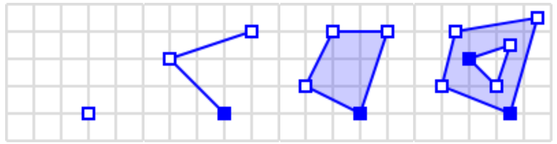
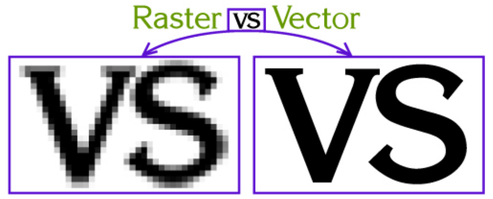
Handling (Big) Geospatial Data

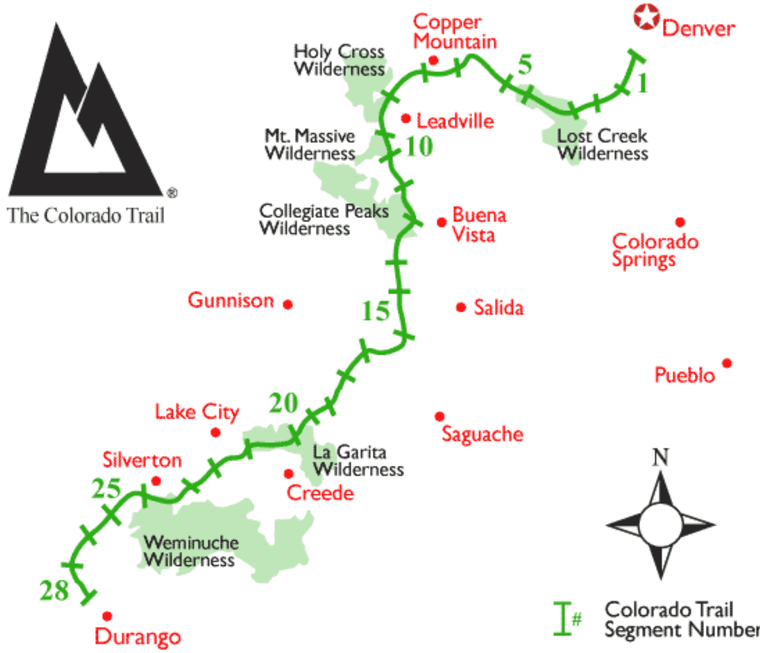
Geospatial (GIS) Data

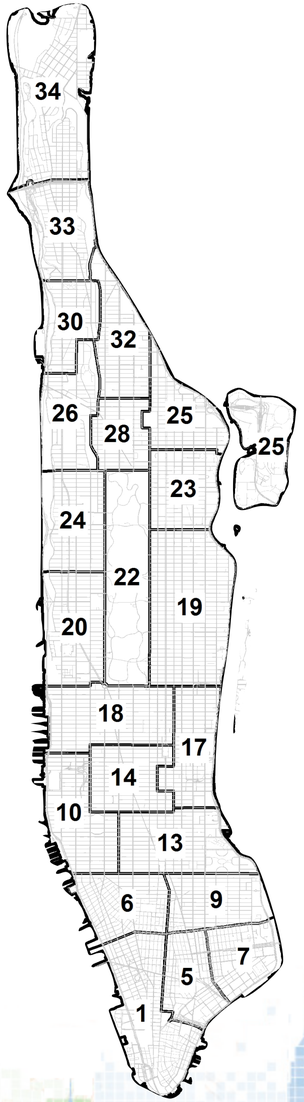
- Data with explicit geographic locations or features
- Vector data:



  - Points: GPS coordinates, point of interests
  - Lines: road networks, bike trails
  - Polygon: state boundaries, park areas
  - 3D Shapes: city models, terrain mapping
- Raster data: geo-referenced images
  - Satellite photos, overlay visualizations



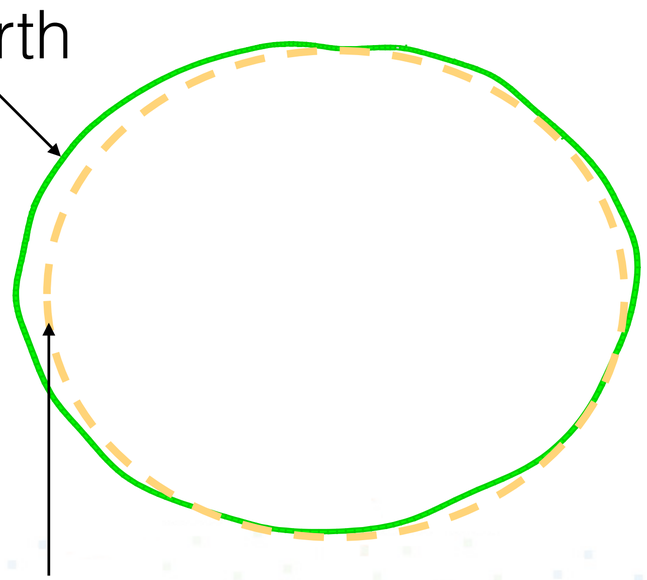


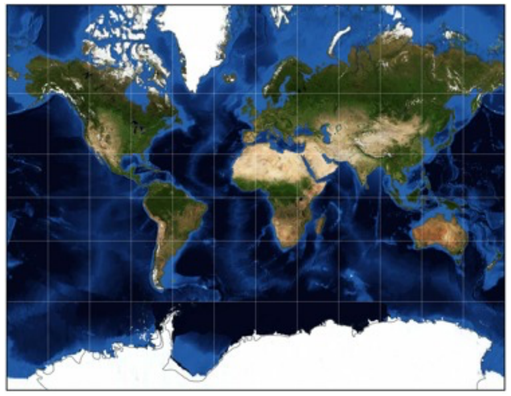


[Images © Wikipedia, runtrails.net, landsurveyorsunited.com, The Office of the Manhattan Borough President]

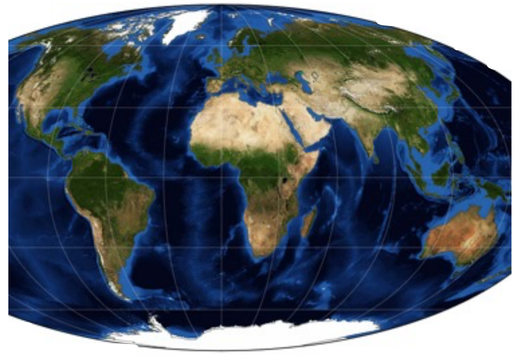
Geographic Coordinate System: Datum and Projection

- Datum: how the earth is being modeled (as an ellipsoid)
- Projection: flattening the 3D ellipsoid onto a 2D surface





Mercator
(pretty)



Mollweide
(equal-area)

Azimuthal
(equidistant)

Spatial Representation

- Vector geometry: sequences of coordinates — e.g. latitude and longitude
  - Point: `(40.7127,-74.0059)` or `(583968.1,4507339.1,10)`
  - Lines/Polygon: `[(40.7127,-74.0059), (40.8127,-74.0059)]`
  - Geometric operations (point in polygon test, intersection test, etc.) are performed on primitives
- Raster geometries: image + metadata (e.g. enclosed geographic bounds)
  - Geometric operations are performed on pixels/fragments
- Location information: postal address, place name, BBL, zipcode, etc.
  - Often get geocoded (convert geographic coordinates) before processing

Spatial Format — Plain text

- Best for embedding in documents, human readable
  - WKT: well-known text, purely geometries, lots of support from well-known DBs

```
POINT (-74.0059 40.7127)

LINESTRING (-74.0059 40.7127,-74.0059 40.8127)
```

  - GeoJSON: support attributes, web app friendly (there're also TopoJSON/KML)

```
{"type": "Feature",
 "geometry": {
  "type": "Point",
  "coordinates": [-74.0059,40.7127]
 },
 "properties": {"name": "My Point"}}
```

  - SVG: scalable vector graphics — focus on rendering (not geographic purely)

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <rect x="25" y="25" width="200" height="200" fill="lime" stroke-width="4" stroke="pink" />
  <circle cx="125" cy="125" r="75" fill="orange" />
  <polyline points="50,150 50,200 200,200 200,100" stroke="red" stroke-width="4" fill="none" />
</svg>
```

Spatial Format — Binary

- Best for efficient data manipulation (e.g. in databases)
  - Shapefile: from ESRI ArcGIS, a collection of files describing primitive records (.shp) , their indices (.shx) and attributes (.dbf)
    - Widely used in the GIS community (almost a “standard”)
  - WKB: well-known binary, a binary version of WKT, lots of DB support
- Most raster data are in binary
  - GeoTIFF: a TIFF image + georeferencing information
  - JPEG2000: Geography Markup Language (GML) for georeferencing

Handling Spatial Data — Python

- Most data can be stored in Python native structures: tuple, list, and dictionaries
- Useful packages for handling spatial data:
  - **`fiona`** : read/write GIS files (a thin API for the GDAL/OGR I/O library)
  - **`geopandas`** : extending **`pandas`** with geo support
  - **`json`** : a built-in package for reading JSON files including GeoJSON
  - **`pyproj`**: map projection (conversion from one projection to another)
  - **`shapefile`** : a light-weight package for reading shapefiles
  - **`shapely`** : provide geometric operations on 2D planes (oblivious to geographic projections), based on PostGIS engine GEOS

Handling Spatial Data — Conversion

GDAL/OGR provides a powerful command line tool for conversion (and transformation) of geospatial data, similar to ImageMagick's convert: **`ogr2ogr`**

https://trac.osgeo.org/gdal/wiki/DownloadingGdalBinaries

- Convert shapefile to geojson

