than Baseline, we observe that for the case of warm caches, the former does not bring much benefit over Basic for most join queries on LGD. The main reason is that Encoding does not save any geometry lookups due to the particular data distribution; every entity from either of the two spatial join inputs participates in at least one non-verified spatial join pair and therefore it cannot be pruned without fetching its geometry. In addition, Basic benefits from the fact that it buffers the complete join inputs before hashing them into the buckets of SHJ, thus, the geometry of each entity is processed only once. On the other hand, SMJ (used by Encoding) produces and verifies the join pair candidates on-the-fly, resulting in the processing of a given geometry multiple times. However, if the size of inputs is very large, Basic can become significantly slower than Encoding (see LGD.J4) because SHJ requires the allocation of a large hash table to accommodate a huge number of buffered geometries. Recall that SHJ, used by Basic, is a blocking operator which requires both inputs to be read as a whole before processing the join and, thus, can become a bottleneck if the inputs are too large.

The results for queries with spatial join components in YAGO are shown in Table 7. Depending on the type of the query and the selectivities of the two parts, our encoding-based approach uses either SMJ or SHJ-ID. Specifically, SMJ is used in queries J1 and J8.3, whereas SHJ-ID is used in J2, J6, and J7. In queries J8.1 and J8.2, Encoding follows the same plans as Basic. In the remaining queries (J3, J4 and J5), Basic and our encoding-based approach produced the same plans as Baseline; these queries include a single connected RDF graph pattern with a rather selective RDF part. As a result, the performance of Basic is similar to that of Encoding for queries J3, J4, J5, J8.1, and J8.2. For the remaining queries, Encoding is slightly faster than Basic with cold caches (the two techniques are comparable for warm caches), because Encoding selects a different plan based on the augmented query graph. A notable exception is YAGO.J7, where Encoding performs much better than Basic, because the spatial join inputs have a different spatial distribution and Encoding can prune many tuples using SHJ-ID.

Note that, for large inputs, SHJ might be slower than SMJ. This is reflected in the performance difference between queries LGD.J6.1, LGD.J6.2 (resp. YAGO.J8.1, YAGO.J8.2),

Table 6: Spatial distance join queries on LGD (total response time in *ms* - optimizer time in parentheses)

Query	Spatial join threshold ε	Number of results		Strabon		GraphDB		SRX					
								Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding	
		RDF	Final	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
LGD.J1	0.003	12,145,200	6,831	> 5 min	> 5 min	> 5 min	> 5 min	111,208 (123)	109,295 (1)	3,936 (280)	188 (128)	2,854 (246)	566 (130)
LGD.J2	0.01	274,576	538	> 5 min	27,194	> 5 min	101,756	21,796 (133)	18,476 (2)	4,497 (326)	256 (188)	2,229 (314)	273 (196)
LGD.J3	0.02	13,423,300	8,742	> 5 min	> 5 min	> 5 min	> 5 min	> 5 min	> 5 min	5,412 (417)	433 (267)	3,952 (401)	712 (275)
LGD.J4*	0.05	171,348,000	795,322	> 5 min	> 5 min	> 5 min	> 5 min	> 5 min	> 5 min	106,856 (613)	97,426 (474)	30,841 (589)	21,593 (475)
LGD.J5*	0.01	15,564,000	2,782	> 5 min	> 5 min	> 5 min	> 5 min	59,468 (142)	49,528 (2)	13,530 (334)	355 (186)	10,976 (310)	1,161 (189)
LGD.J6.1*	0.0005	20,181,600	7	> 5 min	> 5 min	> 5 min	> 5 min	133,685 (141)	119,677 (2)	15,291 (243)	199 (88)	12,943 (208)	204 (91)
LGD.J6.2*	0.001	20,181,600	22	> 5 min	> 5 min	> 5 min	> 5 min	133,693 (139)	120,059 (2)	15,372 (240)	195 (89)	12,970 (208)	198 (91)
LGD.J6.3*	0.01	20,181,600	743	> 5 min	> 5 min	> 5 min	> 5 min	134,433 (137)	119,668 (2)	17,036 (341)	308 (187)	8,533 (297)	1,497 (189)

Table 7: Spatial distance join queries on YAGO (total response time in *ms* - optimizer time in parentheses)

Query	Spatial join threshold ε	Number of results		GraphDB		SRX					
						Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding	
		RDF	Final	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
YAGO.J1*	0.1	6,245,000	2,635	-	-	118,992 (44)	103,626 (1)	14,738 (204)	445 (129)	11,795 (199)	684 (130)
YAGO.J2*	0.1	523,815,000	6,799,189	> 5 min	> 5 min	> 5 min	> 5 min	137,147 (205)	112,954 (129)	136,114 (275)	115,226 (204)
YAGO.J3*	0.1	16,528	832	142,545	9,732	8,994 (57)	162 (1)	10,547 (220)	264 (129)	9,425 (203)	279 (131)
YAGO.J4*	0.1	3,165	451	69,923	1,477	7,990 (52)	124 (1)	8,796 (209)	219 (128)	8,139 (204)	228 (130)
YAGO.J5*	0.1	565	113	-	-	2,836 (48)	95 (1)	3,547 (199)	164 (129)	3,406 (194)	168 (130)
YAGO.J6*	0.1	19,814,600	664,613	> 5 min	> 5 min	> 5 min	292,119 (2)	45,004 (375)	21,696 (130)	43,679 (356)	22,945 (205)
YAGO.J7*	0.1	544,771,000	4,204,184	> 5 min	> 5 min	> 5 min	> 5 min	40,454 (195)	21,169 (129)	16,721 (267)	1,831 (201)
YAGO.J8.1*	0.001	3,519,380	85,188	-	-	86,075 (48)	77,085 (1)	8,889 (185)	150 (115)	8,381 (189)	150 (115)
YAGO.J8.2*	0.01	3,519,380	86,222	-	-	87,701 (48)	77,093 (1)	9,117 (190)	201 (115)	8,535 (186)	202 (115)
YAGO.J8.3*	0.1	3,519,380	131,828	-	-	86,068 (48)	77,152 (1)	9,471 (205)	383 (128)	8,066 (206)	461 (130)

where SRX applies the same plan as Basic using SHJ, and queries LGD.J6.3 (resp. YAGO.J8.3), where SMJ is used.

Overall, with cold caches, the median speedup of Encoding over Baseline across all queries we used is 10.3× and 8.4× for LGD and YAGO respectively. When warm caches are used, the respective median speedup is 136.5× for LGD and 82.1× for YAGO. Regarding spatial distance joins, Encoding and Basic have similar median performance for YAGO. For LGD with cold caches, Encoding has a median 1.3 speedup over Basic, whereas with warm caches Encoding shows a median 0.7 slowdown over Basic.

