Lecture 16:
Inverted Index & RTree

Logistics

- Canvas scores have been updated
  - Rough estimate only! Need to curve participation score.
  - More emphasis for exam and extra credit project components.
- Programming assignment 3 (B+Tree) due on **Nov 2**
- Two-page project updates due on **Oct 29** (extra credit)

Recap

- Trie
- Binary Patricia Trie

Lecture Overview

- Inverted Index
- Web-Scale Search
- RTree

Inverted Index

Inverted Index

- Map **words** to the locations in which they appear in a document.

Word	Line ID
russia	10, 20
napolean	40
spoke	15, 20, 30
with	20, 30, 40

Proximity Search

- Find documents where word1 is within k words of word2.

Word	Line ID, Position within Line
russia	(10, 5), (20, 10)
napoleon	(40, 5)
spoke	(15, 5), (20, 15), (30, 5)
with	(20, 10), (30, 10), (40, 5)

Library Search

- Map words to the documents in which they appear.

Word	Book ID, Line ID, Position
russia	(1, 10, 5), (1, 20, 10)
napoleon	(1, 40, 5), (2, 15, 5)
spoke	(1, 15, 5), (1, 20, 15), (1, 30, 5)
with	(1, 20, 10), (1, 30, 10), (1, 40, 5)

Why Inverted Index?

- Unlike a hash table or a B+tree, an inverted index can handle text data and supports operations like phrase queries and proximity searches.
- Internally, inverted index might be built on top of a hash table.

Structure of an Inverted Index

- The index stores terms as keys, and for each term, it stores the document ID(s) where the term occurs and the positions within those documents.

```
“russia" → {docID: [positions]}
```

```
std::unordered_map<std::string,
std::unordered_map<int, std::vector<int>>>
```

- **Key**: Word (string).
- **Value**: A map from document IDs to a list of positions where the term occurs.

Core Functions

addDocument

getDocuments

proximitySearch

Adding Documents to the Inverted Index

- Add the word to the unordered_map, mapping it to the document ID and word position. Convert document content to lowercase if needed.

```
void addDocument(int docID, const std::string &content) {
    documents.push_back(content);
    std::vector<std::string> words = split(toLower(content));
    for (size_t i = 0; i < words.size(); ++i) {
        index[words[i]][docID].push_back(i);
    }
}
```

Retrieving Documents Containing a Term

- Converts the search word to lowercase for case-insensitive matching.
- Returns the list of document IDs as an unordered_set<int>.

```
std::unordered_set<int> getDocuments(const std::string &word) {
    std::string lowerWord = toLower(word);
    if (index.find(lowerWord) != index.end()) {
        std::unordered_set<int> docIDs;
        for (const auto &entry : index[lowerWord]) {
            docIDs.insert(entry.first);
        }
        return docIDs;
    }
    return {};
}
```

Proximity Search: Finding Words Near Each Other

```
std::unordered_set<int> proximitySearch(
    const std::string &word1, const std::string &word2, int k) {
    ...
    if (index.find(lowerWord1) != index.end() && index.find(lowerWord2) != index.end()) {
        for (const auto &entry1 : index[lowerWord1]) {
            int docID = entry1.first;
            if (index[lowerWord2].find(docID) != index[lowerWord2].end()) {
                const auto &positions1 = entry1.second;
                const auto &positions2 = index[lowerWord2].at(docID);
                for (int pos1 : positions1) { for (int pos2 : positions2) {
                        if (std::abs(pos1 - pos2) <= k) { result.insert(docID); break; }
                    }
     ...
    return result;
}
```

Efficiency of Inverted Index

- Fast lookups for individual terms.
- Supports complex queries like proximity search, boolean AND/OR queries, etc.
- Scales well with large document collections (used in search engines).

```
Search for "apple AND orange”:
retrieves documents that contain both terms,
combining the document sets for "apple" and "orange."
```

Challenges with Inverted Index

- Need to store positions for each occurrence of a word.
- Compression techniques (like delta encoding) can reduce storage requirements.
- Disk-based storage for handling very large datasets.

Web Scale Search

Stanford Digital Library Project (1995)

Héctor García-Molina
Larry Page
Sergey Brin

Stanford Digital Library Project (1995)

- Garcia-Molina focused on organizing large digital datasets for efficient retrieval.
- Influenced early web search engine development by advancing information retrieval in distributed systems.
- Advised Larry Page and Sergey Brin, when they were doing their Ph.D. at Stanford, leading to Google.

