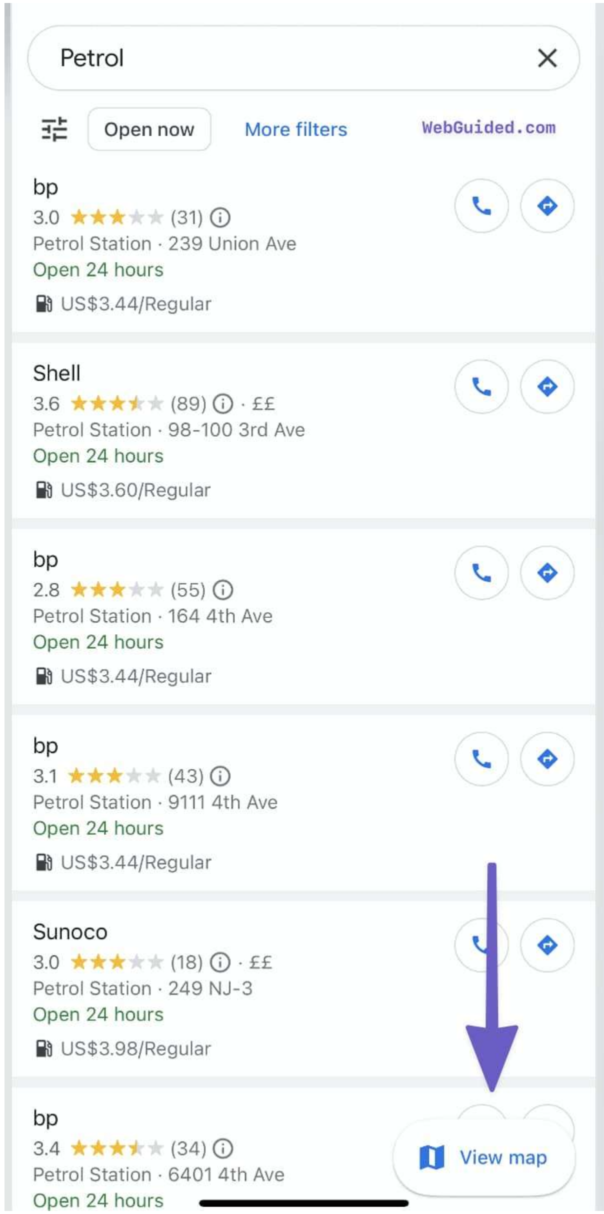
Database Systems
Lecture23 – Multi-Dimensional Index & Vector Database

Beomseok Nam (남범석)
bnam@skku.edu

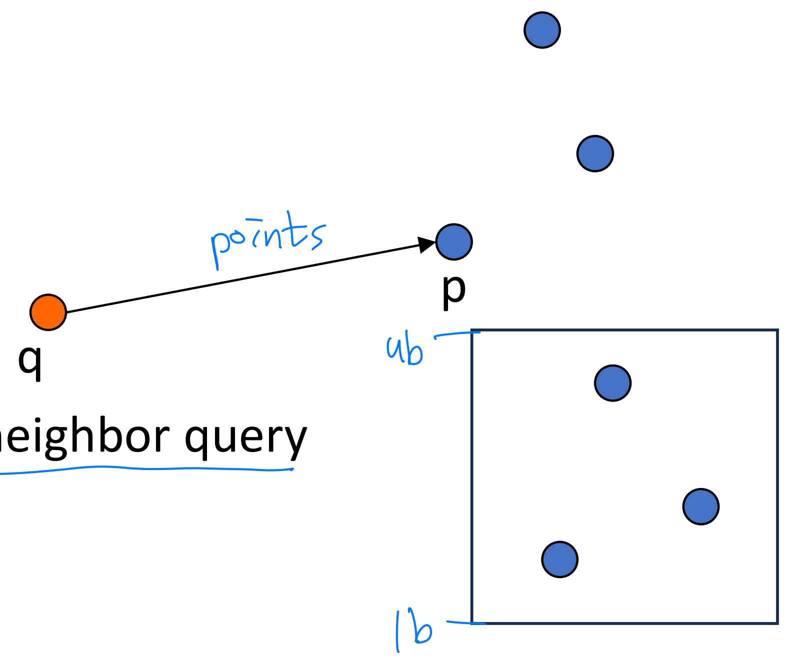
Finding the closest gas station near me



Spatial Data

multi dimension Data

- Data types such as points, lines, and polygons
- **Nearest neighbor queries**, given a point or an object, find the nearest object that satisfies given conditions.
- **Range queries** deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region.



Nearest neighbor query

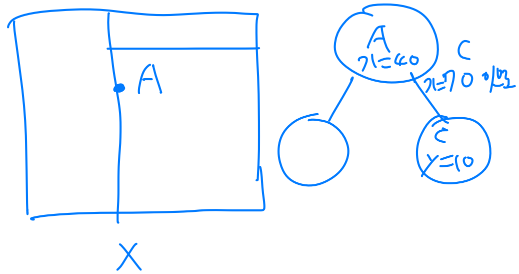
Range query

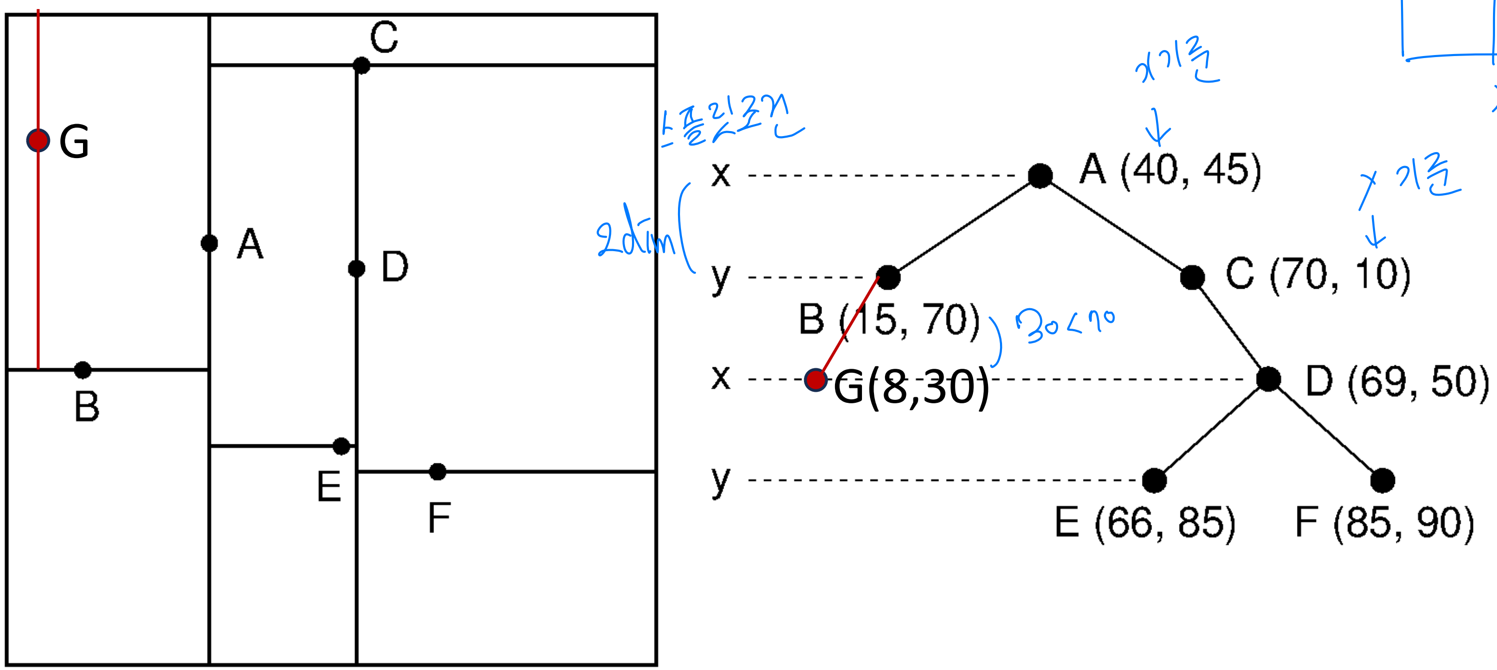
○ 등의 다양한 모양

K-D trees: Space-partitioning Method

BST의 확장버전

- Each level of a K-D tree partitions the space into two.
  - In each node, choose one dimension for partitioning, cycling through the dimensions.