Results on Spatial kNN Queries. All kNN queries in Tables 8-17 have been constructed from (and have the same names with) the range queries used in Tables 4-5. In particular, each kNN query has the same RDF part with the respective range query along with a Filter kNN (?g, "POINT (...)") clause, where "POINT (...)" is the middle point of "RECTANGLE (...)" in the Filter WITHIN(?g, "RECTANGLE (...)") clause of the range query. Tables 8-17 provide results for $k = 5, 10, 20, 50$ and 100.

In both datasets, Basic follows the same plan as Baseline with only exceptions LGD.SL1 and YAGO.LS1, where Basic uses the R-tree. In these two queries, Basic employs the incremental kNN operator (cf. Sect. 4) followed by hash joins, which are favored by the optimizer because the RDF part of the query is selective enough and, as a result, the cost of the building phase of the respective hash tables is low. Such a plan achieves much better performance over Baseline in some cases (e.g. for LGD.SL1 and $k = 5$ with cold caches), however, our experiments demonstrate that Baseline is usually a better option than Basic, especially for $k >$ 20. We attribute this behavior to the following reasons: (i) Baseline evaluates the RDF part of the query by leveraging efficient sequential scans over the compressed B$^+$-trees in combination with very fast merge joins, and (ii) LGD.SL1 and YAGO.LS1 have very selective RDF parts (the most selective for each dataset), therefore, the cost of the final dictionary lookups to fetch the geometries of the entities, as performed by Baseline, is low.

When cold caches are used, Unsorted and Sorted are significantly faster (due to filtering and fewer geometry lookups) than Baseline for all kNN queries except YAGO.LL2; in this case, Baseline is superior to both Unsorted and Sorted but for different reasons. First, Unsorted and Baseline use exactly the same plan to evaluate the RDF part of the query, and their only difference lies in the final verification phase where they must retrieve the actual geometries from the dictionary. We found that the result set of the RDF part of YAGO.LL2 contains many tuples carrying the same geometry, which amplifies the effects of caching and significantly reduces the cost of dictionary lookups. As a result, Baseline outperforms Unsorted due to the CPU overhead of the latter to maintain its priority queues. On the other hand, when Sorted is an option, the optimizer chooses a different plan for the RDF part, which leverages merge joins and outputs tuples in ascending order of their spatial IDs but shows poor performance.

When warm caches are used, Unsorted performs better than Baseline for all kNN queries on LGD except LGD.SL1. This is reasonable since the Unsorted method tends to perform fewer dictionary lookups than Baseline and, in queries with very selective RDF part, such as LGD.SL1, the benefits of Unsorted over Baseline are negligible. In YAGO with warm caches, Unsorted performs similarly or slightly worse (e.g. for YAGO.LS2, YAGO.LL1, and YAGO.LL2) than Baseline for the same reason we mentioned before: many en-

tities in the result of the RDF part of these queries have the same geometry, and the cost of the additional geometry lookups in Baseline is mitigated by caching. The only LGD queries where Baseline is slightly better than Sorted are LGD.SL1 for $k = 100$ and LGD.SS3 for all k values except $k = 5$. Note that, although LGD.LS3 has the same RDF part with LGD.SS3, Sorted is constantly better than Baseline for LGD.LS3 because its kNN predicate is different. Regarding YAGO, Sorted is better than Baseline in all queries apart from YAGO.LS1 for $k = 100$.

Finally, Sorted is superior to Unsorted in most cases. For LGD and warm caches, the only queries where Unsorted is better than Sorted are LGD.LS3, LGD.SS1 for $k = 100$, and LGD.SS3 for all k values except $k = 5$. Still, under cold caches, Sorted is better even in these cases. For YAGO, Sorted is better than Unsorted in all queries but YAGO.LS1 for $k = 100$, where Unsorted is superior under both warm and cold caches. As a general comment, Sorted is preferable over Unsorted and Baseline, as it tends to avoid a significant number of geometry retrievals. Yet, Unsorted is a good alternative since it can overcome the deficiencies of Sorted in the few cases where Baseline is better: LGD.SS3 for all k values except $k = 5$, and YAGO.LS1 for $k = 100$.

Overall, for LGD and with cold caches, Encoding Unsorted (resp. Sorted) has a median 2.9× (resp. 7.9×) speedup over Baseline whereas, for YAGO, the median speedup over Baseline is 2.2× for Unsorted and 2.3× for Sorted. For LGD queries with warm caches, the median speedups of Unsorted and Sorted over Baseline are 1.1× and 1.5× respectively. Last, for YAGO with warm caches, Unsorted and Baseline perform similarly whereas Sorted has a median 1.3× speedup over Baseline.

Comparison with Existing Systems. We compared SRX against three popular RDF stores with geospatial data support, namely Strabon, GraphDB and Virtuoso. For Strabon, we only present results with LGD, as it could not load YAGO even after three days (and even when using the bulk loader we obtained from the authors of [20]; this issue is also reported in [33]).

We allowed each system to allocate the whole available memory of the machine and performed the experiments with cold and warm caches just like for our system. Since these systems have their own data caches, experiments with cold caches were conducted by clearing the OS cache and restarting the system. The symbol '-' in Tables 7, and 13-17 denotes empty or incorrect results. N/A (not applicable) is used in Tables 8-12, and 18 for Virtuoso because its internal function `bif:st_distance`, which is used in spatial join and kNN queries (cf. Appendix), works only for point geometries. In fact, this is why there are no Virtuoso results in Tables 6-7 and Tables 13-17. Although queries J1, J2, and J3 in Table 6 involve only points, they are also not supported by Virtuoso, presumably because it tries to apply the DISTANCE predicate first between all types of geometries. Note that both GraphDB and Virtuoso compute the great-circle distance, whereas Strabon, Encoding, Baseline, and Basic compute the Euclidean distance. This does not prevent us from a fair comparison among GraphDB and Virtuoso since both distance functions have similar CPU cost.

Table 18 summarizes the median speedups of SRX over Strabon, GraphDB, and Virtuoso across all queries we used on both datasets. SRX is 8.5× faster (in the worst case) than the competitors in all cases except for the case of Virtuoso on LGD with warm caches, where SRX median speedups are lower. We cannot comment further on the performance of GraphDB and Virtuoso as they are not open-source systems. On the contrary, the performance of Strabon for range (Table 4), and kNN (Tables 8-12) queries is dominated by the time needed to fetch data from disk: 60s (resp. 12s) with cold (resp. warm) caches on average in both range and kNN queries. We also observe that for these types of queries, the query time tends to increase with the size of the result for the RDF part of the query. Finally, distance join query times (Table 6) in Strabon are dominated by the CPU rather than the I/O time due to the large number of candidate pairs that need to be examined.

