S-store: An Engine for Large RDF Graph Integrating Spatial Information

Dong Wang[†], Lei Zou [⋆†], Yansong Feng[†], Xuchuan Shen[†], Jilei Tian[‡], and Dongyan Zhao[†]

[†]Peking University, Beijing, China
[‡]Nokia Research Center GEL, Beijing, China
{WangD,zoulei,fengyansong,shenxuchuan,zhaody}@pku.edu.cn,
jilei.tian@nokia.com

Abstract. The semantic web data and the SPARQL query language allow users to write precise queries. However, the lack of spatial information limits the use of the semantic web data on position-oriented query. In this paper, we introduce spatial SPARQL, a variant of SPARQL language, for querying spatial information integrated RDF data. Besides, we design a novel index SS-tree for evaluating the spatial queries. Based on the index, we propose a search algorithm. The experimental results show the effectiveness and the efficiency of our approach.

Keywords: spatial query, RDF graph

1 Introduction

The Resource Description Framework (RDF)[11] is the W3C's recommendation as the basement of the semantic web. An RDF statement is a triple as $\langle subject, predicate, object \rangle$, which describes a property value of the subject. In real world, a large amount of RDF data are relevant to spatial information. For example, "London locatedIn England" describes a geographic entity London is located in a geographic location England; and the statement "Albert_Einstein hasWonPrize Nobel_Prize" is related to a geographic location of the event, i.e. Sweden.

Recently, researchers have begun to pay attention to the spatial RDF data. In fact, several real-world spatial RDF data sets have already been released, such as YAGO2[1][10], OpenStreetMap[2][9] etc. YAGO2[10] is an RDF data set based on Wikipidea and WordNet. Additionally, YAGO2 integrates GeoNames[3], which is a geographical database that contains more than 10 million geographical names, for expressing the spatial information of the entities and the statements.

Although the traditional spatial databases can manage spatial data efficiently, the "pay-as-you-go" nature of RDF enables spatial RDF data provide more flexible queries. Furthermore, due to the features of Linked Data, spatial RDF data sets are linked to other RDF repositories, which can be queried using both

⋆ Corresponding author. Email: zoulei@pku.edu.cn.
[1] http://www.mpi-inf.mpg.de/yago-naga/yago/downloads.html
[2] http://planet.openstreetmap.org/
[3] http://www.geonames.org/about.html

semantic and spatial features. Thus, the spatial information integrated RDF data is more suitable for providing location-based semantic search for users. For example, a user wants to find a physicist who was born in A rectangular area between 59°N 12°E and 69°N 22°E (this area is southern Germany), and won some academic award in a rectangular area between 48.5°N 9.5°E and 49.5°N 10.5°E (it is in Sweden). The query can be represented as a SPARQL-like query in the following. Section 4 gives the formalized definition.

```
SELECT  ?x  WHERE{
        ?x      wasBornIn       ?y  ?l1.
        ?x      hasWonPrize     ?z  ?l2.
        ?y      type_star       city.
        ?z      type            Academic_Prize.}
Filter{IN(?l1,[(59,12),(69,22)] AND IN(?l2,[(48.5,9.5),(49.5,10.5)]}
```

Few SPARQL query engines consider spatial queries, and to the best of our knowledge only two proposals exist in literature. Brodt et al. exploit RDF-3X [12] to build a spatial feature integrated query system [3]. They use GeoRSS GML[7] to express spatial features. The R-tree and RDF-3X indexes are used separately for filtering the entities exploiting the spatial and the RDF semantic features, respectively. Besides, the method only supports the range queries over the spatial entities. YAGO2 demo[4] provides an interface for SPARQL like queries over YAGO2 data. However, the system uses hard-coded spatial predicates on spatial statements. Different from the above approaches, we introduce a hybrid index integrating both the spatial and semantic features, and the range queries and spatial joins are both supported in our solution.

In this paper, we introduce the spatial query over the RDF data, a variant of the SPARQL language for integrating the spatial feature constraint such as the range query and the spatial join. The spatial constraints assert the corresponding entities or events located in an absolute location area or near some entities in the query. For instance, users could search for a football club founded before 1900 nearby London, or a park nearby a specific cinema.

For effectively and efficiently solving the spatial queries, we introduce a tree-style index structure (called SS-tree). The SS-tree index is a hybrid tree-style index integrating the semantic features and the spatial features. Based on SS-tree, we introduce a list of pruning rules that consider both spatial and semantic constraints in the query, and propose a top-down searching algorithm. The tree nodes dissatisfying the signature constraints or the spatial constraints are safely filtered, and the subtrees rooted on the nodes are pruned. We make the following contributions in this paper.

1. We formalize the spatial queries, a variant of SPARQL language, on the RDF data integrating the spatial information. Besides, we introduce two spatial predicates: the range query predicate and the spatial join predicate. The spatial queries can express both spatial constraints and semantic constraints.
2. We classify the entities into two categories: the spatial entities and the non-spatial entities. Based on these two categories, we build a novel tree-style index integrating the spatial features and semantic features. Additionally, we

[4] http://www.mpi-inf.mpg.de/yago-naga/yago/demo.html

introduce a list of pruning rules for efficiently exploiting the spatial feature and the semantic feature of the index.
3. We evaluate our approach on a large real-world data set YAGO2, and the result shows our approach outperforms the baseline by several orders of magnitude.

2 Related Work

Many RDF manage systems [1, 17, 18, 12, 16, 2, 4] have been proposed in the past years. RDF-3x[13], Hexstore[16] and gStore[19] are the state-of-the-art RDF manage systems. Since none of the systems takes spatial feature into consideration, all the systems are unsuitable for spatial RDF data management.

