SWI-Prolog Spatial Indexing

Willem Robert van Hage
VU University Amsterdam
The Netherlands
E-mail: W.R.van.Hage@vu.nl

April 17, 2014

Abstract

SWI-Prolog interface to Spatial Index and GEOS libraries, providing spatial indexing of URI's. Supports import and export to GML, KML, and RDF with GeoRSS Simple, GeoRSS GML, and W3C WGS84 vocabulary properties.

Nota bene that the spatialindex and GEOS C++ libraries have to be installed separately for this module to work.

Contents

1 Introduction

The Space package [1] provides spatial indexing for SWI-Prolog. It is based on Geometry Engine Open Source and the Spatial Index Library.

2 Shapes as Prolog Terms

The central objects of the Space package are pairs, $\langle u, s\rangle$ of a URI, u, and its associated shape, s. The URIs are linked to the shapes with the `uri_shape/2` predicate. We will support all OpenGIS Simple Features, points, linestrings, polygons (with ≥ 0 holes), multi-points, multi-polygons, and geometry collections; and some utility shapes like box and circle regions.[1]

Both the URIs and the shapes are represented as Prolog terms. This makes them first-class Prolog citizens, which allows the construction and transformation of shapes using regular Prolog clauses, or Definite Clause Grammars (DCGs). We support input from locations encoded in RDF with the W3C WGS84 vocabulary and with the GeoRSS Simple properties and the GeoRSS `where` property leading to an XML literal consisting of a GML element. The `uri_shape/2` predicate searches for URI-Shape pairs in SWI-Prolog's RDF triple store. It matches URIs to Shapes by using WGS84 and GeoRSS properties. For example, a URI u is associated with the shape $s =$ `point(`$lat, long$`)` if the triple store contains the triples: $\langle u,$ `wgs84_pos:lat` $, lat\rangle$ and $\langle u,$ `wgs84_pos:long` $, long\rangle$; or when it contains one of the following triples:
$\langle u,$ `georss:point`,"*lat long*"$\rangle$ or $\langle u,$ `georss:where`,"`<gml:Point><gml:pos>` *lat long* `</gml:pos></gml:Point>`"$\rangle$. The XML literal containing the GML description of the geometric shape is parsed with a DCG that can also be used to generate GML from Prolog shape terms.

```
?- shape(point(52.3325,4.8673)),
   shape(box(point(52.3324,4.8621),point(52.3348,4.8684))),
   shape(
   polygon([[point(52.3632,4.981)|_],      % the outer shell of the polygon
            [point(52.3631,4.9815)|_] |_ % any number of holes 0..*
           ])).
true.
%% uri_shape(?URI, ?Shape) is nondet.
?- uri_shape('http://www.example.org/myoffice', Shape).  % read from RDF
Shape = point(52.3325,4.8673).
```

3 Adding, Removing, and Bulkloading Shapes

The spatial index can be modified in two ways: By inserting or retracting single URI-shape pairs respectively using the `space_assert/3`, or the `space_retract/3` predicate; or by loading many pairs at once using the `space_bulkload/3` predicate or its parameterless counterpart `space_index_all/0` which simply loads all the shapes it can find with the `uri_shape/2` predicate into the default index. The former method is best for small manipulations of indices, while the

[1]The current version of the Space package, 0.1.2, only supports points, linestrings, and polygons (with holes) and box regions. Development on the other (multi-)shape types is underway.

latter method is best for the loading of large numbers of URI-shape pairs into an index. The Space package can deal with multiple indices to make it possible to divide sets of features. Indices are identified with a name handle, which can be any Prolog atom.[2] The actual indexing of the shapes is performed using lazy evaluation. Assertions and retractions are put on a queue that belongs to an index. The queue is committed to the index whenever a query is performed, or when a different kind of modification is called for (*i.e.* when the queue contains assertions and a retraction is requested or vice versa).

```
?- space_assert(ex:myoffice, point(52.3325,4.8673), demo_index).  % only adds it
true.
?- space_contains(box(point(52.3324,4.8621), point(52.3348,4.8684)),
                  Cont, demo_index).
% uses 'demo_index', so triggers a call to space_index('demo_index').
Cont = 'http://www.example.org/myoffice' .    % first instantiation, etc.
```

```
?- space_bulkload(space, uri_shape, demo_index).
true.
```

```
% If the KML Geometry elements have an ID attribute,
% you can load them from a file, e.g. 'office.kml', like this:
?- space_bulkload(kml_file_uri_shape('office.kml'), 'demo_index').
% Added 12 URI-Shape pairs to demo_index
true.

% You can insert the same objects one by one like this:
?- forall( kml_file_uri_shape('office.kml', Uri, Shape),
           space_assert(Uri, Shape, 'demo_index') ).
true.
```