9.3 Updates Setup

In the experiments for updates we used the same encoding and R-tree configurations as for queries (cf. Sect. 9.1). Thresholds h_1 and h_2 used to trigger re-encodings (cf. Sect. 8) were set to 0.5 and 0.7 respectively. We also experimented with other thresholds (e.g. 0.35 and 0.5) but we did not notice any significant performance difference.

Datasets. The update benchmark relies on Deltas that we are extracted by calculating the difference of two versions of LGD and YAGO: LGD `2013_04_29` to `2015_11_02`[11], and YAGO `2.5.3` to `3.0.2`[12]. Details are presented in the supplementary material (Online Resource 1). In both datasets, each delta consists of a delete, an update, and an insert workload, all measured in number of triples. The column 'HasGeo' shows the number of $\langle s, p, o \rangle$ triples with $p =$ "hasGeometry" whereas the numbers for the rest of the triples are given in column 'Other'. For the YAGO dataset, we encountered a small number of 'HasGeo' deltas between versions `2.5.3` and `3.0.2`; thus, only for this type of triples, we extracted the deltas using the last version of YAGO (`3.1`) instead of `3.0.2`. YAGO `3.1` contains many more 'Other' triples we did not consider here since they are not related to any spatial entities and, hence, they do not affect the performance of re-encoding. Finally, LGD and YAGO deltas have been extracted using only types of triples that appear in both the initial and final versions.

[11] `https://tinyurl.com/ydbscsxf`
[12] `https://tinyurl.com/y7ukhge3`

Table 8: Spatial kNN queries on LGD for $k = 5$ (total response time in *ms* - optimizer time in parentheses)

Query	Number of results (RDF)	Strabon		GraphDB		Virtuoso		SRX							
								Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding (Unsorted)		Encoding (Sorted)	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
LGD.SL1	524	64,581	12,968	7,336	78	108,284	10	1,173 (86)	33 (1)	548 (121)	47 (12)	466 (69)	57 (1)	380 (89)	18 (1)
LGD.SL2	215,355	95,637	44,738	23,353	16,527	91,761	1,001	1,345 (77)	208 (1)	1,342 (76)	208 (1)	449 (69)	100 (1)	177 (79)	8 (1)
LGD.SL3*	13,090	69,479	15,453	9,070	1,289	N/A	N/A	7,122 (85)	77 (1)	7,103 (83)	77 (1)	701 (69)	65 (1)	564 (83)	31 (1)
LGD.LS1	25,617	68,845	16,722	10,099	2,682	98,572	85	1,296 (77)	87 (1)	1,283 (76)	87 (1)	525 (70)	74 (1)	426 (84)	60 (1)
LGD.LS2	191,976	93,550	41,670	26,857	20,171	95,581	325	1,542 (77)	195 (2)	1,541 (77)	191 (1)	530 (70)	102 (1)	185 (73)	25 (1)
LGD.LS3*	5,791	75,161	14,249	16,138	512	N/A	N/A	12,483 (83)	53 (1)	12,471 (79)	53 (1)	464 (69)	48 (1)	188 (73)	12 (1)
LGD.SS1	8,621	66,576	14,342	8,461	941	99,445	33	1,233 (88)	66 (1)	1,266 (87)	66 (1)	502 (70)	63 (2)	487 (90)	50 (1)
LGD.SS2*	13,090	69,496	15,540	9,015	1,248	N/A	N/A	7,174 (87)	77 (1)	7,122 (87)	77 (1)	599 (69)	65 (1)	511 (73)	52 (1)
LGD.SS3*	5,791	75,341	14,061	16,288	519	N/A	N/A	12,375 (87)	53 (1)	12,509 (87)	53 (1)	525 (68)	48 (1)	447 (82)	40 (1)
LGD.LL1	191,976	94,509	42,433	26,931	20,142	97,280	321	1,570 (77)	192 (1)	1,608 (76)	192 (1)	546 (68)	101 (1)	235 (72)	18 (1)

Table 9: Spatial kNN queries on LGD for $k = 10$ (total response time in *ms* - optimizer time in parentheses)

Query	Number of results (RDF)	Strabon		GraphDB		Virtuoso		SRX							
								Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding (Unsorted)		Encoding (Sorted)	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
LGD.SL1	524	64,720	12,941	7,340	80	108,416	12	1,174 (88)	33 (1)	876 (121)	72 (12)	504 (70)	57 (2)	504 (90)	27 (1)
LGD.SL2	215,355	95,608	45,125	23,354	16,536	92,213	1,001	1,327 (77)	208 (1)	1,329 (76)	212 (1)	452 (70)	100 (1)	176 (77)	8 (1)
LGD.SL3*	13,090	69,437	15,487	9,149	1,292	N/A	N/A	7,140 (87)	77 (1)	7,115 (74)	77 (1)	749 (69)	65 (1)	612 (92)	31 (1)
LGD.LS1	25,617	68,666	16,665	10,118	2,711	98,710	85	1,306 (77)	87 (1)	1,273 (76)	87 (1)	509 (70)	74 (1)	470 (84)	60 (1)
LGD.LS2	191,976	93,902	42,130	27,158	20,179	95,801	325	1,577 (77)	192 (1)	1,588 (77)	192 (1)	525 (70)	102 (1)	184 (73)	25 (1)
LGD.LS3*	5,791	75,219	14,073	16,200	519	N/A	N/A	12,483 (87)	53 (1)	12,461 (87)	53 (1)	485 (69)	49 (1)	223 (70)	23 (1)
LGD.SS1	8,621	66,535	14,388	8,473	942	99,528	33	1,217 (87)	66 (1)	1,240 (87)	66 (1)	492 (69)	62 (1)	472 (94)	50 (1)
LGD.SS2*	13,090	69,405	15,523	9,082	1,288	N/A	N/A	7,178 (87)	77 (1)	7,055 (87)	77 (1)	618 (73)	65 (1)	543 (92)	52 (1)
LGD.SS3*	5,791	75,315	13,970	16,302	544	N/A	N/A	12,421 (87)	54 (1)	12,475 (87)	53 (1)	667 (69)	49 (1)	575 (95)	56 (1)
LGD.LL1	191,976	94,205	42,670	27,043	20,148	97,809	322	1,620 (78)	192 (1)	1,634 (77)	192 (1)	544 (73)	101 (1)	252 (73)	19 (1)

