

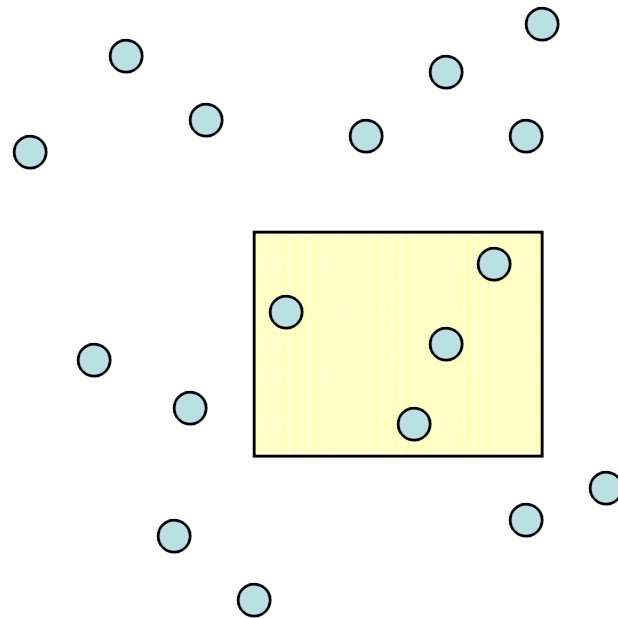
---

# Multi-dimensional Data and Spatial Range Query in Sensor Networks

# Orthogonal range search

---

- Find all the sensors inside a rectangular box.
- Find all the sensors with temperature readings above 70F.



# Multi-dimensional data

---

- Monitor environments.
- Multiple sensors, multiple attributes.
- Query might be multi-dimensional as well.

List all sensors with temperature value 70-80 and light level 10-20.

# Sensor network as a database

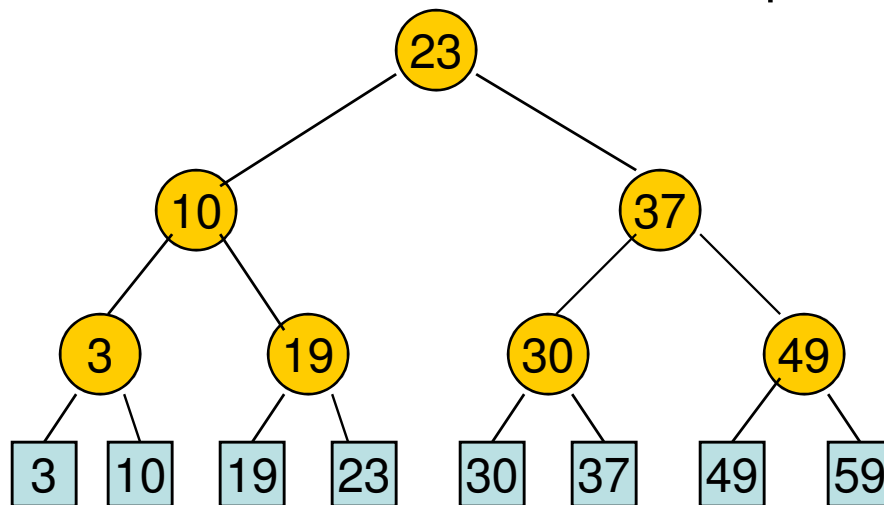
---

- Need an indexing scheme.
- .... In addition, a storage scheme.
  
- First we look at range query in a centralized setting.

# 1D range search

---

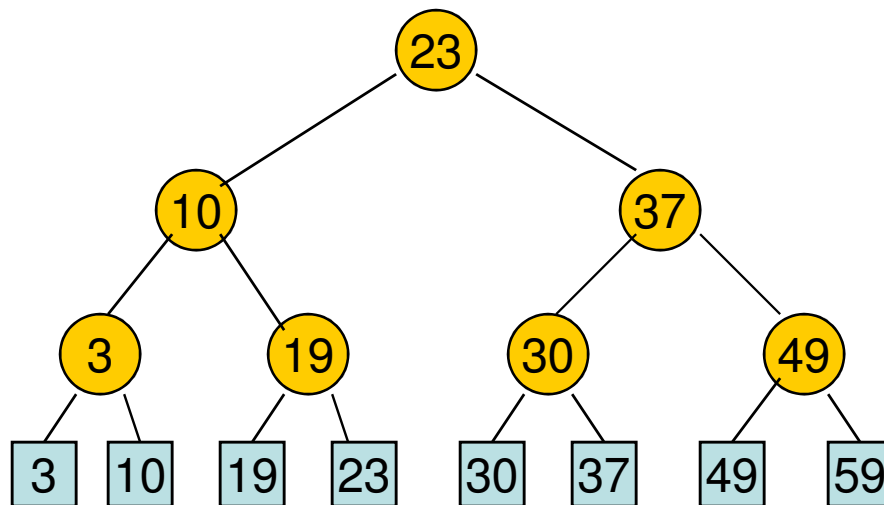
- Find the data inside a query interval  $[x, x']$
- 1D range tree: a balanced partitioning tree on a sorted list.
  - Each leaf stores an input value.
  - Each internal node stores the splitting value.



# 1D range search

---

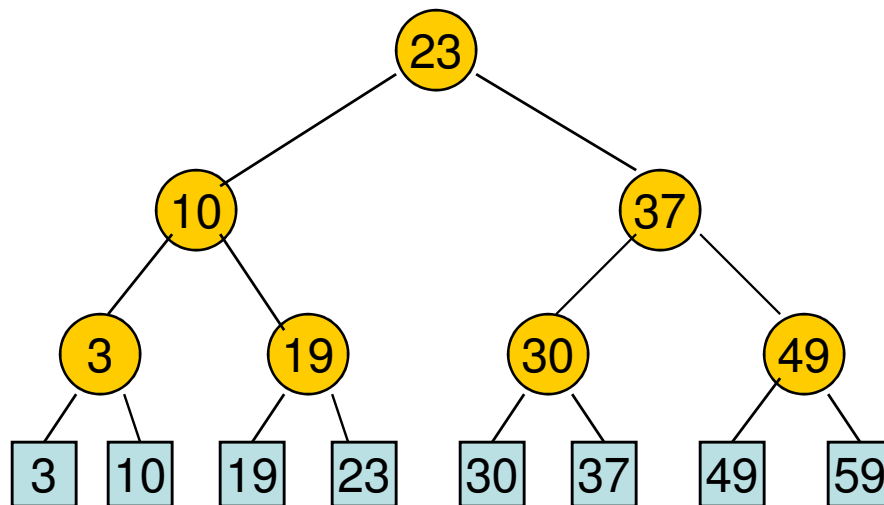
- Find the data inside a query interval  $[x, x']$ 
  - Start from the root and descend the tree to find the interval where  $x$  and  $x'$  stays.
  - Include all the leaves in the sub-trees between the two traversing paths from the root.
- Example  $[9, 33]$ .



