HLD Quad Trees

Contents

[Quad Trees and Scalable Systems for Nearby Search 2](#_Toc184833299)

[Grids, Population Density, and Quad Trees 4](#_Toc184833300)

[Understanding the Concept 6](#_Toc184833301)

[Quadtree Representation and Storage 8](#_Toc184833302)

[Quadtree Storage 9](#_Toc184833303)

[Quadtree Data Structure for Optimized Place Storage 10](#_Toc184833304)

[Quadtrees and the Algorithm to Find Nearby Places 13](#_Toc184833305)

[Algorithm for Spatial Search Using Cells 16](#_Toc184833306)

[Algorithms for Spatial Search 19](#_Toc184833307)

[Finding the Cell ID Using XY Coordinates 20](#_Toc184833308)

[QuadTree Structure and Traversal 23](#_Toc184833309)

[Binary Search Trees and QuadTrees 25](#_Toc184833310)

[Quadtree Implementation for Location-Based Queries 26](#_Toc184833311)

[Some Calculations 29](#_Toc184833312)

[Quad Tree Time Complexity 31](#_Toc184833313)

## Quad Trees and Scalable Systems for Nearby Search

**Introduction**

* **Problem Statement**: Building a scalable system to find nearby places of interest (e.g., restaurants, gas stations) for a user based on their location.
  + Each point on Earth is defined by its latitude and longitude.
  + The system should handle global queries, ensuring scalability and efficiency.
* **Applications**:
  + Google Maps: Ability to provide localized information (e.g., restaurants near the user) anywhere globally.

**Challenges in Building a Scalable System**

1. **Brute Force Approach**:
   * Compute distances between the user's location and every point of interest.
   * Sort the results to find the nearest locations.
   * **Complexity**: Inefficient for large-scale systems, especially with millions of queries daily.
     + Sorting:
     + Max Heap for k - nearest points:
   * **Conclusion**: Non-scalable and impractical.
2. **SQL Databases with Indexing**:
   * Indexing reduces the number of entries scanned.
   * Still involves scanning a significant portion of data, making it less suitable for high-scale systems.

**Proposed Solution: Grid-Based Division of the World**

1. **Grid Concept**:
   * Divide the world into a grid of rectangles (cells) using latitude and longitude lines.
   * Each cell represents a specific geographical area.
   * Every location in the world is assigned to a grid cell.
   * Properties:
     + **Mutually Exclusive**: No location belongs to more than one grid cell.
     + **Collectively Exhaustive**: Every location is part of a grid cell.
2. **How It Works**:
   * A user’s location is mapped to a specific grid cell based on their latitude and longitude.
   * Instead of comparing the user's location to all places of interest globally:
     + Search within the user's grid cell and adjacent cells for nearby locations.
   * This significantly reduces the number of comparisons and increases efficiency.
3. **Mathematical Principle**:
   * Derived from **Set Theory**:
     + **Mutually Exclusive**: No overlap between grid cells.
     + **Collectively Exhaustive**: Entire space is covered by the grid cells.

**Advantages of Grid-Based Approach**

* **Scalability**:
  + As the number of users or places of interest increases, the system can scale by refining grid resolution or distributing data across servers.
* **Efficiency**:
  + Reduces computational complexity by limiting the search space to a few grid cells.
* **Adaptability**:
  + Can handle millions of queries efficiently, making it suitable for systems like Google Maps.

**High-Level Design Considerations**

* Design for **scalability**:
  + Ensure the solution can handle increasing scale by adding resources (e.g., servers).
* Build a system that minimizes computational costs and maximizes responsiveness.

**Key Takeaways**

1. **Grid-Based World Representation**:
   * Break the world into manageable grid cells.
   * Optimize searches by limiting the area of interest.
2. **Scalable System Design**:
   * A trivial solution may work for small-scale problems but is impractical for global systems.
   * Design with scalability and efficiency as primary goals.
3. **Mathematical Foundations**:
   * Use principles like mutually exclusive and collectively exhaustive regions to divide space logically.

## Grids, Population Density, and Quad Trees

**Key Concepts:**

1. **Representation of Ranges with Brackets:**
   * **Open brackets** ((0, 4)): 0 and 4 are excluded; values in between are included.
   * **Closed brackets** ([0, 4]): 0 and 4 are included along with values in between.
   * Mixed brackets ([0, 4) or (0, 4]): Allow for one boundary to be inclusive and the other exclusive.

**Mapping the Earth into Grid Cells:**

* Earth is a sphere, but maps are 2D projections created through approximations.
* Conversion from 3D sphere to 2D grids involves geometric adjustments:
  + Sphere is "unwrapped" and approximated into a flat, rectangular grid.

