Rtree Documentation

Release 1.4.1

Sean Gilles, Howard Butler, and contributors

Aug 13, 2025

CONTENTS

Rtree is a ctypes Python wrapper of libspatialindex that provides a number of advanced spatial indexing features for the spatially curious Python user. These features include:

- Nearest neighbor search
- Intersection search
- Multi-dimensional indexes
- Clustered indexes (store Python pickles directly with index entries)
- Bulk loading
- Deletion
- Disk serialization
- Custom storage implementation (to implement spatial indexing in ZODB, for example)

CHAPTER

ONE

DOCUMENTATION

1.1 Installation

1.1.1 *nix

First, download and install version 1.8.5+ of the libspatialindex library from:

https://libspatialindex.org

The library supports CMake builds, so it is a matter of:

```
$ mkdir build && cd build
$ cmake ..
$ cmake --build . -j
$ cmake --install .
```

You may need to run the `ldconfig` command after installing the library to ensure that applications can find it at startup time.

Rtree can be easily installed via pip:

```
$ pip install rtree
```

or by running in a local source directory:

```
$ pip install -e .
```

You can build and test in place like:

```
$ pytest
```

1.1.2 Windows

The Windows DLLs of libspatialindex are pre-compiled in windows installers that are available from PyPI. Installation on Windows is as easy as:

```
$ pip install rtree
```

1.2 Tutorial

This tutorial demonstrates how to take advantage of *Rtree* for querying data that have a spatial component that can be modeled as bounding boxes.

1.2.1 Creating an index

The following section describes the basic instantiation and usage of *Rtree*.

Import

After *installing* *Rtree*, you should be able to open up a Python prompt and issue the following:

```
>>> from rtree import index
```

`rtree` is organized as a Python package with a couple of modules and two major classes - *rtree.index.Index* and *rtree.index.Property*. Users manipulate these classes to interact with the index.

Construct an instance

After importing the index module, construct an index with the default construction:

```
>>> idx = index.Index()
```

> **Note**
>
> While the default construction is useful in many cases, if you want to manipulate how the index is constructed you will need pass in a *rtree.index.Property* instance when creating the index.

Create a bounding box

After instantiating the index, create a bounding box that we can insert into the index:

```
>>> left, bottom, right, top = (0.0, 0.0, 1.0, 1.0)
```

> **Note**
>
> The coordinate ordering for all functions are sensitive the the index's `interleaved` data member. If `interleaved` is False, the coordinates must be in the form [xmin, xmax, ymin, ymax, ..., ..., kmin, kmax]. If `interleaved` is True, the coordinates must be in the form [xmin, ymin, ..., kmin, xmax, ymax, ..., kmax].

Insert records into the index

Insert an entry into the index:

```
>>> idx.insert(0, (left, bottom, right, top))
```

> **Note**
>
> Entries that are inserted into the index are not unique in either the sense of the *id* or of the bounding box that is inserted with index entries. If you need to maintain uniqueness, you need to manage that before inserting entries into the Rtree.

> **Note**
>
> Inserting a point, i.e. where left == right && top == bottom, will essentially insert a single point entry into the index instead of copying extra coordinates and inserting them. There is no shortcut to explicitly insert a single point, however.

Query the index

There are three primary methods for querying the index. *rtree.index.Index.intersection()* will return you index entries that *cross* or are *contained* within the given query window. *rtree.index.Index.intersection()*

Intersection

Given a query window, return ids that are contained within the window:

```
>>> list(idx.intersection((1.0, 1.0, 2.0, 2.0)))
[0]
```

Given a query window that is beyond the bounds of data we have in the index:

```
>>> list(idx.intersection((1.0000001, 1.0000001, 2.0, 2.0)))
[]
```

Nearest Neighbors

The following finds the 1 nearest item to the given bounds. If multiple items are of equal distance to the bounds, both are returned:

```
>>> idx.insert(1, (left, bottom, right, top))
>>> list(idx.nearest((1.0000001, 1.0000001, 2.0, 2.0), 1))
[0, 1]
```

1.2.2 Using Rtree as a cheapo spatial database

Rtree also supports inserting any object you can pickle into the index (called a clustered index in libspatialindex parlance). The following inserts the picklable object `42` into the index with the given id `2`:

```
>>> idx.insert(id=2, coordinates=(left, bottom, right, top), obj=42)
```

You can then return a list of objects by giving the `objects=True` flag to intersection:

```
>>> [n.object for n in idx.intersection((left, bottom, right, top), objects=True)]
[None, None, 42]
```

> **Warning**
>
> libspatialindex's clustered indexes were not designed to be a database. You get none of the data integrity protections that a database would purport to offer, but this behavior of *Rtree* can be useful nonetheless. Consider yourself warned. Now go do cool things with it.

