Lecture 16:
ND RTree

Logistics

- Two-page project updates due on **Oct 29** (extra credit)
- Programming assignment 3 (B+Tree) due on **Nov 2**

Recap

- Inverted Index
- Web-Scale Search

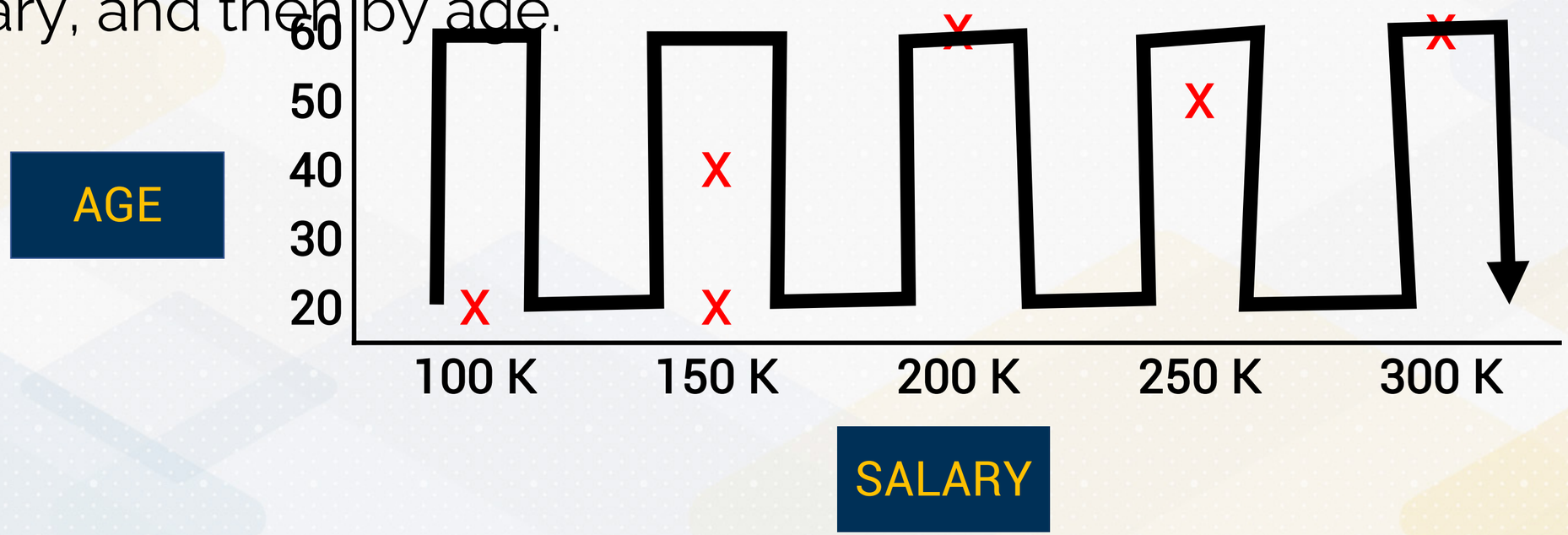
Lecture Overview

- RTree
- ND RTree

RTree

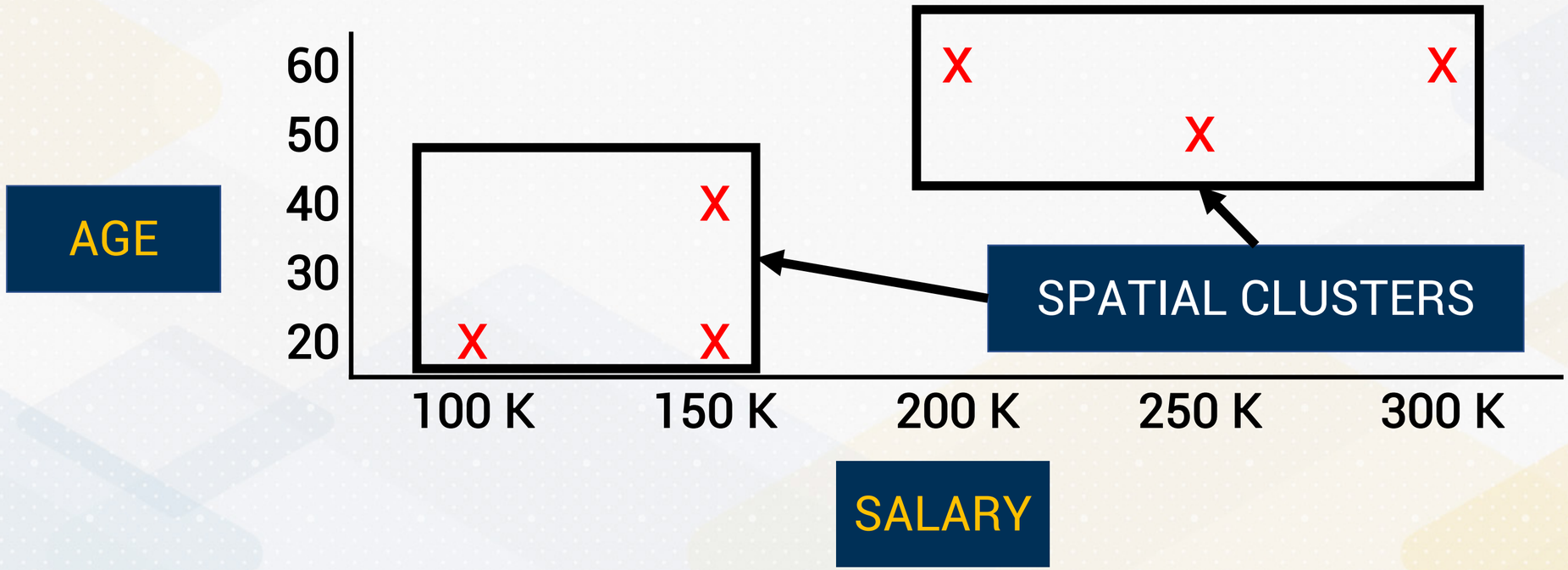
Limitations of B+Tree

- B+ trees are designed for single-dimensional indexing.
- When we create a composite key, such as an index on <salary, age>, we linearize the 2-dimensional space by sorting first by salary, and then by age.



RTree: A Multidimensional Index

- RTree groups the multi-dimensional keys in a way that takes advantage of their "nearness" in multiple dimensions.



