香港中文大學
The Chinese University of Hong Kong

Faculty of Engineering

Department of Computer Science and Engineering

Final Year Project Report

Range Count in External Memory

2012 Fall Semester

Student : Lo Yu Ho

Supervisor : Prof. Tao Yufei

Project ID : TAO1201

Submission Date : 5th December, 2012

Abstract

Consider a dataset S consists of N points in $\mathbb{R}^2$. Given an axis-parallel rectangle Q, a query reports the total number of points falling inside the query rectangle Q. The goal of this project is to preprocess S into a structure such that all queries can be answered efficiently and the structure itself is space efficient and provide the API (Application Programming Interface) to other software developers. The currently built linear-space structure answers a query in $O(log_B^2 N)$ I/Os, and can be further optimized to $O(\log_B N)$ I/Os, where B is the disk block size. The structure can be built in $O\left(\frac{N}{B} log_B N\right)$ I/Os. And also, a sample application is developed to demonstrate the usage of the API on analyzing geometric data.

Table of Content

1. Introduction

The theory behind this project is a publication of Sheng, C. and Tao, Y. [1]. The main idea of the breakthrough is a structure named bundled compressed B-tree (BCB-tree). Section 3 describes the whole idea of BCB-tree and some refinement when get into practice.

The complexity analysis of built structure is under the external memory (EM) model, which is capturing the I/O characteristics of algorithms dealing with dataset that are too large to fit in memory. Thus, data need to be stored in an external device, in our case, a hard disk. The disk has been formatted into disjoint blocks; each has capacity of B words (in this project, it is defined as 4096 words, we call it a 4K block). An I/O is either a read from the disk to memory, or a write to disk from memory. Space complexity measures the number of disk blocks occupied, whereas time complexity gauges the number of I/Os performed. Linear cost is interpreted as $O\left(\frac{N}{B}\right)$ for a dataset cardinality N, while poly-logarithmic cost is interpreted as $O(log_B^c N)$ for some constant c.

The structure currently built has space complexity $O\left(\frac{N}{B}\right)$ and time complexity $O(log_B^2 N)$ I/Os. And the structure can be built in $O\left(\frac{N}{B} log_B N\right)$ I/Os.

Such a breakthrough in time and space complexity is valuable to real life applications. Here gives several applications of it. Consider each point in S to be the location of a tree, a range count query is "retrieve the number of trees located in CUHK", where Q is a rectangle describing the area of CUHK. By using our structure, some other

precise query such as "retrieve the number of trees located in New Asia College" can be done instantly, and you can query other colleges instead of New Asia College. Our structure is not limited to a pre-defined size of region, but a region based on any combination of $\mathbb{R}^2$ coordinates. It is highly flexible.

And one more example of application can be like this, consider each point in S to be a datum of a scatter diagram; Q is a rectangle describing the region of interest. To give a more concrete example, consider S to be the height and weight data of all the Olympic Games athletes since the first Olympic Games in 1896. Let x-axis be the height, y-axis be the weight of an athlete, a range count query is "retrieve the number of athletes tall between 169.5cm to 179.5cm and weight 70.01kg to 80.00kg", where Q is a rectangle origins at point (169.5, 70.01) with width 10.0 and height 9.99.

The fast response on a query allows to build a geostatistical analysis application which requires real-time response to not making users to wait for the results. Such response speed ensures the users can obtain the information they need very easily during analysis while using the application. Our sample application will show the query time cost is very small, such that user can not feel a delay while querying the data. Section 7 describes this sample application.

Section 2 is preliminaries. Section 3 provides the details of BCB-tree and some refinements. Section 4 describes how BCB-tree being used to solve range count problem. Section 5 describes the API. Section 6 provides performance evaluation. Section 7 describes the sample application developed to demonstrate the usage of the API on analyzing geometric data. Section 8 presents the conclusions.

2. Preliminaries

2.1 Gamma Elias code

One of the tricks used in our structure to achieve the goal is a compression scheme that allows converting an integer x into a binary string of $O(\log x)$ bits and permits lossless decompression when given a bit-stream that encodes a list of integers using the same scheme. In our structure, gamma Elias code [2] is used as such scheme.

Number	**Encoding**
1	1
2	010
3	011
⋮	⋮

Table 1: Examples of gamma Elias coding.

Gamma Elias coding converts a positive integer x into a binary string of $\lfloor log_2 x \rfloor$ zeros followed by the binary form that ignoring the leading zeros of x. Table 1 shows the examples of gamma Elias code conversion. To represent a number x, gamma Elias code uses $2\lfloor log_2 x \rfloor + 1$ bits. In practice, the numbers in our structure are dominated by the numbers with small magnitude, thus the length of bits to represent a number is largely reduced. Section 3 describes why our structure is dominated by the numbers with small magnitude.