1.2.3 Serializing your index to a file

One of *Rtree*'s most useful properties is the ability to serialize Rtree indexes to disk. These include the clustered indexes described *here*:

```
>>> import os
>>> from tempfile import TemporaryDirectory
>>> prev_dir = os.getcwd()
>>> temp_dir = TemporaryDirectory()
>>> os.chdir(temp_dir.name)
>>> file_idx = index.Rtree("myidx")
>>> file_idx.insert(1, (left, bottom, right, top))
>>> file_idx.insert(2, (left - 1.0, bottom - 1.0, right + 1.0, top + 1.0))
>>> [n for n in file_idx.intersection((left, bottom, right, top))]
[1, 2]
>>> sorted(os.listdir())
['myidx.dat', 'myidx.idx']
>>> os.chdir(prev_dir)
>>> temp_dir.cleanup()
```

Note

By default, if an index file with the given name `myidx` in the example above already exists on the file system, it will be opened in append mode and not be re-created. You can control this behavior with the *rtree.index.Property.overwrite* property of the index property that can be given to the *rtree.index.Index* constructor.

See also

Performance describes some parameters you can tune to make file-based indexes run a bit faster. The choices you make for the parameters is entirely dependent on your usage.

Modifying file names

Rtree uses the extensions *dat* and *idx* by default for the two index files that are created when serializing index data to disk. These file extensions are controllable using the *rtree.index.Property.dat_extension* and *rtree.index.Property.idx_extension* index properties.

```
>>> p = index.Property()
>>> p.dat_extension = "data"
>>> p.idx_extension = "index"
>>> file_idx = index.Index("rtree", properties=p)
```

1.2.4 3D indexes

As of Rtree version 0.5.0, you can create 3D (actually kD) indexes. The following is a 3D index that is to be stored on disk. Persisted indexes are stored on disk using two files – an index file (.idx) and a data (.dat) file. You can modify the extensions these files use by altering the properties of the index at instantiation time. The following creates a 3D index that is stored on disk as the files `3d_index.data` and `3d_index.index`:

```
>>> from rtree import index
>>> temp_dir = TemporaryDirectory()
```

(continues on next page)

(continued from previous page)

```
>>> os.chdir(temp_dir.name)
>>> p = index.Property()
>>> p.dimension = 3
>>> p.dat_extension = "data"
>>> p.idx_extension = "index"
>>> idx3d = index.Index("3d_index", properties=p)
>>> idx3d.insert(1, (0, 60, 23.0, 0, 60, 42.0))
>>> list(idx3d.intersection((-1, 60, 22, 1, 62, 43)))
[1]
>>> os.chdir(prev_dir)
>>> temp_dir.cleanup()
```

1.2.5 ZODB and Custom Storages

https://mail.zope.org/pipermail/zodb-dev/2010-June/013491.html contains a custom storage backend for ZODB and you can find example python code here. Note that the code was written in 2011, hasn't been updated and was only an alpha version.

1.3 Class Documentation

class rtree.index.**Index**(**args*, ***kwargs*)

An R-Tree, MVR-Tree, or TPR-Tree indexing object

__init__(**args*, ***kwargs*)

Creates a new index

Parameters

- **filename** – The first argument in the constructor is assumed to be a filename determining that a file-based storage for the index should be used. If the first argument is not of type basestring, it is then assumed to be an instance of ICustomStorage or derived class. If the first argument is neither of type basestring nor an instance of ICustomStorage, it is then assumed to be an input index item stream.
- **stream** – If the first argument in the constructor is not of type basestring, it is assumed to be an iterable stream of data that will raise a StopIteration. It must be in the form defined by the `interleaved` attribute of the index. The following example would assume `interleaved` is False:

```
(id,
 (minx, maxx, miny, maxy, minz, maxz, ..., ..., mink, maxk),
 object)
```

The object can be None, but you must put a place holder of `None` there.