```
ogr2ogr -f GeoJSON nyc.geojson nyc.shp
```

- … also reproject the data to EPSG:4326 coordinates (~GPS lat lon):

```
ogr2ogr -f GeoJSON -t_srs EPSG:4326 nyc.geojson nyc.shp
```

Handling Spatial Data — Projection

```
>>> import pyproj

## Create the MapPLUTO projection EPSG:2263
>>> proj = pyproj.Proj(init='EPSG:2263', preserve_units=True)

>>> nycLatitude  = 40.692547
>>> nycLongitude = -73.928113

## Map the NYC lon/lat coordinates onto MapPLUTO
>>> proj(nycLongitude, nycLatitude)
    (1004185.1129963363, 191598.17059893996)

## Map a MapPLUTO coordinates back to lon/lat
>>> proj(1004185.1129963363, 191598.17059893996, inverse=True)
    (-73.928113, 40.69254699999981)
```

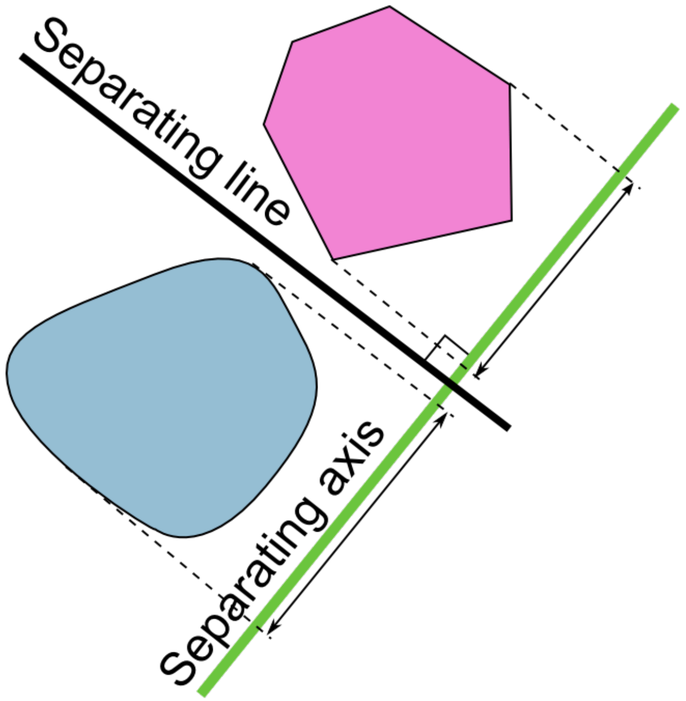
Spatial Query & Spatial Join

- Query vs. Join — common questions: ***where are my nearby objects?***
- Sample Queries:
  - Which neighborhoods does a bike path or a bus line pass by?
  - Which BBLs are part of a flooded zone?
- Joining a list of points with a list of polygons:
  - Determine neighborhoods of taxi pickups
  - Map check-in locations to BBLs
  - Find all subway stations within 1 mile of some GPS locations

How do we do “small” spatial joins?

- *Problem: given two sets of geometries A and B, find which geometries in B that intersects with each geometry in A*
- Naively slow:

```
joins = {}
for a in A:
    joins[a] = []
    for b in B:
        if a.intersects(b):
            joins[a].append(b)
```



- |A| x |B| intersection tests
- How costly is an intersection test?
  - e.g. Separating Axis Theorem — O(|a| x |b|) or ~(|a|x|b|) operations

[Images © Wikipedia]

How do we do “small” spatial joins?

How do we do “small” spatial joins?

- *Problem: given two sets of geometries A and B, find which geometries in B that intersects with each geometry in A*

How do we do “small” spatial joins?

- *Problem: given two sets of geometries A and B, find which geometries in B that intersects with each geometry in A*
- Using geopandas *sjoin(A, B, how, op)*:

How do we do “small” spatial joins?

- *Problem: given two sets of geometries A and B, find which geometries in B that intersects with each geometry in A*
- Using geopandas *sjoin(A, B, how, op)*:

```
joins = geopandas.tools.sjoin(A, B)
```

How do we do “small” spatial joins?

- *Problem: given two sets of geometries A and B, find which geometries in B that intersects with each geometry in A*
- Using geopandas *sjoin(A, B, how, op)*:

```
joins = geopandas.tools.sjoin(A, B)
```

- Is *sjoin(A,B)* the same as *sjoin(B,A)*?

How do we do “small” spatial joins?

- *Problem: given two sets of geometries A and B, find which geometries in B that intersects with each geometry in A*
- Using geopandas *sjoin(A, B, how, op)*:

```
joins = geopandas.tools.sjoin(A, B)
```

- Is *sjoin(A,B)* the same as *sjoin(B,A)*?
  - By default, *sjoin* builds a structure for B to speedup intersection tests

How do we do “small” spatial joins?

- *Problem: given two sets of geometries A and B, find which geometries in B that intersects with each geometry in A*
- Using geopandas *sjoin(A, B, how, op)*:

```
joins = geopandas.tools.sjoin(A, B)
```

- Is *sjoin(A,B)* the same as *sjoin(B,A)*?
  - By default, *sjoin* builds a structure for B to speedup intersection tests
  - |A| x log(|B|) *box* intersection tests + a few *polygon* intersection tests

How do we do “small” spatial joins?

- *Problem: given two sets of geometries A and B, find which geometries in B that intersects with each geometry in A*
- Using geopandas *sjoin(A, B, how, op)*:

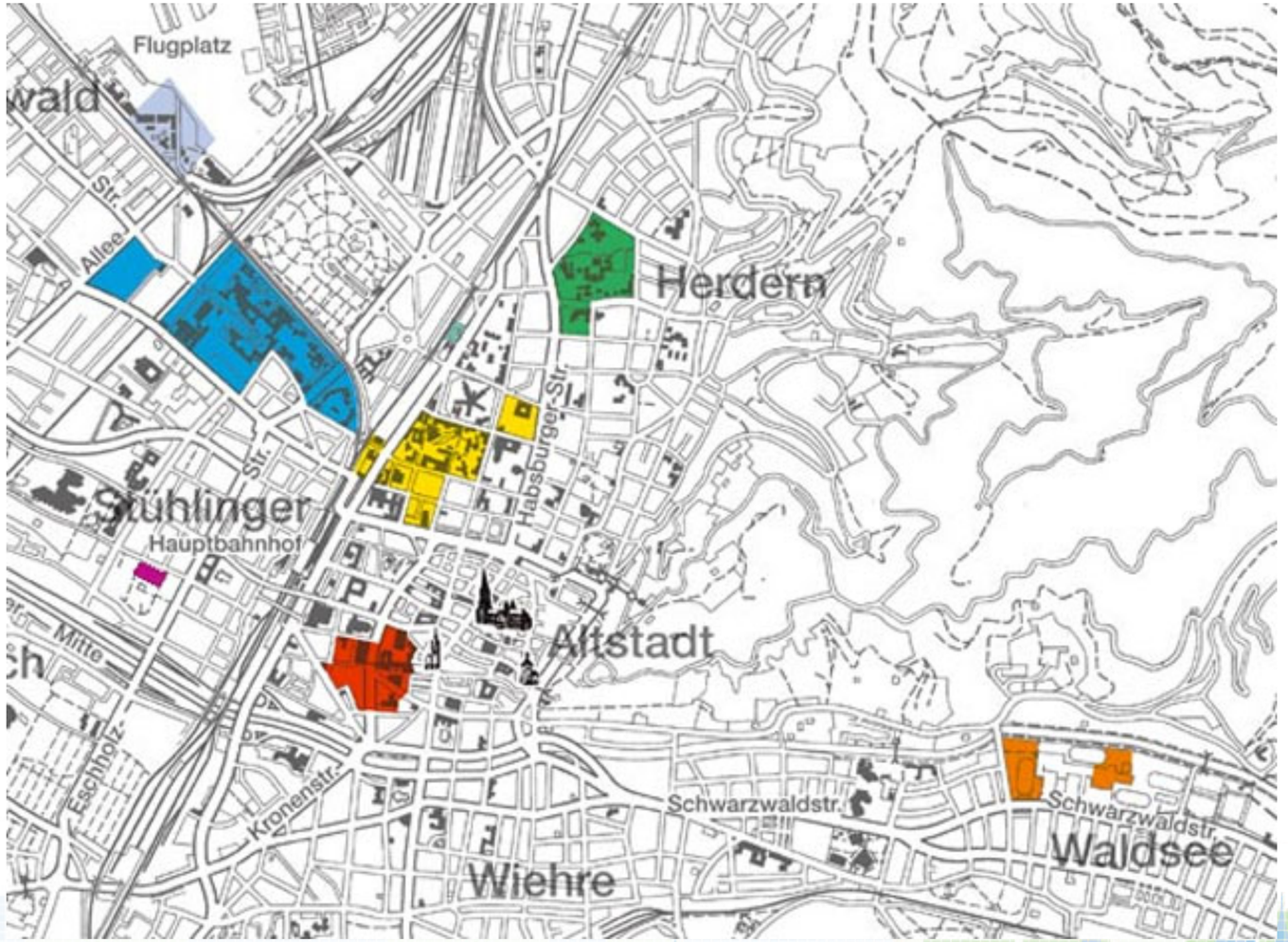
```
joins = geopandas.tools.sjoin(A, B)
```

- Is *sjoin(A,B)* the same as *sjoin(B,A)*?
  - By default, *sjoin* builds a structure for B to speedup intersection tests
  - |A| x log(|B|) *box* intersection tests + a few *polygon* intersection tests

Spatial Index

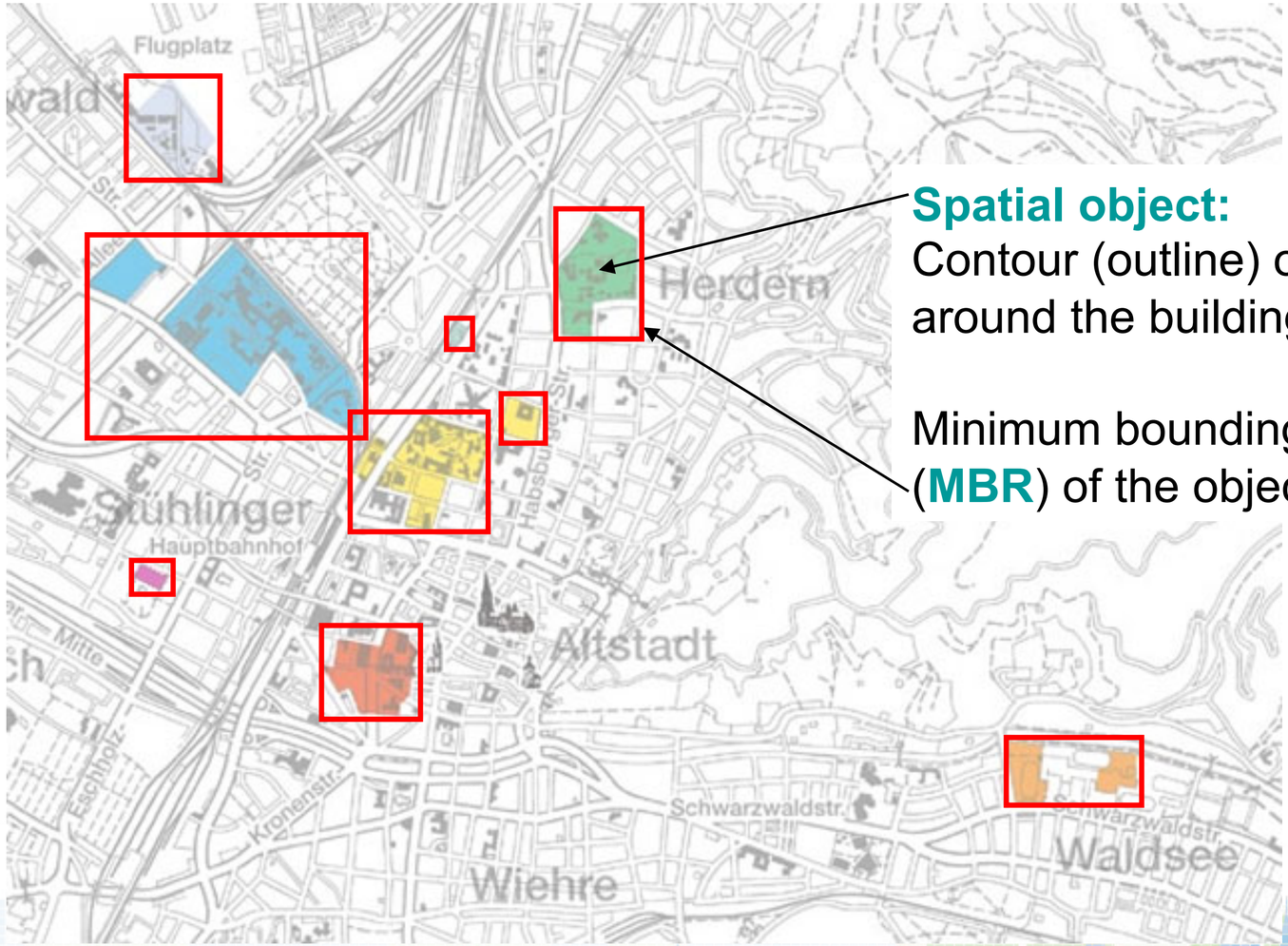
Spatial Index

- Consider: Given a city map, ‘index’ all university buildings in an efficient structure for quick relational-topological search:
  - Find buildings within an area



Spatial Index — MBR for faster search

- Consider: Given a city map, ‘index’ all university buildings in an efficient structure for quick relational-topological search:
  - Faster to find building bounding regions within an area

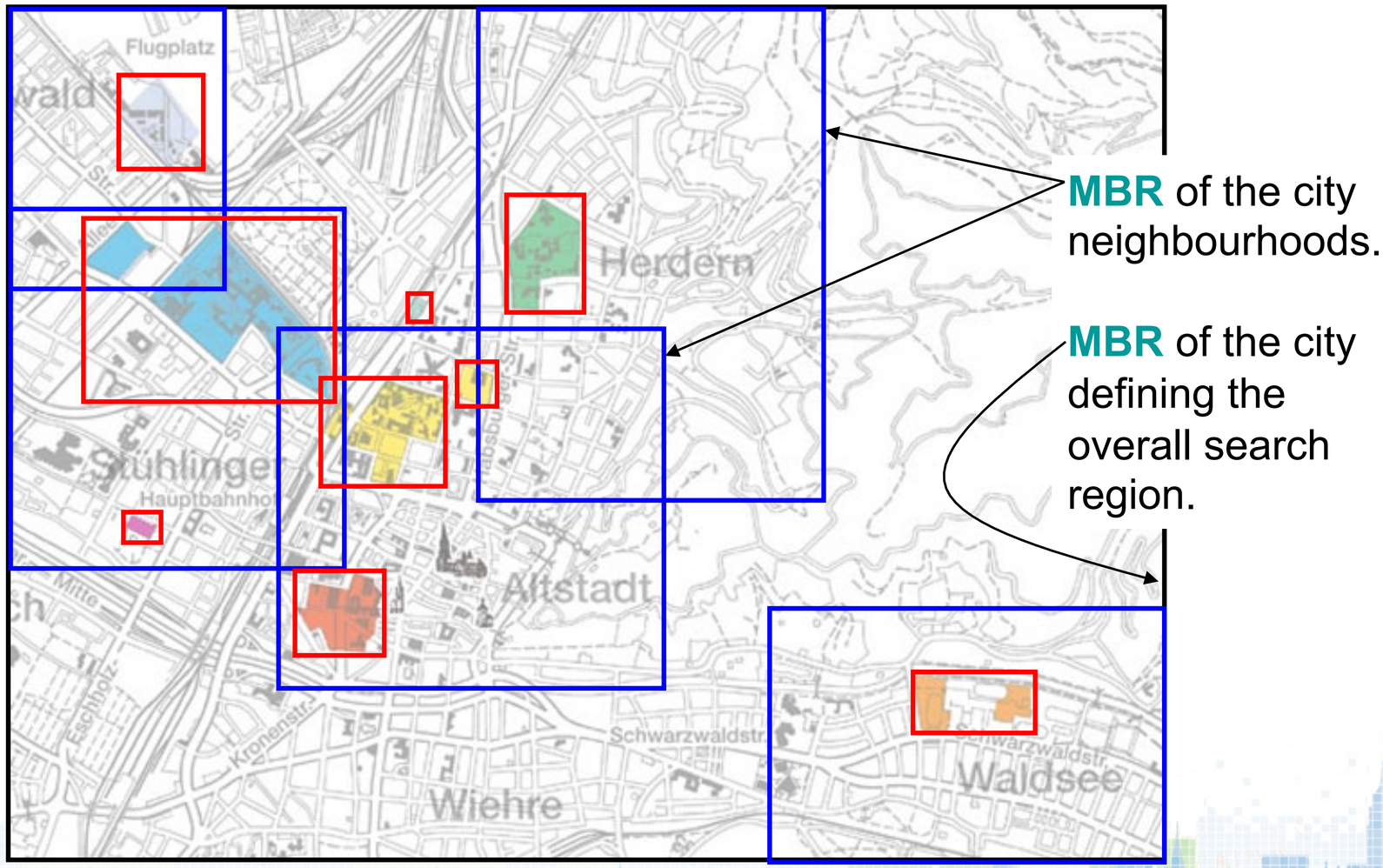


MBR Intersection

- If 2 MBRs do not intersect, the bounded spatial objects do not intersect
- If 2 MBRs intersect, the bounded spatial objects ***might*** be intersecting

Spatial Index — Hierarchical Structure

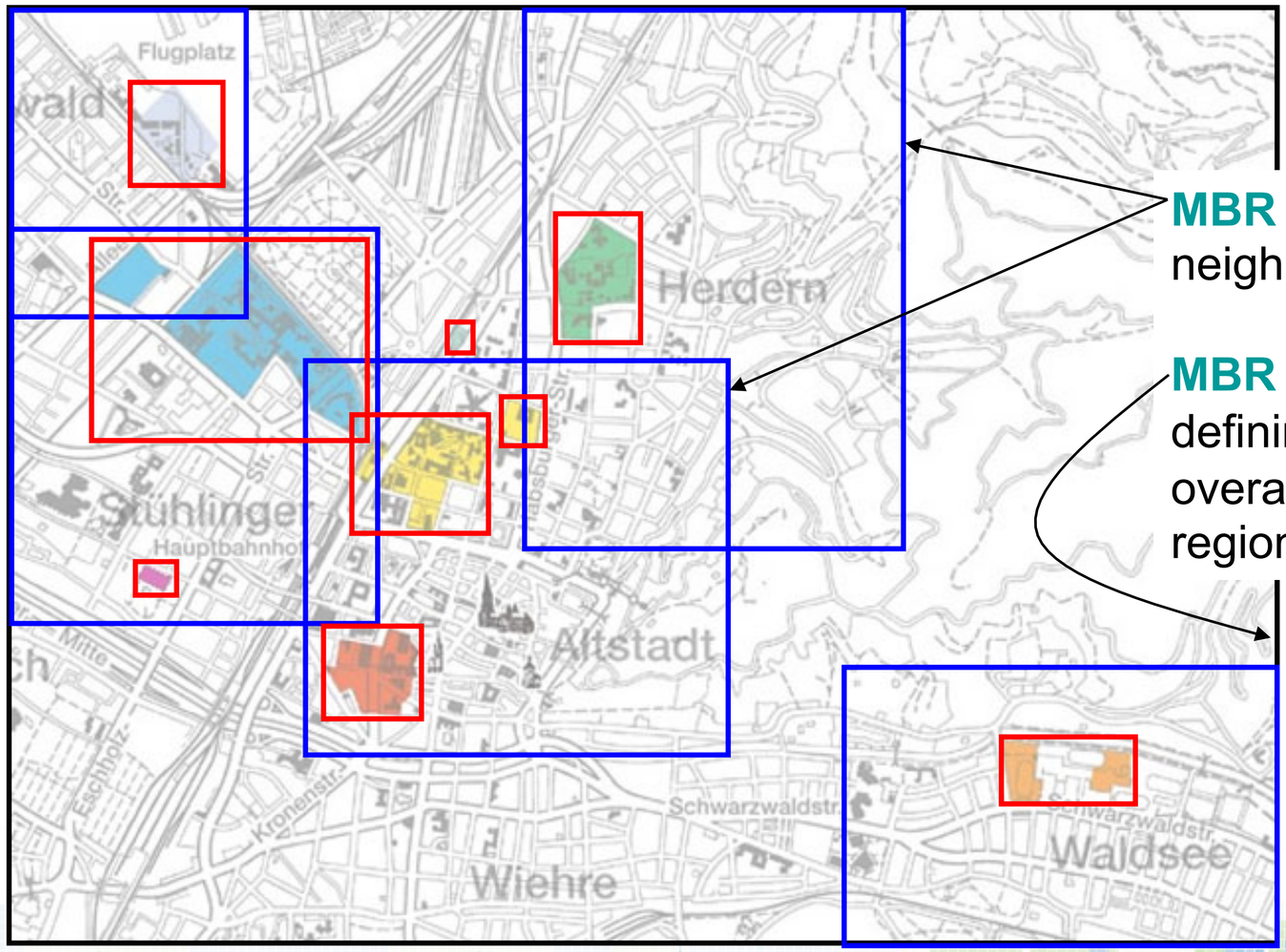
