An Effective Encoding Scheme for Spatial RDF Data*

John Liagouris[†], Nikos Mamoulis[†], Panagiotis Bouros[‡], Manolis Terrovitis[§]

[†]The University of Hong Kong [‡]Humboldt-Universität zu Berlin [§]IMIS 'Athena'

[†]{iagouris, nikos}@cs.hku.hk [‡]bourospa@informatik.hu-berlin.de

[§]mter@imis.athena-innovation.gr

ABSTRACT

The RDF data model has recently been extended to support representation and querying of spatial information (i.e., locations and geometries), which is associated with RDF entities. Still, there are limited efforts towards extending RDF stores to efficiently support spatial queries, such as range selections (e.g., find entities within a given range) and spatial joins (e.g., find pairs of entities whose locations are close to each other). In this paper, we propose an extension for RDF stores that supports efficient spatial data management. Our contributions include an effective encoding scheme for entities having spatial locations, the introduction of on-the-fly spatial filters and spatial join algorithms, and several optimizations that minimize the overhead of geometry and dictionary accesses. We implemented the proposed techniques as an extension to the open-source RDF-3X engine and we experimentally evaluated them using real RDF knowledge bases. The results show that our system offers robust performance for spatial queries, while introducing little overhead to the original query engine.

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. A SPARQL query includes a Select clause, specifying the output variables and a Where clause which includes the conditions that bind the variables together (or with literals), forming a *query graph pattern* that has to be matched in the RDF data graph. The recent GeoSPARQL standard [7], defined by the Open Geospatial Consortium (OGC), extends RDF and SPARQL to represent geographic information and support spatial queries. Real-world entities, represented as resources in RDF, may have geometries, modeled by basic shapes, such as points and polygons. GeoSPARQL uses the OGC's Simple Features ontology for spatial entities. Geospatial filter functions are used to evaluate topological and distance relationships between entities and express spatial predicates in SPARQL queries. stSPARQL [16], developed independently to GeoSPARQL, has similar features.

Despite the large volume of work on indexing and querying large RDF knowledge bases [5, 6, 8, 11, 12, 21, 25, 26, 27, 28, 29, 30], 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], Parliament [3], Strabon [17], and others [10, 23, 24]) focus mainly on supporting GeoSPARQL features, and less on performance optimization. The features and weaknesses of these systems are reviewed in Section 3. On the other hand, there is a large number of spatial entities (i.e., resources) in RDF knowledge bases (e.g., YAGO2 [15]). 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.

In this paper, we attempt to fill this gap by proposing a number of extensions that can be applied to RDF engines in order to efficiently support spatial queries. We present the details of a system, which extends the open-source RDF-3X store [21]. RDF-3X 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 [20]. RDF-3X is shown to have robust performance in comparison studies on various RDF datasets and query benchmarks [8, 21, 28]. Although we have chosen RDF-3X as a proof of concept for implementing our ideas, our techniques are also applicable to other RDF stores which have been developed recently (e.g., [28]). In a nutshell, our system includes the following extensions over RDF-3X:

Index Support for Spatial Queries. Similar to previous spatial extensions of RDF stores (e.g., [10]), our system includes a spatial index (i.e., an R-tree [13]) for the geometries associated to the spatial entities. This facilitates the efficient evaluation of queries with very selective spatial components.

*Work supported by grants Hong Kong RGC HKU 715413E, German Research Foundation (DFG) GRK 1324: Research Training Group METRIK, and EU/Greece KRIPIS: MEDA Project.

Proceedings of the VLDB Endowment, Vol. 7, No. 12

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 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.

We evaluate our system by comparing it with two commercial spatial RDF management systems, Virtuoso [4] and OWLIM-SE [2]. For our evaluation, we use two real datasets: LinkedGeoData (LGD) [1] and YAGO2 [15]. The results demonstrate the superior performance and robustness of our approach.

The rest of the paper is organized as follows. Section 2 includes definitions and examples of GeoSPARQL queries that we consider in this paper. Section 3 reviews related work on RDF stores and spatial extensions thereof. In Section 4, we show how RDF-3X can be extended to use a spatial index for the entities associated with geometries. Section 5 presents our proposal of approximately encoding the geometries of entities inside their IDs. Query evaluation techniques that take advantage of this encoding are presented in Section 6. Section 7 presents our extensions to the query optimizer. Section 8 includes our experimental evaluation and Section 9 concludes the paper.

2. PRELIMINARIES

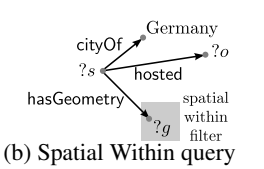
The SPARQL queries we consider in this work follow the format:

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

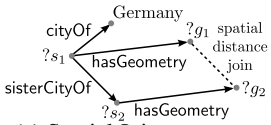
The Select clause includes a set of variables that should be instantiated from the RDF knowledge base (variables in SPARQL are denoted by a ? prefix). A graph pattern in the Where clause consists of triple patterns in the form of $s\ p\ o$ where any of the s, p and o can be either a constant or a variable. 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) and DISTANCE predicates (for spatial joins). However, we emphasize that the results of our work are directly applicable to all spatial predicates defined in the GeoSPARQL standard [7]. In addition, we use a simplified syntax for expressing queries and not the one of the GeoSPARQL standard because the latter is verbose.

subject	*property*	*object*
Dresden	cityOf	Germany
Prague	cityOf	CzechRepublic
Leipzig	cityOf	Germany
Wrocław	cityOf	Poland
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
Dresden	hasGeometry	"POINT (...)"
Prague	hasGeometry	"POINT (...)"
Leipzig	hasGeometry	"POINT (...)"
...	...	...