RTree: A Multidimensional Index

- A hierarchical, multi-dimensional indexing structure that is used to efficiently manage spatial data (e.g., points, lines, rectangles).

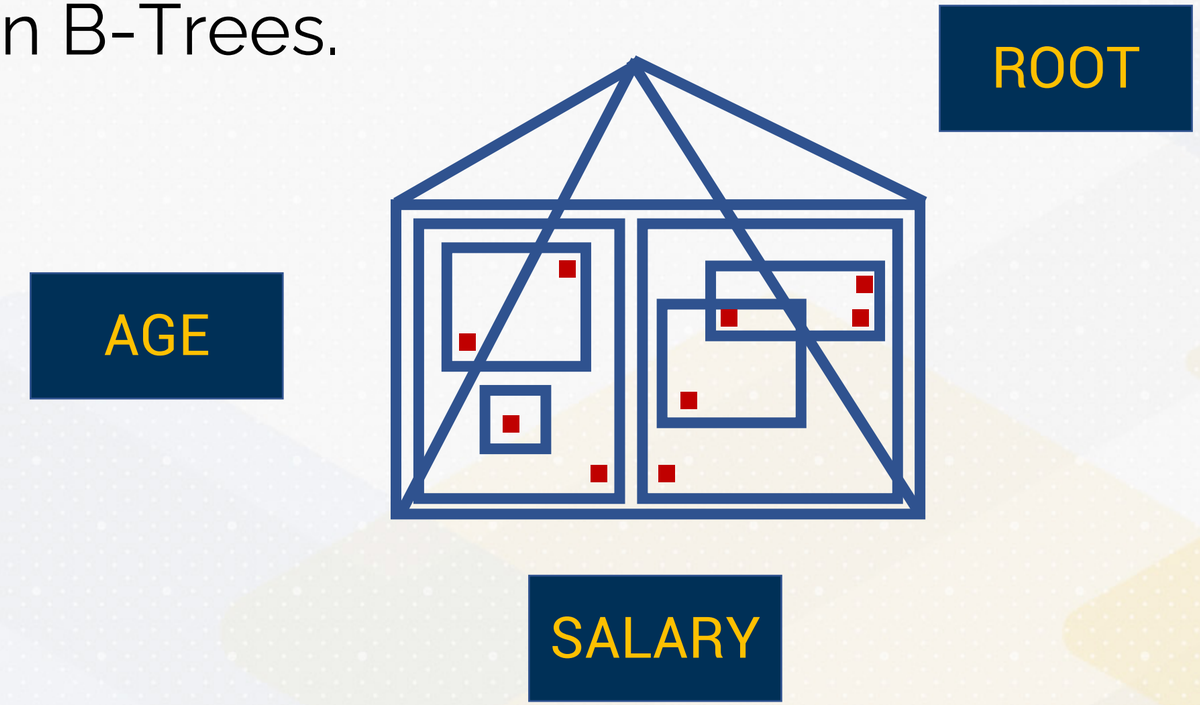
Antonin Guttman (1984)
Berkeley

RTree: A Multidimensional Index

- Spatial queries (GIS, CAD)
  - Find all hotels within a radius of 5 miles from Georgia Tech
  - Find the city with a population of 1,000,000 or more that is nearest to Atlanta
  - Find all cities that lie along the Chattahoochee River in Georgia
- Nearest neighbor queries (content-based retrieval)
  - Given a face, find the five most similar faces.