- Consider: Given a city map, ‘index’ all university buildings in an efficient structure for quick relational-topological search:
  - Divide and conquer, search for bounding regions hierarchically



Spatial Index — False Positive Match

- Consider: Given a city map, ‘index’ all university buildings in an efficient structure for quick relational-topological search

  - For each matched MBRs first, **test the actual polygons**



Space Partitioning Data Structure

- R-Tree, kd-Tree, Octtree, Quadtree
- Complexity of the naive approach:
  - O(|A| x |B|), aka touching every polygon at least once
- Using Space Partitioning data structures:
  - Often reduce to O(|A| x log(|B|))
- R-Tree is the most popular in Databases
  - Supports both read and write, however, we'll be using read() mostly

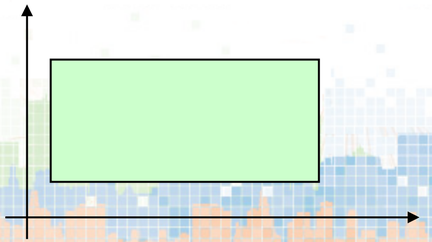
R-Tree Index

- A spatial database consists of a collection of tuples representing spatial objects, known as Entries.
- Each Entry has a unique identifier that points to one spatial object, and its MBR; i.e. Entry = (MBR, pointer).
- An entry ***E*** in a leaf node is defined as (Guttman, 1984):

  E = (I, tuple-identifier)

- Where ***I*** refers to **the smallest binding** n-dimensional region (MBR) that encompasses the spatial data pointed to by its tuple-identifier.
- ***I*** is a series of ***closed-intervals*** that make up each dimension of the binding region.

  Example. In 2D, ***I*** = (***Ix***, ***Iy***), where ***Ix*** = [***xa***, ***xb***], and ***Iy*** = [***ya***, ***yb***].

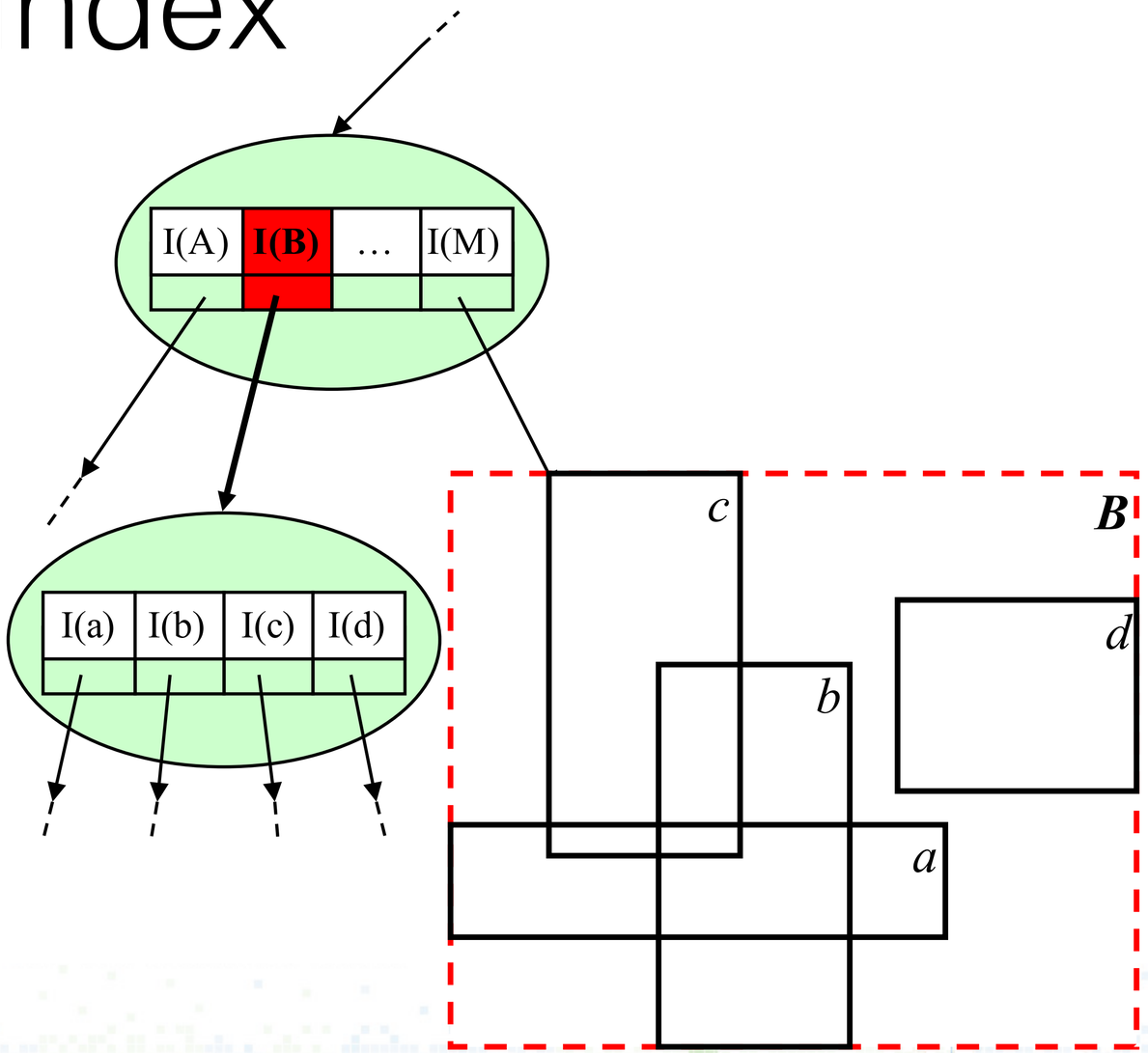


R-Tree Index