4 Query types

We chose three basic spatial query types as our basic building blocks: *containment*, *intersection*, and *nearest neighbor*. These three query types are implemented as pure Prolog predicates, respectively `space_contains/3`, `space_intersects/3`, and `space_nearest/3`. These predicates work completely analogously, taking an index handle and a query shape to retrieve the URI of a shape matching the query, which is bound to the second argument. Any successive calls to the predicate try to re-instantiate the second argument with a different matching URI. The results of containment and intersection queries are instantiated in no particular order, while the nearest neighbor results are instantiated in order of increasing distance to the query shape. The `space_nearest_bounded/4` predicate is a containment query based on `space_nearest/3`, which returns objects within a certain range of the query shape in order of increasing distance.

[2]Every predicate in the Space package that must be given an index handle also has an abbreviated version without the index handle argument which automatically uses the default index.

```
?- space_nearest(point(52.3325,4.8673), N, 'demo_index').
N = 'http://sws.geonames.org/2759113/' ;      % retry, ask for more
N = 'http://sws.geonames.org/2752058/' ;      % retry
N = 'http://sws.geonames.org/2754074/' .      % cut, satisfied
```

5 Importing and Exporting Shapes

Besides supporting input from RDF we support input and output for other standards, likeGML, KML and WKT. All shapes can be converted from and to these standards with the `gml_shape/2`, `kml_shape/2`, and `wkt_shape/2` predicates.

```
% Convert a WKT shape into GML and KML}
?- wkt_shape('POINT ( 52.3325 4.8673 )', Shape),  % instantiate from WKT
   gml_shape(GML, Shape),
   kml_shape(KML, Shape).
Shape = point(52.3325, 4.8673),
GML = '<gml:Point><gml:pos>52.3325 4.8673</gml:pos></gml:Point>',
KML = '<Point><coordinates>4.8673,52.3325</coordinates></Point>' .
```

6 Integration of Space and Semantics

The non-deterministic implementation of the queries makes them behave like a lazy stream of solutions. This allows tight integration with other types of reasoning, like RDF(S) and OWL reasoning or other Prolog rules. An example of combined RDF and spatial reasoning is shown below.

```
% Finds nearest railway stations in the province Utrecht (in GeoNames)
?- uri_shape(ex:myoffice, Office),
   rdf(Utrecht, geo:name, literal('Provincie Utrecht')),
   space_nearest(Office, Near),
   % 'S' stands for a spot, like a building, 'RSTN' for railway station
   rdf(Near, geo:featureCode, geo:'S.RSTN'),
   % 'Near' connected to 'Utrecht' by transitive 'parentFeature'
   rdf_reachable(Near, geo:parentFeature, Utrecht),
   rdf(Near, geo:name, literal(Name)),  % fetch name of 'Near'
   uri_shape(Near, Station),  % fetch shape of station
   % compute actual distance in km}
   space_distance_greatcircle(Office, Station, Distance, km).
Utrecht = 'http://sws.geonames.org/2745909/',  % first instantiation
Near = 'http://sws.geonames.org/6639765/',
Name = 'Station Abcoude' ,
Station = point(52.2761, 4.97904),
Distance = 9.85408 ;                          % etc.
```

```
Utrecht = 'http://sws.geonames.org/2745909/',  % second instantiation
Near = 'http://sws.geonames.org/6639764/',
Name = 'Station Breukelen' ,
Station = point(52.17, 4.9906),
Distance = 19.9199 .                             % etc.
```

Integration of multiple spatial queries can be done in the same way. Since the queries return URIs an intermediate URI-Shape predicate is necessary to get a shape that can be used as a query. An example is shown below.

```
% Find features inside nearby polygons.
?- uri_shape(ex:myoffice, Office),
   space_nearest(Office, NearURI),
   uri_shape(NearURI, NearShape),  % look up the shape of the URI 'Near'
   NearShape = polygon(_),  % assert that it must be a polygon}
   space_contains(NearShape, Contained).
```

7 Architecture

The Space package consists of C++ and Prolog code. The main component is the Prolog module space.pl. All parsing and generation of input and output formats is done in Prolog. All index manipulation is done through the foreign language interface (FLI) from Prolog to C++. The `space_bulkload/3` predicate also communicates back across the FLI from C++ to Prolog, allowing the indexing functions to ask for candidates to index from the Prolog database, for example, by calling the `uri_shape/2` predicate.