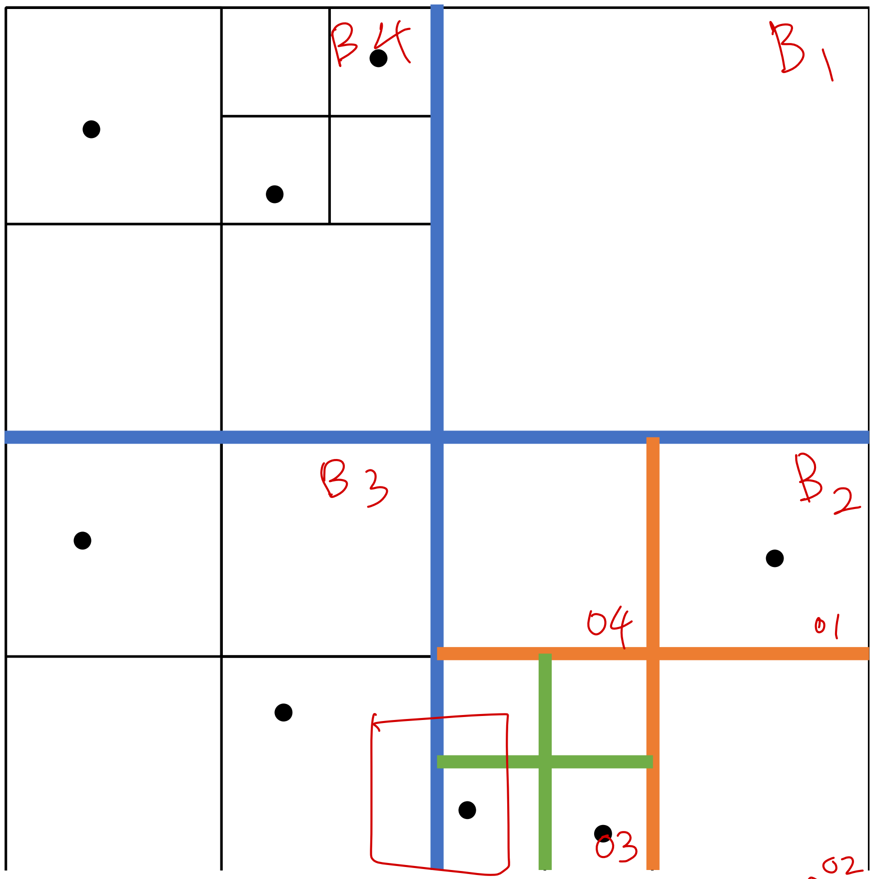


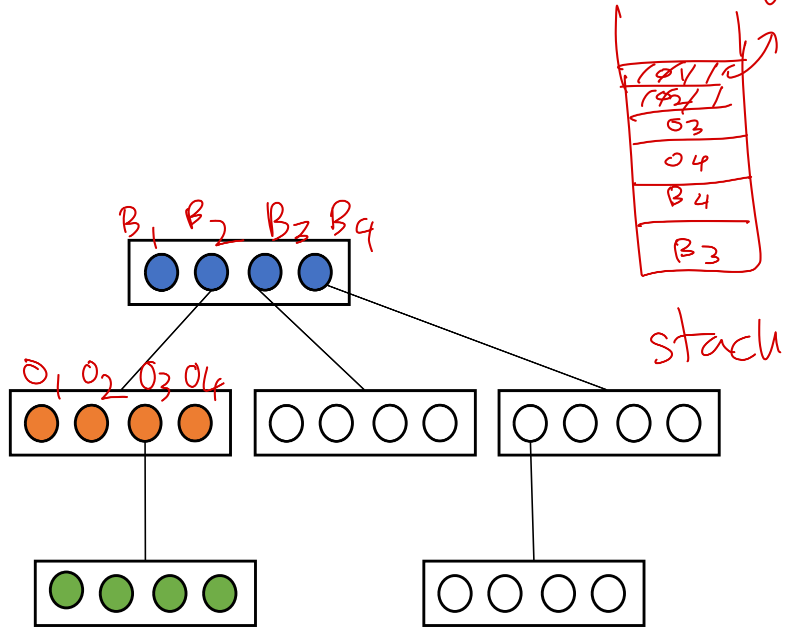
(a) (b)

Quadtrees (space partitioning method)

4개로 나누는 것

- The root node represents the entire target space.
- Each non-leaf node divides its region into four equal sized quadrants





범위가 overlap되지 않으면 pop

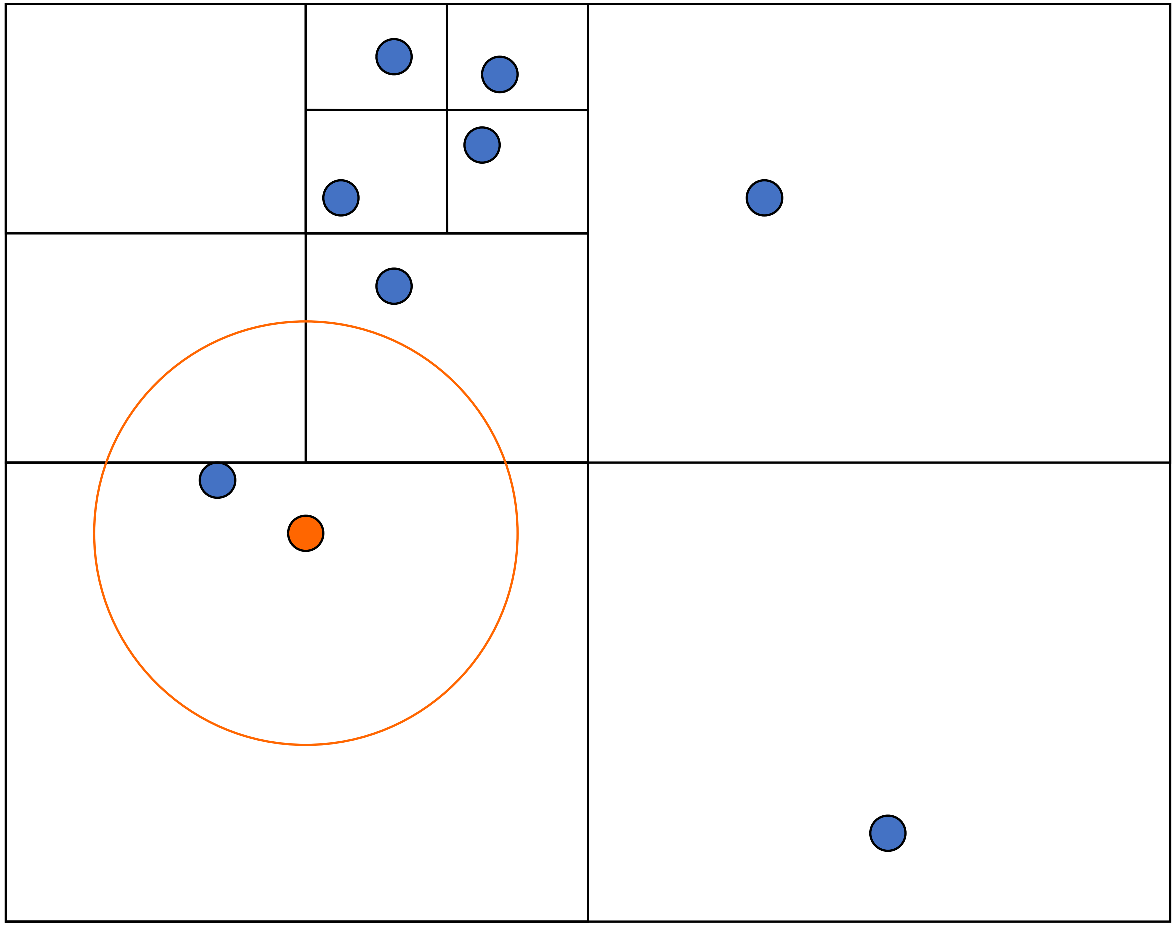
Range query → 이게 하나의 영역에만 있는 것이 아니라 backtracking이 필요!

