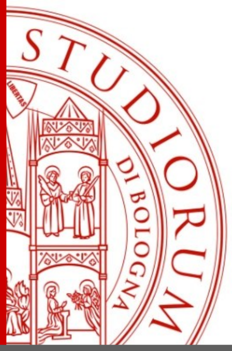


Designing Distributed Geospatial Data-Intensive Applications

Ph.D. Course, **2022**

Instructors:

Prof. **Luca Foschini**, Associate Professor &

Dr. **Isam Mashhour Al Jawarneh**, Postdoctoral Research Fellow

{isam.aljawarneh3, Luca.foschini}@unibo.it

Department of Computer Science and Engineering (DISI), Università di Bologna

IL PRESENTE MATERIALE È RISERVATO AL PERSONALE DELL'UNIVERSITÀ DI BOLOGNA E NON PUÒ ESSERE UTILIZZATO AI TERMINI DI LEGGE DA ALTRE PERSONE O PER FINI NON ISTITUZIONALI

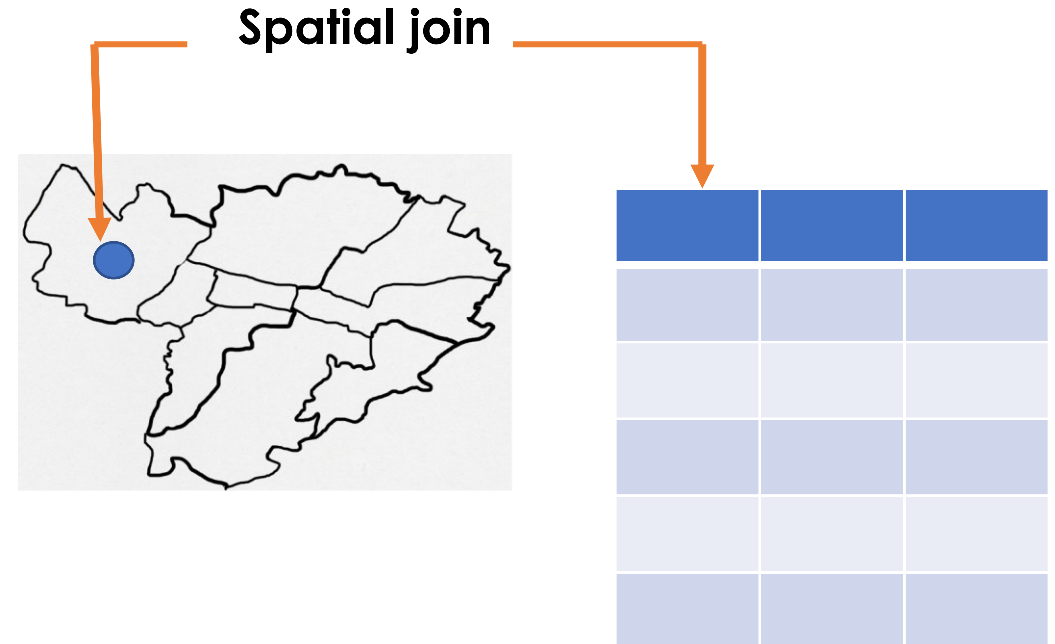
Part 2

Designing highly efficient geospatial data-intensive solutions

22nd July 2022

Spatial join

- Spatial joins are essential in spatial data analysis
  - Combining data from various tables by exploiting spatial relationships (contains, within, etc.,) as the **join key**
  - Most kinds of spatial analysis can be **expressed** as **spatial joins**



SQL-like Example

Spatial joins are joins of two relations, with a geospatial predicate function within the WHERE clause (SQL)

```
-- how many stations within 1 mile range of each zip code?
SELECT
   zip_code AS zip,
   ANY_VALUE(zip_code_geom) AS polygon,
   COUNT(*) AS bike_stations
FROM
   `bigquery-public-data.new_york.citibike_stations` AS bike_stations,
   `bigquery-public-data.geo_us_boundaries.zip_codes` AS zip_codes
WHERE ST_DWithin(
       zip_codes.zip_code_geom,
       ST_GeogPoint(bike_stations.longitude, bike_stations.latitude),
       1609.34)
GROUP BY zip
ORDER BY bike_stations DESC
```

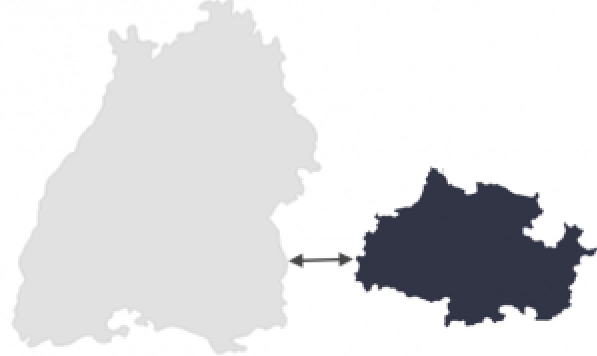
Code source

Types of Spatial Join

Based on the **spatial relationships**

Intersect

Within a distance



Closest

Completely within

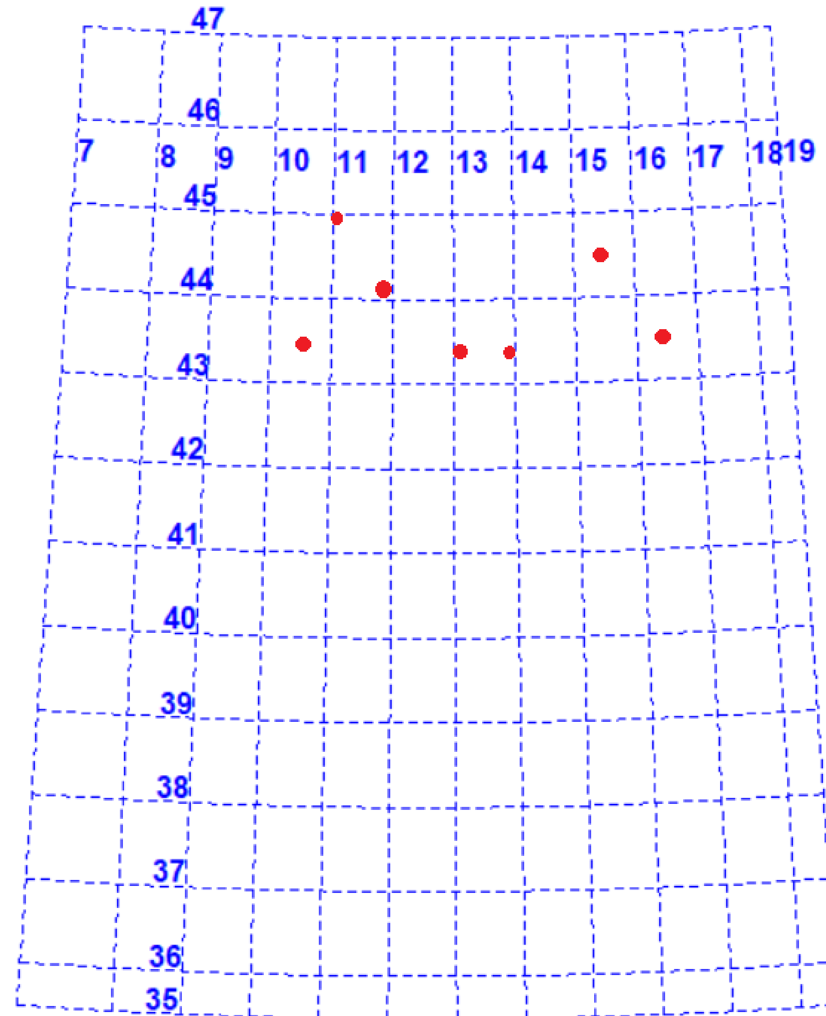
equals

[Images sources]

Spatial join examples

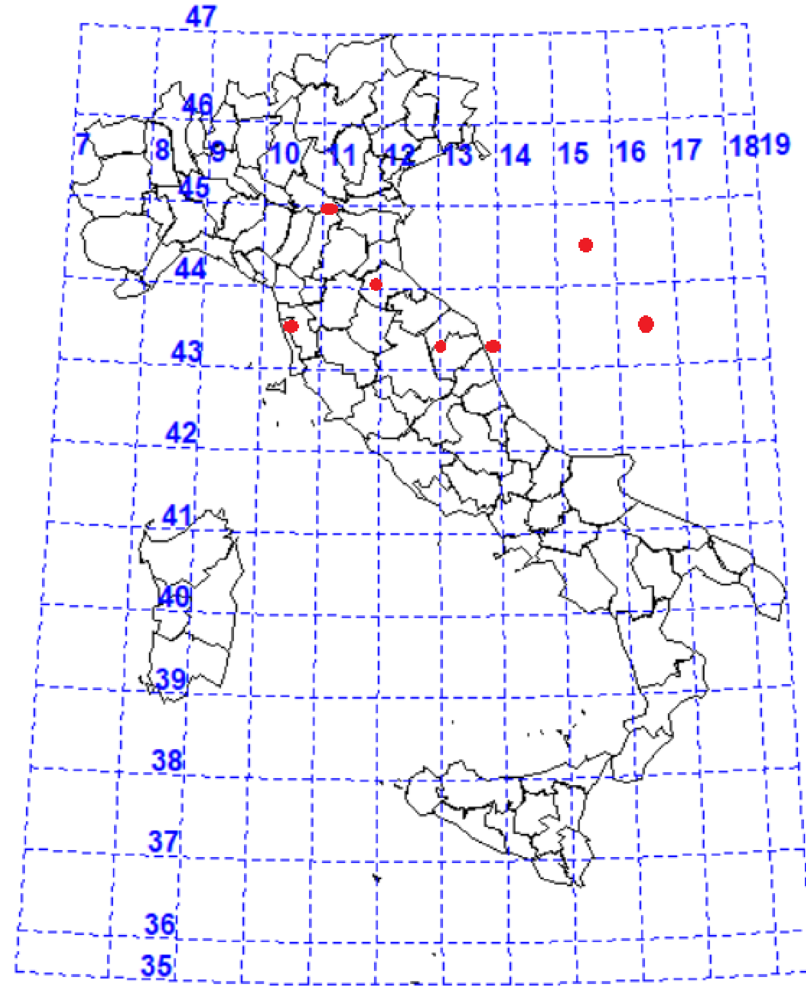
1. Supermarkets (**points**) are within a specific neighborhood (**polygon**). Spatial join affix neighborhood attributes to supermarket locations.
2. Every district (**polygon**) is responsible for maintaining its roads (**lines**). Using spatial join, each road record will add a column specifying to which district it belongs.
3. Cars (**points**) circulating in city roads (**lines**). By using spatial join, we can specify the road segment which the car navigated at a specific moment.

Parametrized spatial data



Embedding area polygons

Overlaying maps



Spatial join

$$R1 \bowtie_{\theta} R2 = \sigma_{\theta}(R1 \times R2)$$

- given: spatial objects o1, o2
  find: { $o_i \in o1$, $o_j \in o2$ | $\theta(o_i$.geometry, o_j.geometry)}
  with θ : ==, intersects, within
- A kind of Theta-join, which is computationally expensive
  - Links tables based on a **spatial relationship** instead of **equality** between two attributes
- Spatial join is a set of all pairs that is formed by **pairing** two **geo-referenced** datasets while applying a spatial predicate (e.g., **intersection**, **inclusion**, etc.,)
  - The two participating sets can be representing **multidimensional** spatial objects.
    - An example spatial join "finding boroughs to which each GPS-represented spatial point (volunteer) belongs, a.k.a. geofencing",
    - which requires joining spatial points with a master table representing boroughs

Example spatial join

- Find all the gas stations within 10 miles of my office

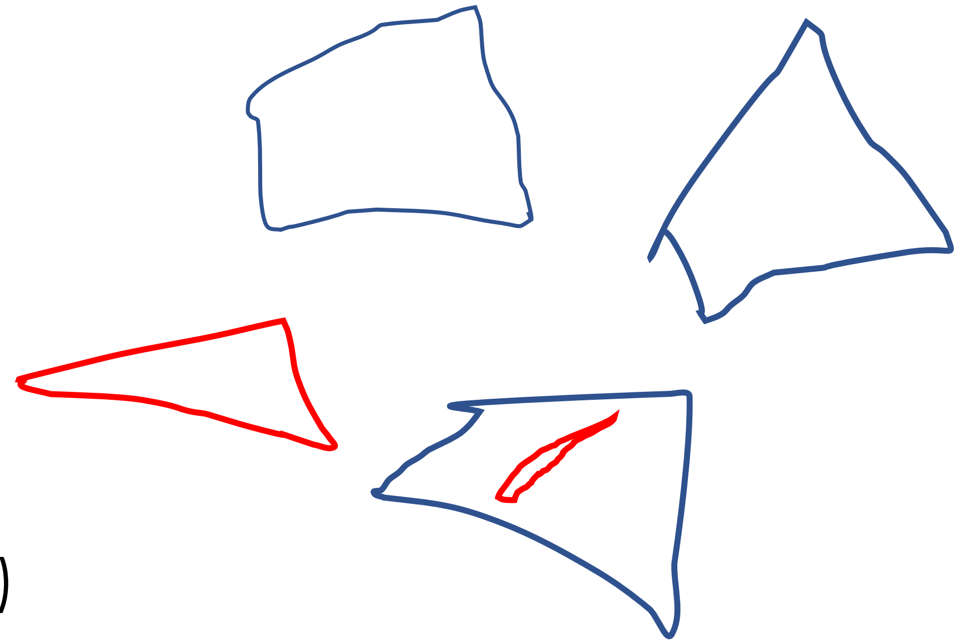
- In relational algebra terms:
$\pi_{name}(stations \bowtie_{distance\ (location, location) < 10} offices)$

```
select distinct s.name from stations
s, offices o where
distance(s.location,o.location) < 10)
```

Naïve spatial join

- Naive evaluation of spatial joins (nested loop join) too inefficient

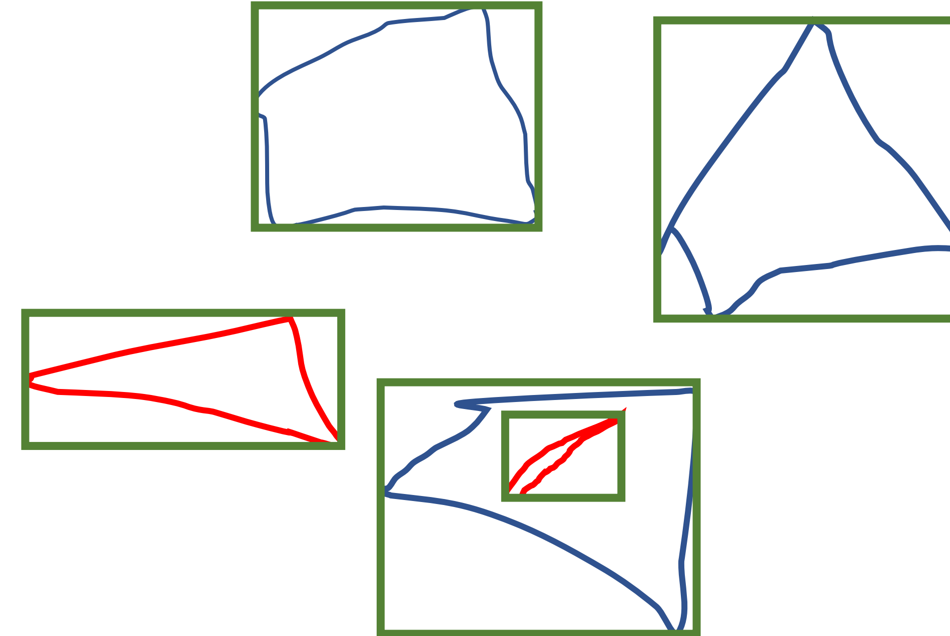
- Input: O1, O2 //objects
- Result = {Ø}
  - for all $o_i \in$ o1 do
    - for all $o_j \in$ o2 do
      - If θ ((o_i.geometry, o_j.geometry) result = result ∪ [o_i,o_j]



How many comparisons?!

Filter-refine approach

- 2 steps
  - **Filter** step
    - Determination of possible hits by evaluation on spatial approximation (lower costs)
  - **Refinement** step
    - Evaluation on accurate geometry only for objects of the filter step



```
Input: O1, O2 //spatial objects
result = {Ø}
    for all o_i ∈ o1 do
        for all o_j j o2 do
        If θ (MBR(o_i.geometry), MBR(o_j.geometry))
            If θ ((o_i.geometry, o_j.geometry)
            result = result ∪ [o_i,o_j]
```

How many comparisons?!

For efficient spatial queries, **spatial indexing** is essential

	LocationID	borough	geometry	zone
0	1	EWR	POLYGON ((-74.18445299999996 40.6949959999999,...	Newark Airport
1	2	Queens	(POLYGON ((-73.82337597260663 40.6389870471767...	Jamaica Bay
2	3	Bronx	POLYGON ((-73.84792614099985 40.87134223399991...	Allerton/Pelham Gardens
3	4	Manhattan	POLYGON ((-73.97177410965318 40.72582128133705...	Alphabet City
4	5	Staten Island	POLYGON ((-74.17421738099989 40.56256808599987...	Arden Heights

Shapefile, NYC

	tpep_pickup_datetime	tpep_dropoff_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude
0	2016-05-01 00:00:00	2016-05-01 00:17:31	-73.985901	40.768040	-73.983986	40.730099
1	2016-05-01 00:00:00	2016-05-01 00:07:31	-73.991577	40.744751	-73.975700	40.765469
2	2016-05-01 00:00:00	2016-05-01 00:07:01	-73.993073	40.741573	-73.980995	40.744633
3	2016-05-01 00:00:00	2016-05-01 00:19:47	-73.991943	40.684601	-74.002258	40.733002
4	2016-05-01 00:00:00	2016-05-01 00:06:39	-74.005280	40.740192	-73.997498	40.737564

taxi dataset

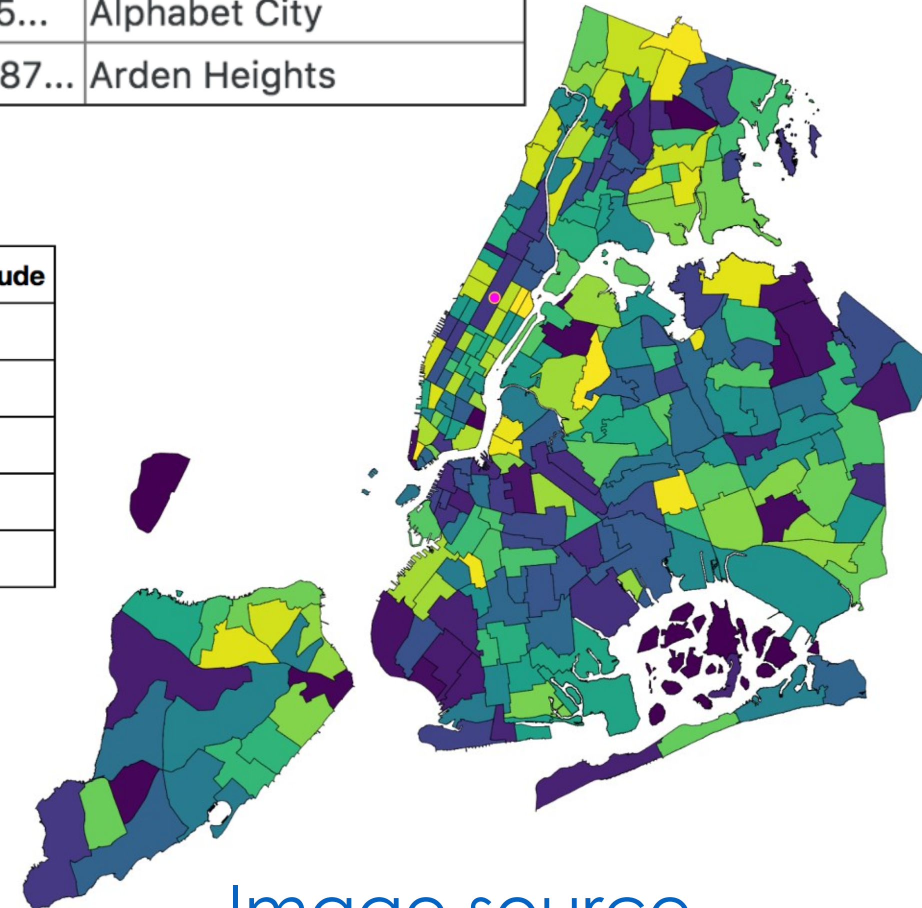


Image source

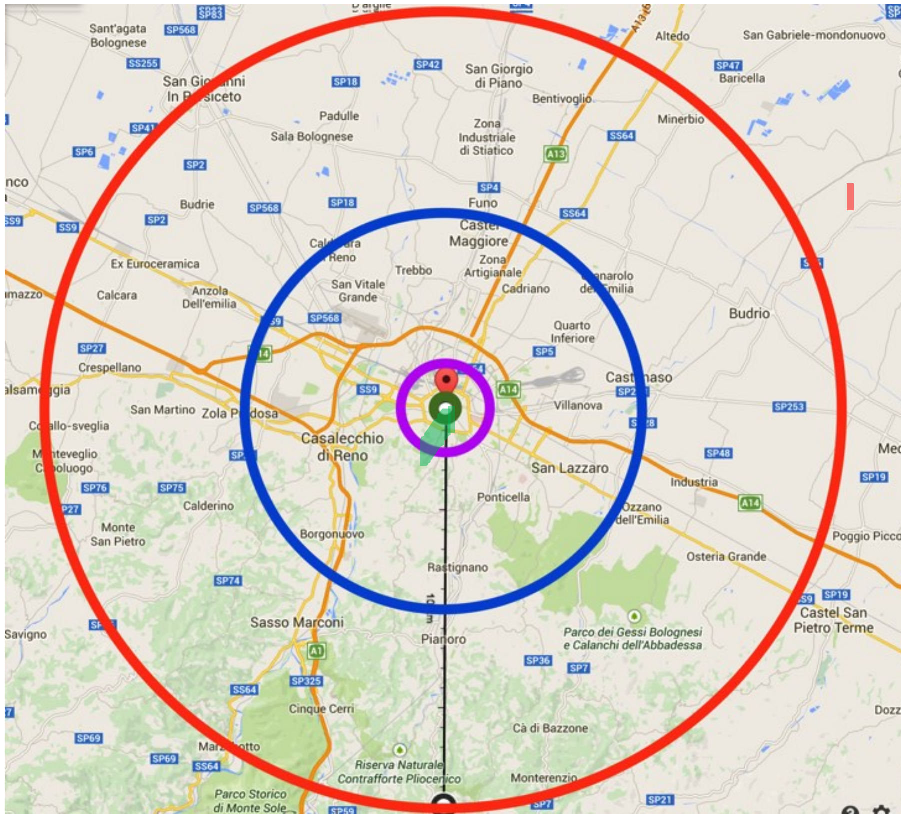
assigning trips pickups to city zones (districts) is an example of a **spatial join**

Code source

```
import geopandas as gpd
from shapely.geometry import Point
df = gpd.read_file('taxi_zones.shp').to_crs({'init': 'epsg:4326'})
df = df.drop(['Shape_Area', 'Shape_Leng', 'OBJECTID'], axis=1)
gpd.sjoin(gpd.GeoDataFrame(crs={'init': 'epsg:4326'},
geometry=[Point(-73.966, 40.78)]),
df, how='left', op='within')
```

	geometry	index_right	LocationID	borough	zone
0	POINT (-73.96599999999999 40.78)	42	43	Manhattan	Central Park

Query Test



- Proximity and containment queries executed on a circular area centered on Bologna
- Center in (44.4949,11.3426)
- Radius range from 500 m to 50 km

What kind of representation for spatial data?