7.1 Incremental Search and Non-determinism

The three search operations provided by the Space package all yield their results incrementally, *i.e.* one at a time. Prolog predicates actually do not have return values, but instantiate parameters. Multiple return values are returned by subsequently instantiating the same variable, so the first call to a predicate can make different variable instantiations than the second call. This standard support of non-deterministic behavior makes it easy to write incremental algorithms in Prolog.

Internally, the search operations are handled by C++ functions that work on an R*-tree index from the Spatial Index Library [2]. The C++ functions are accessed with the SWI-Prolog foreign language interface. To implement non-deterministic behavior the query functions have to store their state between successive calls and Prolog has to be aware which state is relevant to every call.

Every search query creates an instance of a SpatialIndex::IQueryStrategy class (the IncrementalNearestNeighborStrategy class for INN queries, the IncrementalRangeQuery for containment and intersection queries). This class contains the search algorithm, accesses the R*-tree index, and stores the current state of the algorithm. For containment and intersection queries the results can be returned in any particular order so implementing non-deterministic behavior simply involves storing a pointer to a node in the R*-tree and returning every subsequent matching object. For nearest neighbor queries keeping state is slightly more complicated, because it is necessary to keep a priority queue of candidate results at all times to guarantee that the results are returned in order of increasing proximity.

The Spatial Index library does not include an incremental nearest neighbor, so we implemented an adaptation of the algorithm described in [3] as an IQueryStrategy. The original algorithm emits results, for example, with a callback function, without breaking from the search loop that finds all matches. Our adaptation breaks the search loop at every matching object and stores a handle to the state (including the priority queue) so that it can restart the search loop where it left off. This makes it possible to tie the query strategy into the non-deterministic foreign language interface of SWI-Prolog with very little time overhead. A pointer to the IQueryStrategy instance is stored on the Prolog stack, so that every successive call to the procedure knows with which query to continue.

An alternative implementation would be to take the exact IncNearest algorithm described in [3] and to emit the results into a queue. The Prolog stack would then contain a pointer to the queue. Every successive call would dequeue a result from the queue. This strategy is less time efficient, because of two reasons. It does not halt after each match, so it is less efficient when looking for few results. It requires two separate processes to run. One to find results, the other to poll the queue. This means there is some process management and communication overhead.

8 Documentation

8.1 library(space/space): Core spatial database

set_space(+*Option*) *[det]*
set_space(+*IndexName, +Option*) *[det]*

This predicate can be used to change the options of a spatial index (or de default index for `set_space/1`). Some options, like `rtree_storage(S)` where S is disk or memory only have effect after clearing or bulkloading. Others, take effect immediately on a running index. More documentation will be provided in the near future.

space_assert(+*URI, +Shape, +IndexName*) *[det]*
space_assert(+*URI, +Shape*) *[det]*

Insert *URI* with associated *Shape* in the queue to be inserted into the index with name *IndexName* or the default index. Indexing happens lazily at the next call of a query or manually by calling `space_index/1`.

space_retract(+*URI, +Shape, +IndexName*) *[det]*
space_retract(+*URI, +Shape*) *[det]*

Insert *URI* with associated *Shape* in the queue to be removed into the index with name *IndexName* or the default index. Indexing happens lazily at the next call of a query or manually by calling `space_index/1`.

space_index(+*IndexName*) *[det]*
space_index *[det]*

Processes all asserts or retracts in the space queue for index *IndexName* or the default index if no index is specified.

space_clear(+*IndexName*) *[det]*
space_clear *[det]*

Clears index *IndexName* or the default index if no index is specified, removing all of its contents.

space_bulkload(*:Closure, +IndexName*) *[det]*
space_bulkload(*:Closure*) *[det]*
space_bulkload *[det]*

Fast loading of many Shapes into the index *IndexName*. *Closure* is called with two additional arguments: URI and Shape, that finds candidate URI-Shape pairs to index in the index *IndexName*.

`space_bulkload/0` defaults to `uri_shape/2` for :*Closure*.

