

Lecture 17:

RTree



Logistics

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

Recap

- Trie
- Inverted Index
- Web-Scale Search

Lecture Overview

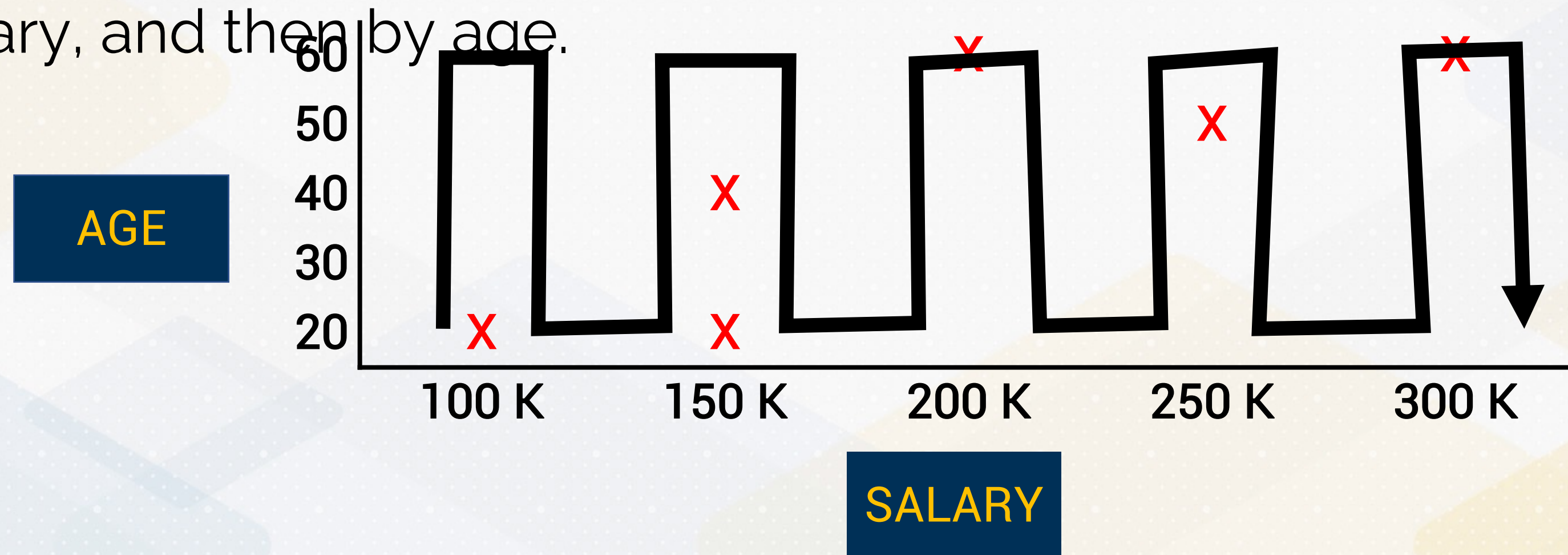
- RTree
- ND-RTree

RTree



Limitations of B+Tree

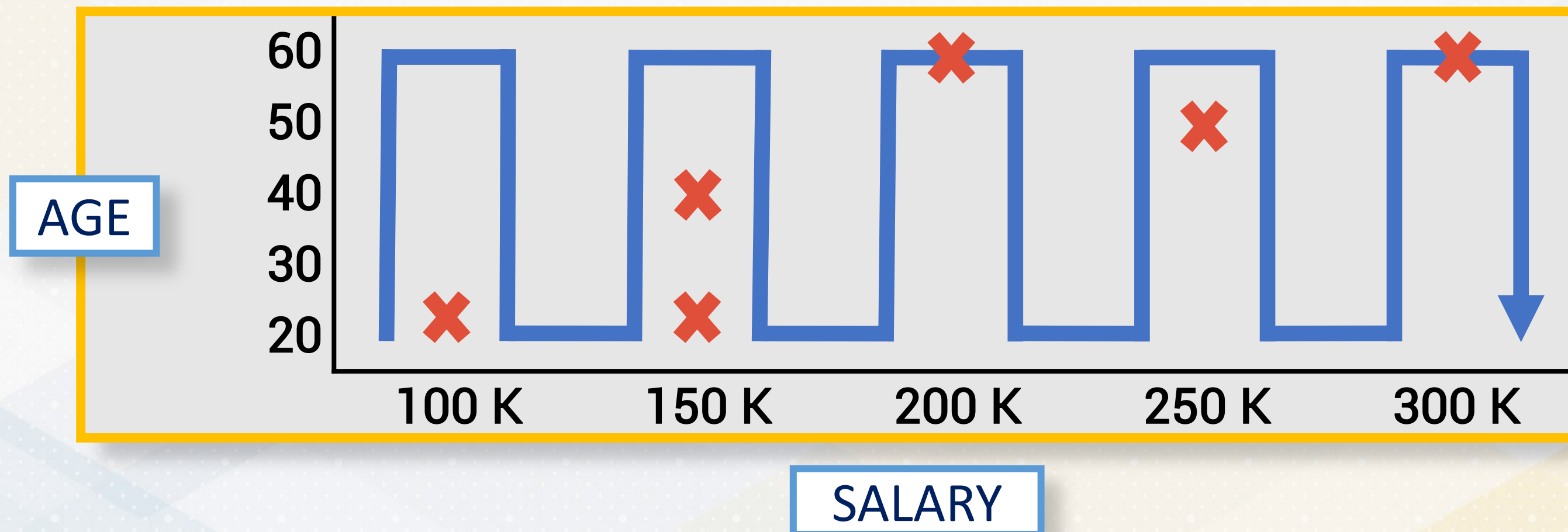
- B+ trees are designed for single-dimensional indexing.
- When we create a composite key, such as an index on <salary, age>, we linearize the 2-dimensional space by sorting first by salary, and then by age.