- So, selected spatial data representation should facilitate **spatial operations**
  - e.g., facilitates **pruning** on **data retrieval**
- The most relevant data structure for representing spatial data is the one that is based on **spatial occupancy**
  - Decomposing the embedding space into buckets (i.e., regions)
  - Commonly known as '*bucketing methods*'

Spatial data structures

Spatial Indexing

- **Shape-aware** organization of spatial data (**objects** & **embedding** space), such that it enables **pruning** the search space in order to answer a spatial query
  - For supporting spatial **selection**, **join** and **proximity**
- Two approaches
  - **Specialized** spatial index structures: e.g., **R-Tree**, **PR quadtree**, **KD tree**, Bin-tree, etc.,
  - Dimensionality reduction: **transform** multidimensional representation of spatial objects (and space) into a **single dimension**
    - Then apply a **linear indexing** (such as **B+-tree**)

Supporting data structures

Linear & single-dimension data structures:

Indexing

Data access

- Queries normally access a small portion of data
  - Accessing the minimum number of tuples is much faster (what is the relevant **path**?)
- Design choices affecting the path:
  - Data arrangement
    - **Sequential** files, **linked** list
  - Index types
    - **Linear** index or **tree-based** or a **mix** of both!
  - **Caching** computations

Basic operations (in relational algebra and NoSQL)

– set operations (e.g., union)
– selection and projection
– join

Selective queries

- Selection query:

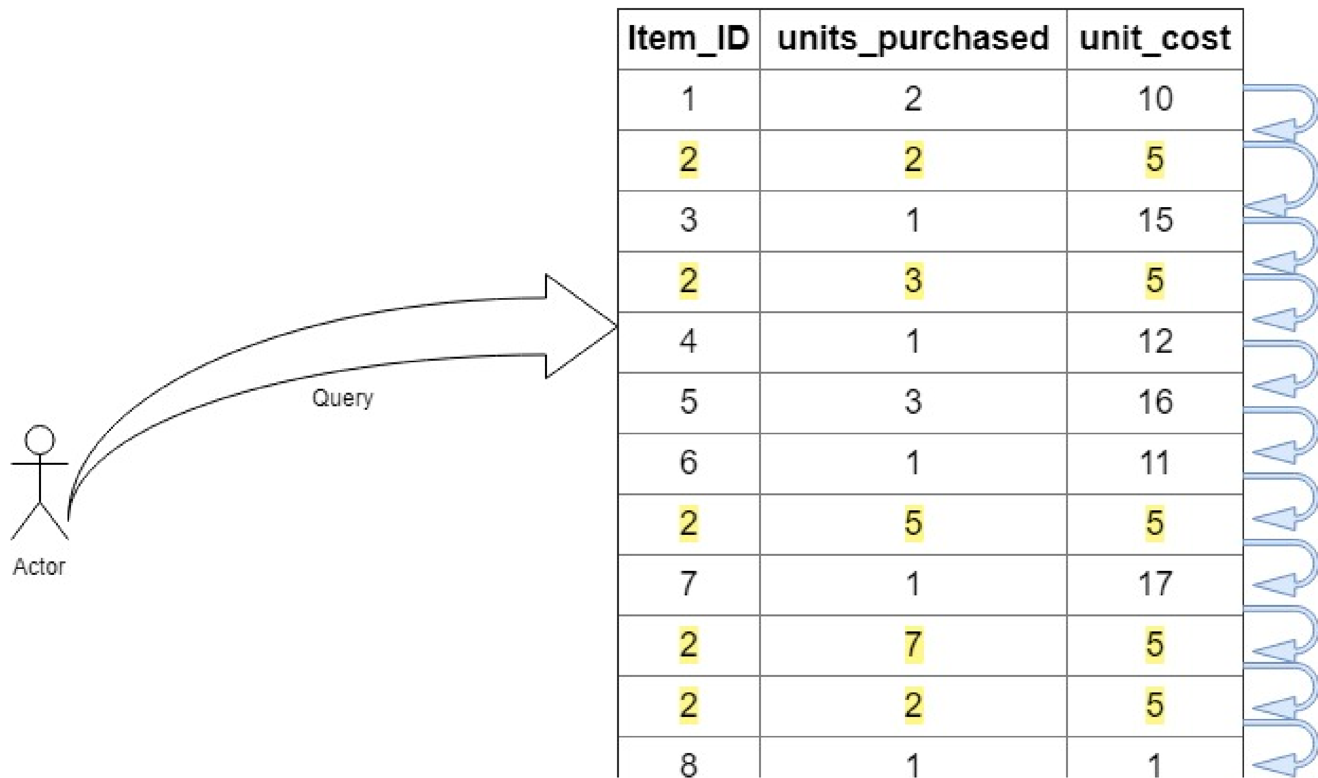
```
SELECT *
FROM R
WHERE <condition>
```

- This is fine in case you are retrieving a large portion (e.g., >80) of the tuples.
- Otherwise, if your query is highly selective (predicate selectivity is low), returning only a small portion of the tuples, then indexing provides performance optimization

Selectivity

- An indicator of how much data is retrieved by applying a selection predicate
  - A fractional number between 0 and 1
    - Selectivity 1 means all data rows will be retrieved
    - Selectivity 0 means that no data rows will be retrieved
  - Useful for estimating the cost associated with a given access method

- Example
  - Table Employee with 10000 rows
    - Select * from Employee
      - Query selectivity = 1
    - Select * from Employee where EmpID = 123
      - Selectivity = 1/10000 = 0.0001
      - Point queries are typically very highly: We need indexing



Item_ID	units_purchased	unit_cost
1	2	10
2	2	5
3	1	15
2	3	5
4	1	12
5	3	16
6	1	11
2	5	5
7	1	17
2	7	5
2	2	5
8	1	1

For point queries: we need full table scan for unindexed data

Indexing

- Think of huge data sets
  - Do not fit in **fast memory**
- Efficient ways for insert, delete and search
  - e.g., **range** query search
- keys point to data → **indexing**
  - Separate files (**index** files) containing key/value pairs
  - Keys are associated with pointers to the real data tuples (**record** files)
  - **Impose** an **order** or organization on index files using a **tree structure**
  - The most common tree indexing is **B-tree** for big disk-based data

Indexes

- To avoid **full table scans**, we need indexes
  - An **index** on an **attribute** helps finding records with specific values on that attribute without the need to do an **exhaustive** **full scan**

Indexing

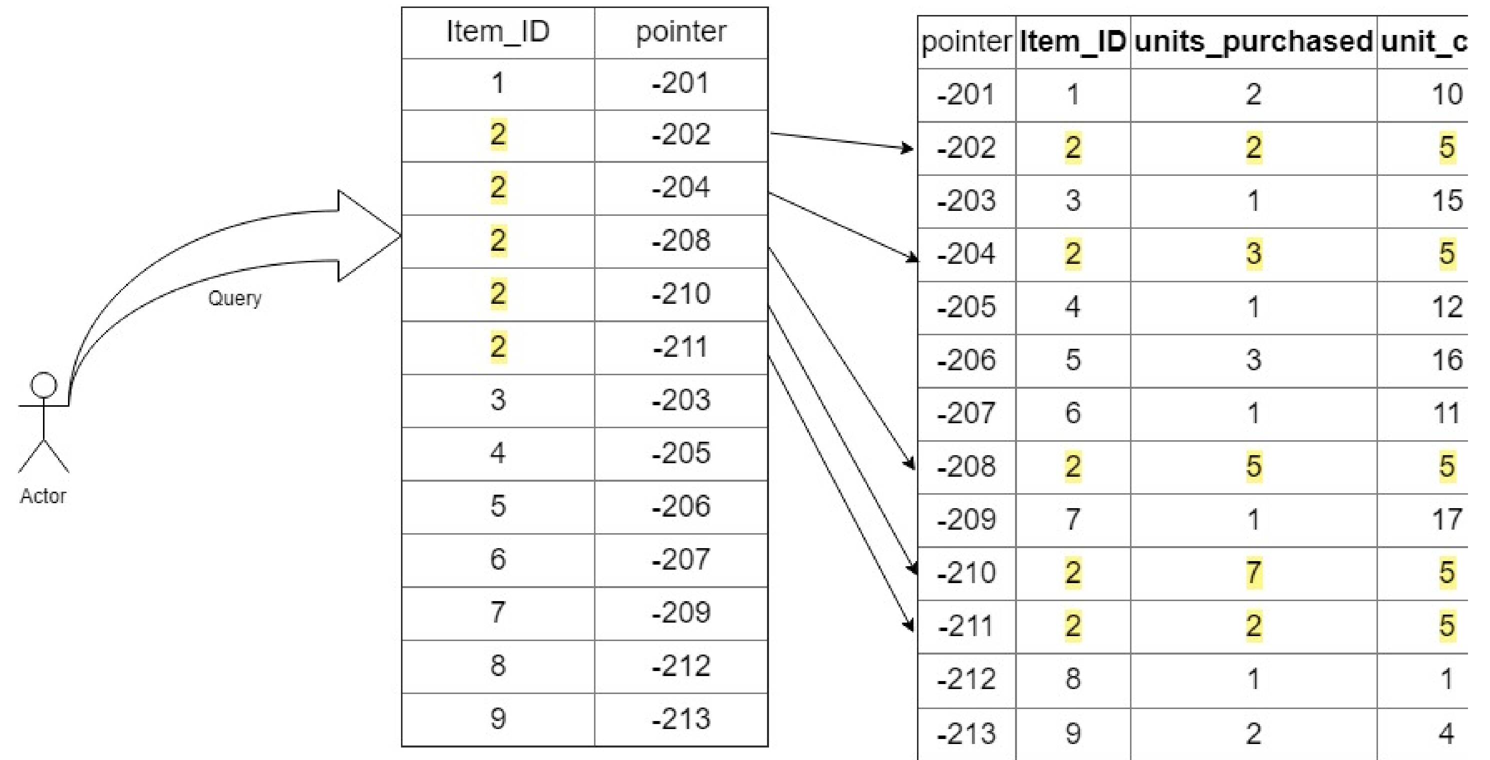
Heuristic overview

Item_ID	pointer
1	-201
2	-202
2	-204
2	-208
2	-210
2	-211
3	-203
4	-205
5	-206
6	-207
7	-209
8	-212
9	-213

Indexed scan

Typically, the following applies:

- Indexing adds a **sorted data structure** for optimizing query efficiency
- Query searches for specific rows in the index structure, then the **pointer** finds the required information
- Indexing **reduces** the number of rows to search: in this case from 13 to 4!

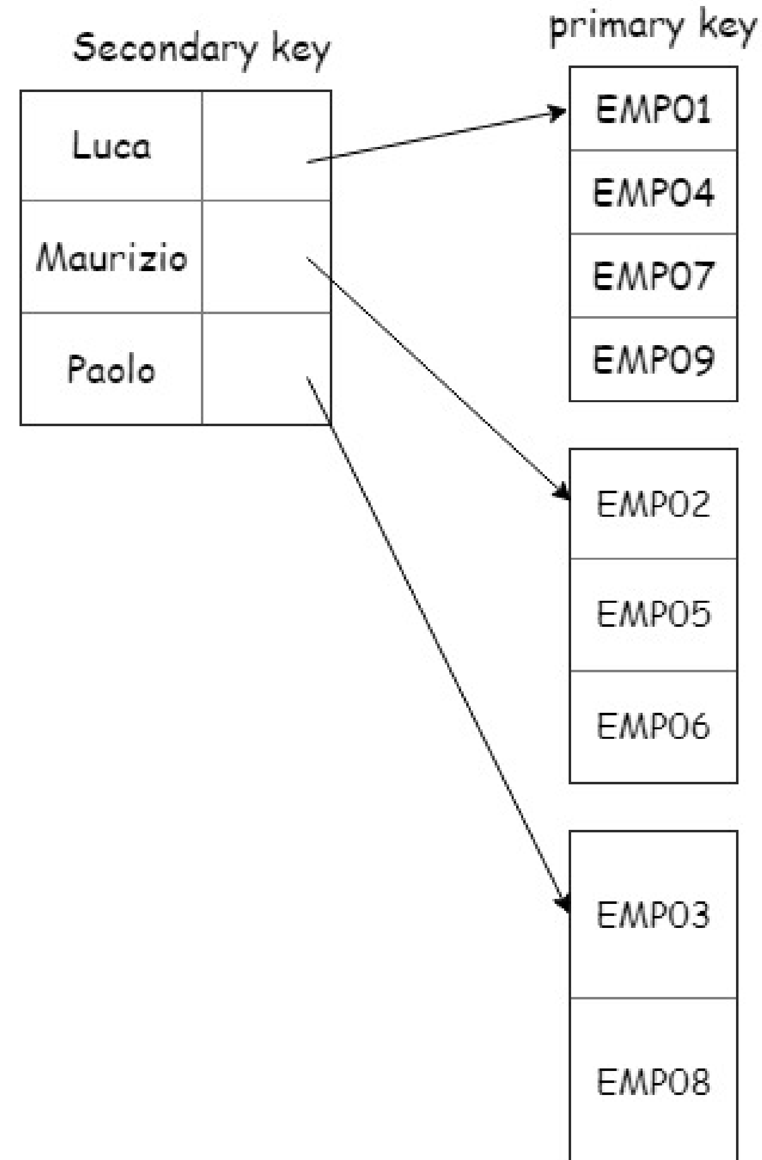


Item_ID	pointer
1	-201
2	-202
2	-204
2	-208
2	-210
2	-211
3	-203
4	-205
5	-206
6	-207
7	-209
8	-212
9	-213

pointer	Item_ID	units_purchased	unit_c
-201	1	2	10
-202	2	2	5
-203	3	1	15
-204	2	3	5
-205	4	1	12
-206	5	3	16
-207	6	1	11
-208	2	5	5
-209	7	1	17
-210	2	7	5
-211	2	2	5
-212	8	1	1
-213	9	2	4

Two-level indexing

- With too many records, the index size grow **exponentially**, that is too big to fit in the **fast memory**
  - Obviously, we need a **second level indexing** probably on **non-unique fields**
    - Linear index is **disk-resident**
    - Second-level index is **memory-resident**

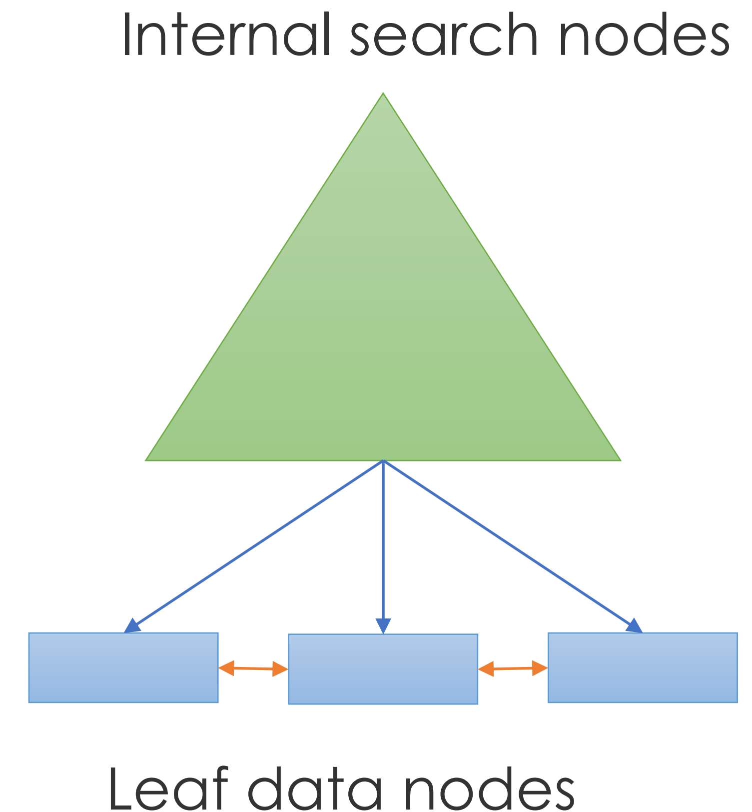


Why not linear indexing

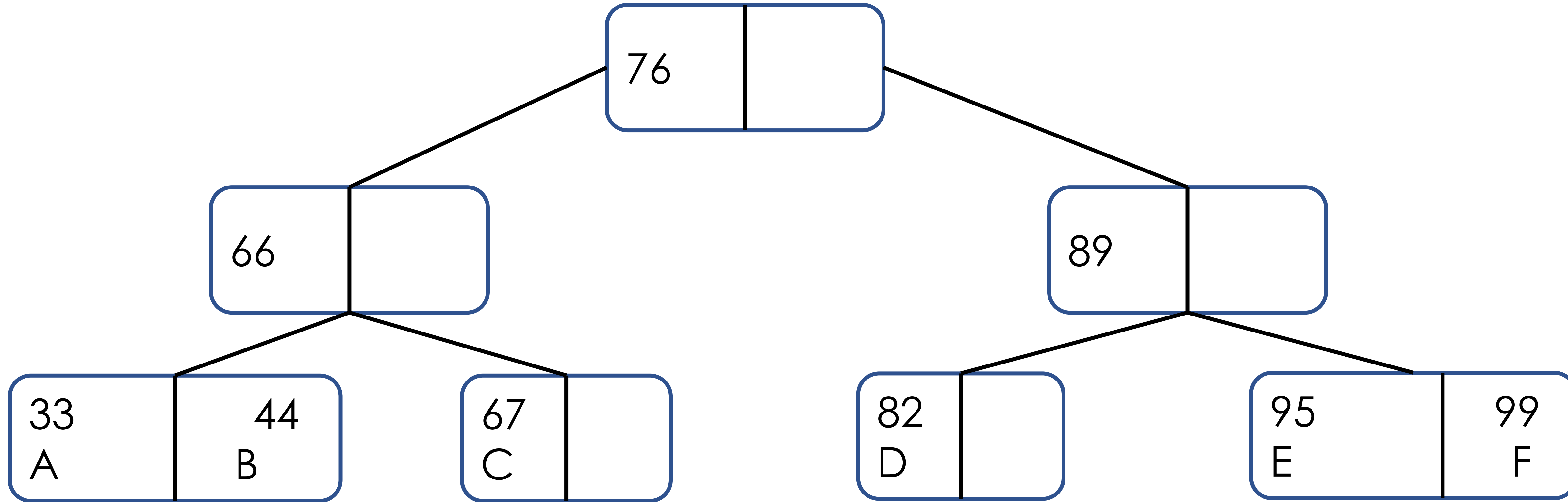
- Linear indexing is only efficient when database is static
  - Insertion and deletion is rare
- Applications on databases share the following characteristics:
  1. Big number of records **updated frequently**
  2. **Search** queries require one or several keys
  3. Key **range queries** or min/max queries are used
- Better data structures must be used: **Trees**!

B+ tree

- **B+ tree** stores records only at the **leaf nodes**
- **Internal nodes** store **key** values, they are utilized only as **placeholders** to guide the search.
  - This means that internal nodes differ significantly from **leaf** nodes (in structure)
  - Internal nodes store **keys** to guide the search, associating each key with a **pointer** to a child **B+ tree** node
  - Actual records reside solely in leaf nodes,
    - But sometimes leaf nodes store keys and pointers to real records in an independent disk file, in case the B+ tree is being solely utilized as an index
  - The leaf nodes of a B+ tree are typically linked together in a **doubly linked list** structure (in-order)
- Advantages
  - efficient traversal & search performance, memory efficiency



Example B+ tree



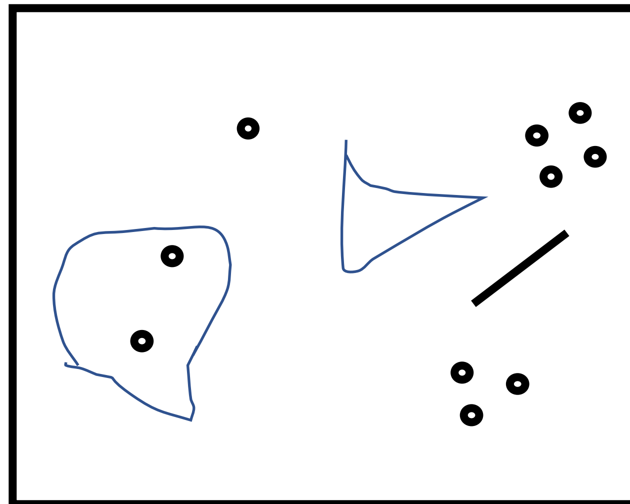
B+ trees are exceptionally good for range queries

B+ Trees

- But how do those fit into our discussion about geospatial data!
- In multidimensional space, there is **no unique ordering**! Not possible to use B+ trees ☹
- Search trees such as **B-trees**, are designed for searching on a one-dimensional key
  - Some databases require support for multiple keys

Why multidimensional indexing

- Having a set of geometrical **objects** (points, lines, polygons)
- The problem is to find a proper **organization** on disk, such that we enable **pruning** the **search space** while answering a spatial query (**point**, **range**, **kNN**)

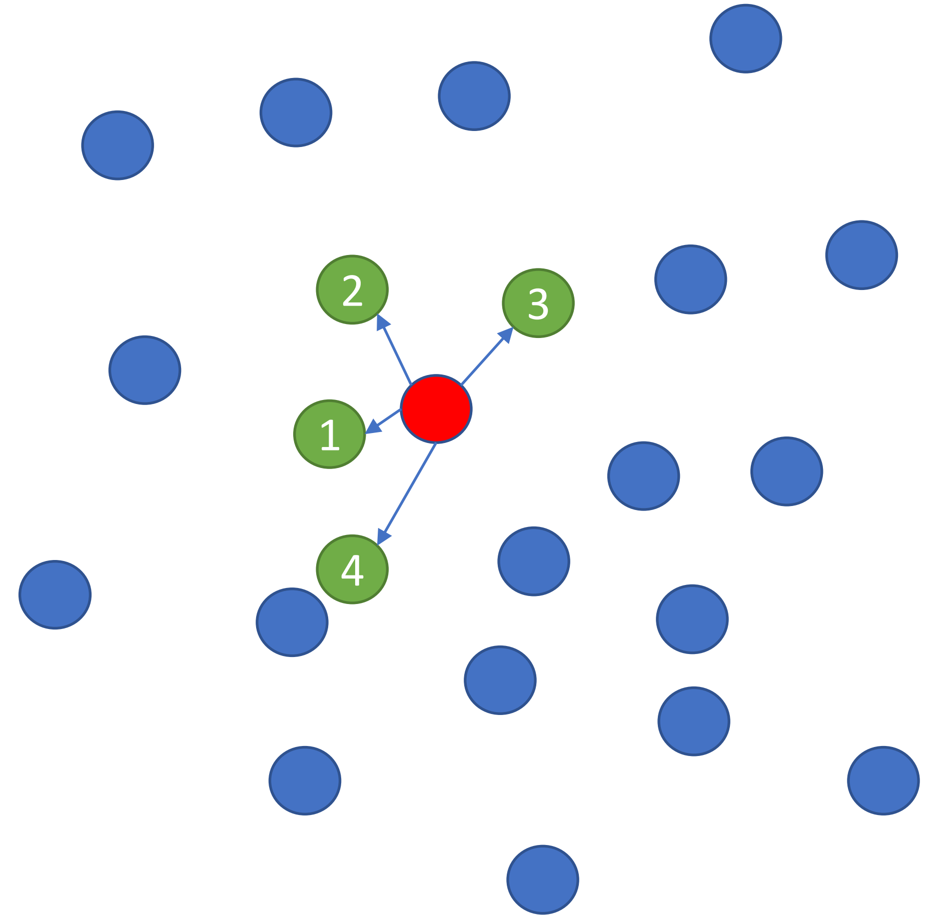


K nearest neighbors

- Given millions of mobility points, such as taxi pickups, how do we find the closest pickup trips to a query point
- An brute-force solution

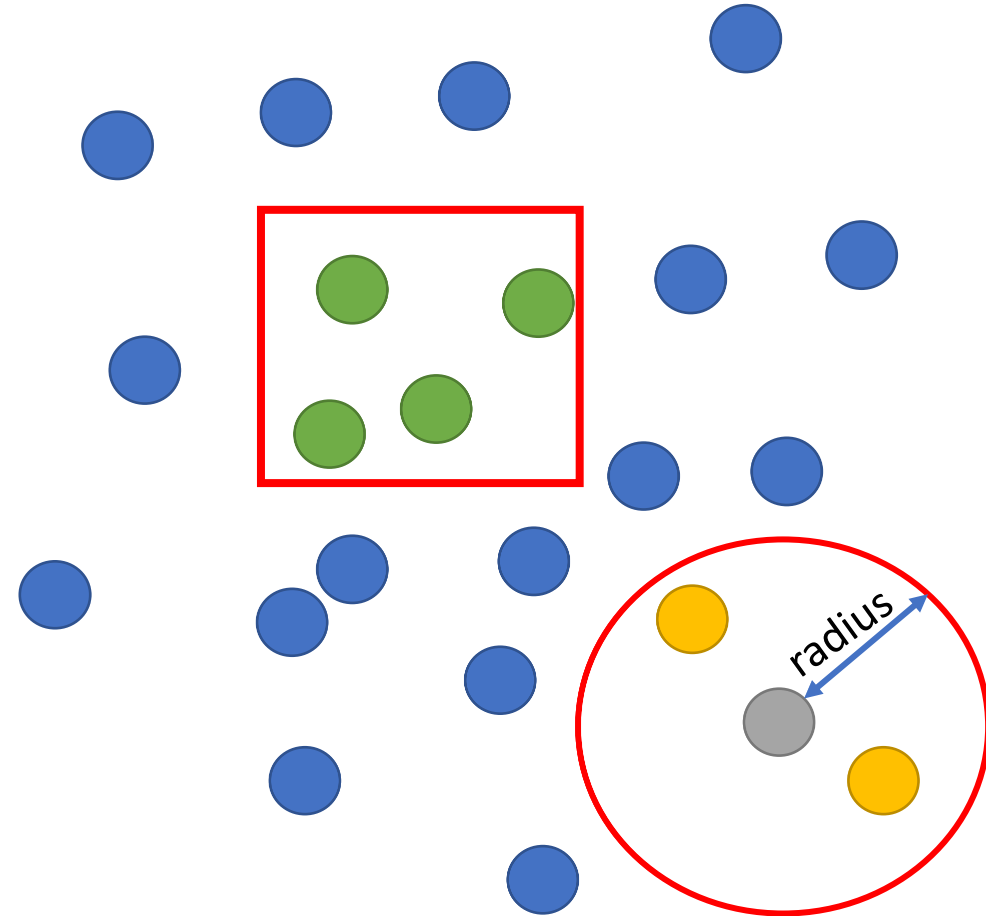
(1) Calculate the distances between every point and the query point
(2) Sort points by their distance in reference to the query point (in ascending order)
(3) Return the first K points that are the nearest