Brodt et al.[3] and Virtuoso[6] utilize RDF query engines and spatial index to manage spatial RDF data. [3] uses RDF-3x as the base index, and adds an spatial index for filtering entities before or after RDF-3x join operations. These two approaches only support range query (and spatial join[6]) on entities, and the spatial entities follow the GeoRSS GML[14] model. YAGO2 demo employs six (hard coded) spatial predicates "northOf", "eastOf", "southOf", "westOf", "nearby" and "locatedIn" over statements. Users can construct queries as a list of triple patterns with the spatial predicates. Other spatial queries are not supported. The technical detail and the performance are not reported.

3 Preliminary

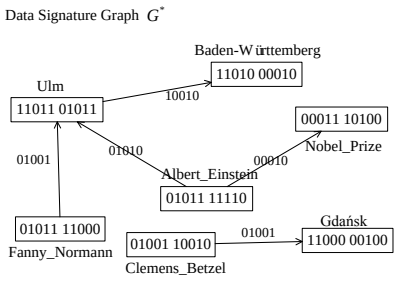


Fig. 1. Signature Graph

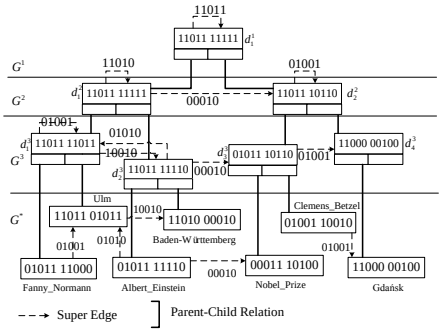


Fig. 2. VS-tree

Since our method is based on our previous work gStore system and VS-tree index[19], to make the paper be self-contained, this section provides a simple overview of gStore and VS-tree. More details can be found at [19]. By encoding each entity and class vertex into a bit string called *signature*, gStore transforms an RDF graph into a data signature graph. Then, a tree-style index VS-tree is proposed over the data signature graph. The VS-tree is an extended S-tree[5]. The nodes on the same level and the corresponding edges constitute a signature graph. A pruning rule for subgraph query over the data signature graph is proposed for executing SPARQL queries.

The signature sig of each subject s depends on all the edges $\{e_1, e_2, \ldots, e_n\}$ adjacent to s. For each e_i, gStore uses a list of hash functions to generate a signature $sig.e_i$, where the front N bits denote the predicate, and the following M bits denote the object. The valid bits depend on the hash code of the corresponding textual information. To determine the valid bits, gStore exploits several hash functions. The signature sig of s follows $sig = sig.e_1|sig.e_2|\ldots|sig.e_n$.

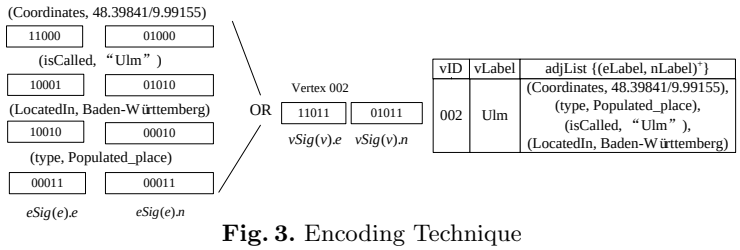


Fig. 3. Encoding Technique

For example, in Figure 4, there's four edges starting from *Ulm* (#8, #9, #10 and #11). Suppose that we set the first five bits for the predicate and the following five bits for the object, we can get four signatures 1100001000, 1000101010, 1001000010 and 0001100011 corresponding to the four edges. Thus, *Ulm* can be represented as 1101101011. Figure 3 shows the encoding processing for "Ulm". Figure 1 shows the signature graph of Figure 4. Note that only the entity and class vertices in the RDF graph are encoded.

After the signature graph is generated, the VS-tree is built by inserting nodes into VS-tree sequentially. The corresponding VS-tree is shown in Figure 2.

4 Problem Definition

We formally define the spatial RDF and spatial SPARQL query as follow.

Definition 1. *An entity e is called a* spatial entity *if it has an explicit location labeled with the coordinates x and y (for the two-dimensional situation). The other entities are called the non-spatial entities.*

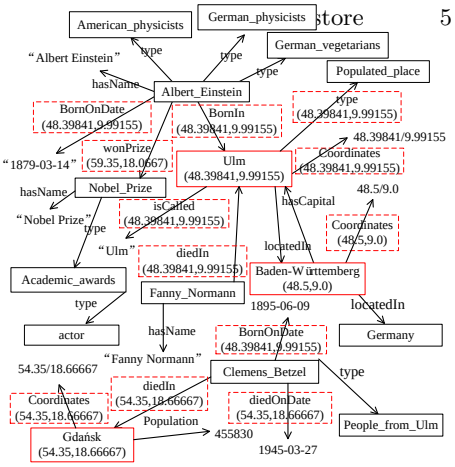
Definition 2. *A statement is a four-tuple* $\langle s, p, o, l \rangle$*, where s, p, o and l represent for subject, predicate, object and location, respectively. The location feature denotes the location where the statement happens. Note that l can be null. If the l of a statement is not null, the statement is called a* spatial statement*. Otherwise, it's called a* non-spatial statement*. A collection of statements (including spatial and non-spatial statements) is called a* spatial RDF data set*.*

Definition 3. *A* spatial triple pattern *is a four-tuple* $\langle s, p, o, l \rangle$*, where s, p, o and l represent for subject, predicate, object and location respectively. Each item can be a variable. Note that if l is not a variable, it should be omitted.*

Definition 4. *A* spatial query *is a list of spatial triple patterns with some spatial filter conditions. If there's no spatial filter condition, the spatial query is reduced to a traditional SPARQL query.*