Table 10: Spatial kNN queries on LGD for $k = 20$ (total response time in *ms* - optimizer time in parentheses)

Query	Number of results (RDF)	Strabon		GraphDB		Virtuoso		SRX							
								Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding (Unsorted)		Encoding (Sorted)	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
LGD.SL1	524	64,510	12,954	7,376	87	108,909	12	1,133 (88)	33 (1)	1,269 (121)	116 (12)	545 (70)	56 (1)	532 (92)	28 (1)
LGD.SL2	215,355	95,644	45,126	23,356	16,540	93,970	1,004	1,363 (79)	209 (1)	1,330 (77)	209 (1)	453 (69)	100 (1)	178 (75)	8 (1)
LGD.SL3*	13,090	69,274	15,584	9,150	1,306	N/A	N/A	7,164 (87)	77 (1)	7,168 (82)	77 (1)	750 (69)	65 (1)	608 (92)	30 (1)
LGD.LS1	25,617	68,805	16,686	10,214	2,725	99,406	85	1,302 (77)	88 (1)	1,281 (77)	87 (1)	519 (70)	74 (1)	472 (92)	60 (1)
LGD.LS2	191,976	93,585	41,657	27,164	20,180	95,925	327	1,598 (77)	192 (1)	1,615 (77)	192 (1)	528 (70)	102 (1)	182 (72)	26 (1)
LGD.LS3*	5,791	75,127	14,025	16,284	546	N/A	N/A	12,437 (87)	54 (1)	12,434 (86)	54 (1)	542 (69)	48 (1)	312 (78)	34 (1)
LGD.SS1	8,621	66,618	14,380	8,533	950	100,382	34	1,223 (88)	66 (1)	1,204 (86)	65 (1)	515 (70)	62 (1)	515 (94)	50 (1)
LGD.SS2*	13,090	69,536	15,699	9,094	1,300	N/A	N/A	7,170 (87)	77 (1)	7,149 (86)	77 (1)	608 (69)	65 (1)	551 (88)	52 (1)
LGD.SS3*	5,791	75,403	14,025	16,306	545	N/A	N/A	12,457 (87)	54 (1)	12,467 (87)	54 (1)	667 (69)	48 (1)	560 (83)	55 (1)
LGD.LL1	191,976	94,252	42,573	27,223	20,150	98,285	322	1,590 (77)	192 (1)	1,646 (77)	192 (1)	588 (69)	101 (1)	266 (72)	19 (1)

Table 11: Spatial kNN queries on LGD for $k = 50$ (total response time in *ms* - optimizer time in parentheses)

Query	Number of results (RDF)	Strabon		GraphDB		Virtuoso		SRX							
								Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding (Unsorted)		Encoding (Sorted)	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
LGD.SL1	524	64,803	13,097	7,399	110	110,023	13	1,139 (85)	33 (1)	3,686 (129)	265 (12)	668 (70)	56 (1)	653 (92)	32 (1)
LGD.SL2	215,355	95,515	44,992	23,359	16,584	94,177	1,005	1,335 (77)	208 (1)	1,324 (76)	210 (1)	459 (69)	100 (1)	177 (76)	9 (1)
LGD.SL3*	13,090	69,473	15,580	9,298	1,399	N/A	N/A	7,096 (87)	77 (1)	7,091 (87)	77 (1)	883 (69)	65 (1)	851 (94)	33 (1)
LGD.LS1	25,617	68,860	16,667	10,255	2,729	99,767	86	1,315 (77)	87 (1)	1,304 (77)	88 (2)	564 (71)	75 (1)	501 (84)	61 (1)
LGD.LS2	191,976	93,740	41,659	27,320	20,257	96,761	328	1,611 (77)	193 (2)	1,639 (78)	191 (1)	528 (69)	102 (1)	223 (82)	27 (1)
LGD.LS3*	5,791	75,280	13,946	16,333	592	N/A	N/A	12,423 (87)	54 (1)	12,463 (87)	54 (1)	617 (69)	49 (1)	388 (82)	41 (1)
LGD.SS1	8,621	66,514	14,341	8,556	973	100,768	34	1,222 (87)	66 (1)	1,159 (87)	65 (1)	534 (70)	62 (1)	523 (91)	50 (1)
LGD.SS2*	13,090	69,224	15,513	9,269	1,367	N/A	N/A	7,163 (87)	77 (1)	7,170 (86)	77 (1)	688 (69)	65 (1)	629 (85)	52 (1)
LGD.SS3*	5,791	75,301	14,037	16,517	574	N/A	N/A	12,479 (87)	54 (1)	12,351 (87)	54 (1)	791 (69)	48 (1)	752 (82)	60 (1)
LGD.LL1	191,976	93,971	42,339	27,360	20,280	98,525	323	1,633 (77)	192 (1)	1,573 (77)	192 (1)	638 (70)	101 (1)	303 (80)	22 (2)

Table 12: Spatial kNN queries on LGD for $k = 100$ (total response time in *ms* - optimizer time in parentheses)

Query	Number of results (RDF)	Strabon		GraphDB		Virtuoso		SRX							
								Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding (Unsorted)		Encoding (Sorted)	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
LGD.SL1	524	64,623	13,252	7,644	116	110,171	13	1,118 (87)	33 (1)	7,683 (127)	506 (12)	739 (70)	57 (1)	751 (92)	38 (1)
LGD.SL2	215,355	95,492	44,878	23,361	16,676	94,825	1,014	1,324 (77)	208 (1)	1,346 (77)	208 (1)	451 (69)	100 (1)	180 (76)	9 (1)
LGD.SL3*	13,090	69,496	15,557	9,455	1,470	N/A	N/A	7,156 (87)	77 (1)	6,981 (86)	77 (1)	1,059 (69)	65 (1)	1,109 (93)	50 (1)
LGD.LS1	25,617	68,781	16,618	10,342	2,759	101,913	87	1,298 (77)	87 (1)	1,300 (77)	87 (1)	630 (69)	75 (1)	578 (68)	73 (1)
LGD.LS2	191,976	93,874	41,536	27,716	20,369	96,912	330	1,562 (77)	192 (1)	1,583 (76)	191 (1)	540 (69)	102 (1)	233 (86)	27 (1)
LGD.LS3*	5,791	75,184	13,970	16,678	632	N/A	N/A	12,469 (87)	54 (1)	12,493 (87)	53 (1)	795 (69)	48 (1)	602 (83)	52 (1)
LGD.SS1	8,621	66,668	14,667	8,774	1,048	101,206	35	1,251 (87)	66 (1)	1,224 (87)	66 (1)	655 (70)	63 (2)	613 (91)	65 (1)
LGD.SS2*	13,090	69,620	15,798	9,332	1,484	N/A	N/A	7,183 (87)	77 (1)	7,170 (86)	77 (1)	764 (77)	65 (1)	734 (93)	53 (1)
LGD.SS3*	5,791	75,234	14,030	16,713	625	N/A	N/A	12,459 (87)	53 (1)	12,425 (87)	54 (1)	1,127 (69)	49 (1)	1,080 (82)	69 (1)
LGD.LL1	191,976	94,626	42,383	27,424	20,281	99,163	324	1,598 (79)	192 (1)	1,609 (77)	192 (1)	623 (69)	101 (1)	309 (84)	22 (1)