See also the `uri_shape/2` predicate for an example of a suitable functor.

space_contains(*+Query, ?Cont, +IndexName*) *[nondet]*
space_contains(*+Query, ?Cont*) *[nondet]*

Containment query. Unifies *Cont* with shapes contained in *Query* Shape (or shape of *Query* URI) according to index *IndexName* or the default index.

space_intersects(*+Query, ?Inter, +IndexName*) *[nondet]*
space_intersects(*+Query, ?Inter*) *[nondet]*

Intersection query. Unifies *Inter* with shapes intersecting with *Query* Shape (or Shape of *Query* URI) according to index *IndexName* or the default index. (intersection subsumes containment)

space_nearest(*+Query, -Near, +IndexName*) *[nondet]*
space_nearest(*+Query, -Near*) *[nondet]*

Incremental Nearest-Neighbor query. Unifies *Near* with shapes in order of increasing distance to *Query* Shape (or Shape of *Query* URI) according to index *IndexName* or the default index.

uri_shape(*?URI, ?Shape*) *[nondet]*

Finds pairs of URIs and their corresponding Shapes based on WGS84 RDF properties (e.g. wgs84:lat), GeoRSS Simple properties (e.g. georss:polygon), and GeoRSS GML properties (e.g. georss:where).

`uri_shape/2` is a dynamic predicate, which means it can be extended. If you use `uri_shape/2` in this way, the *URI* argument has to be a canonical *URI*, not a QName.

uri_shape(*?URI, ?Shape, +Source*) *[nondet]*

Finds pairs of URIs and their corresponding Shapes using `uri_shape/2` from RDF that was loaded from a given *Source*.

space_index_all(*+IndexName*) *[det]*
space_index_all *[det]*

Loads all URI-Shape pairs found with `uri_shape/2` into index *IndexName* or the default index name.

shape(*+Shape*) *[det]*

Checks whether *Shape* is a valid supported shape.

space_distance(*+Point1, +Point2, -Distance*) *[det]*

Calculates the distance between *Point1* and *Point2* by default using pythagorean distance.

See also `space_distance_greatcircle/4` for great circle distance.

space_distance_greatcircle(+*Point1, +Point2, -Dist***)** *[det]*
space_distance_greatcircle(+*Point1, +Point2, -Dist, +Unit***)** *[det]*

Calculates great circle distance between *Point1* and *Point2* in the specified *Unit*, which can take as a value km (kilometers) or nm (nautical miles). By default, nautical miles are used.

8.2 library(space/georss)

georss_candidate(*?URI, ?Shape***)** *[nondet]*

Finds *URI-Shape* pairs by searching for RDF triples that link *URI* to a *Shape* with GeoRSS RDF properties (e.g. georss:where, georss:line, georss:polygon). Both GeoRSS Simple and GML are supported.

georss_candidate(*?URI, ?Shape, +Source***)** *[nondet]*

Finds *URI-Shape* pairs using `georss_candidate/2` in RDF that was loaded from a certain *Source*.

georss_simple_candidate(*?URI, ?Shape***)** *[nondet]*

Finds *URI-Shape* pairs by searching for GeoRSS Simple properties (e.g. georss:point, georss:line, georss:polygon) in the RDF database.

georss_uri_shape_triple(+*URI, +Shape, -Subject, -Predicate, -Object***)** *[det]*
georss_uri_shape_triple(-*URI, -Shape, +Subject, +Predicate, +Object***)** *[det]*

Converts between a *URI-Shape* pair and its GeoRSS simple RDF triple form.

georss_gml_candidate(*?URI, ?Shape***)** *[nondet]*

Finds *URI-Shape* pairs by searching for GeoRSS GML properties (i.e. georss:where) in the RDF database. Uses `gml_shape/2` to parse the XMLLiteral representing the GML shape.

8.3 library(space/wgs84)

wgs84_candidate(*?URI, ?Point***)** *[nondet]*

Finds *URI*-Shape pairs of RDF resources that are place-tagged with W3C WGS84 properties (i.e. lat, long, alt). *Point* = point(?Lat,?Long) ; *Point* = point(?Lat,?Long,?Alt).

wgs84_candidate(*?URI, ?Point, +Source***)** *[nondet]*

Finds *URI*-Shape pairs of RDF resources that are place-tagged with W3C WGS84 properties (i.e. lat, long, alt). From RDF that was loaded from a certain *Source*.

lat(*?URI, ?Lat***)** *[nondet]*

Finds the WGS84 latitude of resource *URI* (and vice versa) using the rdf_db index. *Lat* is a number.

long(*?URI, ?Long***)** *[nondet]*

Finds the WGS84 longitude of resource *URI* (and vice versa) using the rdf_db index. *Long* is a number.

alt(*?URI, ?Alt***)** *[nondet]*

Finds the WGS84 altitude of resource *URI* (and vice versa) using the rdf_db index. *Alt* is a number.

coordinates(*?URI, ?Lat, ?Long*) *[nondet]*
coordinates(*?URI, ?Lat, ?Long, ?Alt*) *[nondet]*
Finds the WGS84 latitude, longitude and possibly altitude of resource *URI* (and vice versa) using the rdf_db index. *Lat*, *Long*, and *Alt* are numbers.