	Subject	Predict	Object	Location(x,y)
#1	Albert_Einstein	BornIn	Ulm	(48.39841,9.99155)
#2	Albert_Einstein	BornOnDate	1879-03-14	(48.39841,9.99155)
#3	Albert_Einstein	type	American_physicists	
#4	Albert_Einstein	type	German_physicists	
#5	Albert_Einstein	type	German_vegetarians	
#6	Albert_Einstein	WonPrize	Nobel_Prize	(59.35,18.0667)
#7	Albert_Einstein	hasName	"Albert Einstein"	
#8	Ulm	Coordinates	48.39841/9.99155	(48.39841,9.99155)
#9	Ulm	isCalled	"Ulm"	(48.39841,9.99155)
#10	Ulm	LocatedIn	Baden-W ürttemberg	
#11	Ulm	type	Populated_place	(48.39841,9.99155)
#12	Baden-W ürttemberg	hasCapital	Ulm	
#13	Baden-W ürttemberg	Coordinates	48.5/9.0	(48.5,9.0)
#14	Baden-W ürttemberg	LocatedIn	Germany	
#15	Nobel_Prize	type	Academic_awards	
#16	Nobel_Prize	hasName	"Nobel Prize"	
#17	Fanny_Normann	diedIn	Ulm	(48.39841,9.99155)
#18	Fanny_Normann	hasName	"Fanny Normann"	
#19	Fanny_Normann	type	actor	
#20	Clemens_Betzel	diedIn	Gdańsk	(54.35,18.66667)
#21	Clemens_Betzel	diedOnDate	1945-03-27	(54.35,18.66667)
#22	Clemens_Betzel	BornOnDate	1895-06-09	(48.39841,9.99155)
#23	Clemens_Betzel	type	People_from_Ulm	
#24	Gdańsk	Coordinates	54.35/18.66667	(54.35,18.66667)
#25	Gdańsk	Population	455830	

(a)



(b)

Fig. 4. Spatial RDF Graph

Figure 4(a) shows a subset of a spatial RDF data set. *Ulm*, *Baden-Württemberg* and *Gdańsk* are spatial entities, and some statements are spatial statements, such as #1, #2 and #6. Besides, there're a lot of non-spatial entities and non-spatial statements. For example, people have no spatial information, since we can't locate a person on the map. Similarly, the statements like $\langle People\ hasName\ Name\rangle$ are non-spatial statements. In S-store, we use "spatial predicate" to represent the spatial queries. In this stage, we support the range query and the spatial join semantics. In practice, we use $sl(?x)$ for denoting the spatial label of variable $?x$. Besides, $dist(a,b)$[5]$< r$ denotes the distance between a and b should below the threshold r, where a and b should be a specific location or a variable. If either of a and b is a constant, the query is called a *range query*. If both of a and b are variables, the query is called a *spatial join query*. Note that a spatial query can be a range query and a spatial join query at the same time. The following query examples Q_1 and Q_2 demonstrate the range query and the spatial join query respectively. The former one queries a person who died in a popular place near coordinates (48.39841,9.99155), and the latter one queries two people where the first person died near the place where the second person died.

Q1:

```
Select  ?x  Where
{?x     diedIn      ?y.
?y      type        Populated_place.
} Filter{dist(sl(?y),(48.39841,9.99155))<1}
```

Q2:

```
Select  ?x1,?x2     Where
{?x1    diedIn      ?y1     ?l1.
?x2     diedIn      ?y2.
} Filter{dist(sl(?l1),sl(?y2))<1}
```

[5] In this paper, for the ease of the presentation, we adopt the Euclidean distance between two locations. Actually, we can use "the earth's surface distance" to define the distance between two locations based on latitudes and longitudes.

The spatial RDF data set and the spatial query can be also modeled as graphs (Definitions 5 and 6). The query processing is to find the matches (Definition 7) of a spatial query graph Q in a spatial RDF data graph G. Figure 4(b) shows the graph corresponding to the spatial RDF data set in Figure 4(a), where the spatial entities and the spatial statements are all surrounded by the red rectangles.

Definition 5. *The spatial RDF data graph is denoted as* $G = \langle V, E, L_V, L_E, S_V, S_E \rangle$*, where*

(1)$V = V_l \cup V_e \cup V_c \cup V_b$ *denote all RDF vertexes where* V_l*,* V_e *and* V_c *are the sets of literal vertices, entity vertices, class vertices and blank nodes respectively.*

(2) E *is the collection of the edges between vertices.*

(3) $L_V = \{URI\} \cup \{LiteralValue\}$ *is the collection of text label of each vertex, where* $v \in \{V_e \cup V_c\} \leftrightarrow label(v) \in \{URI\}$ *and* $v \in V_l \leftrightarrow label(v) \in \{LiteralValue\}$*. For* $v \in V_b$*,* $label(v) = \phi$*.*

(4)L_E *is the collection of edge labels, i.e., all predicates plus null value.*

(5)S_V *and* S_E *represent the spatial labels of* V *and* E *respectively, where the spatial labels denote where the entity locates (the event happens) in, i.e., the latitude and longitude (only valid for spatial entities and spatial statements).*

Definition 6. *The spatial RDF query graph is denoted as* $G = \langle V, E, L_V, L_E, SC_V, SC_E \rangle$*, where*

(1)$V = V_l \cup V_e \cup V_c \cup V_b \cup V_p$*, where* V_p *denotes the parameter vertices, and* V_l*,* V_e*,* V_c *and* V_b *are the same as in Definition 5.*

(2)E *and* L_E *are the same as in Definition 5.*

(3) L_V *is the same as in Definition 5. For* $v \in V_p$*,* $label(v) = \phi$*.*

(4)SC_V *and* SC_E *represent the spatial constraints of* V *and* E *respectively, where the spatial constraints can be an absolute area or the relative position for some parameter.*