(a) RDF triples



(b) Spatial Within query



(c) Spatial Join query

Figure 1: Example of RDF data and two spatial queries

As an example, consider the (incomplete) RDF knowledge base listed in Figure 1(a). 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 (Leipsiz, Bach) are results of this query. The query is represented by the pattern graph of Figure 1(b). 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 Figure 1(c), 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 Figure 1(a), (Dresden, Worcław) and (Leipzig, Hanover) are results of this query while (Dresden, Ostrava) is not returned as the distance between Dresden and Ostrava is around 500km.

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)

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

Figure 2: Use of Dictiorary

statements in a single, very large *triples* table. The RDF-3X system [21] 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. Figure 2 illustrates a dictionary and the ID-encoded triples for the RDF base of Figure 1(a). 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 [22] 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, PO, PS, 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 [20] 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 [25] 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 [12] 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 [26] and Sesame [11]), where correlated tables are clustered into the same table and triples with infrequent properties are placed into a *left-over* table. Abadi et al. [5] 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. [6] 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 [28]; 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 [8]. 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 [29] 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 [30] is an earlier, graph-based approach, which models SPARQL queries as graph pattern matching queries on the RDF graph.

Spatial Extensions of RDF Stores. Parliament [7], built on top of Jena [26], implements most of the features of GeoSPARQL. Strabon [17], developed in parallel to Parliament, extends Sesame [11] 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 at a separate table, which is indexed by an R-tree [13]. 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. [10] extend RDF-3X [21] 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 [23] 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 [24] is a spatial extension of gStore [30]. 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). Finally, commercial systems, like Oracle, Virtuoso [4], and OWLIM-SE [2] have spatial extensions, however, details about their internal design are not public.

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 and discusses drawbacks of it that motivated us to design and use a spatial encoding scheme described in Sections 5 and 6. This basic extension improves the spatial RDF-3X extension of Brodt et al. [10] to support spatial join evaluation.

Spatial Indexing. Spatial entities i.e., resources associated to spatial literals like POINT and POLYGON, are indexed by an R-tree [13]. 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 entity 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. Figures 3(a) and 3(b) illustrate two alternative plans for the spatial selection query of Figure 1(b). The plan of Figure 3(a) 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 Figure 3(b) 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).

Spatial Joins. The R-tree can also be used to evaluate spatial join Filter conditions, by applying join algorithms based on R-trees. We implemented three algorithms for this purpose. First, the R-tree join algorithm [9] 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 [19] for the case where the R-tree can be used only for one variable. Finally, we implemented a spatial hash join (SHJ) algorithm [18] for the case where both inputs of the spatial join filter condition are output by other operators. (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.) 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. Figures 3(c) and 3(d) illustrate two alternative plans for the spatial join query of Figure 1(c). The plan of Figure 3(c) applies an R-tree self-join [9] 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 Figure 3(d) 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.

[1]For entities that have point geometries, the spatial selection can always be evaluated exactly using only the R-tree. On the other hand, if the entities have polygon geometries, the R-tree search may allow for false positives; in this case, the final results of the spatial filter are confirmed by retrieving the exact polygon geometries from the dictionary, using the IDs of the entities.

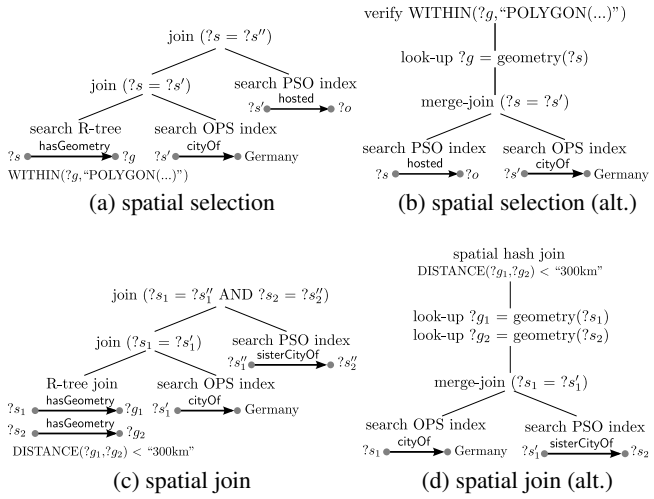


Figure 3: Query plans in the basic extension

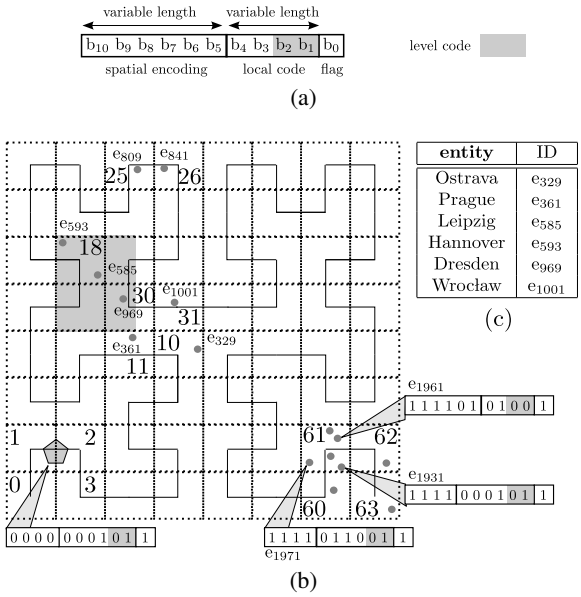


Figure 4: Spatial encoding of entity IDs

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.