This is an inefficient solution for millions of points



Range and radius queries (Window query)

- Find all points confined within a rectangle (range query) or a circle (radius or proximity query)
- The brute-force approach is to check all points.
  - Inefficient if the datasets are very big and receives hundreds of queries every second



What do we need

- For efficient **NN** and **range** queries, at scale, **spatial index** worth the effort
    - But what is the **read/write ratio** for your spatial data.
    - Remember that indexes are expensive!
- An enduring principle shared by all spatial structures for efficient spatial searches is what is known as '**branch and bound**'
    - Organizing spatial data in **tree-like** structures which allows **pruning** the search space upon receiving a **spatial query**
    - By **discarding** the tree branches that do not meet the **spatial predicate** (search criteria) → skipping data

Multidimensional search

- Database of city records
- Vehicle ID & long/latitude
  - **B-tree is** efficient for searches on **Vehicle ID** or **one of the coordinates**, **Long** OR **lat**.
    - However, not common for **two-dimensional** space
  - Another possible solution
    - **Combining the coordinates**, producing a **single key**: **dimension reduction**
      - Not good for **geospatial range searches**

Types of spatial data structures

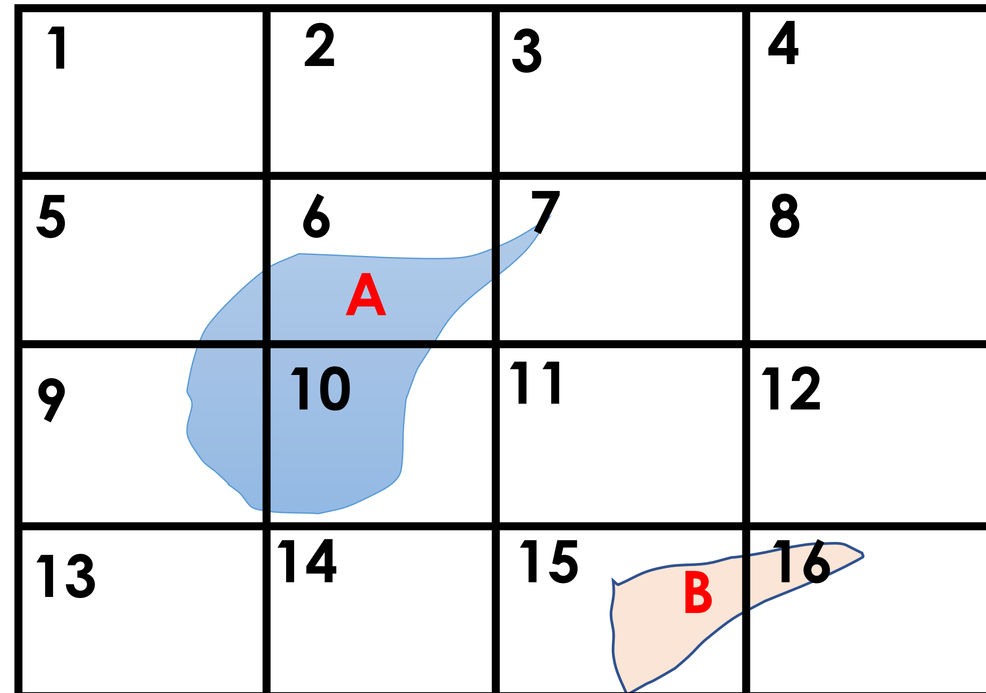
- Two types of spatial data structures
  - **Data-driven**
    - Based upon a partitioning of the data items themselves
      - **R-trees** and ***KD*-trees**
  - **Space-driven**
    - Organized by a partitioning of the embedding space, akin to order-preserving hash functions
      - **quad trees** and **grid files**

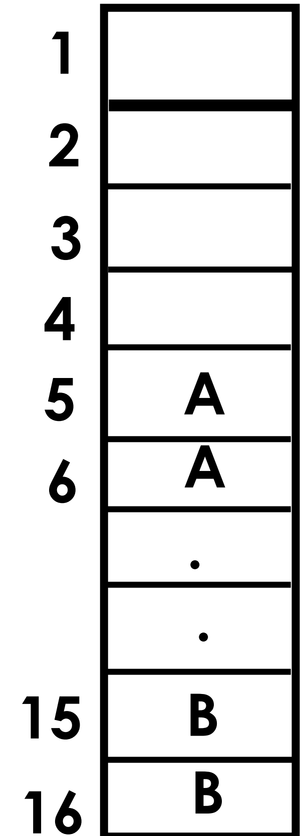
Space-driven spatial data structures

- Dividing the **embedding 2-D** space into **grid cells** (**equal-sized** OR based on **data distribution**)
  - Mapping spatial object's **MBRs** to **cells** based on spatial relationship (**intersects**, **overlaps**)
  - Can be used in spatial extensions with B^{+}-tree,
    - which is dynamic and efficient in memory space and query time
- Some examples
  - **Fixed grid index**
  - **Quadtree**

Fixed grid index

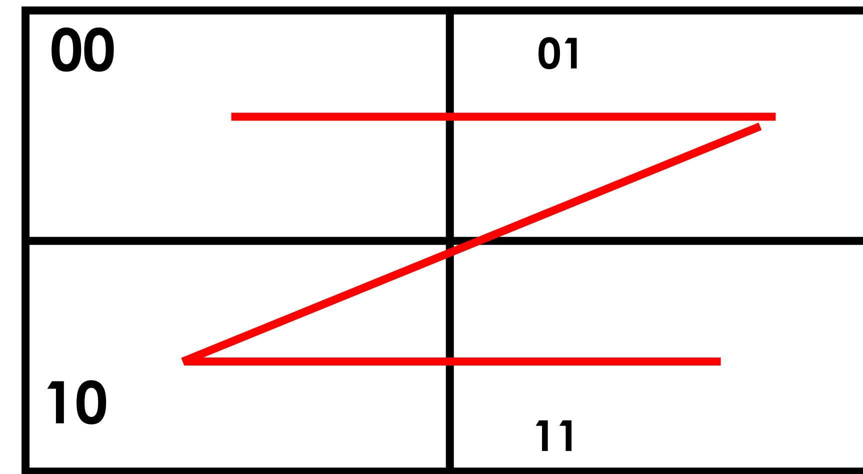
- *Multidimensional* **array** of equal-sized **cells**
  - Each one is attached to a list of spatial objects
    - **intersecting** or **overlapping** with the cell

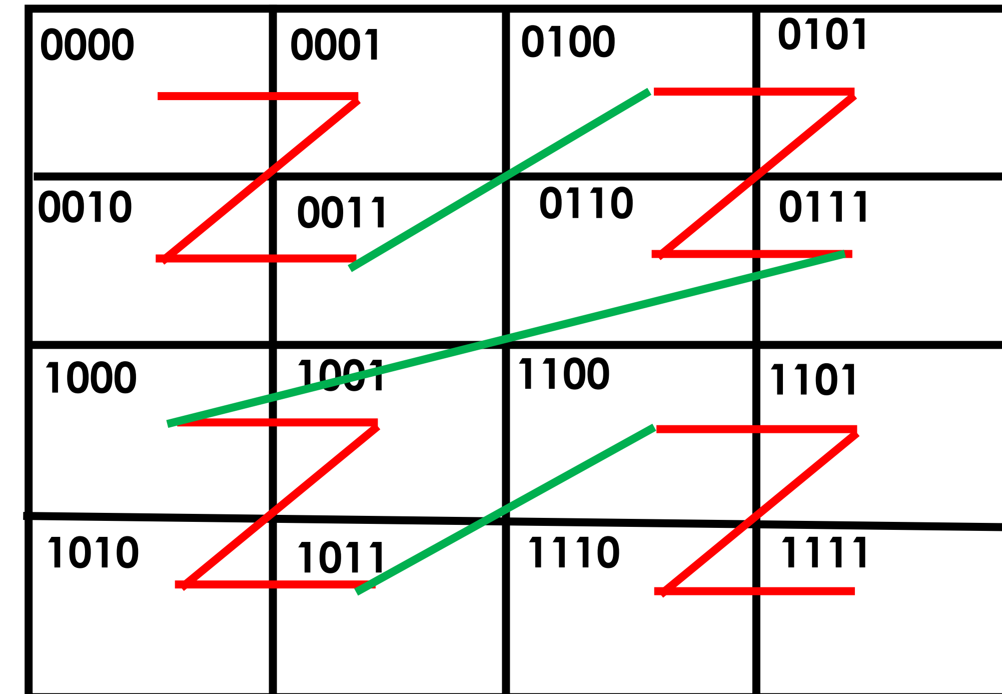




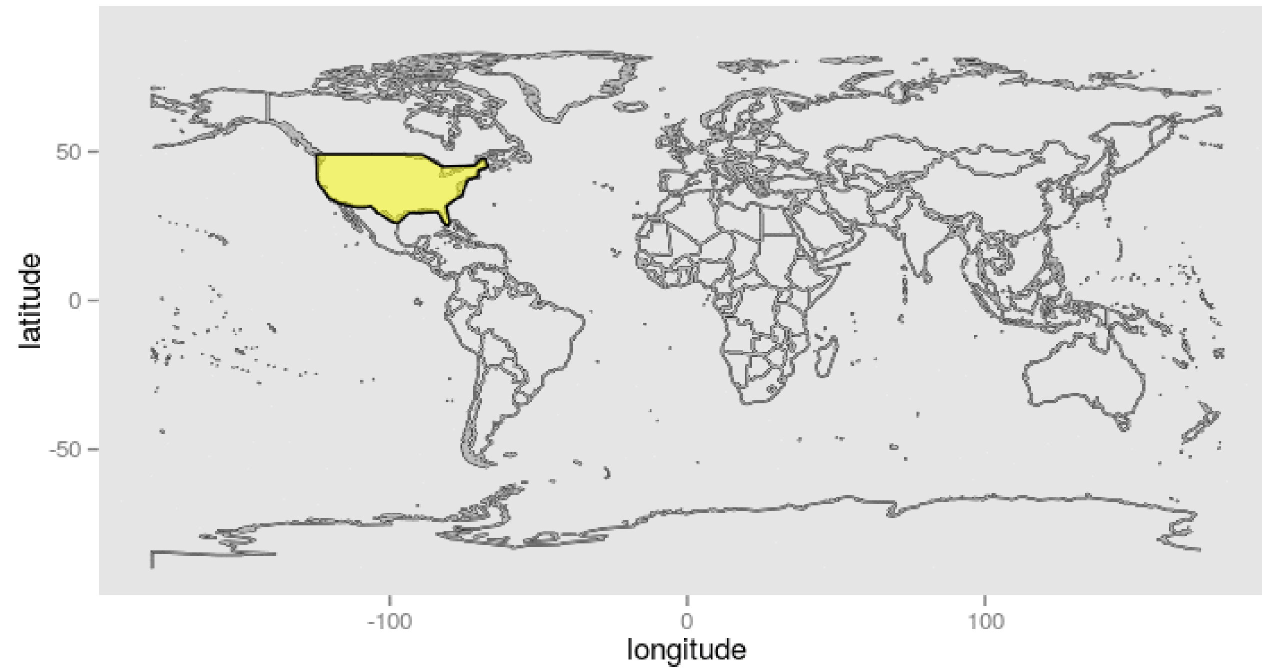
space filing curves: z-order

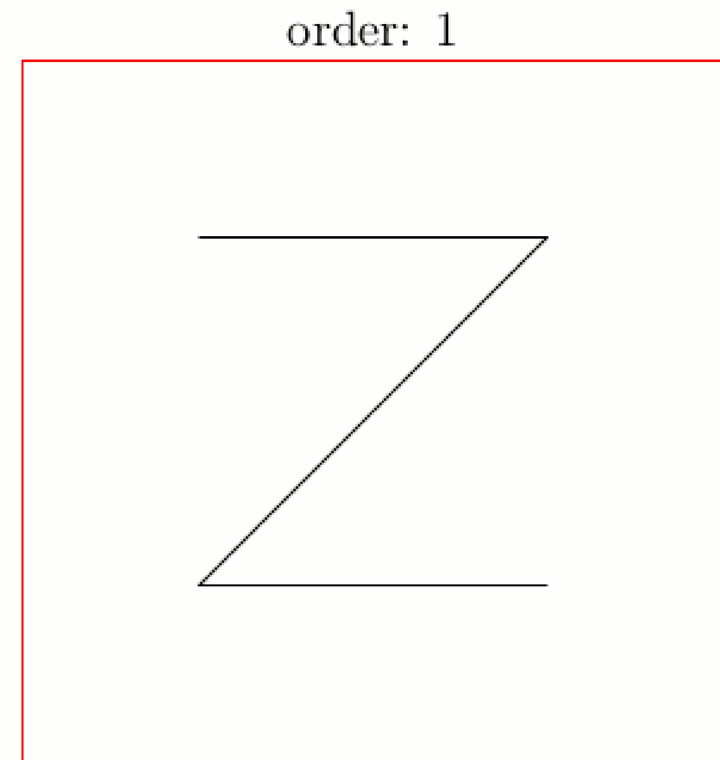
- These grid hierarchy cells are numbered in a linear fashion called ***space-filling*** *curves*.
  - useful because it partially **preserves proximity (spatial co-locality)** → two cells geographically nearby in 2D plane (flattened Earth) are highly likely to be close in the sequential order
    - Various spatial filling curves → we focus on z-order curve
- Z-order labels each cell similar to a complete quadtree and numbers each quadrant in binary **bit string** format 00, 01, 10, 11
  - An associated bit string for each at each level, corresponding to the level cell belongs to (01 in level 1, and 0101 in level 2) → bit interleaving
    - 1110 is obtained by selecting 11 at the top-level and 10 within the top-level quadrant
  - Lexicographical order of the bit strings specifies the order that is imposed on all cells of a subdivision





- *mapping* ***multidimensional*** *data to* ***single-dimension*** *with locality preserved!*

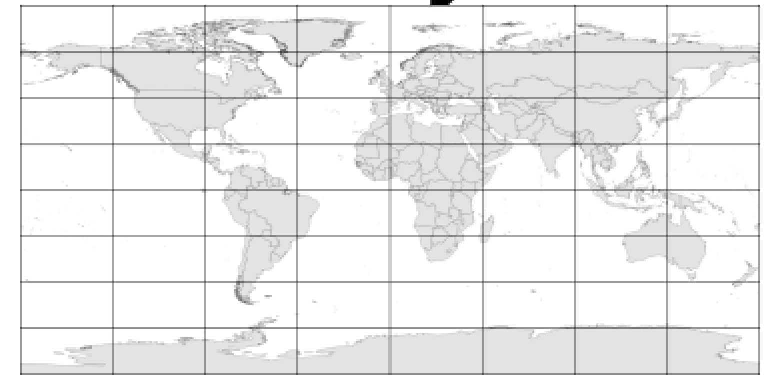


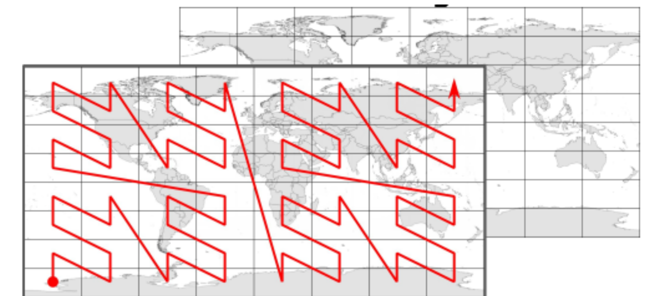


space filing curves: z-order (cont.)

- Space Filling Curves are used to co-locate related data in the same set of files
  - **map multidimensional** data to **single** dimension while preserving spatial **co-locality**
- NoSQL databases support only single dimensions
  - Typically, a sorted key-value index
  - Spatial data is multidimensional
  - Use **Space Filling Curves**
    - Divide the embedding space into grid cells
    - order grid cells with a space filling curve (Z-Order curves)
    - Label grid cells in relative to the order that the curve
      passes through them
    - Associate a byte representation of the label to the data contained in each grid cell

Image source



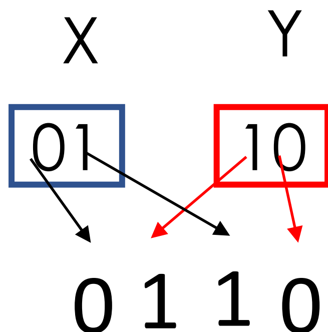


Z2 "GeoHash"

Calculation of Z-order values

- **Bit-interleaving**
  - Quadrant z-value → **alternating** bits from the binary representations of x and y coordinates

11	0101	0111	1101	1111
10	0100	0110	1100	1110
01	0001	0011	1001	1011
00	0000	0010	1000	1010
	00	01	10	11



Single-dimension indexing of spatial data

- One-dimensional orderings
  - Mapping multidimension to one dimension
  - preserve spatial proximity
- Insert Z-elements into a **B-Tree** (single dimension indexing structure) (cf. UB-Tree) as spatial keys in **lexicographical** order (z-order)
- **Range** & **containment** queries (with rectangle r) are then simplified
  - Because of the proximity-preserving of z-ordering (spatial co-locality)
  - Find **z-elements** of r (**covering** z-elements)
  - For each **z-element** (z) in the covering scan the part of the **B-tree leaf sequence** containing z as a **prefix** (filter step)
  - Apply the **actual geometrical operation (costly)** to check for containment (refine step)
    - False positives

- **Partition** the space with a uniform **grid**
- Attaching numbers to cells so that **neighboring** cells have similar numbers

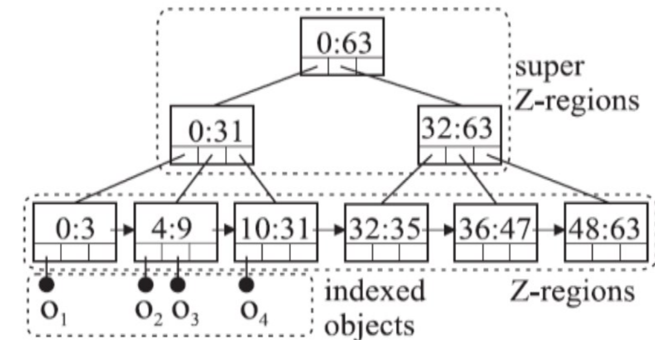


Image source

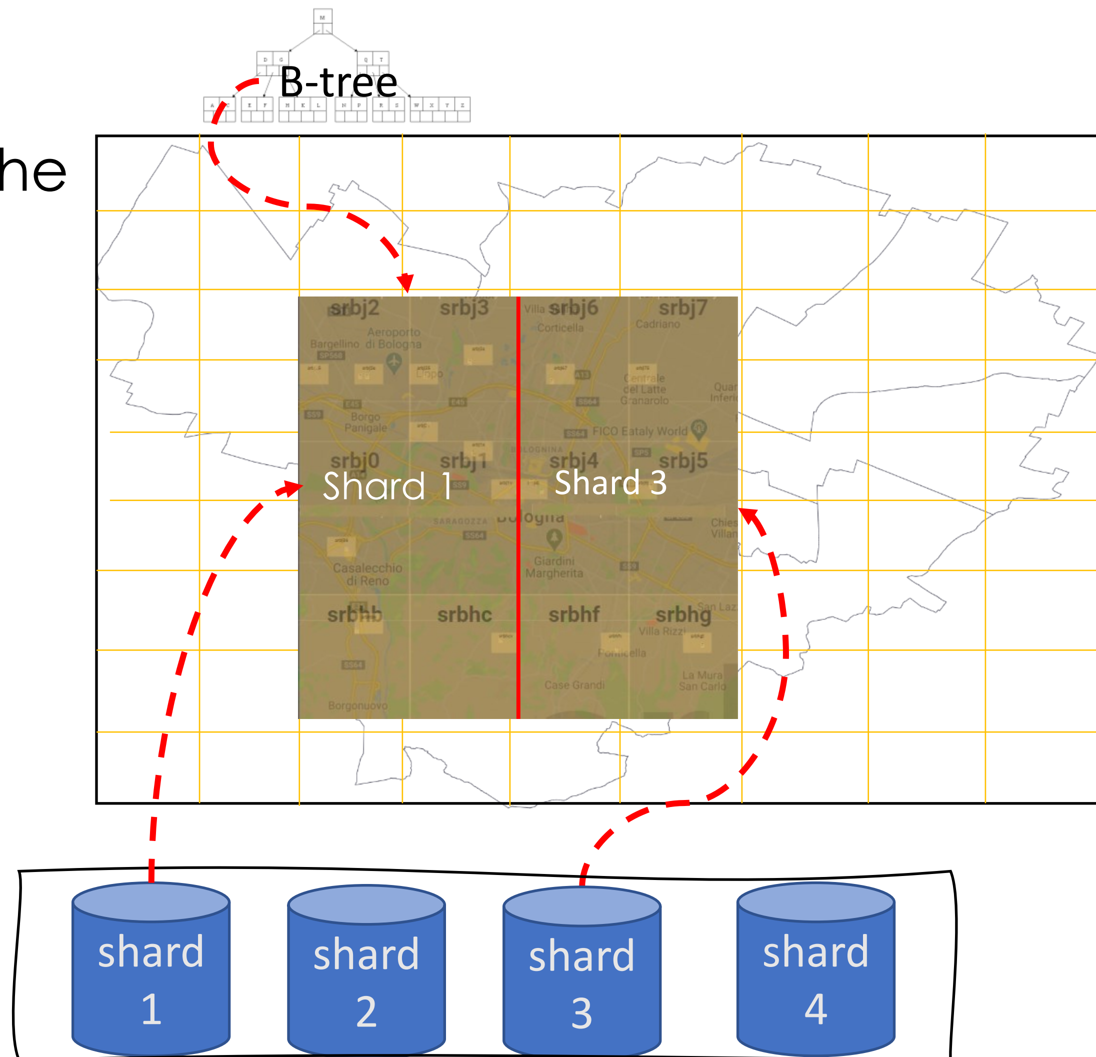
Spatial query optimizer for NoSQL

- MongoDB router forwards requests to few shards, pruning the search space

- **Overlay** the embedding space with a fixed-grid network

- **Generate** a geohash covering and a list of interacting points

- **Impose** B-tree index on the geohash covering & the **interacting** spatial points