Table 13: Spatial kNN queries on YAGO for $k = 5$ (total response time in *ms* - optimizer time in parentheses)

Query	Number of results (RDF)	GraphDB		SRX							
				Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding (Unsorted)		Encoding (Sorted)	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
YAGO.SL1*	11,547	86,349	3,054	6,108 (51)	59 (1)	6,170 (51)	58 (1)	2,411 (72)	57 (1)	2,111 (86)	43 (1)
YAGO.SL2*	6,030	102,461	1,977	7,553 (120)	69 (1)	7,537 (124)	69 (1)	3,068 (169)	66 (1)	2,864 (228)	43 (1)
YAGO.LS1*	2,226	-	-	2,432 (53)	58 (1)	3,943 (110)	98 (16)	1,349 (80)	60 (1)	1,158 (82)	43 (1)
YAGO.LS2*	285,613	>5 min	137,117	13,340 (129)	715 (1)	13,253 (129)	717 (1)	6,009 (193)	720 (1)	5,567 (211)	676 (1)
YAGO.SS1*	6,030	101,674	1,977	7,530 (111)	69 (1)	7,526 (115)	69 (1)	3,096 (192)	66 (1)	2,803 (210)	40 (1)
YAGO.SS2*	7,074	52,999	683	4,100 (51)	60 (1)	4,106 (50)	60 (1)	2,305 (82)	59 (1)	1,850 (70)	21 (1)
YAGO.LL1*	285,613	>5 min	138,160	13,248 (139)	715 (1)	13,220 (127)	718 (1)	6,071 (185)	719 (1)	5,654 (226)	689 (1)
YAGO.LL2*	152,693	169,217	15,148	4,709 (50)	2,992 (1)	4,721 (51)	2,987 (1)	5,750 (75)	3,016 (1)	13,118 (83)	2,273 (1)

Table 14: Spatial kNN queries on YAGO for $k = 10$ (total response time in *ms* - optimizer time in parentheses)

Query	Number of results (RDF)	GraphDB		SRX							
				Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding (Unsorted)		Encoding (Sorted)	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
YAGO.SL1*	11,547	86,351	3,055	6,045 (50)	58 (1)	6,097 (50)	58 (1)	2,395 (66)	57 (1)	2,115 (86)	43 (1)
YAGO.SL2*	6,030	102,464	1,979	7,560 (115)	69 (1)	7,548 (121)	69 (1)	3,117 (190)	66 (1)	2,850 (228)	43 (1)
YAGO.LS1*	2,226	-	-	2,431 (51)	58 (1)	3,922 (91)	97 (16)	1,304 (77)	61 (1)	1,152 (82)	43 (1)
YAGO.LS2*	285,613	>5 min	137,120	13,284 (127)	716 (1)	13,208 (137)	716 (1)	5,983 (185)	721 (1)	5,547 (209)	674 (1)
YAGO.SS1*	6,030	101,675	1,979	7,544 (109)	69 (1)	7,574 (121)	69 (1)	3,112 (195)	67 (1)	2,804 (208)	40 (1)
YAGO.SS2*	7,074	53,002	684	4,117 (53)	59 (1)	4,111 (51)	60 (1)	2,327 (77)	59 (1)	1,844 (66)	22 (1)
YAGO.LL1*	285,613	>5 min	138,161	13,343 (127)	716 (1)	13,347 (131)	714 (1)	6,030 (179)	723 (1)	5,713 (227)	690 (1)
YAGO.LL2*	152,693	169,219	15,149	4,726 (51)	2,987 (1)	4,711 (50)	2,990 (1)	5,811 (82)	3,019 (1)	13,114 (86)	2,275 (1)

Table 15: Spatial kNN queries on YAGO for $k = 20$ (total response time in *ms* - optimizer time in parentheses)

Query	Number of results (RDF)	GraphDB		SRX							
				Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding (Unsorted)		Encoding (Sorted)	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
YAGO.SL1*	11,547	86,354	3,057	6,073 (50)	59 (1)	6,117 (50)	59 (1)	2,415 (72)	57 (1)	2,096 (68)	43 (1)
YAGO.SL2*	6,030	102,468	1,982	7,549 (114)	69 (1)	7,577 (122)	68 (1)	3,103 (195)	66 (1)	2,854 (228)	43 (1)
YAGO.LS1*	2,226	-	-	2,449 (51)	57 (1)	3,946 (105)	97 (16)	1,335 (80)	61 (1)	1,141 (78)	43 (1)
YAGO.LS2*	285,613	>5 min	137,122	13,345 (127)	718 (1)	13,308 (125)	716 (1)	6,017 (185)	722 (1)	5,589 (211)	676 (1)
YAGO.SS1*	6,030	101,678	1,980	7,543 (114)	69 (1)	7,542 (113)	69 (1)	3,092 (191)	67 (1)	2,829 (214)	40 (1)
YAGO.SS2*	7,074	53,006	686	4,098 (50)	60 (1)	4,098 (50)	59 (1)	2,293 (82)	59 (1)	1,838 (70)	23 (1)
YAGO.LL1*	285,613	>5 min	138,165	13,261 (123)	718 (1)	13,313 (129)	717 (1)	6,055 (191)	721 (1)	5,657 (226)	688 (1)
YAGO.LL2*	152,693	170,222	15,251	4,720 (55)	2,990 (1)	4,720 (51)	2,983 (1)	5,820 (77)	3,017 (1)	13,084 (86)	2,280 (1)

Table 16: Spatial kNN queries on YAGO for $k = 50$ (total response time in *ms* - optimizer time in parentheses)