- An **entry** *E* in a *non-leaf* node is defined as:

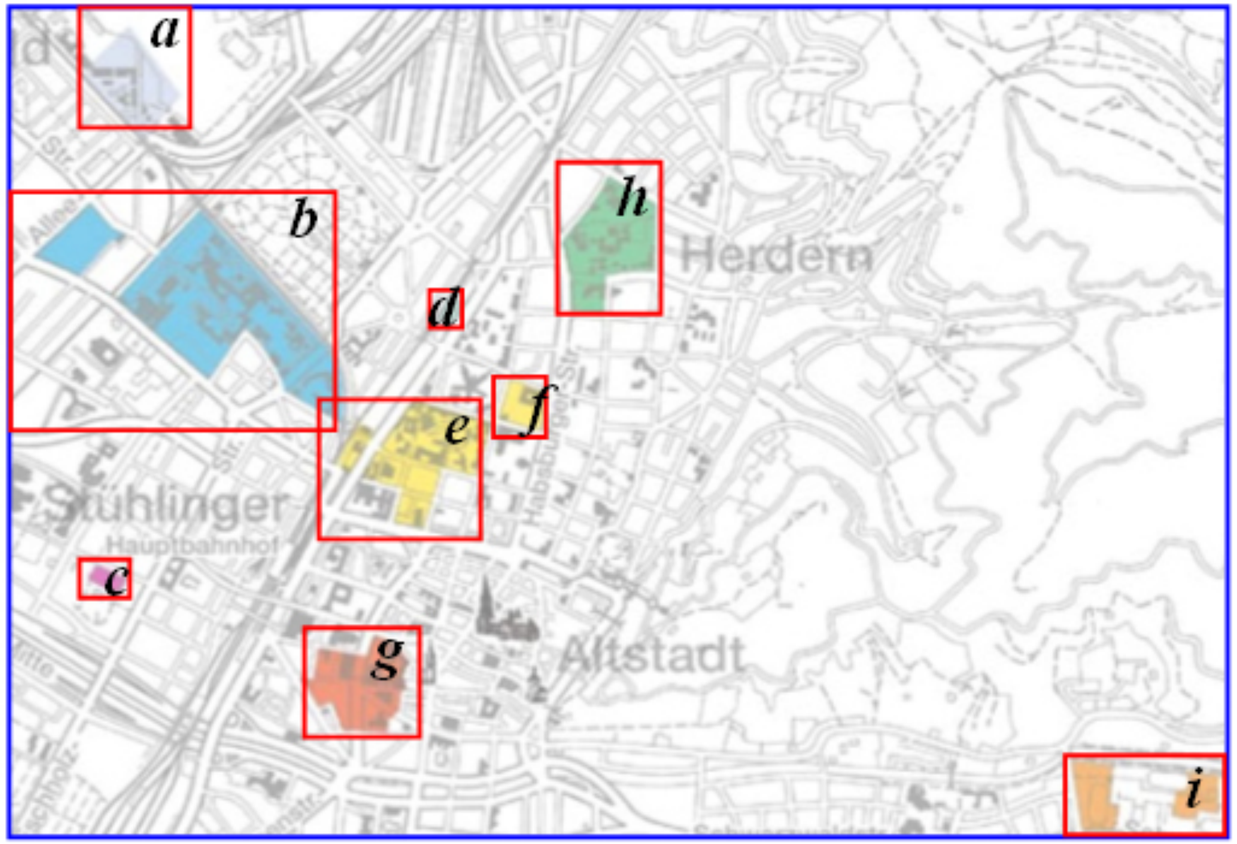
  $E = (I, \textit{child-pointer})$

where the *child-pointer* points to the child of this node, and I is the ***MBR*** that encompasses all the regions in the ***child-node's pointer's entries***.

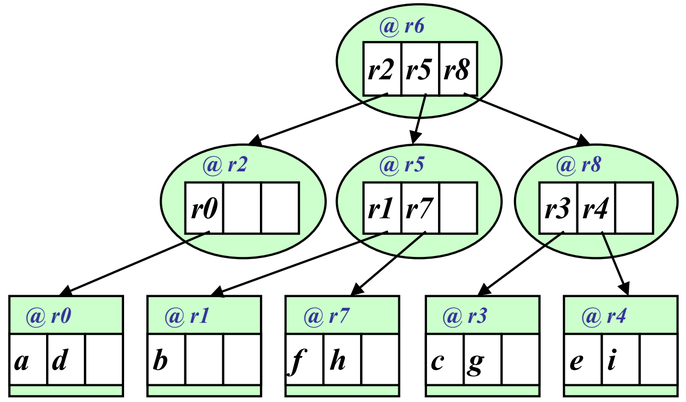


R-Tree Search

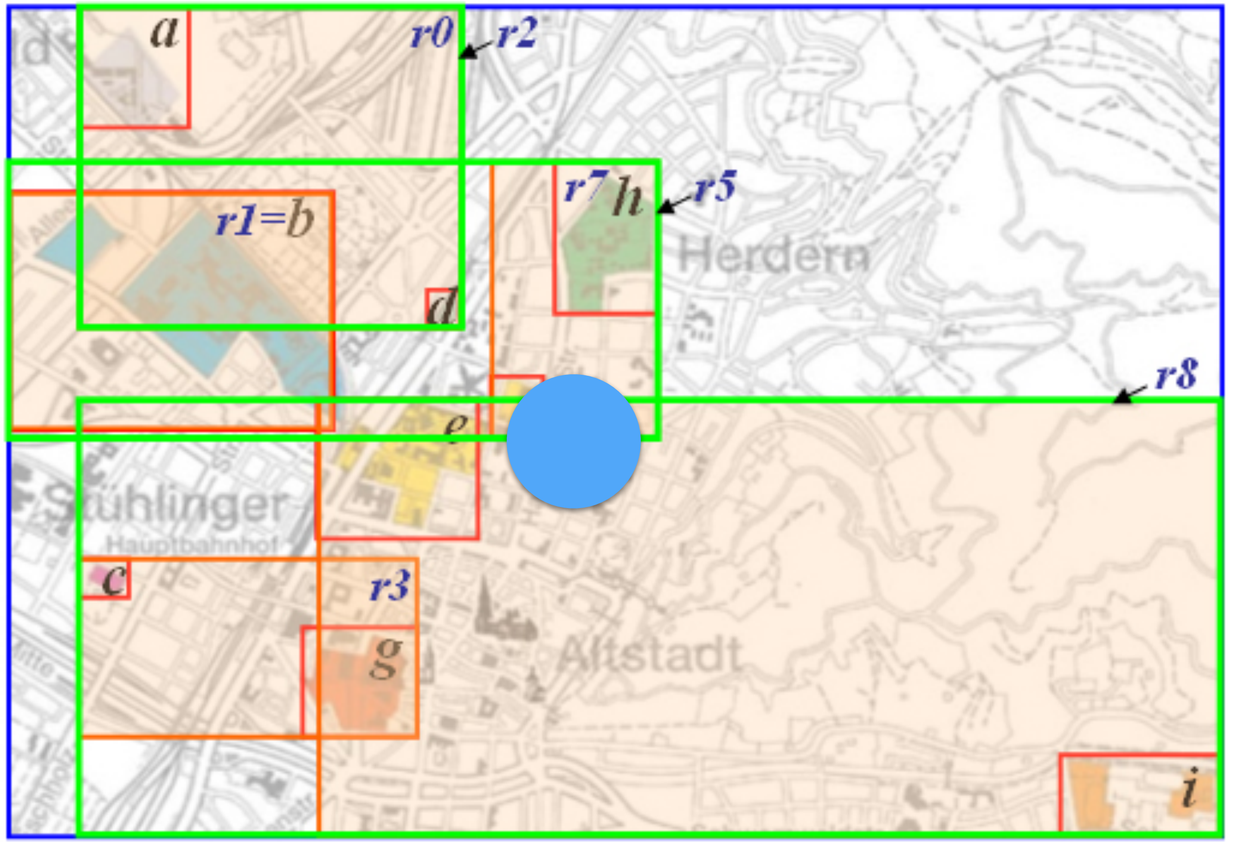
- *Find and report all university building sites that are within 5km of the city centre.*
- Approach:
  - Build the R-Tree using rectangular regions a, b, … i.
  - Formulate the query range Q.
  - Query the R-Tree and report all regions overlapping Q.



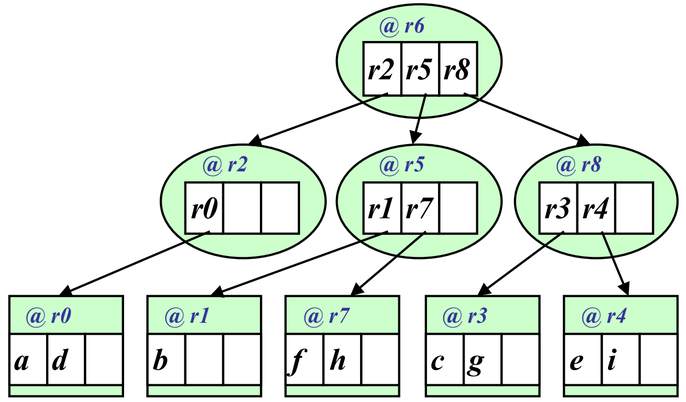
R-Tree Search



- *Find and report all university building sites that are within 5km of the city centre.*
- Approach:
  - Build the R-Tree using rectangular regions a, b, … i.
  - Formulate the query range Q.
  - Query the R-Tree and report all regions overlapping Q.

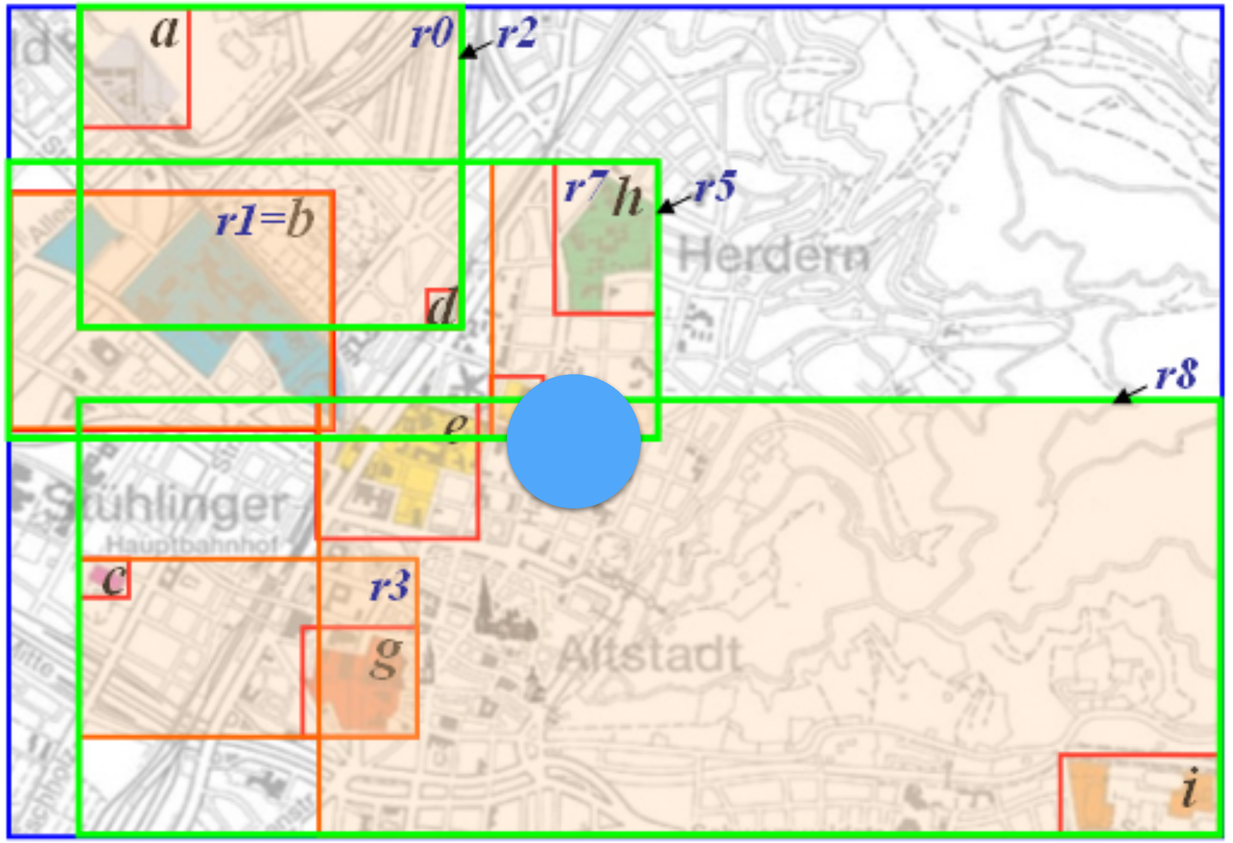


R-Tree Search



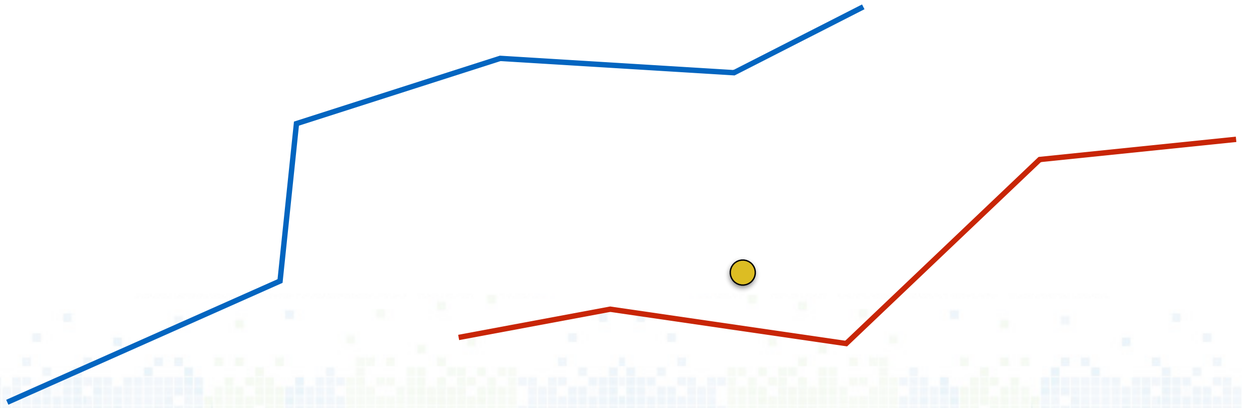
depth of 3

- *Find and report all university building sites that are within 5km of the city centre.*
- Approach:
  - Build the R-Tree using rectangular regions a, b, … i.
  - Formulate the query range Q.
  - Query the R-Tree and report all regions overlapping Q.



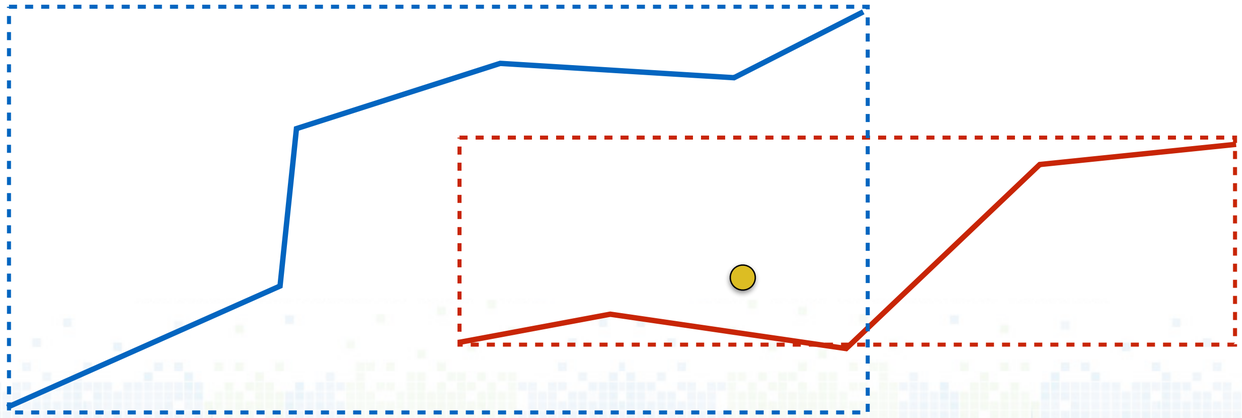
Spatial Index on Complex Geometries

- Spatial index works well on simple primitives: points, lines, and polygons.
- What about complex geometries such as MultiPoint, LineString, MultiLineString, MultiPolygon, etc.?