Range search

- Near neighbor (range search):
  - put the root on the stack backtracking을 위해
  - repeat
    - pop the next node *T* from the stack
    - for each child *C* of *T*:
      - if *C* is a leaf, examine point(s) in *C*
      - if *C* intersects with the ball of radius *r* around *q*, add *C* to the stack

범위가 원하는 값과 overlap 된다면

Nearest neighbor

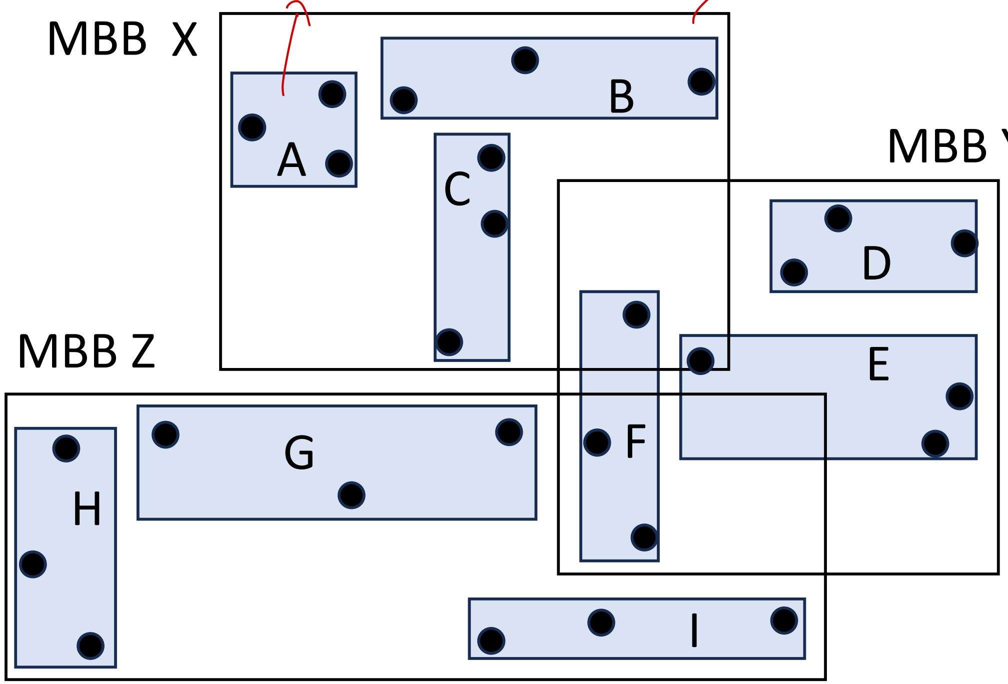


R-trees: Data-partitioning Method

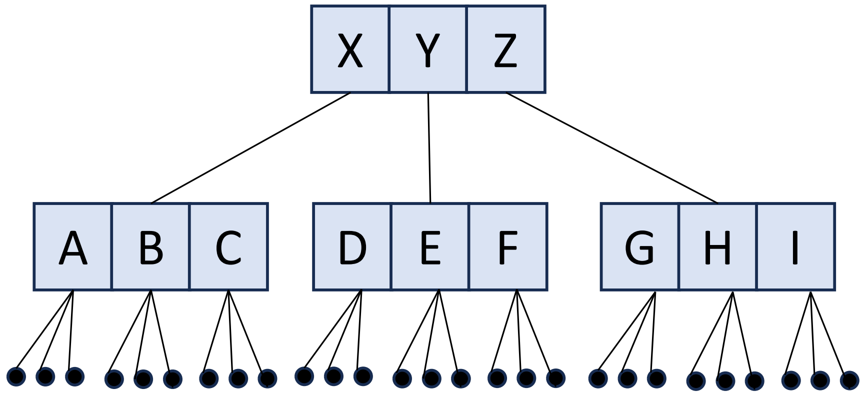
- N-dimensional extension of B^+-trees
- The **bounding box** of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node
  - *Bounding boxes of children of a node are allowed to overlap*

가까운 것끼리

another minimum bounding box



B^+ tree 같이

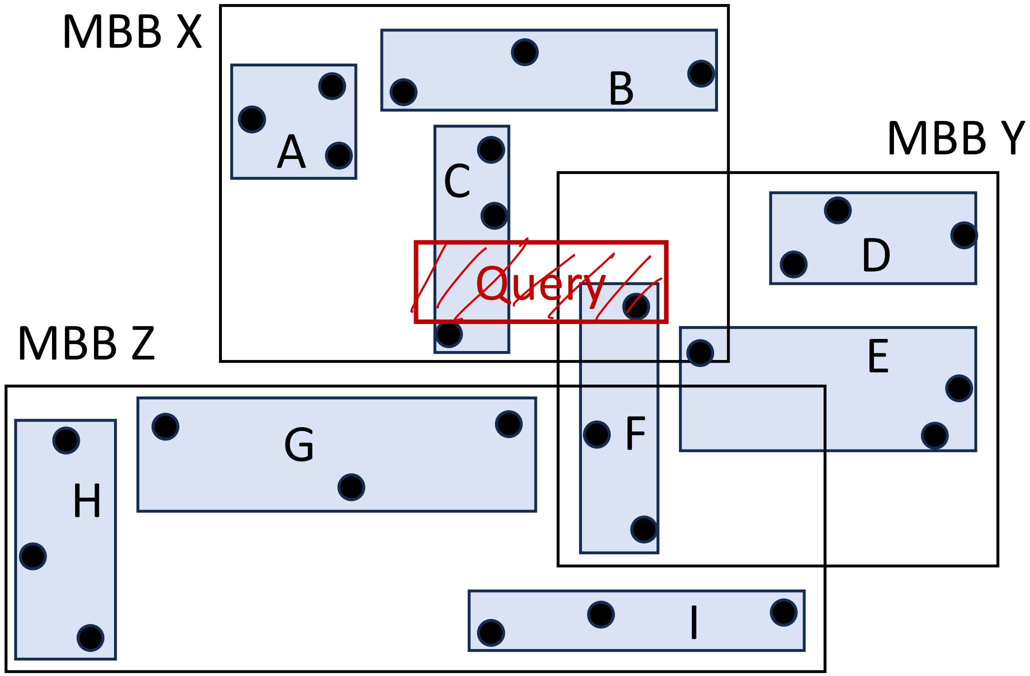


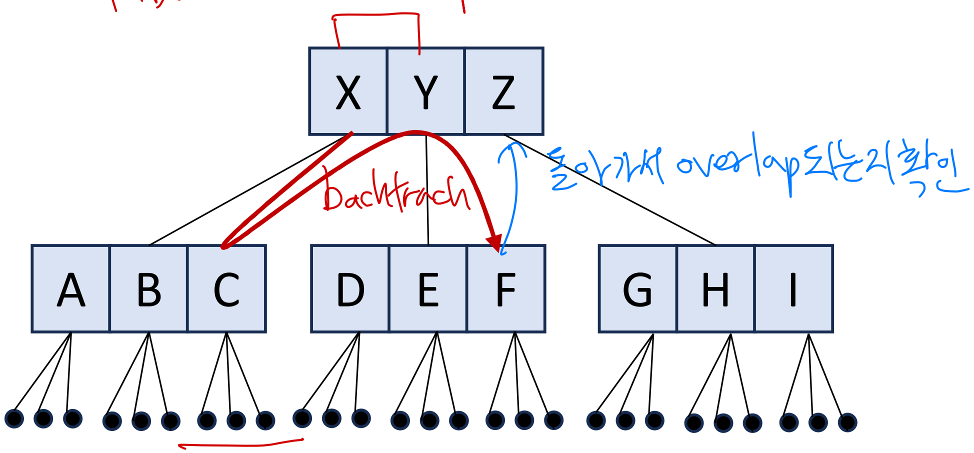
R-trees: Data-partitioning Method

- N-dimensional extension of B+-trees
- The **bounding box** of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node
  - *Bounding boxes of children of a node are allowed to overlap*
  - *Range Query → Visit all overlapping child nodes → Backtracking*

space partition이 아

MBB끼리 겹쳐 overlap된다 → Data partition!