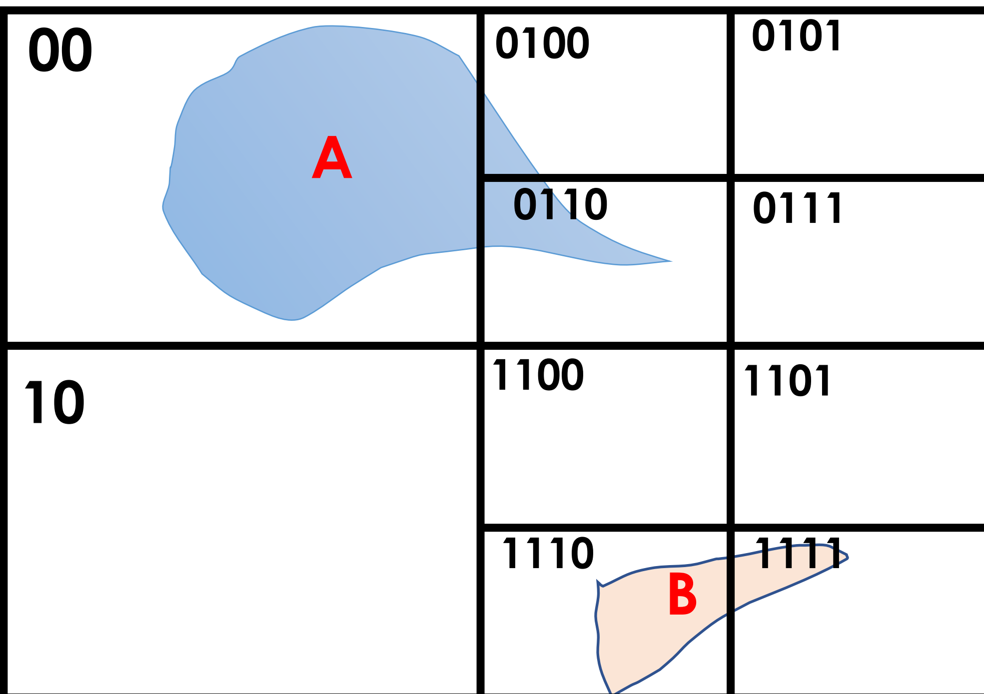


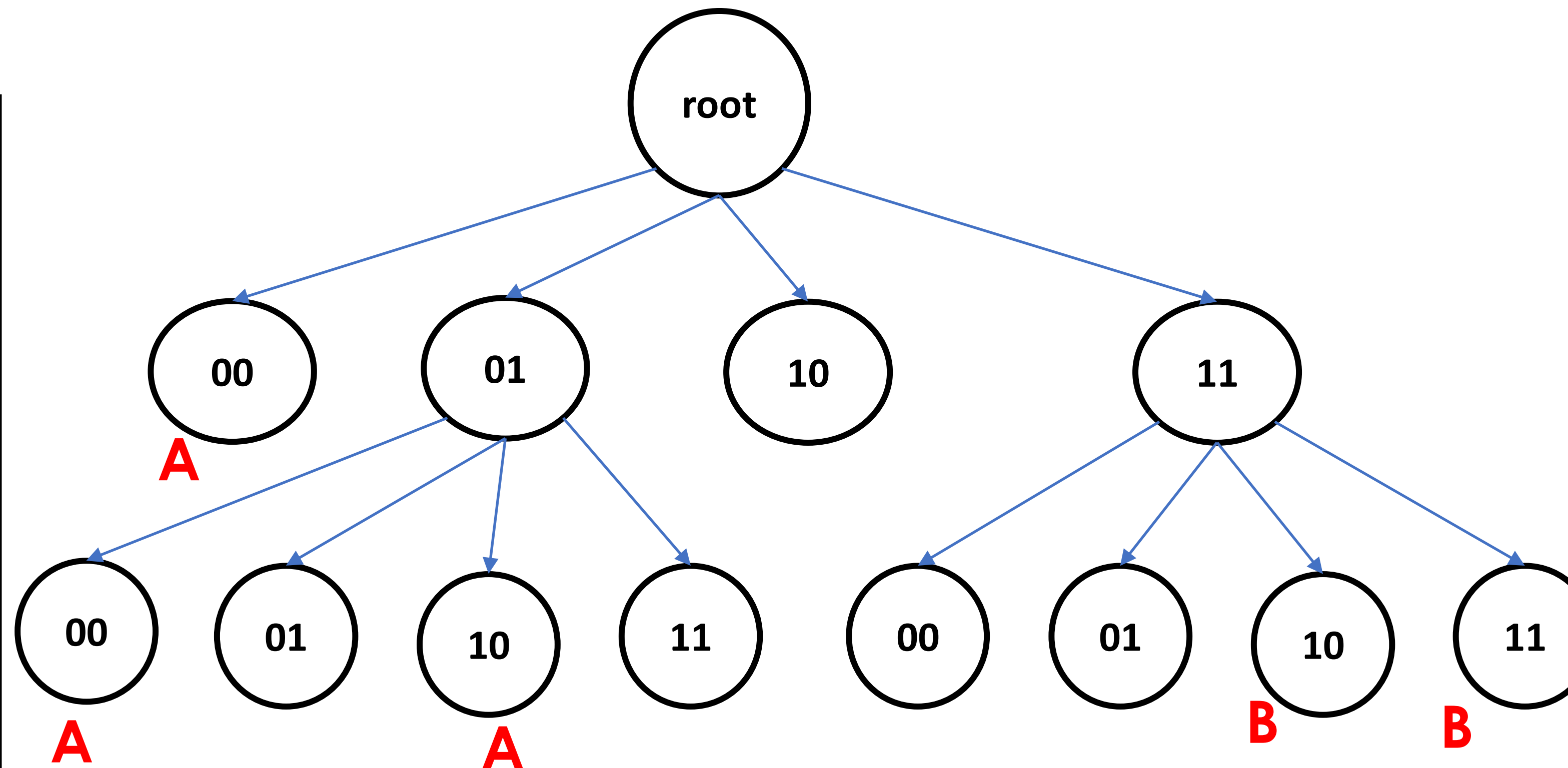
Quadtree

- Very popular **spatial indexing structure**
  - A form of **grid** indexing with varying sizes of grid cells that depend on the data **distribution** (i.e., **density** of the spatial objects)
- Each node in the tree covers a **bounding box** for part of the embedding space being indexed,
  - **root** node covers the entire **embedding** space

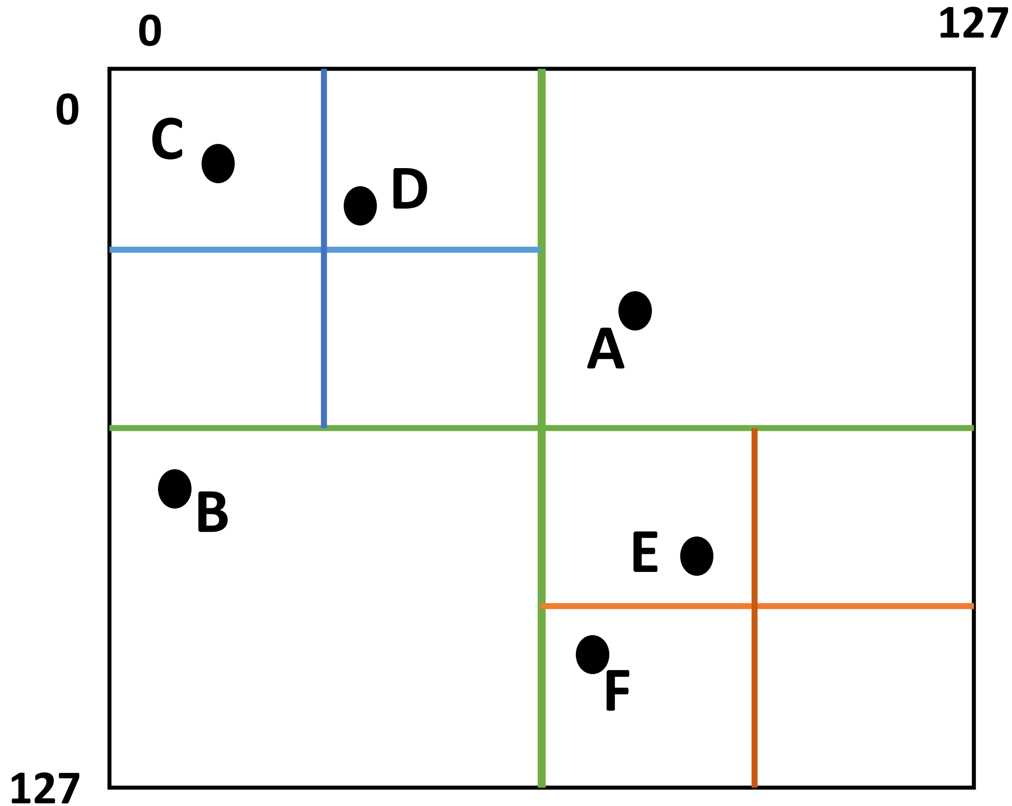
Quadtree

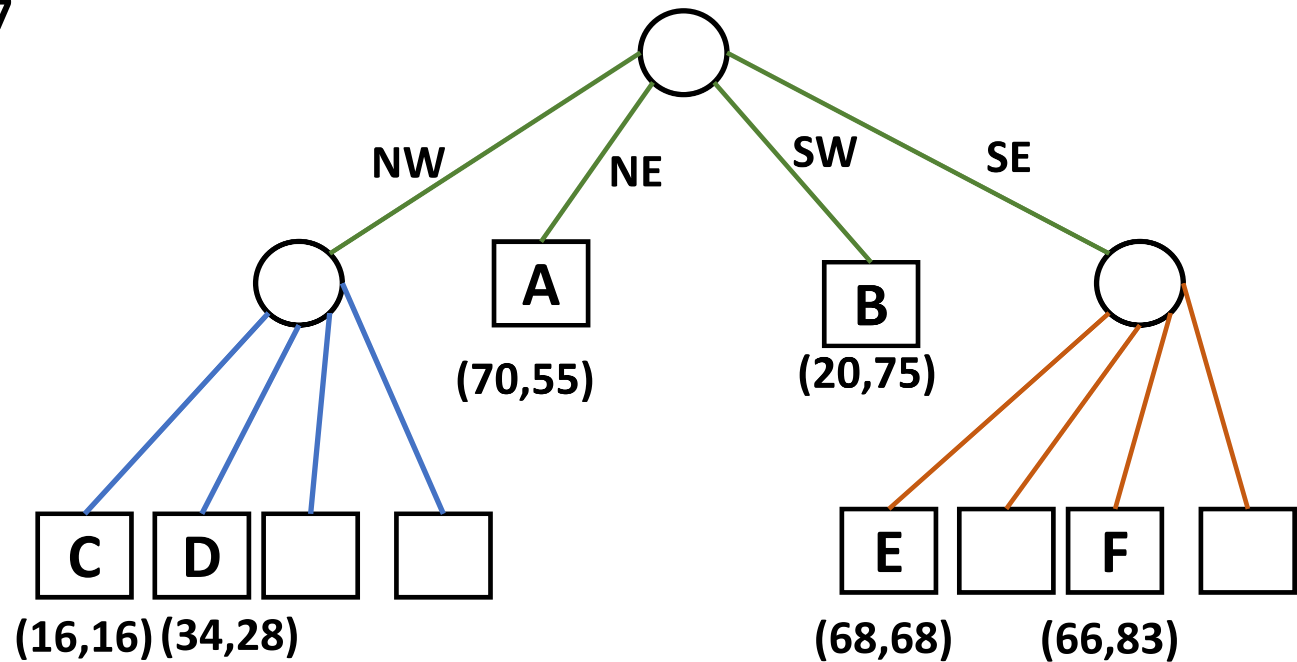
- **Recursive** division of the embedding **space** into **quadrants** (four subdivisions) until each quadrant hosts a prespecified number of points
- Each node
  - A **leaf** node containing **indexed spatial points**, or
  - An **internal** node, having exactly **four children (Quad)**, one child for each **quadrant** obtained by recursively halving the area in both directions





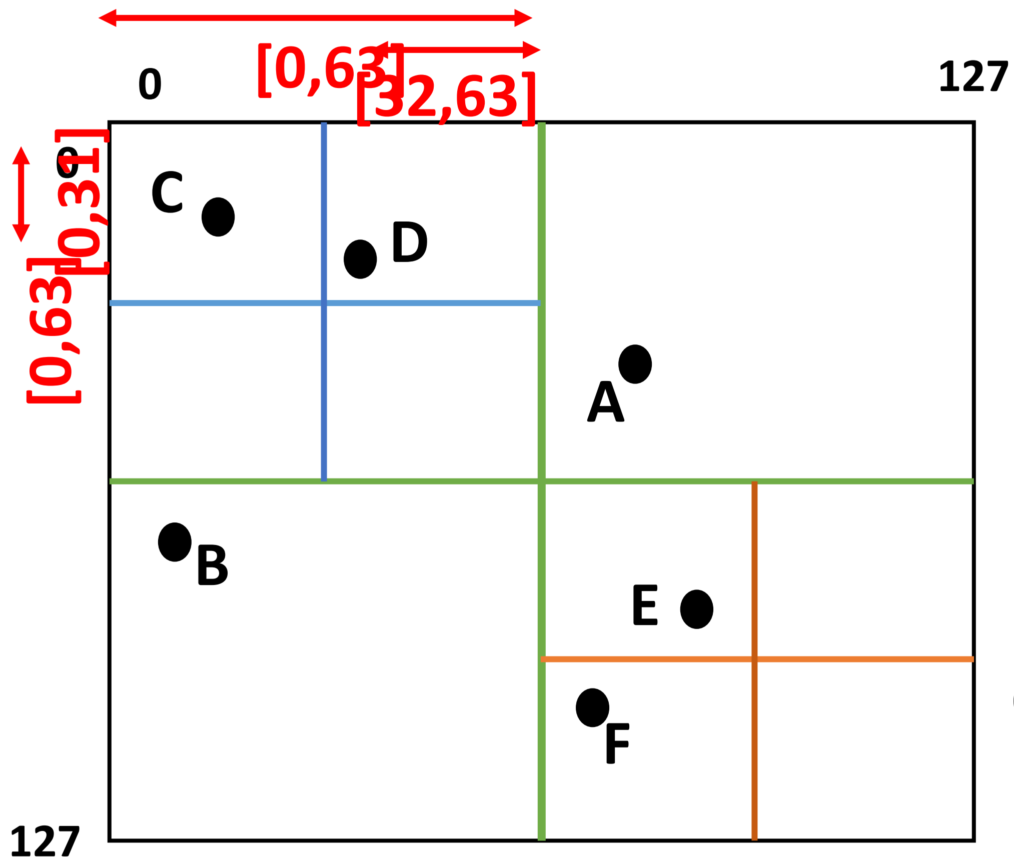
PR quadtree insertion

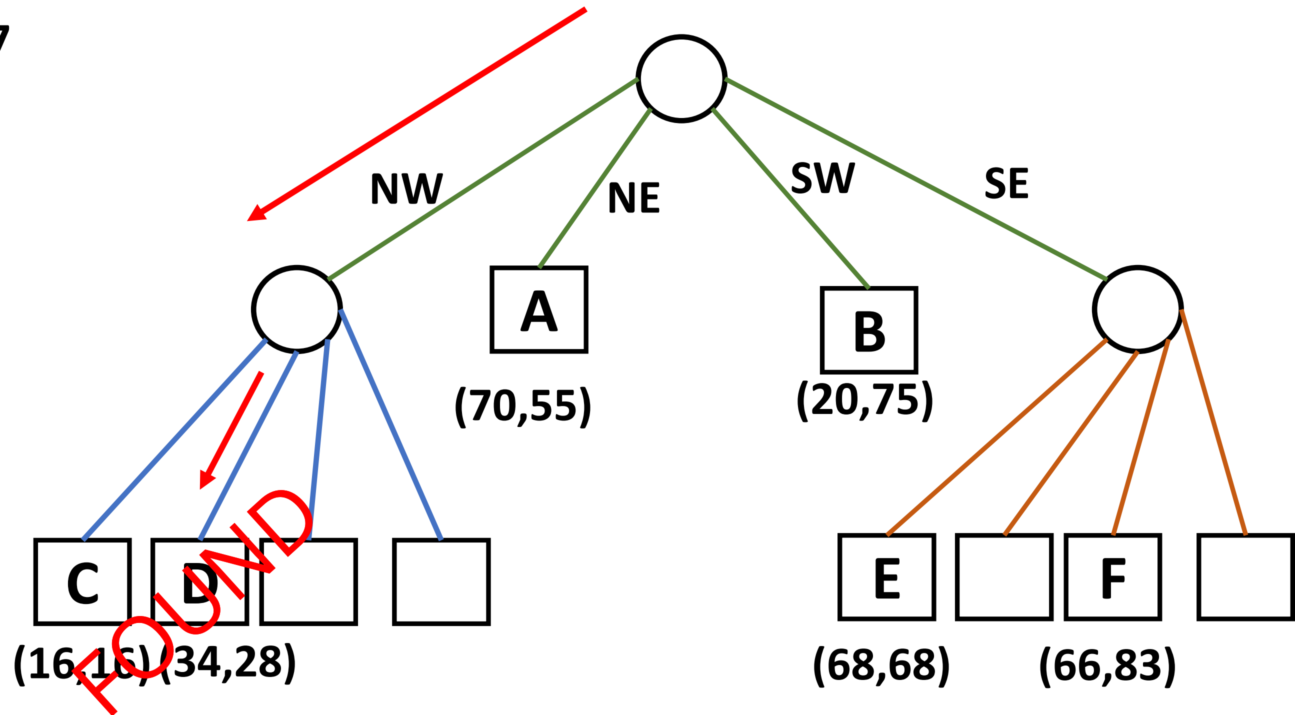




- Recursive decomposition so that only one single point in each leaf node
- approximately half of the leaf nodes will contain no data field

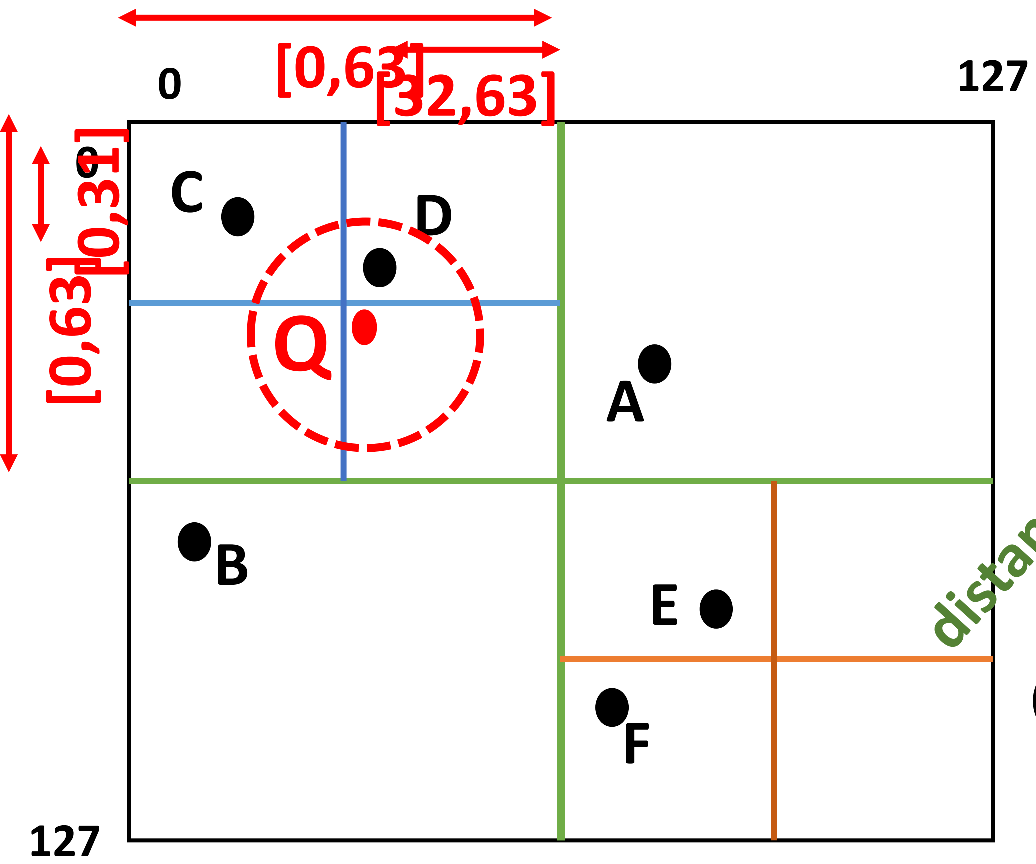
PR quadtree point search

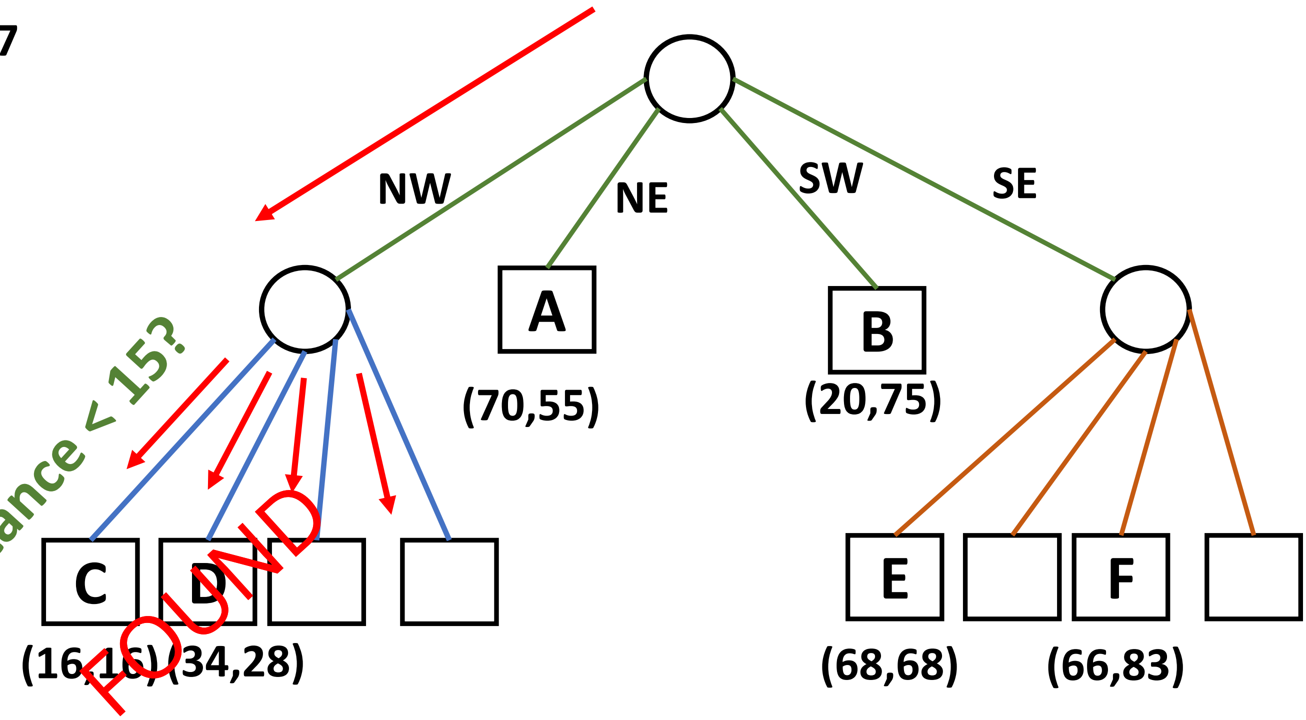




Search for (34,28)

PR quadtree region search

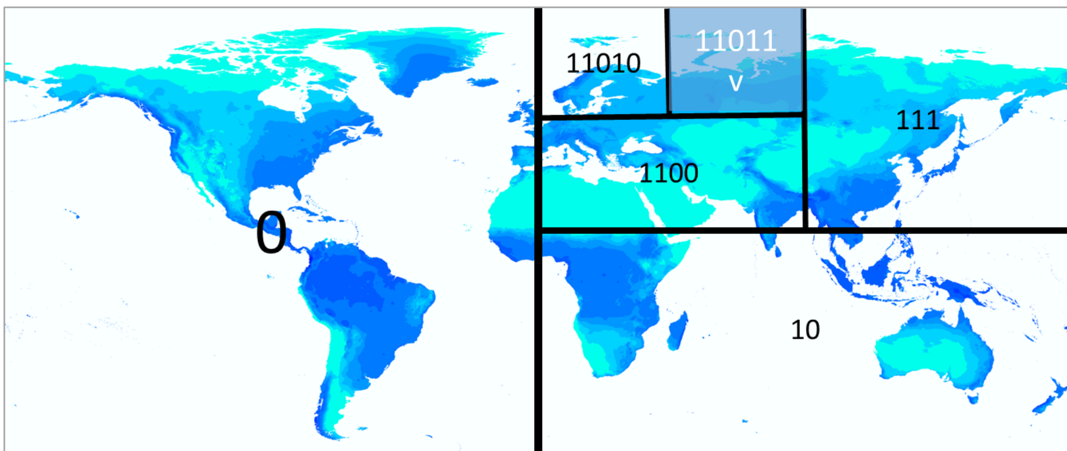


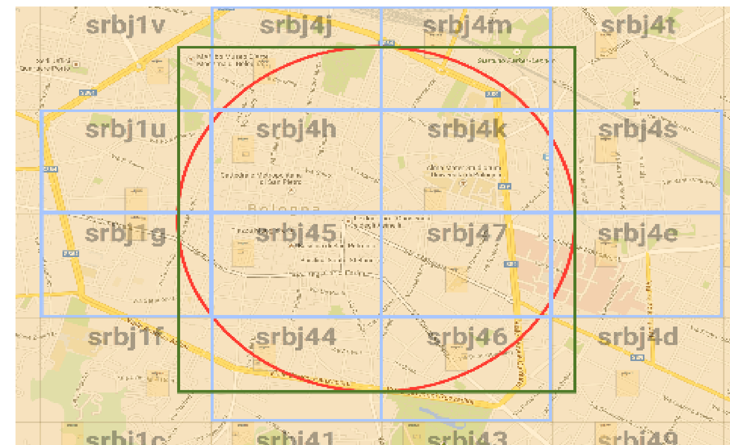


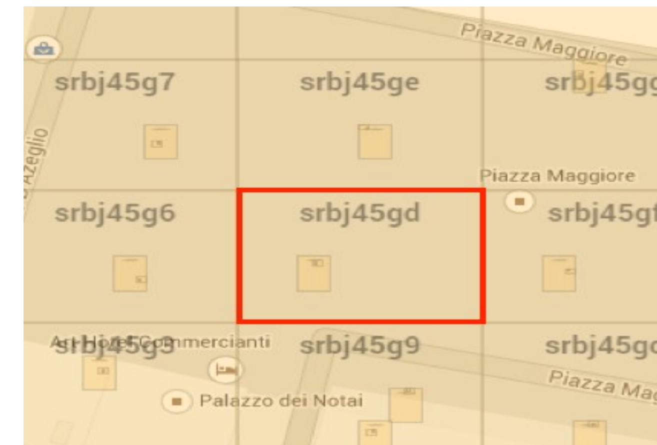
- Search for points that are at most 15 units far from the search point Q (40,40)
- Even C does not fall within the circle, we have to search the NW quadrant, because part of the circle is enclosed within it!