**Use of Grid Cells for Location-based Services:**

1. **User's Current Grid Cell:**
   * Identify the grid cell the user is in.
   * Search and sort locations of interest within that grid cell.
   * This reduces the search space significantly compared to searching the entire globe.
2. **Handling Edge Cases:**
   * **Insufficient Results in a Cell:** Extend the search to neighbouring grid cells.
   * **Boundary Cases:** Ensure points near grid edges are not excluded from the search.

**Challenges with Equal-Sized Grids:**

1. **Unequal Population Distribution:**
   * High-density areas (e.g., Bangladesh, parts of India): Small cells may contain many places of interest.
   * Low-density areas (e.g., deserts, oceans): Large cells may contain no places of interest.
2. **Efficiency Concerns:**
   * **Big grid cells:**
     + High-density regions result in slow queries as there are too many points to compare.
   * **Small grid cells:**
     + Low-density regions result in 0 results, wasting computational resources.

**Why Equal-Sized Grids are Inefficient:**

1. **Densely Populated Areas:**
   * Small cells are needed to manage high data density efficiently.
   * Large cells slow down queries due to excessive data points.
2. **Sparsely Populated Areas:**
   * Small cells often contain no data, wasting computational effort.
3. **Inconsistent Query Performance:**
   * Users in densely populated areas experience slower queries.
   * Users in sparsely populated areas experience faster queries.

**Introduction to Quad Trees:**

* **Solution to Unequal Grids:**
  + Adaptive data structures like **quad trees** divide regions dynamically based on data density.
  + Quad trees address issues with fixed grid sizes by:
    - Splitting large cells in dense regions into smaller cells.
    - Keeping cells larger in sparsely populated areas.
  + Ensures balanced performance across different regions of the globe.

## Understanding the Concept

**Introduction**

* **Quadtrees** are a data structure used to manage spatial data efficiently.
* The **main idea**: Divide the underlying world (or space) into **grid cells** of **variable sizes**, unlike uniform grids.
* Useful for applications where spatial data is not evenly distributed.

**Why Quadtree?**

* **Challenge with uniform grids**: Equal-sized grid cells may not adequately represent varying densities of data.
* **Solution**: Variable-sized grid cells to manage different densities efficiently.
* The word "quad" indicates the division of space into **4 parts** recursively.

**Intuitive Concept of Quadtrees**

1. **Start with the entire world as a single grid cell**:
   * Treat the whole space as **one problem**.
   * Ask: *Is the data manageable within a threshold (e.g., 100 places)?*
2. **If the threshold is exceeded**, divide the grid cell into 4 equal subregions.
3. **Recursively repeat**:
   * For each subregion, ask the same question.
   * Divide again if the threshold is exceeded.
4. **Stop dividing** when the subregion contains data within the threshold.

**Connection with Recursion**

* **Recursive Approach**:
  + Break down a larger problem into smaller problems.
  + Solve each smaller problem in the same manner.
  + Representation: A **tree structure** with recursive subproblems.

**Steps to Build a Quadtree**

1. Start with the **entire world as one grid cell**.
2. Check if the data exceeds the threshold.
3. If yes:
   * Divide the cell into **4 equal subregions**.
   * Ask the same question for each subregion.
4. Continue dividing until each subregion has data within the threshold.
5. **Leaf nodes** of the tree represent regions where no further division is needed.

**Key Insights**

1. **Dense regions**:
   * Require more divisions to meet the threshold.
   * Result in deeper branches in the tree.
2. **Sparse regions**:
   * Require fewer divisions.
   * Result in shallower branches in the tree.
3. **Example**:
   * In densely populated areas (e.g., cities), grid cells become smaller.
   * In sparsely populated areas (e.g., oceans), grid cells remain larger.

**Visualization of the Process**

1. Represent the **world as a tree**:
   * Root: Entire world.
   * Internal nodes: Subregions created during division.
   * Leaves: Final subregions meeting the threshold.
2. Tree depth varies:
   * **Deeper branches** for high-density areas.
   * **Shallower branches** for low-density areas.

**Applications of Quadtrees**

* Efficiently managing spatial data in:
  + Geographical information systems (GIS).
  + Computer graphics.
  + Spatial indexing in databases.
* Handling unevenly distributed data efficiently.

**Conclusion**

* **Quadtree** is a powerful data structure for organizing spatial data.
* By leveraging **variable grid sizes**, it adapts to the density of data efficiently.
* Its recursive approach and tree representation make it intuitive and effective for a variety of applications.