Web-Scale Search

- While our example operates on a small dataset (e.g., lines from a text file), **web-scale inverted index** is distributed across thousands of servers and handles billions of documents.
- Optimizations include:
- **Sharding** (dividing the index into smaller pieces)
- **Replication** (storing multiple copies of the index) to handle the massive scale of the web.

Distributed Inverted Index

- Word-Based Sharding

Shards (subset of Words)	Replicas
Shard 1 (Words 1…100)	Servers A, B
Shard 2 (Words 101 … 200)	Servers C, D
Shard 3 (Words 201 .. 300)	Servers E, F
…	…

Distributed Inverted Index

- Document-Based Sharding

Shards (subset of Docs)	Replicas
Shard 1 (Docs 1…10)	Servers A, B
Shard 2 (Docs 11 … 20)	Servers C, D
Shard 3 (Docs 21 .. 30)	Servers E, F
…	…

Web-Scale Search

- After retrieving all relevant documents from the inverted index, we can use ranking algorithms to rank documents based on relevance.
- Ranking Algorithms: **PageRank** or **machine learning models** etc.
- Although ranking is not a part of the basic inverted index, the index is the first step in narrowing down the list of potentially relevant documents.

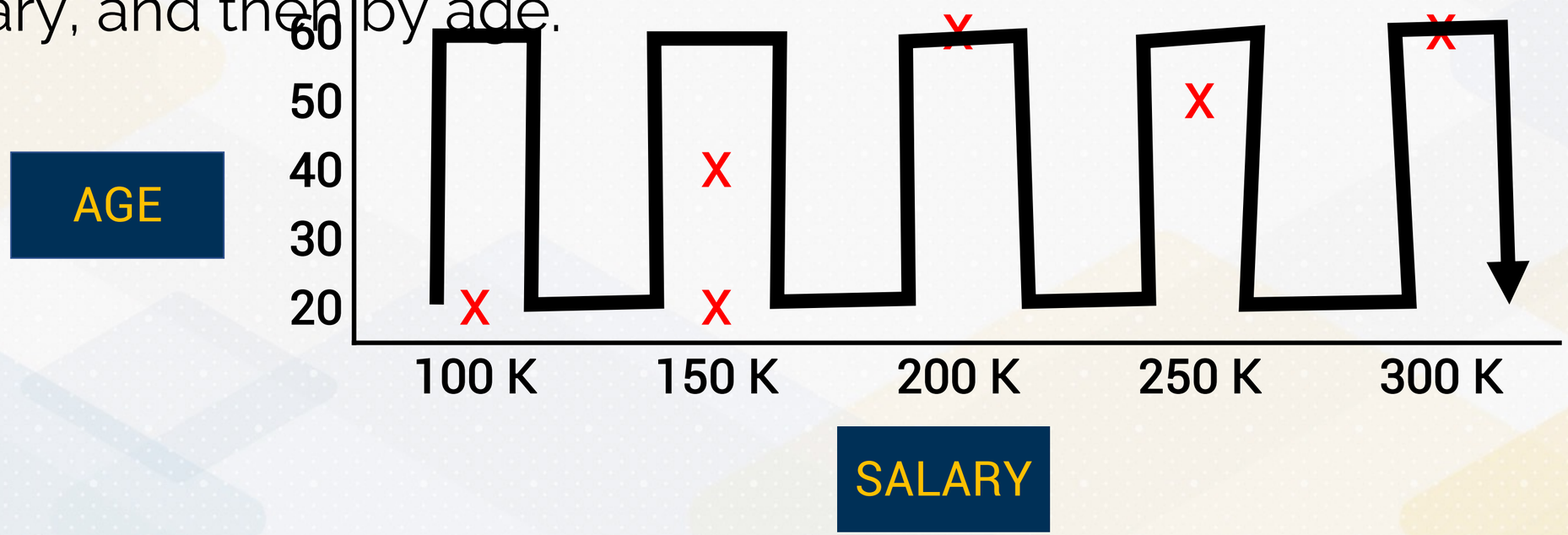
The Future

- The challenge for the next generation is to improve how we **store**, **retrieve**, and **make sense** of the world's data.