Geohash

- For **geocoding** points as a short **string** and use them in web URLs
  - It is basically a **binary string**, with every character indicating **alternating** divisions of a **longitude/latitude** rectangle
- Split the rectangle into two **equal sized** splits with **Geohash codes** ("0" and "1”).
  - Objects residing on left have Geohash beginning with ‘0’ , while those on right half have a Geohash beginning with "1“
- Assign a **plain text** (**base-32** and **base-36**) **encoding**
  - The length of Geohash ranges from **1** to **12** → longer Geohash has a **granular** precision (covering smaller area)







[Image source]

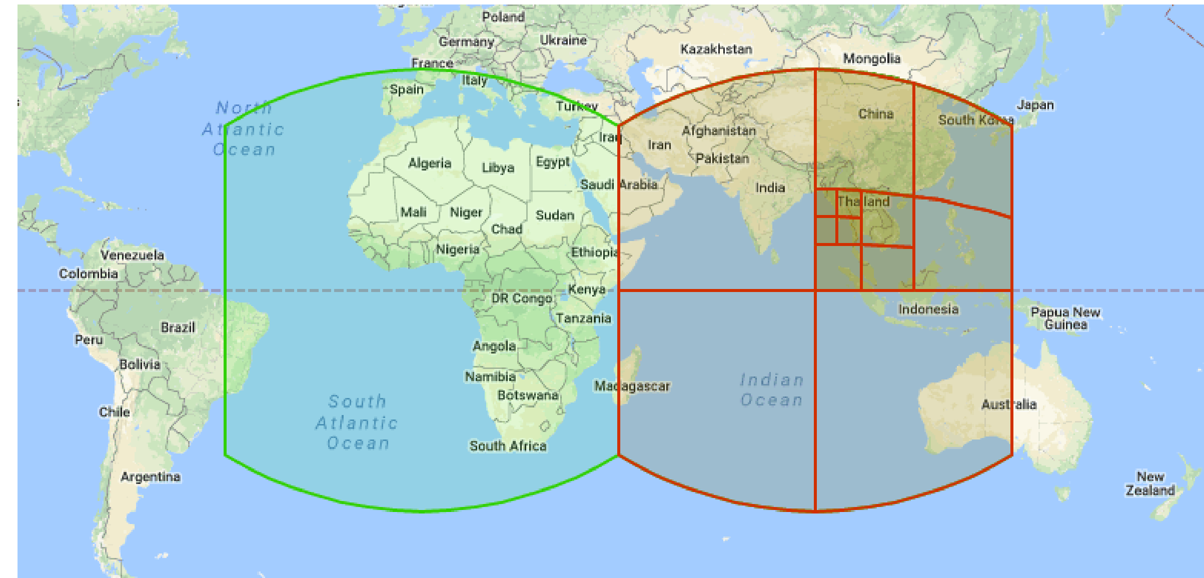
Geohash covering

S2 explained

- framework for **decomposing** the unit **sphere** into a **hierarchy** of ***cells***
  - **Hierarchical** decomposition of **sphere** into **cells**
  - ***approximate regions*** using **cells**
  - cell **edges** appear to be **curved**
    - straight lines on the sphere (similar to the routes that airplanes fly)
- **Levels** (number of times the cell has been subdivided (starting with a face cell))
  - range from 0 to 30
  - top **level** → projecting the six faces of a cube onto the unit sphere,
  - lower **levels** → subdividing each cell into four children recursively

- The smallest cells at level 30 are called *leaf cells*; there are $6 * 4^{30}$ of them in total, each about 1cm across on the Earth's surface.

Image source

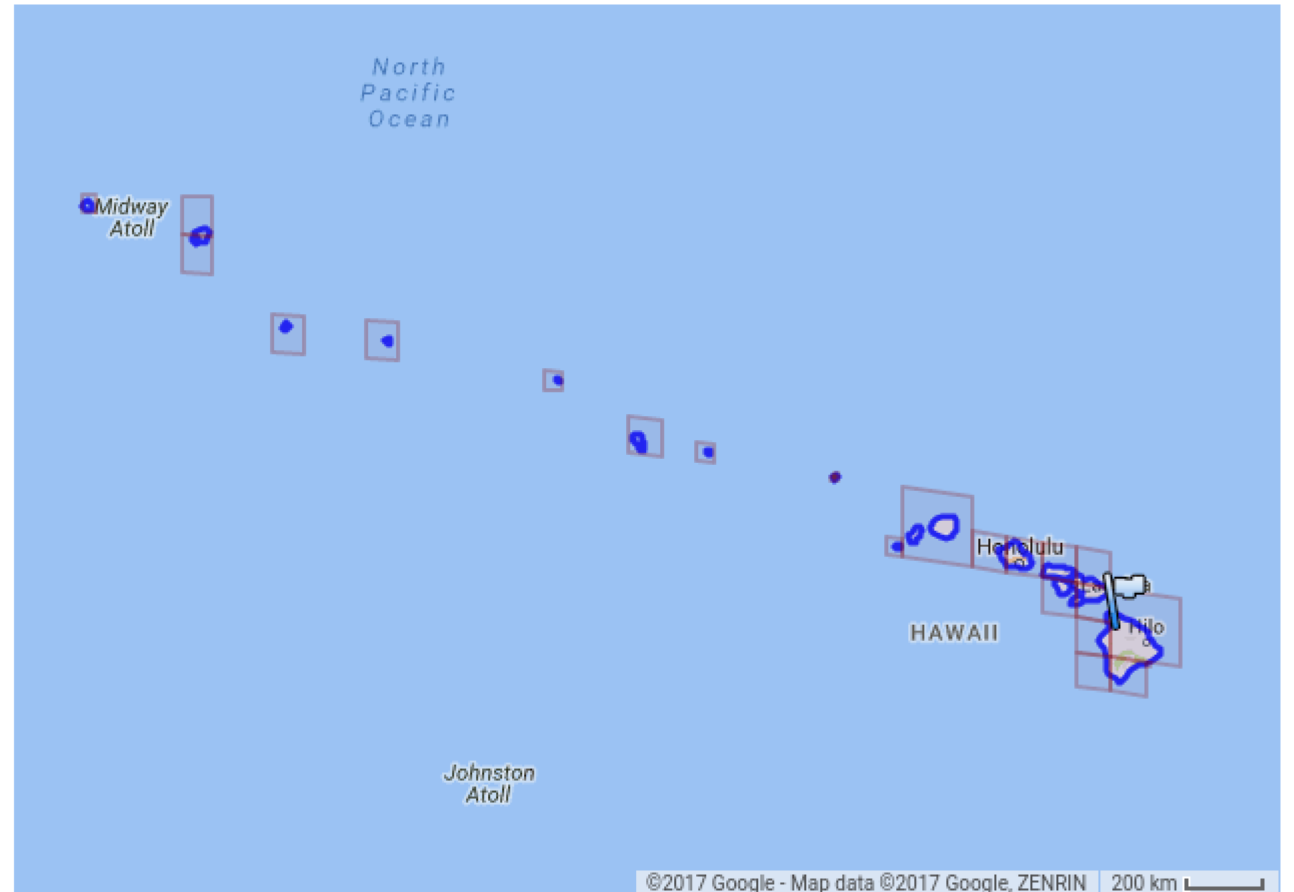


Level	Min Area	Max Area
0	85,011,012 km^2	85,011,012 km^2
1	21,252,753 km^2	21,252,753 km^2
12	3.31 km^2	6.38 km^2
30	0.48 cm^2	0.93 cm^2

S2 explained (cont.)

- useful for spatial **indexing** and for **approximating regions** (polygons) as a collection of cells (S2 coverer)
  - Points (spatial **point** objects) represented as leaf cells
  - Regions (**polygons**) are represented as collections of cells
  - Each cell is identified uniquely by a **64-bit** S2CellId

approximation of Hawaii as a collection of S2 cells



Google's S2

- S2 cells are ordered sequentially along a **space-filling curve**
  - *S2 space-filling curve*
  - six **Hilbert curves** linked together to form a single continuous loop over the entire **sphere**

S2 Coverer for part of Bologna

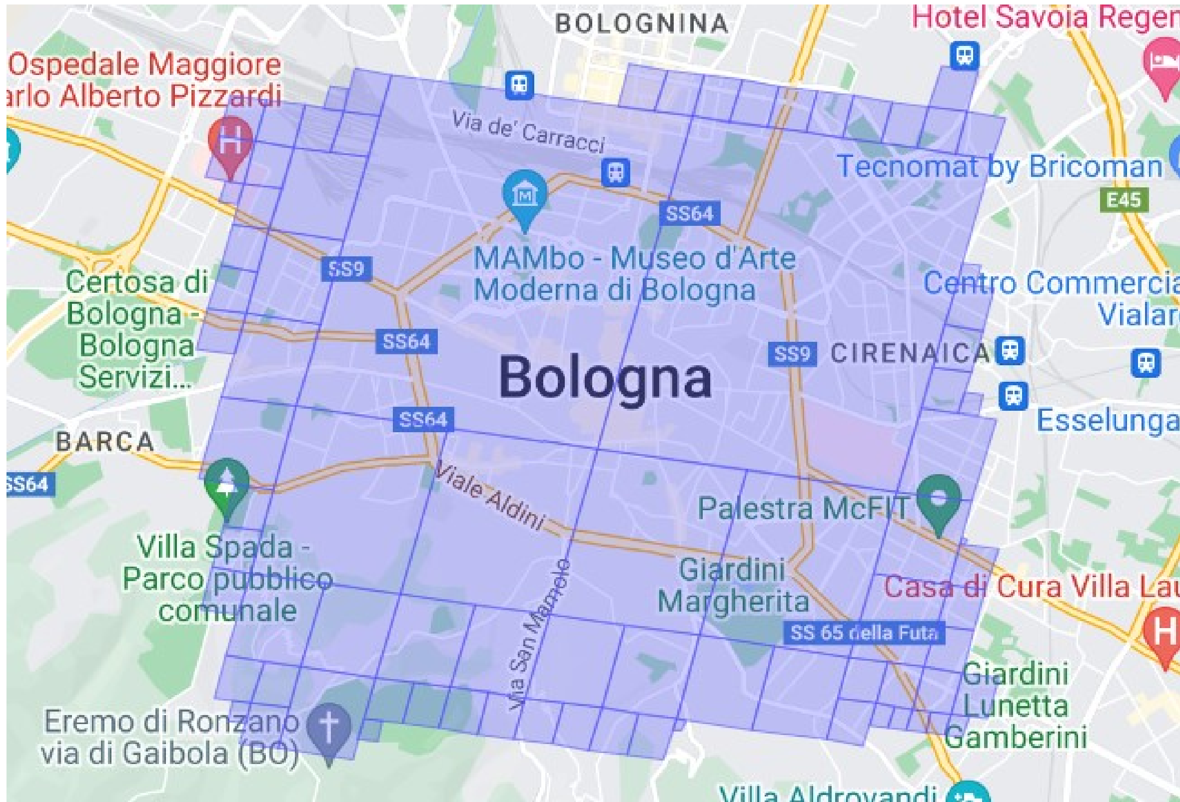
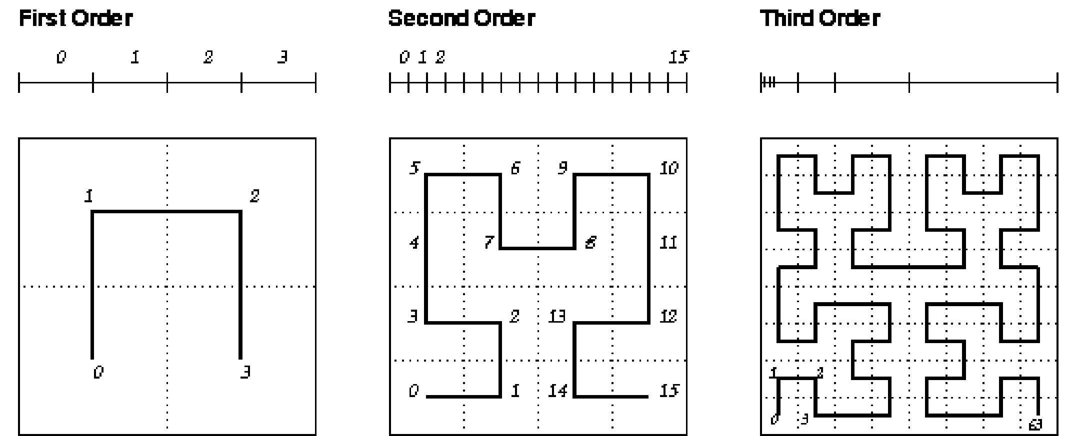


Image generated by this tool



draw a one-dimensional line that fill every part of a two-dimensional space

Image source

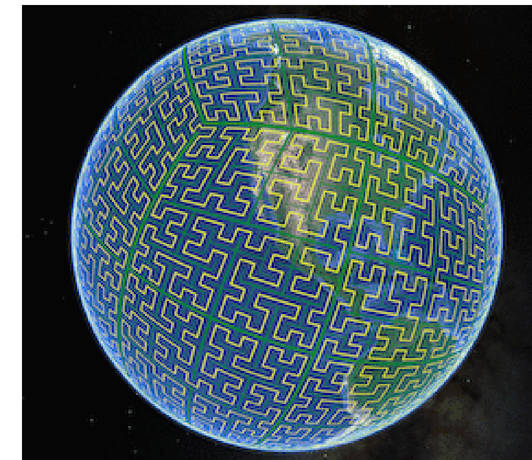
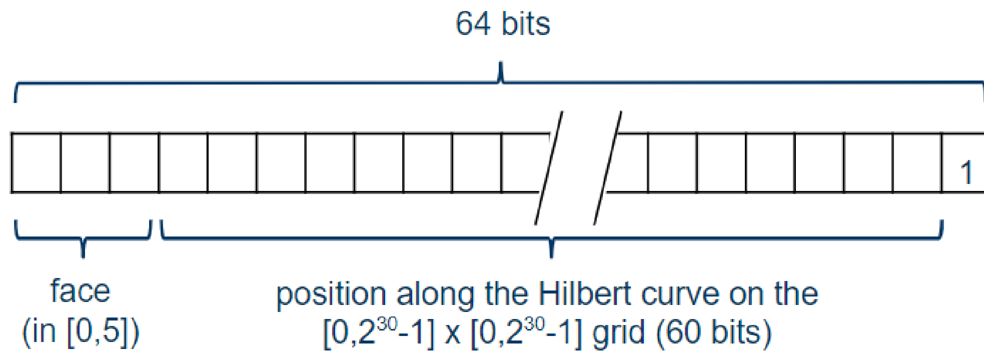
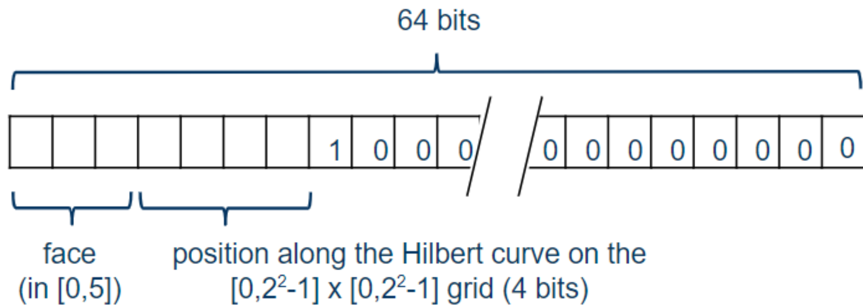


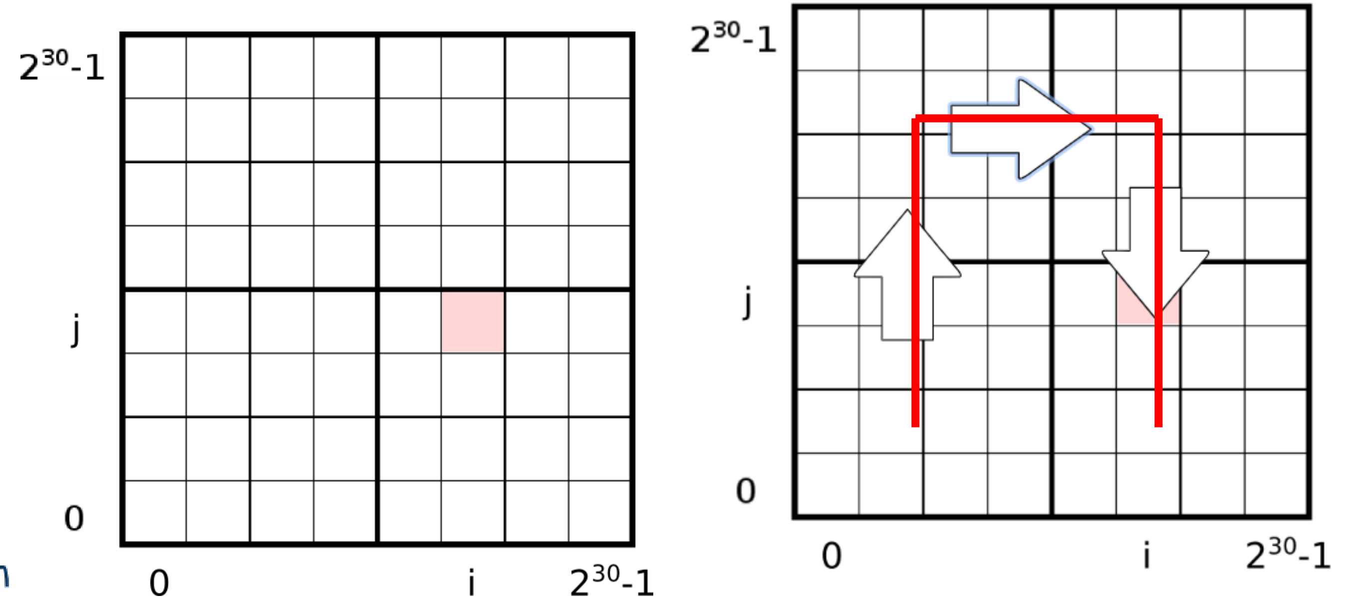
Image source

S2 Cell Hierarchy

- Enumerate cells along a Hilbert space-filling curve
- fast to encode and decode (bit flipping)
- preserves **spatial co-locality**







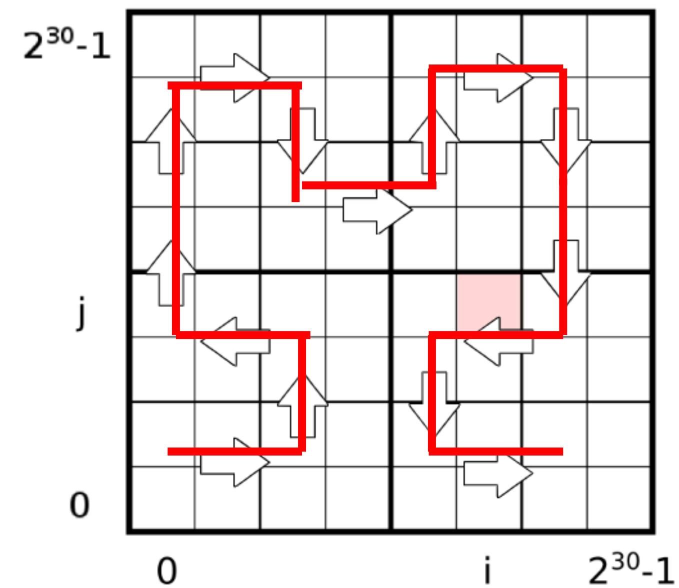


Image source

Google's S2

- Geofence Earth with a planet-size **cube**
- fill each with a **Hilbert** curve (yellow)
- project the **Hilbert** curve onto the Earth's surface (red)
  - Efficient approach to represent locations as **single** numbers

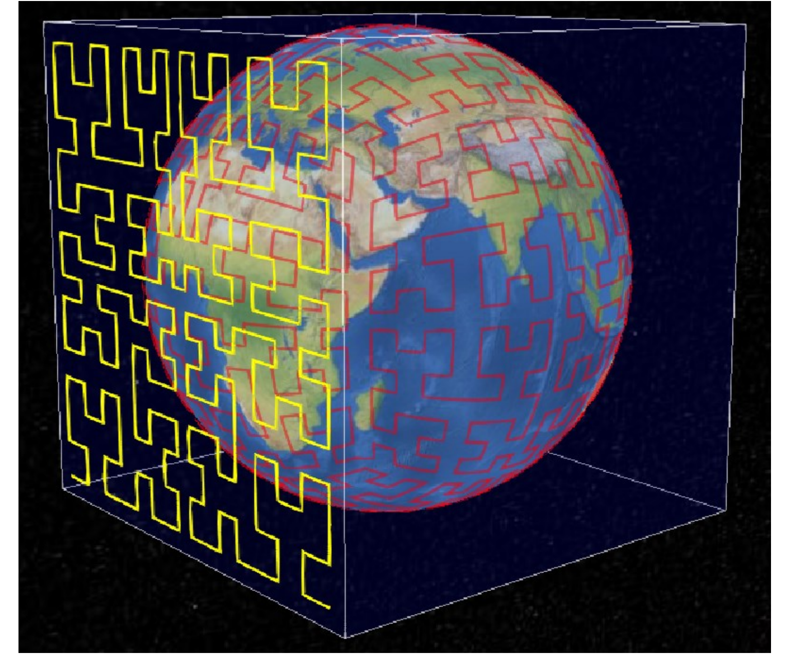
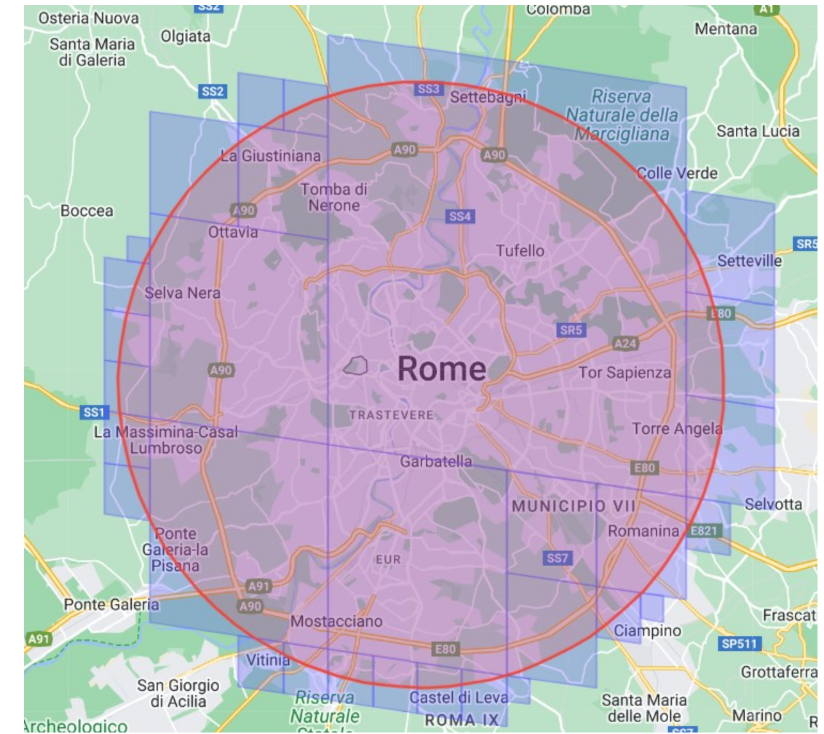