8.4 library(space/freebase)

freebase_candidate(*?URI, ?Point*) *[nondet]*
Finds *URI*-Shape pairs of RDF resource that are place-tagged with Freebase's location.location.geoposition notation that capture WGS84 latitude/longitude positions.

freebase_candidate(*?URI, ?Point, ?Source*) *[nondet]*
Finds *URI*-Shape pairs of RDF resource that are place-tagged with Freebase's location.location.geoposition notation that capture WGS84 latitude/longitude positions. From RDF that was loaded from a certain *Source*.

8.5 library(space/dbpedia)

dbpedia_candidate(*?URI, ?Point*) *[nondet]*
Finds *URI*-Shape pairs of RDF resource that are place-tagged with DBpedia's coordinatenProperty notation that capture WGS84 latitude/longitude positions.

dbpedia_candidate(*?URI, ?Point, ?Source*) *[nondet]*
Finds *URI*-Shape pairs of RDF resource that are place-tagged with DBpedia's coordinatenProperty notation that capture WGS84 latitude/longitude positions. From RDF that was loaded from a certain *Source*.

8.6 library(space/wkt)

wkt_shape(*?WKT, ?Shape*) *[semidet]*
Converts between the *WKT* serialization of a *Shape* and its native Prolog term representation.

8.7 library(space/kml)

kml_file_to_georss(*+KMLfile*) *[det]*
kml_file_to_georss(*+KMLfile, +RDFfile*) *[det]*
Converts the contents of an KML file into GeoRSS RDF in the RDF database of Prolog. The Geometries are converted to GeoRSS properties and values. Documents, Folders, etc. are ignored. MultiGeometry objects are expanded into separate simple Geometries. Geometries with an XML ID are assigned that ID as URI, other Geometries are assigned a RDF blank node. The kml:name and kml:description are translated to RDF properties.

georss_to_kml_file(*+KMLfile*) *[det]*
georss_to_kml_file(*+KMLfile, +Options*) *[det]*
Converts the contents of the RDF database of Prolog into a KML file without style information and without Folders. kml:name and kml:description properties in the RDF database are converted to their KML counterparts. *Options* can be used to pass Document level options, for example, the name of the dataset. *Options* can also include a `graph(Graph)` option to specify which RDF named graph should be converted to KML.

kml_shape(*?Stream, ?Shape*) *[semidet]*

kml_shape(*?Stream, ?Shape, ?Attributes, ?Content*) *[semidet]*

Converts between the KML serialization of a shape and its internal Prolog term representation. *Attributes* and *Content* can hold additional attributes and XML content elements of the KML, like ID, name, or styleUrl.

kml_uri_shape(*?KML, ?URI, ?Shape*) *[semidet]*

Converts between the *KML* serialization of a *URI*-shape pair and its internal Prolog term representation. It is assumed the *KML* Geometry element has a ID attribute specifying the *URI* of the shape. e.g.

```
<PointID="http://example.org/point1"><coordinates>52.37,4.89</coordinates></P
```

kml_file_shape(*+File, ?Shape*) *[semidet]*

kml_file_shape(*+File, ?Shape, ?Attributes, ?Content*) *[semidet]*

Reads shapes from a KML file using `kml_shape/2`. `kml_file_shape/4` also reads extra attributes and elements of the KML Geometry. e.g. <Point targetId="NCName"><extrude>0</extrude>...</Point> will, besides parsing the Point, also instantiate *Content* with [`extrude(0)`] and *Attributes* with [`targetId('NCName')`].

kml_file_uri_shape(*+File, ?URI, ?Shape*) *[semidet]*

Reads *URI*-shape pairs from *File* using `kml_uri_shape/2`.

kml_save_header(*+Stream, +Options*) *[semidet]*

Outputs a KML header to *Stream*. This can be followed by calls to `kml_save_shape/3` and `kml_save_footer/1`.

