DATABASE MANAGEMENT SYSTEMS LABORATORY

CS39202

Term Project

High Dimensional Indexing

Project Report

Team: The SQL Squad

Roopak Priydarshi (20CS30042)
Monish Natarajan (20CS30033)
Gaurav Malakar (20CS10029)
Abhijeet Singh (20CS30001)
Gopal (20CS30021)

➢ Project Code

https://github.com/gXurav07/Indexing-with-R-Tree

➢ Introduction

The objective of this project is to create an efficient method for storing and retrieving data points that exist in a high-dimensional space, like images and audio files that can be represented as high dimensional vectors. The focus will be on indexing and searching images. Apart from that this project also involves extracting features from images and building a User Interface using python.

➢ Objectives

- Implement the R-tree algorithm from scratch, including methods for inserting, deleting, and searching data points.
- Select a high-dimensional dataset, such as image or audio data, to index and search.
- Develop an efficient indexing system that can store and retrieve data points efficiently.
- Use the R-tree index to perform efficient searches on the dataset.
- Evaluate the performance of the R-Tree indexing system on different datasets and compare it with other indexing systems.

➢ Project Methodology

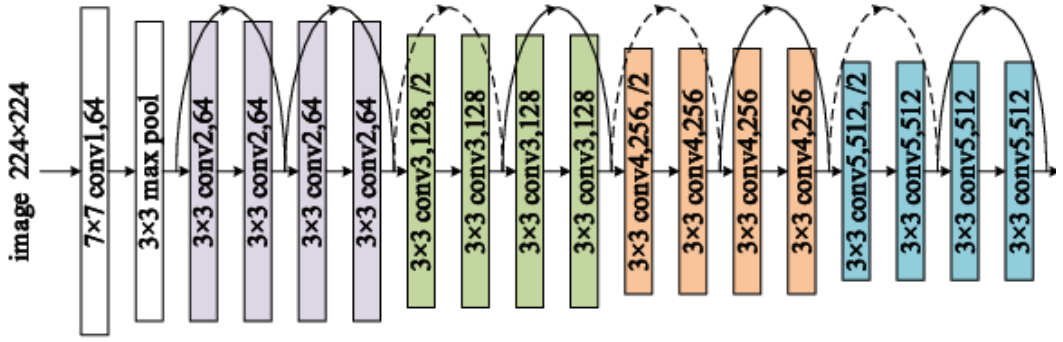
- We are implementing a DL based model for extracting useful features from an image and representing it as a high dimensional vector. We have used ResNet-18 as our DL based model for feature extraction and the complete process has been described below in detail.

  Feature Extraction

  We perform feature extraction using a ResNet-18 model pre-trained on ImageNet. Input images are resized to 224x224 and passed through the ResNet backbone to obtain a 1000 dimensional feature vector which is used for R-tree indexing.

  Steps involved in feature extraction:

  - Load the ResNet-18 pre-trained model from a deep learning library such as PyTorch or TensorFlow.
  - Preprocess the input image to match the format expected by the ResNet-18 model. This typically involves resizing the image to a fixed size, converting it to a tensor, and normalizing the pixel values.
  - Pass the preprocessed image through the ResNet-18 model to obtain the feature map output.
  - Flatten the feature map output into a 1D vector of features.
  - We can apply dimensionality reduction techniques such as principal component analysis (PCA) or t-distributed stochastic neighbor embedding (t-SNE) to reduce the dimensionality of the feature vector.



ResNet-18 Model Architecture

- To write functions for **insertion**, **deletion** and **searching the nearest neighbors** (closest matching images) in the R-tree.

Insertion

- For inserting in a R-tree involves first the minimum bounding rectangle (MBR) is found that encloses the object being inserted and then adding this MBR to the index structure.
- To do this, the R-tree searches the index structure starting from the root node, and then selects the node with the smallest increase in its MBR size after the new MBR is added. If there is a tie, the node with the smallest area is chosen. If a non-leaf node is chosen, the search continues recursively in the chosen node. If a leaf node is chosen, the new MBR is inserted in the node.
- However, the insertion may cause some nodes to become overfull, which means that they contain more than the maximum number of entries allowed by the node capacity. To handle this, the R-tree splits the overfull node into two nodes, reassigns the entries between the two new nodes, and propagates the split to the parent node.
- This process continues recursively until a non-overfull node is reached or the root node is split. If the root node is split, a new root node is created, and the height of the tree is increased by one. Finally, the index statistics, such as the MBRs of the nodes, are updated to reflect the new structure of the index.

Deletion

- Start at the root node and traverse the tree to find the node containing the entry to be deleted.
- Remove the entry from the node.
- If the node is a leaf node and has fewer than the minimum number of entries, remove the node from the tree.
- If the node is not a leaf node and has fewer than the minimum number of entries, check its sibling nodes to see if they have more than the minimum number of entries.
- If a sibling node has more than the minimum number of entries, redistribute the entries between the two nodes so that they both have at least the minimum number of entries.