## Quadtree Representation and Storage

**Introduction to Quadtrees**

1. **Definition**:
   * A quadtree divides a rectangular space into four quadrants recursively.
   * Each quadrant can further be divided into smaller quadrants until a base condition is met.
2. **Recursive Depth**:
   * Some areas (e.g., landmass) might be divided deeply, resulting in many recursive calls.
   * Other areas (e.g., oceans) might stop earlier due to fewer divisions.
3. **Why Quadtrees?**
   * Dividing a rectangle into 4 smaller rectangles is intuitive as it reduces both dimensions (X and Y) proportionally.
   * Alternative methods (e.g., dividing into 3 parts or 2 parts) might not utilize both dimensions efficiently.

**Threshold Configuration**

* The threshold for division (e.g., 100, 500, or 1000 places) determines when the division stops.
* This threshold is configurable and application-specific.

## Quadtree Storage

1. **Representation**:
   * Each cell in the quadtree is uniquely identified by a **Cell ID**.
   * Each cell's boundaries are defined using **4 coordinates (A, B, C, D)**, represented as latitudes and longitudes.
2. **Practical Storage**:
   * Stored as a **table**:
     + **Cell ID**: Unique identifier for each cell.
     + **Coordinates (A, B, C, D)**: Define the cell's boundaries in 2D space (latitude and longitude).

A screenshot of a computer

Description automatically generated

**Storing Places**

1. **Details for Each Place**:
   * **Place ID**: Unique identifier for each place.
   * **Coordinates (X, Y)**: Represented as latitudes and longitudes.
   * **Name**: Name of the place.
   * **Type**: Metadata (e.g., restaurant, salon).
   * **Cell ID**: Links the place to its corresponding cell (foreign key relationship).

A screenshot of a computer

Description automatically generated

1. **Data Structure**:
   * Two separate tables:
     1. **Quadtree Representation Table**:
        + Contains Cell IDs and their 4 boundary coordinates.
     2. **Places Table**:
        + Contains Place IDs, coordinates, names, types, and the corresponding Cell ID.

## Quadtree Data Structure for Optimized Place Storage

**Problem Statement**

* There are millions of places worldwide.
* The challenge is to efficiently find places of interest nearby for a user query.
* The goal is to design a scalable and efficient system to handle these queries.

**Solution Approach**

* **Static Quadtree Creation**:
  + The quadtree is precomputed as part of system initialization (boot-up).
  + This approach does not rely on any initial user query.
  + The entire universe of places is organized into a quadtree upfront.
* **Quadtree Basics**:
  + The world is divided recursively into quadrants (cells).
  + Each cell holds up to a predefined threshold (e.g., 100 places).
  + Recursive division continues until each cell has fewer than the threshold number of places.

**Key Features of the Quadtree**

1. **Cell Structure**:
   * **Leaf Nodes**:
     + Represent the smallest unit of division in the quadtree.
     + Actual cells containing places (final subdivisions).
   * **Non-Leaf Nodes**:
     + Represent logical larger cells.
     + Contain multiple smaller leaf cells.
2. **Cell ID Assignment**:
   * Only **leaf nodes** are assigned unique cell IDs.
   * Non-leaf nodes are logical groupings and do not receive cell IDs.
3. **Representation of Places**:
   * Places are assumed to be points for simplicity.
   * For large places (e.g., parks), they can be represented as multiple points.
4. **Variable Cell Sizes**:
   * Regions with dense places will have smaller cells.
   * Sparse regions will have larger cells.
   * The quadtree adapts to the distribution of places, leading to efficient query processing.

**Practical Implementation**

1. **Static Representation**:
   * The current implementation is static, meaning the structure does not change dynamically.
   * It is designed for efficient querying from a precomputed state.
2. **Dynamic Representation**:
   * The teacher mentioned the possibility of making the quadtree dynamic in the future.
   * This would involve updating the quadtree as the distribution of places changes over time.

**Threshold for Cell Division**

* The threshold is based on the overall number of places, not the type of place.
* Example:
  + If the threshold is 100, cells will be divided until each has fewer than 100 places of all types combined.
  + Adjusting the threshold is a domain-specific decision.

**Applications and Examples**

1. **Large Places Representation**:
   * Large places like parks or railway stations can be split into multiple points for storage and querying.
   * Example: Cab services like Ola or Uber show multiple pickup points within a railway station.
2. **Place Types and Threshold**:
   * Types (e.g., restaurant, bus stop) do not determine cell division.
   * Threshold applies to the total number of places irrespective of type.