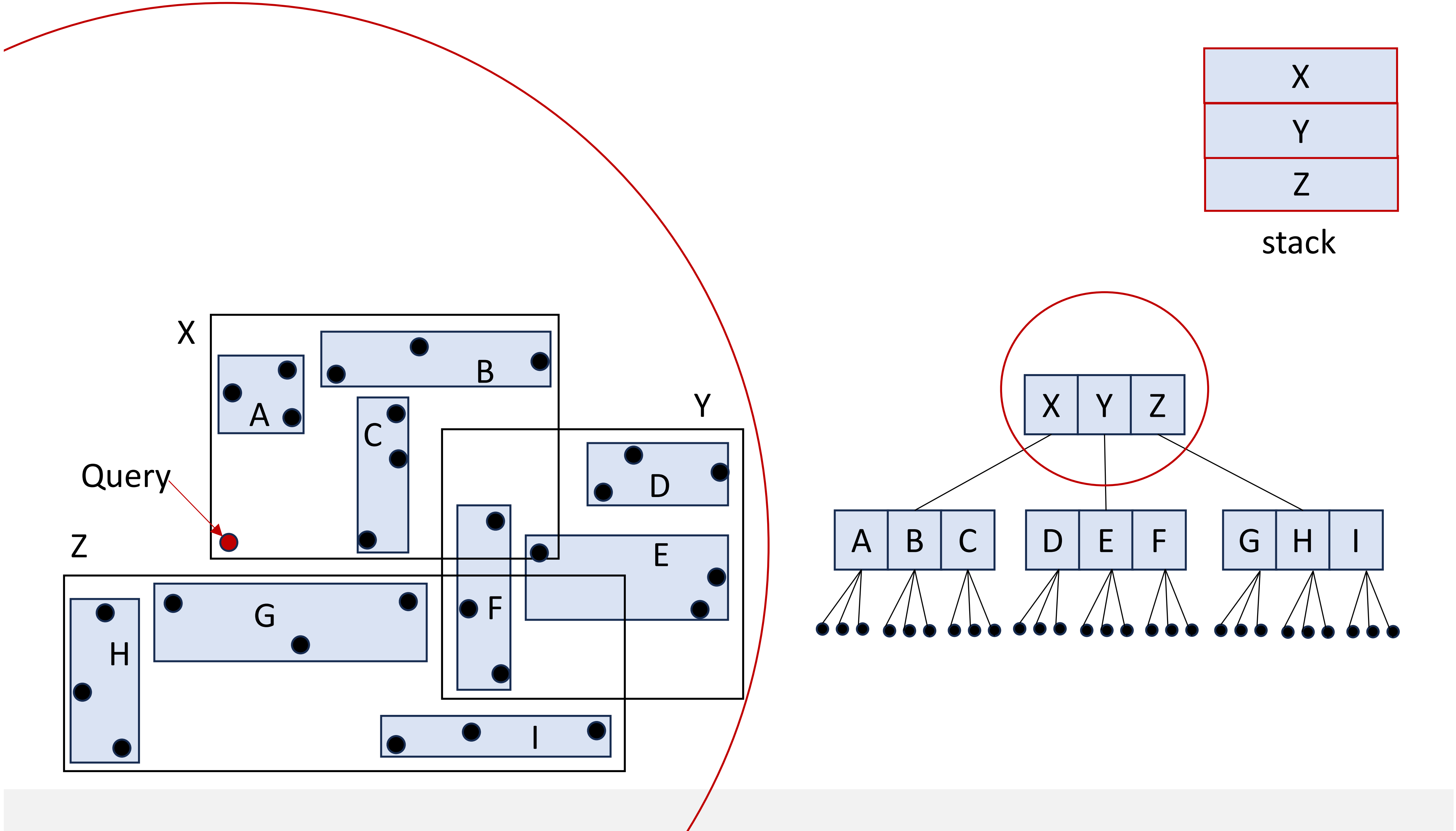


Nearest Neighbor Query (NN-Query)

- Start range search with $r = \infty$
  - Or, guess a range r that contains at least one object say O
    - if the current guess does not include any object, increase range size until an object found.
- put the root on the stack
- Repeat
  - pop the next node T from the stack
  - for each child C of T:
    - if C is a leaf, examine object(s) in C
    - Whenever an object with smaller distance is found, update r; Only investigate nodes with respect to current r
    - if C intersects with the ball of radius r around q, add C to the stack