Image source

Our locations are represented as a specific **point** on a long **line**

Example S2 covering

- Given a region, find a set of S2 covering cells
- Parameters: max number of cells, max cell level, min cell level
- Max **level** :13, max **cells**: 45
- **132587f**,1325884,**1325888c**,132588f,**1325894**,132589c,**13258b**,13258c1,**13258c7**,13258c9,**13258cb**,13258eac,**1325f35**,1325f37,**1325f5**,1325f61,**1325f67**,132f58b,**132f58d**,132f593,**132f594c**,132f5c4,**132f5d1**,132f5d7,**132f5dc**,132f5f,**132f64**,132f7b4,**132f7cc**,132f7d4



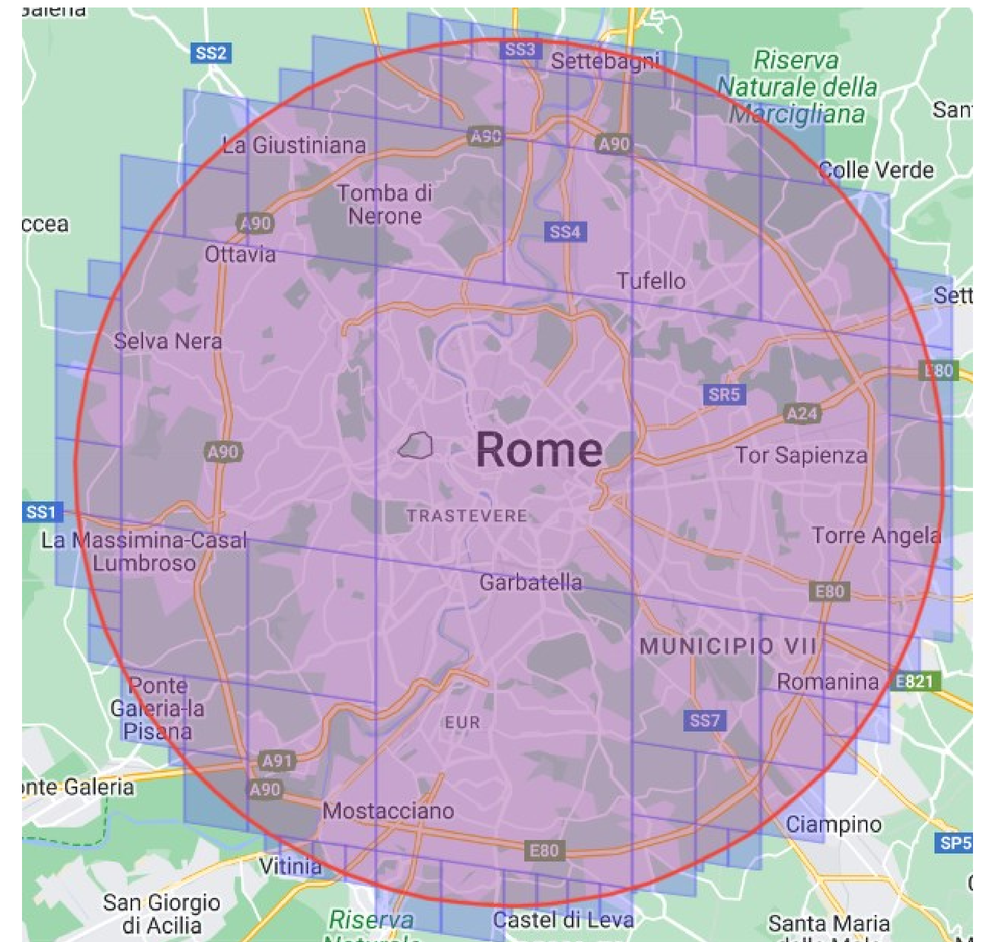
Max # cells	Median ratio (covering area / region area)	Worst ratio
4	3.31	15.83
8	1.98	4.03
20	1.42	1.94
100	1.11	1.19

Generated by [Region Coverer]

Example S2 covering (granular levels)

- Max **level** :30, max **cells**: 100
  - finer covering set of S2 cells
  - tradeoff
    - more precise coverage → fewer false positives
    - more cells → added computational complexity
- cell "levels" (meaning size)
- maximum number of cells covering an area

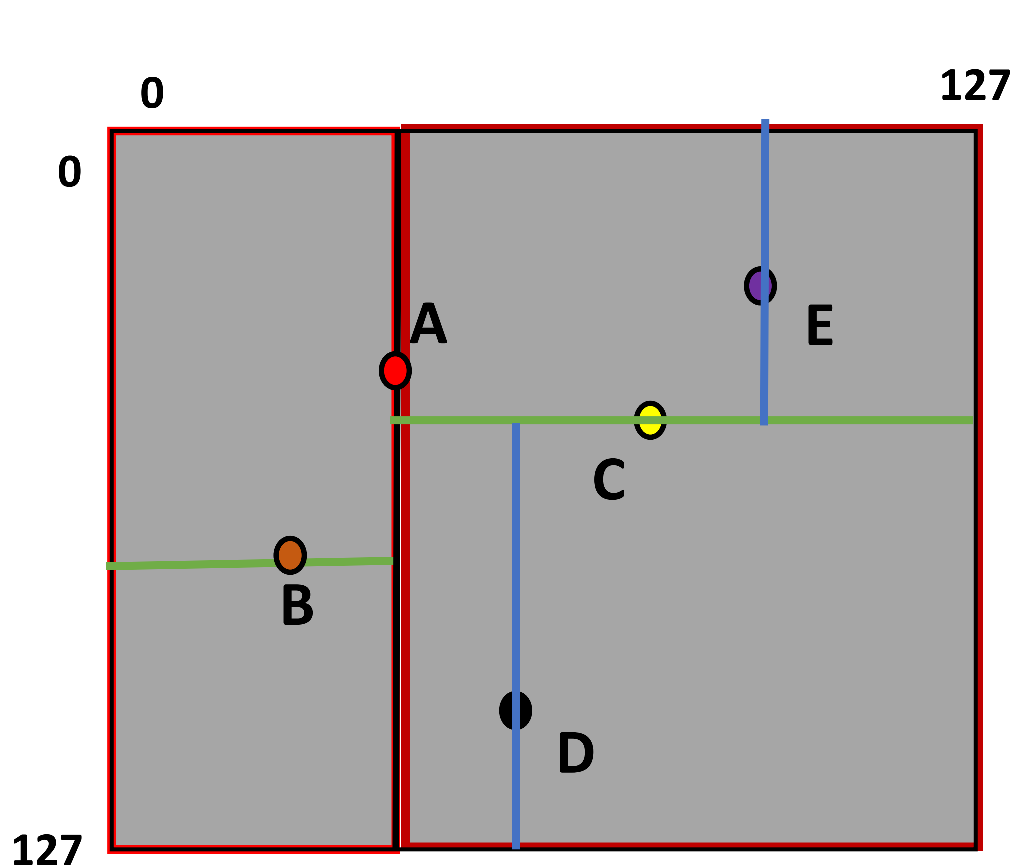
Generated by [Region Coverer]

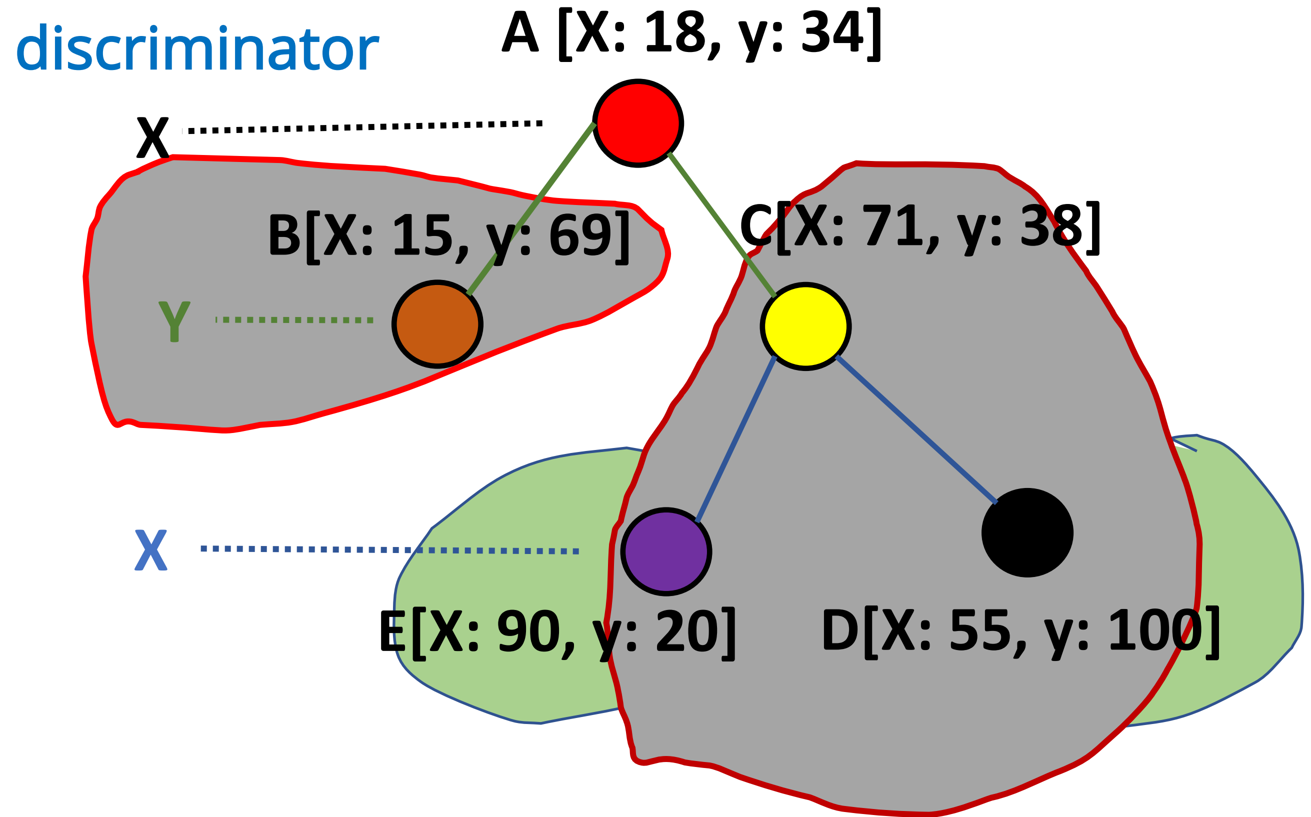


Data-driven spatial data structures

- data-driven→ based upon a **partitioning** of the **data** items themselves
  - Utilizes spatial **containment** relationship in place of the order of the index.
  - Structures that **adapt** themselves to spatial object's **MBRs**

KD Tree insertion

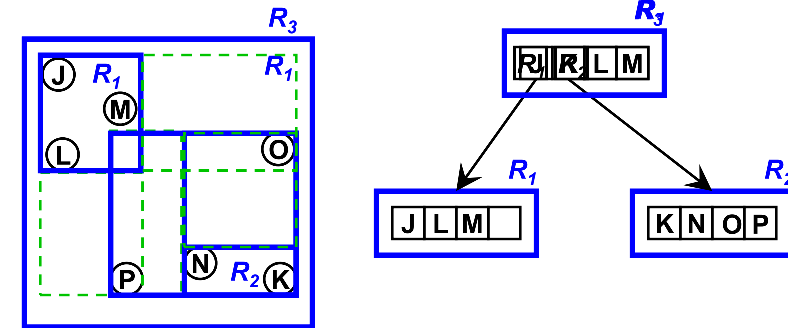


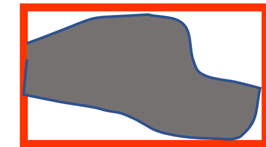


- Recursive decomposition so that only one single point in each leaf node
- approximately half of the leaf nodes will contain no data field

R-tree

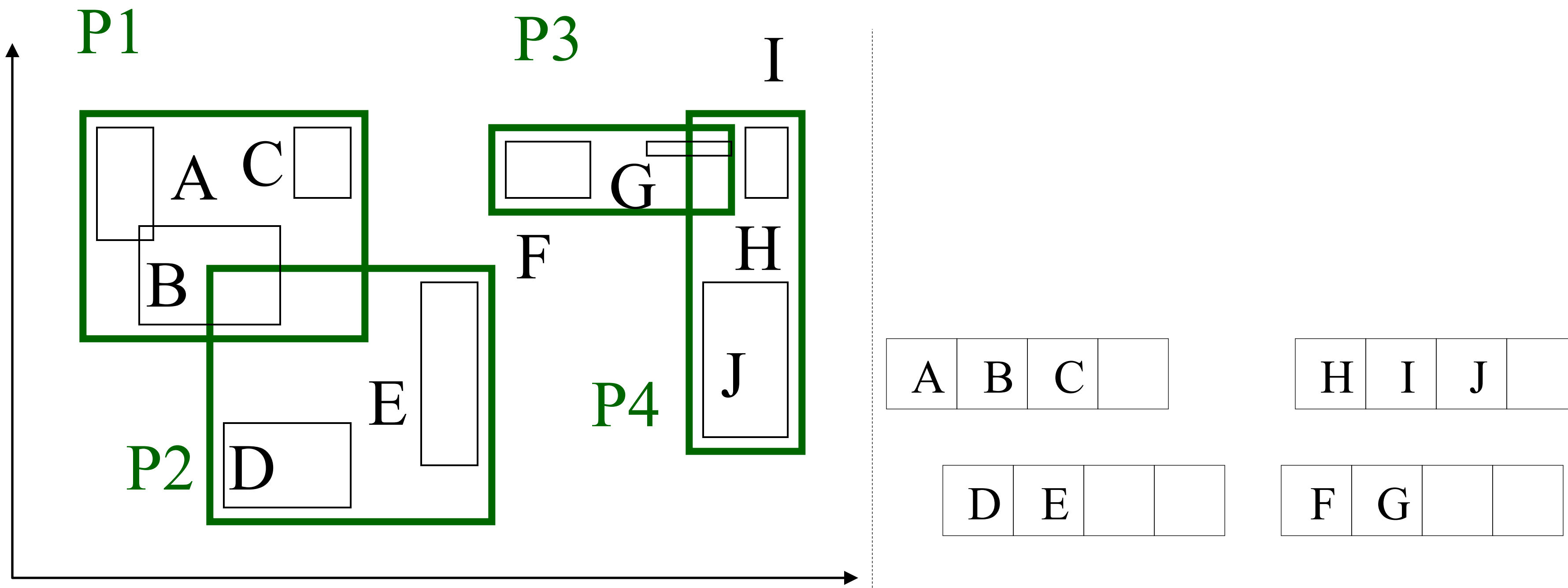
- Minimum bounding rectangle (**MBR**)
  - Group **geographically nearby** objects in same leaf nodes
  - Each node represents the smallest rectangle that encloses child nodes
  - Insertion: Find the node that requires the least expansion to include the new object
- **Disk-resident**
- **Index** nodes (internal search nodes) and **data** (leaf) nodes
  - All leaf nodes on the same level
  - Spatial objects belong to one of the leaf nodes **only**
    - But MBRs may **overlap** (a problem) such as R1 and R2
  - If the **R-tree** is used **solely** as an **index**, leaf nodes contain **pointers** to spatial objects



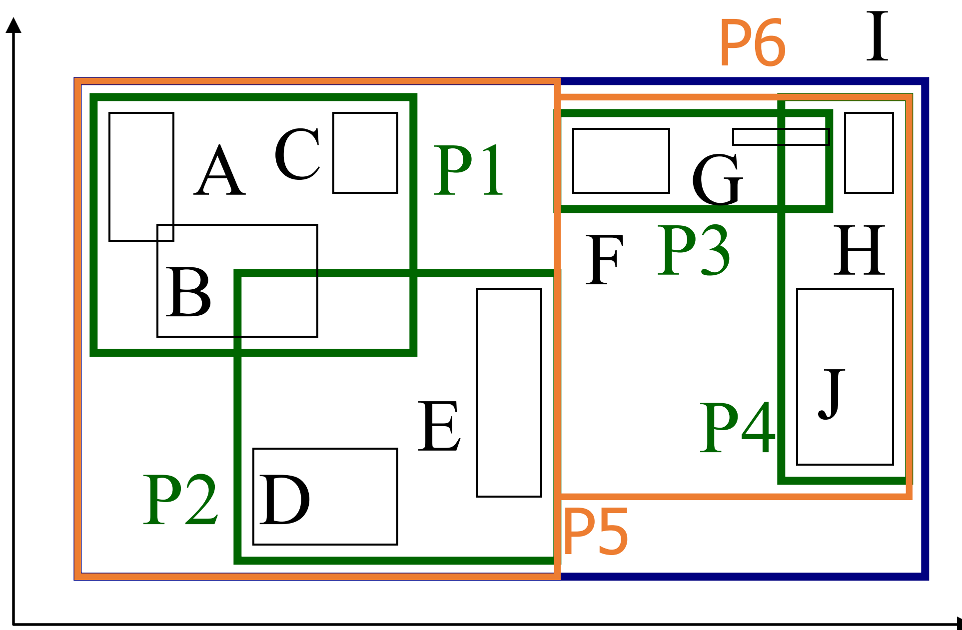


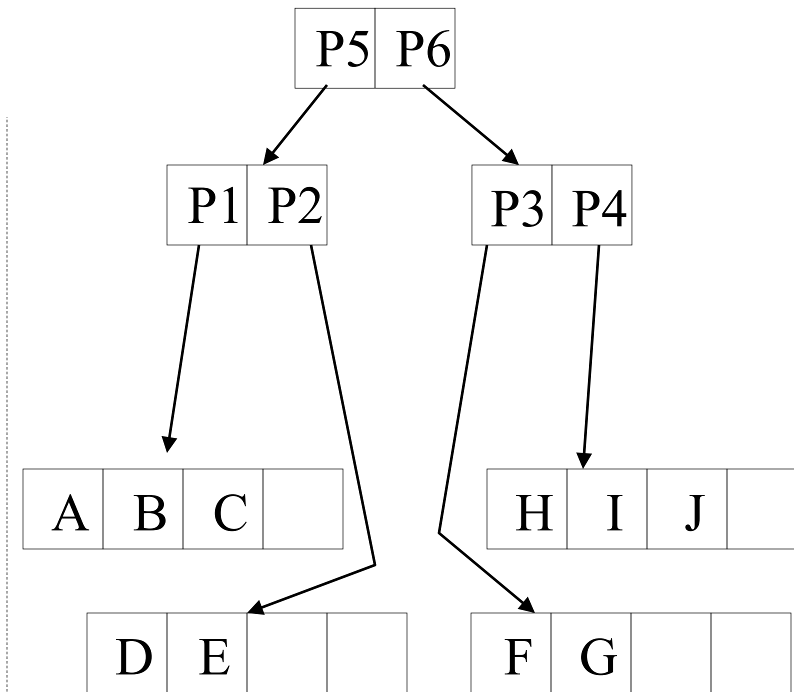
MBR

Another R-Tree example



Another R-Tree example



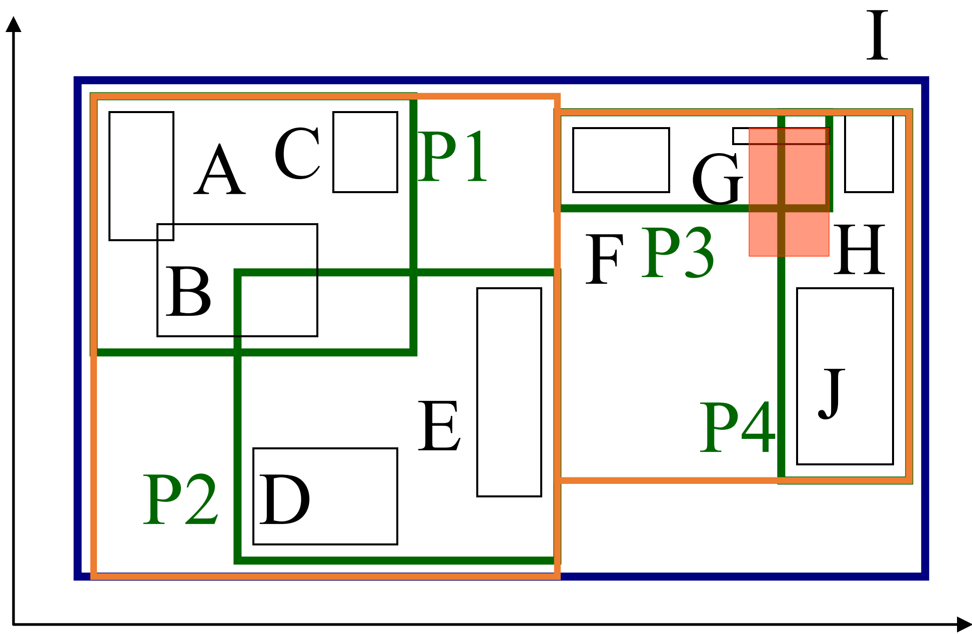


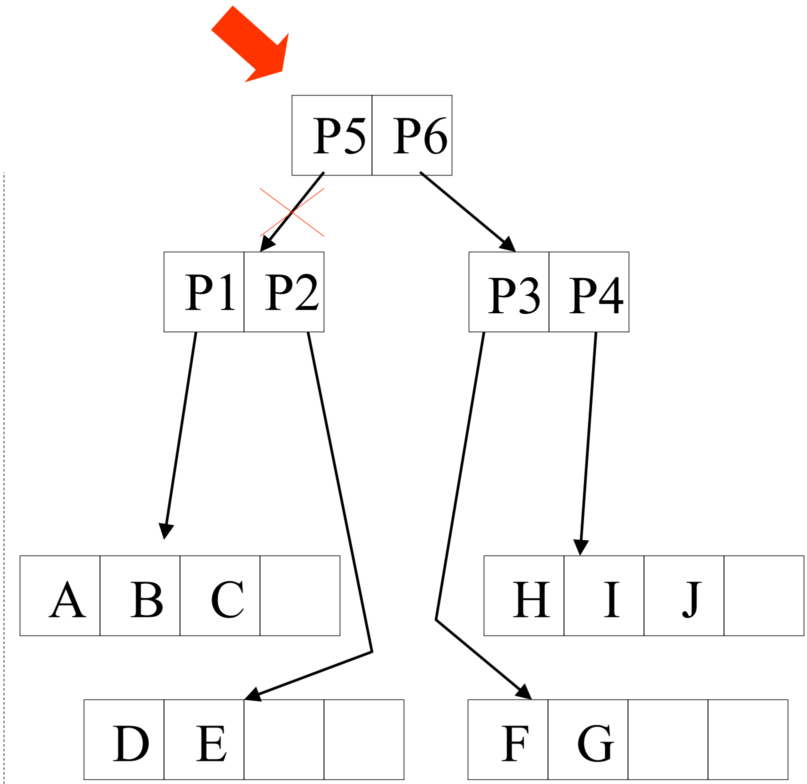
Efficient range query algorithm

- Indexed data (using R-Tree or PR Quadtree) means that data is represented by MBRs
- So, given the query window MBR, it is easy to do a **filter** stage first, checking which **MBRs** from the **tree index** are contained within the **MBR** of the **query window**
  - For each of those **branches**, we **retrieve** the **spatial objects**
  - Apply the **refine** stage checking whether the **candidate** truly satisfies the predicate (**within**, **intersects**, **overlaps**, etc.)