**Quadtree Structure Summary**

* **Root Node**:
  + Represents the entire world; not a leaf node.
  + Does not receive a cell ID.
* **Leaf Nodes**:
  + Represent actual cells with fewer than the threshold number of places.
  + Assigned unique cell IDs.
* **Non-Leaf Nodes**:
  + Logical groupings of multiple leaf cells.
  + Used to organize and structure the quadtree.

## Quadtrees and the Algorithm to Find Nearby Places

**Concept of Quadtrees**

1. **Quadtree Representation**:
   * A quadtree divides a 2D space into four quadrants (cells).
   * Each cell is uniquely identified by a Cell ID.
   * Every cell can be a **leaf** or **non-leaf**:
     + **Leaf cells**: Contain actual places of interest (e.g., a barber shop).
     + **Non-leaf cells**: Do not store places but are used for hierarchy.
2. **Additional Attributes in Representation**:
   * **Is Leaf**: Indicates whether the cell is a leaf.
   * **Parent Cell ID**: Each cell has exactly one parent, creating a recursive tree structure.

**Understanding the Algorithm**

The goal is to find nearby places efficiently based on user input.

**Input Parameters**

1. User’s location:
   * X-coordinate
   * Y-coordinate
2. **Type of place** (e.g., temple, school).
3. **Limit**: Maximum number of results to fetch.

**Steps of the Algorithm**

**Initial Algorithm**

1. **Find the user's leaf cell ID**:
   * Use a find\_grid function that maps the user's (X, Y) location to a specific leaf cell ID.
2. **Query places in the leaf cell**:
   * Retrieve a list of places from the places table where cell ID = user’s cell ID and type = user’s requested type.
   * Use indexing on cell ID and type to make this query efficient.
3. **Sort and return the results**:
   * Sort the list of places by distance from the user.
   * Include additional sorting parameters like popularity or other business logic.

**Challenges in Initial Algorithm**

1. **Insufficient Results**:
   * A single cell may not contain enough places of interest, especially if:
     + The cell’s density is low for the requested type.
2. **Boundary Issues**:
   * If the user is near the boundary of a cell, relevant places in neighboring cells might be excluded.

**Updated Algorithm**

To address the challenges:

1. **Find user’s leaf cell ID**:
   * Same as step 1 in the initial algorithm.
2. **Identify neighbouring cell IDs**:
   * Find neighbouring cells around the user’s current cell.
   * This step expands the search space slightly to include adjacent areas.
3. **Query places in user’s cell and neighbours**:
   * Retrieve places from the places table for all relevant cells (current cell + neighbours).
4. **Sort and return results**:
   * Combine places from all queried cells.
   * Sort by:
     + Distance
     + Popularity
     + Other business logic as needed.

**Key Observations**

1. **Efficiency**:
   * The search space remains optimal, focusing only on 2-5 nearby cells instead of the entire dataset.
2. **Scalability**:
   * Quadtrees inherently reduce the search space, making this approach scalable for large datasets.
3. **Flexibility**:
   * The algorithm adapts to user needs by dynamically adjusting the search area based on location and query parameters.

**Practical Considerations**

1. **Indexing**:
   * Proper indexing of places table on cell ID and type is crucial for query speed.
2. **Heuristic for Neighbours**:
   * The number of neighbouring cells to include can be fine-tuned based on:
     + Cell size.
     + Density of places.
     + Expected query load.
3. **Sorting Criteria**:
   * Business logic can prioritize certain attributes (e.g., popularity over distance).
4. **Function Implementation**:
   * find\_grid: Converts (X, Y) to a leaf cell ID.
   * Neighbour identification logic: Determines adjacent cells to include in the query.

## Algorithm for Spatial Search Using Cells

**Key Concepts**

1. **Cell Structure**:
   * Cells are defined in 2D space using their boundaries (latitude and longitude coordinates).
   * Each cell has a unique CellID and contains places of interest.
2. **Places of Interest**:
   * Each place has its own longitude and latitude, which are stored in the database.
   * A cell can hold up to 100 places of interest.
   * If the number of places exceeds 100, the cell is divided into smaller cells.
3. **Leaf Cells and Parent Cells**:
   * Leaf cells are the smallest units in the cell hierarchy.
   * Each cell knows its parent cell via a ParentCellID relationship.
4. **Searching in Cells**:
   * Start by identifying the cell where the user is located (leaf cell).
   * Check neighboring cells for places of interest if necessary.

**Algorithm Details**

1. **Finding Leaf Cell**:
   * Identify the user's cell based on their current coordinates.
   * The algorithm determines the leaf cell containing the user.
2. **Finding Neighbouring Cells**:
   * Neighbourhood cells are identified around the user’s cell.
   * These cells are also queried for places of interest if the user’s cell has less than 100 places.
3. **Pagination or Incremental Search**:
   * If no places are found in the user's cell and its neighbours, expand the search to more cells incrementally.
   * This can be done by running successive queries with a larger search area.