  - Identifying trajectories that a GPS coordinate belong to:



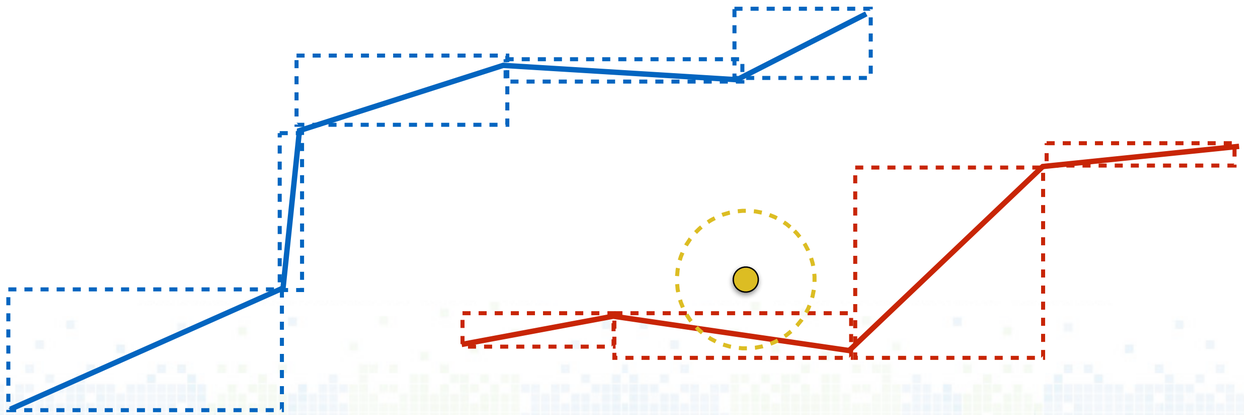
Spatial Index on Complex Geometries

- Spatial index works well on simple primitives: points, lines, and polygons.
- What about complex geometries such as MultiPoint, LineString, MultiLineString, MultiPolygon, etc.?
  - Identifying trajectories that a GPS coordinate belong to:



Spatial Index on Complex Geometries

- Spatial index works well on simple primitives: points, lines, and polygons.
- What about complex geometries such as MultiPoint, LineString, MultiLineString, MultiPolygon, etc.?
  - Decompose into simple primitives, then apply spatial index



Using Spatial Index

1. Compute bounding boxes and assign unique IDs for input geometries
   - When doing the join, selecting the one with fixed geometries (or usually smaller) as the one to be indexed
2. Create an index based on the bounding boxes
3. Perform spatial search on a geometry (e.g. *which area it belongs to?*)
   - Create a bounding box of the interested area around the geometry
   - Use the index to find potential overlapping bounding boxes and IDs
   - Use the IDs to retrieve the actual input geometries in Step 1, then test for real spatial intersection

How to use R-Tree in Python?

- https://github.com/geopandas/geopandas/blob/master/geopandas/tools/sjoin.py
  - An add-on tool for geopandas, taken care of bounding box mapping
  - Index is not persistent and will be rebuilt on every call
- Directly uses the **rtree** package (has to maintain bounding boxes manually)