Key Characteristics

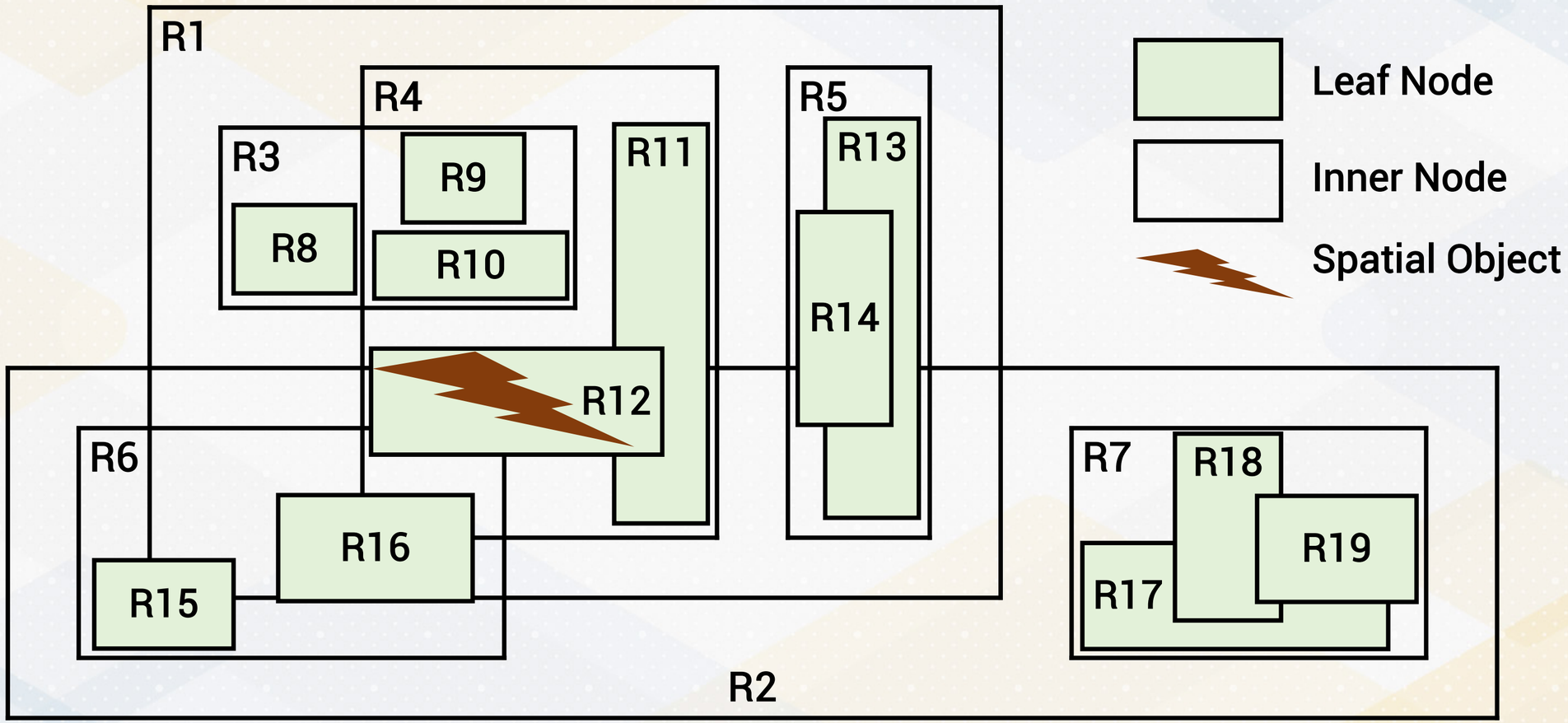
- Like B+ Trees, R-Trees are balanced, for efficient search operations.
- Partitions space using **bounding rectangles** instead of splitting at specific values, as in B-Trees.



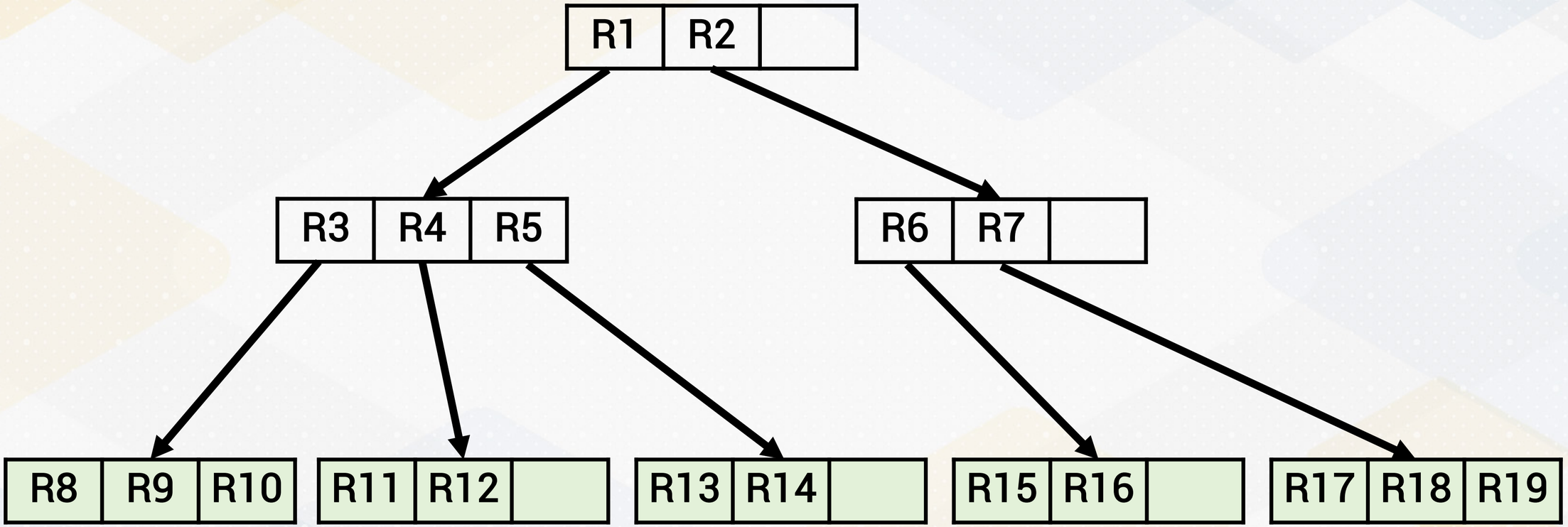
RTree Structure

- **Point**: A 2D coordinate representing a location in space.
- **Rectangle**: Defines a bounding box that encloses points or other rectangles.
- **R-Tree Node**:
  - **Leaf Node**: Contains points and is the smallest level of the tree.
  - **Inner Node**: Contains child nodes and their bounding rectangles.
  - **Root Node**: The top-most node of the R-Tree, which points to internal nodes or leaves.