Limitations of B+Tree

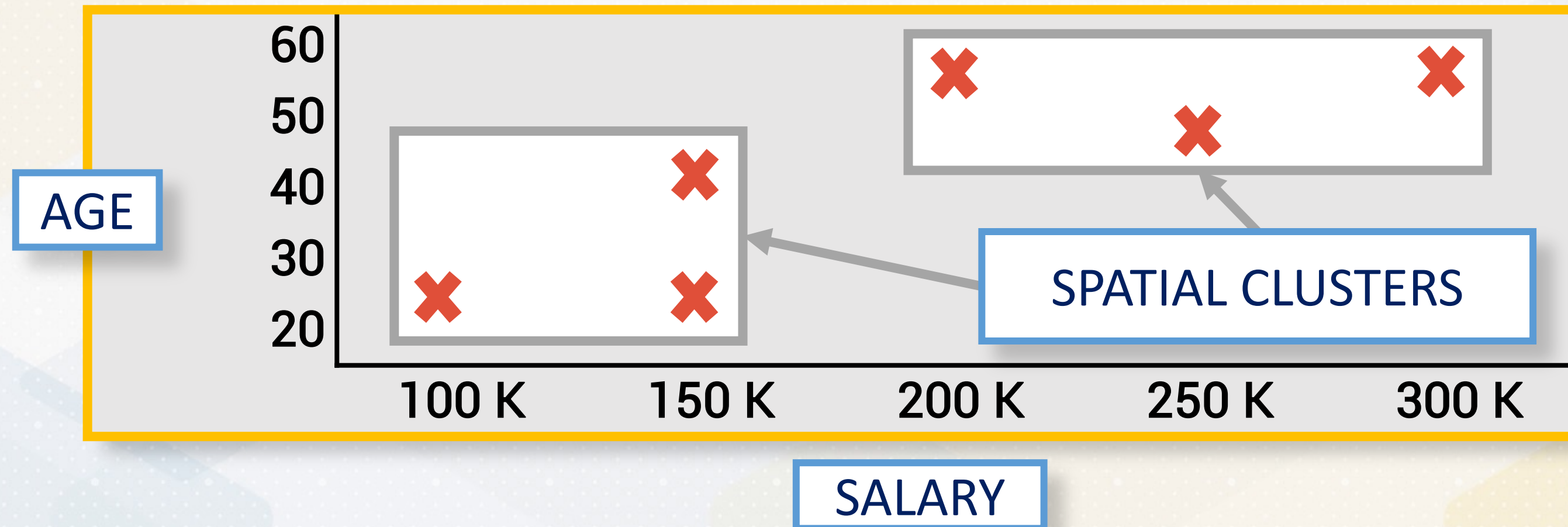


Designed for Single-Dimensional Indexing



R-Tree: A Multidimensional Index

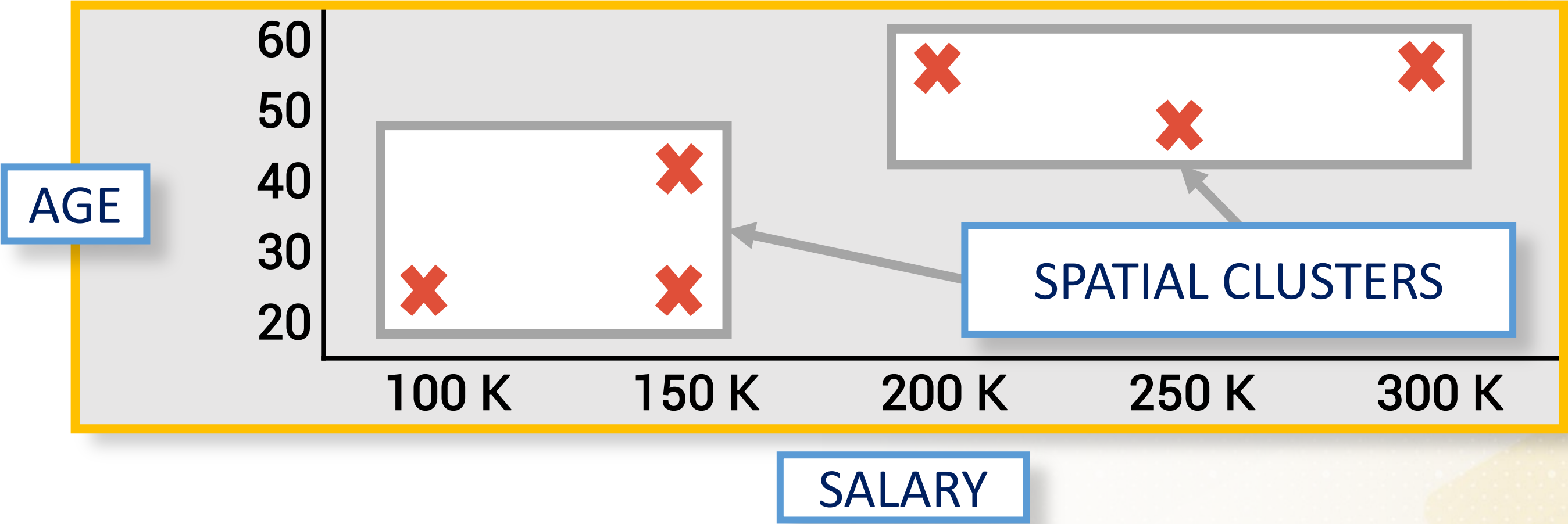
R-Tree groups multi-dimensional keys based on spatial proximity