# 1D range search

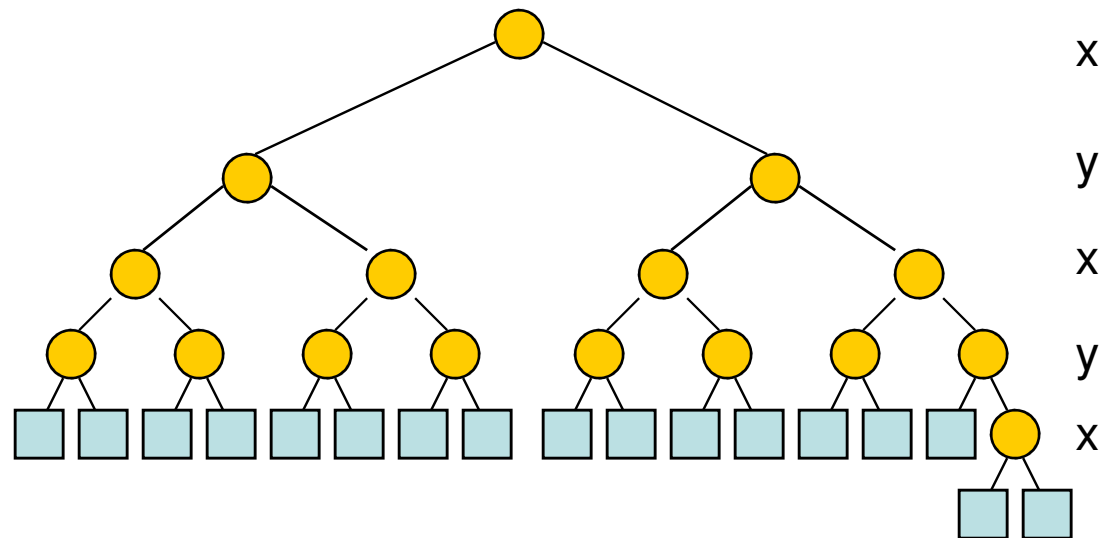
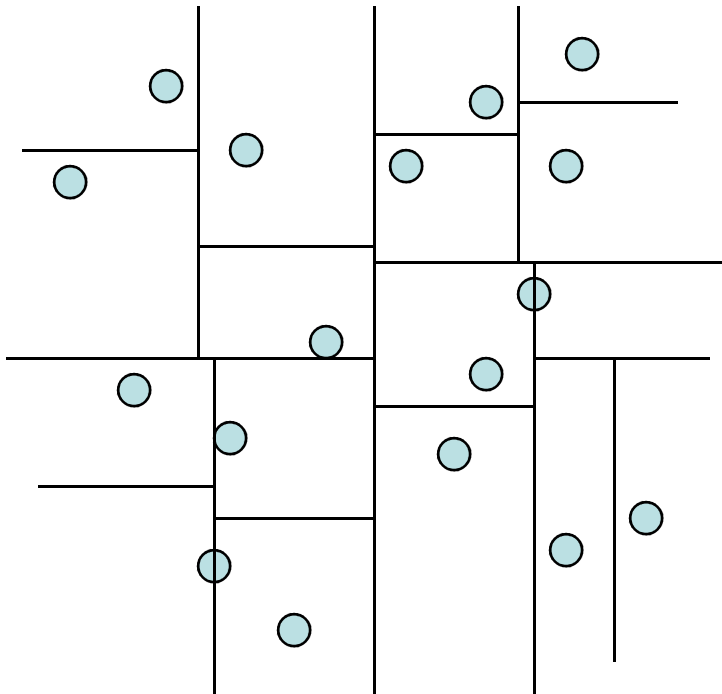
---

- Storage:  $n+n/2+n/4+\dots+1=2n=O(n)$
- Height of the tree:  $O(\log n)$
- Query time:  $O(\log n+k)$ , where  $k$  is the output size.



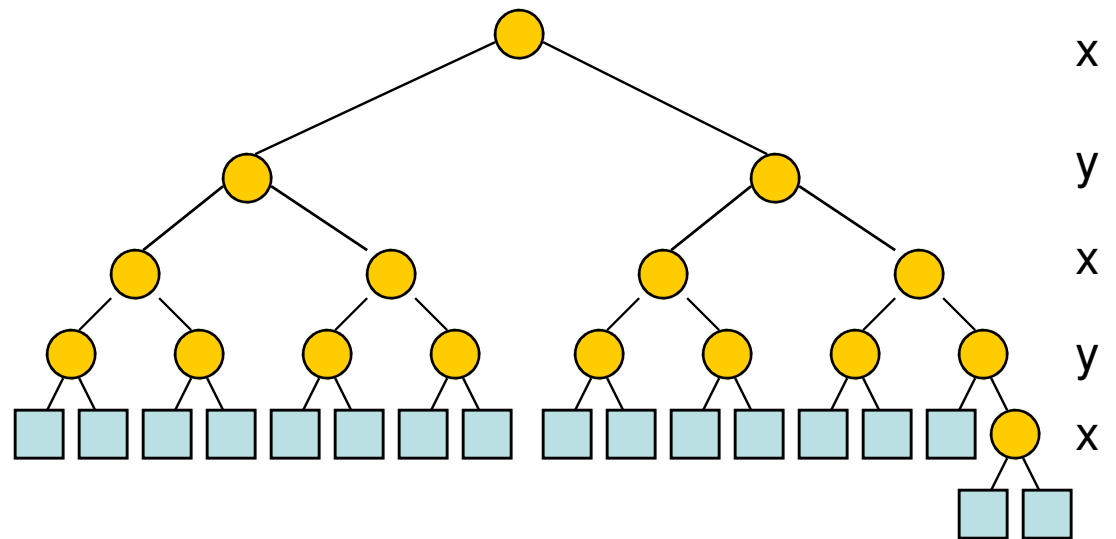
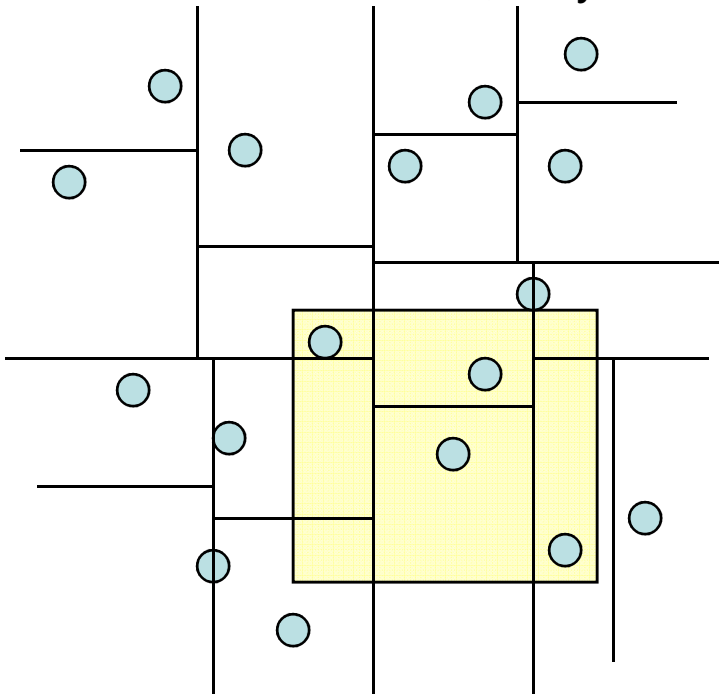
# Kd-tree

- A recursive space partitioning tree.
  - Partition along x and y axis in an alternating fashion.
  - Each internal node stores the splitting node along x (or y).



# Kd-tree

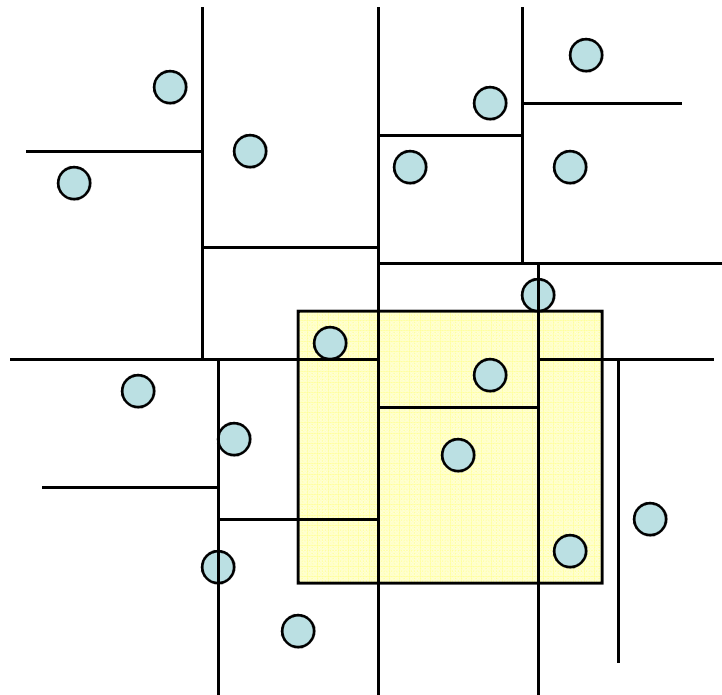
- 2D query  $R=[x, x'] \times [y, y']$ .
  - Check with each internal node whether the cutting line intersects  $R$ .
    - If yes, recurse on both.
    - If no, only recurse on the half plane that intersects  $R$ .



