Using ZODB to store and query geospatial data with python

Daniel Scherer

NoSQL and especially Object-Oriented Databases (OODB) have some major advantages against other database concepts, particularly when you are working with geospatial data, as it is very complex data, combining location, its Reference System, the geometry (Polygon, Line, etc.), its style, and many more meta and attribute data depending on the application.
With relational databases, all those fields have to be predefined in a fixed schema, to store the data and writing or reading the data requires SQL queries and additional actions to generate an object from the table and back. NoSQL databases take more advantages of the more and more cheaper memory and processing power but still have limited geospatial features [1]. Those features are sufficient for most of the day to day use cases, though.
With a NoSQL OODB, the geospatial data can be stored as an object and changed independently from the database. It can be restored seamlessly from the database directly without any conversions required. This text describes how to use an OODB with geospatial data. It gives two examples with Python and ZODB which should help to understand the usage in a small project.

Python has become one of the most important programming languages in geospatial applications. Both open source and commercial GIS Software support Python scripting. Additionally, there are several useful python modules available to create powerful applications for geospatial data analysis.

The Zope Object Database (ZODB) [2] is part of the Z Object Publishing Environment (ZOPE) [3]. However, it can be used individually as a native python OODB to store persistent objects with atomic transactions. ZODB provides undo and history features and is scalable by using the Zope Enterprise Objects (ZEO) as a network storage [4].
It works with Python 2 and 3.

Simple Example

For this example, we define a Geometry superclass and Point, Linestring and Polygon subclasses (Appendix A). Note, that the Geometry superclass subclasses Persistent to keep track of object changes automatically [5].

Now a polygon is constructed as an example object (Code Example 1).

In Code Example 2 a storage filename is chosen where the ZODB database will be instantiated and a connection is established. To store the objects efficiently a BTree is created as a root object. The geometry is then added to the BTree with an appropriate and unique key. By calling *transaction.commit()* and *storage.close()* the changes get committed and saved to the local file.

```python
from geometries import Point, LineString, Polygon

# Create points
p0 = Point(35,10)
p1 = Point(45,45)
p2 = Point(15,40)
p3 = Point(10,20)
p4 = Point(20,30)
p5 = Point(35,35)
p6 = Point(30,20)

# Construct polygon from points
outerRing = LineString([p0,p1,p2,p3,p0])
innerRing = LineString([p4,p5,p6,p4])
polygon = Polygon(outerRing,innerRing)
```

Code Example 1: Construction of a polygon

```python
import ZODB, ZODB.FileStorage, transaction, BTrees.OOBTree

# Choose storage file and establish connection
storage = ZODB.FileStorage.FileStorage('ZODB.fs')
db = ZODB.DB(storage)
connection = db.open()

# BTree as root object
connection.root.geometries = BTrees.OOBTree.BTree()

# Add geometry with user defined key
connection.root.geometries['polygon-1'] = polygon

# Commit changes and save
transaction.commit()
storage.close()
```

Code Example 2: Storing the polygon in the ZODB

To read the object from the database, a connection to the same file storage and database has to be established. Next, the geometry can be selected from the BTree with the previously assigned key. Because of the Persistence superclass *polygon.wkt()* can be called without further requirements to interact with the object (Code Example 3).

```python
import ZODB, ZODB.FileStorage

# Choose storage file and establish connection
storage = ZODB.FileStorage.FileStorage('ZODB.fs')
db = ZODB.DB(storage)
connection = db.open()

# Select geometry with user defined key and call WKT method
polygon = connection.root.geometries['polygon-1']
print(polygon.wkt())
```

Code Example 3: Loading the polygon from the ZODB

Spatial Index

While the simple example may meet some projects requirements, most projects will require to query objects by their location. Therefore, a spatial index or RTree should be used.

The RTree [6] provides Nearest Neighbor and Intersection search and Multi-dimensional indexes. It is a wrapper of "libspatialindex" [7] which must be downloaded and installed separately. The RTree could be used in place of the BTree in the previous example, but to keep it simple an additional RTree is used as the spatial index. So, the object is still added to the BTree and its bounding box is stored with the same id as key in the RTree, which is saved as an additional file.
The *insert()* function (Code Example 4) helps to ensure that the equal unique key is used in both indexes. The IOBTree is used, as the RTree only supports integer keys. The function tests if there is already a tree in the database and creates a new unique key.

```
import transaction, BTrees.IOBTree
from rtree.index import Index

def insert(geometry):
    # Open spatial index
    idx = Index('spatial')

    # Get new unique key or set up BTree as root object
    try:
        id = max(list(root.geometries.keys())) + 1
    except AttributeError:
        root.geometries = BTrees.IOBTree.BTree()
        id = 0

    # Add bounding box of geometry to spatial index with assigned key
    idx.insert(id, geometry.bbox())

    # Add geometry to BTree with assigned key
    root.geometries[id] = geometry

    # Commit changes and save spatial index
    idx.close()
    transaction.commit()
```
Code Example 4: Insert function

Similar to the Code Example 2 a connection to the database is established to insert the geometries with the *insert()* function in Code Example 5.

```python
import ZODB, ZODB.FileStorage

# Choose storage file and establish connection
storage = ZODB.FileStorage.FileStorage('ZODB_RTRee.fs')
db = ZODB.DB(storage)
connection = db.open()
root = connection.root

# Add geometries
insert(polygon)
insert(p2)
insert(p6)

# Save changes
storage.close()
```