- If no sibling node has more than the minimum number of entries, merge the node with one of its sibling nodes, creating a new node that contains all of the entries from both nodes.

Searching the nearest neighbors

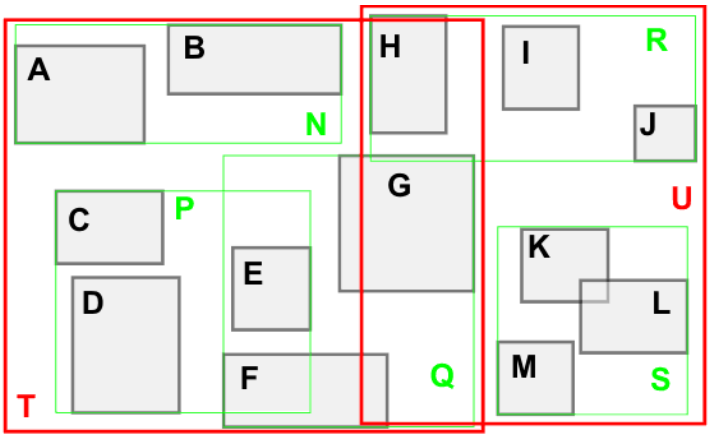
- Start at the root node of the R-tree.
- Traverse the R-tree to find the leaf node(s) that contain the query point.
- For each leaf node found in step 2, compute the distance between the query point and each object (rectangle) in the leaf node.
- Sort the objects in each leaf node by distance to the query point.
- Starting with the leaf node closest to the query point, search for the nearest neighbors in a depth-first manner by recursively traversing the R-tree.
- During the recursive traversal, keep track of the minimum distance found so far and update it if a closer neighbor is found.
- Stop the search when the desired number of nearest neighbors have been found or when all nodes have been visited
- Code snippet:

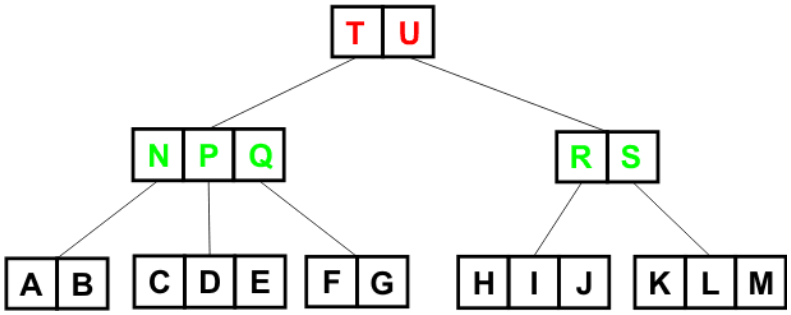
```
def nearest(node, point, k, heap):
    if node.is_leaf():
        for obj in node.objects:
            distance = compute_distance(obj, point)
            heap.push(distance, obj)
        while len(heap) > k:
            heap.pop()
    else:
        for child in node.children:
            distance = compute_distance(child.rectangle, point)
            if distance < heap.max_distance():
                nearest(child, point, k, heap)
```

Explanation of Code Snippet:

- In this code, node is the current node being visited, point is the query point, k is the number of nearest neighbors to find, and heap is a priority queue that stores the k nearest neighbors found so far. compute_distance is a function that computes the distance between two rectangles or between a rectangle and a point. The function first checks if the current node is a leaf node. If it is, it computes the distance between the query point and each object in the leaf node and adds it to the priority queue. If the priority queue has more than k elements, it removes the farthest element. If the current node is not a leaf node, it checks if the distance between the query point and the bounding rectangle of the child node is less than the maximum distance in the priority queue. If it is, it recursively visits the child node.