**Data Handling and Queries**

1. **Initial Cell Division**:
   * All places are initially placed in a single large cell.
   * A query is run to check the number of places in the cell.
   * If the count exceeds 100, the cell is divided, and places are reallocated to the new cells.
2. **SQL Queries**:
   * The queries involve sorting and filtering places based on their proximity to the user's location.
   * Neighbourhood cells are included in the search for broader coverage.

**Key Points from Discussion**

1. **Cell Size vs. Place Count**:
   * The physical size of a cell does not determine the number of places within it.
   * Even a large cell can have fewer than 100 places.
2. **Optimality**:
   * Searching in multiple cells (e.g., 6 cells with 600 places) is still computationally efficient compared to searching the entire dataset of 100 million places.
3. **Edge Cases**:
   * In areas like the Pacific Ocean, where no places of interest exist, the search algorithm gracefully handles the lack of results by expanding the search area or returning an empty result set.
4. **Business Use Case**:
   * The specific implementation of search (e.g., how many cells to check, incremental expansion of the search area) depends on the business requirements.

**Next Steps in Learning**

1. Understanding the two core parts of the algorithm:
   * How to find the leaf cell of a user based on their XY coordinates.
   * How to find neighbouring cells efficiently.
2. Implementation of the cell division process:
   * Maintaining a list of places and their cell assignments.
   * Dividing cells dynamically as needed.

**Additional Questions Discussed**

* **Do we store the longitude and latitude of each place or only the cell?**
  + Longitude and latitude are stored for each place in the places table.
* **How can we track the number of places in each cell?**
  + A list is maintained to keep track of the number of places in each cell.
* **What happens if the threshold is exceeded in a cell?**
  + The cell is divided, and places are redistributed among the new cells.

## Algorithms for Spatial Search

The lecture addresses two main algorithms for spatial search problems:

1. **Finding the CellID of a User Given Their XY Coordinates**.
2. **Identifying the Neighborhoods of a Particular Cell**.

**Algorithms for Spatial Search Using Cells - Connecting with Binary Search Trees**

**Key Discussion Points**

1. **Problem Context**:
   * The teacher connects the concept of spatial cell-based search with the structure and algorithmic principles of a **Binary Search Tree (BST)**.
   * The task is to identify how to determine:
     + The **CellID** for a user’s XY coordinates.
     + The **neighborhood cells** for a given cell.
2. **Understanding Binary Search Trees**:
   * A BST is a tree data structure where each node satisfies:
     + The left child is smaller than the parent node.
     + The right child is larger than the parent node.
   * The given example tree (visualized) is a valid BST

A diagram of a diagram

Description automatically generated

1. **Key Algorithm for Searching in BST**:
   * To find a specific value (e.g., 12):
     + Start at the root (e.g., 10).
     + Compare the target value (12) with the current node:
       - If smaller, move to the left child.
       - If larger, move to the right child.
     + Repeat this process until the value is found or the subtree is null.
   * **Complexity**:
     + Logarithmic time complexity: , where N is the number of nodes.
     + Each comparison takes .
2. **Connection to Spatial Search**:
   * The algorithm for finding a **CellID** for a user's coordinates can be analogized to searching in a BST:
     + Use boundaries or spatial partitions similar to how a BST uses node values to guide searches.
     + At each level, decide whether to "go left or right" (e.g., determine which subregion contains the point).

## Finding the Cell ID Using XY Coordinates

**Key Concepts**

The lecture discusses how to use a quadtree-like structure for mapping XY coordinates to a specific cell in a hierarchical grid. This involves subdividing a parent cell into four quadrants (child cells) and recursively narrowing down the search to identify the correct cell.

**1. Root Cell Definition**

* The spatial area is represented as a single root cell.
* **Root Cell Coordinates**:
  + Bottom-left: (0, 0)
  + Bottom-right: (10K, 0)
  + Top-left: (0, 10K)
  + Top-right: (10K, 10K)

A graph with numbers and symbols

Description automatically generated with medium confidence

**2. Subdivision of the Root Cell**

Each cell (starting with the root) is subdivided into four quadrants:

1. **Midpoints of Edges**:
   * Midpoint of the left edge: (0,5K)
   * Midpoint of the bottom edge: (5K,0)
   * Midpoint of the top edge: (5K,10K)
   * Midpoint of the right edge: (10K,5K)
2. **Central Point**:
   * Midpoint of the cell (intersection of diagonals): (5K,5K)