# Kd-tree

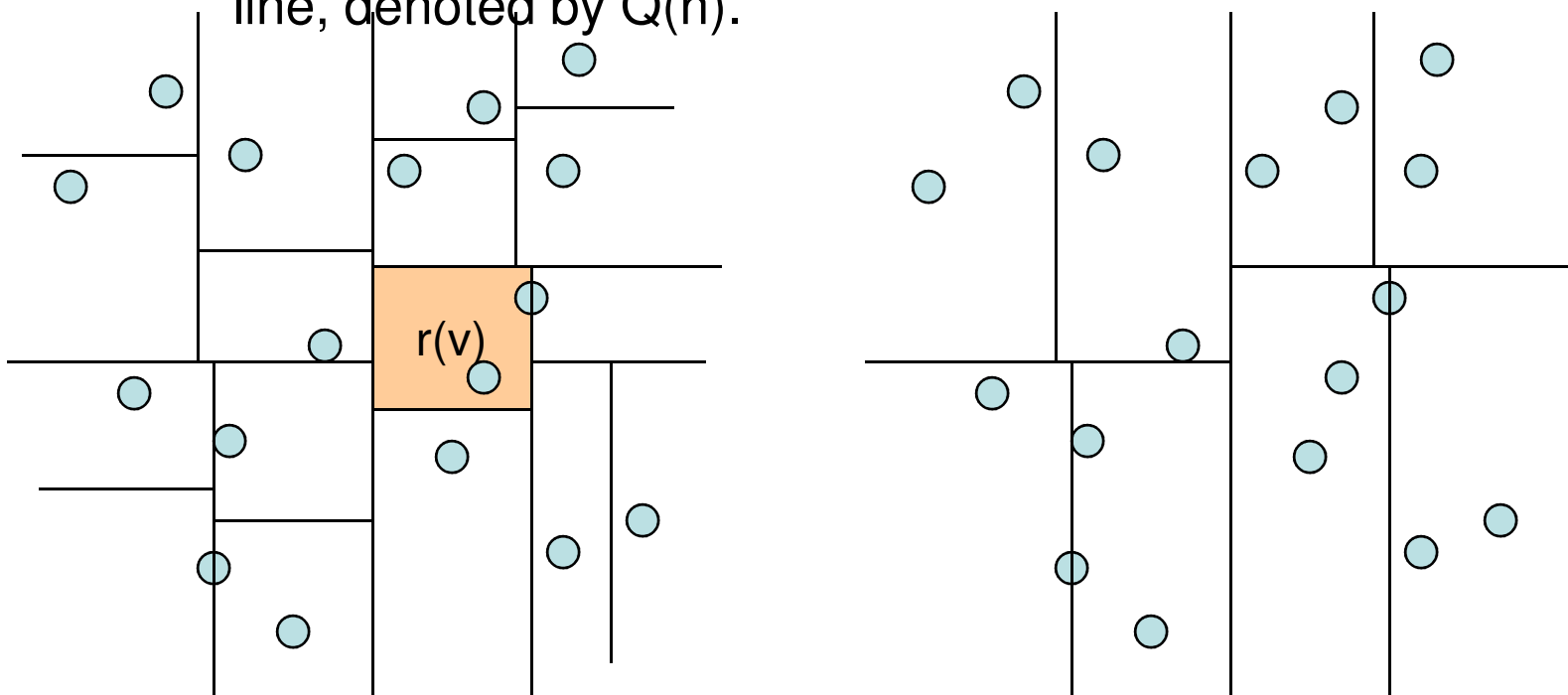
---

- Storage:  $O(n)$
- Height of the tree:  $O(\log n)$
- Query cost?  $O(n^{1/2}+k)$ , where  $k$  is the output size.



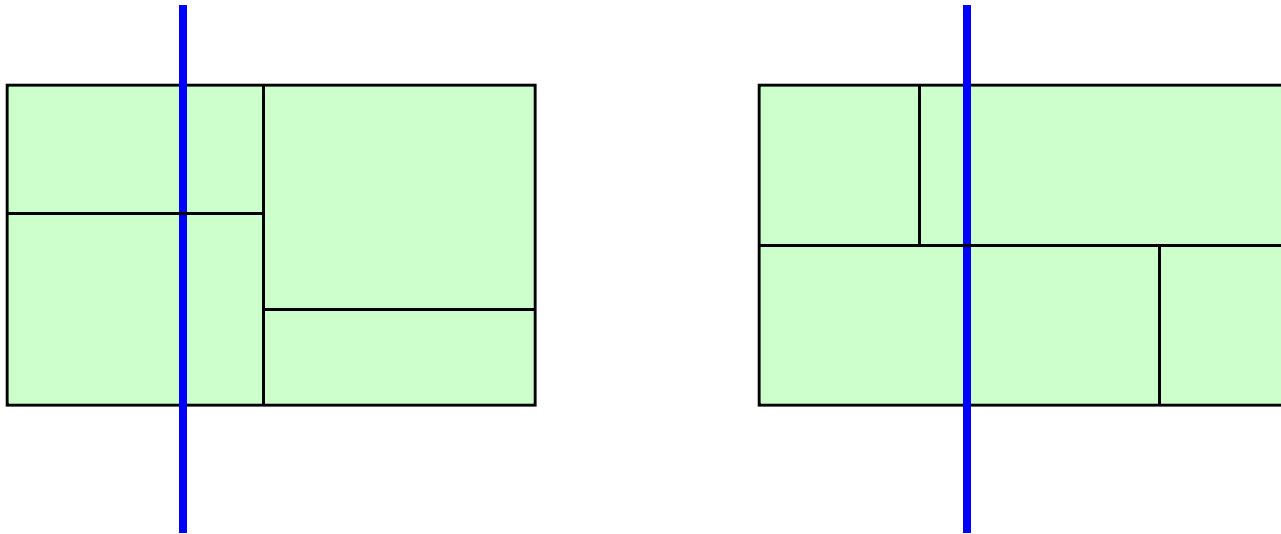
# Kd-tree

- Query cost?  $O(n^{1/2}+k)$ , where  $k$  is the output size.
- Intuition: we visit 2 types of nodes:
  - $r(v)$  is fully contained in  $R$  (this is counted in  $k$ ).
  - $r(v)$  is not fully contained in  $R$  – intersected by boundaries of  $R$ .
- Thus we bound the number of nodes intersected by a vertical line, denoted by  $Q(n)$ .



# Kd-tree

- Thus we bound the number of nodes intersected by a vertical line, denoted by  $Q(n)$ .
- Look at the 4 grandchildren, the line intersects at most 2 of them.
- Thus  $Q(n) = 2Q(n/4) + O(1) = O(n^{1/2})$ .
- The query cost is  $O(k) + 4Q(n) = O(n^{1/2} + k)$ .



## Kd-tree in $\mathbb{R}^d$

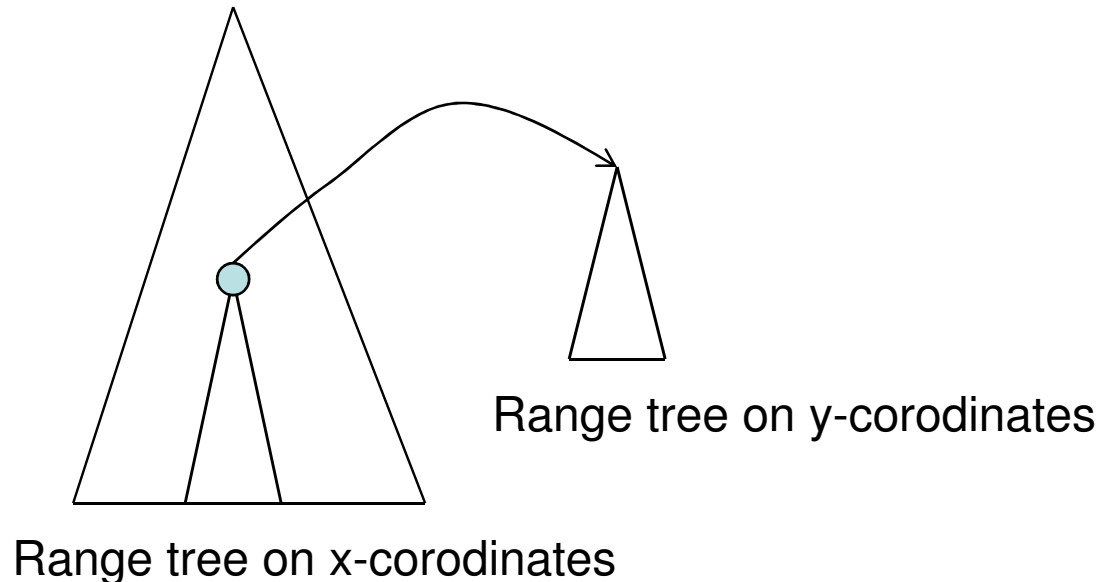
---

- High dimensional kd-tree.
- If the dimension is  $d$ , we can build a kd-tree with  $O(n)$  size, and query cost  $O(n^{1-1/d}+k)$ , where  $k$  is the output size.
- Query cost is too high.
- We can get it down if we sacrifice on space.
- Range tree:  $O(n \log^{d-1} n)$  space and  $O(\log^d n + k)$  query cost.