Figure 4(b) 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 Figure 4(a), 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 Figure 4(b), 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. Figure 4(c) illustrates which IDs encode the cities of Figure 1(a).

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 Figure 4(b) and assume than 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 Figure 4(b) 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 Hibert-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 Section 7). Finally, extending current systems (e.g., RDF-3X) to use this spatial encoding is quite easy.

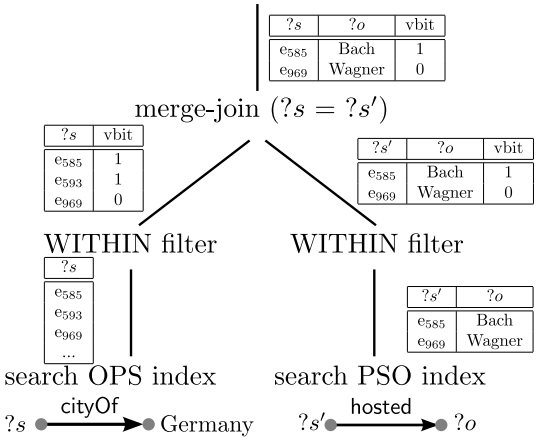


Figure 5: Plan for the query of Figure 1(b)

6. QUERY EVALUATION

We now show how our encoding scheme further extends the basic framework presented in Section 4 to apply spatial filters early and on-the-fly and significantly accelerate the evaluation of GeoSPARQL queries. In a nutshell, after each non-spatial operator that instantiates entity variables, which also appear in a spatial Filter condition, the condition is applied on the spatially encoded IDs of the entities. In general, the sooner we apply these on-the-fly spatial filters, the better because they do not incur any I/O cost and their CPU cost is negligible.[2] After the application of a spatial filter, we append a *verification bit* (or vbit) to the tuples that survive the filter. If, for a tuple, this 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.

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 Figure 1(b). 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 Figure 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 Figure 3(b). Taking advantage of the spatial encoding, before the merge-join, the plan of Figure 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 Figure 4(b). 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

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

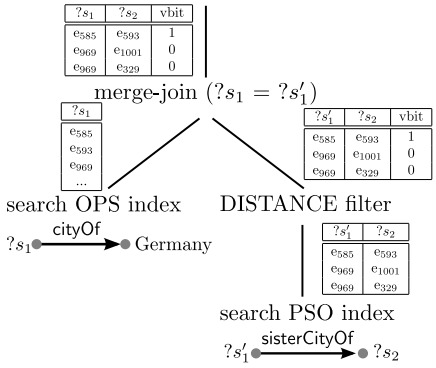


Figure 6: Plan for the query of Figure 1(c)

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 Figure 1(c). A possible query evaluation subplan is given in Figure 6, which follows the flow of the plan shown in Figure 3(d); however, the plan of Figure 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 Figure 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 Figure 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_{585}, 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 Section 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

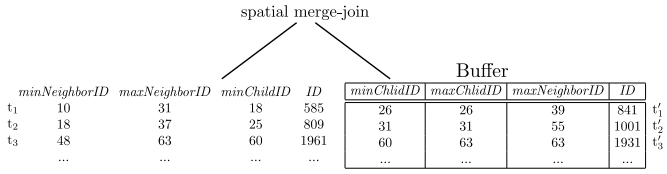


Figure 7: Example of SMJ

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 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 Section 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 Figure 4(b) 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.[3] For example, consider the buffered inputs of Figure 7 that have to be joined. The *maxNeighborID* of the first entity e_{585} on the left is smaller than the *min-*

[3]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.

ChildID 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}).[4] 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 bit-shifting operations. 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 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

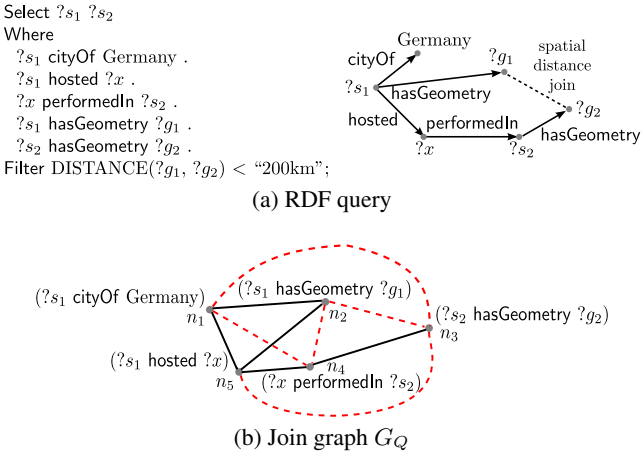
If either of the two inputs of a spatial join are not ordered with respect to the joined entities, SMJ is not applicable. In this case we can still use the IDs of the joined entities to perform the filter step of the spatial join. The idea is to apply a *spatial hash join* (SHJ-ID) algorithm (similar to that proposed in [18]) using the approximate geometries of the entities taken from their IDs.[5] 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 Runtime Optimizations

RDF-3X uses a lightweight Sideways Information Passing (SIP) mechanism for skipping redundant values when scanning the indexes [22]. 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.