For a TPR-Tree, this would be in the form:

```
(id,
 ((minx, maxx, miny, maxy, ..., ..., mink, maxk),
  (minvx, maxvx, minvy, maxvy, ..., ..., minvk, maxvk),
  time),
 object)
```

- **storage** – If the first argument in the constructor is an instance of ICustomStorage then the given custom storage is used.
- **interleaved** – True or False, defaults to True. This parameter determines the coordinate order for all methods that take in coordinates.
- **properties** – An `index.Property` object. This object sets both the creation and instantiation properties for the object and they are passed down into libspatialindex. A few properties are curried from instantiation parameters for you like `pagesize` and `overwrite` to ensure compatibility with previous versions of the library. All other properties must be set on the object.

Warning

The coordinate ordering for all functions are sensitive the index's `interleaved` data member. If `interleaved` is False, the coordinates must be in the form [xmin, xmax, ymin, ymax, ..., ..., kmin, kmax]. If `interleaved` is True, the coordinates must be in the form [xmin, ymin, ..., kmin, xmax, ymax, ..., kmax]. This also applies to velocities when using a TPR-Tree.

A basic example

```
>>> from rtree import index
>>> p = index.Property()

>>> idx = index.Index(properties=p)
>>> idx
rtree.index.Index(bounds=[1.7976931348623157e+308,
                          1.7976931348623157e+308,
                          -1.7976931348623157e+308,
                          -1.7976931348623157e+308],
                          size=0)
```

Insert an item into the index:

```
>>> idx.insert(4321,
...            (34.3776829412, 26.7375853734, 49.3776829412,
...             41.7375853734),
...            obj=42)
```

Query:

```
>>> hits = idx.intersection((0, 0, 60, 60), objects=True)
>>> for i in hits:
...     if i.id == 4321:
...         i.object
...         i.bbox
...
42
[34.37768294..., 26.73758537..., 49.37768294..., 41.73758537...]
```

Using custom serializers:

```
>>> class JSONIndex(index.Index):
...     def dumps(self, obj):
```

(continues on next page)

(continued from previous page)

```
...         # This import is nested so that the doctest doesn't
...         # require simplejson.
...         import simplejson
...         return simplejson.dumps(obj).encode('ascii')
...
...     def loads(self, string):
...         import simplejson
...         return simplejson.loads(string.decode('ascii'))

>>> stored_obj = {"nums": [23, 45], "letters": "abcd"}
>>> json_idx = JSONIndex()
>>> try:
...     json_idx.insert(1, (0, 1, 0, 1), stored_obj)
...     list(json_idx.nearest((0, 0), 1,
...                           objects="raw")) == [stored_obj]
... except ImportError:
...     True
True
```

property bounds

Returns the bounds of the index

Parameters

coordinate_interleaved – If True, the coordinates are turned in the form [xmin, ymin, ..., kmin, xmax, ymax, ..., kmax], otherwise they are returned as [xmin, xmax, ymin, ymax, ..., ..., kmin, kmax]. If not specified, the `interleaved` member of the index is used, which defaults to True.

close()

Force a flush of the index to storage. Renders index inaccessible.

count(*coordinates*)

Return number of objects that intersect the given coordinates.

Parameters

coordinates (*Any*) – This may be an object that satisfies the numpy array protocol, providing the index's dimension * 2 coordinate pairs representing the *mink* and *maxk* coordinates in each dimension defining the bounds of the query window. For a TPR-Tree, this must be a 3-element sequence including not only the positional coordinate pairs but also the velocity pairs *minvk* and *maxvk* and a time pair for the time range as a float.

The following example queries the index for any objects any objects that were stored in the index intersect the bounds given in the coordinates:

```
>>> from rtree import index
>>> idx = index.Index()
>>> idx.insert(4321,
...            (34.3776829412, 26.7375853734, 49.3776829412,
...             41.7375853734),
...            obj=42)

>>> print(idx.count((0, 0, 60, 60)))
1
```

This example is similar for a TPR-Tree:

```
>>> p = index.Property(type=index.RT_TPRTree)
>>> idx = index.Index(properties=p)
>>> idx.insert(4321,
...            ((34.3776829412, 26.7375853734, 49.3776829412,
...              41.7375853734),
...              (0.5, 2, 1.5, 2.5),
...              3.0),
...            obj=42)

>>> print(idx.count(((0, 0, 60, 60), (0, 0, 0, 0), (3, 5))))
...
1
```

delete(*id*, *coordinates*)

Deletes an item from the index with the given `'id'` and
coordinates given by the `coordinates` sequence. As the index can contain multiple items with the same ID and coordinates, deletion is not guaranteed to delete all items in the index with the given ID and coordinates.