R-Tree: A Multidimensional Index

(Salary, Age): Point

Clusters: Bounding Rectangles

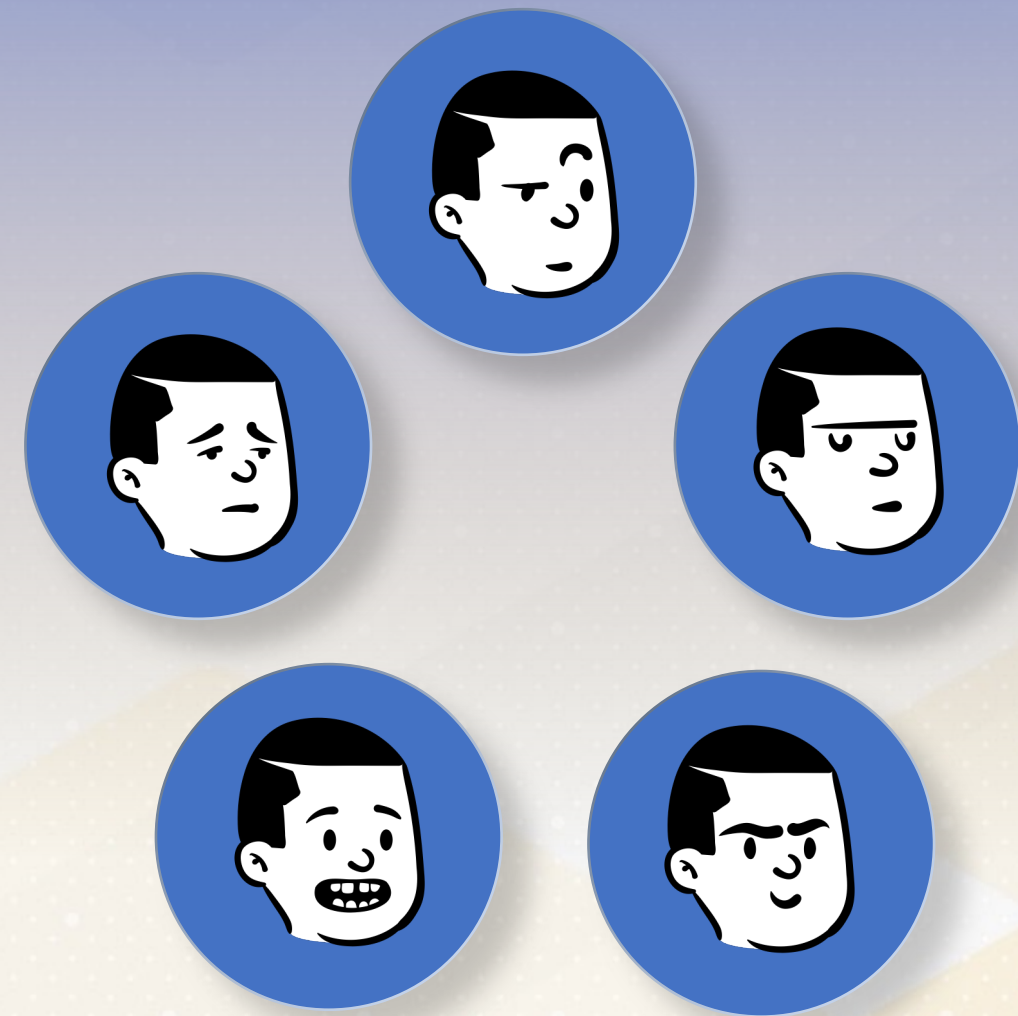


Supported Queries

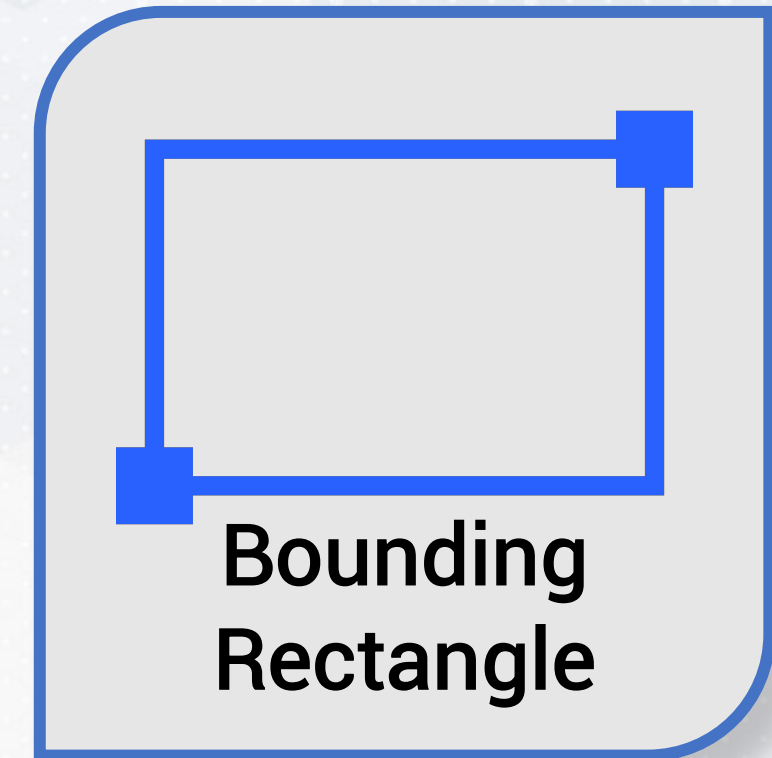
Spatial Queries



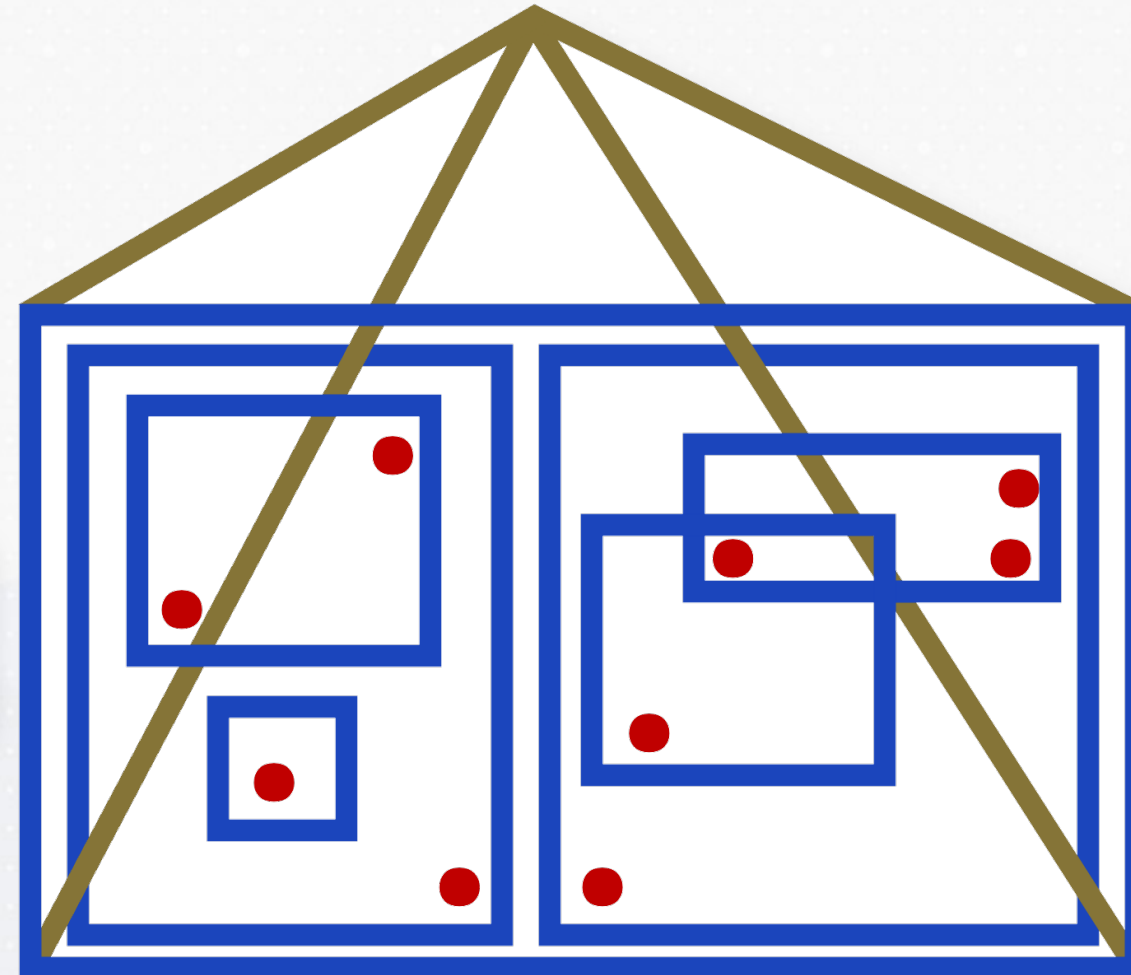
Nearest Neighbor Queries



Key Characteristics



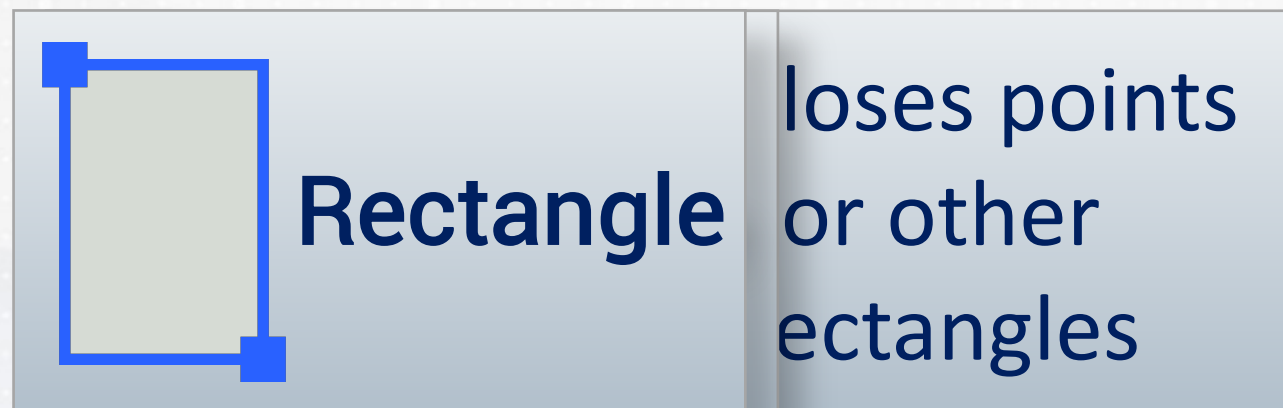
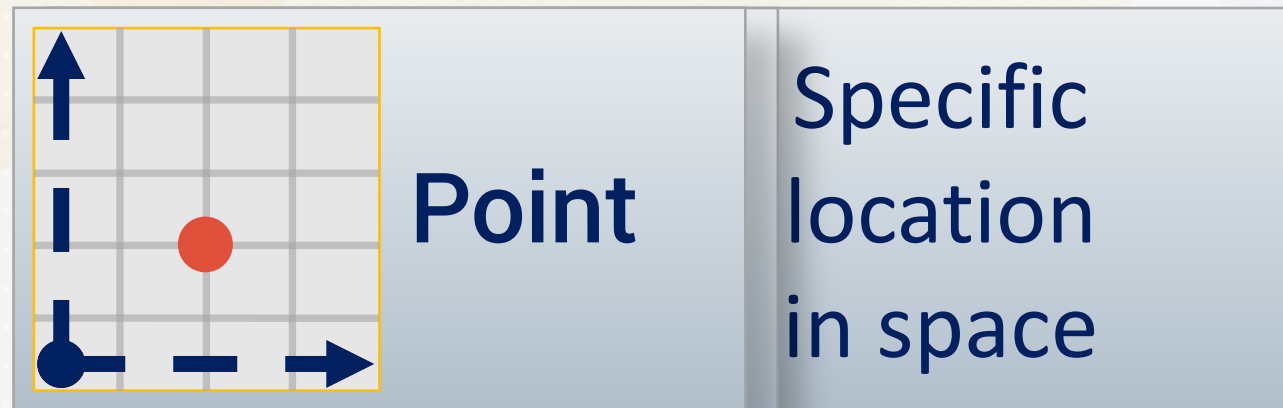
AGE