Definition 7. *Consider a spatial RDF graph* G *and a spatial query graph* Q *with* n *vertices* $\{v_1, \ldots, v_n\}$*. A set of* n *distinct vertices* $\{u_1, \ldots, u_n\}$ *in* G *is said to be a match of* Q *iff. the following conditions hold:*

1. If $v_i \in \{V_l \cup V_c \cup V_e\}$*,* $u_i \in \{V_l \cup V_c \cup V_e\}$ *and* $label(v_i) = label(u_i)$*;*

2. If $v_i \in V_b$*, there is no constraint over* u_i*;*

3. If $v_i \in V_p$*, the spatial label* $S(u_i)$ *must satisfy the spatial constraint* $SC(v_i)$*;*

4. If there is an edge $\overline{v_i v_j}$ *from* v_i *to* v_j *in* Q *, there is also an edge* $\overline{u_i u_j}$ *from* u_i *to* u_j *in* G*. If* $\overline{v_i v_j}$ *has predicate* p *and spatial constraint* $SC(\overline{v_i v_j})$*,* $\overline{u_i u_j}$ *must have the same predicate* p *and spatial label* $S(\overline{u_i u_j})$ *satisfy* $SC(\overline{v_i v_j})$*.*

5 Overview of S-store

S-store employs a hybrid index that integrates both R-tree[8] and VS-tree[19]. Therefore, the pruning strategies of R-tree and gStore are also integrated as the searching strategy for S-store. Our framework consists of the pre-processing, the index construction and the query processing stages.

In the pre-processing stage, we first encode each vertex and edge as a bit string (we call it a *signature*). The encoding technique is shown in Section 3, and more information can be found in [19]. Subsequently, we build the *spatial signature graph* $G^\star$. Figure 5 shows a running example.

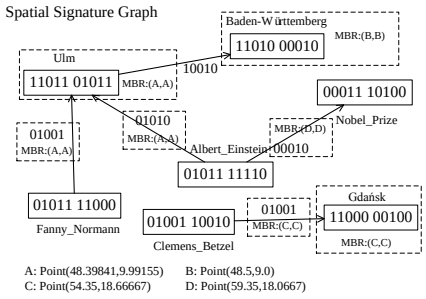


Fig. 5. Spatial Signature Graph

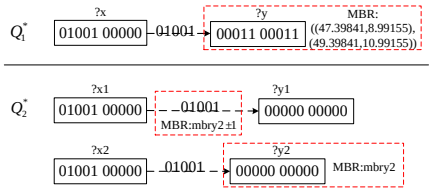


Fig. 6. Q1 and Q2

In the index construction stage, we construct a tree-style index based on the spatial signature graph for effectively reducing the search space. The index is called *SS-tree*. Figure 7 shows an running example. The nodes on the same level of the SS-tree form a spatial signature graph. If there's a match of a query Q in a lower spatial signature graph, there must be a corresponding match in each higher spatial signature graph. Therefore, we need to guarantee that SS-tree is a height-balanced tree.

In the query processing stage, given a query graph Q, we first convert Q into the *spatial signature query graph* $Q^\star$ as in the pre-processing stage. Figure 6 shows the spatial signature query graphs of the Q_1 and the Q_2. Note that, if there is a set of vertices in G matches a query graph Q, there must be a corresponding match in $G^\star$ of $Q^\star$. Subsequently, we implement a top-down searching algorithm over SS-tree to find the matches of $Q^\star$ in $G^\star$. At last, we retrieve the corresponding textual result and return it to the user.

Definition 8. *Given a spatial signature graph $G^\star$ and a spatial signature query graph $Q^\star$ with n signature vertices $\{q_1, \ldots, q_n\}$, a set of distinct signature vertices $\{sig_1, \ldots, sig_n\}$ in $G^\star$ is a match of $Q^\star$ iff. the following conditions hold:*

1. $\forall q_i$, $sig_i.signature \& q_i.signature = q_i.signature$;

2. $\forall q_i$, the spatial label $S(sig_i)$ must satisfy the spatial constraint $SC(q_i)$;

3. If there is an edge $\overline{q_iq_j}$ from q_i to q_j in $Q^\star$, there is also an edge $\overline{sig_isig_j}$ from sig_i to sig_j in $G^\star$, and $\overline{q_iq_j}.signature \& \overline{sig_isig_j}.signature = \overline{q_iq_j}.signature$. If $\overline{q_iq_j}$ has spatial constraint $SC(\overline{q_iq_j})$, $\overline{sig_isig_j}$ must have the spatial label $S(\overline{sig_isig_j})$ satisfy $SC(\overline{q_iq_j})$.

6 Index Construction

In this section, we would introduce our spatial RDF index SS-tree. The index is presented as a tree-style. Generally speaking, we build the SS-tree based on VS-tree in gStore. The difference between S-store and gStore is that S-store can answer spatial queries.

6.1 Spatial Signature Graph Generation

First, we convert a data graph into a spatial signature data graph before building SS-tree. Since the spatial signature data graph can be regarded as a signature graph including spatial features, we generate the signature graph as described in Section 3. Then, for each vertex (v_i) and each edge e_j, we set the $MBR(v_i)$ and the $MBR(e_j)$. The signature and the MBR features of the spatial signature

data graph are used to compute the features of the tree nodes on the high level. the unsatisfied tree nodes can be filtered early to save the space and time cost. Due to the space limit, the detailed information is omitted.

6.2 SS-tree Construction

The entities can be separated into two parts based on the spatial features. C_1 is the non-spatial entity set, and C_2 is the spatial entity set. For example, in Figure 4, collection C_1 = {Albert_Einstein, Fanny_Normann, Clemens_Betzel, Nobel_Prize} and collection C_2 = {Ulm, Baden-Württemberg, Gdańsk} respectively.