Query	Number of results (RDF)	GraphDB		SRX							
				Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding (Unsorted)		Encoding (Sorted)	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
YAGO.SL1*	11,547	88,397	3,189	6,088 (51)	59 (1)	6,089 (50)	59 (1)	2,417 (74)	57 (1)	2,111 (72)	43 (1)
YAGO.SL2*	6,030	104,512	2,104	7,552 (115)	69 (1)	7,540 (113)	69 (1)	3,114 (194)	66 (1)	2,879 (230)	44 (1)
YAGO.LS1*	2,226	-	-	2,475 (51)	58 (1)	3,884 (89)	97 (16)	1,371 (82)	61 (1)	1,155 (82)	43 (1)
YAGO.LS2*	285,613	>5 min	137,227	13,337 (125)	716 (1)	13,245 (129)	715 (1)	5,993 (187)	725 (1)	5,589 (208)	677 (1)
YAGO.SS1*	6,030	104,722	1,996	7,543 (116)	69 (1)	7,536 (115)	69 (1)	3,084 (181)	67 (1)	2,831 (210)	40 (1)
YAGO.SS2*	7,074	54,011	753	4,096 (50)	60 (1)	4,096 (50)	60 (1)	2,311 (77)	59 (1)	1,876 (66)	23 (1)
YAGO.LL1*	285,613	>5 min	139,201	13,307 (123)	716 (1)	13,351 (131)	716 (1)	6,029 (177)	725 (1)	5,667 (204)	689 (1)
YAGO.LL2*	152,693	171,984	15,305	4,701 (51)	2,989 (1)	4,728 (51)	2,990 (1)	5,804 (79)	3,012 (1)	13,122 (86)	2,275 (1)

Table 17: Spatial kNN queries on YAGO for $k = 100$ (total response time in *ms* - optimizer time in parentheses)

Query	Number of results (RDF)	GraphDB		SRX							
				Baseline (RDF-3X)		Basic extension (Sec. 4)		Encoding (Unsorted)		Encoding (Sorted)	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
YAGO.SL1*	11,547	91,800	3,308	6,085 (50)	59 (1)	6,109 (58)	59 (1)	2,429 (74)	57 (1)	2,146 (70)	43 (1)
YAGO.SL2*	6,030	107,319	2,261	7,552 (115)	69 (1)	7,520 (113)	69 (1)	3,149 (195)	66 (1)	2,871 (228)	44 (1)
YAGO.LS1*	2,226	-	-	2,447 (51)	57 (1)	10,788 (98)	320 (16)	1,367 (78)	60 (1)	1,590 (70)	147 (1)
YAGO.LS2*	285,613	>5 min	137,529	13,344 (129)	715 (1)	13,258 (127)	717 (1)	6,009 (189)	725 (1)	5,557 (201)	680 (1)
YAGO.SS1*	6,030	107,657	2,211	7,545 (120)	69 (1)	7,541 (114)	69 (1)	3,135 (193)	66 (1)	2,850 (208)	40 (1)
YAGO.SS2*	7,074	55,145	878	4,118 (50)	59 (1)	4,096 (50)	60 (1)	2,363 (80)	59 (1)	1,879 (68)	24 (1)
YAGO.LL1*	285,613	>5 min	142,926	13,252 (129)	713 (1)	13,374 (137)	718 (1)	6,077 (185)	727 (1)	5,729 (227)	691 (1)
YAGO.LL2*	152,693	175,855	15,543	4,718 (53)	2,989 (1)	4,729 (50)	2,987 (1)	5,783 (80)	3,017 (1)	13,135 (86)	2,274 (1)

Table 18: Median speedups of SRX over Strabon, GraphDB, Virtuoso for LGD and YAGO across all queries of each type

Query Type	Strabon (LGD)		GraphDB (LGD)		GraphDB (YAGO)		Virtuoso (LGD)		Virtuoso (YAGO)	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
range	168.4	933.4	34.4	151.8	8.9	43.7	145.9	4.7	13.1	9.5
distance	31.2	339.8	31.2	397	8.5	13	N/A	N/A	N/A	N/A
unsorted kNN	130.9	264.6	22	20	33.3	30	192.9	2.1	N/A	N/A
sorted kNN	140.2	437.5	24.9	34.2	36.2	49.5	290	6.9	N/A	N/A

Update Workloads. To generate realistic update workloads we split deltas in batches of varying size (in number of triples). For the smaller LGD dataset, we present experiments for batch sizes of 1M, 2M, 4M, and 8M triples whereas, for YAGO, we use batch sizes of 1M, 16M, 32M, and 64M triples. Each batch is marked as *delete*, *update* or *insert* and contains randomly selected triples from the respective delta. For instance, to form the delete batches of a specific batch size b, we first collect all delete triples ('hasGeometry' and 'Other') included in the delete delta, we shuffle them, and group them in batches of size b each. In all experiments of the next section, the batches are applied in a particular order, simulating the transition from the initial to the final version of the dataset: first all deletions, then all updates, and finally all insertions. The update benchmark can be found in the supplementary material (Online Resource 1).

9.4 Updates Comparison

We compare SRX with Strabon, GraphDB, and Virtuoso. Results are given in Fig. 10 and Fig. 11 for LGD and YAGO respectively. Update experiments with Strabon (Fig. 12) were feasible only on small subsets of LGD and are discussed later on. All reported times for SRX and RDF-3X reflect the total time needed to construct the in-memory differential indexes and synchronize them with the base indexes (i.e. the B^+-trees on disk).

Deletes. As shown in Fig. 10 and Fig. 11, the latency of processing a delete batch with RDF-3X and SRX shows a slight improvement across consecutive batches of the same size (left-most part of each plot) and tends to increase on average with the batch size. The slight improvement of delete latency over time is more apparent with YAGO and is reasonable since the overall database tends to get smaller. SRX latencies are in general higher than those of RDF-3X, and this is because SRX performs additional dictionary lookups and base index scans when trying to re-encode entities at lower levels of the grid. Overall, the median slowdown of SRX over RDF-3X accross all delete batches is small: 0.5× for LGD, and 0.8× for YAGO. The slowdown is smaller for YAGO because the geometries in the delete batches of YAGO are almost half on average compared to LGD, hence, re-encoding is triggered less frequently. Note that the high processing latencies in the first delete batches of each experiment are due to some warm-up issues.