RTree Structure



RTree Structure



Point

- Represents a location in 2D space (x, y).

```
struct Point {
    float x, y;
    Point(float x, float y) : x(x), y(y) {}
};
```

Rectangle

- Defines a bounding box using its minimum and maximum coordinates.

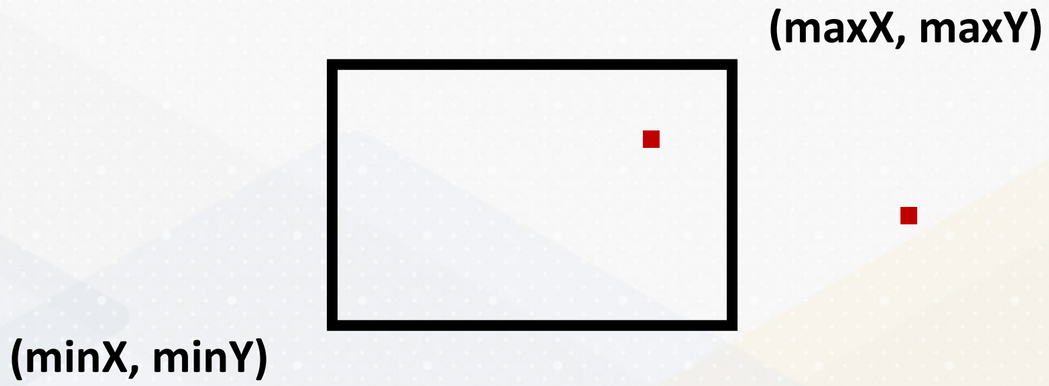
```
struct Rectangle {
    float minX, minY, maxX, maxY;
    Rectangle(float minX, float minY, float maxX, float maxY)
        : minX(minX), minY(minY), maxX(maxX), maxY(maxY) {}

    bool contains(const Point& p);
    bool intersects(const Rectangle& other) const;
};
```

Rectangle: Contains

- Check if a point lies inside the rectangle.

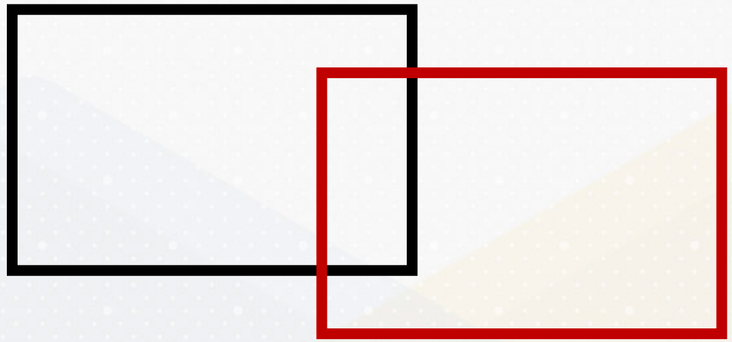
```
bool contains(const Point& p) const {
    return (p.x >= minX && p.x <= maxX && p.y >= minY && p.y <= maxY);
}
```



Rectangle: Intersection

- Determine if two rectangles overlap.

```
bool intersects(const Rectangle& other) const {
    return !(other.minX > maxX || other.maxX < minX ||
             other.minY > maxY || other.maxY < minY);
}
```



Inserting Points into the R-Tree

- For internal nodes, recursively find the best child node (based on minimal enlargement) to insert the point.

```
void insert(RTreeNode* node, const Point& point, const Rectangle& rect) {
    ...
    else {
        int bestChild = chooseBestChild(node, rect);
        insert(node->children[bestChild], point, rect);
        node->childrenRectangles[bestChild].expand(rect);  // Update bounding rectangle
    }
}
```

Best Child with Least Enlargement

- Choose the child node whose **bounding rectangle needs the smallest enlargement** to accommodate the new rectangle, so that the R-tree remains more compact.

```
int chooseBestChild(RTreeNode* node, const Rectangle& rect) {
        int bestChild = 0;
        for (size_t i = 0; i < node->children.size(); ++i) {
            Rectangle enlarged = node->childrenRectangles[i];
            enlarged.expand(rect);
            float enlargement = ...
            if (enlargement < minEnlargement) {
                minEnlargement = enlargement; bestChild = i;
            }
        }
        return bestChild;
    }
```