# Range tree

---

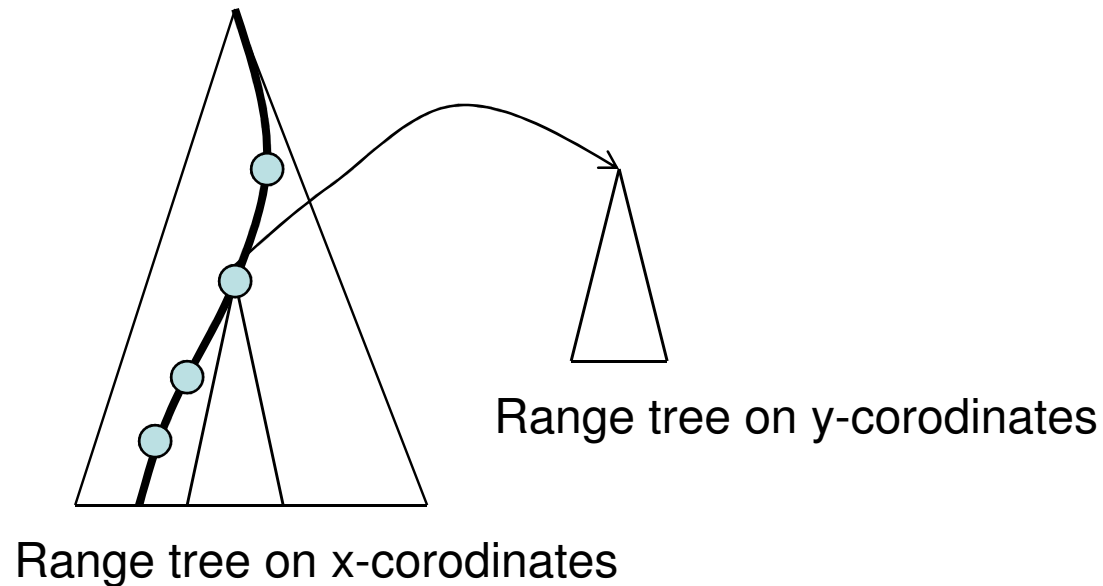
- Recall the 1d range tree.
- 2D range tree:
  - First build a 1D range tree on x-coordinates
  - For each internal node, take all the nodes in its subtree, build a 1D range tree on y-coordinates.
- Total space:  $O(n \log n)$



# Range tree

---

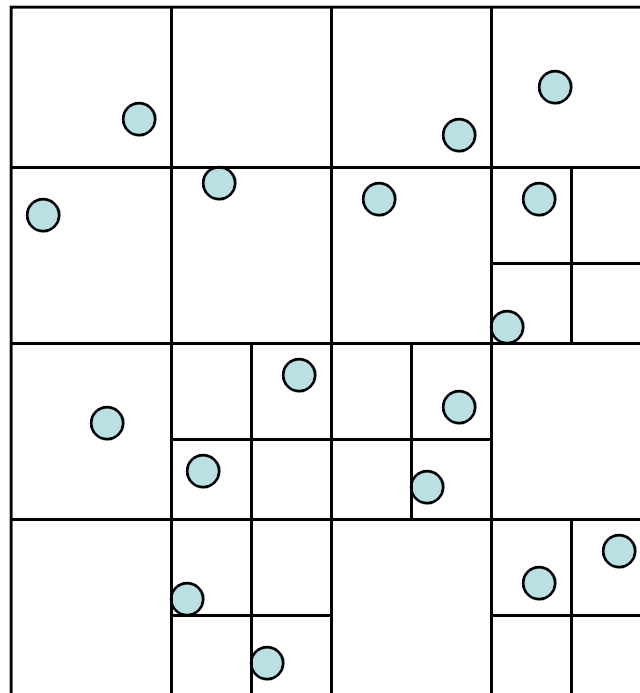
- Query:
  - First search the 1D range tree on the x-coordinates
  - For each node on the traversal path, search on the y-coordinates.
- Query cost:  $O(\log^2 n + k)$



# Quad-tree

---

- A recursive space partitioning tree.
- The depth might be as high as  $\Omega(n)$ .
- Worst-case query cost is not bounded. For uniform sensor distribution the depth is  $O(\log n)$ .



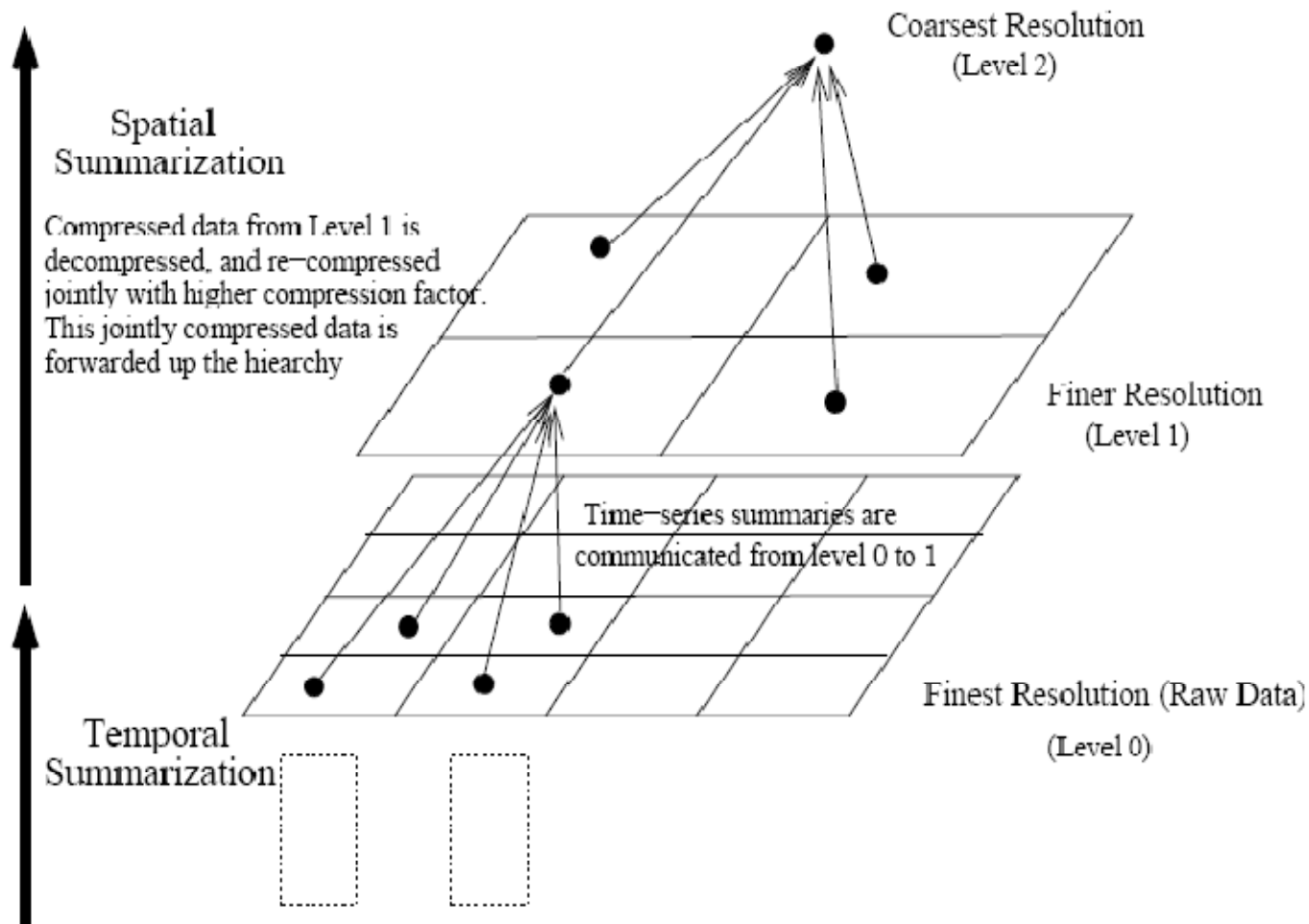
# Indexing in a sensor network?

---

- Where is the index stored?
- How to traverse the tree?
- 1<sup>st</sup> approach: map a quad-tree to the sensor field.
- 2<sup>nd</sup> approach: distributed storage and indexing.