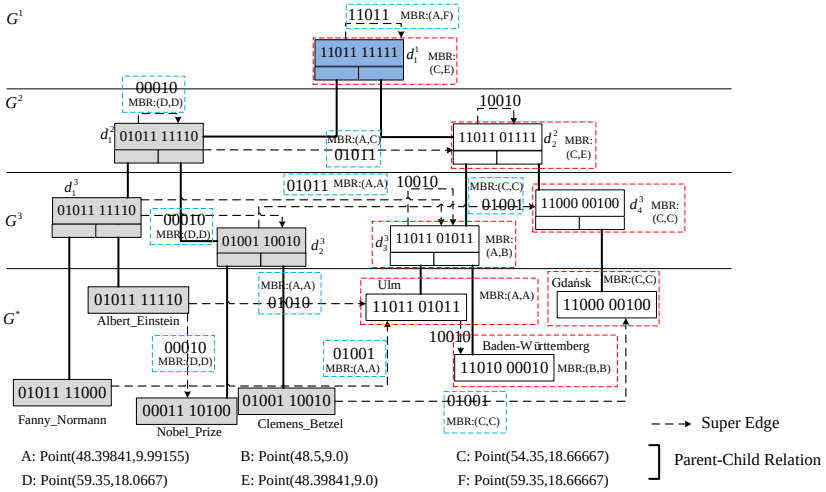


Fig. 7. SS-tree

Based on C_1 and C_2, we can generate two induced spatial signature graphs $G_1^\star$ and $G_2^\star$ from $G^\star$. The induced graph $G_i^\star$ can be composed into $V_i^\star$ and $E_i^\star$, where $V_i^\star$ =the vertices corresponding to C_i, and $E_i^\star = \{\overline{v_k v_l} | v_k \in V_i^\star \wedge v_l \in V_i^\star\}$.

In the following, we use "compute the features" to denote the bottom-up feature constructing process. The process obeys the following rule:

– *SS-tree Rule:* Consider two spatial signature nodes v_1,v_2 and their father nodes n_1, n_2. The following conditions hold:
(1) $n_1.sig \& v_1.sig = v_1.sig$, $n_2.sig \& v_2.sig = v_2.sig$;
(2) $v_1.MBR \in n_1.MBR$, $v_2.MBR \in n_2.MBR$;
(3) If there's an edge $\overline{v_1 v_2}$ between v_1 and v_2, there must be an edge $\overline{n_1 n_2}$ between n_1 and n_2, where $\overline{n_1 n_2}.sig \& \overline{v_1 v_2}.sig = \overline{v_1 v_2}.sig$ and $\overline{v_1 v_2}.MBR \in \overline{n_1 n_2}.MBR$, even if $n_1 = n_2$.

Non-spatial Entities For collection C_1, we build a VS-tree over $G_1^\star$. The VS-tree constructing method can be found in [19]. After the VS-tree's completed, we compute the MBR features for each edge based on the SS-tree rule (2). T_1 consists of the VS-tree and the spatial features. For example, in Figure 7, the gray nodes and the edges between them compose the sub-SS-tree T_1. The edge $\overline{d_1^3 d_2^3}$ has the spatial feature because the edge between the child node *Albert_Einstein* of d_1^3 and the child node *Nobel_Prize* of d_2^3 owns a spatial feature.

Spatial Entities For entity collection C_2, we build a R-tree over $G_2^\star$. Based on the R-tree structure, we first compute the features of the R-tree vertices base on SS-tree rule (1). Then, we add the edges between the upper level vertices of the R-tree and compute the features of the added edges based on the SS-tree rule (3). For example, in Figure 7, the white nodes and the edges between them compose the sub-SS-tree T_2. The signature of the node d_3^3 1101101011 = 1101101011|1101000010, where the former signature belongs to the node "Ulm" and the latter signature belongs to the node Baden-Württemberg, and "Ulm" and Baden-Württemberg are the children of d_3^3.

Combination Since SS-tree should be a height-balanced tree, we should modify the tree height to balance T_1 and T_2. We employ the following operation called one step growth to increase tree height. Given a tree T, we add a node n as the father of T's root, and n would be the new root of T.

Given trees T_1 and T_2, we grow the lower tree with several steps to ensure the tree heights are equal. And then, we add a node n as new tree T's root, and set the roots of T_1 and T_2 as n's children. The new tree is called T_3.

Based on T_3, we first add all edges $e_{ij} = \{\overline{v_i v_j} | v_i \in G_1^\star \wedge v_j \in G_2^\star\}$ between $G_1^\star$ and $G_2^\star$, and then add the corresponding edges and compute the features for the edges based on the SS-tree rule (3). For example, the edge between "Albert_Einstein" and "Ulm" is added in this stage, and the corresponding edges $\overline{d_1^3 d_3^3}$ and $\overline{d_1^2 d_2^2}$ are added subsequently.

7 Query Processing

Given a spatial query Q, we first convert the Q to a spatial signature graph $Q^\star$. The converting processing consists of three steps.

(1) Encode the triple patterns as described in Section 6.1.
(2) For each range query predicate, we add the corresponding absolute MBR on the specific variables.
(3) For each spatial join predicate, we add the relevant MBRs on the variables.

The $Q_1^\star$ and $Q_2^\star$ corresponding to Q_1 and Q_2 are shown in Figure 6. The signatures are generated as G to $G^\star$, where the variables contribute no valid bit. The range query predicate of $Q1$ is converted to the absolute MBR binding $?y$ in $Q_1^\star$, and the spatial join predicate of Q_2 is converted to the relevant MBRs in $Q_2^\star$.

After the corresponding $Q^\star$ is generated, we next search the matches of $Q^\star$ in $G^\star$ exploiting the SS-tree. Consider a spatial signature query graph $Q^\star = \{q_1, \ldots, q_n\}$, we first generate the node candidate set $NodeSet_i$ for each variable q_i, and then verify each candidate in the query candidate set $QSet = \{NodeSet_1 \times \ldots \times NodeSet_n\}$ to generate the matches of $Q^\star$ in $G^\star$. At last, we generate the matches of Q in G based on the matches of $Q^\star$.