To decode a bit string s, which may contains a list of integers encoded in gamma Elias coding, we read and count 0s from s until reach the first 1 and call the count of zeroes N. Then take the next N + 1 bits as the binary representation of an integer. If s has not been exhausted, the process is repeated to decompress the next value, and finally we

decompressed a list of integers.

There are two major drawback of this encoding scheme. One is not coding zero or negative integers, but this can be easily handled by adding another two bits. Another drawback is the terminator of an encoded string is not defined; this can also be solved by solving the zero and negative integer's drawback at the same time. Thus two bits will be added to the encoding scheme.

2.2 Modified gamma Elias code

Bit pattern	Meaning
00	positive sign
01	zero
10	negative sign
11	terminator

Table 2: Functional meaning of additional two bits.

We append two bits in front of the first zero of an encoded integer. One bit represents the sign of the integer, another bit represents whether it is zero. Meanwhile, there are two combinations are representing zero, positive zero and negative zero. Thus, we make use of this to solve the second drawback. Positive zero still represents a zero, while negative zero represents a terminator of an encoded string. Table 2 shows the functional meaning of additional two bits in modified gamma Elias code.

The modified gamma Elias code now compresses an integer x into $2\lfloor log_2 x \rfloor + 3$ bits and generalized to encode zero, positive integers or negative integers while the termination of an encoded string can also be detected.

2.4 Reduction for range count

Although we defined the problem to answer a query rectangle Q as a 4-sided rectangle, the result of a 4-sided query can be obtained in constant time from four 2-sided queries [2].

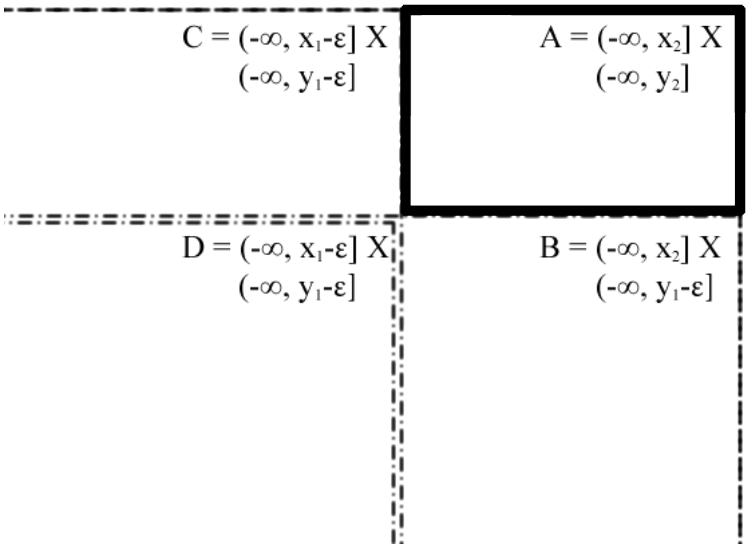

Figure 1: Graphical illustration of area A, B, C and D.

We define a 4-sided rectangle as $[x_1, x_2] \times [y_1, y_2]$. And a 2-sided query is in the form of (-infinity, x] × (-infinity, y]. As shown in Figure 1, a 2-sided query A = (-infinity, x_2] × (-infinity, y_2], B = (-infinity, x_1 - ε] × (-infinity, y_2], C = (-infinity, x_2] × (-infinity, y_1 - ε], D = (-infinity, x1 - ε] × (-infinity, y1 - ε], where ε is an arbitrarily small positive number, in our case of integer, we take it as 1. To answer a query rectangle Q, we query the result of A, B, C and D, and fit into the equation:

$$Q = A - B - C + D$$

Thus, the result of a 4-sided query can be obtained by those of four 2-sided queries.

3. Bundled compressed B-tree

This section is the details of BCB-tree structure. Section 3.1 describes the main idea presented in the publication of Sheng, C. and Tao, Y.. Section 3.2 describes some refinements of the idea proposed for practical reasons.

3.1 Main idea

To illustrate the idea of BCB-tree, we may get into a relevant problem call bundled predecessor problem first. Consider there are three sets of integer key and value (or, label) pairs. For example,

P1: {(16, 1), (25, 2), (29, 3)}

P2: {(16, 1), (20, 2)}

P3: {(24, 1), (26, 2), (27, 3), (37, 4)}

Given a key q, the query reports the labels associated with the largest key in every P_i, where $i \in [b]$, that is not greater than q. If not such a key, 0 is reported for that Pi. For example, we query a key 20 in the above example dataset; P_1 reports 1, P_2 reports 2 and P_3 reports 0.

Let $P_1, ..., P_b$ be $b \leq B$ sets of integer keys, where B is the size of a block. Each key $k \in P_i$ is associated with a label $\ell(k, i)$, in the example above, is the value of the key and value pairs. We refer to $\{P_1, ..., P_b\}$ as a bundle, each P_i as a category.

