Assignment 3

Due date: August 23, 2020, 11:55pm IST

1 Introduction

In this project, you will implement R-tree data structure on top of the buffer manager provided to you. Although we will be providing pseudo codes of various functions to you, we strongly suggest you read about R-tree data structure in detail before starting your assignment. You will be implementing the following functions of R-trees:

1. **Insert**: Given a d-dimensional integer vector, insert the vector to the tree structure.
2. **Bulk Load**: Given a pre-sorted list of vectors, bulk-load the vectors and construct the R-tree.
3. **Point Search**: Given a d-dimensional vector, find if it is present in the R-tree.

Feel free to go use the following links to learn about R-trees:

1. R-Trees Original Paper: http://www-db.deis.unibo.it/courses/SI-LS/papers/Gut84.pdf
2. Simpler Paper: `http://www.bowdoin.edu/~ltoma/teaching/cs340/spring08/Papers/Rtree-chap1.pdf`
3. STR Bulk Load: https://www.researchgate.net/publication/3686660_STR_A_Simple_and_Efficient_Algorithm_for_R-Tree_Packing

Unlike your most data structures assignments, this assignment is a different in the following factors:

- You can no longer assume that your data is in memory.
- Access to the data is only through a very specific API, provided to you by our implementation of a buffer manager.

1.1 Configuration

1. `g++ compiler version`: 8.2.1. Requires C++ 11 standard.
2. We have tested on: Ubuntu 16.04 LTS
3. Your code should correctly compile and run on: Ubuntu 16.04 LTS
4. Only the standard C++ libraries (including STL) are allowed.

2 File Manager

The buffer manager implementation is available at `https://github.com/ankit-1517/dbms_rTree`. It consists of two main files: `buffer_manager.h/cpp` and `file_manager.h/cpp` along with a bunch of supporting functions in various other files (look at the documentation for details). In this Section, we will briefly describe the functionality provided in the `file_manager`.
Important note: Your access to the data file is *solely* through the functions provided by the `file_manager`. *Do not* directly use any functions from `buffer_manager`. Some useful constants are defined in the file `constants.h`.

2.1 Structure of the data file

Since we have not implemented a separate record manager, it is up to you to implement one if you like. Note that the only record type we have is integers. The contents of a page are defined by the following parameters:

1. **Size of the page**: The constant `PAGE_CONTENT_SIZE` defined in `constants.h`.
2. **Occupancy fraction**: Occupancy fraction decides how much of a page is occupied and how much is left empty[1]. By default, this is 1, which means that all available space is packed with integers.

Further, any file that is supplied as a test case or that you create as the output of your code has to strictly adhere to the following format.

1. Integers occupy `sizeof(int)` space. This is typically 4 bytes in most systems.
2. Integers are always packed from the beginning of the page. Therefore, the empty space is contiguous and starts after the last integer in the page until the end of the page.
3. If a page contains vector of d-dimensional points, the space occupied by each point is $d * sizeof(int)$. No point is split between two pages, if you need to split a point, it is simply written to the next page.
4. So if a data structure uses s bytes of space, the page can contain only $floor(PAGE\_CONTENT\_SIZE/s)$ number of nodes of the data structure.

3 File manager API

A typical usage of the file manager API can be found in `sample_run.cpp`. Please pay attention to how integers are stored and retrieved from a page, since the same procedure has to be followed in order to generate output files from your programs. Alternate ways of storing integers will result in erroneous reads from our testing software.

1. Go through the File Manager API in detail and understand the use of functions such as `MarkDirty()` and `UnpinPage()`. These functions are essential to ensure that the buffer manager is able to evict pages to make way for new ones. Note that a page that is read is automatically pinned.
2. Go through the `errors.h` file to understand what kind of errors may be thrown. *Do not make changes to this file.* But you may need to make use of these errors in your own `try-catch` blocks.
3. Go through the `constants.h` file. The two main constants that are of interest here are `BUFFER_SIZE` which denotes the number of buffers available in memory and `PAGE_SIZE` which in turn determines the `PAGE_CONTENT_SIZE`. All these constants may be change to test your code.

4 Structure of a Node

In this experiment, assume that the constructed tree can be arbitrarily large, so you need to store it in disk. A node can have maxCap number of children. In a standard tree structure, you store a *pointer* to the children in the node. But as you can't directly store a pointer to the disk, you would store the *unique identifier(ID)* of the children nodes in the current node. For a given node, you will store the following entries:

1. Store the identifier of the node as an int
2. Store the Maximum Bounding Region (MBR) of the node as 2*d ints
3. Store the ID of the parent of current node
4. For each child e of the node,
    - Store the MBR of the child in 2*d ints
    - Store the ID of the child

[1] Recall that a B+-tree typically leaves a part of its nodes empty. We are trying to emulate a similar behaviour.