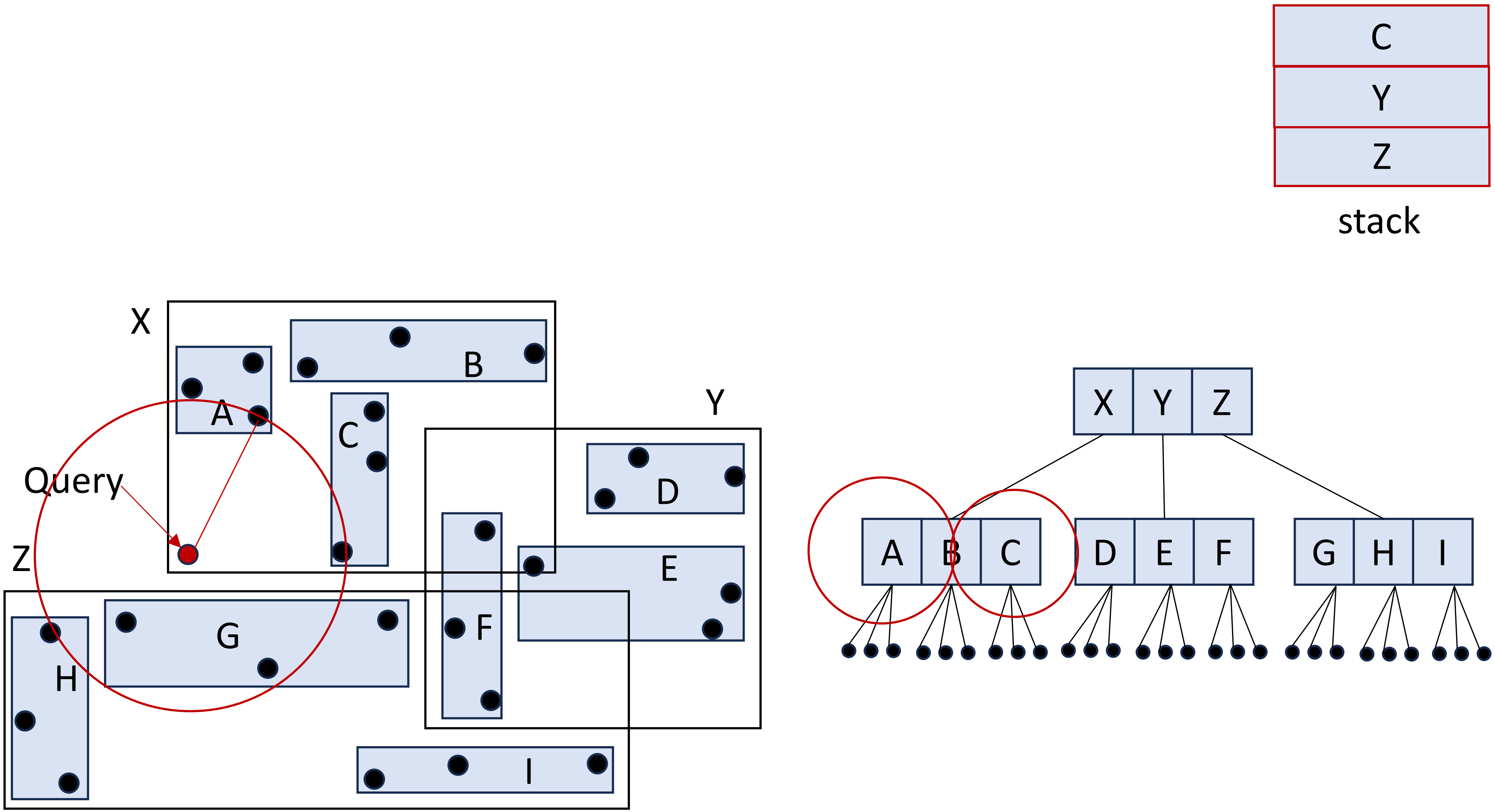
R-trees: Data-partitioning Method

- Example: NN-Query Processing with R-tree



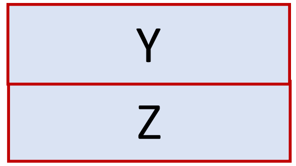
R-trees: Data-partitioning Method

- Example: NN-Query Processing with R-tree

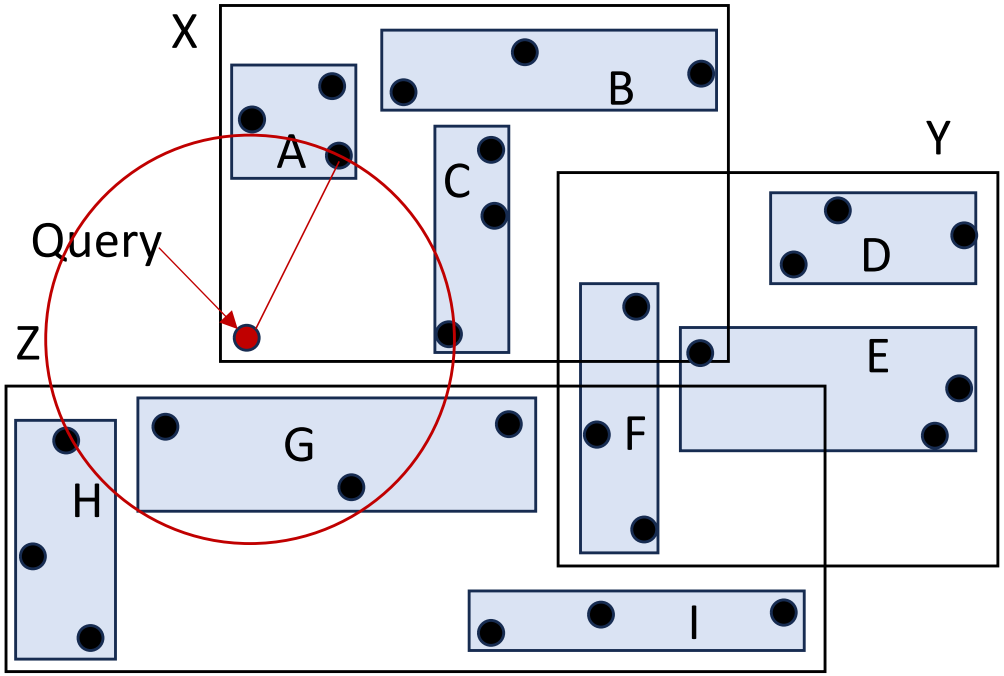


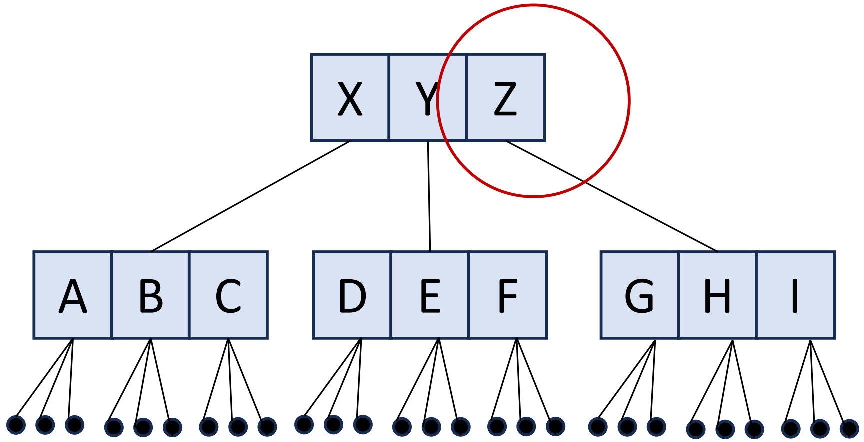
R-trees: Data-partitioning Method

- Example: NN-Query Processing with R-tree



stack



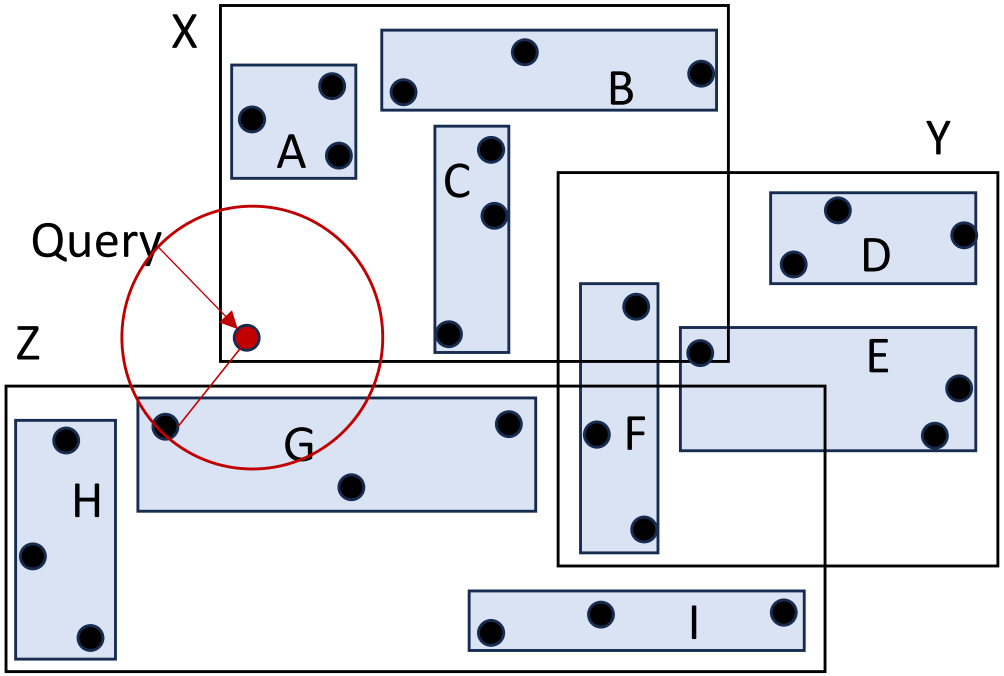


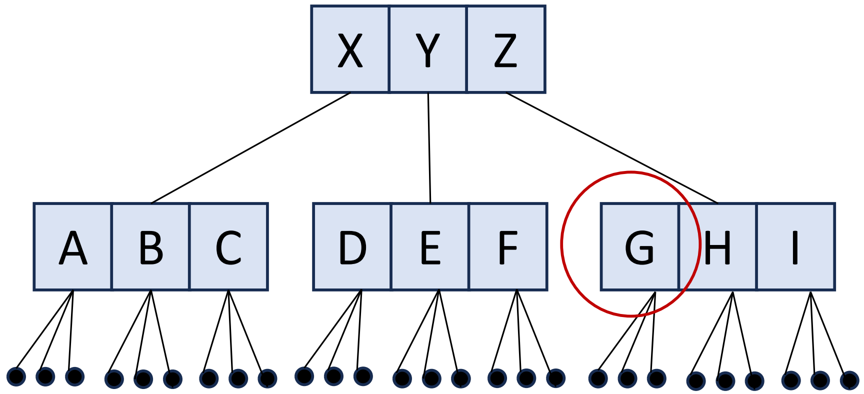
R-trees: Data-partitioning Method

- Example: NN-Query Processing with R-tree

H
Z

stack



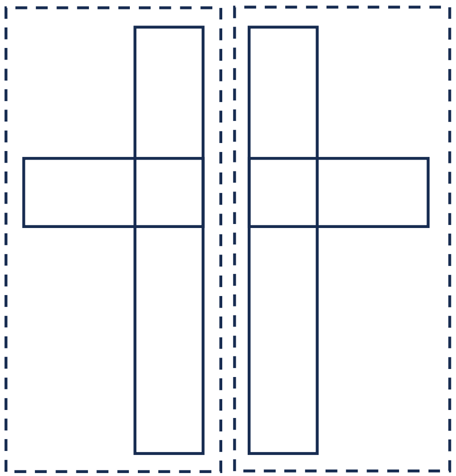


Insert

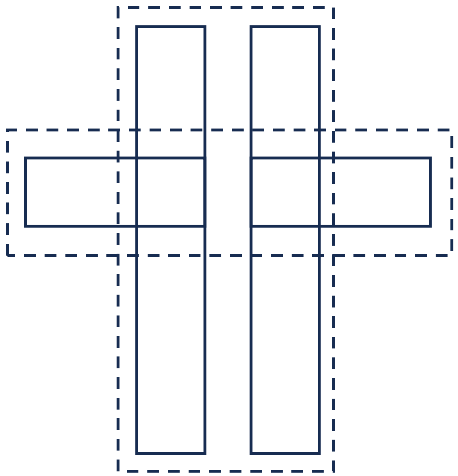
```
RTree-Insert(T, entry)
 leaf ← ChooseLeaf(T.root, entry) // locate place to insert
 Insert entry into leaf
 IF leaf overflows THEN
     newNode ← SplitNode(leaf)
     AdjustTree(leaf, newNode) // propagate changes upward
 ELSE
     AdjustTree(leaf, NIL) // propagate MBB changes upward
 IF root was split THEN
     create new root with children = previous root and newNode
```

Split

- How to partition the M+1 MBBs into two nodes?
  - 1. The total area of the two nodes is minimized
    - Large dead space hurts search performance
  - 2. The overlapping of the two nodes is minimized
    - Overlap causes backtracking and hurts performance
- Sometimes the two goals are conflicting



No overlap, large MBB



Overlap, small MBB

Split

- Optimal solution: check every possible partition, complexity $O(2^{M+1})$

- A quadratic algorithm:
  - Pick two “seed” entries e1 and e2 far from each other, that is to maximize area(MBB(e1,e2)) – area(e1) – area(e2)
    - Here MBB(e1,e2) is the *minimum bounding box* containing both e1 and e2
    - complexity = $O((M+1)^2)$
  - Insert the remaining (M-1) entries into the two groups
    - Continued on the next slide