# DIMENSIONS: summaries

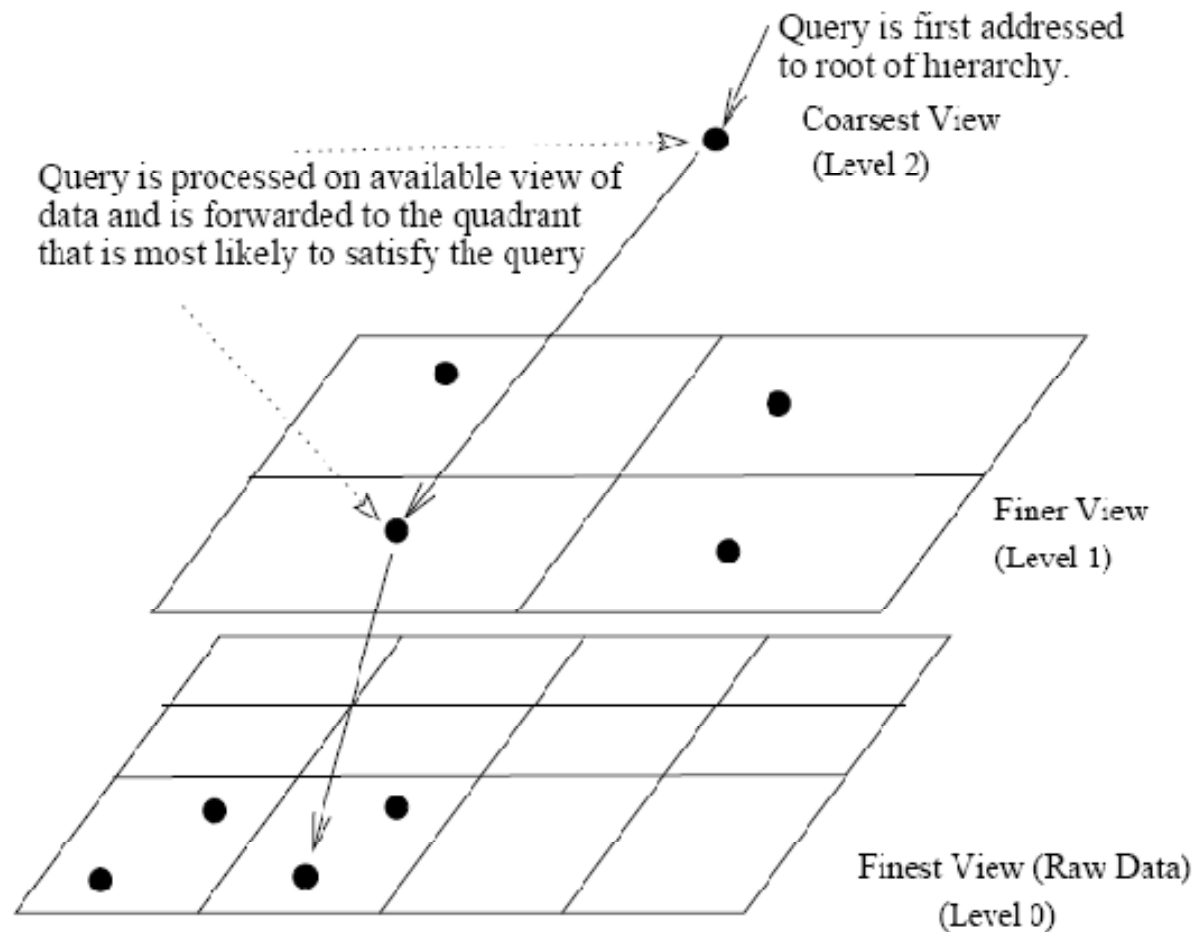
- Use a quad-tree partitioning.



# DIMENSIONS: query

---

- Top-down query processing



# Issues with DIMENSIONs

---

- Uneven load: nodes holding coarse data are visited more often.
- Root becomes traffic bottleneck.

# Distributed index for multi-dimensional data

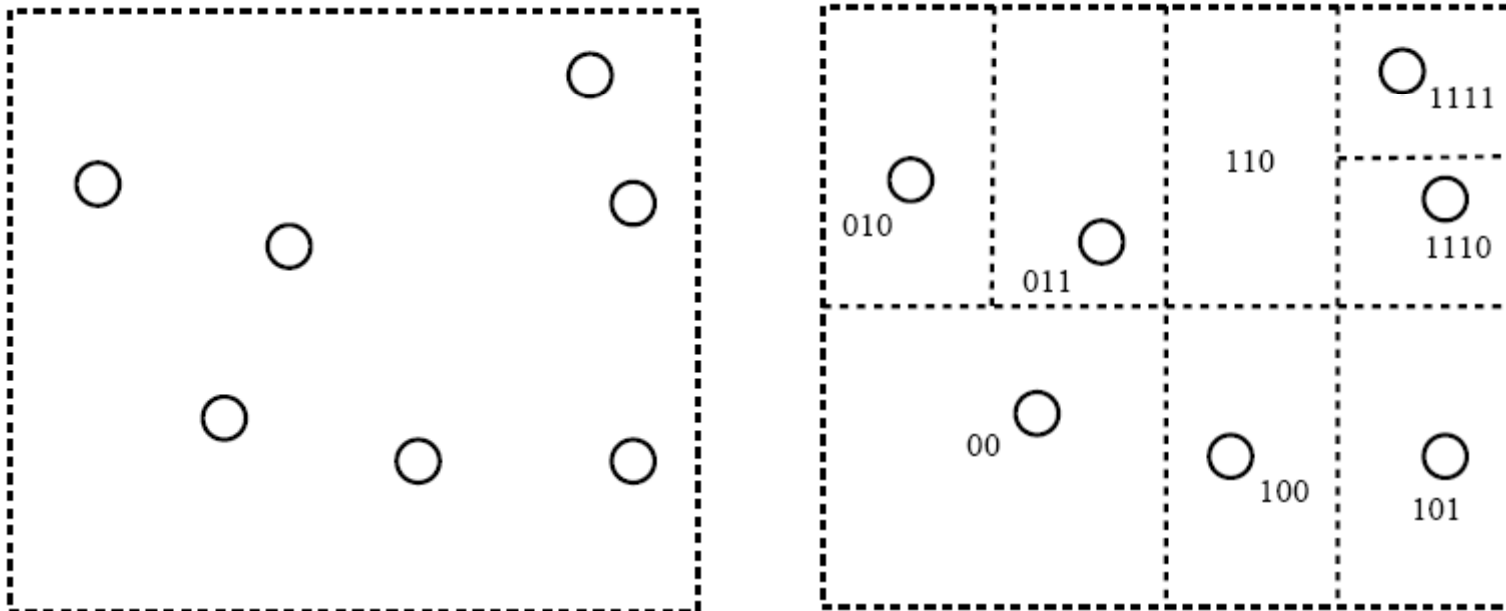
---

- Construct the distributed indices.
- Locality preserving geographic hash: events with close attributes values are likely to be stored close.
- Kd-tree partitioning.