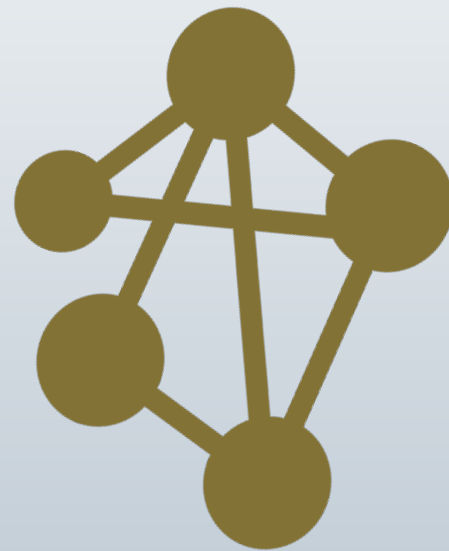
ROOT

SALARY

R-Tree Data Types



R-Tree Nodes



**R-Tree
Node**

Leaf Node

Contains
Points; Lowest
Level of Tree

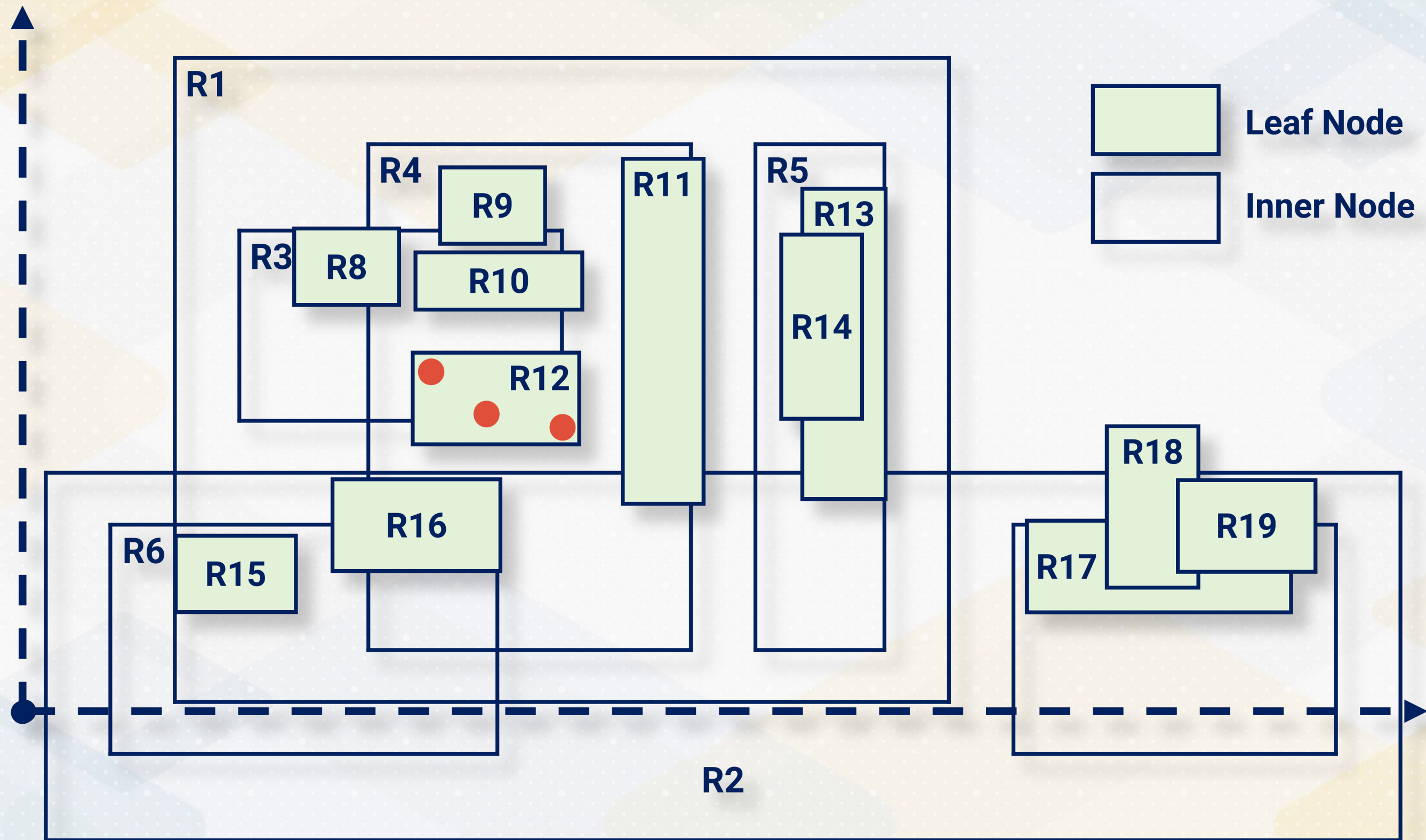
Inner Node

Contains Child
Nodes and
their Bounding
Rectangles

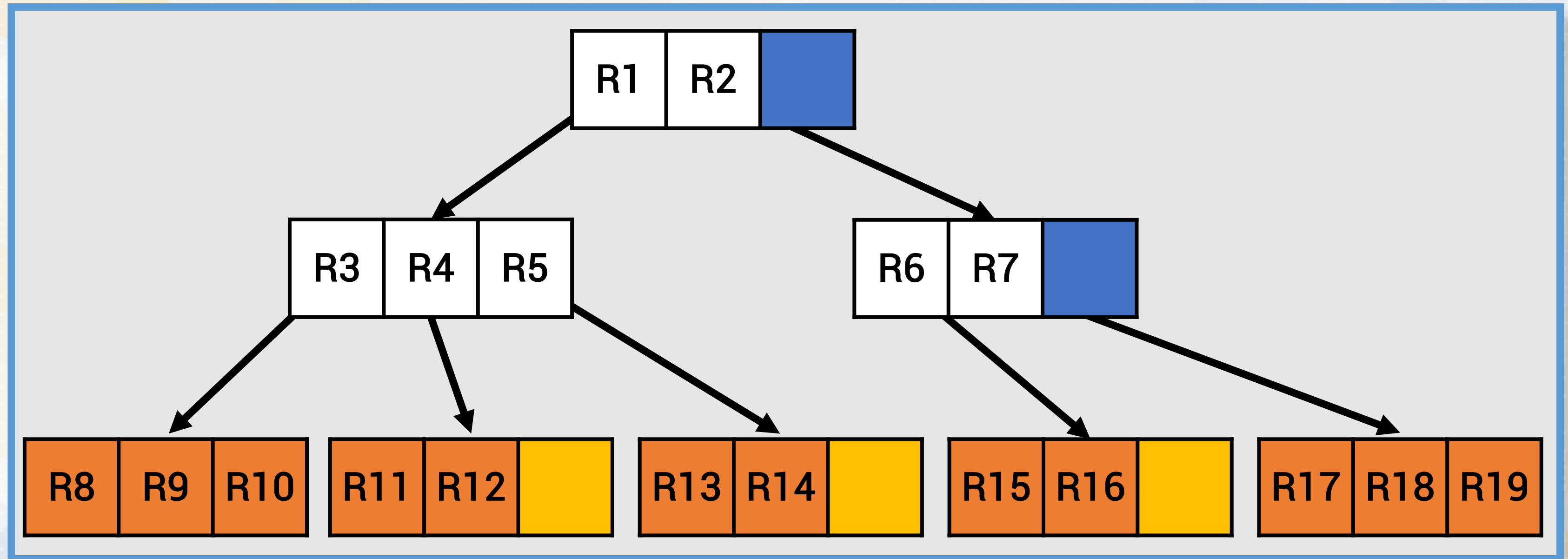
Root Node

Top-Most Inner
Node;
Entry
Point

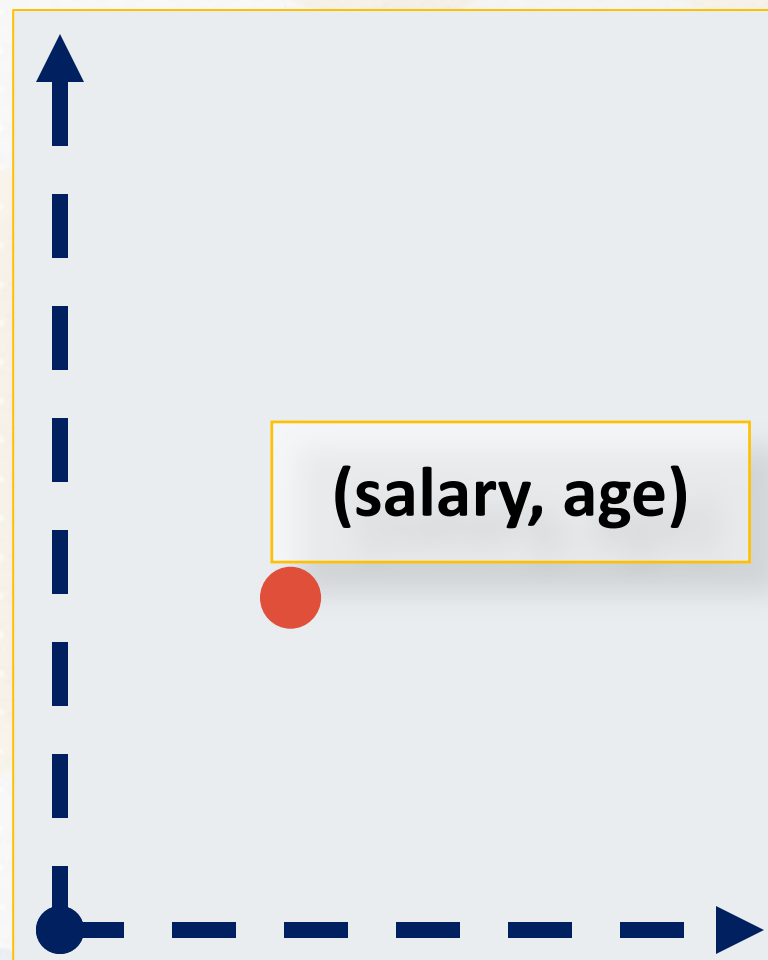
R-Tree Structure



R-Tree Structure



Point

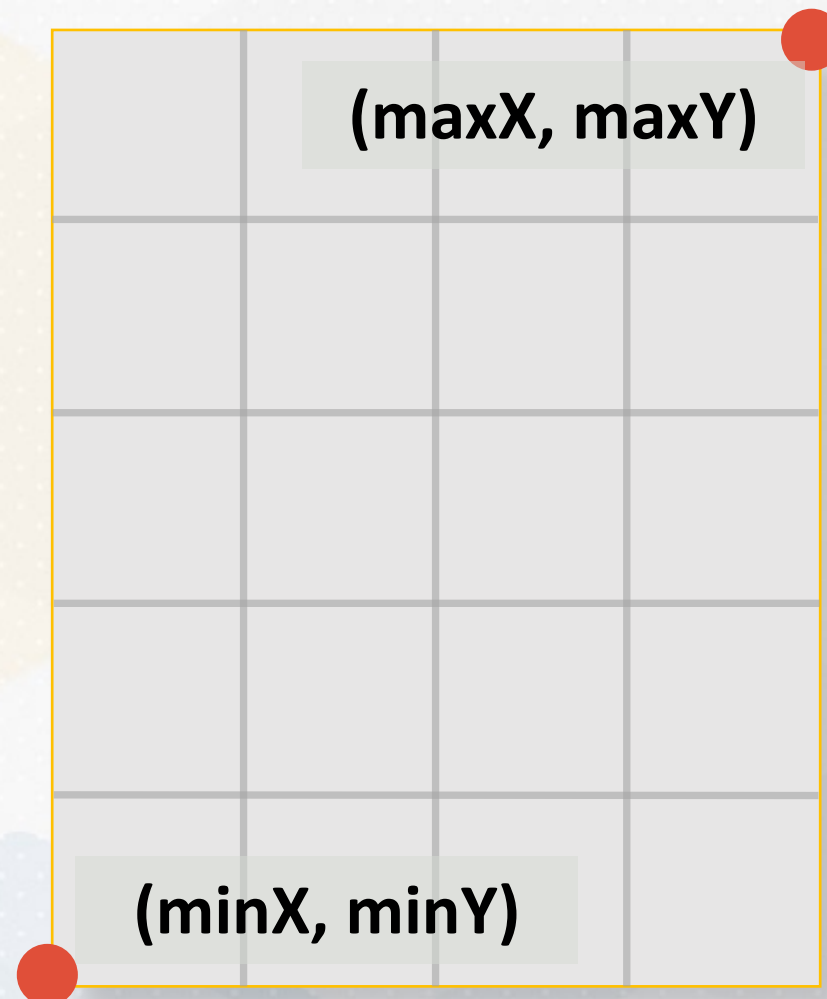


$(x, y) = (\text{salary}, \text{age})$

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