Parameters

- **id** (*int*) – A long integer ID for the entry, which need not be unique. The index can contain multiple entries with identical IDs and coordinates. Uniqueness of items should be enforced at the application level by the user.
- **coordinates** (*Any*) – Dimension * 2 coordinate pairs, representing the min and max coordinates in each dimension of the item to be deleted from the index. Their ordering will depend on the index's `interleaved` data member. These are not the coordinates of a space containing the item, but those of the item itself. Together with the id parameter, they determine which item will be deleted. This may be an object that satisfies the numpy array protocol. For a TPR-Tree, this must be a 3-element sequence including not only the positional coordinate pairs but also the velocity pairs *minvk* and *maxvk* and a time pair for the original time the object was inserted and the current time as a float.

Example:

```
>>> from rtree import index
>>> idx = index.Index()
>>> idx.delete(4321,
...            (34.3776829412, 26.7375853734, 49.3776829412,
...             41.7375853734))
```

For the TPR-Tree:

```
>>> p = index.Property(type=index.RT_TPRTree)
>>> idx = index.Index(properties=p)
>>> idx.delete(4321,
...            ((34.3776829412, 26.7375853734, 49.3776829412,
...              41.7375853734),
...              (0.5, 2, 1.5, 2.5),
...              (3.0, 5.0)))
```

insert(*id*, *coordinates*, *obj=None*)

Inserts an item into the index with the given coordinates.

Parameters

- **id** (*int*) – A long integer that is the identifier for this index entry. IDs need not be unique to be inserted into the index, and it is up to the user to ensure they are unique if this is a requirement.
- **coordinates** (*Any*) – This may be an object that satisfies the numpy array protocol, providing the index's dimension * 2 coordinate pairs representing the *mink* and *maxk* coordinates in each dimension defining the bounds of the query window. For a TPR-Tree, this must be a 3-element sequence including not only the positional coordinate pairs but also the velocity pairs *minvk* and *maxvk* and a time value as a float.
- **obj** (*object*) – a pickleable object. If not None, this object will be stored in the index with the id.

The following example inserts an entry into the index with id *4321*, and the object it stores with that id is the number *42*. The coordinate ordering in this instance is the default (interleaved=True) ordering:

```
>>> from rtree import index
>>> idx = index.Index()
>>> idx.insert(4321,
...            (34.3776829412, 26.7375853734, 49.3776829412,
...             41.7375853734),
...            obj=42)
```

This example is inserting the same object for a TPR-Tree, additionally including a set of velocities at time *3*:

```
>>> p = index.Property(type=index.RT_TPRTree)
>>> idx = index.Index(properties=p)
>>> idx.insert(4321,
...            ((34.3776829412, 26.7375853734, 49.3776829412,
...             41.7375853734),
...             (0.5, 2, 1.5, 2.5),
...            3.0),
...            obj=42)
```

intersection(*coordinates: Any*, *objects: Literal[True]*) → Iterator[*Item*]

intersection(*coordinates: Any*, *objects: Literal[False] = False*) → Iterator[int]

intersection(*coordinates: Any*, *objects: Literal['raw']*) → Iterator[object]

Return ids or objects in the index that intersect the given coordinates.

Parameters

- **coordinates** – This may be an object that satisfies the numpy array protocol, providing the index's dimension * 2 coordinate pairs representing the *mink* and *maxk* coordinates in each dimension defining the bounds of the query window. For a TPR-Tree, this must be a 3-element sequence including not only the positional coordinate pairs but also the velocity pairs *minvk* and *maxvk* and a time pair for the time range as a float.
- **objects** – If True, the intersection method will return index objects that were pickled when they were stored with each index entry, as well as the id and bounds of the index entries. If 'raw', the objects will be returned without the *rtree.index.Item* wrapper.