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

[5]Recall that the actual geometries of the entities have not been retrieved yet; otherwise, SHJ [18] would be used (see Section 4).



(a) RDF query

(b) Join graph G_Q

Figure 8: Augmenting a query graph

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 Figure 3(a)), 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 Section 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.

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 Section 4) and (ii) the query evaluation techniques described in Section 6, based on the spatial encoding of entity IDs.

7.1 Augmenting the Query Graph

Consider the query depicted in Figure 8(a). 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 Section 6.3. We call such variables *spatial variables*. More formally:

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 Figure 8(a) is the graph shown in Figure 8(b), 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 collect all pairs of nodes in G_Q that include at least one spatial variable. In the example of Figure 8(b), 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 query 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 selection on x. In the augmented G_Q for our example (Figure 8(b)) 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) in the set of filters of Q, so that this filter can be applied in any (intermediate) relation that contains the spatial variable $?s$. Similarly, for each pair (s_i,s_j) of joined spatial variables, we add the corresponding spatial join condition in the set of filters of Q, so that this filter can be applied on the fly 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

```
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;
for each pair ?s_i, ?s_j of joined spatial variables do
    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
        if there is no edge in G_Q between n_i and n_j then
            Add a new edge denoting the filtering phase of the
              spatial join of ?s_i and ?s_j;
        else
            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;
For each spatial variable ?s appearing in a WITHIN predicate, add
  WITHIN(?s, GEOMETRY) to filtering conditions of Q;
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;
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 query optimizer is going to consider in each case.

Case	Algorithm(s) to Consider
L and R sorted on entity IDs	SMJ (Section 6.3)
L and R results of ($?s_i$ hasGeometry $?g_i$)	SMJ or R-tree Join [9]
L sorted on entity IDs R result of a pattern ($?s_2$ hasGeometry $?g_2$)	SMJ (Section 6.3), SISJ [19], or Index Nested Loops
L unsorted R result of a pattern ($?s_2$ hasGeometry $?g_2$)	SHJ-ID (Section 6.4), SISJ [19] or Index Nested Loops
L and R unsorted	SHJ-ID, SHJ [18] or Nested Loops

Table 1: Spatial Join Scenarios in Optimal Plan Construction

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 (Section 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 [9] on the R-tree index (see Section 4). 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 [19] (see Section 4). 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 (Section 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 any of the inputs is too small.

7.3 Query Optimization

We extend the query optimizer of RDF-3X to consider all possible spatial join cases and algorithms outlined in Section 7.2. In addition, the optimizer considers the case of performing a spatial selection Filter using the R-tree (see Section 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 Figure 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 [19]). 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 Figure 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.

8. EXPERIMENTAL EVALUATION

In this section we present an experimental evaluation of our techniques on spatially enriched RDF data. Section 8.1 discusses the implementation details of our methodology and the experimental setup. Section 8.2 compares our extended version of RDF-3X against the original system [21] and two commercial triple stores with spatial query support, namely Virtuoso [4] and OWLIM-SE [2].

8.1 Setup

Implementation Details. We implemented our system in C++ (g++ 4.8) and all experiments were conducted on a machine with an i7-3820 CPU at 3.60GHz, a RAID hard disk of 6Tb, and 60Gb of main memory running Linux Debian (3.11-2-amd64). For the R-tree implementation, we used the open-source SaIL library [14].

Datasets. We experimentally evaluate our system using two real datasets: LinkedGeoData (LGD) [1] and YAGO2 [15]. LGD contains user-contributed content from the OpenStreetMap project. YAGO2 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 1.7GB (LGD) and 16GB (YAGO2). The R-trees (using 4KB nodes) occupy 152MB and 212MB, respectively. The size of the grid in both datasets is 358MB (89M cells in total for all levels). Note that, despite the agressive indexing, in both cases, we end up with a database size having less than 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); this is reflected by the percentage of geometries that reside at the different levels of our encoding scheme (see Table 3). YAGO2 includes a significant percentage of geometries 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.