Rectangle

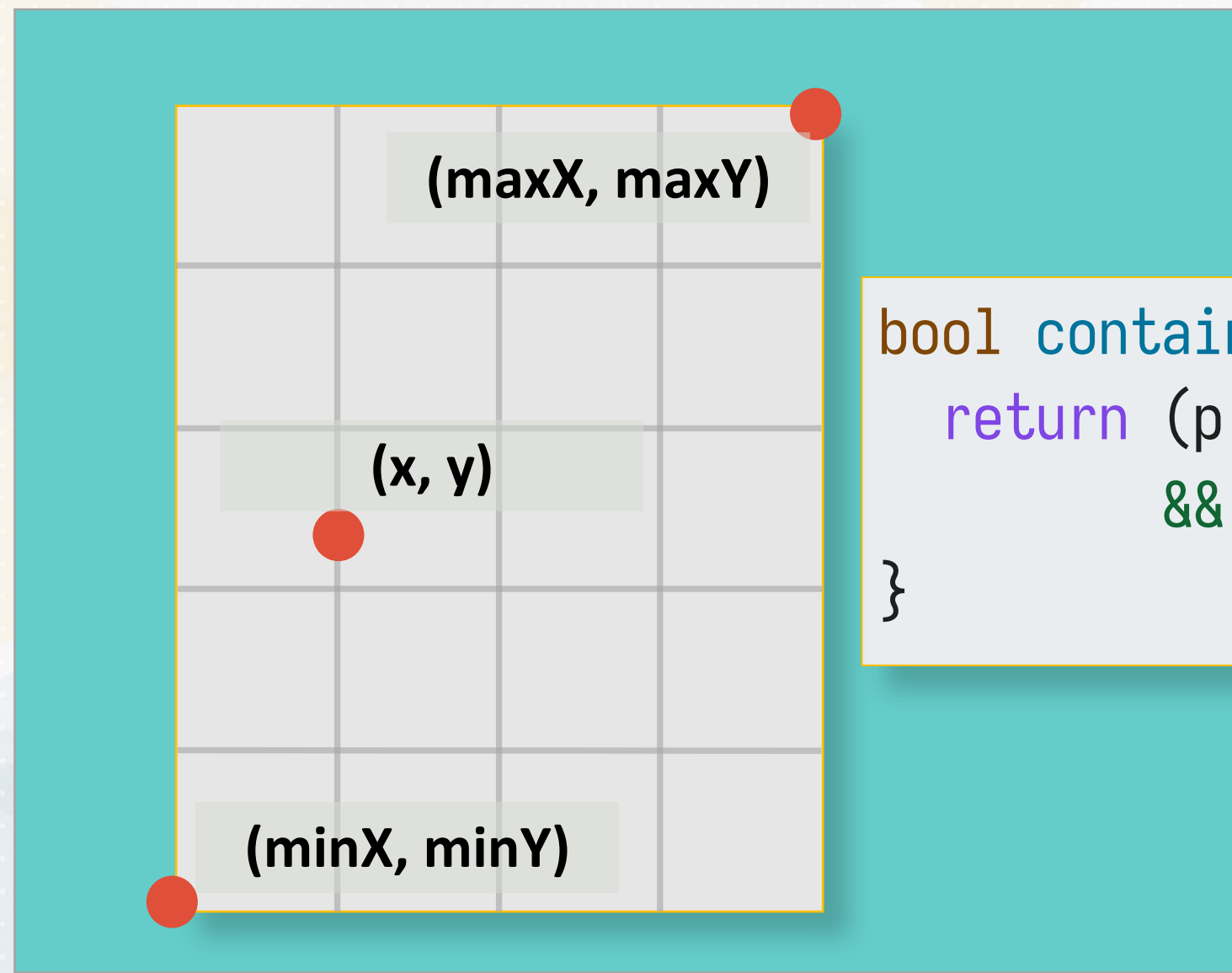
Defines a bounding box using its minimum and maximum coordinates



```
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

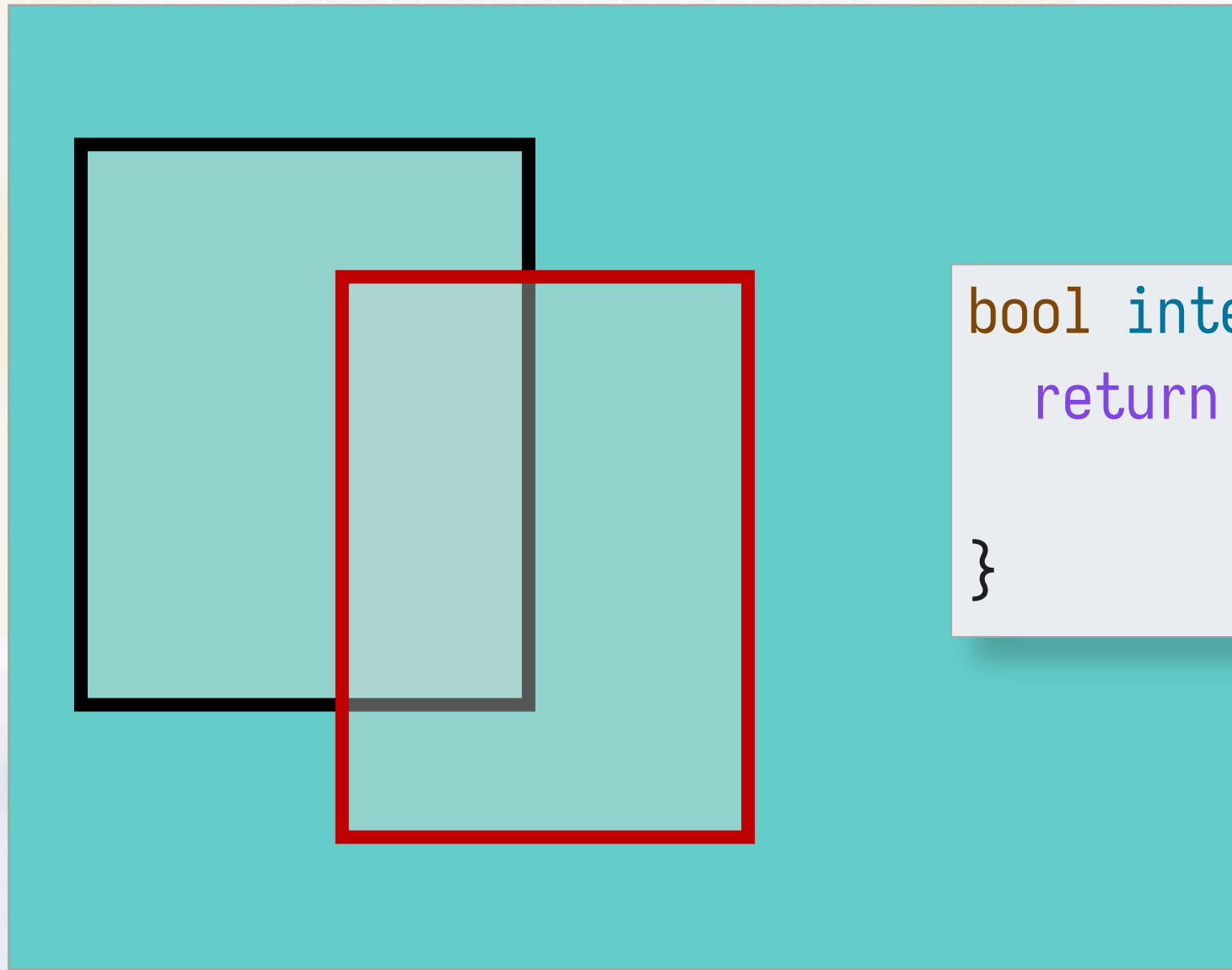
Check if a point lies inside the rectangle



```
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

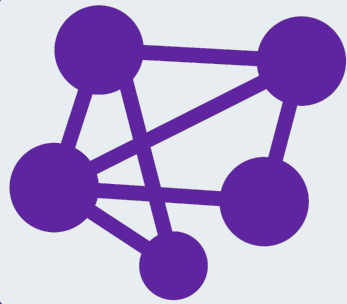


```
bool intersects(const Rectangle &other) const {  
    return !(other.minX > maxX || other.maxX < minX  
            || other.minY > maxY || other.maxY < minY);  
}
```