Split

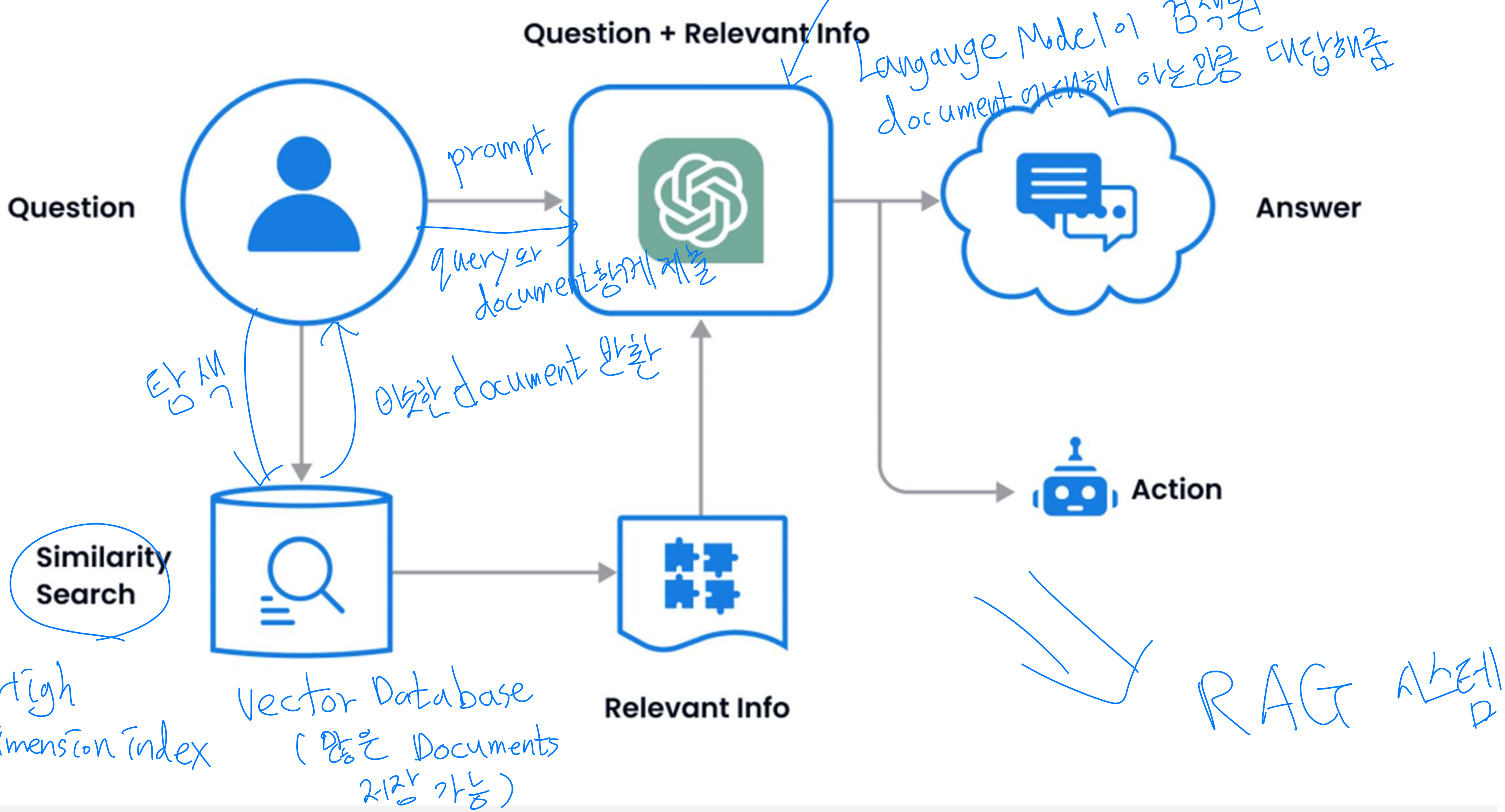
- A greedy method
  - At each step, pick an unassigned entry and assign it to one of the two groups based on:
    - Minimum area enlargement caused by adding the entry
    - If tied:
      - Select the group with smaller area
    - If still tied:
      - Select the group with fewer elements
  - Loop Until...
    - All entries are assigned, or
    - One group reaches (M - m + 1) entries
      → All remaining entries go to the other group
  - If the parent is also full, split the parent as well.

LLM and Vector DB (Vector Store)

- **LLMs are context-limited** → Need external knowledge
- Vector DBs help retrieve **relevant documents** based on **semantic similarity**
- A Vector Store is a specialized database that stores and retrieves data using **vector embeddings** — numerical representations of text, images, or other unstructured data.
  - Enables **semantic search** (meaning-based, not exact keyword match)
  - Supports **context retrieval** in Retrieval-Augmented Generation (**RAG**)

LLM and Vector DB (Vector Store)

- Text → Embedding → Vector Store → Similar Documents → LLM



High-Dimensional Nearest Neighbor Search

- Example application: Reverse image search
  - Represent image by a vector
  - Pixel values arranged in a vector
  - More advanced features (SIFT, SURF, ORB)
  - Similar vectors ↔ similar images

High-Dimensional Vectors

- 100-1000 dimensions
- **Curse of dimensionality**
  - Many methods scale poorly as the dimension increases
  - Considering one coordinate at a time is no longer enough

Vector Indexes

Index Type	Description	Pros	Cons	Example Libraries
Brute Force (Flat)	Compares query against all vectors	100% accuracy	Very slow for large data	FAISS (IndexFlatL2)
IVF (Inverted File Index)	Clusters vectors, searches in relevant subsets	Fast, scalable	Slight accuracy drop	FAISS (IndexIVFFlat)
HNSW (Hierarchical Navigable Small World)	Graph-based approximate search	Very fast, high accuracy	Expensive to build	hnswlib, FAISS, Qdrant
PQ (Product Quantization)	Compresses vectors to save memory	Memory-efficient, scalable	Loss of precision	FAISS (IndexIVFPQ)
Annoy	Uses random projection trees	Lightweight, fast	Lower accuracy	Spotify Annoy
Ball Tree / KD-Tree	Traditional tree structures	Good for low dimensions	Poor performance in high-dim	Scikit-learn
ScaNN (Google)	Learned indexing for high recall	Fast and accurate	More complex to tune	ScaNN library

ANN (Approximate Nearest Neighbors)

- Exact Nearest Neighbor (ENN):
  - Becomes computationally expensive in high-dimensional spaces (as dimensions increase, *distance calculations become less meaningful* and more costly).
  - Even brute-force methods with O(n) complexity outperforms in high-dimensional spaces.

- Approximate Nearest Neighbor (ANN):
  - Provides a trade-off between speed and accuracy.
  - Provides near-accurate results in a fraction of the time, making it viable for ML applications (e.g., 90-95% accurate results in milliseconds).

Performance of ANN : Recall

- **Recall** is a metric to measure the **ability of correctly identifying all relevant instances** (i.e., all true positives).

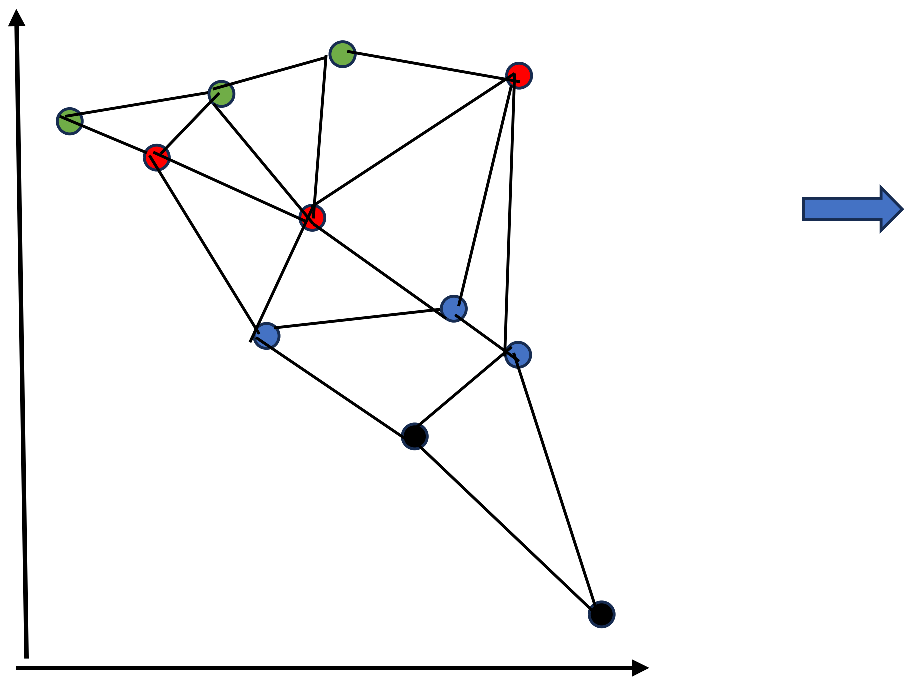
- $\text{Recall} = \dfrac{\text{True Positives (TP)}}{\text{True Positives (TP) + False Negatives (FN)}}$

- **True Positives (TP)**: Correctly predicted positive cases.
- **False Negatives (FN)**: Actual positive cases that were incorrectly predicted as negative.