The following example queries the index for any objects any objects that were stored in the index intersect the bounds given in the coordinates:

```
>>> from rtree import index
>>> idx = index.Index()
>>> idx.insert(4321,
...            (34.3776829412, 26.7375853734, 49.3776829412,
...             41.7375853734),
...            obj=42)

>>> hits = list(idx.intersection((0, 0, 60, 60), objects=True))
>>> [(item.object, item.bbox) for item in hits if item.id == 4321]
...
[(42, [34.37768294..., 26.73758537..., 49.37768294...,
       41.73758537...])]
```

If the *rtree.index.Item* wrapper is not used, it is faster to request the 'raw' objects:

```
>>> list(idx.intersection((0, 0, 60, 60), objects="raw"))
[42]
```

Similar for the TPR-Tree:

```
>>> p = index.Property(type=index.RT_TPRTree)
>>> idx = index.Index(properties=p)
>>> idx.insert(4321,
...            ((34.3776829412, 26.7375853734, 49.3776829412,
...             41.7375853734),
...             (0.5, 2, 1.5, 2.5),
...             3.0),
...            obj=42)

>>> hits = list(idx.intersection(
...     ((0, 0, 60, 60), (0, 0, 0, 0), (3, 5)), objects=True))
...
>>> [(item.object, item.bbox) for item in hits if item.id == 4321]
...
[(42, [34.37768294..., 26.73758537..., 49.37768294...,
       41.73758537...])]
```

intersection_v(*mins*, *maxs*)

Bulk intersection query for obtaining the ids of entries which intersect with the provided bounding boxes. The return value is a tuple consisting of two 1D NumPy arrays: one of intersecting ids and another containing the counts for each bounding box.

Parameters

- **mins** – A NumPy array of shape *(n, d)* containing the minima to query.
- **maxs** – A NumPy array of shape *(n, d)* containing the maxima to query.

nearest(*coordinates: Any*, *num_results: int*, *objects: Literal[True]*) → Iterator[*Item*]

nearest(*coordinates: Any*, *num_results: int*, *objects: Literal[False] = False*) → Iterator[int]

nearest(*coordinates: Any*, *num_results: int*, *objects: Literal['raw']*) → Iterator[object]

Returns the k-nearest objects to the given coordinates.

Parameters

- **coordinates** – This may be an object that satisfies the numpy array protocol, providing the index's dimension * 2 coordinate pairs representing the *mink* and *maxk* coordinates in each dimension defining the bounds of the query window. For a TPR-Tree, this must be a 3-element sequence including not only the positional coordinate pairs but also the velocity pairs *minvk* and *maxvk* and a time pair for the time range as a float.
- **num_results** – The number of results to return nearest to the given coordinates. If two index entries are equidistant, *both* are returned. This property means that `num_results` may return more items than specified
- **objects** – If True, the nearest method will return index objects that were pickled when they were stored with each index entry, as well as the id and bounds of the index entries. If 'raw', it will return the object as entered into the database without the *rtree.index.Item* wrapper.

> **Warning**
>
> This is currently not implemented for the TPR-Tree.

Example of finding the three items nearest to this one:

```
>>> from rtree import index
>>> idx = index.Index()
>>> idx.insert(4321, (34.37, 26.73, 49.37, 41.73), obj=42)
>>> hits = idx.nearest((0, 0, 10, 10), 3, objects=True)
```

nearest_v(*mins*, *maxs*, ** (Keyword-only parameters separator (PEP 3102))*, *num_results=1*, *max_dists=None*, *strict=False*, *return_max_dists=False*)

Bulk k-nearest query for the given bounding boxes. The return value is a tuple consisting of, by default, two 1D NumPy arrays: one of intersecting ids and another containing the counts for each bounding box.