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 also possible, but it significantly increases the size of the triple indexes, thus, we 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 can be kept in memory (for selectivity estimation purposes). In our case, the grid size is less than 1Gb 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, lines, multipoints) span the borders of quadrants at high levels of the grid and some multipoints in YAGO2 have very large MBRs.

Queries. All queries we used in our experiments consist of 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 either a WITHIN predicate (for spatial range queries) or a DISTANCE predicate (for spatial distance joins). The range queries have similar form as that of Figure 1(b); 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) will be discussed in Section 8.2. All query expressions can be found in the Appendix.

Dataset	Triples	Entities	Points	Polygons	Lines	Multipoints
LGD (3 Gb)	15.4M	10.6M	590K	264K	2.6M	0
YAGO2 (22 Gb)	205.3M	108.5M	4M	0	0	780K

Table 2: Characteristics of the datasets

Level	0 (bottom)	1	2	3	4	5	6	≥ 7
LGD	42.7	13.7	13.2	11.1	7.9	5.1	3.0	3.3
YAGO2	50.3	19.2	8.1	4.5	3.0	2.4	1.9	10.6

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

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 Section 7) and the time spent in the ID-to-string dictionary lookups for the variables in the Select clause.

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.[6] 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 full-fledged database system. To illustrate the effect of caching (by the system's kernel) in the overall response time of the system, we report query evaluation times on warm and cold caches separately.

8.2 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 Section 4 (Basic) and the original RDF-3X system (Baseline). Only for queries where the spatial component is more selective (LS class), Basic utilizes the R-tree in order to retrieve the entities that fall in the given range; 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 Section 6.1) to reduce the number of entities that have to be spatially verified. Our approach 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 our approach 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 LL queries and in queries involving entities having 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 our approach 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 component of the query.

[6]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.

Query	Number of results			OWLIM-SE		Virtuoso		Baseline		Basic extension		Encoding	
	RDF	Spatial	Combined	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
LGD.SL1	524	2,537,757	411	5,836	3,054	15,207	28	1,274 (134)	36 (1)	1,285 (175)	33 (23)	1,079 (141)	75(23)
LGD.SL2	215,355	2,943,209	186,302	9,667	5,379	19,781	2,047	1,693 (123)	938 (1)	1,720 (235)	890 (72)	1,346 (186)	464 (68)
LGD.SL3*	13,090	2,537,757	9814	-	-	-	-	2,748 (126)	609 (1)	2,670 (150)	730 (31)	1,909 (140)	228 (23)
LGD.LS1	25,617	9,002	86	1,281	59	15,059	53	641 (122)	157 (1)	705 (161)	37 (12)	297 (159)	23 (9)
LGD.LS2	191,976	908	3	912	46	13,808	20	1,136 (122)	688 (1)	364 (159)	14 (11)	290 (165)	16 (8)
LGD.LS3*	5,791	908	9	-	-	-	-	4,037 (128)	137 (1)	348 (161)	14 (10)	336 (171)	12 (8)
LGD.SS1	8,621	9,002	69	1,032	58	15,434	54	638 (125)	89 (1)	718 (162)	37 (13)	311 (159)	20 (11)
LGD.SS2*	13,090	9,002	120	-	-	-	-	2,086 (130)	593 (1)	1,956 (167)	441 (12)	546 (163)	20 (14)
LGD.SS3*	5,791	9,002	7	-	-	-	-	4,148 (119)	136 (1)	4,000 (149)	101 (19)	463 (127)	19 (14)
LGD.LL1	191,976	350,405	13,416	4,254	585	17,852	182	1,275 (128)	710 (1)	1,344 (145)	678 (27)	491 (135)	81 (19)

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

Query	Number of results			Baseline		Basic extension		Encoding	
	RDF	Spatial	Combined	Cold	Warm	Cold	Warm	Cold	Warm
YAGO2.SL1*	11,547	364,992	891	3,503 (47)	64 (1)	3,604 (160)	88 (17)	3,116 (169)	66 (15)
YAGO2.SL2*	6,030	31,260	69	2,344 (161)	76 (1)	2,634 (333)	96 (16)	2,054 (251)	63 (16)
YAGO2.LS1*	2,226	138	7	1,535 (51)	58 (1)	4,345 (182)	221 (16)	1,382 (168)	59 (17)
YAGO2.LS2*	285,613	41,945	4,471	4,030 (156)	1,198 (1)	6,919 (316)	1,482 (16)	2,873 (332)	172 (15)
YAGO2.SS1*	6,030	8,440	3	1,987 (162)	77 (1)	2,173 (333)	98 (17)	1,823 (328)	63 (17)
YAGO2.SS2*	7,074	7,042	32	2,200 (46)	60 (1)	2,241 (178)	87 (16)	2,043 (171)	59 (15)
YAGO2.LL1*	285,613	184,743	10,454	4,480 (157)	1,203 (1)	5,015 (337)	1,597 (16)	3,411 (331)	189 (16)
YAGO2.LL2*	152,693	107,625	88	5,770 (49)	3,561 (1)	6,081 (187)	3,652 (16)	4,795 (172)	2,272 (16)

Table 5: Spatial range queries on YAGO2 (total response time in msecs - optimizer time in parentheses)

The difference in the optimization times (in parentheses) between warm and cold caches in all alternatives is because of including 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 our approach 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 our approach and Basic compared to Baseline (due to the use of spatial statistics) is more profound.

