Assignment 3

Due Date: March 21, 2022

1 Introduction

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

1. Insert: Given a d-dimensional integer vector (a point in d-dimensional space), insert the vector in to the KDB-tree.
2. Point Search: Given a d-dimensional point, find if it is present in the KDB-tree.
3. Range Search: Given a hyper-rectangle, find all data points that lie in the specified hyper-rectangle.

Feel free to go use the following links to learn about KD-trees and KDB trees:

1. KD-Trees Original Paper: `https://dl.acm.org/doi/pdf/10.1145/361002.361007`
2. KD-Trees wiki: `https://en.wikipedia.org/wiki/K-d_tree`
3. KDB-Trees Original Paper: `https://dl.acm.org/doi/pdf/10.1145/582318.582321`
4. KDB-Tree wiki: `https://en.wikipedia.org/wiki/K-D-B-tree`
5. KD Tree & KDB tree example: `https://bit.ly/3IAAThx` or `https://db.inf.uni-tuebingen.de/staticfiles/teaching/ss14/db2/db2-04.pdf` (Slides 15-24)

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/Samidha09/COL362_632_Assignment3`. 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 empty1. 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 $\times$ sizeof (int). No point is split between two pages, if you need to split a point, it is simply written to the next page.

Note: You can use -1000000 as padding value if needed.

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.

- 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.
- 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.
- 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 KDB Trees

4.1 Tree Structure

The index KDB tree has two types of nodes – region nodes and point nodes. The region nodes stores a collection of regions stored in the following format,

$$[\langle r_i, cid_i\rangle, \ldots, \langle r_k, cid_k\rangle]$$

where, r_i is the region description and cid_i is the node id of the child of the respective region. A region r_i is represented using two d-dimensional data points, r_{min_i} and r_{max_i} such that r_{min_i} gives a lower bound across all the dimensions and the r_{max_i} gives the upper bound on the data points stored in that region. A point node stores the data points and their location in the database. Each entry in a point nodes is represented as,

$$[\langle point_m, loc_m\rangle, \ldots, \langle point_n, loc_n\rangle]$$

Along, with the information about the regions, points and pointers, each node also stores a variable `split_dim` that denotes along which dimension the node can be splitted. Later, we discuss the scenarios where node splitting will be required. To sum up the different nodes structure,

1. **Region Node**: A node storing information about l regions will store,
   - Total $l \times d \times 2$, integer values for describing l regions with 2 d-dimensional points per region,
   - l integer values storing child nodes ids,

- 1 integer for splitting dimension, `split_dim`
- 1 integer to indicate node type (region/point) , `node_type`
- 1 integer for storing node's parent, `node_parent`

2. **Point Node**: A node storing o data points will store,
   - $o \times d$ integers for storing o d-dimensional data points,
   - o integer values storing location points,
   - 1 integer for splitting dimension, `split_dim`,
   - 1 integer to indicate node type (region/point) , `node_type`
   - 1 integer for storing node's parent, `node_parent`

Example:

- One page = one region/point node.
- Let PAGE SIZE - 72 B and d=2.
- Page identifier consumes 1 integer. Therefore PAGE CONTENT SIZE = 72-4 = 68 B.
- Space required for storing split dimension (int) = 4B
- Space available after storing split dimension = 68 - 4 B = 64 B
- Space required for node type indicator (int: region/point node) = 4B
- Space available after storing node type indicator = 64 - 4B = 60 B
- Space available after storing node parent = 60 - 4B = 56 B
- Space taken by a region descriptor = (d*2 + 1)*4 = (2*2 + 1)*4 = 20 B, Here we have taken d*2 since for each dimension we store the min and max values. We assume that child identifier (cid) can be stored as an integer.
- Number of regions in a region node, l = floor(56/20) = 2
- Space taken by a single point= (d+1)*4 B = (2+1)*4 B = 12 B.
- Number of points in a point node, o = floor(56/12) = 4

4.2 Operations on KDB Tree

You are required to implement the following operations that can be performed over a KDB tree,

1. **Insertion**: Given a point $\{x_1, \ldots, x_d\}$ insert it into a KDB tree and reorganize the tree so as to ensure efficient space utilization.
2. **Point Search**: Traverse the tree to find the point node where a given data point $\{x_1 \ldots, x_d\}$ is stored.
3. **Range Search**: Given a region $r(r_{min}, r_{max})$, find all the data points that lies in that region.

Next, we provide pseudocode for these operators in detail.

4.2.1 Insertion

Let variable `root` points to the root node of the KDB tree and we aim to add a d-dimensional point stored in variable `point`.

Note: The splitting element should be the median of the $point_i$ for a point page and the median of the r_{min_i} for a region page. In case of decimal value take ceil.

4.2.2 Point Query

Similar to that of insertion, find the data node that can contain the queried point. Go over all the entries of the data node to find if the queried point is present in the dataset. Also print the number of region nodes touched while performing the point query.