Then, we describe our structure to solve the above problem. We want to pack $P_1, ..., P_b$ into a list of triples that organize the keys in ascending order, and meanwhile, each

triple can be identified to their original set P_i, as well as their associated label.

For each key $k \in P_i$ ($i \in [b]$), define $\delta(k, i) = \ell(k, i) - \ell(k',i)$, where k' is the integer in Pi preceding k. For the set of P1, $\delta(25, 1) = \ell(25, 1) - \ell(16, 1) = 1$. If k' does not exist, take $\ell(k', i)$ as 0, thus $\delta(k, i) = \ell(k, i)$ (i.e. $\delta(16, 1) = 1$ in the above example). You may see that in counting problem, $\delta(k, i)$ is always 1, but for the generality of BCB-tree API, we decided to keep this property in implementation.

Set $K = \Sigma_i|P_i|$, and let P be the multi-set that unions all of $P_1, ...,P_b$. Let $k_1, ...,k_K$ be the keys of P in ascending order. Define $\hat{\delta}(1) = k_1$, and $\hat{\delta}(j) = k_j - k_{j-1}$ for $j > 1$. We create a list Δ of length K as follows. The j-th entry of Δ is a triple $(\hat{\delta}(j), i, \delta(k_j, i))$, where i is such that $k_j \in P_i$. Since there are totally b categories, i can be represented with $\lfloor log_2 b \rfloor$ bits. We store $\hat{\delta}(j)$ and $\delta(k_j, i)$ using the gamma Elias code. Because of $\hat{\delta}(j)$ and $\delta(k_j, i)$ are small in magnitude in most of the cases, gamma Elias code largely reduced the length of bits to store a triple.

Now, to restore any key/label of any P_i, we have to scan through Δ from the beginning. To remedy this drawback, we organize the structure in following manner. Let us define a fat block to be 4 consecutive blocks (i.e., 4B words). Recall that each triple in Δ corresponds to a key in P, so the tuples can be grouped by their corresponding keys. Each group has at most b tuples, a fat block may contain one or more adjacent groups as long as it still has enough space; otherwise, we place the succeeding group in a new fat block. Thus each group occupies at most $3b \leq 3B$ words, except possibly the last one. Denote by $I(v)$ the minimal interval enclosing all the keys in a fat block v, thus all the $I(v)$ are disjoint. And, each fat block v is associated with a relay set relay(v), which contains b values corresponding to a category. The i-th ($i \leq b$) value in relay(v) equals $\ell(k, i)$, where k is the greatest integer in P_i of previous fat block.

To convert each triple ($\hat{\delta}(j)$, i, $\delta(k_j, i)$) in v to (k_j, i, $\ell(k_j, i)$) accurately, we access the relay block to retrieve the i-th value in relay(v) and scan through the fat block until k_j and $\ell(k_j, i)$ is recovered.

Finally, we create a B-tree on the interval of keys of all fat blocks *v*. The B-tree, the fat blocks and their relay sets constitute the BCB-tree. The analysis of space and time complexity can be found in the publication. I just highlight the time complexity of building cost, which is $O\left(\frac{N}{B}\right)$ if all the keys have been sorted, query cost, which is $O(log_B N)$ and space complexity, which is $O\left(\frac{N}{Blog_B N}\right)$.

3.2 Refinement

In practice, there are a few changes have been made to have a better performance and release the constraint that all the keys in a category has to be distinct.

First, we consider the case that keys are not distinct in a category, the number of tuples in a fat block may exceed b and unbounded because one single key may has multiple tuples in a category after we released the constraint. Thus, when the keys are grouped to form a fat block, it may causes a combined size larger than a fat block, i.e. each group may occupies more than 3B words, which breaks our assumption in section 3.1. We decided not to group the tuples by their corresponding keys if we have to release the distinct key constraint. Instead of grouping the tuples, we treat the tuples as individuals. Tuples will be placed in a fat block as long as that fat block is not fully filled, no matter the tuples are in the same group or not. But because of this, key intervals of fat blocks may overlap the adjacent fat block for at most one single key

(the boundary one). This breaks the disjoint $I(v)$ assumption, but it is totally fine. By doing so, we released the constraint of all the keys has to be distinct in the same category.

Figure 2 shows the data arrangement in block manner.

Second, the total number of categories is original limited to B, recall that P_1, ...,P_b, where $b \leq B$. To store the relay set, B categories means B integers have to be stored for one fat block. Taking integers as 32-bit integers, 4B storage is required for storing the relay set of a fat block. If we limit P_1, ...,P_b, where $b \leq \frac{B}{4}$, the relay set can be fit into one single block, which improved the query performance (3 blocks less for each query). Since each fat block is occupied at most $3b \leq 3B$ words, we can place the relay set into the fat block alongside with triples. You may see this in Figure 2, the gamma Elias code data occupies 3B words and followed by B words, which is the relay set, and formed a fat block. However, section 4 will describe that we further limit $\frac{B}{4}$ to $\frac{B}{8}$ due to another reason, but each relay set still occupying one single block, i.e. only half of the block is utilized.