**3. Recursive Search Algorithm**

**Objective**: Determine the quadrant containing the given XY coordinates.

**Steps**:

1. Start at the **root cell** with the known coordinates.
2. Compare the XY coordinates of the user with the boundaries of the cell:
   * Determine if the point lies in the **top-left, top-right, bottom-left, or bottom-right** quadrant.
   * For example:
     + If , move to the **top-left** quadrant.
     + If , move to the **top-right** quadrant.
     + If , move to the **bottom-left** quadrant.
     + If , move to the **bottom-right** quadrant.
3. **Recurse**:
   * Use the new quadrant as the current cell.
   * Compute its boundaries using the midpoints and repeat the comparison.
4. **Stop at a Leaf Node**:
   * The search ends when the smallest possible cell containing the point is reached (a leaf node).

**4. Relationship to Binary Search**

* Similar to binary search:
  + A decision at each step narrows down the search space.
  + Instead of splitting into 2 parts (left/right), the quadtree splits into **4 parts** (quadrants).
  + Time complexity remains logarithmic with respect to the number of subdivisions: .

**5. General Formula**

Given the boundaries of a parent cell, the coordinates of its four children can be derived:

* **Top-left child**:
  + Bottom-left: Parent's bottom-left.
  + Top-right: Parent’s midpoint.
* **Top-right child**:
  + Bottom-left: Midpoint (x-axis of Parent's right).
  + Top-right: Parent’s top-right.
* Similar logic applies to the bottom-left and bottom-right children.

**6. Practical Implementation**

* **Input**: XY coordinates of a user.
* **Output**: Cell ID corresponding to the user's location.
* The algorithm uses a systematic comparison with midpoints to guide the search and identifies the cell ID based on the user's final location.

## QuadTree Structure and Traversal

**Overview**

* A **QuadTree** is a tree data structure where each node has exactly four children.
* The root node represents the entire space and is recursively divided into four equal rectangular regions.
* Each rectangle is represented by its coordinates (top-left, bottom-right, etc.).

**Step-by-Step Process**

1. **Root Node Initialization**:
   * The root represents the entire rectangular space.
   * For example: Root coordinates: (0,0), (10,0), (10,10), (0,10).
2. **Dividing the Space**:
   * The root rectangle is divided into 4 equal sub-regions:
     + **1st Child** (top-left): (0,10), (5,10), (5,5), (0,5)
     + **2nd Child** (top-right): (5,10), (10,10), (10,5), (5,5)
     + **3rd Child** (bottom-right): (5,5), (10,5), (10,0), (5,0)
     + **4th Child** (bottom-left): (0,5), (5,5), (5,0), (0,0)
3. **Recursive Division**:
   * Each child region is further subdivided into 4 regions.
   * The subdivision continues until reaching a desired granularity or "leaf cells."

A diagram of a rectangular object with arrows pointing at the top

Description automatically generated with medium confidence**Finding a Point in the QuadTree**

* **Example Query**: Find the location of the point (2, 7).
  + Start at the root node and determine the sub-region:
    - (2, 7) lies in the **1st Child** since it satisfies the bounds of the top-left rectangle.
  + Move recursively to the corresponding child node and repeat:
    - Check which sub-region within the 1st child contains (2, 7).
  + Continue this process until reaching a **leaf cell**.

**Key Observations**

* **Recursive Nature**: The algorithm relies on recursively narrowing down the search area.
* **Boundary Conditions**:
  + Ensure that the coordinates of the point are compared against the bounds of the current rectangle at each level.
* **Leaf Cell Identification**:
  + The smallest unit (leaf cell) will provide the unique ID for the region containing the point.

## Binary Search Trees and QuadTrees

**Binary Search Tree (BST) - Time Complexity**

1. **Search Time Complexity**:
   * At each step:
     + **Comparison**: Spend **O(1)** time to compare.
     + **Halving Problem Space**: At every step, reject half the problem space.
   * This leads to **log₂(N)** steps, where the base is 2 (as the problem is halved at each step).
   * **Time Complexity**: **O(log₂(N))**.

**QuadTree - Search Time Complexity**

1. **Search in QuadTrees**:
   * At each step:
     + **Comparison**: Spend **O(1)** time to compare.
     + **Quartering Problem Space**: The problem space is reduced to **1/4** at every step (not 1/2 like in a BST).
   * This leads to **log₄(N)** steps, where the base is 4 (as the problem is quartered at each step).
   * **Time Complexity**: **O(log₄(N))**.
2. **Interpretation**:
   * QuadTree searches are slightly faster in terms of depth, as the problem space reduces more rapidly compared to a BST.