Inserting Points



Inserting Points into the R-Tree



Inner
Node

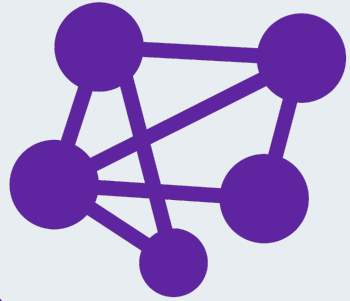
- Recursively finds best child node
- With least enlargement of bounding rectangle

```
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);  
    }  
}
```

Best Child with Least Enlargement

```
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;  
}
```


Leaf Node Insertion

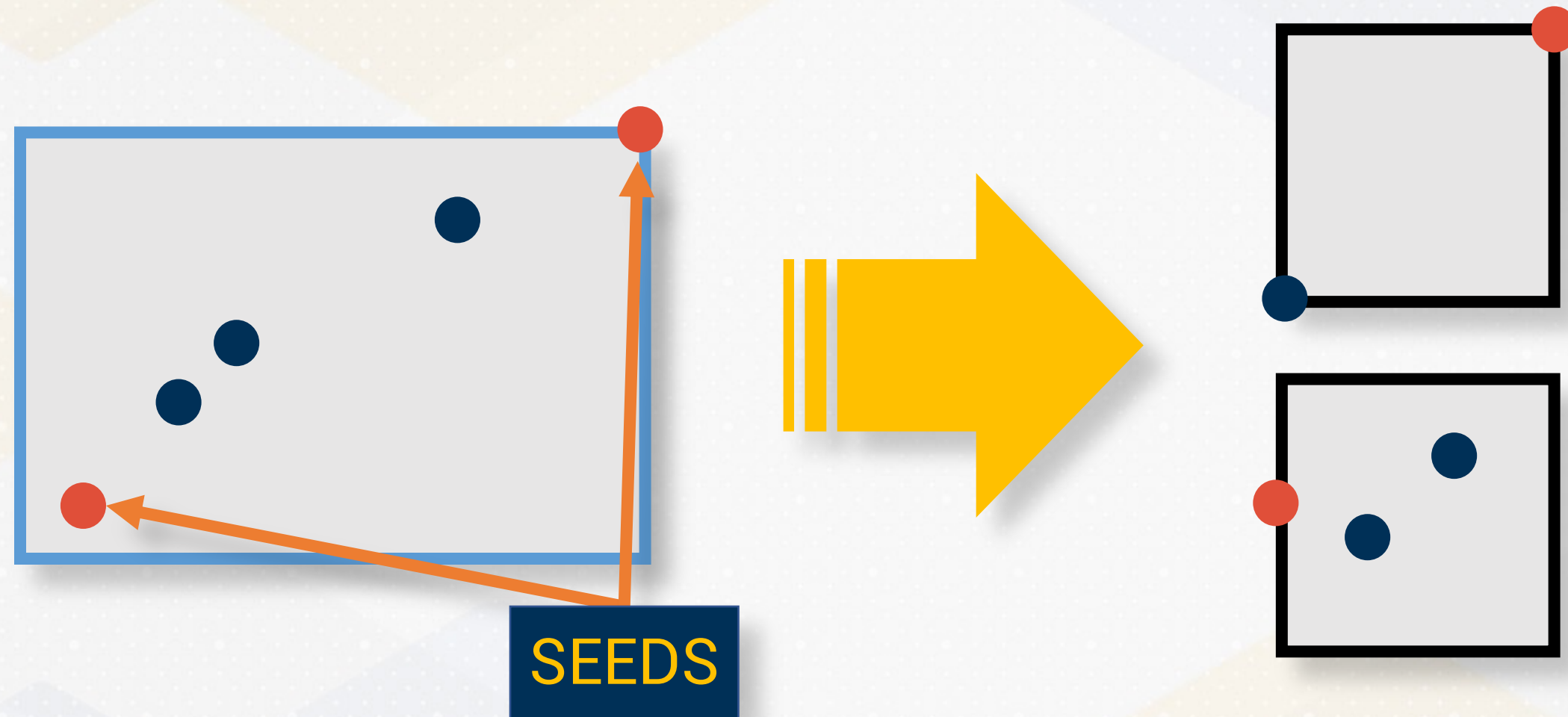


**Leaf
Node**

- Add point to the leaf node
- Split if node exceeds maximum point capacity

```
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  
        }  
    }  
    ...  
}
```

Quadratic Node Splitting Algorithm



Quadratic Node Splitting Algorithm

```
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;  
            }  
        }  
    }  
}
```

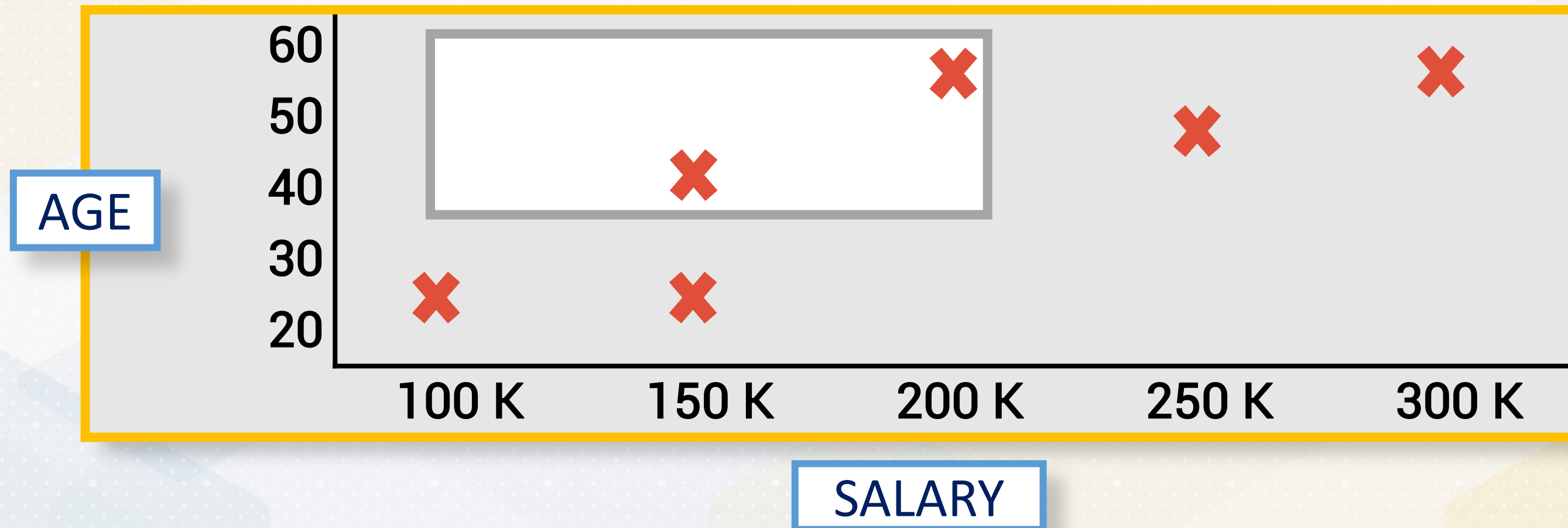

Range & Nearest Neighbor Query



Range Query in R-Tree

$100\text{ K} < \text{Salary} < 200\text{ K}$

$40 < \text{Age} < 60$



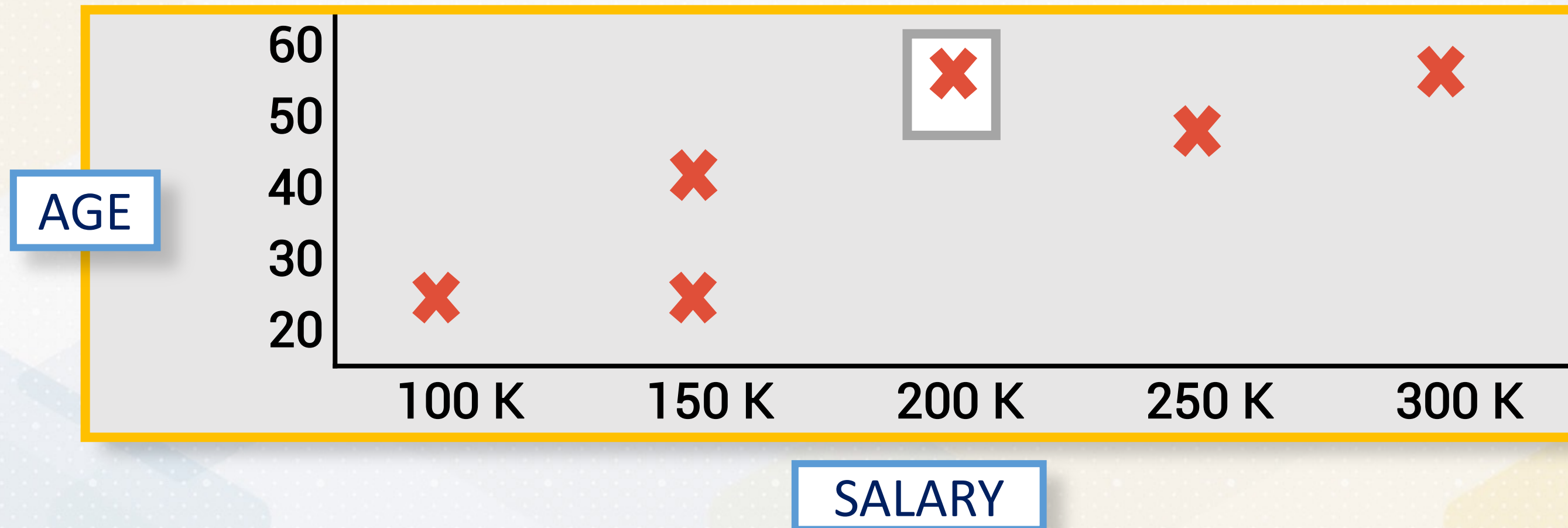
Range Query in R-Tree

```
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 Query in R-Tree

Salary ~ 200 K

Age ~ 50



Nearest Neighbor Query: Inner Node