Parameters

- **mins** – A NumPy array of shape *(n, d)* containing the minima to query.
- **maxs** – A NumPy array of shape *(n, d)* containing the maxima to query.
- **num_results** – The maximum number of neighbors to return for each bounding box. If there are multiple equidistant furthest neighbors then, by default, they are *all* returned. Hence, the actual number of results can be greater than requested.
- **max_dists** – Optional; a NumPy array of shape *(n,)* containing the maximum distance to consider for each bounding box.
- **strict** – If True then each point will never return more than *num_results* even in cases of equidistant furthest neighbors.
- **return_max_dists** – If True, the distance of the furthest neighbor for each bounding box will also be returned.

class rtree.index.**Property**(*handle=None*, *owned=True*, ***kwargs*)

An index property object is a container that contains a number of settable index properties. Many of these properties must be set at index creation times, while others can be used to adjust performance or behavior.

property buffering_capacity: int

Buffering capacity

property custom_storage_callbacks

Callbacks for custom storage

property custom_storage_callbacks_size: int

Size of callbacks for custom storage

property dat_extension

Extension for .dat file

property dimension: int

Index dimension. Must be greater than 0, though a dimension of 1 might have undefined behavior.

property filename

Index filename for disk storage

property fill_factor: int

Index node fill factor before branching

property idx_extension

Extension for .idx file

property index_capacity: int

Index capacity

property index_id

First node index id

property index_pool_capacity: int

Index pool capacity

property leaf_capacity: int

Leaf capacity

property near_minimum_overlap_factor: int

Overlap factor for MVRTrees

property overwrite

Overwrite existing index files

property pagesize: int

The pagesize when disk storage is used. It is ideal to ensure that your index entries fit within a single page for best performance.

property point_pool_capacity: int

Point pool capacity

property region_pool_capacity: int

Region pool capacity

property reinsert_factor

Reinsert factor

property split_distribution_factor: int

Split distribution factor

property storage: int

Index storage.

One of RT_Disk, RT_Memory or RT_Custom.

If a filename is passed as the first parameter to :class:index.Index, RT_Disk is assumed. If a CustomStorage instance is passed, RT_Custom is assumed. Otherwise, RT_Memory is the default.

property tight_mbr

Uses tight bounding rectangles

property tpr_horizon

TPR horizon

property type: int

Index type. Valid index type values are RT_RTree, RT_MVTree, or RT_TPRTree. Only RT_RTree (the default) is practically supported at this time.

property variant: int

Index variant. Valid index variant values are RT_Linear, RT_Quadratic, and RT_Star

property writethrough

Write through caching

class rtree.index.**Item**(*loads*, *handle*, *owned=False*)

A container for index entries

__init__(*loads*, *handle*, *owned=False*)

There should be no reason to instantiate these yourself. Items are created automatically when you call *rtree.index.Index.intersection()* (or other index querying methods) with objects=True given the parameters of the function.

property bbox: list[float]

Returns the bounding box of the index entry

1.4 Miscellaneous Documentation

1.4.1 Exceptions

exception rtree.exceptions.**RTreeError**

RTree exception, indicates a RTree-related error.

1.4.2 Finder module

Locate *libspatialindex* shared library and header files.

rtree.finder.**get_include**()

Return the directory that contains the spatialindex *.h files.

Returns
Path to include directory or "" if not found.

Return type
str

rtree.finder.**load**()

Load the *libspatialindex* shared library.

Returns
Loaded shared library

Return type
CDLL

1.5 Changes

1.5.1 1.4.1: 2025-08-13

- Rename main branch references by @mwtoews in #356
- Fixing an incorrect reassignment in nearest_v and intersection_v by @Atilleusz in #358
- Add spatialindex version to tests, add common pytest configuration by @mwtoews in #360
- Refactor array-loading methods, add tests by @mwtoews in #361
- Minor refactor of code blocks in docs by @mwtoews in #362
- Resolve some issues in the batch API by @FreddieWitherden in #367
- fix #369 (load libspatialindex without changing cwd) by @remicres in #370
- arm64 wheels on windows by @w8sl in #378 and #371

Full Changelog

1.5.2 1.4.0: 2025-03-06

- Python 3.9+ is now required (#321)
- Add support for array-based bulk insert with NumPy (#340 by @FreddieWitherden)
- Upgrade binary wheels with libspatialindex-2.1.0 (#353)
- Rename project and other build components to "rtree" (#350)

1.5.3 1.3.0: 2024-07-10

- Upgrade binary wheels with libspatialindex-2.0.0 (#316)
- Fix binary wheels for musllinux wheels (#316)
- Update code style, replace isort and black with ruff, modern numpy rng (#319)
- Remove libsidx version testing (#313)

1.5.4 1.2.0: 2024-01-19

- Fix test failure with built library (#291 by @sebastic)
- Include spatialindex headers and add *get_include()* (#292 by @JDBetteridge)

1.5.5 1.1.0: 2023-10-17

- Python 3.8+ is now required (#273)
- Move project metadata to pyproject.toml (#269)
- Refactor built wheels for PyPI (#276)
- Fix memory leak when breaking mid-way in _get_objects and _get_ids (#266) (thanks @akariv!)