RTree

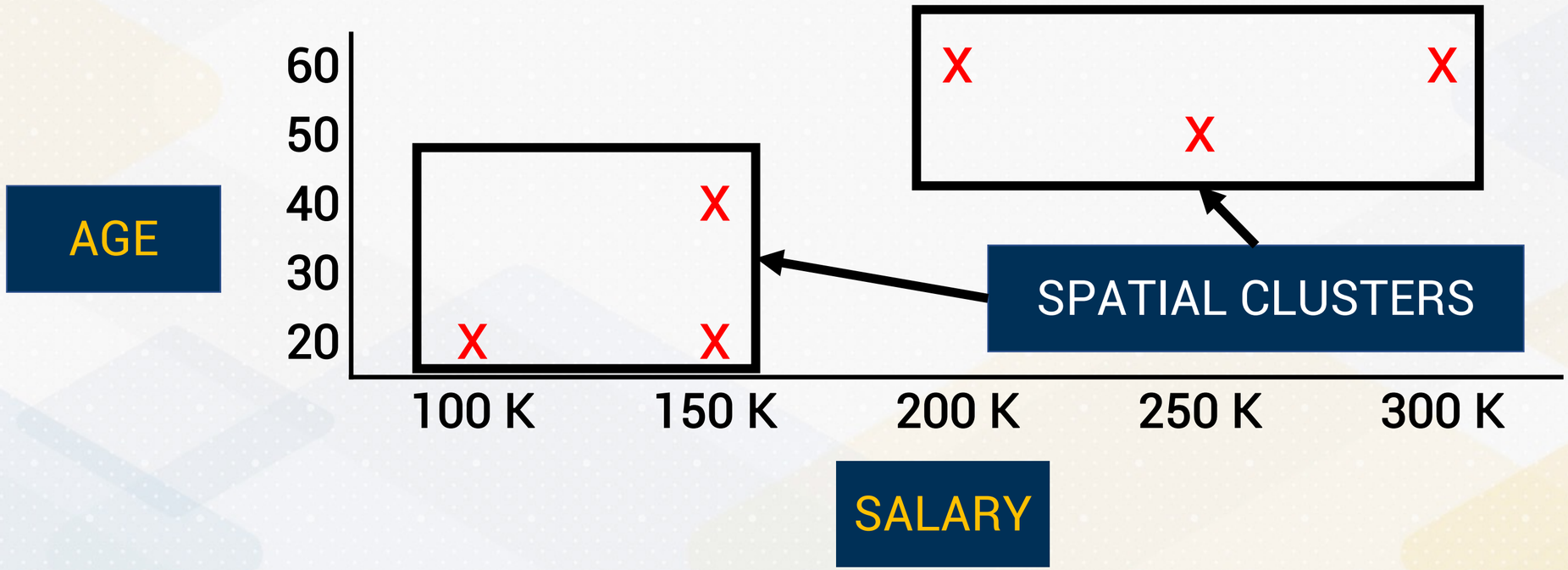
Limitations of B+Tree

- 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).