Inserting Points into the R-Tree

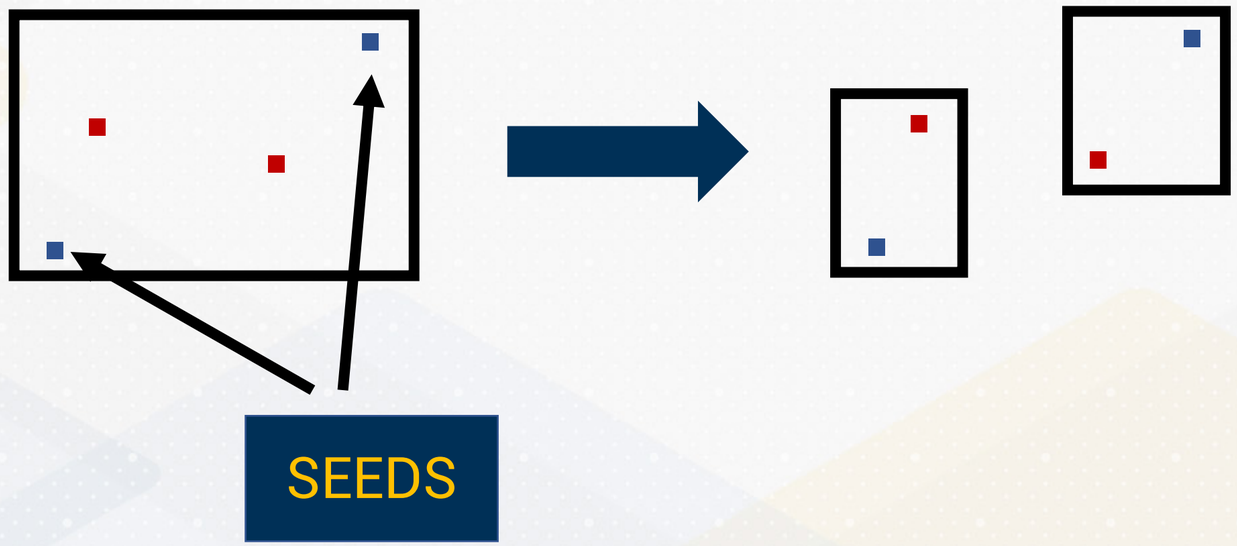
- Insert the point into the leaf node. If the leaf node is full, split the node into two, ensuring the tree stays balanced.

```
void insert(RTreeNode* node, const Point& point, const Rectangle& rect) {
    if (node->isLeaf) {
        node->points.push_back(point);
        if (node->points.size() > maxPoints) {
            split(node);  // Split the node if it exceeds max points
        }
    }
    ...
}
```

Node Splitting in R-Tree

- **Quadratic Split Algorithm**:
- When a node exceeds the maximum number of points, the node is split into two new nodes.
- The split is based on finding the two **most distant points** (seeds), and then assigning the remaining points to the node whose bounding box requires the least enlargement.

Node Splitting in R-Tree



Node Splitting in R-Tree

```
void chooseSeeds(const std::vector<Point>& points, int& seed1, int& seed2) {
    float maxDistance = -1;
    for (size_t i = 0; i < points.size(); ++i) {
        for (size_t j = i + 1; j < points.size(); ++j) {
            float distance = std::sqrt(std::pow(points[i].x - points[j].x, 2) +
                                       std::pow(points[i].y - points[j].y, 2));
            if (distance > maxDistance) {
                maxDistance = distance;
                seed1 = i;
                seed2 = j;
            }
        }
    }
}
```

Range Query in R-Tree

- Multidimensional range queries: 40 < age < 60 AND 200K < salary < 300K
- Recursively checks all the nodes whose bounding rectangles intersect with the query rectangle.
- For leaf nodes, it returns all points contained within the rectangle.
- For internal nodes, it recursively explores child nodes that may intersect the query.

Range Query in R-Tree

```
void query(RTreeNode* node, const Rectangle& rect, std::vector<Point>& results) {
    if (node->isLeaf) {
        for (const Point& p : node->points) {
            if (rect.contains(p)) {
                results.push_back(p);
            }
        }
    } else {
        for (size_t i = 0; i < node->children.size(); ++i) {
            if (rect.intersects(node->childrenRectangles[i])) {
                query(node->children[i], rect, results);
            }
        }
    }
}
```

Nearest Neighbor Search

- **Purpose**: Finds the k nearest neighbors of a point.
- Recursively computes the distance between the query point and bounding rectangles.
- It prioritizes searching through nodes closer to the query point by using a priority queue.

Nearest Neighbor Search (Internal Node)

```
void nearestNeighbor(RTreeNode* node, const Point& queryPoint, int k,
    std::priority_queue<std::pair<float, Point>>& pq) {
    ...
    else {
        std::vector<std::pair<float, RTreeNode*>> childDistances;
        for (size_t i = 0; i < node->children.size(); ++i) {
            float distance = node->childrenRectangles[i].minDistance(queryPoint);
            childDistances.push_back({distance, node->children[i]});
        }
        std::sort(childDistances.begin(), childDistances.end());
        for (const auto& child : childDistances) {
            nearestNeighbor(child.second, queryPoint, k, pq);
        }
    }
}
```

Nearest Neighbor Search (Leaf Node)

```
void nearestNeighbor(RTreeNode* node, const Point& queryPoint, int k,
    std::priority_queue<std::pair<float, Point>>& pq) {
    if (node->isLeaf) {
        for (const Point& p : node->points) {
            float distance = std::sqrt(std::pow(p.x - queryPoint.x, 2) + std::pow(p.y -
queryPoint.y, 2));
            pq.push(std::make_pair(distance, p));
            if (pq.size() > k) pq.pop();
        }
    }
    ...
}
```

Balancing in R-Trees

- Like B+ Trees, R-Trees are balanced to ensure efficient search times (in the order of O(log n)).
- Splitting and merging operations ensure that the tree remains balanced after insertions or deletions.