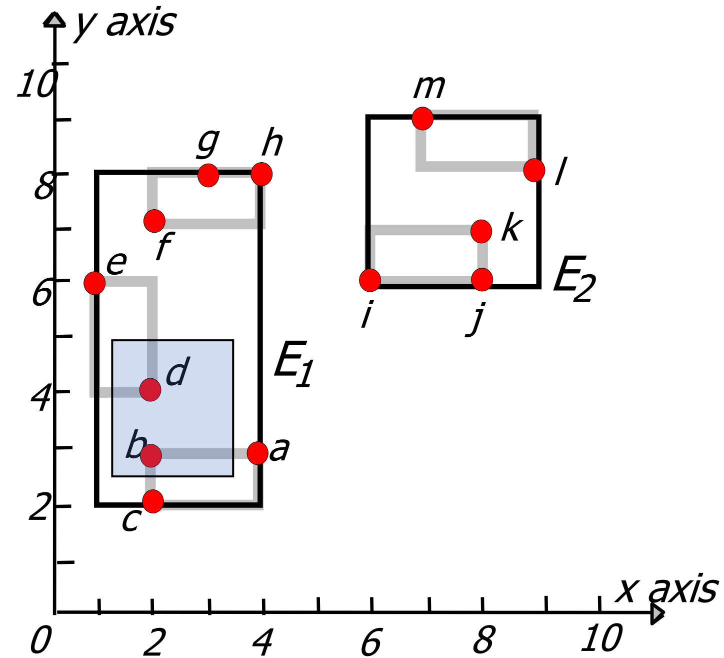
R-trees : Search

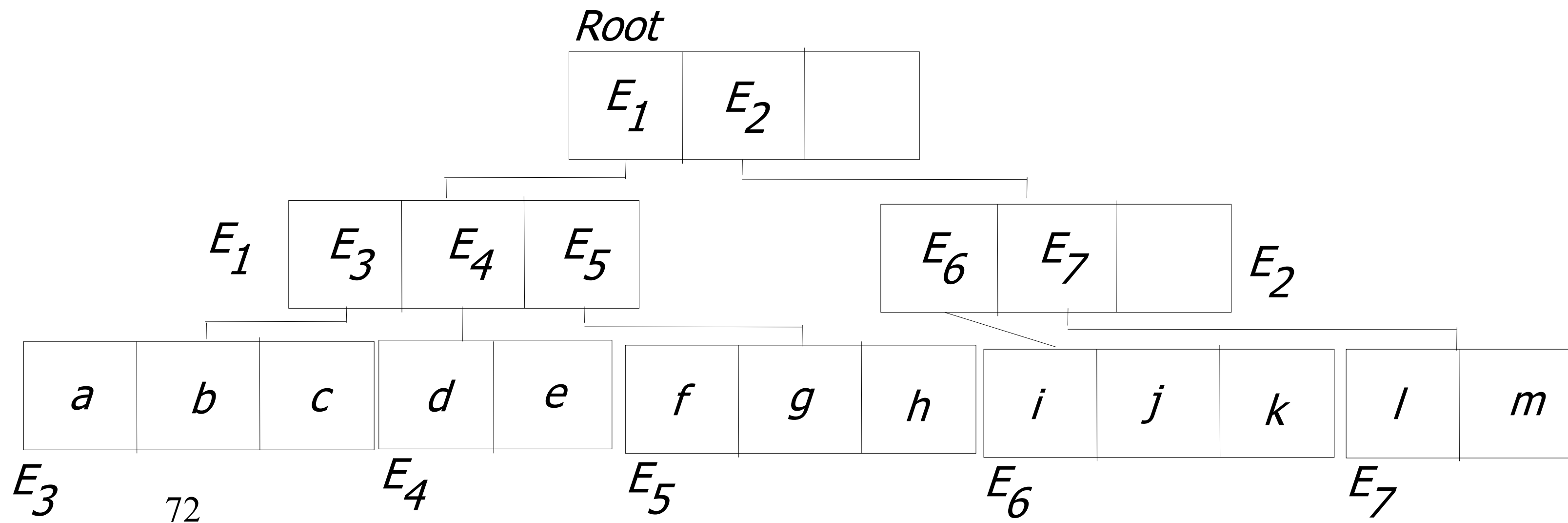
point query may follow several paths (tree branches)



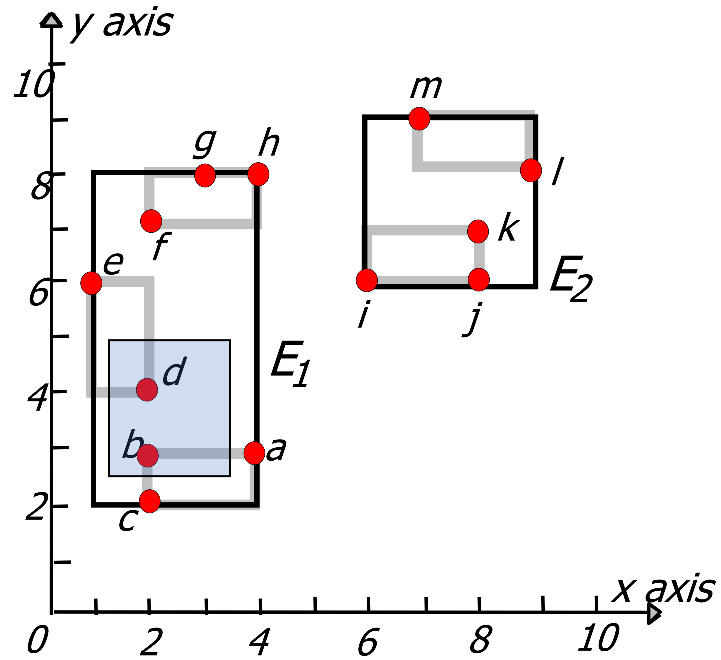


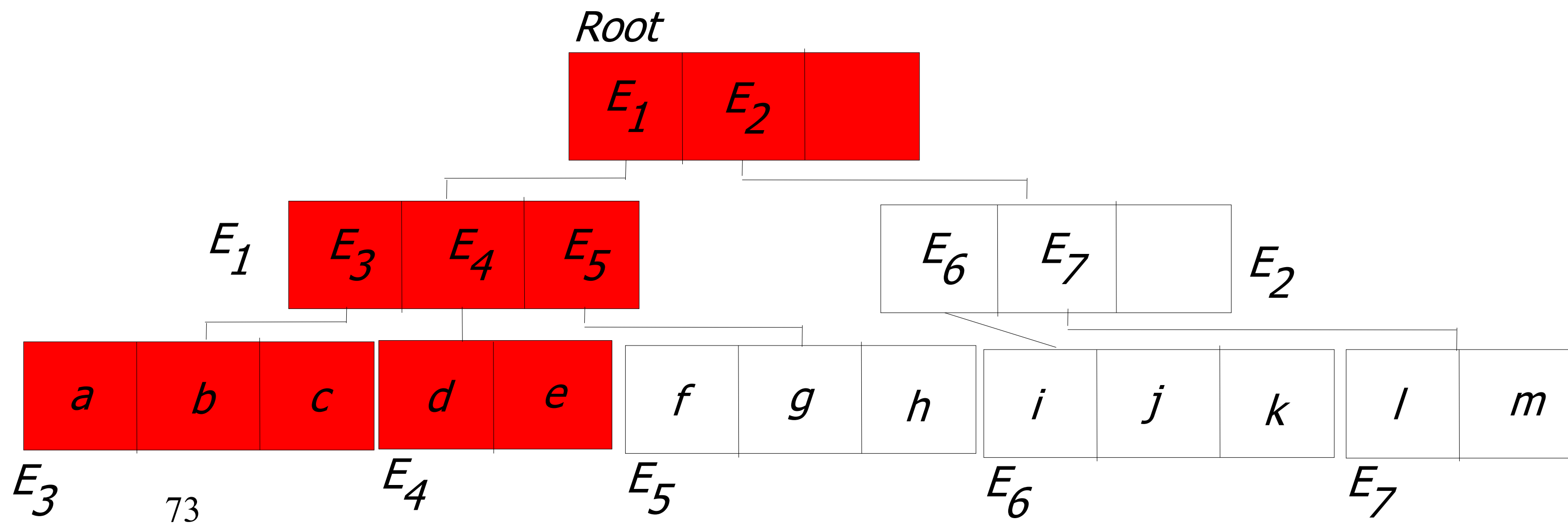
R-tree, Range Query



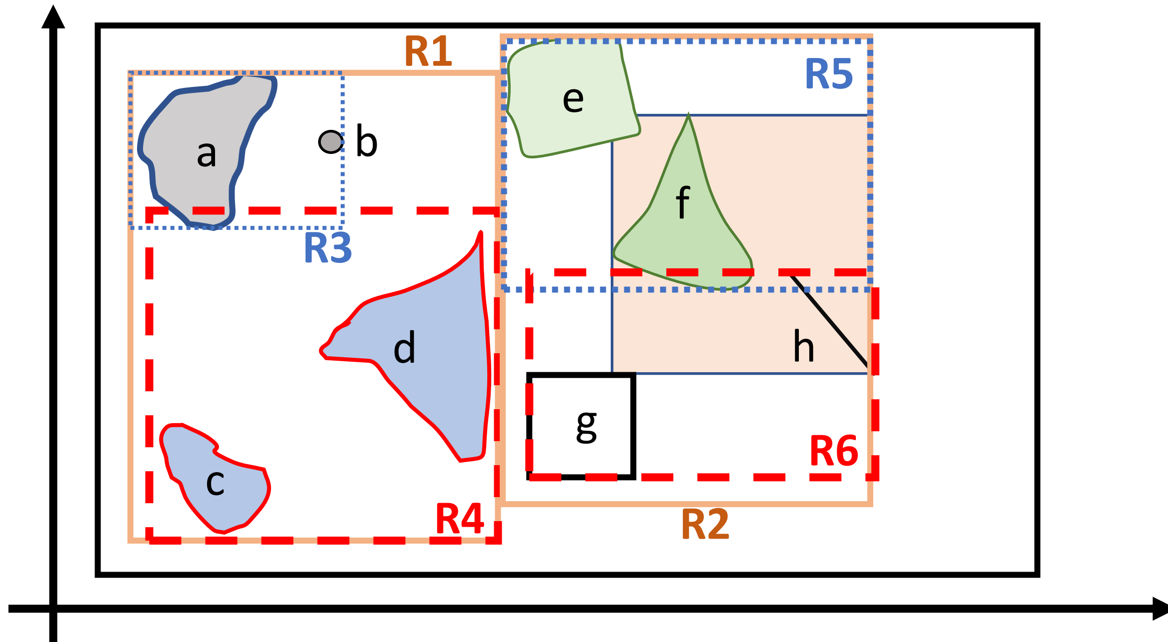


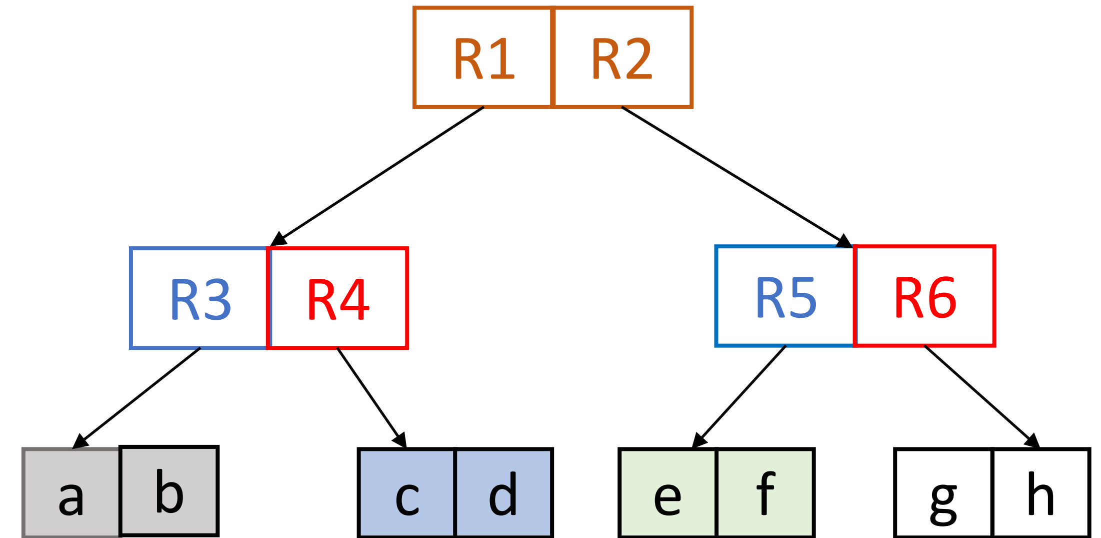
Range Query





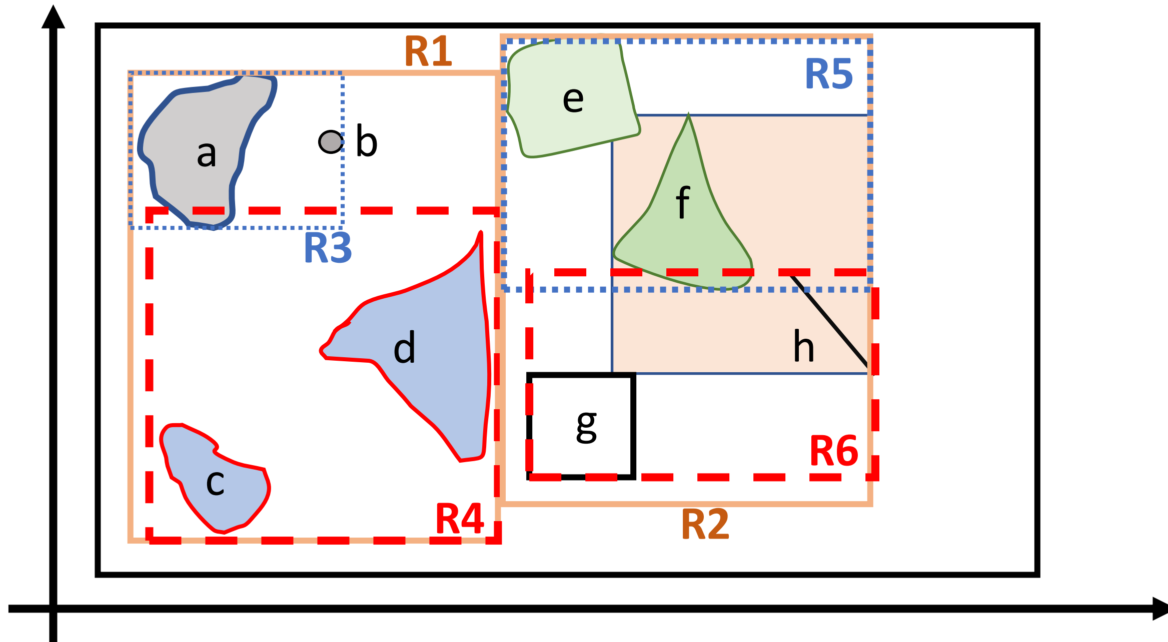
R-Tree construction

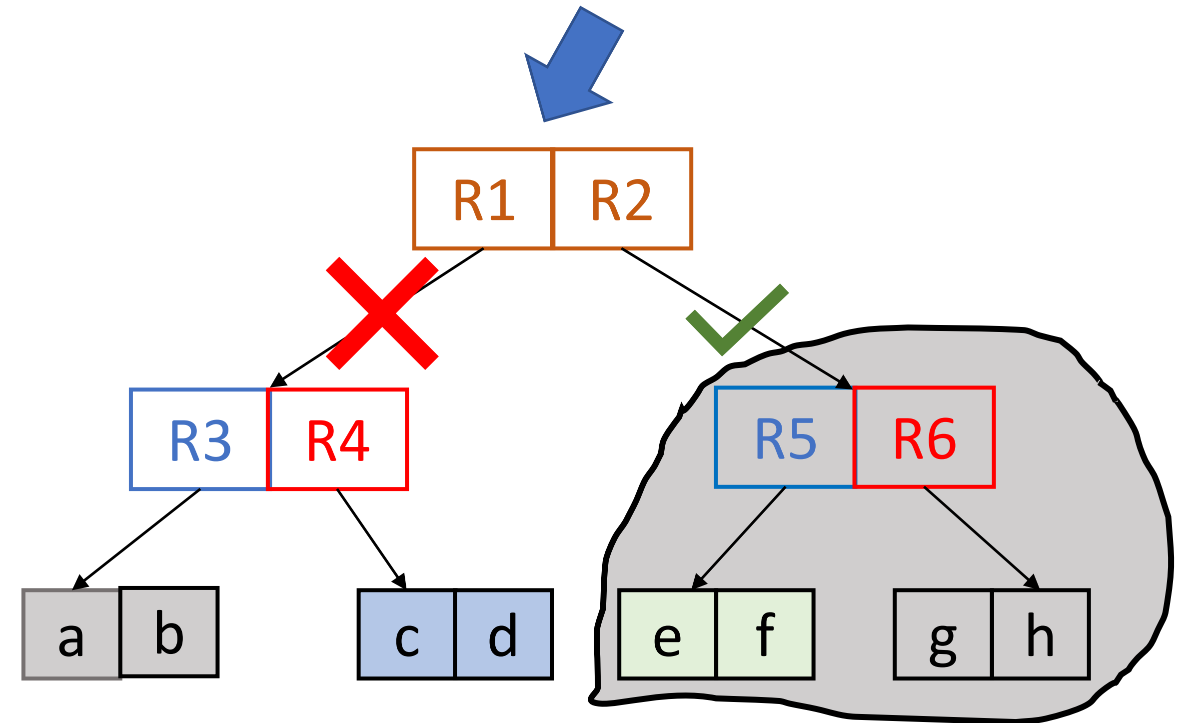




Query window

Range query in R-Tree

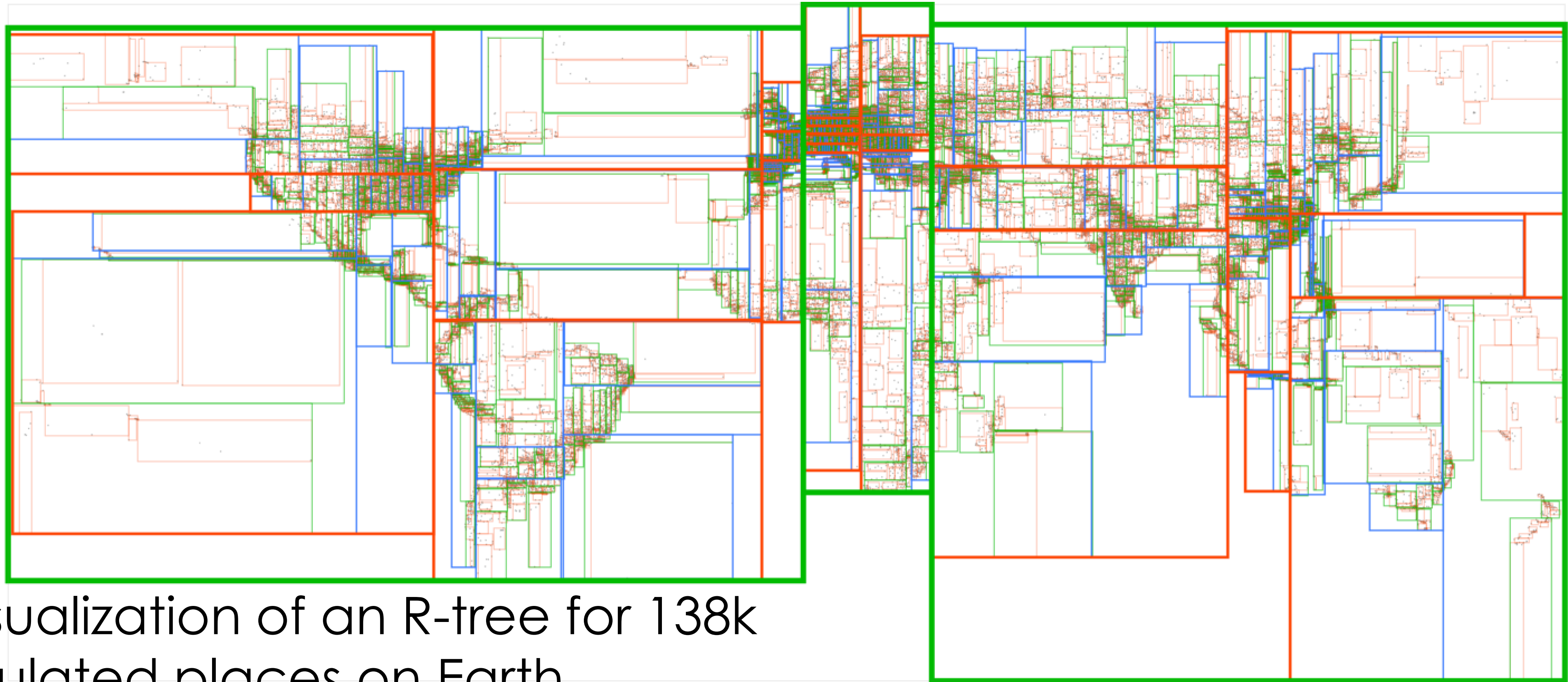




Query window

R-Tree example

a query window which does not intersect the **bounding rectangle** cannot intersect any of its contained objects → **MBR join**

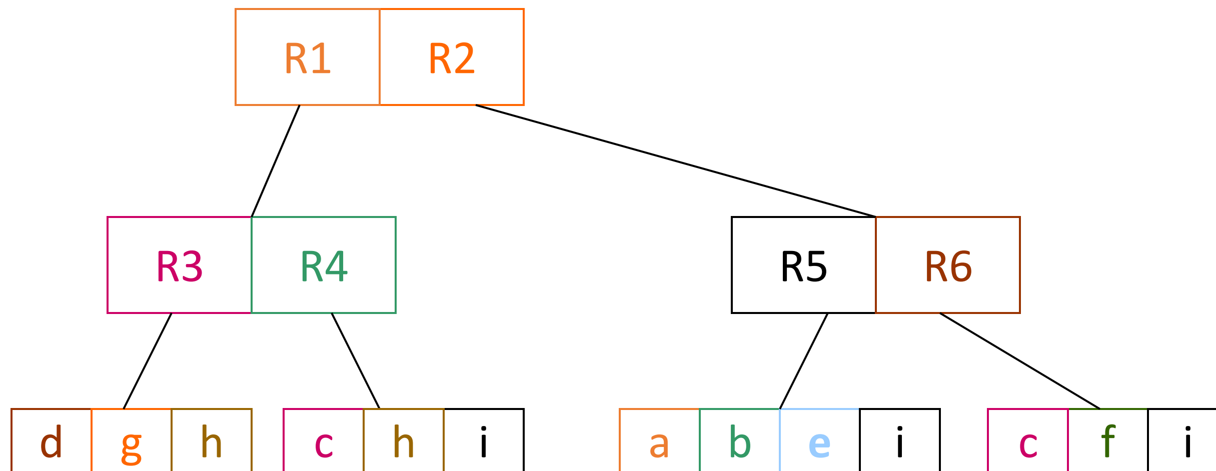


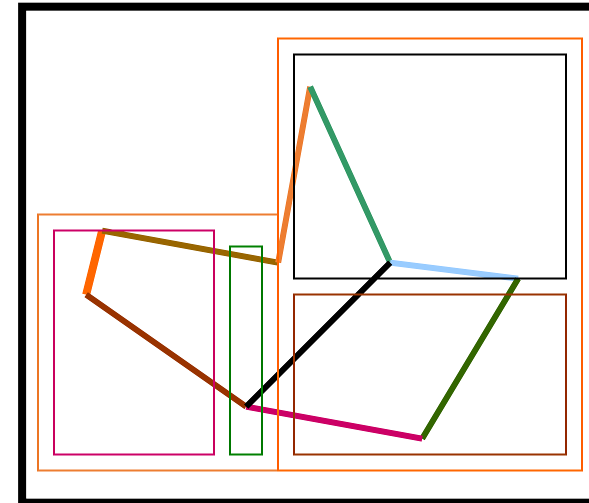
A visualization of an R-tree for 138k populated places on Earth

Image source

R+ - Trees

- **Disjoint decomposition** of the embedding space
  - **No overlaps** between MBRs
  - Spatial objects appear in all MBRs they intersect with
- Efficient **point query** as only one path need to be scanned from root to leaf





Geospatial indexing methods comparison

Index	storage	Efficient query type	Comments
R-tree	Disk-resident	Point, window, **kNN**	
KD-tree	In-memory	Point, window, **kNN**	**Inefficient** for highly **skewed** data
Quad-tree	In-memory	Point, window, **kNN**	**Inefficient** for highly **skewed** data
Z-curve + B^+-tree	Disk-resident	Point, window	**Order** of **Z-curve** has an impact on performance

How to choose a spatial data structure

- performance factors
  - ***Preprocessing*** *Cost*. Index **construction** cost
  - ***Storage*** *Cost*. Index **storage**
  - ***Query*** *Cost*. The **search** time or **query** cost by utilizing the index structure
- **Space-driven** spatial index → structure of the index is created first, then data is added step-wise
  - Does not require changes to the index structure for insertion
  - Facilitates **merging** (**fusing**) heterogeneous data sources indexed with common grid
- **Data-driven** structures → efficient for **storage** and **faster** in search scans, but tied to specific data

Storage and processing of big geospatial data

Example Cloud software frameworks

(Geomesa, GeoSpark, GeoFlink, geospatial in MongoDB, GeoSparkViz, HadoopViz, etc.)

Problem

- Big geospatial data
  - GDELT: Global Database of Event, Language, and Tone
    - ~225-250 million records
  - **Mobility data** is gathered by cell phone providers
    - Millions of records
- How do we handle big vector geospatial data?
  - millions to billions of rows of vector geospatial data (mostly points) arriving every day?