Similar results are observed for range queries on the YAGO2 dataset (see Table 5). All queries in this case involve entities that may have multipoint geometries (therefore they are marked by a star). Like before, Encoding always chooses to evaluate the RDF part of the queries first. Basic chooses the same plan as Baseline in all cases, except for LS queries, where it opts to evaluate the spatial selection using the R-tree. Note that in this case the cost of Basic is very high (even higher than Baseline). After analysis, we found that this is due to the bad performance of the R-tree on the YAGO2 dataset; the range queries access roughly half of the R-tree nodes. The reason is that many multipoints in YAGO2 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.

Results on Spatial Joins. Tables 6 and 7 show the costs of spatial distance join queries on LGD and YAGO2, 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 YAGO2, 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. 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 Figures 3(c) and (d)). In all queries we tested, Basic chose the SHJ option 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 Sections 6.3 and 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 our encoding does not bring benefit over Basic for join queries on LGD. The main reason for this is that, due to the data distribution, our approach does not save any geometry look-ups; 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

Query	Spatial join threshold ϵ	Number of results	OWLIM-SE		Virtuoso		Baseline		Basic extension		Encoding	
			Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
LGD.J1	0.003	6,831	164,522	159,365	19,694	8,083	114,147 (120)	110,414 (2)	1,478 (260)	211 (141)	1,754 (253)	680 (144)
LGD.J2	0.01	538	13,577	9,423	9,752	3,044	20,447 (107)	19,053 (2)	1,696 (303)	269 (202)	1,812 (326)	497 (219)
LGD.J3	0.02	8,742	>5mins	>5mins	22,683	17,125	>5mins	>5mins	1,909 (383)	498 (284)	2,150 (385)	834 (300)
LGD.J4*	0.05	795,322	-	-	-	-	>5mins	>5mins	117,881 (611)	115,634 (497)	25,698 (613)	23,688 (498)
LGD.J5*	0.01	2,782	-	-	-	-	56,815 (106)	55,324 (2)	2,772 (318)	392 (201)	3,664 (309)	1,456 (204)
LGD.J6.1*	0.0005	7	-	-	-	-	130,954 (117)	127,525 (2)	3,726 (206)	213 (98)	3,922 (228)	214 (99)
LGD.J6.2*	0.001	22	-	-	-	-	130,969 (115)	127,519 (1)	3,833 (212)	203 (98)	4,059 (214)	207 (101)
LGD.J6.3*	0.01	743	-	-	-	-	130,943 (117)	127,625 (2)	4,645 (307)	330 (201)	5,357 (312)	1,755 (203)

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

Query	Spatial join threshold ϵ	Number of results	Baseline		Basic extension		Encoding	
			Cold	Warm	Cold	Warm	Cold	Warm
YAGO2.J1*	0.1	2,635	108,464 (83)	103,980 (15)	4,618 (308)	798 (239)	4,424 (305)	892 (238)
YAGO2.J2*	0.1	6,799,189	>5 min	>5 min	153,869 (325)	151,088 (240)	144,317 (526)	140,472 (450)
YAGO2.J3*	0.1	832	3,804 (79)	195 (12)	4,018 (321)	388 (240)	4,059 (323)	386 (238)
YAGO2.J4*	0.1	451	3,603 (84)	180 (13)	3,553 (317)	339 (239)	3,611 (318)	337 (240)
YAGO2.J5*	0.1	113	2,518 (69)	107 (10)	2,784 (320)	278 (239)	2,786 (319)	276 (238)
YAGO2.J6*	0.1	664,613	>5 min	>5 min	32,734 (468)	29,185 (241)	31,596 (676)	27,972 (451)
YAGO2.J7*	0.1	4,204,184	>5 min	>5 min	42,545 (312)	38,044 (238)	6,359 (517)	2,435 (447)
YAGO2.J8.1*	0.001	85,188	>5 min	>5 min	3,039 (195)	171 (119)	2,947 (199)	457 (119)
YAGO2.J8.2*	0.01	86,222	>5 min	>5 min	3,071 (188)	273 (120)	2,916 (183)	458 (119)
YAGO2.J8.3*	0.1	131,828	>5 min	>5 min	3,537 (320)	713 (239)	3,029 (308)	584 (238)

Table 7: Spatial distance join queries on YAGO2 (total response time in msecs - optimizer time in parentheses)

complete join inputs before hashing them into SHJ buckets, 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 (this explains the favorable performance of Basic over Encoding in the case of warm caches). 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 its entire inputs to be read before processing the join and can become a bottleneck if the inputs are too large.

In the YAGO2 dataset, we were able to define alternative queries with spatial join components and the results 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. In most queries, the performance of Basic is similar to that of Encoding for the reasons we explained before. For queries YAGO2.J2 and YAGO2.J6 our approach is slightly faster than Basic, because our approach selects a rather different plan, based on the augmented query graph. In query YAGO2.J7, our approach performs much better than Basic, because the spatial join inputs have a different spatial distribution and Encoding can prune many tuples using SHD-ID.