1.5.6 1.0.1: 2022-10-12

- Fix up type hints #243 (thanks @oderby)
- Python 3.11 wheels #250 (thanks @ewouth)

1.5.7 1.0.0: 2022-04-05

- Python 3.7+ is now required (#212) (thanks @adamjstewart!)
- Type hints (#215 and others) (thanks @adamjstewart!)
- Python 3.10 wheels, including osx-arm64 #224
- Clean up libspatialindex C API mismatches #222 (thanks @musicinmybrain!)
- Many doc updates, fixes, and type hints (thanks @adamjstewart!) #212 #221 #217 #215
- __len__ method for index #194
- Prevent get_coordinate_pointers from mutating inputs #205 (thanks @sjones94549!)
- linux-aarch64 wheels #183 (thanks @odidev!)
- black (#218) and flake8 (#145) linting

1.5.8 0.9.3: 2019-12-10

- find_library and libspatialindex library loading #131

1.5.9 0.9.2: 2019-12-09

- Refactored tests to be based on unittest #129
- Update libspatialindex library loading code to adapt previous behavior #128
- Empty data streams throw exceptions and do not partially construct indexes #127

1.5.10 0.9.0: 2019-11-24

- Add Index.GetResultSetOffset()
- Add Index.contains() method for object and id (requires libspatialindex 1.9.3+) #116
- Add Index.Flush() #107
- Add TPRTree index support (thanks @sdhiscocks #117)
- Return container sizes without returning objects #90
- Add set_result_limit and set_result_offset for Index paging 44ad21a

Bug fixes:

- Better exceptions in cases where stream functions throw #80

- Migrated CI platform to Azure Pipelines https://dev.azure.com/hobuinc/rtree/_build?definitionId=5
- Minor test enhancements and fixups. Both libspatialindex 1.8.5 and libspatialindex 1.9.3 are tested with CI

1.5.11 0.8: 2014-07-17

- Support for Python 3 added.

1.5.12 0.7.0: 2011-12-29

- 0.7.0 relies on libspatialindex 1.7.1+.
- int64_t's should be used for IDs instead of uint64_t (requires libspatialindex 1.7.1 C API changes)
- Fix __version__
- More documentation at http://toblerity.github.com/rtree/
- Class documentation at http://toblerity.github.com/rtree/class.html
- Tweaks for PyPy compatibility. Still not compatible yet, however.
- Custom storage support by Mattias (requires libspatialindex 1.7.1)

1.5.13 0.6.0: 2010-04-13

- 0.6.0 relies on libspatialindex 1.5.0+.
- *intersection()* and *nearest()* methods return iterators over results instead of lists.
- Number of results for *nearest()* defaults to 1.
- libsidx C library of 0.5.0 removed and included in libspatialindex
- objects="raw" in *intersection()* to return the object sent in (for speed).
- *count()* method to return the intersection count without the overhead of returning a list (thanks Leonard Norrgård).
- Improved bulk loading performance
- Supposedly no memory leaks :)
- Many other performance tweaks (see docs).
- Bulk loader supports interleaved coordinates
- Leaf queries. You can return the box and ids of the leaf nodes of the index. Useful for visualization, etc.
- Many more docstrings, sphinx docs, etc

1.5.14 0.5.0: 2009-08-06

0.5.0 was a complete refactoring to use libsidx - a C API for libspatialindex. The code is now ctypes over libsidx, and a number of new features are now available as a result of this refactoring.

- ability to store pickles within the index (clustered index)
- ability to use custom extension names for disk-based indexes
- ability to modify many index parameters at instantiation time
- storage of point data reduced by a factor of 4
- bulk loading of indexes at instantiation time

- ability to quickly return the bounds of the entire index
- ability to return the bounds of index entries
- much better windows support
- libspatialindex 1.4.0 required.

1.5.15 0.4.3: 2009-06-05

- Fix reference counting leak #181

1.5.16 0.4.2: 2009-05-25

- Windows support

1.5.17 0.4.1: 2008-03-24

- Eliminate uncounted references in add, delete, nearestNeighbor (#157).