If the node has less than maxCap children, store the rest as INT_MINs to maintain uniformity and keep the size of a node constant. If the node is a leaf node, you can store the MBR as d number of INT_MINs and the d-dimensional point and the ID of children as 0 or -1.

Do note, that if the size of a node is S, the page can contain M = floor(PAGE_CONTENT_SIZE / S) number of nodes. So a node with ID i should be present in page number i/M. We strongly recommend you to implement your assignment without changing the node structure. However, if you feel you can modify the structure to make the implementation more efficient, you can do so later, after you have fully written your code once and tested it thoroughly.

5 Insertion

Assume you have built an R-Tree and wish to insert a point to the tree. First, you'll need to find the right leaf node, where the point is to be inserted. Then, you'll need to insert the point and re-adjust the tree so that the constraints are maintained.

Algorithm 1: Insertion in R-Tree

Input : A d-dimensional point P to be inserted, Identifier of the root node n
Output: Identifier/Pointer to the root node of the R Tree, with P inserted.

1 Get the root node RN, having the ID n, by retrieving it from the correct page in the file.
2 `// Note that in the following algorithm, for d = 2, the MBR will be a rectangle. For d > 2, we'll have a hyper-rectangle, and for those cases, area here means the volume of the hyper-rectangle.`
3 Let $m = ceil(maxCap/2)$
4 Traverse the tree from root RN to the appropriate leaf. At each level, select the node, L, whose MBR will require the minimum area enlargement to cover the point P.
5 In case of ties, select the node whose MBR has the minimum area. If there's again a tie, though highly unlikely practically, select the node that comes first in the list of children. Then recursively search in its children.
6 **if** *theselected leaf L can accommodate P (number of points stored < maxCap)* **then**
7 | Insert E into L.
8 | Update all MBRs in the path from the root to L, so that all of them cover P.
9 **end**
10 **else**
11 | Let E be the set consisting of all L's entries and the new entry P.
12 | Select as seeds two entries e1, e2 E, where the distance between e1 and e2 is the maximum among all other pairs of entries from E. Form two nodes, L1 and L2 , where the first contains e1 and the second e2.
| | **FYI:** The above algorithm is called Quadratic Split. You can also explore linear and exponential splits, and see how those affect the performance. Though, for this assignment, you'll work only with quadratic split.
13 | Examine the remaining members of E one by one and assign them to L1 or L2, depending on which of the MBRs of these nodes will require the minimum area enlargement so as to cover P. If a tie occurs, assign the entry to the node whose MBR has the smaller area. If a tie occurs again, assign the entry to the node L1 if it contains the smaller number of entries, else assign it to L2.
14 | If during the assignment of entries, there remain entries to be assigned, and the one node contains m entries, assign all the remaining entries to this node without considering the aforementioned criteria (so that the node will contain at least m entries).
15 | Update the MBRs of nodes that are in the path from root to L, so as to cover L1 and accommodate L2.
16 | Perform splits at the upper levels if necessary.
| | In case the root has to be split, create a new root.
17 | Write all the necessary changes to the file wherever necessary, and mark the pages dirty accordingly.
18 **end**

6 STR Bulk Loading

Inserting the points one by one is a very slow process and infeasible for a large number of points. We instead use bulk load to construct an R-Tree from scratch for a large number of points. For this, we need to sort the points in a particular way, described later in the next section. You will get the points pre-sorted in a binary file, which you will access through the file manager.

For now, let's assume we have the points sorted. View the numbers as blocks of maxCap number of points. Each block corresponds to a node in the *tree*. Assign a node to this block, note that this node is a leaf.

In an R-Tree, all of the leaf nodes occur at the same depth. So once you have assigned leaf nodes to all the points, we need to construct the entire tree by assigning common parents to these leaf nodes and then assigning further ancestors recursively.

Algorithm 2: STR Bulk Loading

Input : A file accessible through the File Manager API containing pre-sorted points of d-integers, the number of points N to be inserted into the tree

Output: A file accessible through the File Manager API containing the nodes of the tree

1. Define the number of slabs, $pieces = ceil(N/maxCap)$.
2. Each piece will contain $S = minimumof(maxCap, numberofremainingpoints)$
3. Iteratively, pick S number of points from the file (note that value of S, of course, will be the same for all nodes, except the very last one). Assign a common node LN, which is a leaf node, to all of these S points.
4. Now, recursively assign parents to these nodes using AssignParents function.
5. Return the pointer to the root node of the tree.