Options is an option list that can contain the option `name(Name)` specifying the Name of the document.

To be done options to configure optional entities, like styles

kml_save_shape(*+Stream, +Shape, +Options*) *[semidet]*

Outputs a KML serialization of *Shape* to *Stream*. This can be preceded by a call to `kml_save_header/2` and followed by more calls to `kml_save_shape/3` and a call to `kml_save_footer/1`.

Options is an option list that can contain the option `attr(+List)` or `content(+List)` that can be used to add additional attributes or xml element content to a shape. This can be used to specify things like the ID or name.

Layout elements, like Placemark and Folder, have their own separate extra attributes to supply additional attributes and content. These can contain the special terms geom_attributes and geom_content that pass their content to the shape contained by the Placemark. For example, rendering a Placemark with the ID "placemark12" of an extruded Point shape with its URI as name of the Placemark and as ID of the shape and an additional styleUrl works as follows:

```
kml_save_shape(Stream,
                placemark(point(53.0,3.9),
                           [ id(placemark12),
```

```
                                  geom_attributes([ id(URI) ])
                                ],
                                [ name(URI),styleUrl(URI),
                                  geom_content([ extrude(1) ])
                                ]),
                  []).
```

kml_save_footer(+*Stream***)** *[det]*

Outputs a KML footer to stream *Stream*. This can be preceded by calls to `kml_save_header/2` and `kml_save_shape/3`.

8.8 library(space/gml)

gml_shape(*?GML, ?Shape***)** *[semidet]*

Converts between the *GML* serialization of a shape and its internal Prolog term representation.

8.9 library(space/space_web_loader)

space_load_url(+*URL***)** *[det]*

Retrieve RDF over HTTP from a *URL*, load it in the rdf_db and index all URI-Shape pairs that can be found in it into the default index.

space_load_url(+*URL*, +*Options***)** *[det]*

Load using `space_load_url/1`, given extra options.

index(+*IndexName***)**
Index the URI-Shape pairs into index named *IndexName*.

graph(+*Graph***)**
Store the URI-Shape pairs in the named graph *Graph*. The pairs are recorded as `uri_shape(URI,Shape,Graph)`.

space_unload_url(+*URL***)** *[det]*

Unload the RDF that was fetched from *URL* and remove all URI-Shape pairs that are contained in it from the default index.

space_unload_url(+*URL*, +*Options***)** *[det]*

Unload the RDF that was fetched from *URL* and remove all URI-Shape pairs that are contained in it. Accepts extra options:

index(+*IndexName***)**
Remove from the index named *IndexName*.

graph(+*Graph***)**
Remove the URI-Shape pairs from the named graph *Graph*.

space_crawl_url(+*URL***)** *[det]*

Retrieve RDF over HTTP from a *URL*, load it in the rdf_db and index all URI-Shape pairs that can be found in it into the default index. Also attempt to resolve all URIs that appear as object in a link_property statement downloaded from the *URL*. Retrieve these URIs and process them in the same way. Iterate this process until there are no new links that have not already been crawled.

space_crawl_url(+*URL*, +*Options*) *[det]*

Crawl using `space_crawl_url/1`, with additional options.

index(+*IndexName*)

Index the URI-Shape pairs into index named *IndexName*.

graph(+*Graph*)

Store the URI-Shape pairs in the named graph *Graph*. The pairs are recorded as `uri_shape(URI,Shape,Graph)`.

space_uncrawl_url(+*URL*) *[det]*

Unload the RDF that was fetched from *URL* and remove all URI-Shape pairs that are contained in it from the default index. Also unload all data that were crawled by iteratively resolving the URIs linked to with a link_property.

space_uncrawl_url(+*URL*, +*IndexName*) *[det]*

Unload using `space_uncrawl_url/1`, but remove the URI-Shape pairs from the index named *IndexName*.

index(+*IndexName*)

Remove the URI-Shape pairs from index named *IndexName*.

graph(+*Graph*)

Remove the URI-Shape pairs from the named graph *Graph*.

Index

References

[1] Willem Robert van Hage, Jan Wielemaker and Guus Schreiber. The Space package: Tight Integration Between Space and Semantics. *Proceedings of the 8th International Semantic Web Conference Workshop: TerraCognita 2009.*

[2] Marios Hadjieleftheriou, Erik Hoel, and Vassilis J. Tsotras. Sail: A spatial index library for efficient application integration. *Geoinformatica*, 9(4), 2005.

[3] Gísli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318, 1999.