# Zones

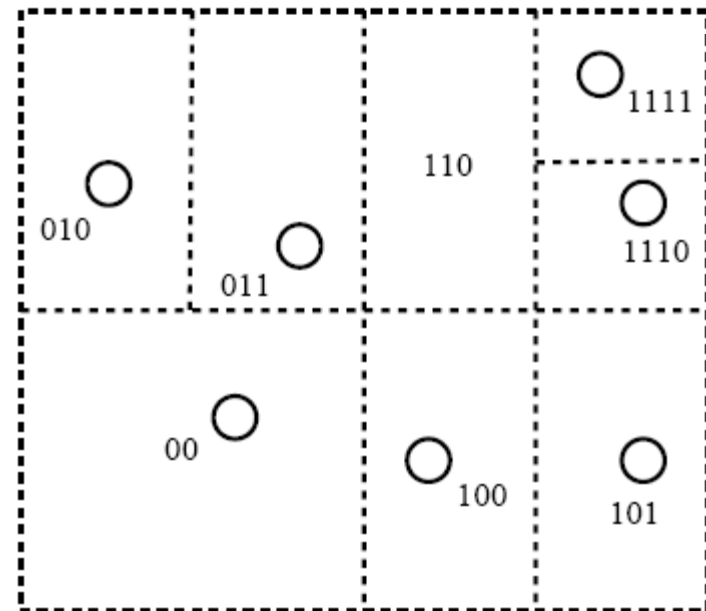
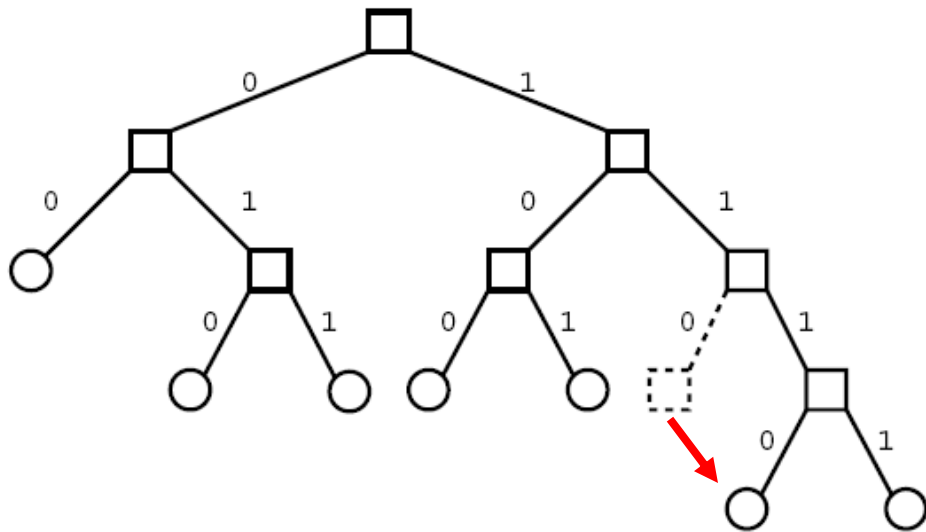
---

- The sensor network is partitioned to equal (geographical) size regions along x and y directions alternatively.
- Each cell is given a zone code – left (bottom) is 0, right (top) is 1.



# Zone-tree

- Each node  $x$  owns a zone – the largest one that contains  $x$  only.
- If a zone is empty, it is owned by the backup node – the rightmost zone in the left sibling tree, or the leftmost zone in the right sibling tree.



# Data-centric hashing

- Hash a multi-dimensional event to a zone.
- A multi-dimensional event  $\{A_i\}$ ,  $i=1, \dots, m$ ,  $A_i \in [0, 1]$ .
- Suppose the zone code has  $k$  bits,  $k$  is a multiple of  $m$ .
- For  $i=1$  to  $m$ , if  $A_i < 0.5$ , the  $i$ -th bit is assigned 0, otherwise 1.
- For  $i=m+1$  to  $2m$ , if  $A_{i-m} < 0.25$  or  $0.5 \leq A_{i-m} < 0.75$ , the  $i$ -th bit is assigned 0, otherwise 1.

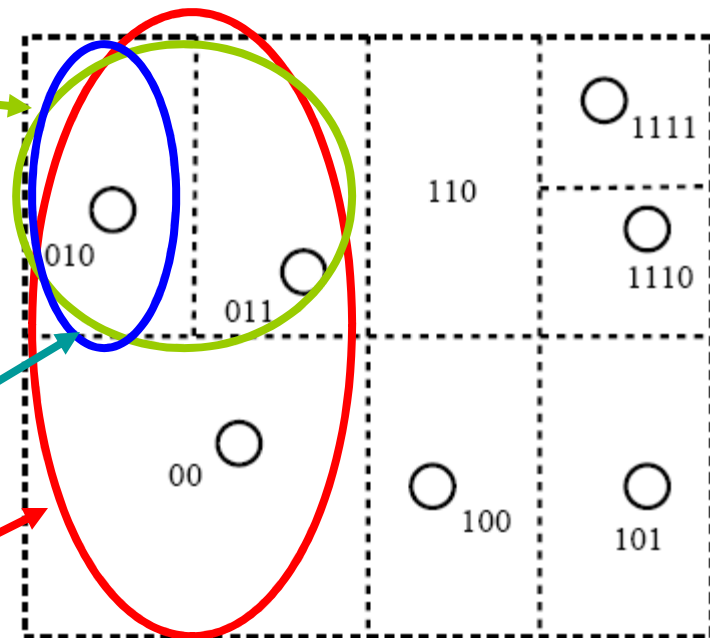
$$A_1 < 0.5, A_2 < 0.5$$

For example:  $[0.3, 0.8]$  is stored at 5-bit zone code 01110.

The event is hashed to the node that owns the zone.

$$A_1 < 0.25 \text{ or } 0.5 \leq A_1 < 0.75, A_2 < 0.5$$

$$A_1 < 0.5$$



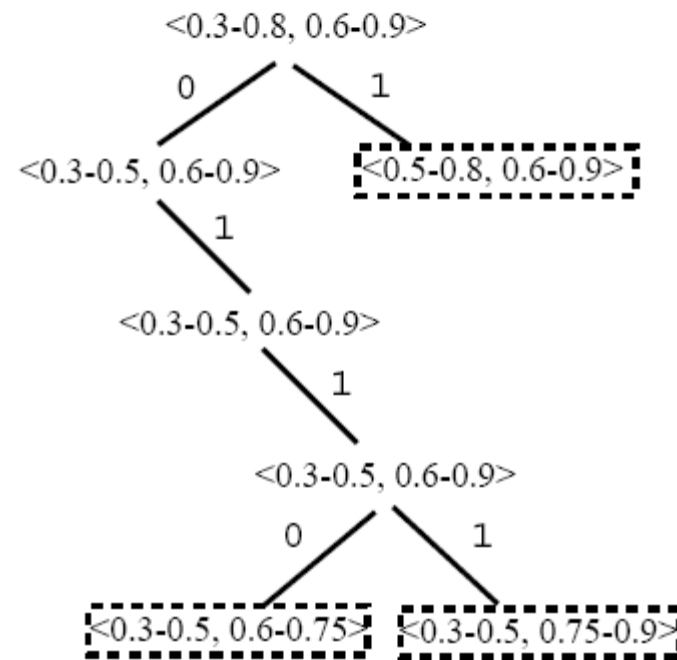
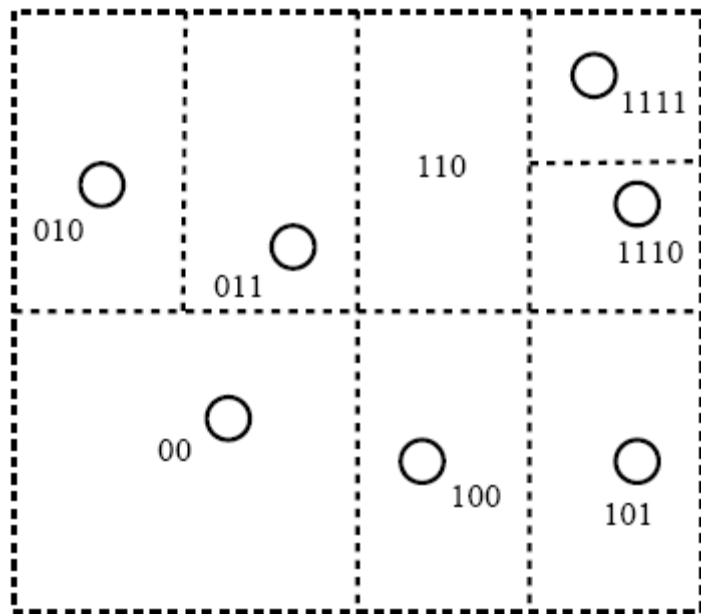
## Data-centric routing

---

- The encoding node (where the event  $E$  is generated) may not know the # bits of the hashed zone.
- Node  $A$  encodes the node by using the length of its own code and generates the zone code  $c(E)$ .
- Node  $A$  routes by GPSR to the centroid of the zone  $c(E)$ .
- Intermediate nodes may refine code  $c(E)$ .
- If the current node  $B$  finds a match of its own code and the event code  $c(E)$ , then  $B$  stores the event.