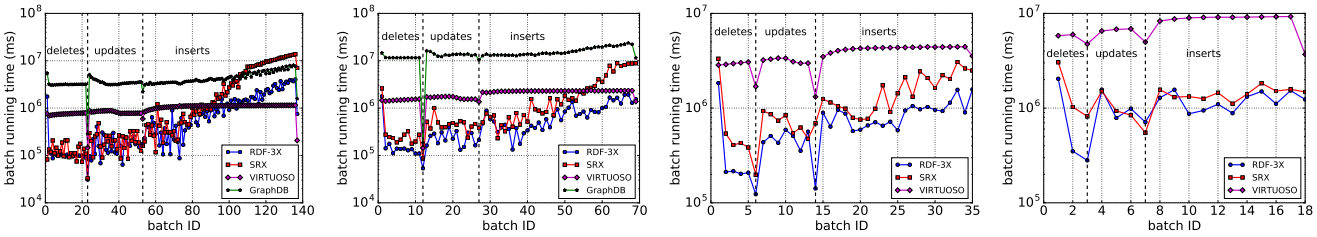
Inserts. The right-most part of each plot in Fig. 10 and Fig. 11 (labeled with "inserts") shows the performance of SRX and RDF-3X for inserts. In contrast to deletes, the latency of processing an insert batch tends to increase over time for both SRX and RDF-3X, although it appears more stable for larger batches, especially for LGD (Fig. 10d). We attribute this both to the differential index construction and to the index synchronization phases (cf. Sect. 8).

During the differential index construction, SRX and RDF-3X perform a number of database scans to check whether an input triple is new or old[13]. Hence, when the database increases in size as more insert batches are processed, the cost of these scans tends to increase. Second, and most important, whenever an index synchronization is performed and a leaf page of a base B^+-tree overflows, the system generates a new page for the additional triples but avoids tree compaction. This is a common technique that tries to minimize the update latency at the cost of redundant leaf pages, which are expected to be filled by subsequent updates or compacted periodically when the system is idle. The technique achieves better leaf page utilization when used for bulk inserts of large size but does not perform well for small frequent inserts like those in Fig. 10a-b and Fig. 11b-c. Applying many small insert batches one after the other results in a large number of almost empty leaf pages (hence, a large increase in I/Os), and the performance of the system degrades significantly. This is in fact the reason we do not present times for insert batches in Fig. 11a; after some point in this experiment, the processing of each single insert batch started taking almost double the time of the previous batch and we had to stop it. The problem we described is exacerbated due to the extensive use of indexes in RDF-3X (all of which have to be updated with the new triples) and becomes even more profound in SRX due to the additional reassigment of IDs when new geometries are introduced for existing entities (*lines* 6-16 in UPDATES ON EXISTING ENTITIES). The latter also explains why the difference in performance of inserts between SRX and RDF-3X is amplified over time in Fig. 10a-c. Overall, the median slowdown of SRX over RDF-3X across all insert batches is 0.6× for both LGD and YAGO.

It should be noted, that SRX inherits its design from the last version of RDF-3X which targets read-intensive workloads and performs well under bulk inserts, but does not focus on Online transaction processing. There is some preliminary work on OLTP in [30], but it has not been integrated to the current RDF-3X. We leave the efficient management of single triples and very small insert as future work.

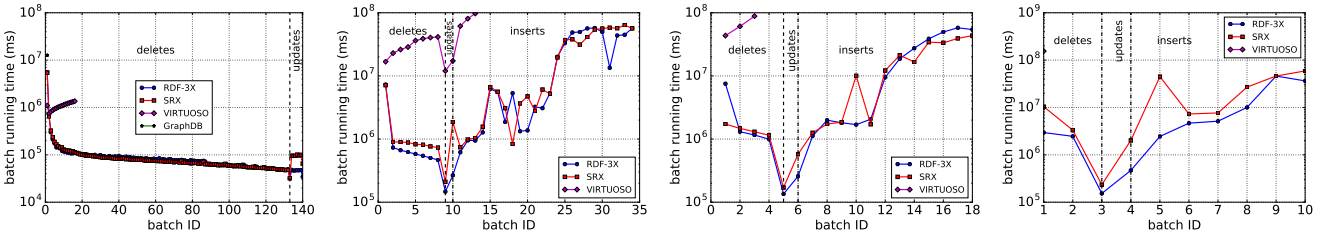
Updates. The middle part of each plot in Fig. 10 and Fig. 11 (labeled with "updates") shows the performance of SRX and RDF-3X in processing update batches. Here, the latency fluctuates slightly but tends to remain stable in the long term. Since an update in SRX and RDF-3X is implemented as a delete followed by an insert, we attribute the reasons behind this behavior to the combined performance of delete and insert batches, as explained before. Overall, the median slowdown of SRX over RDF-3X is 0.7× for LGD and 0.3× for YAGO.

[13] This check was not included in the version of RDF-3X we had but we added it for consistency.



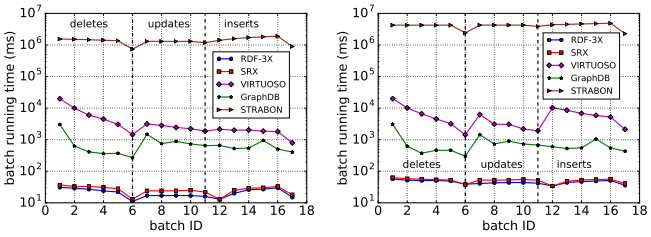
(a) b = 1M triples **(b)** b = 2M triples **(c)** b = 4M triples **(d)** b = 8M triples

Fig. 10: Latency (*ms*) of processing batches of size b on LGD



(a) b = 1M triples **(b)** b = 16M triples **(c)** b = 32M triples **(d)** b = 64M triples

Fig. 11: Latency (*ms*) of processing batches of size b on YAGO



(a) initial base = 10K triples **(b)** initial base = 100K triples

Fig. 12: Latency (*ms*) of processing batches of size 1K triples on two different subsets **(a)** and **(b)** of LGD; the former dataset is a subset of the latter

Comparison with Existing Systems. We also performed experiments with Strabon, GraphDB, and Virtuoso, which we allowed to use the whole available memory of our machine, as for queries. Results are shown in Fig. 10 for LGD, in Fig. 11 for YAGO, and in Fig. 12 for the LGD subsets. In YAGO, both GraphDB and Virtuoso were quite slow and, for this reason, we stopped the respective experiment at the point where each system had been running for as long as SRX took to apply all delete, update, and insert batches. Moreover, GraphDB did not perform updates for batch sizes 4M and 8M in LGD, and 16M, 32M, and 64M in YAGO, due to memory overflow. In the following, we first discuss the update results on LGD and YAGO, and we then explain the results on the LGD subsets that we generated specifically for Strabon.