```
void nearestNeighbor(RTreeNode * node, const Point &queryPoint, int k,
                    std::priority_queue<std::pair<float, Point>> &pq) {

    ...
    else {
        std::vector<std::pair<float, R - TreeNode *>> 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);
        }
    }
}
```

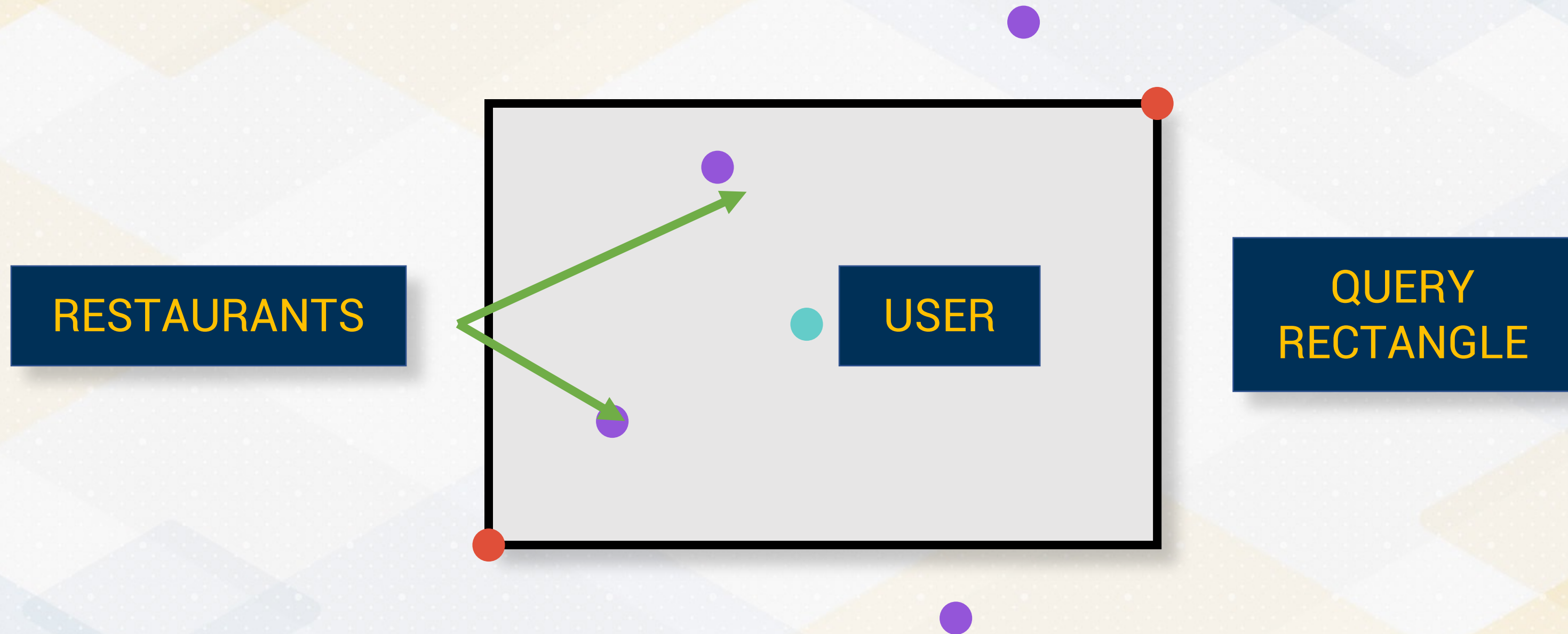

Nearest Neighbor Query: Leaf Node

```
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();
        }
        ...
    }
}
```


Applications



Restaurant Search on Maps



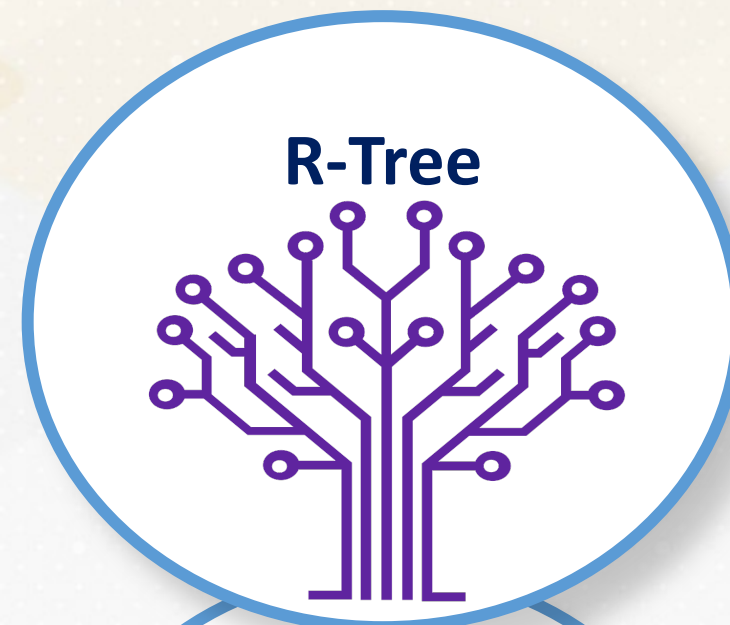
Ride Matching