Algorithm 3: Assign Parents

Input : A file accessible through the File Manager API containing nodes of the trees, start and end (end excluded) index of the current top-most layer for which we need to assign parents

Output: None

1. Number of nodes N to assign parents = end - start
2. These nodes are present in $B(= N/maxCap)$ blocks. Each block will contain S = minimum of (maxCap, number of remaining nodes)
3. Iteratively, pick S number of points from the file. Assign a common parent node LN to all of these S points and add it to the end of file. Add a new page if needed.
4. If the number of nodes is more than 1, recursively call AssignParents with updated start and end index, corresponding to the new top-most layer.

6.1 STR Sorting

You DO NOT need to implement this, we have already sorted the points for you. The sorting was in-memory, and so, the algorithm used is also in-memory and done separately. Of course, for learning, for very large databases, you can try to implement external merge-sort, which can then be modified to be STR.

7 Point Query

Assume you have already constructed a tree. Given a d-dimensional point, you need to search for that point in the tree and return True if and only if you find the point. Given a node n and a point p, check among all the children of the node. If the MBR of the child e can contain the point p, recurse on the point p and node

e. If none of the children contain p, return False, else return true.

Algorithm 4: Point Query in R-Tree

Input : A d-dimensional point P, Identifier of the node n
Output: Bool denoting whether the point is present in the R Tree or not

```
Get the node N, having the ID n, by retrieving it from the correct page in the file.
if N is not a leaf node then
    Examine each entry e = (m, child) of N, where m is MBR of child and child is the identifier of
     the child node, find those whose MBR contains the point P.
    For each such entry e, call pointQuery(P, child)
end
else
    Examine all entries of N, where each entry will essentially be a point stored in the leaf.
    If any entry matches with P, return true, otherwise return false (false is only for this
     subtree/node, and NOT for the entire tree.
end
```

8 Submission details

8.1 Submission format

You will need to submit a zip file containing your code named `entrynumber1_entrynumber2.zip` (e.g. 2017MCS0001_2017MCS0002.zip).

1. The zip file shall create a folder with same name.
2. The folder shall only contain your code(and not the buffer manager files). We will add buffer manager code to it during evaluation.
3. The folder shall contain a `Makefile`. The Makefile will be used to compile your code with the following fixed targets.

The Makefile will be used to compile your code with the target rtree, which will implement all the above defined functions.

Your Makefile should also contain "clean" target for cleaning up compiled binaries.

8.2 Program execution

You compiled submission will be run as -

```
./rtree query.txt maxCap dimensionality output.txt
```

query.txt will contain the list of queries to be executed. Each line of this file is a query. The format for each query and its output is given later. maxCap is the max number of children a node in the tree can have. Dimensionality is the number of dimensions in each point. output.txt is the file where you have to write your outputs. After writing the outputs of a query, add 2 empty lines to separate it from outputs of next query. Do note that you will only construct a single tree for the entire query file, and apply the individual queries sequentially.

1. Bulk Load The query will be in format 'BULKLOAD FileName NumPoints'. Open the file FileName using the File Manager and perform STR Bulk Load on all the NumPoints points in the file. The output for this should be BULKLOAD.
2. Point Query The query will be in format 'QUERY ' followed by d integers separated by space ' '. These d-integers correspond to the d-dimensional point to search in the tree. Output 'TRUE' if and only if the point is present in the tree, else 'FALSE'.
3. Insert The query will be in the format 'INSERT ' followed by d space-separated integers. Insert the corresponding point to the tree and output 'INSERT'.

The submission of this assignment will be through Moodle. For checking the correctness of your code, we have provided you with some sample testcases with the expected outputs, which you can use to verify your outputs. However, your code will be tested against larger and more varied testcases too, so we advise you to form your own testcases to ensure correctness.

8.3 Evaluation

Your submitted code will be evaluated on a diverse variety of testcases. The file sizes may vary from few KBs to hundreds of MBs (or even few GBs). The number of queries may vary between 10 to few millions. Your code needs to complete in a time limit of 10 minutes per run.
As the outputs of insert query and bulk load query are internal structures, you will be graded according to the point queries. However, to provide more flexibility, we will also allow partial marks if you only wish to complete the assignment partially, and skip bulk load or insert algorithms/tasks. You can expect to get around 60-70% marks for this.