➢ R-Tree for a two-dimensional data after insertion





➢ R-Tree in optimizing high dimensional data search

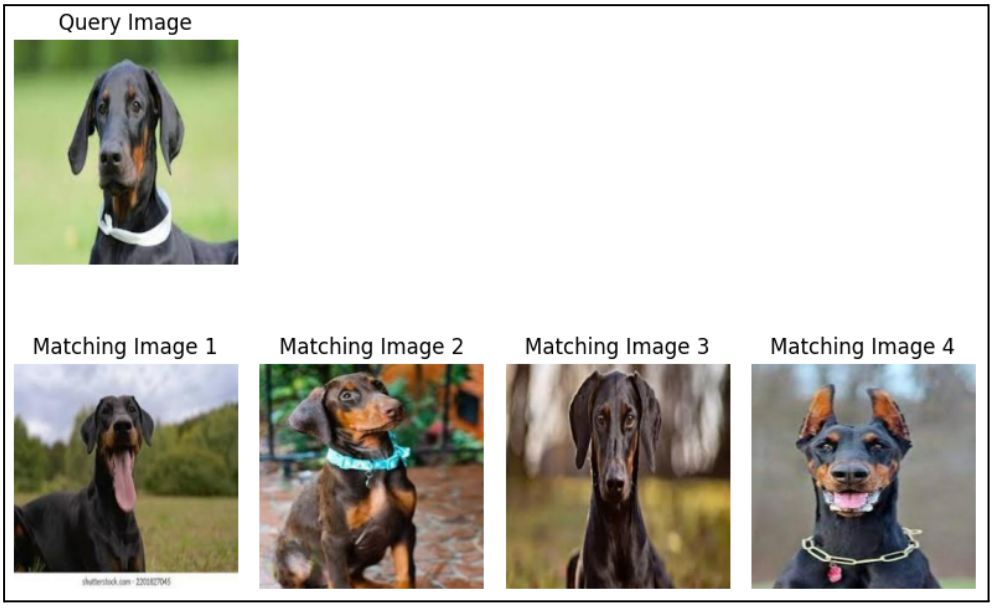
- R-tree optimizes the process of searching high-dimensional data by partitioning the data space into a hierarchy of rectangular regions called "bounding boxes" or "minimum bounding rectangles" (MBRs). This structure is commonly used in applications where objects have a position or location in space, and queries often involve specifying a region in space and finding all objects that intersect that region.
- R-trees organize nearby objects in the same minimum bounding rectangles (MBRs), making searches more efficient by reducing the number of MBRs to be examined. The search starts at the root node and recursively traverses the tree, examining each MBR to determine if it intersects the search region. If an MBR intersects the search region, the search continues recursively into the MBR's sub-tree. By organizing the data in this way, R-tree can quickly narrow down the search space to a subset of MBRs that are likely to contain the desired data, and then perform a more detailed search within that subset.
- This makes R-trees especially useful for high-dimensional data since the number of dimensions greatly affects the search time. By grouping nearby objects together, R-trees can perform efficient searches even in high-dimensional spaces. Overall, R-trees minimize the number of MBRs that need to be examined, resulting in faster search times and making them a popular data structure for spatial data.
- In the average case, the time complexity of an R-tree search is proportional to the number of MBRs that need to be examined, which can be expressed as O(N^(1/p) + M), where N is the number of data objects, p is the dimensionality of the data, and M is the number of MBRs that intersect the search region. This assumes that the data is uniformly distributed and that the R-tree is well-balanced.

➢ R Tree's Effectiveness over other Methods

- **Grid-based methods:** Grid-based methods divide the space into a grid and perform searches based on the grid cells. Grid-based methods are simple and fast, but they may not be suitable for datasets with varying density. R-tree is generally considered to perform better than grid-based methods for datasets with varying density.
- **Quadtree:** Quadtree is a hierarchical data structure that recursively divides a region into four equal quadrants. Quadtree is useful for finding all points within a given distance of a query point. R-tree is generally faster than quadtree for range queries and point-in-polygon queries.
- **KD-tree**: kd-tree is a binary tree that divides the space into halves along the median of the data points. KD-tree is useful for nearest neighbor searches. R-tree is generally slower than kd-tree for nearest neighbor searches. R-tree's partitioning is based on areas, making it more efficient than kd-tree, which divides the data by individual points. This makes R-tree better suited for range queries and point-in-polygon queries.

➢ Results

We inserted an image dataset (7000 animal images having 4 classes: Elephant, Dog, Cat, Tiger) into our R-tree. Here are the query results containing 4 most similar images in order of relevance (Note that the query images were not part of the indexed images):



Query Image

Matching Image 1

Matching Image 2

Matching Image 3

Matching Image 4

Query Image

Matching Image 1

Matching Image 2

Matching Image 3

Matching Image 4

Query Image

Matching Image 1

Matching Image 2

Matching Image 3

Matching Image 4

➢ Conclusion

The implementation of the R-Tree algorithm for high-dimensional indexing is a valuable project that provides a solid understanding of how high-dimensional data can be indexed and searched efficiently. The project provides valuable experience in Python programming, data indexing, and performance evaluation. By indexing a high-dimensional dataset, such as audio or image data, the project provides a practical example of how the R-Tree/KD-Tree algorithm can be used to index and search real data.

➢ Scope of Improvement

- Try better feature extractors such as RESNET-101 and RESNET-152 which are heavy weight but will work better with high computing capacities.
- Make data specific optimizations improvements in the insertion, deletion and search functions of R Tree.
- Incorporate this into a web based app to improve the user experience.
- Take a larger dataset for more useful applications.

➢ References

There are some links attached below of the research papers that we referred to for getting a better understanding of the problem statement and to learn about the ways to implement this task.

- Image retrieval system using R-tree self-organizing map: https://www.researchgate.net/publication/222877896_Image_retrieval_system_using_R-tree_self-organizing_map
- An efficient content based image retrieval method for retrieving images http://www.ijicic.org/ijicic-10-11046.pdf
- R-tree - Wikipedia.” *Wikipedia*, Wikimedia Foundation, 16 Apr. 2023, https://en.wikipedia.org/wiki/R-tree
- He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016. https://arxiv.org/abs/1512.03385

----------:THE END:----------