# Routing queries

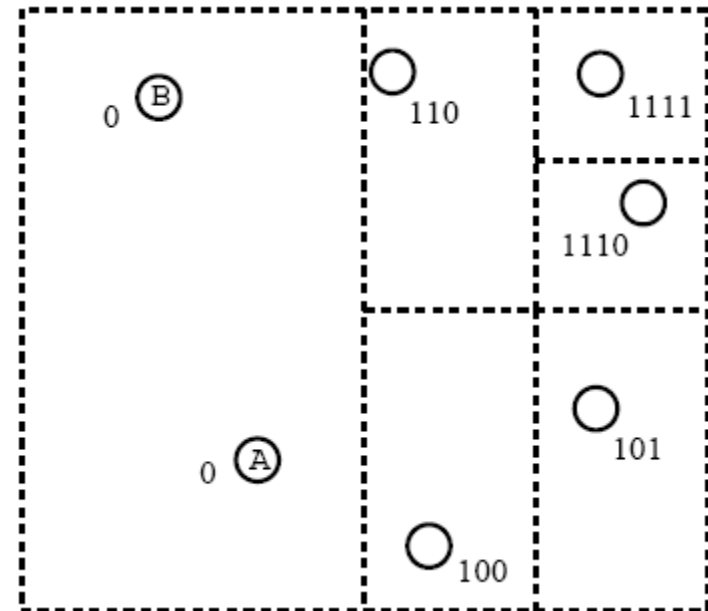
- Looking for a point event is the same as routing an event.
- A range query is routed to a zone corresponding to the entire range, and then progressively split into smaller sub-queries.



# Event routing helps resolving undecided zones

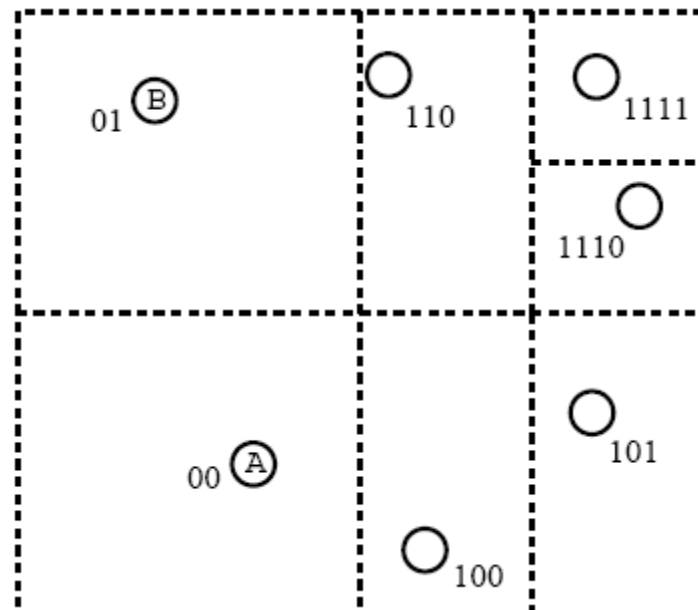
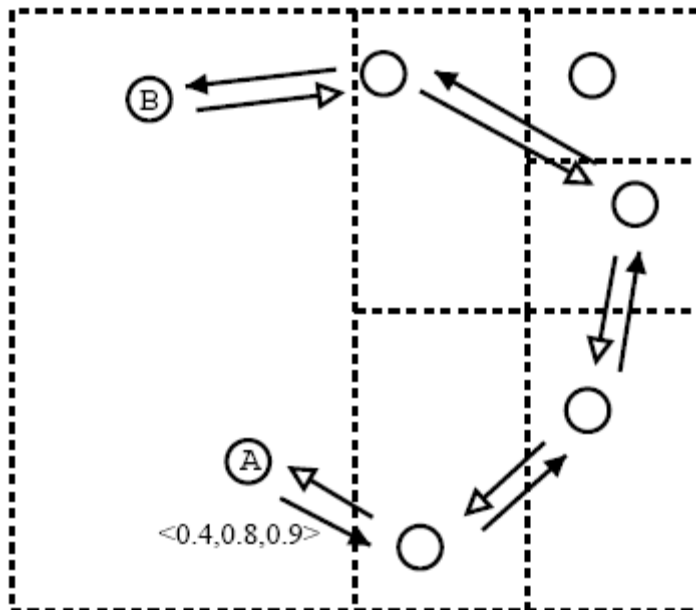
---

- How does each node know its own zone code?
- Assume that every node knows the outer boundary.
- A node checks its 1-hop neighbors and decides on the largest zone that only contains itself.
- This may not fully resolve all the boundaries.



# Event routing helps resolving undecided zones

- A claims the ownership of event E.
- But A is not sure of its upper boundary. So A sends out the event E by GPSR (face routing) with a destination near A.
- Node B that receives this message shrink its zone.



## DIM summary

---

- Data storage explores query locality. Range query can be supported.
- Events are not necessarily stored close to where they are generated.
- Each event costs about  $O(n^{1/2})$  communication cost.
- When data is highly skewed, most data are handled by a small number of sensors which become bottleneck.

# Major problem: data storage

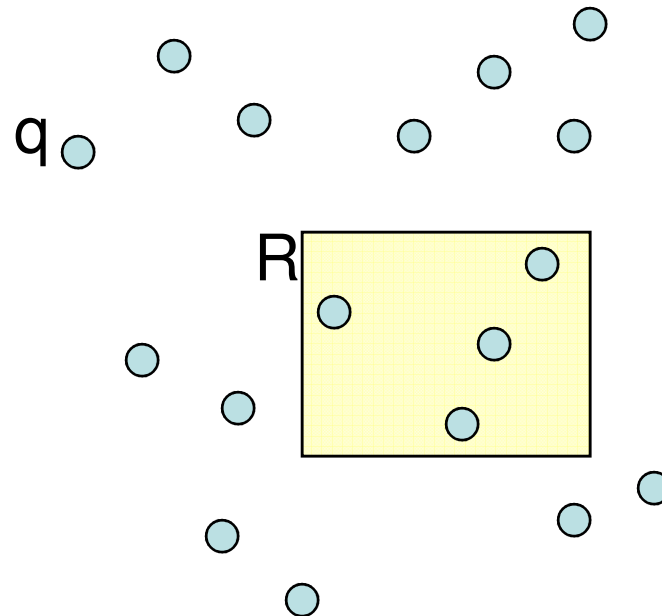
---

- Similar data (in attribute space) should be stored close.
- Data should be stored close to where they were generated. --- location is an important attribute of the data.
- The two considerations may be in conflict.

# Fractional cascading in sensor network

---

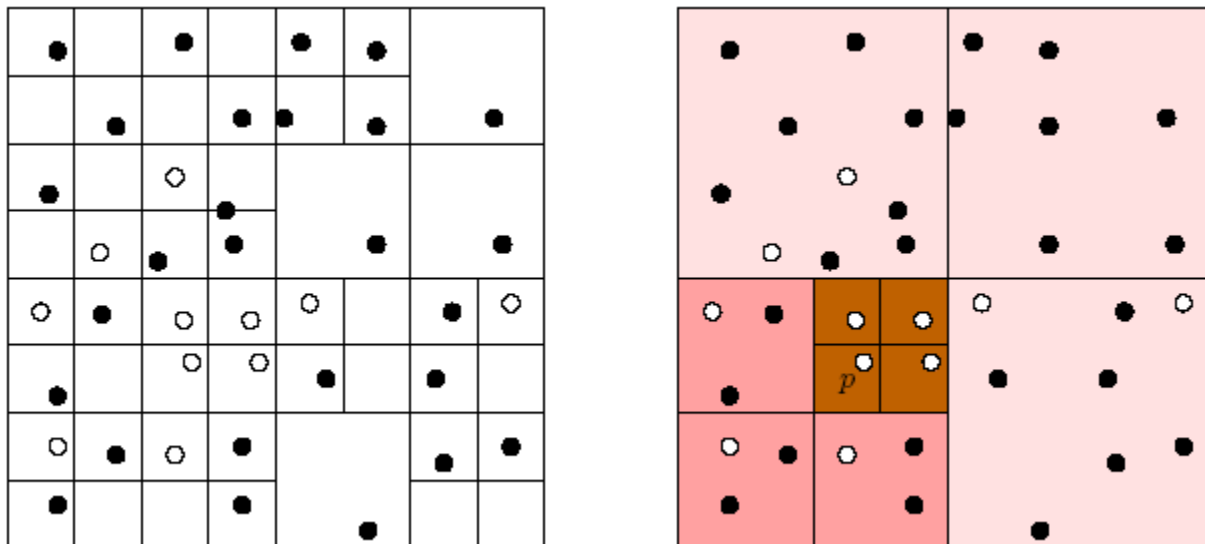
- Geographical range query  $(q, R, T)$ :  $q$  is where the query is generated,  $R$  is the rectangular range,  $T$  is a temperature range or other aggregates.
- Aggregates about region  $R$  should be returned to query node.