Splitting in R-Trees

- The goal is to minimize the **enlargement** of the bounding rectangles when adding new points.
- R-Trees use the **quadratic split algorithm** to ensure that nodes are split efficiently, minimizing the overall area expansion.

Restaurant Search on Maps

- Find nearby restaurants efficiently using spatial queries.
- **Inserting Restaurants**: Each restaurant can be represented as a Point in 2D space (latitude, longitude).
- **Range Queries**: To find restaurants within a given radius of a user's location, create a Rectangle that represents the bounding box (search radius).

```
Point restaurant({latitude, longitude}, "RestaurantName");
tree.insert(restaurant);

Rectangle searchArea({min_latitude, min_longitude}, {max_latitude, max_longitude});
std::vector<Point> nearbyRestaurants = tree.query(searchArea);
```

Uber Ride Matching

- Match riders to the nearest available drivers.
- **Inserting Driver Locations**: Each driver's location is represented as a Point in 2D space.
- **Nearest Neighbor Search**: When a rider requests a ride, use the nearestNeighbor() function to find the closest driver to the rider's location.

```
Point driver({latitude, longitude}, "DriverID");
tree.insert(driver);

Point riderLocation({rider_latitude, rider_longitude}, "RiderLocation");
int k = 10; // We want the nearest drivers
std::vector<Point> nearestDrivers = tree.nearestNeighbor(riderLocation, k);
```

Game Engine Spatial Partitioning

- Efficiently manage and query game objects in large open-world games.
- **Inserting Game Objects**: Game objects (e.g., player positions) are represented as points in an N-dimensional space (e.g., 3D for X, Y, Z coordinates).
- **Collision Detection / Range Queries**: To check for collisions or visible objects, retrieve all game objects within a bounding box (e.g., area around a player).

```
Point gameObject({x, y, z}, "GameObjectID");
tree.insert(gameObject);

Rectangle collisionArea({minX, minY, minZ}, {maxX, maxY, maxZ});
std::vector<Point> nearbyObjects = tree.query(collisionArea);
```

Advantages

- Efficient for both point and range queries.
- Supports dynamic data (inserts and deletions) while maintaining balance.
- Optimized for spatial queries.

Challenges

- The quadratic split algorithm can be computationally expensive.
- R-Trees can suffer from overlapping bounding boxes, which may increase search time for large datasets.
- There are variants of this data structure that improve performance by better handling splits and minimizing overlaps.

Conclusion

- R-Trees are crucial for handling spatial data, supporting range queries and nearest neighbor searches.
- They efficiently balance space usage and query performance using bounding rectangles.

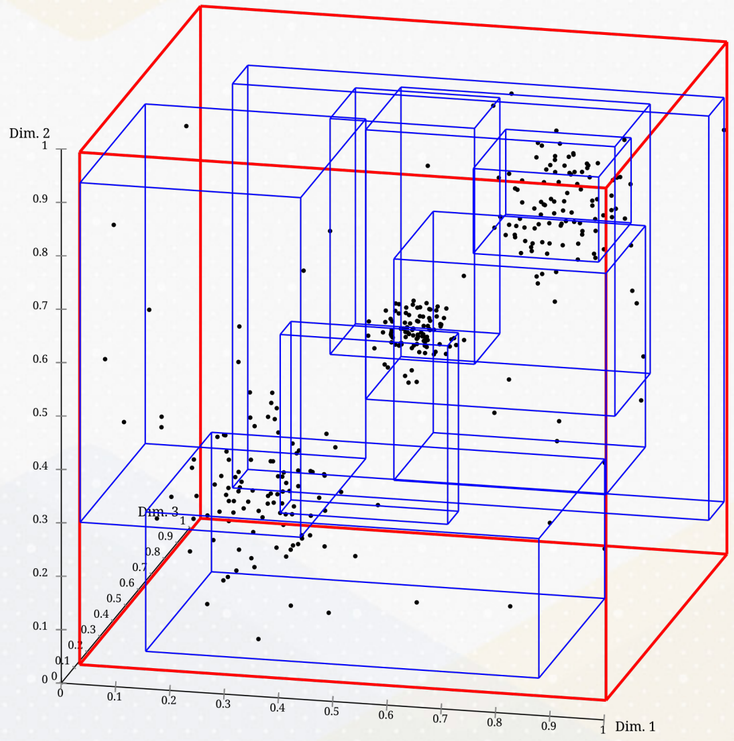
ND RTree

Limitations of RTree

- R trees are designed for two-dimensional indexing.
- ND R-Tree is an extension of the R-Tree, supporting indexing in arbitrary high-dimensional spaces.
- Real-world applications include:
- **Image Recognition:** Indexing feature vectors from neural networks to efficiently find similar images.
- **Recommendation Systems:** Managing high-dimensional user-item interaction data for content-based recommendations.

ND RTree: A Multidimensional Index

- ND R-Tree efficiently manages data in hundreds of dimensions (e.g., 128 or 512).