- Intuition
  - **Recall** answers the question: *"Out of all the actual nearest neighbors, how many does it correctly find?"*
  - It focuses on **minimizing missed positives**.

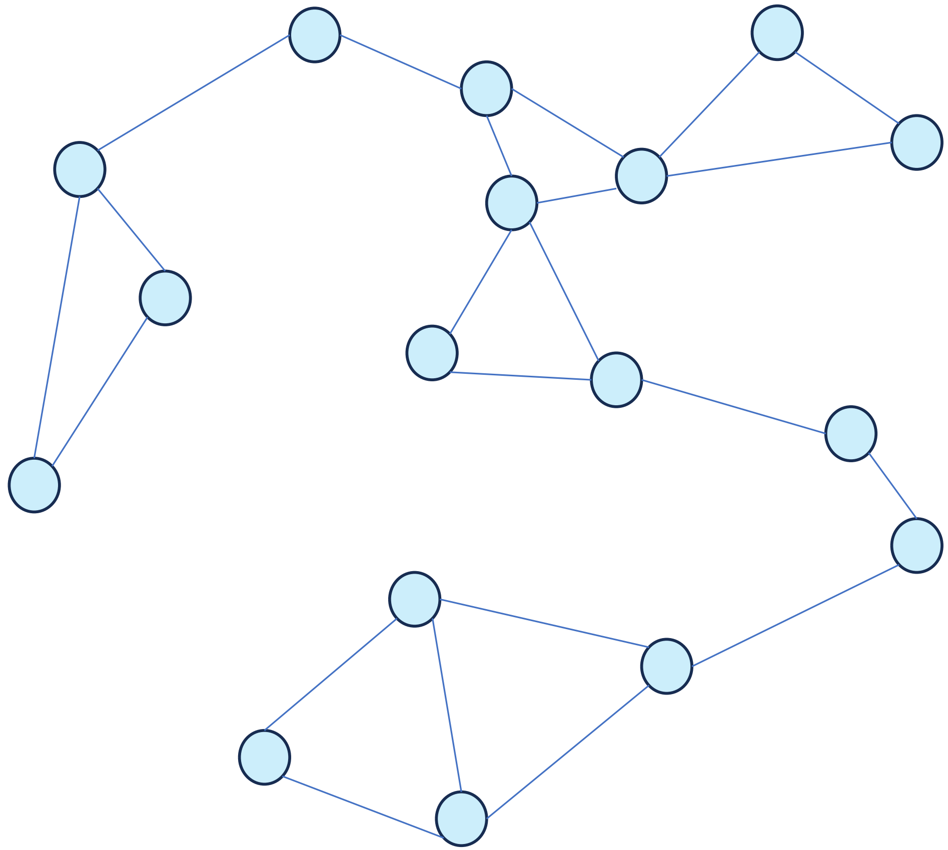
ANN (Approximate Nearest Neighbors)

- Proximity graph
  - A **proximity graph** is a graph where each node represents a data point, and edges connect nodes based on their **proximity** (closeness) according to a specific distance metric (e.g., Euclidean distance, cosine similarity).
  - Each node is connected to its **k-nearest neighbors**.



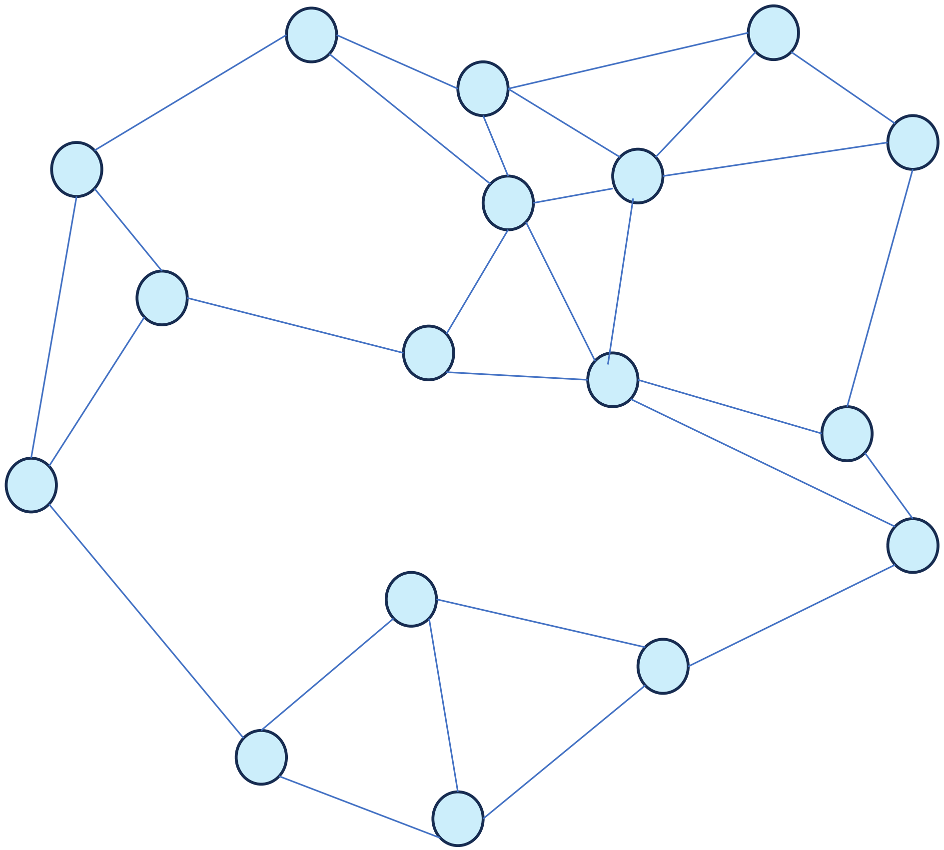
Proximity Graph (2NN)

- edges connect vertices that are close to each other based on distance



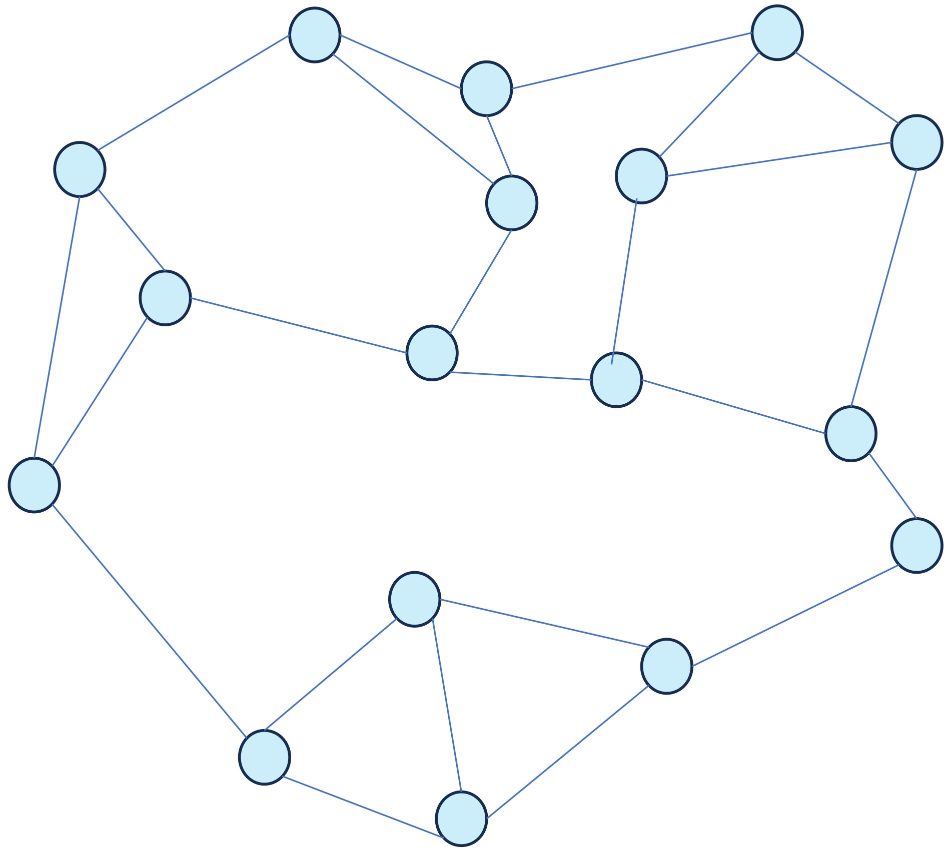
Proximity Graph (3NN)