```
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);
```


Limitation of R-Tree



**Cannot
Support
Higher
Dimensions**



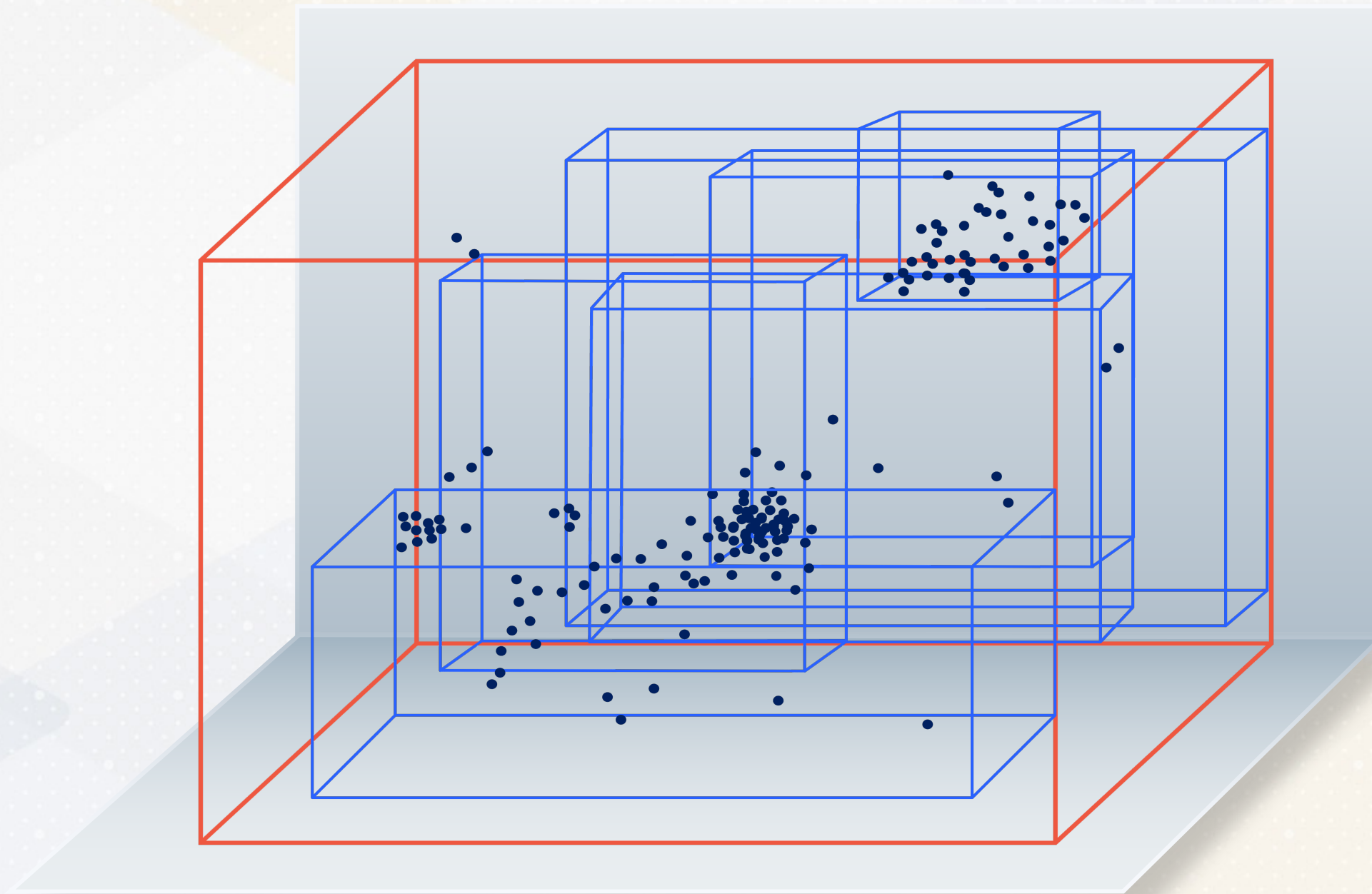
**Higher
Dimensional
Applications**

ND RTree

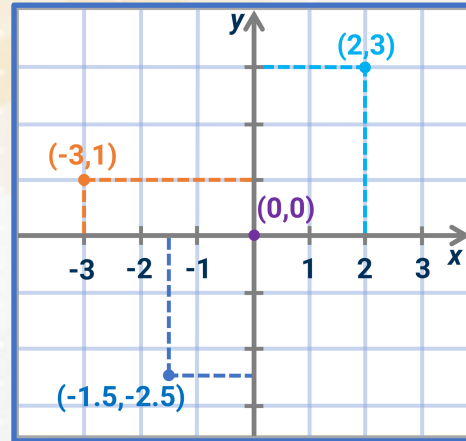


ND R-Tree: A Multidimensional Index

Efficiently manage data in tens of dimensions

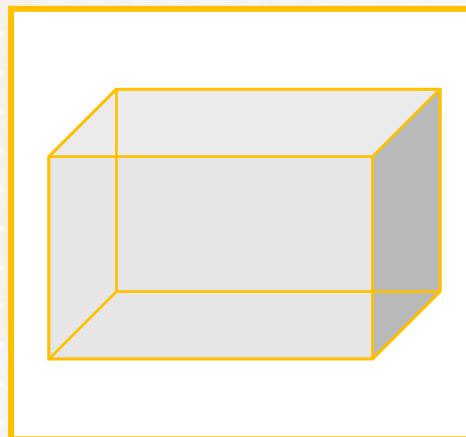


ND R-Tree Structure



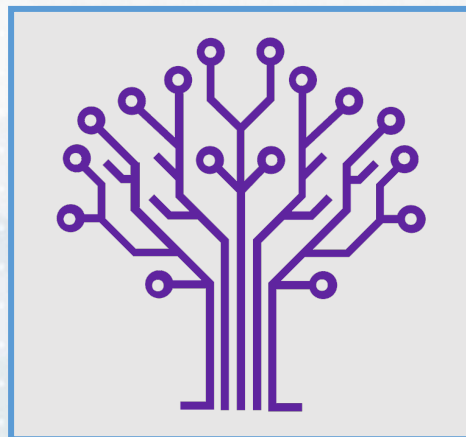
Point

N-dimensional Point



Hyper-
Rectangle

N-dimensional Rectangle



Nodes

Leaf Node

Inner Node

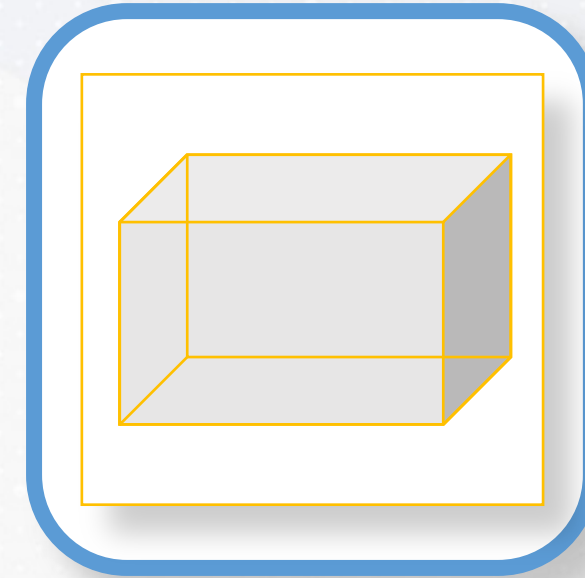
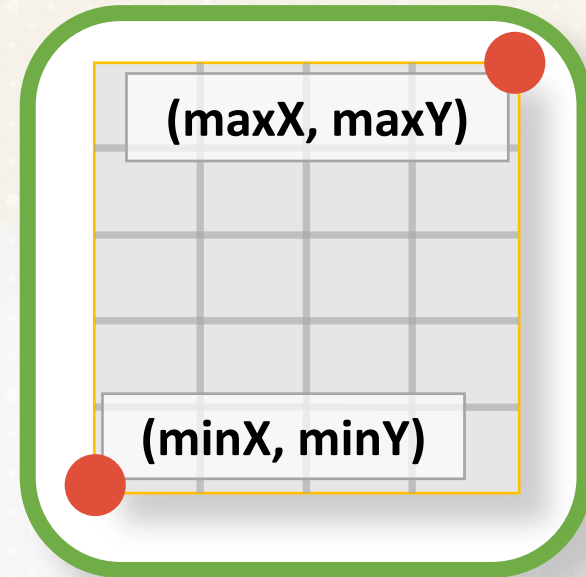
Root Node

N-Dimensional Point

Example: 512-dimensional point with image features

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

N-Dimensional Hyperrectangle



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


Game Engine



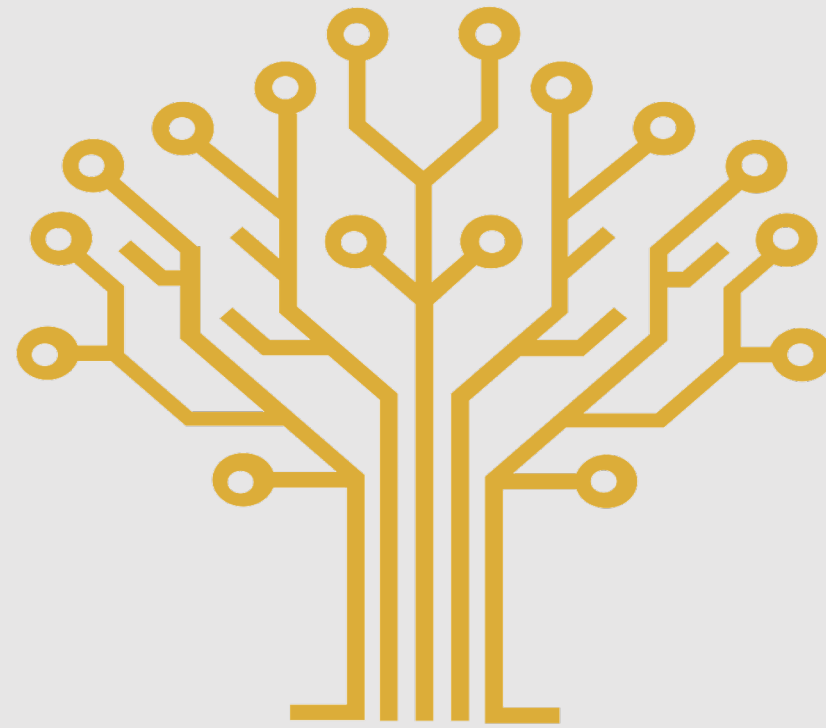
```
Point gameObject({x, y, z}, "GameObjectID");  
tree.insert(gameObject);
```

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

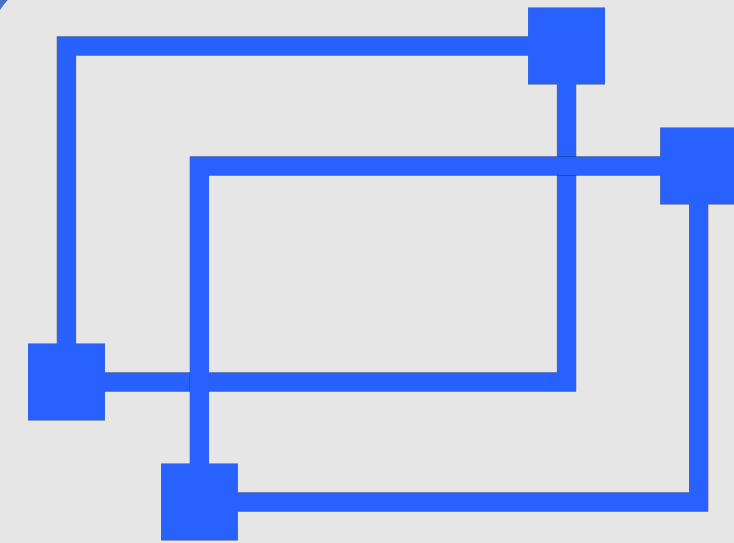
Challenge



Balanced
Efficient Queries



ND R-Tree



Rectangle Overlap
Fewer Subtrees

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

- RTree
- ND-RTree