ND RTree Structure

- **Point**: Represents a location in N-dimensional space with coordinates and an optional label.
- **Hyper-Rectangle**: A bounding hyperrectangle in N-dimensional space to enclose points or rectangles.
- **R-Tree Node**:
  - **Leaf Node**: Contains points and is the smallest level of the tree.
  - **Inner Node**: Contains child nodes and their bounding hyperrectangles.
  - **Root Node**: The top-most node of the R-Tree, which points to internal nodes or leaves.

Point

- High-dimensional points (e.g., 512 dimensions).
- **Example:** A feature vector representing an image with 512 dimensions, where each value is a descriptor of the image content.

```
struct Point {
    std::vector<float> coordinates;
    std::string label;
};
```

Rectangle

- A bounding box that represents the minimum and maximum feature values for a cluster of points in a high-dimensional space.

```
struct Rectangle {
    std::vector<float> minCoords, maxCoords;
    bool contains(const Point& p) const;
};
```

Nodes

- Internal nodes use bounding hyper-rectangles to efficiently index child nodes.

```
struct RTreeNode {
    bool isLeaf;
    std::vector<Point> points;
    std::vector<Rectangle> childrenRectangles;
    std::vector<RTreeNode*> children;
};
```

Inserting Points into the ND R-Tree

- Points are inserted into the ND R-Tree by navigating down the tree to find the best-fit leaf node.

```
void insert(RTreeNode* node, const Point& point, const Rectangle& rect) {
    ...
    else {
        int bestChild = chooseBestChild(node, rect);
        insert(node->children[bestChild], point, rect);
        node->childrenRectangles[bestChild].expand(rect);  // Update bounding rectangle
    }
}
```

Best Child with Least Enlargement

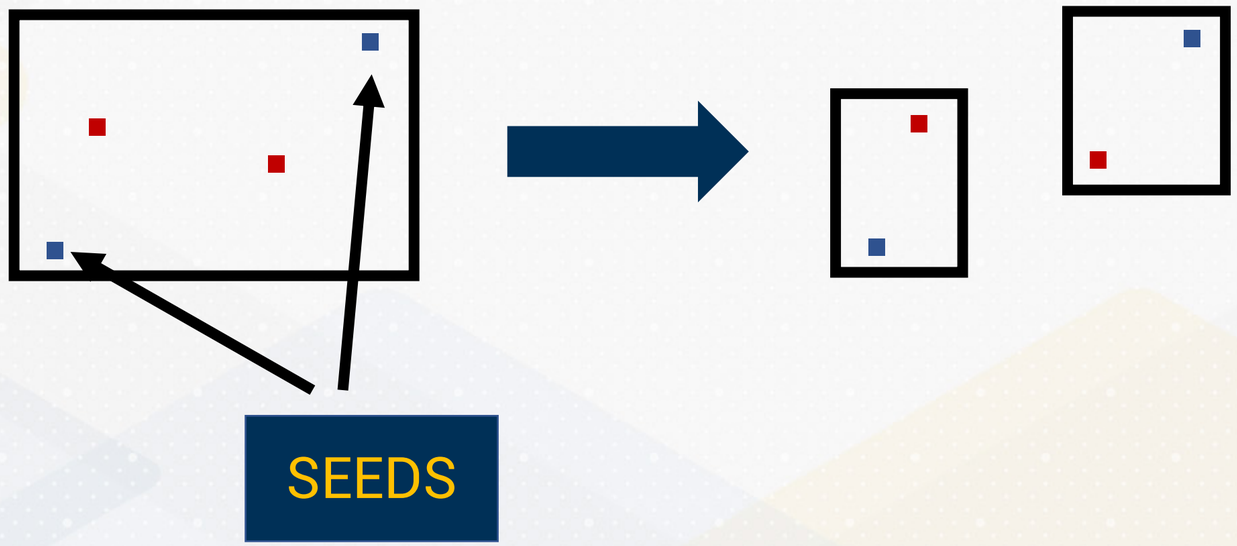
- Minimizes bounding hyperrectangle enlargement to maintain spatial locality.

```
int chooseBestChild(RTreeNode* node, const Rectangle& rect) {
    int bestChild = 0;
    float minEnlargement = std::numeric_limits<float>::max();
    for (size_t i = 0; i < node->children.size(); ++i) {
        Rectangle enlarged = node->childrenRectangles[i];
        enlarged.expand(rect);
        float enlargement = enlarged.area() - node->childrenRectangles[i].area();
        if (enlargement < minEnlargement) {
            minEnlargement = enlargement;
            bestChild = i;
        }
    }
    return bestChild;
}
```

Node Splitting in ND R-Tree

- If a node overflows, it is split using a quadratic algorithm that aims to minimize bounding box overlap.
- **Quadratic Split:** Selects the two most distant points (seeds) and distributes the remaining entries to minimize area enlargement.

Node Splitting in ND R-Tree



Node Splitting in R-Tree

```
void chooseSeeds(const std::vector<Point>& points, int& seed1, int& seed2) {
    float maxDistance = -1;
    for (size_t i = 0; i < points.size(); ++i) {
        for (size_t j = i + 1; j < points.size(); ++j) {
            float distance = std::sqrt(std::pow(points[i].x - points[j].x, 2) +
                                       std::pow(points[i].y - points[j].y, 2));
            if (distance > maxDistance) {
                maxDistance = distance;
                seed1 = i;
                seed2 = j;
            }
        }
    }
}
```

Nearest Neighbor Search

- **Purpose**: Finds the k nearest neighbors of a point.
- **Recursive Search Approach:** Traverses nodes in increasing order of distance from the query point, using a priority queue.