For LGD and YAGO, SRX performs significantly better compared to GraphDB and Virtuoso in both datasets, especially for deletes and updates, even though it also applies spatial re-encoding of entities. For inserts, SRX is almost an order of magnitude faster (in the worst case) with large batches, and gets worse than GraphDB and Virtuoso only when using batch sizes of 1M and 2M in LGD. We have already explained the reasons behind SRX's performance degradation for inserts in the previous. Overall, the median speedup of SRX over GraphDB (resp. Virtuoso) for deletes, inserts, and updates is: 35.3×, 8.9×, and 22.6× (resp. 5.9×, 2.4×, and 4.1×) for LGD. For YAGO, the respective median speedup of SRX over Virtuoso is: 41.2×, 82.9×, and 9.2× (GraphDB was slow for YAGO and managed to apply only the first delete batch in Fig. 11a).

The two datasets of Fig. 12 are selected subsets from LGD, which we describe in the supplementary material of this paper (Online Resource 1). Overall, the median speedup of SRX over GraphDB (resp. Virtuoso) for deletes, inserts, and updates on both these datasets is: 12.4×, 16.2×, and 22.6× (resp. 132.1×, 91.9×, and 79.8×), while SRX performs very close to RDF-3X. Compared to results we get with LGD and YAGO, it is worth noticing that GraphDB performs better than Virtuoso with smaller datasets. Still, as for the case of queries, we cannot further comment on the performance of GraphDB and Virtuoso over all update experiments because they are not open-source systems.

Strabon is orders of magnitude slower than SRX, and at least two orders of magnitude slower than Virtuoso. Strabon performs poorly because it does not support bulk updates; instead, each record in the update batch is treated as an individual database transaction. This approach has a high over-

head in performance and cannot scale with frequent updates. As a side note, Strabon builds on the Sesame RDF store [14] and extends Sesame's query engine and optimizer, but not its transaction processing module.

10 Conclusion and Future Work

In this paper we presented SRX, a system for spatial RDF data management built on top of the popular RDF-3X system. SRX employs a flexible scheme that encodes approximations of the geometries of the RDF entities into the entities' IDs. The encoding is based on a hierarchical decomposition of the 2D space and can be effectively exploited in the evaluation of SPARQL queries with various types of spatial filters (ranges, distance joins, kNNs). We did experiments with real datasets showing that our approach minimizes the evaluation cost of the spatial component in all RDF queries, while incurring a small overhead during updates.

In the future, we plan to extend our update mechanism to support online updates in the spirit of [30] and extend our query optimizer to consider the spatial distribution of entities that support a *characteristic set* [26]. A promising research direction that we also plan to pursue is to investigate how our encoding-based techniques can be adapted to distributed spatial analytics systems, such as those in [32], to improve their performance.

References

1. GraphDB. http://graphdb.ontotext.com.
2. Linkedgeodata. http://linkedgeodata.org/About.
3. Parliament. http://parliament.semwebcentral.org.
4. Virtuoso. http://virtuoso.openlinksw.com.
5. YAGO. https://en.wikipedia.org/wiki/YAGO_(database).
6. D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.
7. C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Old techniques for new join algorithms: A case study in RDF processing. In *ICDE Workshops, 2016*.
8. C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD*, 2016.
9. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" loaded: A scalable lightweight join query processor for RDF data. In *WWW*, 2010.
10. R. Battle and D. Kolas. Enabling the geospatial semantic web with parliament and geosparql. *Semantic Web*, 3(4):355–370, 2012.
11. M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient RDF store over a relational database. In *SIGMOD*, 2013.
12. T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *SIGMOD*, 1993.
13. A. Brodt, D. Nicklas, and B. Mitschang. Deep integration of spatial query processing into native RDF triple stores. In *GIS*, 2010.
14. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: An architecture for storing and querying RDF data and schema information. In *Semantics for the WWW*. MIT Press, 2001.
15. E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *VLDB*, 2005.
16. A. Eldawy and M. F. Mokbel. The era of big spatial data: A survey. *Foundations and Trends in Databases*, 6(3-4):163–273, 2016.
17. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.
18. M. Hadjieleftheriou, E. G. Hoel, and V. J. Tsotras. Sail: A spatial index library for efficient application integration. *GeoInformatica*, 9(4):367–389, 2005.
19. M. Koubarakis and K. Kyzirakos. Modeling and querying metadata in the semantic sensor web: The model stRDF and the query language stSPARQL. In *ESWC*, 2010.
20. K. Kyzirakos, M. Karpathiotakis, and M. Koubarakis. Strabon: A semantic geospatial DBMS. In *ISWC*, 2012.
21. J. Liagouris, N. Mamoulis, P. Bouros, and M. Terrovitis. An effective encoding scheme for spatial rdf data. *Proc. VLDB Endow.*, 7(12):1271–1282, 2014.
22. M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *SIGMOD*, 1996.
23. N. Mamoulis. *Spatial Data Management*. Morgan & Claypool Publishers, 2011.
24. N. Mamoulis and D. Papadias. Slot index spatial join. *TKDE*, 15(1):211–231, 2003.
25. K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.
26. T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, 2011.
27. T. Neumann and G. Weikum. RDF-3X: A RISC-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, 2008.
28. T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, 2009.
29. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.
30. T. Neumann and G. Weikum. x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *Proc. VLDB Endow.*, 3(1-2):256–263, 2010.
31. P. Nikitopoulos, A. Vlachou, C. Doulkeridis, and G. A. Vouros. DiStRDF: Distributed spatio-temporal RDF queries on Spark. In *EDBT/ICDT*, 2018.
32. V. Pandey, A. Kipf, T. Neumann, and A. Kemper. How good are modern spatial analytics systems? *Proc. VLDB Endow.*, 11(11):1661–1673, 2018.
33. K. Patroumpas, G. Giannopoulos, and S. Athanasiou. Towards geospatial semantic data management: Strengths, weaknesses, and challenges ahead. In *GIS*, 2014.
34. C.-J. Wang, W.-S. Ku, and H. Chen. Geo-store: A spatially-augmented sparql query evaluation system. In *GIS*, 2012.
35. D. Wang, L. Zou, Y. Feng, X. Shen, J. Tian, and D. Zhao. S-store: An engine for large RDF graph integrating spatial information. In *DASFAA*, 2013.
36. C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, 2008.
37. K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *SWDB*, 2003.
38. Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan. Efficient indices using graph partitioning in RDF triple stores. In *ICDE*, 2009.
39. P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: A fast and compact system for large scale RDF data. *Proc. VLDB Endow.*, 6(7):517–528, 2013.
40. K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *Proc. VLDB Endow.*, 6(4):265–276, 2013.
41. L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering SPARQL queries via subgraph matching. *Proc. VLDB Endow.*, 4(8):482–493, 2011.