**Finding a Cell ID in a QuadTree**

* **Objective**: Determine the **leaf cell ID** where a specific object (e.g., user or point) resides.
* Steps:
  + At each step, compare coordinates to decide which quadrant to move into.
  + Continue recursively until the leaf cell containing the object is identified.
  + **Time Complexity**: **O(log₄(N))** due to quartering of space at each step.

**Neighbourhood Identification in QuadTrees**

1. **Objective**: Find the neighbouring cells of a given cell.
2. **Methods**:
   * **Tree Traversal Approach**:
     + Use tree-based algorithms, such as finding the next node in the same level.
     + Analogous to the "next node problem," where nodes are connected as a linked list at the same level.
   * **Parent-Child Relationships**:
     + Leverage the parent cell ID to find siblings and neighbouring cells.
     + Check parent of the parent if neighbours span across multiple parent nodes.
3. **Concepts Discussed**:
   * Neighbouring cells may not always share the same parent.
   * Check the hierarchy (parent, grandparent) to determine adjacency.
   * Flexibility in approach based on problem requirements.

## Quadtree Implementation for Location-Based Queries

**Key Concepts:**

1. **Quadtree Representation:**
   * Quadtree is a hierarchical data structure used to represent spatial data.
   * Each node represents a cell with four child nodes (subdivisions).
   * Each cell stores:
     + Four coordinates (boundaries).
     + A parent cell ID.
     + Child cell IDs (if not a leaf).
2. **Basic Queries:**
   * **Find Nearby Places:**
     + Input: Latitude and Longitude (lat-long).
     + Traverse from the root node, checking which child cell contains the point.
     + Use coordinates to determine the appropriate child node in logarithmic steps.
     + Continue until reaching a leaf node or the relevant cell.
3. **Recursive Quadtree Traversal:**
   * Start at the root cell.
   * For each node:
     + Fetch its coordinates and child IDs.
     + Compare the point with child cell boundaries to identify the relevant child.
   * Repeat until the point lies within a specific leaf cell.
   * Complexity: O(log⁡4N)O(\log\_4 N)O(log4​N) for a tree with NNN cells.
4. **Dynamic Nature of Places:**
   * Places (restaurants, landmarks, etc.) are not static; they can be added or removed.
   * Two essential operations:
     + **Find Nearby Places:** Leverages the quadtree for quick location-based queries.
     + **Add New Place:** Updates the quadtree dynamically.

**Handling Dynamic Updates:**

1. **Adding a New Place:**
   * Input: Coordinates (x, y), type (e.g., temple), and metadata.
   * Process:
     1. Use a helper function (find\_grid) to locate the relevant cell.
     2. Check if the cell's capacity (e.g., 100 places) is exceeded.
     3. If capacity is exceeded:
        + **Divide the cell** into four smaller cells (subdivide).
        + Redistribute existing places among the new child cells based on their locations.
        + Add the new place to the appropriate child cell.
     4. If capacity is not exceeded:
        + Simply add the new place to the cell.
   * Update the "places table" with the new information.
2. **Dividing Cells:**
   * When splitting a cell:
     1. Create four new cells (child nodes).
     2. Assign places from the parent cell to its children based on their locations.
     3. No guarantee of equal distribution (not necessarily 25-25-25-25).
3. **Updating the Quadtree:**
   * Reflect changes in the database (e.g., updated child IDs, parent-child relationships).
   * Adjustments ensure the structure remains efficient for subsequent queries.

**Points of Emphasis:**

1. **Precomputation:**
   * Initial quadtree construction involves precomputing cell boundaries, child relationships, and metadata.
   * Serves as the foundation for all subsequent operations.
2. **Efficiency:**
   * Finding nearby places relies on the logarithmic complexity of quadtree traversal.
   * Each query involves a constant number of database point queries at each level.
3. **Dynamic Queries:**
   * The system must handle dynamic changes (e.g., new places, closed places) efficiently to maintain accurate results.
4. **Threshold-Based Division:**
   * The threshold for splitting a cell is critical (e.g., 100 places).
   * Ensures quadtree remains balanced and efficient.

**Steps for Query and Update Operations:**

1. **Find Nearby Places:**
   * Traverse the quadtree from the root.
   * Use coordinate comparisons to identify the relevant child cell.
   * Continue recursively until the correct cell is located.
2. **Add New Place:**
   * Use find\_grid to locate the cell for the new place.
   * Check if the cell exceeds its capacity:
     + If yes, subdivide and redistribute.
     + If no, add the place directly.
   * Update the database and quadtree structure accordingly.