Comparison with Existing Systems. We also compared our system against two popular RDF stores with geospatial query support, namely OWLIM-SE and Virtuoso. Tables 4 and 6 include the performance of these systems on range and join queries respectively, on the LGD dataset. Note that OWLIM-SE and Virtuoso support point data only, therefore we loaded the data to them after simplifying all geometries into points (we kept one point only from each geometry) and include in our comparison only queries for which the involved geometries are points. 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 tool. In sum, our system performs significantly better in all queries, especially in spatial distance joins. We cannot comment about the reasons, since OWLIM-SE and Virtuoso are closed source and there are no published works describing their functionality and query optimization techniques. Finally, regarding YAGO2, OWLIM-SE on one hand could not load the dataset even by using all 64Gb of the available RAM, while on the other hand, Virtuoso successfully loaded the dataset but we could not evaluate any of the queries correctly (in all of them zero or incorrect results were returned).

9. CONCLUSION

In this paper we presented a number of techniques that extend an RDF store to effectively manage of spatial RDF data. We introduced a flexible scheme that encodes approximations of the spatial features of RDF entities into their IDs. The encoding is based on a hierarchical decomposition of the 2D space, it is independent from the physical design of the underlying triple store, and it can be effectively exploited in the evaluation of SPARQL queries with spatial filters. We implemented our ideas by extending the popular RDF-3X system and conducted detailed experiments with real datasets. In summary, our approach minimizes the evaluation cost incurred due to the spatial component in all RDF queries. In addition, it allows the consideration of different plans due to a query graph augmentation technique performed by our optimizer. In the future, we plan to extend our query optimizer to consider the spatial distribution of entities that support a *characteristic set* [20]. For example, cities that are coastal (and belong to a characteristic set with this property) have different distribution than the general spatial data distribution of entities. In addition, we will investigate the idea of embedding discretized spatial coordinates of the data into

the leaves of the B^+-tree indexes of RDF-3X, in order to avoid dictionary lookups to retrieve the geometries of entities. Finally, we plan to investigate alternative encoding schemes that are suited for extremely skewed spatial distributions and entities with very large geometries.

10. REFERENCES

[1] Linkedgeodata. http://linkedgeodata.org/About.

[2] Owlim-se. http://owlim.ontotext.com/display/OWLIMv43/OWLIM-SE.

[3] Parliament. http://parliament.semwebcentral.org.

[4] Virtuoso. http://virtuoso.openlinksw.com.

[5] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[6] 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.

[7] R. Battle and D. Kolas. Enabling the geospatial semantic web with parliament and geosparql. *Semantic Web*, 3(4):355–370, 2012.

[8] 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.

[9] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD*, 1993.

[10] A. Brodt, D. Nicklas, and B. Mitschang. Deep integration of spatial query processing into native rdf triple stores. In *GIS*, 2010.

[11] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: An architecture for storin gand querying rdf data and schema information. In *Spinning the Semantic Web*, pages 197–222, 2003.

[12] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient sql-based rdf querying scheme. In *VLDB*, 2005.

[13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[14] M. Hadjieleftheriou, E. G. Hoel, and V. J. Tsotras. Sail: A spatial index library for efficient application integration. *GeoInformatica*, 9(4):367–389, 2005.

[15] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194:28–61, 2013.

[16] M. Koubarakis and K. Kyzirakos. Modeling and querying metadata in the semantic sensor web: The model strdf and the query language stsparql. In *ESWC (1)*, pages 425–439, 2010.

[17] K. Kyzirakos, M. Karpathiotakis, and M. Koubarakis. Strabon: A semantic geospatial dbms. In *ISWC (1)*, pages 295–311, 2012.

[18] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *SIGMOD*, 1996.

[19] N. Mamoulis and D. Papadias. Slot index spatial join. *TKDE.*, 15(1):211–231, 2003.

[20] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *ICDE*, 2011.

[21] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, 1(1):647–659, 2008.

[22] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD*, 2009.

[23] C.-J. Wang, W.-S. Ku, and H. Chen. Geo-store: a spatially-augmented sparql query evaluation system. In *GIS*, 2012.

[24] 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 (2)*, pages 31–47, 2013.

[25] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[26] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient rdf storage and retrieval in jena2. In *SWDB*, 2003.

[27] 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.

[28] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: a fast and compact system for large scale rdf data. *PVLDB*, 6(7):517–528, 2013.

[29] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. *PVLDB*, 6(4):265–276, 2013.

[30] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gstore: Answering sparql queries via subgraph matching. *PVLDB*, 4(8):482–493, 2011.

APPENDIX

Range queries used in the experiments

All **LGD** queries have the following template:

Select $?s$
Where $?s$ name $?n$.$?s$ label $?l$.
$?s$ type **[TYPE]** . $?s$ hasGeometry $?g$.
Filter
WITHIN($?g$, "RECTANGLE(**[MBR]**)")

The table on the right shows how **[TYPE]** and **[MBR]** are instantiated in each query

QueryID	[TYPE]	[MBR]
LGD.SL1	police	-5,50,0,55
LGD.SL2	bus_stop	-10,50,0,60
LGD.SL3*	park	-5,50,0,55
LGD.LS1	pub	-5,45,0,50
LGD.LS2	bus_stop	-10,45,-5,50
LGD.LS3*	road	-10,45,-5,50
LGD.SS1	restaurant	-5,45,0,50
LGD.SS2*	park	-5,45,0,50
LGD.SS3*	road	-5,45,0,50
LGD.LL1	bus_stop	-5,55,0,60

YAGO2.SL1
Select $?gn$ $?fn$ $?pr$ Where $?p$ hasGivenName $?gn$. $?p$ hasFamilyName $?fn$. $?p$ hasWonPrize $?pr$. $?p$ diedIn $?c$. $?c$ hasGeometry $?g$.
Filter WITHIN($?g$, "RECTANGLE(-100, 20, -80, 40)")

YAGO2.SL2
Select $?gn$ $?fn$ Where $?p$ hasGivenName $?gn$. $?p$ hasFamilyName $?fn$. $?p$ a Wordnet_scientist_110560637 . $?p$ wasBornIn $?c$. $?c$ hasGeometry $?g$.
Filter WITHIN($?g$, "RECTANGLE(-95, 40, -90, 45)")

YAGO2.LS1
Select $?p$ $?w$ Where $?p$ hasAcademicAdvisor $?a$. $?a$ worksAt $?w$. $?w$ isLocatedIn $?l$. $?l$ hasGeometry $?g$.
Filter WITHIN($?g$, "RECTANGLE(-160, -50, -150, -40)")

YAGO2.LS2
Select $?e$ $?c$ Where $?e$ happenedIn $?l$. $?l$ a $?c$. $?c$ subClassOf Wordnet_city_108524735 . $?l$ hasGeometry $?g$.
Filter WITHIN($?g$, "RECTANGLE(-130, 40, -120, 50)")

YAGO2.SS1
Select $?gn$ $?fn$ Where $?p$ hasGivenName $?gn$. $?p$ hasFamilyName $?fn$. $?p$ a Wordnet_scientist_110560637 . $?p$ wasBornIn $?c$. $?c$ hasGeometry $?g$.
Filter WITHIN($?g$, "RECTANGLE(-105, 45, -100, 50)")

YAGO2.SS2
Select $?p$ $?w$ Where $?p$ graduatedFrom $?u$. $?p$ worksAt $?w$. $?u$ isLocatedIn $?l$. $?l$ hasGeometry $?g$.
Filter WITHIN($?g$, "RECTANGLE(-110, 50, -100, 60)")

YAGO2.LL1
Select $?e$ $?c$ Where $?e$ happenedIn $?l$. $?l$ a $?c$. $?c$ subClassOf Wordnet_city_108524735 . $?l$ hasGeometry $?g$.
Filter WITHIN($?g$, "RECTANGLE(-90, 30, -80, 40)")

YAGO2.LL2
Select $?p$ Where $?p$ hasArea $?a$. $?p$ isLocatedIn $?l$. $?l$ hasGeometry $?g$.
Filter WITHIN($?g$, "RECTANGLE(-100, 30, -90, 40)")

Join queries used in the experiments

LGD.J1 (point-point)
Select $?s1$ $?s2$ Where $?s1$ type hotel . $?s1$ hasGeometry $?g1$. $?s2$ type hotel . $?s2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "0.003"

LGD.J2 (point-point)
Select $?s1$ $?s2$ $?l1$ $?l2$ Where $?s1$ name $?l1$. $?s1$ label $?b1$. $?s1$ type police . $?s1$ hasGeometry $?g1$. $?s2$ name $?l2$. $?s2$ label $?b2$. $?s2$ type police . $?s2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "0.01"

LGD.J3 (point-point)
Select $?s1$ $?s2$ Where $?s1$ name $?l1$. $?s1$ label $?b1$. $?s1$ type pub . $?s1$ hasGeometry $?g1$. $?s2$ name $?l2$. $?s2$ label $?b2$. $?s2$ type police . $?s2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "0.02"

LGD.J4* (polygon-polygon)
Select $?s1$ $?s2$ Where $?s1$ type park . $?s1$ hasGeometry $?g1$. $?s2$ type park . $?s2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "0.05"

LGD.J5* (point-polygon)
Select $?s1$ $?s2$ Where $?s1$ label $?b1$. $?s1$ type police . $?s1$ hasGeometry $?g1$. $?s2$ label $?b2$. $?s2$ type park . $?s2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "0.01"

LGD.J6* (point-line), **[EPS]** $\in \{0.01, 0.001, 0.0005\}$
Select $?s1$ $?s2$ Where $?s1$ label $?b1$. $?s1$ type hotel . $?s1$ hasGeometry $?g1$. $?s2$ label $?b2$. $?s2$ type road . $?s2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "**[EPS]**"

YAGO2.J1*
Select $?c1$ $?c2$ Where $?a1$ hasAirportCode $?c1$. $?a1$ hasGeometry $?g1$. $?a2$ hasAirportCode $?c2$. $?a2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "0.1"

YAGO2.J2*
Select $?p1$ $?p2$ Where $?p1$ hasGivenName $?gn1$. $?p1$ hasFamilyName $?fn1$. $?p1$ hasWonPrize $?pr1$. $?p1$ wasBornIn $?c1$. $?c1$ hasGeometry $?g1$. $?p2$ hasGivenName $?gn2$. $?p2$ hasFamilyName $?fn2$. $?p2$ hasWonPrize $?pr2$. $?p2$ wasBornIn $?c2$. $?c2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "0.1"

YAGO2.J3*
Select $?p$ $?c1$ $?c2$ Where $?p$ hasGivenName $?gn$. $?p$ hasFamilyName $?fn$. $?p$ actedIn $?m$. $?m$ isLocatedIn $?c1$. $?c1$ hasGeometry $?g1$. $?p$ wasBornIn $?c2$. $?c2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "0.1"

YAGO2.J4*
Select $?p1$ $?p2$ Where $?p1$ hasFamilyName $?fn1$. $?p1$ wasBornIn $?c1$. $?c1$ hasGeometry $?g1$. $?p1$ isMarriedTo $?p2$. $?p2$ wasBornIn $?c2$. $?c2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "0.1"

YAGO2.J5*
Select $?p$ Where $?p$ hasFamilyName $?fn$. $?p$ livesIn $?c1$. $?c1$ hasGeometry $?g1$. $?p$ worksAt $?c2$. $?c2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "0.1"

YAGO2.J6*
Select $?p1$ $?p2$ Where $?p1$ graduatedFrom $?u1$. $?u1$ hasGeometry $?g1$. $?p2$ actedIn $?m2$. $?m2$ isLocatedIn $?l2$. $?l2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "0.1"

YAGO2.J7*
Select $?p1$ $?p2$ Where $?p1$ graduatedFrom $?u1$. $?u1$ hasGeometry $?g1$. $?p2$ actedIn $?m2$. $?m2$ isLocatedIn $?l2$. $?l2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "0.1"

YAGO2.J8*, **[EPS]** $\in \{0.1, 0.01, 0.001\}$
Select $?p1$ $?p2$ Where $?p1$ worksAt $?w1$. $?w1$ hasGeometry $?g1$. $?p2$ worksAt $?w2$. $?w2$ hasGeometry $?g2$.
Filter DISTANCE($?g1$,$?g2$) < "**[EPS]**"