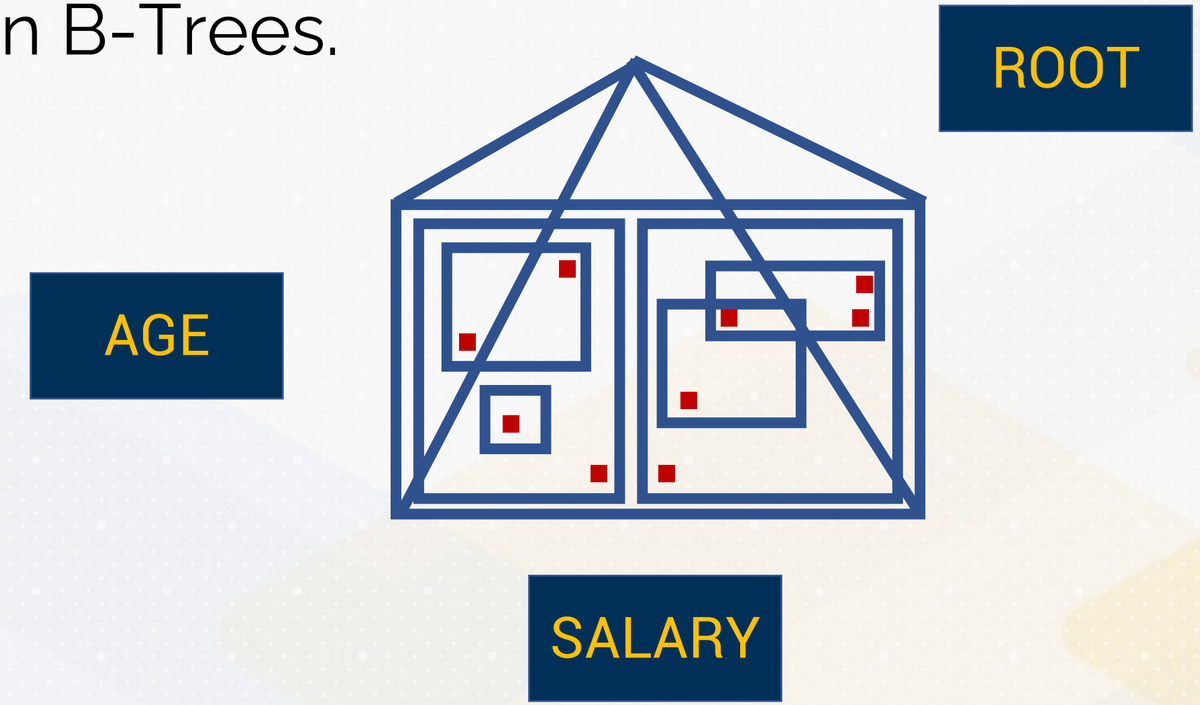
Antonin Guttman (1984)
Berkeley

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

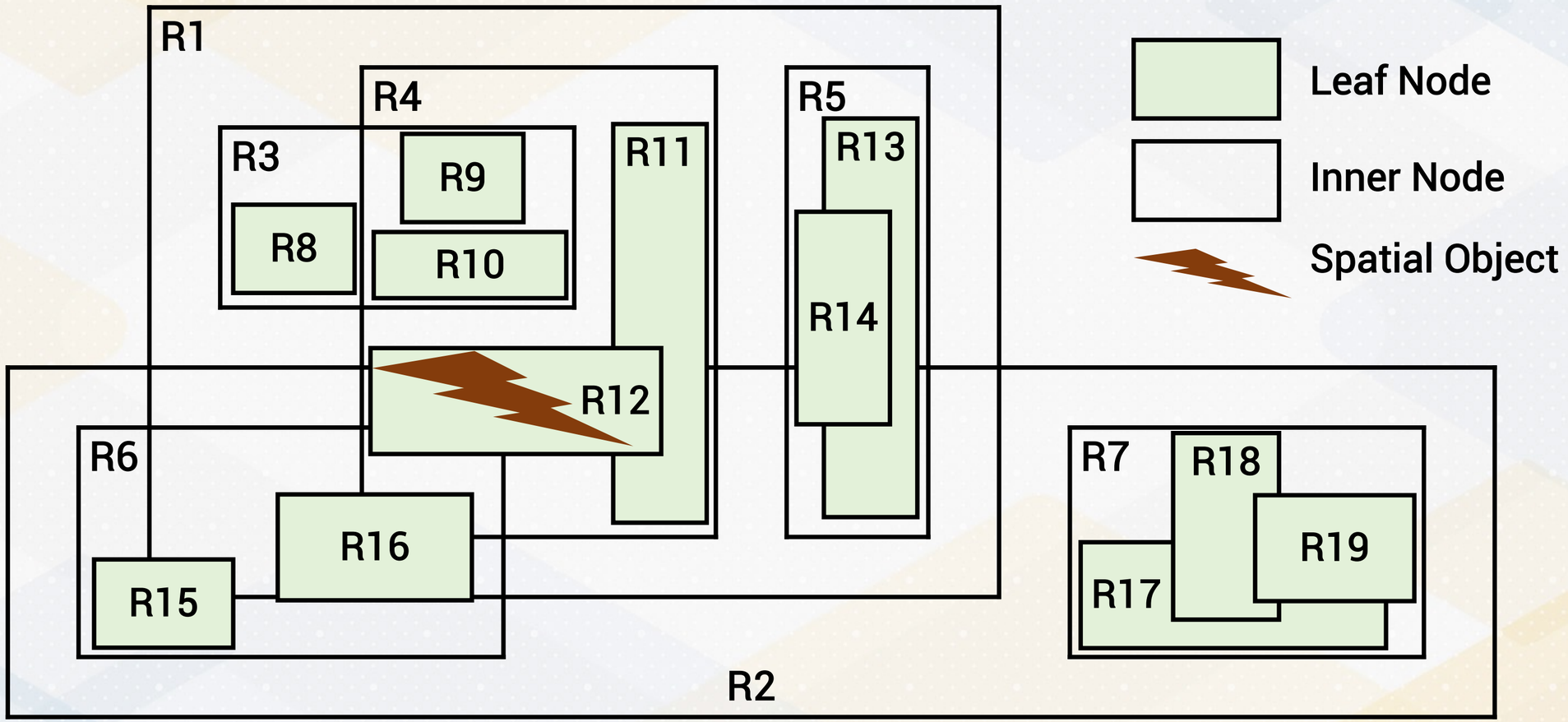
- Like B+ Trees, R-Trees are balanced, for efficient search operations.
- Partitions space using **bounding rectangles** instead of splitting at specific values, as in B-Trees.