Nearest Neighbor Search (Internal Node)

```
void nearestNeighbor(RTreeNode* node, const Point& queryPoint, int k,
                     std::priority_queue<std::pair<float, Point>>& pq) {
    if (node->isLeaf) { ..}
    else {
        std::vector<std::pair<float, RTreeNode*>> childDistances;
        for (size_t i = 0; i < node->children.size(); ++i) {
            float distance = node->childrenRectangles[i].minDistance(queryPoint);
            childDistances.push_back(std::make_pair(distance, node->children[i]));
        }
        std::sort(childDistances.begin(), childDistances.end());
        for (const auto& child : childDistances) {
            nearestNeighbor(child.second, queryPoint, k, pq);
        }
    }
}
```

Nearest Neighbor Search (Leaf Node)

```
void nearestNeighbor(RTreeNode* node, const Point& queryPoint, int k,
                     std::priority_queue<std::pair<float, Point>>& pq) {
    if (node->isLeaf) {
        for (const Point& p : node->points) {
            float distance = 0.0;
            for (size_t i = 0; i < p.coordinates.size(); ++i) {
                distance += std::pow(p.coordinates[i] - queryPoint.coordinates[i], 2);
            }
            distance = std::sqrt(distance);
            pq.push(std::make_pair(distance, p));
            if (pq.size() > static_cast<size_t>(k)) {
                pq.pop();
            }
        }
    } ...
}
```

Range Query in ND R-Tree

- ND R-Tree can efficiently perform **multidimensional range queries**, which are common in high-dimensional datasets.
- The algorithm recursively traverses nodes whose bounding rectangles intersect with the query region.
- For leaf nodes, it returns all points contained within the rectangle.
- For internal nodes, it recursively explores child nodes that may intersect the query.

Range Query in R-Tree

```
void query(RTreeNode* node, const Rectangle& rect, std::vector<Point>& results) {
    if (node->isLeaf) {
        for (const Point& p : node->points) {
            if (rect.contains(p)) {
                results.push_back(p);
            }
        }
    } else {
        for (size_t i = 0; i < node->children.size(); ++i) {
            if (rect.intersects(node->childrenRectangles[i])) {
                query(node->children[i], rect, results);
            }
        }
    }
}
```

Image Retrieval in Computer Vision

- **Goal**: In computer vision, image features can be represented as high-dimensional vectors. Use R-trees for efficient similarity search (e.g., finding similar images or objects in a dataset based on feature vectors).

```
std::vector<float> imageFeatureVector(512, 0.0f);  // 512-dimensional feature vector
Point imagePoint(imageFeatureVector, "ImageID");
tree.insert(imagePoint);

Point queryImage(queryFeatureVector, "QueryImage");
int k = 5;  // Find 5 most similar images
std::vector<Point> similarImages = tree.nearestNeighbor(queryImage, k);
```

Climate Data Analysis

- Perform multidimensional analysis on climate data, which typically involves multiple factors (temperature, humidity, wind speed, etc.). R-trees can be used to efficiently query and analyze multidimensional climate data.

```
std::vector<float> climateData = {temperature, humidity, windSpeed, ...};
Point climatePoint(climateData, "ClimateDataID");
tree.insert(climatePoint);

std::vector<float> minValues = {minTemp, minHumidity, minWindSpeed};
std::vector<float> maxValues = {maxTemp, maxHumidity, maxWindSpeed};
Rectangle queryRange(minValues, maxValues);
std::vector<Point> matchingClimateData = tree.query(queryRange);
```

Challenges

- The key difficulty of ND R-tree is to build an efficient tree that:
  - On one hand is balanced (so the leaf nodes are at the same height)
  - On the other hand the rectangles do not cover too much empty space and do not overlap too much (so that during search, fewer subtrees need to be processed).

Curse of Dimensionality in R-Trees

- As the number of dimensions (n) increases, the volume of space grows exponentially.
- Data points become sparsely distributed, making it difficult for algorithms like R-Trees to efficiently manage and query data.
- The intuitive notion of "closeness" in lower dimensions becomes less meaningful in high-dimensional spaces.

Curse of Dimensionality in R-Trees

- **Inefficiency in Node Splitting:**
- In high-dimensional spaces, minimum bounding rectangles (MBRs) tend to overlap significantly.
- Increased overlap makes it difficult for R-trees to prune search space efficiently, leading to more nodes being visited during queries.

Curse of Dimensionality in R-Trees

- **Increased Query Time:**
- The expected number of node accesses grows as dimensionality increases, which reduces the performance of range queries and nearest neighbor searches.
- R-trees perform well in 2D or 3D, but as the number of dimensions grows (e.g., >10), their performance degrades.

Curse of Dimensionality in R-Trees

- **Large Minimum Bounding Rectangle (MBR) Volumes:**
- Minimum Bounding Rectangles expand disproportionately with additional dimensions, causing them to enclose vast empty regions of space.
- Many queries will require searching through multiple large MBRs, even if relevant data points are scarce in those areas.

Conclusion

- Inverted Index
- Web-Scale Search
- RTree