# Storage scheme

---

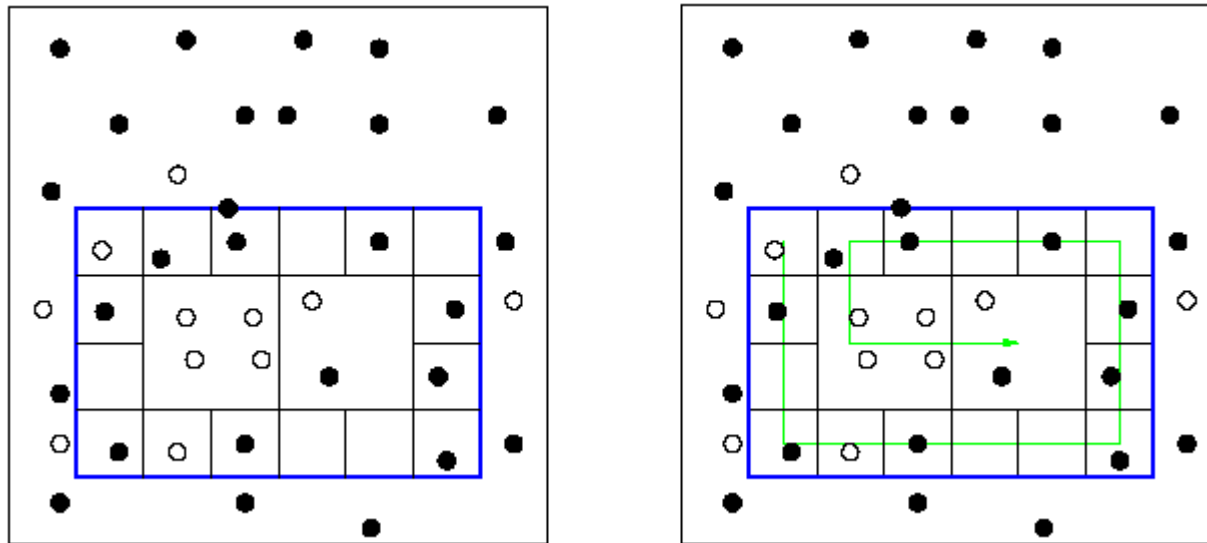
- The aggregated value of a quad node is stored in all the sensors in the parent subtree.
- Each node stores  $O(\log n)$  data.
- Construction: bottom up. Cost  $O(n \log n)$ .



# Query scheme

---

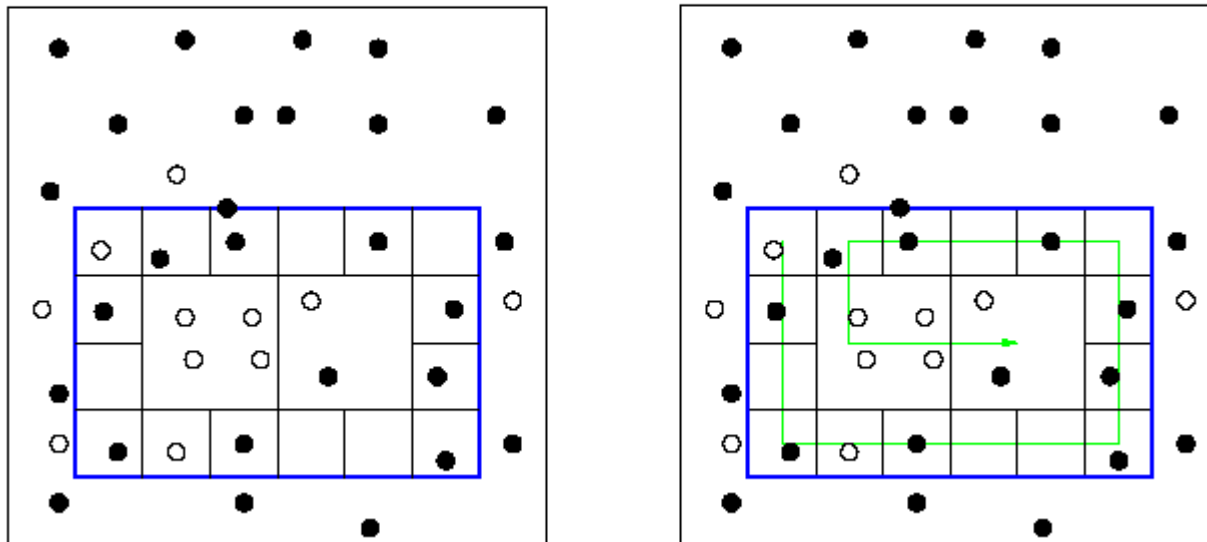
- The query region  $R$  is partitioned into canonical regions – the maximal quads completely inside  $R$ .
- Use a spiral routing to visit a sensor in each canonical regions.
- Recurse on each canonical piece.



# Query cost

---

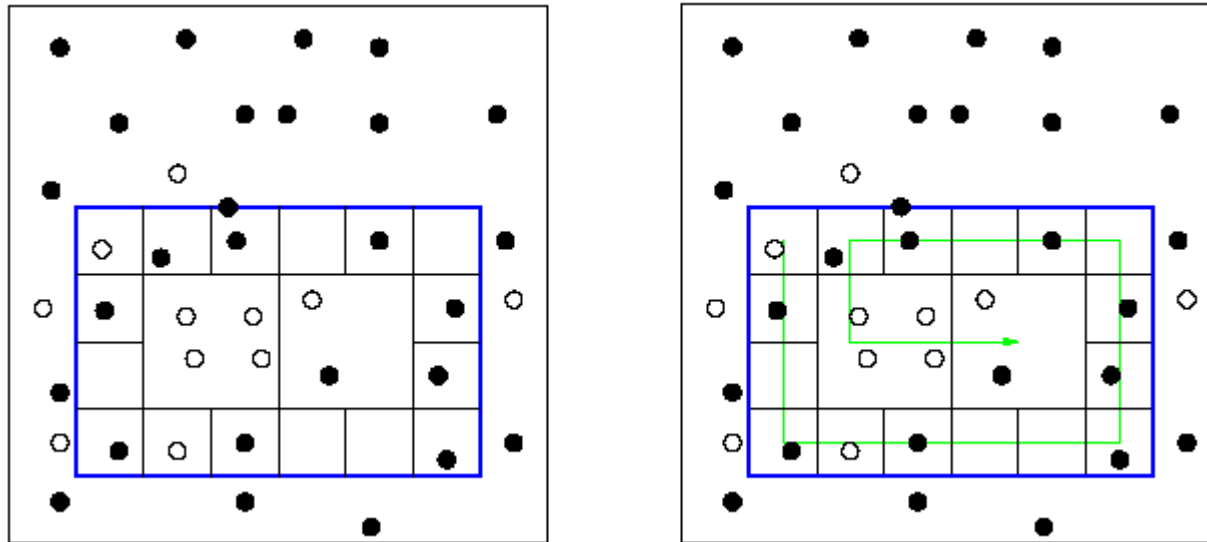
- The query cost for  $(q, R, [T, \infty))$  is  $O(D + \sqrt{Ak} + P \log P)$
- $A$  is the area,  $P$  is the perimeter,  $k$  is the output size.
- Cost 1: spiral visit:  $O(P \log P)$



# Query cost

- Cost 2: the communication cost of recursion in each canonical piece with side length  $L(u)$  and output  $k(u)$  is  $O(L(u)\sqrt{k(u)})$
- The total recursion cost is  $\sum_{u \in C} L(u)\sqrt{k(u)}$ ,

$$\sum_{u \in C} L(u)\sqrt{k(u)} \leq \sqrt{\sum_{u \in C} L(u)^2 \sum_{u \in C} (\sqrt{k(u)})^2} = \sqrt{Ak}$$



# Summary

---

- Store similar data close
  - Work in the space of the data field
  - Bring all similar data together
  - May need to travel far
- Store data nearby
  - Respect space locality for geographical range query.
  - Communication cost is low.
  - Range search in data space is challenging.
- Can you get the best of both worlds?

# The remaining classes

---

- Network boundary detection
- Coding theory with applications in routing and storage.
- Sensor selection.
- Synchronization.
- Gossip algorithms.
- Percolation theory and connectivity.
- Reminder on class project: you can email me for any questions/ideas and I'll try to help.

# Agenda

---

- Thursday lecture by Rik Sarkar on boundary detection algorithms
- Next week: spring break, no class.
- April 14<sup>th</sup>, invited speaker in CS2311.
- April 16<sup>th</sup>, lecture
- April 21<sup>st</sup>, student presentation: Nikhil Joshi and Michele Albano