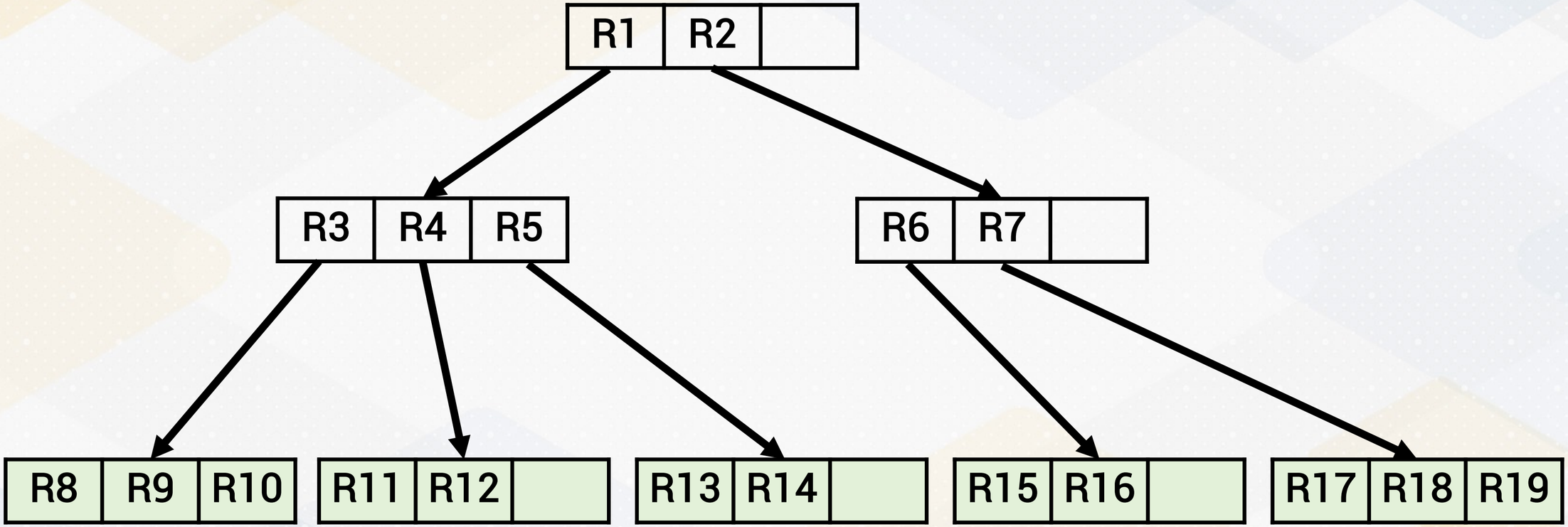
RTree Structure

- **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.