```
>>> from rtree import index
>>> idx = index.Index()
>>> left, bottom, right, top = (0.0, 0.0, 1.0, 1.0)
>>> idx.insert(0, (left, bottom, right, top))
>>> list(idx.intersection((1.0, 1.0, 2.0, 2.0)))
[0]
>>> list(idx.intersection((1.00001, 1.00001, 2.0, 2.0)))
[]
>>> idx.insert(1, (left, bottom, right, top))
>>> list(idx.nearest((1.0000001, 1.0000001, 2.0, 2.0), 1)) [0, 1]
```

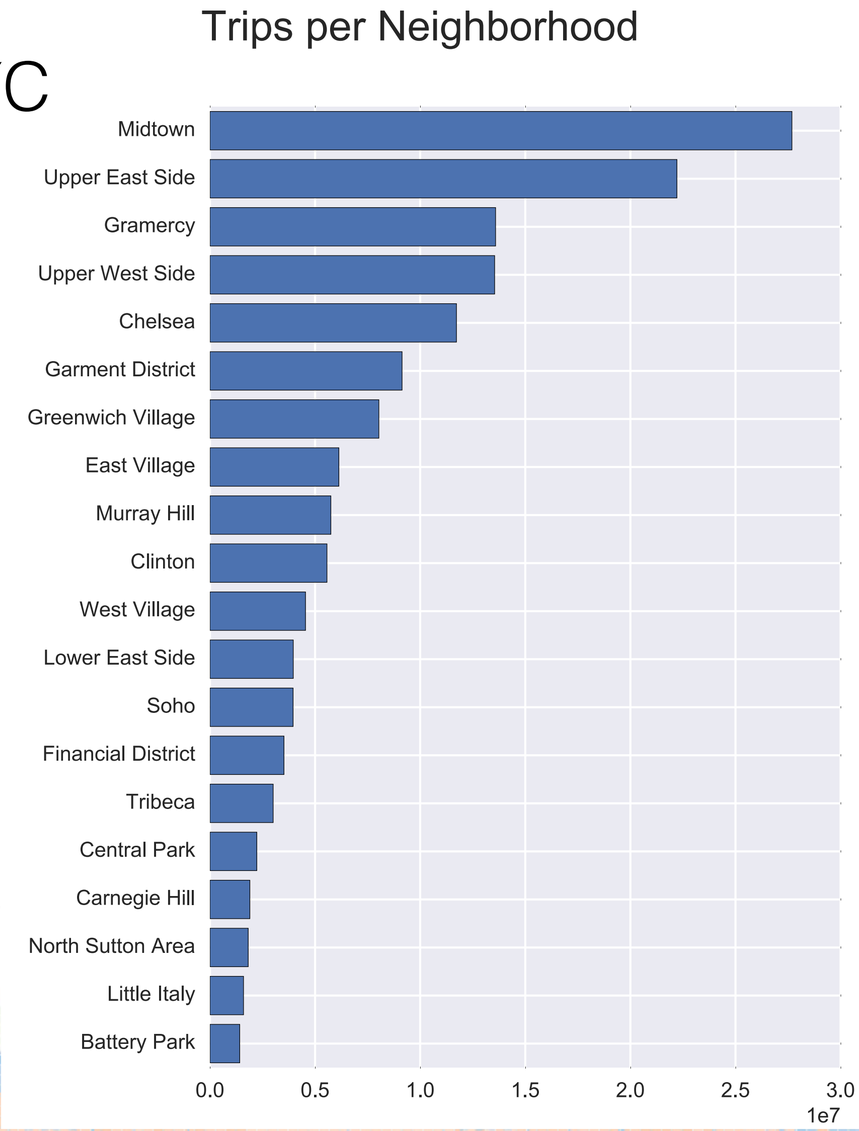
http://toblerity.org/rtree/tutorial.html

Using Spatial Index at Scale

- Multiple spatial searches can be performed using a single R-Tree
  - Build an R-tree of all searchable regions
  - Searches on the elements in parallel using this tree
- The R-tree must be shared across all partitions
  - For a small to moderate number of regions, rebuild the R-tree for each partitions
  - For a large number of regions, build the R-tree in the driver

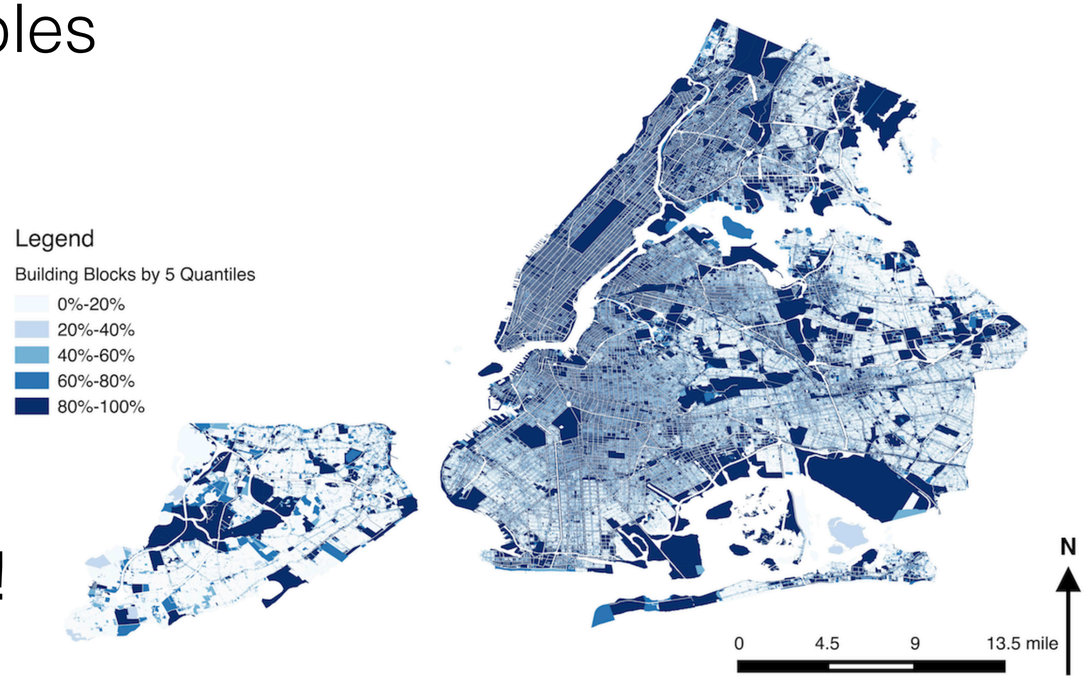
Example: Count Taxi Trips per Region

- Finding the most picked up neighborhood in NYC
- Algorithm:
  - Build an R-Tree for neighborhoods
  - Map points into neighborhood IDs
  - Count the number of trips per each ID
- Benefits of using spatial index: 100x speedup
  - 8 Box + ~2 Polygon tests per trip (vs. 100s)



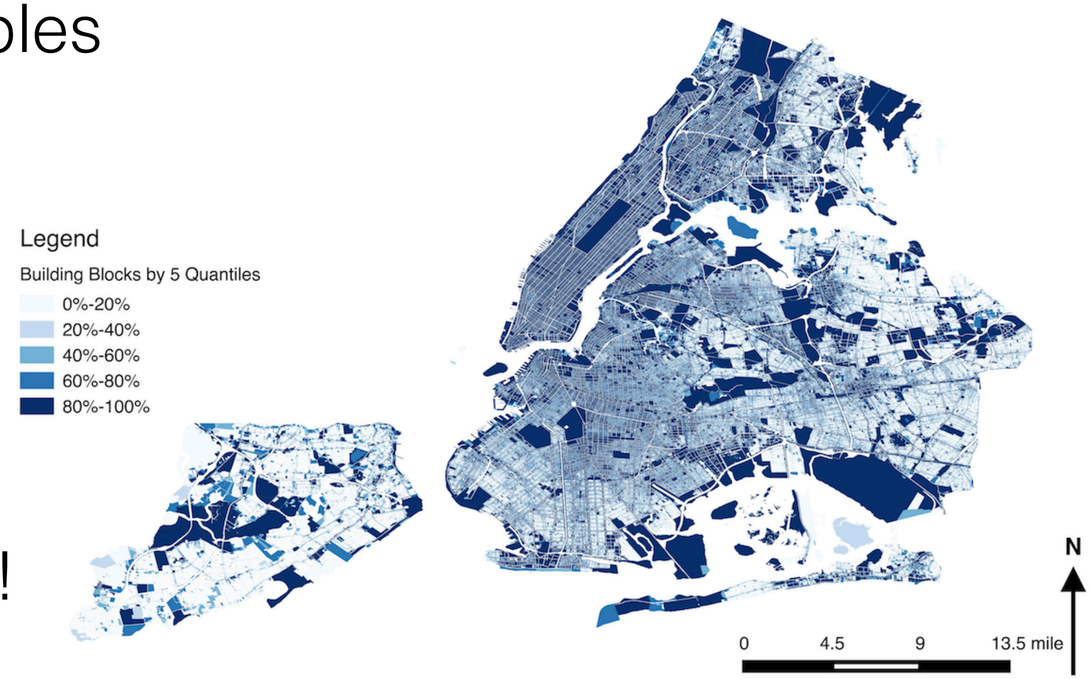
Example: Count Taxi Trips per BBL!

- A lot more input geometries: ~1 million BBLs
- A lot more points to test: ~4 billion samples
- Without any indices:
  - 4000 trillion Polygon tests!
- Using spatial index:
  - 4 billion x (20 box + ~4 Polygon) tests!
  - Took 40 minutes using 256 cores



Example: Count Taxi Trips per BBL!

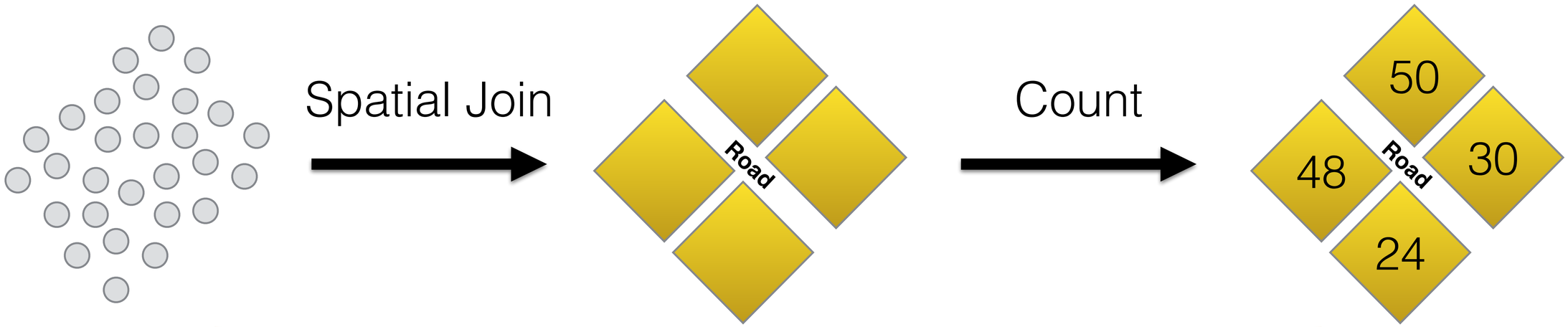
- A lot more input geometries: ~1 million BBLs
- A lot more points to test: ~4 billion samples
- Without any indices:
  - 4000 trillion Polygon tests!
- Using spatial index:
  - 4 billion x (20 box + ~4 Polygon) tests!
  - Took 40 minutes using 256 cores



Still very expensive!

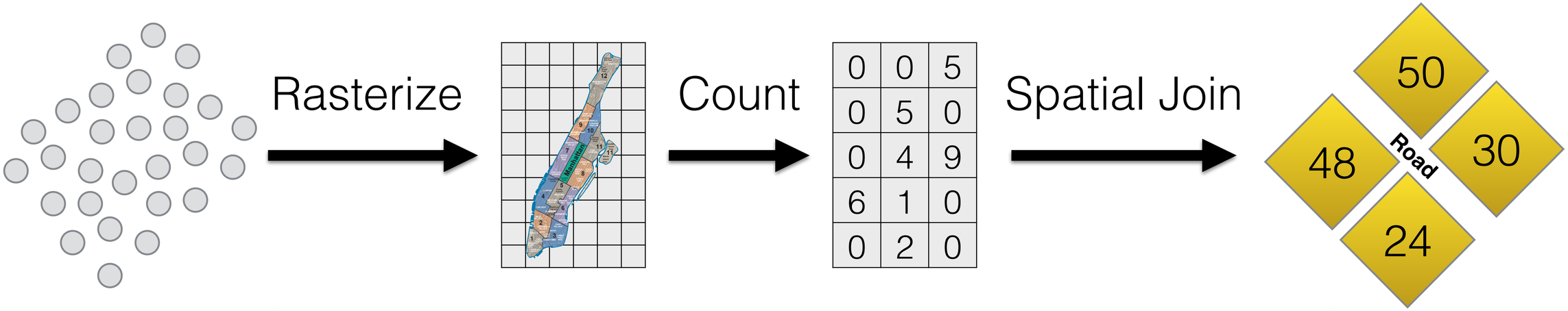
Example: Count Taxi Trips per BBL!

- Another approach:
  - Rasterize counts before spatial joins
- Before: 4 billion x (20 box + ~4 Polygon) tests!



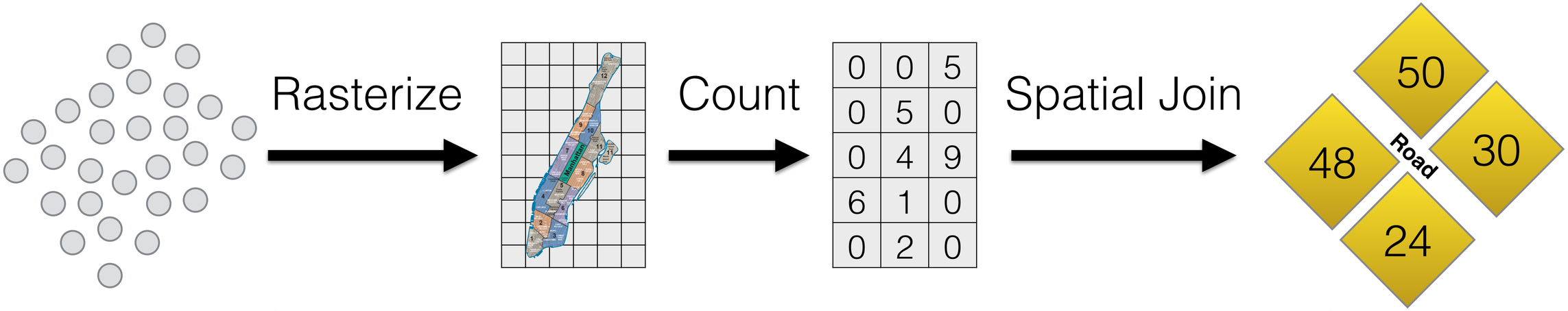
Example: Count Taxi Trips per BBL!

- Another approach:
  - Rasterize counts before spatial joins
- After



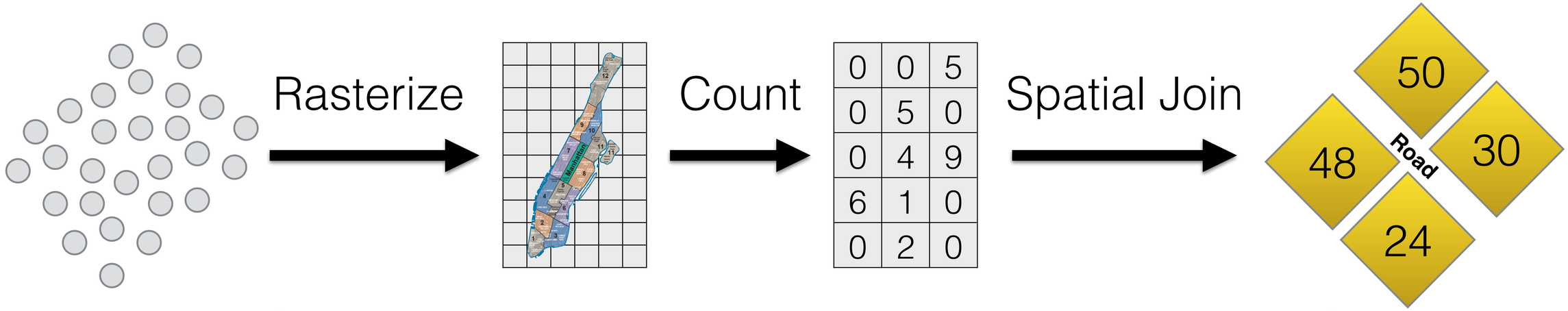
Example: Count Taxi Trips per BBL!

- Another approach:
  - Rasterize counts before spatial joins
- After



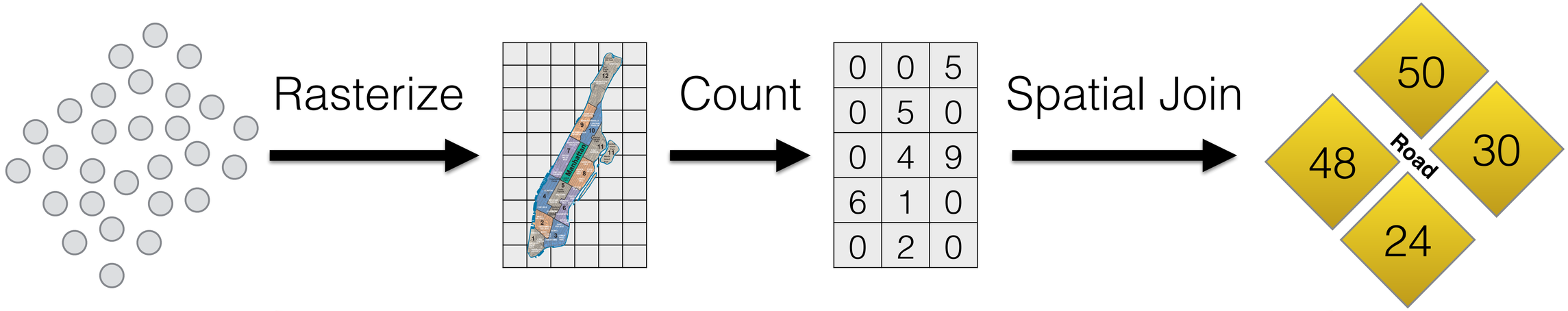
Example: Count Taxi Trips per BBL!

- Another approach:
  - Rasterize counts before spatial joins
- After 4 billion lookups + millions x (20 Box + ~4 Polygon) tests!



Example: Count Taxi Trips per BBL!

- Another approach:
  - Rasterize counts before spatial joins
- After 4 billion lookups + millions x (20 Box + ~4 Polygon) tests!



10 minutes

Spatial Queries for Big Data

- A very active research area, recent efforts:
  - SpatialHadoop (2013–)
  - Hadoop-GIS (2013–), ~3D (2015–)
  - GeoSpark (2015–)
  - LocationSpark (2016–)
  - GeoMatch (2016–)
- For PySpark, it is still best to handle our own spatial operations
  - Only geopandas *sjoin* sparingly since builds an r-tree for every call, and keep everything in memory

Note on PySpark's Spatial Queries

- If your shapefile is small enough, ship the shapefile with your jobs.
- If not, upload them to HDFS (preferable in GeoJSONs), and use WholeTextFiles to read the data within the Driver and broadcast them to all workers.
- Build a spatial index (e.g. an R-Tree) **once per partition**
- If you do a lot of spatial joins, please consider using convex hulls, which are usually much smaller than actual geometries, thus, also faster in running spatial joins.
- Initialize all your packages (fiona, pyproj, etc.) within your function

Deploying Spatial Spark Applications

```
spark-submit \
  --conf spark.executorEnv.LD_LIBRARY_PATH=$LD_LIBRARY_PATH \
  --files myshapefile.geojson \
  myscript.py outputfolder
```

Notes: myshapefile.geojson is a local file, not to be hosted on HDFS

Next - running on HPC?