- Full proximity graphs become too large in high-dimensions
  - Too many edges may introduce noise



Sparse Neighborhood Graph (SNG)

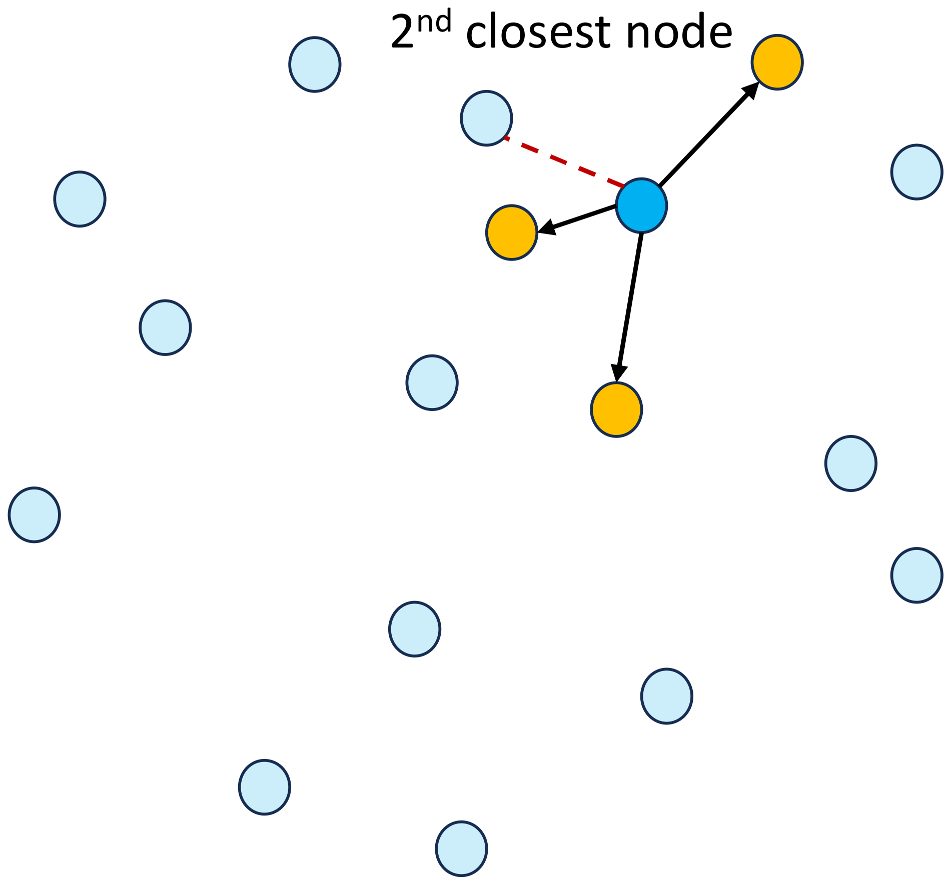
- Proximity graph, where only a **subset of edges** are retained to reduce memory or computational cost



In undirected SNG, some nodes may have fewer than K edges

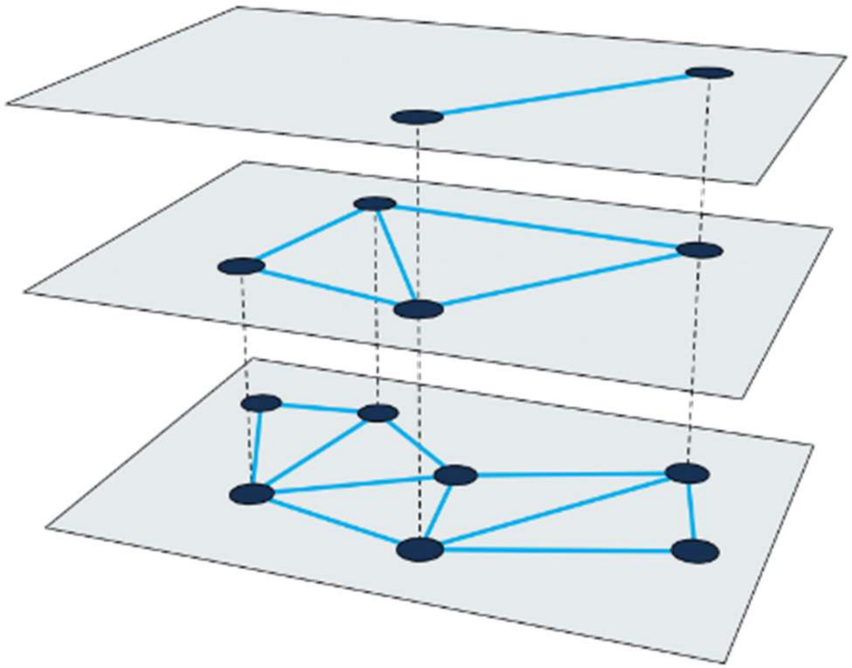
Sparse Neighborhood Graph (SNG)

- Nearest nodes are not always selected as neighbors



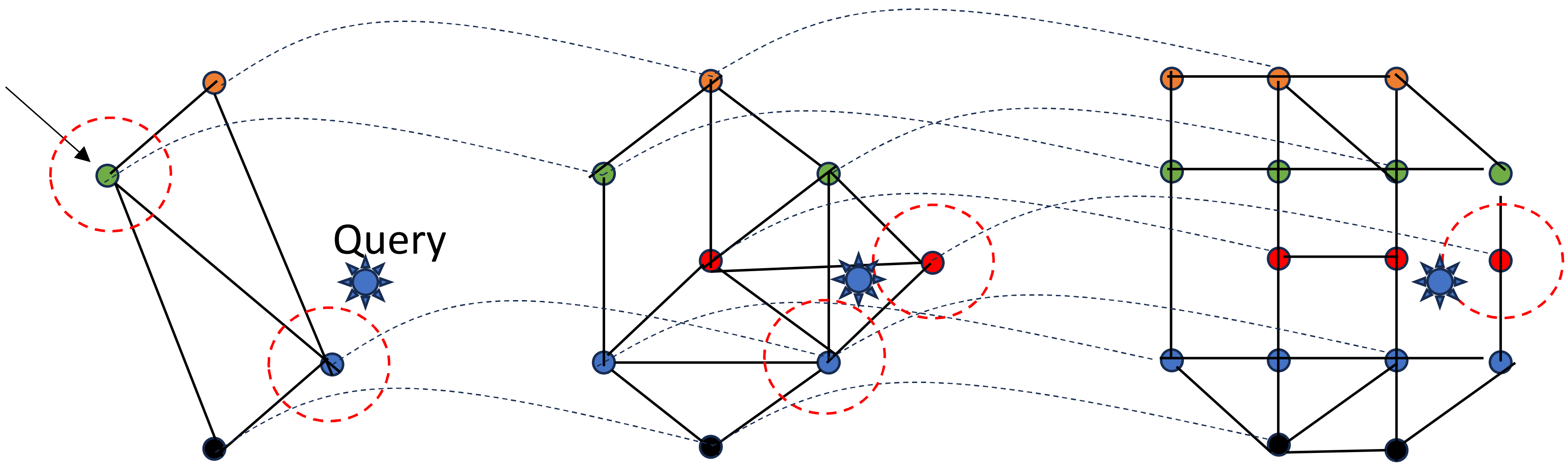
HNSW – Hierarchical Navigable-Small World

- **Hierarchical Navigable Small World (HNSW)** is an algorithm used for Approximate Nearest Neighbor (ANN) search.
- HNSW constructs hierarchical graphs where each node is connected to a set of nearby neighbors, creating a "small world" with short paths between any two points



HNSW – Hierarchical Navigable-Small World

- **Multiple Layers**: Nodes are organized in a hierarchy of layers of proximity graphs (similar to SkipLists).
  - **Lower layers** have denser connections and capture local neighborhoods.
  - **Higher layers** are sparsely connected and provide global navigation across the graph.
- Greedy search in each layer
- Elements inserted one by one by searching in so far constructed index



HNSW – Hierarchical Navigable-Small World

- **Index Construction**:
  1. Insert data points into multiple layers of the graph.
  2. High-level layers capture global relationships; lower-level layers store local neighborhood information.
  3. Each node connects to a subset of the most similar points in its layer.

- **Search Process**:
  1. Start at the **topmost layer** with a randomly selected node.
  2. Traverse the graph, moving to closer neighbors until reaching the bottom layer.
  3. In the **lowest layer**, perform a local search to find the nearest neighbors.