Code Example 5: Storing the polygon with the insert function

Now the database can be queried from another process by location utilizing the spatial index in Code Example 6. The RTree has the two functions *intersection(bbox)* and *nearest(bbox)*, which return the ids of the geometries intersecting the given bounding box or of the nearest neighbor.

```python
import ZODB, ZODB.FileStorage, transaction
from rtree.index import Index

# Choose storage file and establish connection
storage = ZODB.FileStorage.FileStorage('ZODB_RTRee.fs')
db = ZODB.DB(storage)
connection = db.open()
geometries = connection.root.geometries

# Open spatial index
idx = Index('spatial')

# Choose an bounding box of interest
# left, bottom, right, top
bbox = (15, 40, 15, 40)

# Search nearest geometry to bounding box
print([geometries[int(n)].wkt() for n in idx.nearest(bbox)])
```

Code Example 6: Querying the spatial index to find the nearest geometry and loading it from the ZODB

Of course, this example is not perfect. There may be issues when there are many writing and reading processes at once. There is also an additional function needed to remove an id from the spatial index when the respective object is deleted from the ZODB database. But for a small project where you don’t want to interact inconveniently with a database using SQL, this method should be sufficient.

References

[1] Agarwal and Rajan (2017); Analyzing the performance of NoSQL vs. SQL databases for Spatial and Aggregate queries; Free and Open Source Software for Geospatial (FOSS4G) Conference Proceedings: Vol. 17, Article 4

[2] Zope Foundation: ZODB - a native object database for Python
http://www.zodb.org/ (16.12.2018)
update 11.02.2025: https://zodb.org/en/latest/

[3] Zope Community: Welcome to Zope Project and Community
http://www.zope.org/ (16.12.2018)
update 11.02.2025: https://www.zope.dev/

[4] Carlos de la Guardia and the Zope community, 2010: ZODB Book
https://zodb.readthedocs.io/en/latest/introduction.html (16.12.2018)
update 11.02.2025: https://zodb.org/en/latest/introduction.html

[5] ZODB Developers: automatic persistence for Python objects
https://persistent.readthedocs.io/ (16.12.2018)
update 11.02.2025: https://persistent.readthedocs.io/en/latest/

[6] Howard Butler et. al., 2011: Rtree: Spatial indexing for Python
http://toblerity.org/rtree/ (16.12.2018)
update 11.02.2025: https://toblerity.org/rtree/

[7] Marios Hadjieleftheriou, 2012: libspatialindex 1.8.0 documentation
http://libspatialindex.github.io/ (16.12.2018)
update 11.02.2025: https://libspatialindex.org/en/latest/

Appendix

The Geometry Classes

```python
import persistent
class Geometry(persistent.Persistent):
    type = None
    SRID = None

    def getType(self):
        return self.type

    def getSRID(self):
        return self.SRID

    def wkt(self):
        return self.getType() + " " + self.text()

    def text(self):
        pass

    def getPoints(self):
        pass

    def bbox(self):
        points = self.getPoints()
        x, y = zip(*points)
        return (min(x), min(y), max(x), max(y))
```

```python
class Point(Geometry):
    def __init__(self, x, y, SRID=None):
        self.x = x
        self.y = y
        self.type = "POINT"
        self.SRID = SRID

    def coordinates(self):
        return (self.x, self.y)

    def text(self):
        return "(%d %d)" % (self.x, self.y)

    def getPoints(self):
        return self.coordinates()

    def bbox(self):
        return (self.x, self.y, self.x, self.y)
```

```python
class LineString(Geometry):
    def __init__(self, points, SRID=None):
        self.points = points
        self.type = "LINESTRING"
        self.SRID = SRID

    def text(self):
        t = "("
        for point in self.points:
            t += point.text()[1:-1] + ", "
        t = t[0:-2] + ")"
        return t

    def getPoints(self):
        pointList = []
        for point in self.points:
            pointList.append(point.coordinates())
        return pointList
```

```python
class Polygon(Geometry):
    def __init__(self, outerRing, innerRing=None, SRID=None):
        points = outerRing.getPoints()
        if points[0] == points[-1] and len(points) > 2:
            self.outerRing = outerRing
            self.innerRing = innerRing
            self.type = "POLYGON"
            self.SRID = SRID
        else:
            raise ValueError("linear ring with minimum 3 points expected")

    def getPoints(self):
        m = self.outerRing.getPoints()
        if self.innerRing is not None:
            m += self.innerRing.getPoints()
        return m

    def text(self):
        t = "(" + self.outerRing.text()
        if self.innerRing is not None:
            t += "," + self.innerRing.text()
        return t + ")"
```

Daniel Scherer, M.Eng.
Deutsches Geodätisches Forschungsinstitut (DGFI-TUM)
TUM School of Engineering and Design
Technische Universität München
email: daniel.scherer@tum.de

<u>Note from Franz Xaver Schütz:</u>
Daniel Scherer wrote the first version of his article for FORTVNA PAPERS I in December 2018. I had invited him to this contribution because he had used the object-oriented database zodb with python in my master course, which I didn't know about until then. Unfortunately, the volume FORTVNA PAPERS I can only be published and printed now. However, his ideas and his contribution are still of great interest.