7.1 Pruning Rules

For efficiently generating the node candidate set, we have the following five pruning rules. Pruning rule 1 is based on the fact that only spatial entities can be bound by spatial predicates. Pruning rules 2 and 3 are based on that if the distance between v_1 and v_2 is no less than the distance between v_i and v_j where v_i and v_j are the descendant of v_1 and v_2 respectively. Pruning rule 4 is based

on that $v.sig\&v_i.sig = v_i.sig$ if v_i is the descendant of v. Pruning rule 5 is based on that if there's no satisfied edge between v_1 and v_2, there's no satisfied edge between v_i and v_j where v_i and v_j are the descendant of v_1 and v_2 respectively. Due to the space limit, we can't state the pruning rules in detail.

Pruning Rule 1 If a variable is bound with a spatial predicate, the subtree T_1 induced by C_1 can be pruned safely.

Pruning Rule 2 Consider a variable v bound with a range query predicate, if there is a tree node n where $v.mbr$ has no intersection with $n.mbr$, the subtree rooted on n can be pruned safely.

Algorithm 1 Query Processing

Require: $Q^\star = \langle v_1, \ldots, v_n \rangle$, SS-tree T, root r of T, signature data graph $G^\star$.
Ensure: The node candidate sets $\{NodeSet\}$ of nodes of $Q^\star$ in $G^\star$.

```
1: Set each NodeSet_i = r //initialize the node candidate set.
2: while true do
3:   if ∀NodeSet_i ∈ G* then
4:     return {NodeSet} //the sets contains real data points.
5:   for all NodeSet_i do
6:     NodeSet_i =the children of each node n_i ∈ NodeSet_i
7:     Set MBR_i = ⋃{n|n ∈ NodeSet_i}
8:   for all node n_i ∈ NodeSet_i do
9:     if n_i ∈ T_1 ∧ v_i is binding then
10:      remove n from tempNodeSet // pruning rule 1.
11:    if v_i.sig&n_i.sig! = n_i.sig then
12:      remove n from tempNodeSet //pruning rule 2.
13:    if v_i is bound by range query predicates then
14:      if intersection(v_i.mbr, n_i.mbr) = φ then
15:        remove n from tempNodeSet //pruning rule 3.
16:    if ∃e = v_i v_j then
17:      if n_i.neighbour ∩ NodeSet_j = φ then
18:        remove n from tempNodeSet //pruning rule 4.
19:    if dist(v_i, v_j) <= l then
20:      if dist(n_i, MBR_j) > l then
21:        remove n from tempNodeSet //pruning rule 5.
```

Pruning Rule 3 Consider two variables v_i and v_j bound by a spatial join predicate, and $NodeSet_i$ is the candidate set of v_i and $NodeSet_j$ is the candidate set of v_j. Suppose the max distance is set to be $MaxDist$. Let $n_i \in NodeSet_i$, if the distance from MBR of n_i to any node $n_j \in NodeSet_j$ is larger than $MaxDist$, n_i can be safely pruned.

Pruning Rule 4 Consider a variable v, if there is a tree node n where $v.sig\&n.sig! = n.sig$, the subtree rooted on n can be pruned safely.

Pruning Rule 5 Consider two linked variables v_i and v_j with an edge $e = \overline{v_i v_j}$ from v_i to v_j, and $NodeSet_i$ is the candidate set of v_i and $NodeSet_j$ is the candidate set of v_j in the same spatial signature graph. Let $n_i \in NodeSet_i$, if there's no edge from n_i to any node $n_j \in NodeSet_j$, n_i can be safely pruned. What's more, if there's a range predicate on e, the unsatisfied edges are considered nonexistent. The pruning rule is based on the fact that if there's no satisfied edge from n_i to any node $n_j \in NodeSet_j$, there's no satisfied edge from the descendants of n_i to any descendants of the $n_j \in NodeSet_j$.

Algorithm 1 describes the top-down node candidate sets generating process. The use of the pruning rules is shown in Line 9-21.

7.2 Verification

Consider the node candidate set $\{NodeSet\}$, we generate a list of nodes $\langle n_1, \ldots, n_n \rangle$ from each item of $\{NodeSet\}$ respectively, and verify if $\langle n_1, \ldots, n_n \rangle$ forms the connected regions corresponding to the connected regions in $Q^\star$. If $\langle n_1, \ldots, n_n \rangle$ can form, we consider it as a match candidate of $Q^\star$, or we discard it otherwise. The generating process can be realized by employing a BFS algorithm starting from the smallest node candidate sets in each connected region. For example, $Q_2^\star$ has two connected regions. Since $?x_1$ and $?x_2$ have the highest selectivity in each connected region respectively, the $NodeSet_{x_1}$ and $NodeSet_{x_2}$ are selected as the start points. Then, we run BFS from $NodeSet_{x_1}$ and $NodeSet_{x_2}$. If there's an edge $e = \overline{v_k v_l}$ in $Q^\star$, there must be an corresponding edge.