1.5.18 0.4: 2008-01-24

- Testing improvements.
- Switch dependency to the single consolidated spatialindex library (1.3).

1.5.19 0.3: 26 November 2007

- Change to Python long integer identifiers (#126).
- Allow deletion of objects from indexes.
- Reraise index query errors as Python exceptions.
- Improved persistence.

1.5.20 0.2: 19 May 2007

- Link spatialindex system library.

1.5.21 0.1: 13 April 2007

- Add disk storage option for indexes (#320).
- Change license to LGPL.
- Moved from Pleiades to GIS-Python repo.
- Initial release.

1.6 Performance

See the benchmarks.py file for a comparison of various query methods and how much acceleration can be obtained from using Rtree.

There are a few simple things that will improve performance.

1.6.1 Use stream loading

This will substantially (orders of magnitude in many cases) improve performance over *insert()* by allowing the data to be pre-sorted

```
>>> from rtree import index
>>> def generator_function(somedata):
...     for i, obj in enumerate(somedata):
...         yield (i, (obj.xmin, obj.ymin, obj.xmax, obj.ymax), obj)
...
>>> r = index.Index(generator_function(somedata))
```

After bulk loading the index, you can then insert additional records into the index using *insert()*

1.6.2 Override dumps to use the highest pickle protocol

```
>>> import pickle
>>> import rtree
>>> class FastRtree(rtree.Rtree):
...     def dumps(self, obj):
...         return pickle.dumps(obj, -1)
...
>>> r = FastRtree()
```

Update from January 2024

Pickling is currently broken and awaiting a pull request to fix it. For more information, see the pull request on GitHub.

1.6.3 Use objects="raw"

In any *intersection()* or *nearest()* or query, use `objects="raw"` keyword argument:

```
>>> xmin, ymin, xmax, ymax = 0.0, 0.0, 1.0, 1.0
>>> objs = r.intersection((xmin, ymin, xmax, ymax), objects="raw")
```

1.6.4 Adjust index properties

Adjust *rtree.index.Property* appropriate to your index.

- Set your *leaf_capacity* to a higher value than the default 100. 1000+ is fine for the default pagesize of 4096 in many cases.
- Increase the *fill_factor* to something near 0.9. Smaller fill factors mean more splitting, which means more nodes. This may be bad or good depending on your usage.

1.6.5 Limit dimensionality to the amount you need

Don't use more dimensions than you actually need. If you only need 2, only use two. Otherwise, you will waste lots of storage and add that many more floating point comparisons for each query, search, and insert operation of the index.

1.6.6 Use the correct query method

Use `count()` if you only need a count and `intersection()` if you only need the ids. Otherwise, lots of data may potentially be copied. If possible also make use of the bulk query methods suffixed with *_v*.

1.7 History of Rtree

Rtree was started by Sean Gillies as a port of the libspatialindex linkages that QGIS maintained to provide on-the-fly indexing support for GUI operations. A notable feature of R-trees is the ability to insert data into the structure without the need for a global partitioning bounds, and this drove Sean's adoption of this code. Howard Butler later picked up Rtree and added a number of features that libspatialindex provided including disk serialization and bulk loading by writing a C API for libspatialindex and re-writing Rtree as a ctypes wrapper to utilize this C API. Brent Pedersen came along and added features to support alternative coordinate ordering, augmentation of the pickle storage, and lots of documentation. Mattias (http://dr-code.org) added support for custom storage backends to support using Rtree as an indexing type in ZODB.

Rtree has gone through a number of iterations, and at 0.5.0, it was completely refactored to use a new internal architecture (ctypes + a C API over libspatialindex). This refactoring has resulted in a number of new features and much more flexibility. See *Changes* for more detail.

> **Note**
>
> A significant bug in the 1.6.1+ libspatialindex C API was found where it was using unsigned integers for index entry IDs instead of signed integers. Because Rtree appeared to be the only significant user of the C API at this time, it was corrected immediately. You should update immediately and re-insert data into new indexes if this is an important consideration for your application.

Rtree 0.5.0 included a C library that is now the C API for libspatialindex and is part of that source tree. The code bases are independent from each other and can now evolve separately. Rtree is pure Python as of 0.6.0+.

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

V

W