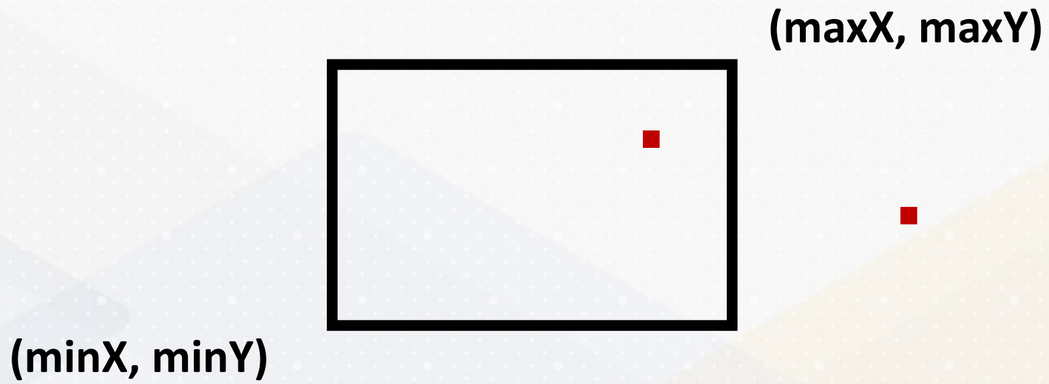
```
struct Rectangle {
    float minX, minY, maxX, maxY;
    Rectangle(float minX, float minY, float maxX, float maxY)
        : minX(minX), minY(minY), maxX(maxX), maxY(maxY) {}

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

Rectangle: Contains

- Check if a point lies inside the rectangle.

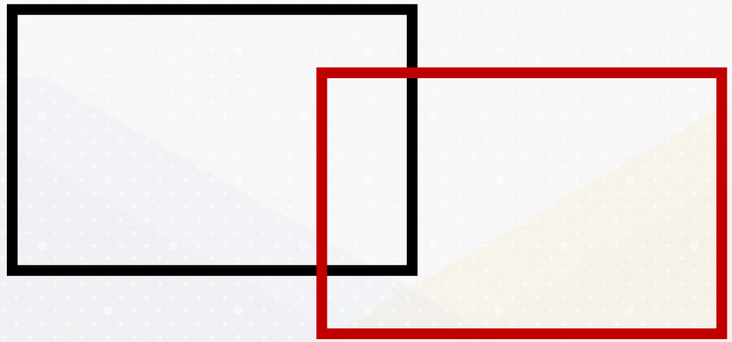
```
bool contains(const Point& p) const {
    return (p.x >= minX && p.x <= maxX && p.y >= minY && p.y <= maxY);
}
```



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

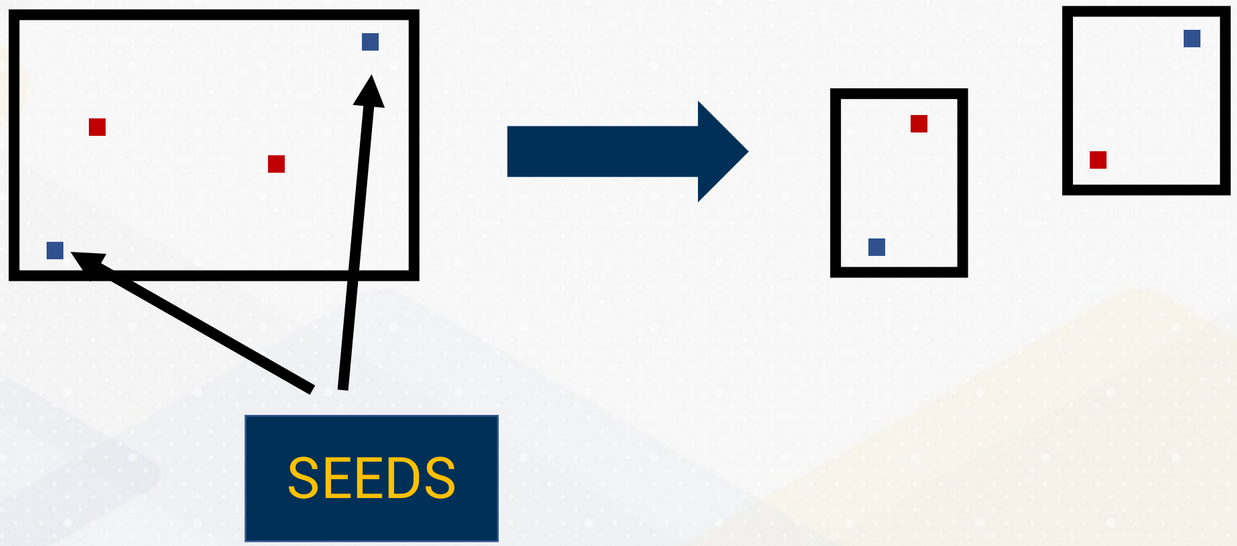
- Insert the point into the leaf node. If the leaf node is full, split the node into two, ensuring the tree stays balanced.

```
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.

Applications of R-Trees

- **Spatial Databases**: Commonly used to index geographical data, including points of interest, maps, and GPS data.
- **GIS Systems**: Geographic Information Systems use R-Trees for querying and managing spatial data (e.g., finding nearby restaurants).

Applications of R-Trees

- **Game Development**: Used to manage objects in large 2D or 3D spaces for collision detection and spatial queries.
- **Ride Sharing**: Applications like Uber use R-Trees to store and query ride locations and find the nearest driver to a user.

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.

Conclusion

- Inverted Index
- Web-Scale Search
- RTree