Algorithm 2 *Varification*

Require: node candidates $\{NodeSet\}$, $Q^\star = \langle v_1, \ldots, v_n \rangle$, Q, G.
Ensure: the matches $\{M\}$ of Q.
1: Set the match candidate list of $Q^\star$ $L = \phi$.
2: **for** each connected region $Q_i \subseteq Q^\star$ **do**
3: Select the $NodeSet_j$ with the smallest size in Q_i.
4: Set the Q_i's match candidate set $M_c^i = \phi$. //Initialize the match candidate sets.
5: **for** each node $n_k \in NodeSet_j$ **do**
6: Run the BFS process from n_k.
7: **if** $\exists$match candidate m_c^i of Q_i **then**
8: $M_c^i.add(m_c^i)$. //If all edges are valid, it's a match candidate.
9: Set $M_c^\star = M_c^1 \times \ldots \times M_c^k$. //The match candidates of $Q^\star$.
10: Set $M^\star = \phi$. //The matches of $Q^\star$.
11: **for** each $m_c^\star \in M_c^\star$ **do**
12: **if** all spatial join predicates are valid on $m_c^\star$ **then**
13: $M^\star.add(m_c^\star)$.
14: Set $M = \phi$. //The matches of Q.
15: **for** each $m^\star \in M^\star$ **do**
16: Get the subgraph $m \subseteq G$ corresponding to $m^\star$.
17: **if** all literal constraints are valid on m **then**
18: $M.add(m)$.
19: return M.

Given a match candidate $Q_c^\star$ of $Q^\star$, we verify if all the spatial constraints are satisfied. The satisfied match candidates are the matches of $Q^\star$. Subsequently, since the encoding technique may bring false positive error, we verify if all edges in Q are satisfied given a match of $Q^\star$. The valid candidates are the matches of Q. Then, the matches of Q is returned to users.

8 Experiments

To the best of our knowledge, only YAGO2 Demo and the system implemented by A. Brodt et al.[3] (DisRDF for short) are available spatial RDF data management system. YAGO2 Demo only accepts range queries over spatial statements based on several hard-coded spatial predicates. DisRDF models the spatial entities with various shapes and only accepts range queries over the spatial entities. Since we support both range query and spatial join semantic over the spatial entities and the spatial statements, the comparisons to other approaches are not applicable. Thus, we focus on the performance and the specific characteristics of our approach.

8.1 Data Set & Setup

Data Set YAGO2 is a real data set based on Wikipedia ,WordNet and GeoNames. The latest version of YAGO2 have more than 10 million entities and 440 million statements. We obtain a spatial RDF data set from YAGO2 by removing some statements that describe the date when another statement is extract or the URL where another statement is extract from. The condensed data set has more than 10 million entities/classes and more than 180 million statements. More than 7 million entities are spatial entities, and more than 90 million statements are spatial statements.

Queries & Setup In order to evaluate our approach, we manually generate 10 sample spatial SPARQL queries that have different features. The sample queries are divided into 5 classes, i.e., A, B, C, D, E. The queries in set A are star queries with the range query predicates over the entities. The queries in set B are the queries with the range query predicates over the entities. The queries in set C are the queries with spatial join predicates over entities. The queries in set D are the queries with range query and spatial join predicates over statements. The queries in set E are combined queries. The queries are given in our technical report [15].

	A1	A2	B1	B2	C1	C2	D1	D2	E1	E2
Spatial Queries	3	1177	1	10	18	25	2	23	7	12
SPARQL Queries	10,137,491	8,567	36	50	36	50	36	50	40	50

Table 1. The Result Set Size of Queries

Table 1 shows the selectivity of each query. In order to show the inefficiency of post-processing method (i.e., finding SPARQL query results by ignoring the spatial constraints and then verifying the candidates by the spatial predicates), we also report the result sizes of all queries discarding the spatial constraints. We run all queries on a PC with an Intel Xeon CPU E5645 running at 2.40 GHz and 16 GB main memory.

Node Capability	Index Size(MB)	Tree Height	Node Count
30	5,537	6	571,064
50	4,376	5	341,905
100	3,342	4	170,121
150	2,938	4	113,365

Table 2. Statistics of Node Capability

Index Style	Node Capability	Index Size(MB)	Tree Height	Node Count
SS-tree	100	3,342	4	170,121
	150	2,938	4	113,365
VS-tree+	100	4,332	3	204,890
	150	3,990	3	138,931

Table 3. Statistics of Tree-Construction

8.2 Evaluating Node Capability

In this subsection, we evaluate whether the different *node capabilities* (i.e., the maximal number of child nodes of each node in the tree index) affect the off-line and the on-line performance. In the evaluation, the node capabilities are set to 30, 50, 100, 150 respectively.

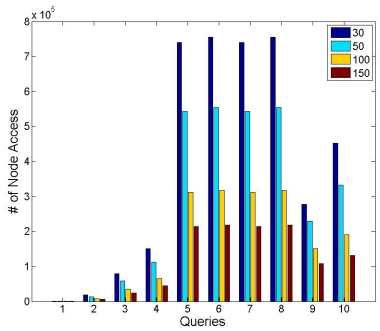


Fig. 8. Node Capability - Nodes Access

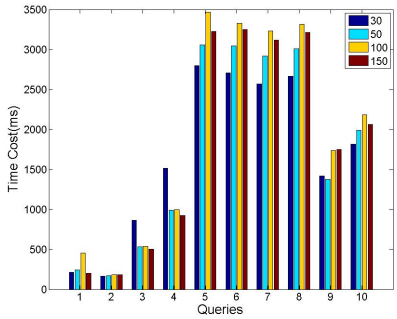


Fig. 9. Node Capability - Time Cost

Table 2 shows the storage cost of SS-tree with different node capabilities. Obviously, lower node capability leads to larger node count, higher tree height and larger storage requirement, vice versa. Figure 8 shows the count of node access during search. Clearly, the count of node access depends on the capability of each node. Note that the count of data points involved during search = # of accessed nodes × node capability, which means the operation count on data points may be lower in the lower node capability situation. Figure 9 reports the query time cost of each query. The query time cost is proportional to the count of data points involved during search in most cases.