## Some Calculations

**Step 1: Understanding the structure**

You are working with a quadtree structure where each leaf node holds some number of places (locations). The number of leaf nodes depends on how many places are allocated per leaf node. You are making calculations based on the worst-case scenario (every leaf node holds only one place), the best-case scenario (every leaf node holds the maximum of 100 places), and the average-case scenario (each node holds 20 places).

**Step 2: Worst-case calculation (1 place per node)**

In the worst case:

* You have 100 million places to allocate, and each leaf node holds just one place.
* The number of leaf nodes will be 100 million (one node per place).

Now, for each non-leaf level of the quadtree:

* At the second-to-last level, the number of nodes is .
* At the third-to-last level, the number of nodes is
* This process continues, reducing the number of nodes by a factor of 4 for each level until reaching the root node.

The total number of nodes can be calculated using the formula for the sum of a geometric series:

where:

* ‘a’ is the first term (in this case, 100 million),
* ‘r’ is the common ratio (here, ​),
* ‘n’ is the number of terms.

This gives you an approximation of around 133 million nodes.

**Step 3: Average-case calculation (20 places per node)**

If each node holds 20 places, the number of leaf nodes would be:

Now, applying the same reduction process, the number of nodes at each level will decrease by a factor of 4, and the total number of nodes would be around 6.5 million.

**Step 4: Storage requirement per node**

Each node requires:

* 8 bytes for the cell ID,
* 64 bytes for the coordinates (4 coordinates, each 4 bytes),
* 1 byte for the leaf status (Boolean),
* 8 bytes for the parent cell ID,
* 32 bytes for child cell IDs (4 children, each 8 bytes).

So, the total storage required per node is:

**Step 5: Total storage calculation**

With 6.5 million nodes, each taking 113 bytes:

**Step 6: Conclusion**

So, the total storage required for the quadtree structure with 6.5 million nodes, each holding 113 bytes, would be approximately **734.5 MB**.

This is the total storage required based on the average-case assumption where each leaf node holds 20 places.

## Quad Tree Time Complexity

1. **Topic Overview**:
   * The lecture discusses the time complexity associated with the **quad tree** data structure.
   * Quad trees are used for spatial indexing, particularly in problems involving large datasets or multidimensional spaces.
2. **Number of Nodes**:
   * The total number of nodes in the dataset is given as **6.5 million** (or ).
   * The concept being discussed is how to handle such large datasets efficiently with the help of the quad tree.
3. **Time Complexity**:
   * The time complexity for the quad tree is **logarithmic**.
   * The specific base used in the logarithmic calculation is **4**.
   * This is because, at every step, the problem is divided into **4 parts** (i.e., a spatial division into 4 quadrants). Out of these 4, only one part is chosen for further exploration, discarding the other 3.
   * This logarithmic behaviour indicates that the time complexity of finding or processing a specific node in a quad tree is relatively efficient.
4. **Logarithmic Calculation**:
   * To calculate the time complexity, the formula being considered is:
   * **Logarithmic base 4** is being used here because of the 4-way partitioning at each step in the quad tree.
5. **Understanding Logarithmic Growth**:
   * The teacher points out that logarithmic functions grow very slowly, even for large numbers like .
   * In terms of steps, the number of steps required to reach the solution will be **small** (less than 100 steps).
   * For example, the logarithm of **6.5 million** in base 2 (i.e., ) is roughly around **20** steps, but with base 4, the number of steps will be even fewer.
6. **Efficiency**:
   * This logarithmic growth in time complexity means that even with very large datasets (like **6.5 million** nodes), the number of steps needed to process the data remains small and manageable.
   * The teacher highlights that this efficient time complexity is one of the reasons quad trees are effective for large-scale problems, as they allow for quick identification and retrieval.
7. **Storage Complexity**:
   * Quad trees are also efficient in terms of **storage complexity**, meaning they don’t require excessive memory to store all the nodes. The structure is designed to minimize the need for storage, as it discards parts of the data during each step.
8. **Conclusion**:
   * The lecture concludes by emphasizing that the combination of **small time complexity** and **efficient storage** makes the quad tree an ideal solution for spatial indexing and other applications involving large datasets.

**Key Takeaways:**

* **Quad tree time complexity** is **logarithmic with base 4**.
* With **6.5 million nodes**, the number of steps needed to find a specific node or solution is **less than 100**.
* **Logarithmic growth** means that even with large datasets, the time required remains efficient.
* **Storage complexity** is low, allowing the entire dataset to be stored efficiently.
* This efficiency makes quad trees an excellent data structure for spatial indexing and large-scale problems.