Algorithm 1 `Insertion(point)`

```
1: if root points to null then
2:     Create a data node and add the point to it.
3:     Update the the root to this new node
4:     Terminate
5: else
6:     node ← root
7:     while node is not a point node do
8:         Find region $r_i$ to which point can lie in
9:         node ← $cid_i$
10:    end while
11:    Add the point to node
12:    if node overflows then
13:        call Reorganization(node) routine
14:    end if
15: end if
```

Algorithm 2 `Reorganization(node)`

```
1: split_dim ← node.split_dim
2: split_ele ← ChoseSplitPoint(node.items,split_dim)
3: lNode, rNode ← NodeSplit(node, split_ele)
4: parentNode ← node.parent
5: Add lNode, lNode.Id to parentNode
6: Add rNode, rNode.Id to parentNode
7: Delete node,node.Id from parentNode
8: if parentNode overflows then
9:     Call Reorganization(parentNode)
10: end if
```

4.2.3 Range Query

A range query defines a hyper-rectangle and intends to find all the data points that lie in specified hyper-rectangle. The query is defined same as the regions in the KDB trees are defined, i.e. a minimum value and a maximum value for each dimension. Let `rQuery(`min_q, max_q`)` be the range query where min_q is a d-dimensional point specifying lower bound on the values in the queried region and, similarly, max_q be a d-dimensional data point specifying the upper bound.

Algorithm 3 `NodeSplit(node, split_ele)`

```
1: for item ∈ node do
2:     if item is left to split_ele then
3:         Add item to lNode
4:     else if item is to the right of split_ele then
5:         Add item to rNode
6:     else
7:         nextToSplit ← item.childId
8:         lChild, rChild ← NodeSplit(nextToSplit, split_ele)
9:         lRegion rRegion ← Create two regions by redefine boundaries of item along split_ele
10:        Add (lRegion, lChild.id)to lNode
11:        Add (rRegion, rChild.id)to rNode
12:    end if
13: end for
14: Delete node
       return lNode, rNode
```

Algorithm 4 RangeQuery(rQuery)

```
result ← {}
if root points to null then
    return result
end if
toExplore ← {}
Add root to toExplore
while toExplore is not empty do
    Let node be a node in the toExplore set
    if node is a point node then
        Add all points in node lying in the query range to result
    else
        for item ∈ node do
            if Region defined by item overlaps with rQuery then
                Add item.childId to toExplore
            end if
        end for
    end if
    Remove node from toExplore
end while
    return result
```

5 Submission Details

5.1 Submission format

The assignment will be done **individually**. You will need to submit a zip file containing your code named entrynumber.zip (e.g. 2020MCS0001.zip) on Moodle.

- The zip file shall create a folder with same name.
- The folder shall only contain your code (and *not* the buffer manager files). We will add buffer manager code to it during evaluation.
- The folder shall contain a Makefile. The Makefile will be used to compile your code with the following fixed targets: **kdbtree** (to run all the above functions) and **clean** (to clean all the extraneous files/compiled binaries)

5.2 Program execution

You compiled submission will be run as: **./kdbtree query.txt d 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. d is the dimension of points. 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.

- **Insert**: The query will be in the format ‘INSERT ’ followed by d space-separated integers. Insert the corresponding point to the tree. The output should be ‘INSERTION DONE:’ followed by the data points contained in the point node where the query point was inserted. Note that the point node should be the one that is formed post insertion. Print each point on a new line in the form of d space separated integers.
  Example:
  INSERTION DONE:
  $< pt\_1 >$
  $< pt\_2 >$
  .
  .
  $< pt\_k >$

  Note: $< pt\_i >$ is a list of d space separated integers constituting i^{th} point, k is the number of points present in point node after insertion of query point.

- **Point Query:** The query will be in format 'PQUERY ' followed by d integers separated by space ' '. These d-integers correspond to the d-dimensional point to search in the tree.

  Output Number of region nodes touched(overlapping) while performing query. Print 'TRUE' if and only if the point is present in the tree, else 'FALSE' in the next line.
  Example:
  $< Num\_region\_nodes\_touched >$
  $< TRUE/FALSE >$
  Note: In case of FALSE, print 0 for number of region nodes touched.

- **Range Query:** The query will be in format 'RQUERY ' followed by 2*d integers separated by space ' '. Two consecutive integers represent the min and max values for the corresponding dimension. For e.g. the first two integers would represent the range of dimension 0 (r_{min_0}, r_{max_0}) and so on. Output the number of region nodes touched(overlapped) while performing the query and the list all the points lying in the hyper-rectangle in the following manner: In every new line print 'POINT: ' followed by a result point in the form of d space separated integers. After this give a space and then print 'NUM REGION NODES TOUCHED: ' followed by the number of region nodes touched to find that particular result point. Print the next point and number of region nodes touched to find it in the next line in the similar format as explained above.
  Example:
  POINT: $< pt\_1 >$ NUM REGION NODES TOUCHED: $< Num\_region\_nodes\_touched\_1 >$
  POINT: $< pt\_2 >$ NUM REGION NODES TOUCHED: $< Num\_region\_nodes\_touched\_2 >$
  .
  .
  POINT: $< pt\_k >$ NUM REGION NODES TOUCHED: $< Num\_region\_nodes\_touched\_k >$

  Note: $< pt\_i >$ is a list of d space separated integers constituting i^{th} point, $< Num\_region\_nodes\_touched\_i >$ is the number of region nodes touched(overlapped) to find the i^{th} point and k is the number of points found as result of the range query.

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.

Note: The buffer size and the number of available buffers can be varied during testing.

5.3 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 20 minutes per run.
As the outputs of insert query are internal structures, you will be graded according to the point and range queries. The marking scheme will be as follows:

1. 40% marks will be for insertion.
2. 25% marks will be for point query.
3. 35% marks will be for range query.
4. Range query score will be proportional to the number of correct points retrieved.

Note: 0 marks will be awarded if insertion is not implemented or if in-memory version of the data structure is implemented. All cases of plagiarism will be treated as per course policy.