GeoMesa

- Constellation of **tools** for **querying** and **analytics** of **big geospatial data** on **distributed computing systems**.
  - **Streaming**, **persisting**, **managing**, and **analyzing** spatial data **at scale,** with QoS guarantees
  - Efficient **spatial indexing** atop **HBase**, **Bigtable** and **Cassandra** storage systems for **scalable storage** of **vector geospatial data** (point, line, polygon)
  - Near real time **geospatial data** **stream processing** atop Apache **Kafka**
  - Supports Apache **Spark** for geospatial big data stream & batch processing
  - Integrate well with mapping clients (Web Feature/Mapping Service, WFS and WMS)
- In summary, all the **Lambda architecture** layers are supported, in addition to mapping (**geo-visualization**)

GeoMesa Architectural Overview

- **Scalable**, **cloud-based** data **storage**
  - Apache **Accumulo**, Apache **HBase**, and Google Cloud **Bigtable**,
- Apache **Kafka** message broker for **streaming data**
- Apache **Storm** for **batch distributed processing (replaying)** of streaming data with GeoMesa
- Apache **Spark** for large-scale analytics of stored (**batch**) and **streaming** data

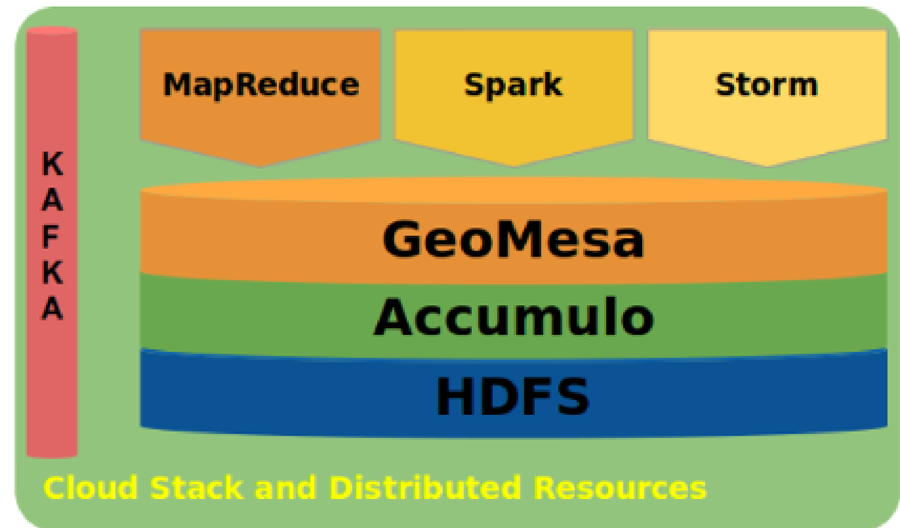


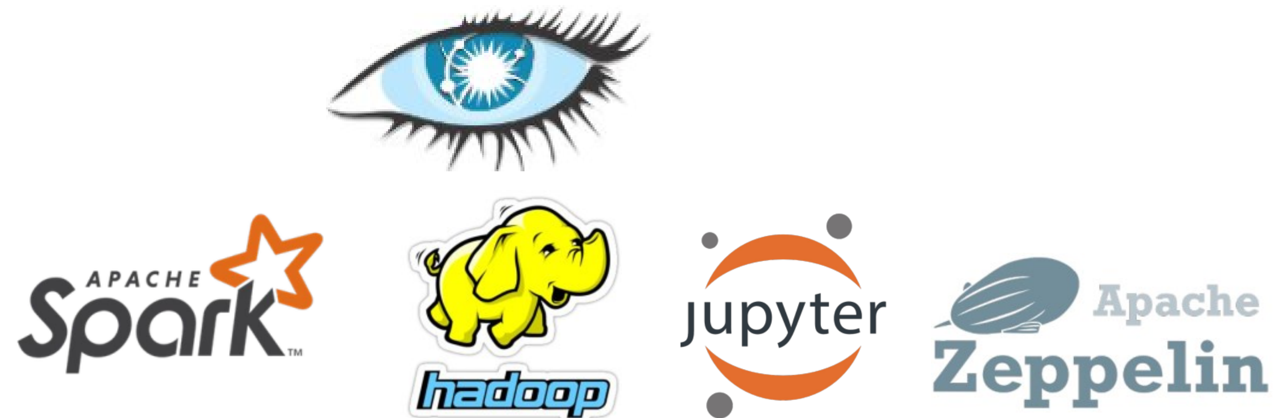
Image source

Technology stack supported in GeoMesa

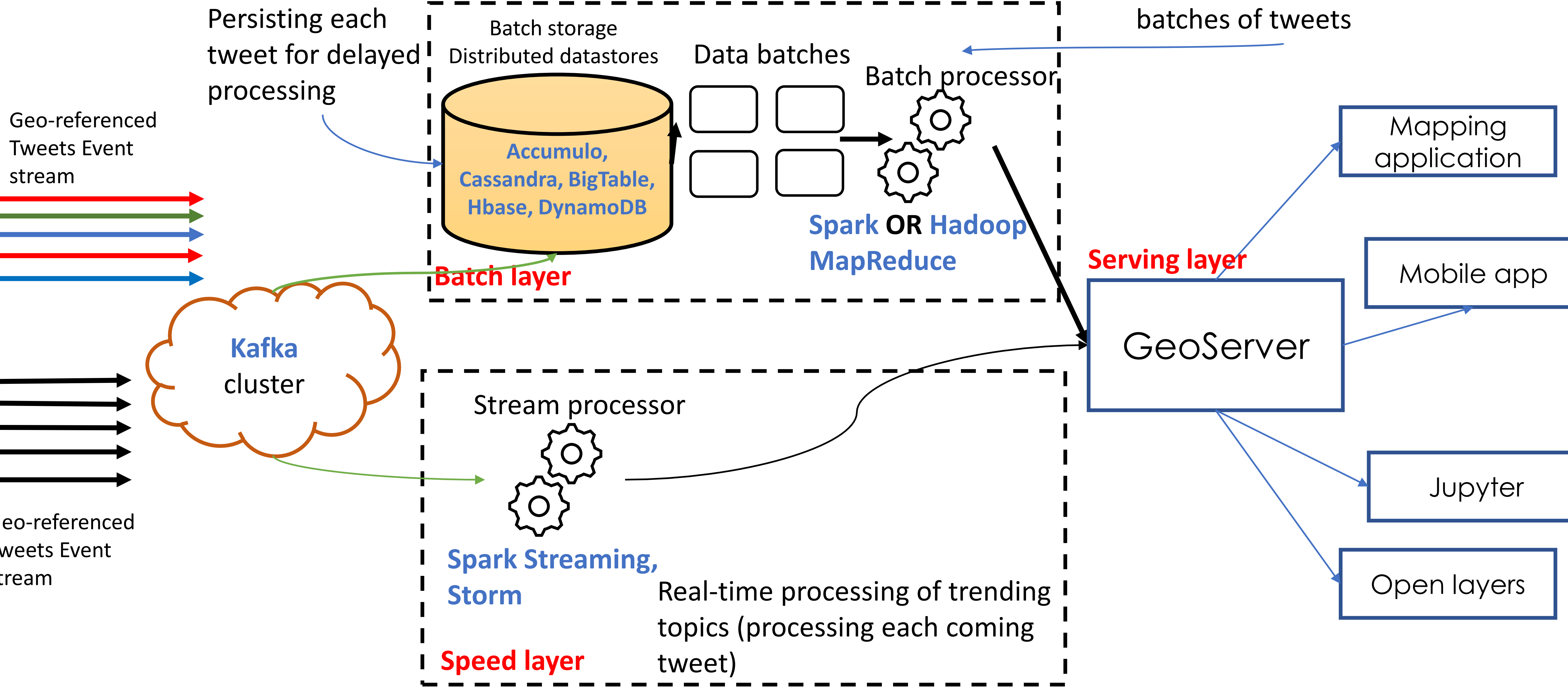
Streaming

Persisting

Analyzing



Lambda Architecture revisited with GeoMesa Geospatial intrinsic support



Spatial Analytic Pipeline with GeoMesa encapsulated

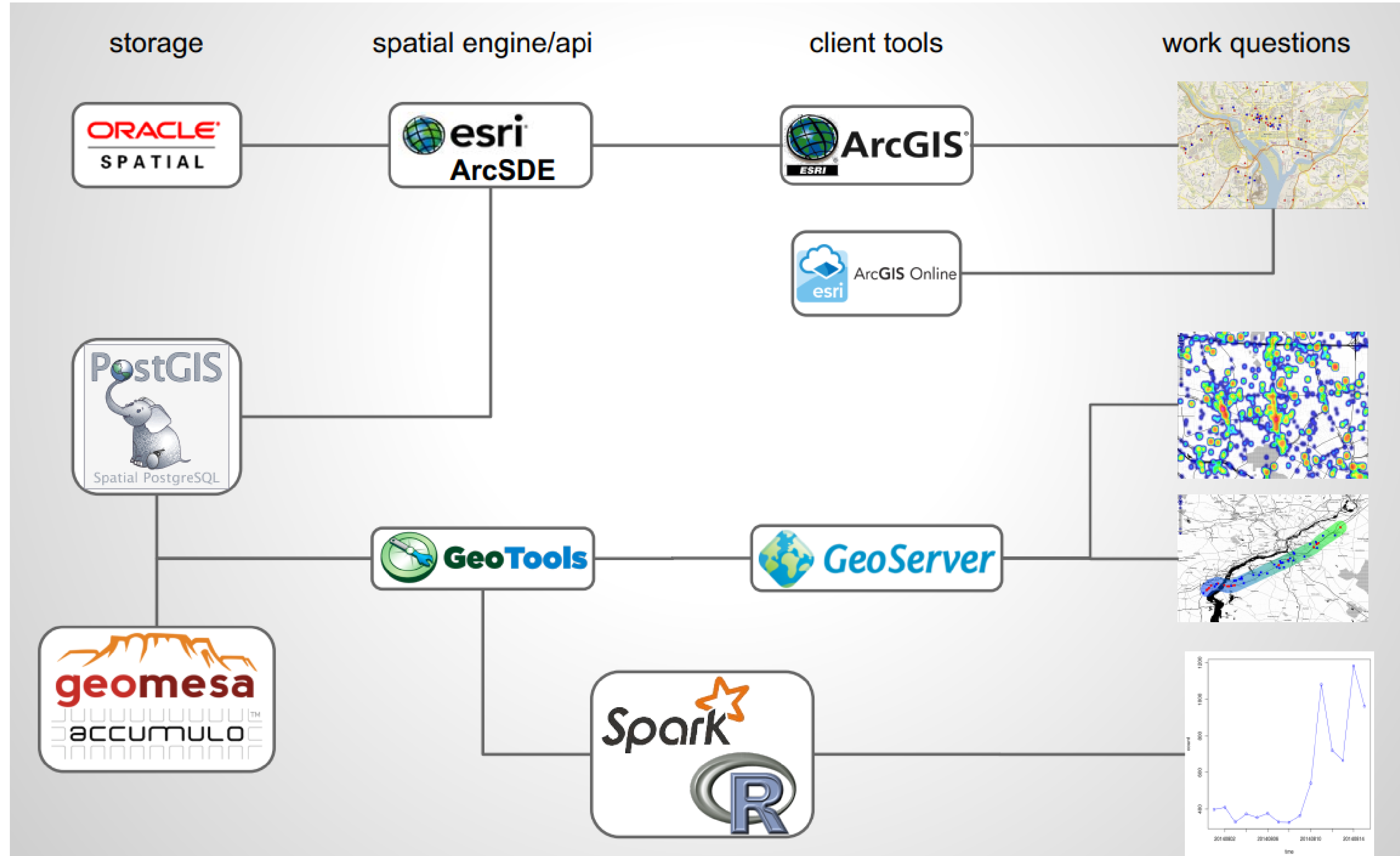


Image source

JSON examples for geo-referenced Tweets

{ "geo": null, "coordinates": null, "place": { "id": "07d9db48bc083000", "url": "https://api.twitter.com/1.1/geo/id/07d9db48bc083000.json", "place_type": "poi", "name": "McIntosh Lake", "full_name": "McIntosh Lake", "country_code": "US", "country": "United States", **"bounding_box": { "type": "Polygon", "coordinates": [[[-105.14544, 40.192138], [-105.14544, 40.192138], [-105.14544, 40.192138], [-105.14544, 40.192138]]] }**, "attributes": { } } }

Tweet with Twitter Place

{ "geo": { "type": "Point", "coordinates": [40.74118764, -73.9998279] }, "coordinates": { **"type": "Point", "coordinates": [-73.9998279, 40.74118764] },** "place": { "id": "01a9a39529b27f36", "url": "https://api.twitter.com/1.1/geo/id/01a9a39529b27f36.json", "place_type": "city", "name": "Manhattan", "full_name": "Manhattan, NY", "country_code": "US", "country": "United States", "bounding_box": { "type": "Polygon", "coordinates": [[[-74.026675, 40.683935], [-74.026675, 40.877483], [-73.910408, 40.877483], [-73.910408, 40.683935]]] }, "attributes": { } } }

Tweet with exact location

Code source

Example geo-Query

- Find the tweets near Bologna which were re-tweeted eight times at least
- SELECT * FROM tweetsDF WHERE
  retweetsCount > 8
  AND (lat > 44.5 AND lat < 44.7)
  AND (lon > 11.3 AND lon < 11.5)
- This is inefficient
  - We need specialized libraries

SELECT * FROM tweetsDF, cities WHERE
retweetsCount > 8
AND ST_Contains(tweetsDF.geom, city.geom)
AND cities = “Bologna”

SELECT * FROM tweetsDF, cities WHERE
retweetsCount > 8
AND ST_dwithin(tweets.geom, city.geom, 3000)
AND cities = “Bologna”

Tweeting while Driving : GeoMesa

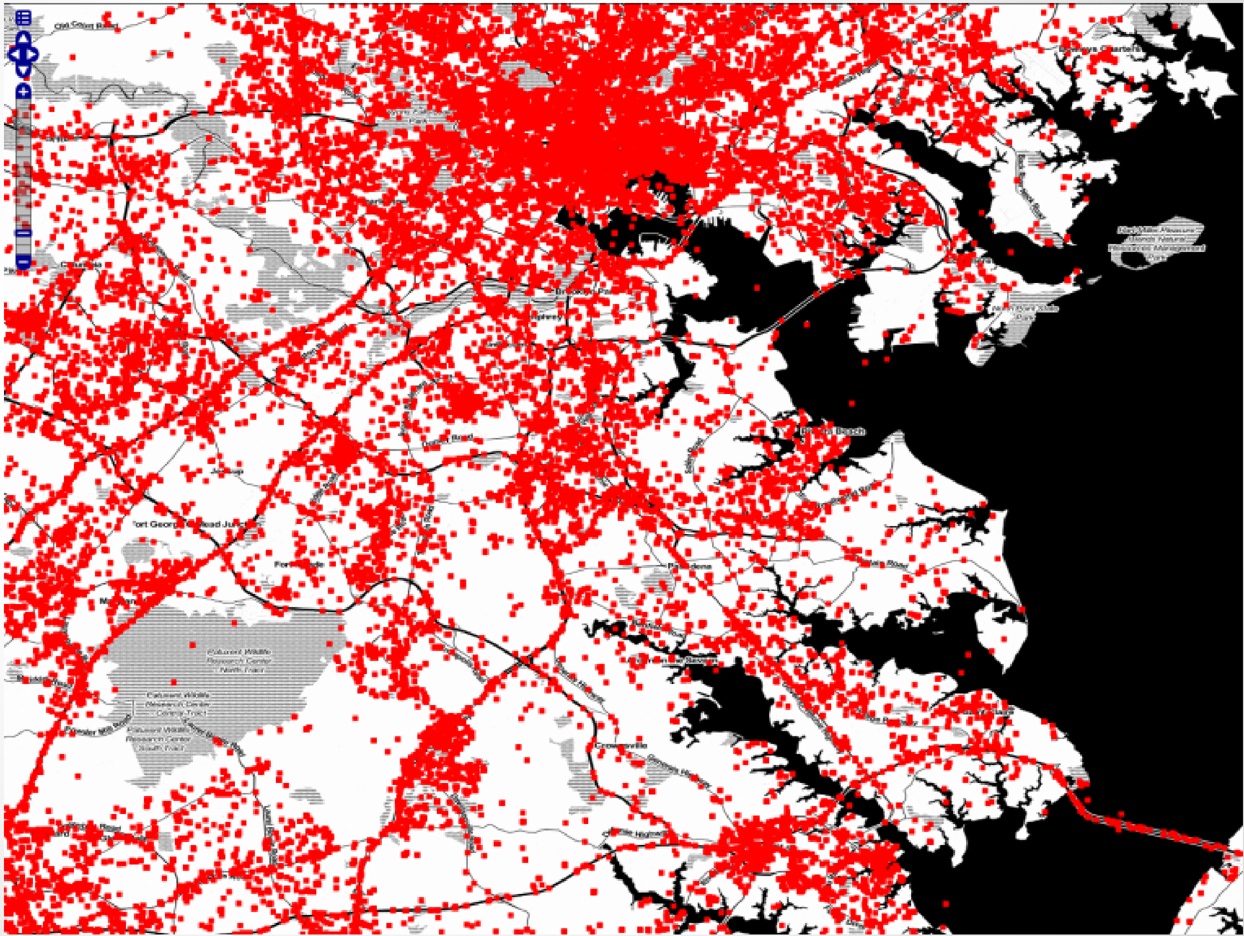
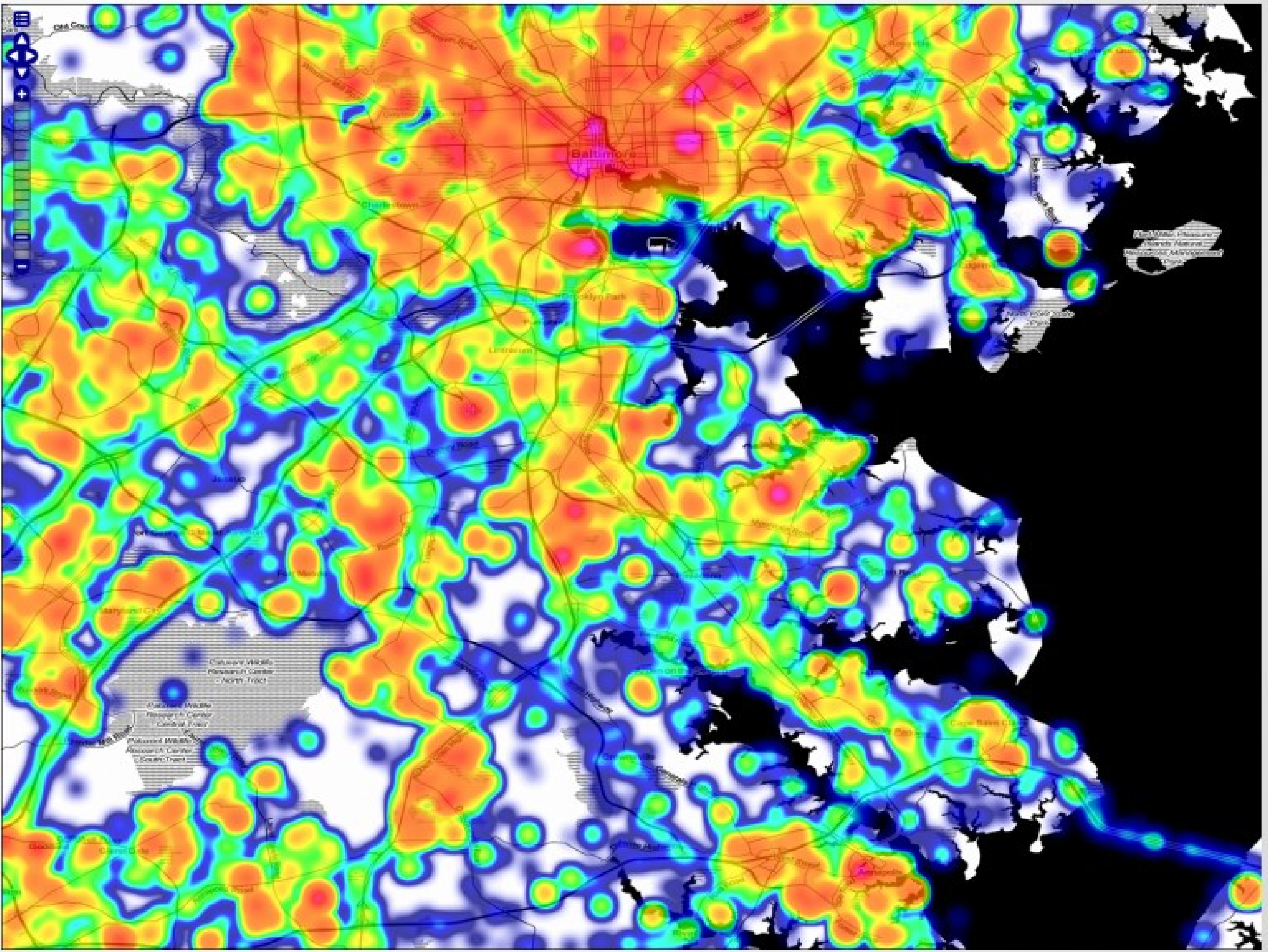


Image source

Tweeting while Driving Heatmap: GeoMesa



[Image source]

Geospatial Indexing in GeoMesa

- **Dynamic** indexing
- **Geohash** to encode **geospatial** data
  - The backing datastore of GeoMesa is **Accumulo**
  - Key/value store, with an indexing based on the **lexicographical** ordering of the keys
  - Requires mapping **2-D** coordinates into a **single** dimension (Accumulo keys)
- Given a query **polygon**, find the **list** with minimum number of **geohashes covering** the polygon
  - Shaded red are Geohashes that constitute **prefixes** that remain in the **result set**
  - Dark-shaded geohashes are **rejected**, because they do not intersect the **covering polygon**

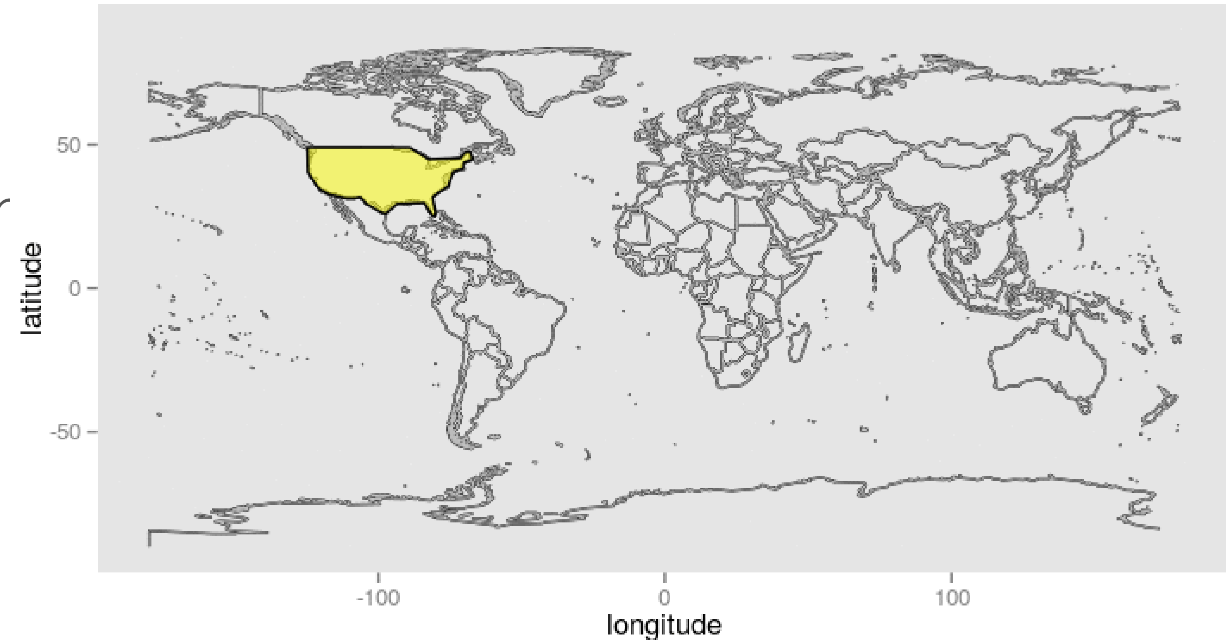


Image source

Geospatial Indexing in GeoMesa

Two basic types based on space-filling curves

- **Z2**
  - A two-dimensional **Z-order** curve to **index latitude** and **longitude** for **point vector** data.
  - Created if the feature type has the geometry type **Point**.
- **xz2**
  - uses a 2-D implementation of XZ-ordering [1] to index **latitude** and **longitude** for **non-point vector data (lines and polygons)**.
  - An extension of **Z-ordering** designed for spatially objects with **extents** (i.e., non-point geometries such as **line** strings or **polygons**).
  - Created if the feature type has a non-Point geometry.