8.3 Evaluating Entity Organization

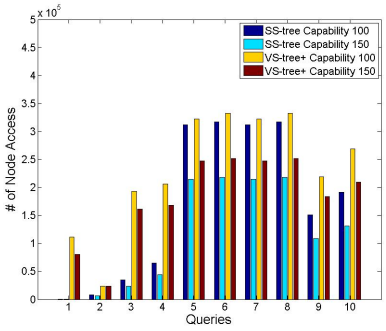


Fig. 10. SS-tree - Nodes Access

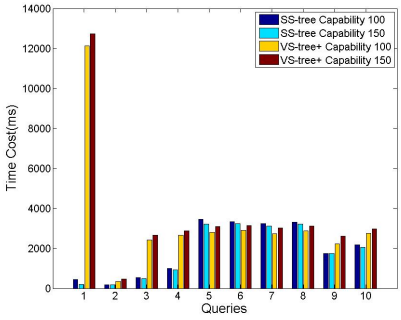


Fig. 11. VS-tree+ - Time Cost

In this subsection, we evaluate whether the different entity organization styles affect the off-line and the on-line performance. We compare the SS-tree and the VS-tree plus spatial features (denoted as VS-tree+). Table 3 shows the results. The SS-tree demands lower storage space than VS-tree+ when the node capabilities are the same. Figure 10 and 11 show the count of nodes accessed and the time cost of each query. As we supposed, SS-tree works better for spatial feature filtering. Since SS-tree organize spatial entities in a R-tree, SS-tree works better on query 1, 2, 3, 4, 9 and 10, where the queries have spatial predicates on nodes.

8.4 Evaluating Performance

For evaluating the efficiency of our approach, we implement a baseline approach based on the method of [3]. The baseline approach adopts the post-processing solution, running SPARQL queries by ignoring the spatial predicates and then refining the candidates by considering the spatial constraints. In this subsection, we make a comparison between S-store and the baseline approach.

In practice, the baseline approach exploits gStore[19] as the RDF management system, and the node capability is set to 150. Besides, the MySQL is used to retrieve the coordinates of the entities and the statements.

The query response times are shown in Table 4, where G-store+ denotes the baseline approach. Since A1 and A2 have many candidate results (see Table 1), the time cost of the baseline is unacceptable. We can't get the results of A1 in reasonable time (more than half an hour), and the time cost for A2 is about 113 seconds. However, our approach (S-store) can answer the query A1 and A2 in 213 and 165 milliseconds, respectively. Although the other queries have just a few candidate results without spatial predicates, S-store still outperforms the baseline approach.

	Time Cost (ms)									
	A1	A2	B1	B2	C1	C2	D1	D2	E1	E2
S-store	213	165	863	1,518	2,800	2,710	2,571	2,668	1,418	1,816
G-store+	>30min	112,406	5,894	9,555	4,478	4,127	3,624	6,750	5,839	3,779
Speed-up Ratio		99.8%	85.4%	84.1%	37.5%	34.3%	29.1%	60.5%	75.7%	51.9%

Table 4. The Performance comparison

9 Conclusions

In this paper, we introduce spatial queries, a variant of SPARQL language, for querying RDF data with spatial features. Spatial queries employ spatial predicates for expressing the range query and the spatial join constraints. Besides, we introduce a novel index called SS-tree for evaluating the spatial queries. Based on SS-tree, we propose several pruning rules and a searching algorithm. The experimental results show the effectiveness and the efficiency of our approach. The spatial queries just cost a few seconds on YAGO2 data set, which has more than 10 million entities and 180 million statements.

10 Acknowledgments.

Dong Wang and Lei Zou were supported by NSFC under Grant No.61003009. Xuchuang Shen and Dongyan Zhao were supported by NSFC under Grant No.61272344

and National High Technology Research and Development Program of China under Grant No. 2012AA011101. Lei Zou's work was partially supported by State Key Laboratory of Software Engineering(SKLSE), Wuhan University, China.

References

1. D. J. Abadi, A. M. 0002, S. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *VLDB J.*, 18(2), 2009.
2. D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.
3. A. Brodt, D. Nicklas, and B. Mitschang. Deep integration of spatial query processing into native rdf triple stores. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 33–42. ACM, 2010.
4. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *ISWC*, 2002.
5. U. Deppisch. S-tree: A dynamic balanced signature index for office retrieval. In *SIGIR*, 1986.
6. O. Erling and I. Mikhailov. Rdf support in the virtuoso dbms. In *Networked Knowledge - Networked Media*.
7. G. Gröger, T. Kolbe, A. Czerwinski, and C. Nagel. Opengis city geography markup language (citygml) encoding standard. *Open Geospatial Consortium Inc. Reference number of this OGC® project document: OGC*, 2008.
8. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.
9. M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *Pervasive Computing, IEEE*, 7(4):12–18, 2008.
10. J. Hoffart, F. Suchanek, K. Berberich, and G. Weikum. Yago2: a spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 2012.
11. G. Klyne, J. Carroll, and B. McBride. Resource description framework (rdf): Concepts and abstract syntax. *W3C recommendation*, 10, 2004.
12. T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, 1(1), 2008.
13. T. Neumann and G. Weikum. x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases. *PVLDB*, 1(1), 2010.
14. R. Singh, A. Turner, M. Maron, and A. Doyle. Georss: Geographically encoded objects for rss feeds, 2008.
15. 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 *http* : $//www.icst.pku.edu.cn/intro/leizou/TR/2013/TR-DB-ICST-PKU-2013-001.pdf$, 2013.
16. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1), 2008.
17. K. Wilkinson. Jena property table implementation. In *SSWS*, 2006.
18. K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient rdf storage and retrieval in jena2. In *SWDB*, 2003.
19. L. Zou, J. Mo, L. Chen, M. Özsu, and D. Zhao. gstore: answering sparql queries via subgraph matching. *Proceedings of the VLDB Endowment*, 4(8):482–493, 2011.