Another refinement is adding a metadata block at the beginning of the structure to store the necessary information. We have to place the B-tree data block after the fat blocks because we do not know the number of B-tree blocks (which indexing all the fat blocks interval $I(v)$, thus it depends on the number of fat blocks) before processed all the input into BCB-tree. You can see from Figure 2 that we place the B-tree blocks after all the fat blocks. Then, we need a metadata block to store the information like

the location of root node of B-tree and the location of B-tree leaf nodes to make a correct jump over data blocks.

Throughout the construction of fat blocks, the derived fat block intervals (which same as data in the leaf blocks of B-tree) are growing when more input data is read, which may exceeds the memory available. Thus, a temporary file will be written to disk to store such information and being fetched when constructing the leaf nodes of B-tree. Such temporary file will be deleted once the BCB-tree construction process is completed.

4. BCB-tree in range count

4.1 Structure

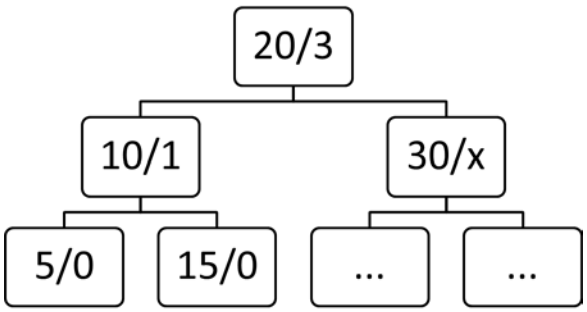

Figure 3: A binary tree that each node associates with an element and a value.

Consider a binary tree, each node associates with an element and a value; the value indicating how many nodes is in its left sub-tree. At a leaf node, the value is 0, and its parent node has a value of 1. If its parent node has two children, the parent node of it, let us call it root node, has a value of 3. If we have a query falling in the right sub-tree of root node, we can conclude that there are at least 3 elements preceding our query. Let us use Figure 3 as an example, if our query is > 30, let it be 31, then we can

conclude that there are at least 3 + x elements preceding our query 31, where x is the number of elements in the left sub-tree of 30. Going down to query the right sub-tree of 30, and found it is a leaf node, if the leaf node is smaller than our query 31, we conclude 3 + x + 1 to be the answer, otherwise we do not add that one to have 3 + x to be the answer.

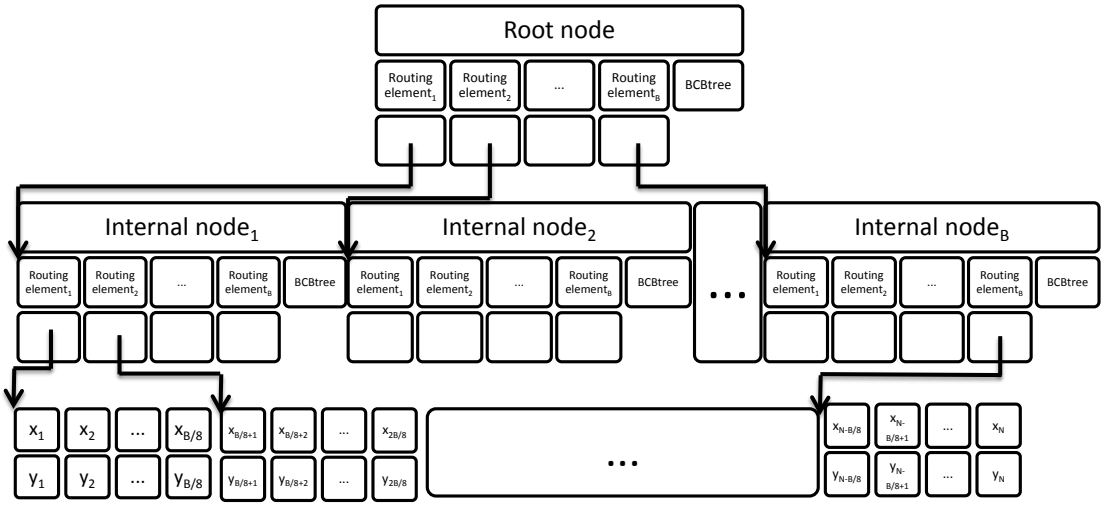

Figure 4: A two-level B-tree structure with internal nodes associated with a BCB-tree. The leaf node is x and y-axis coordinate of each points.

By a similar way with above example and BCB-tree, we can construct a structure that can report how many points are preceding our query in 2-dimensional. Instead of associating with a value, each internal node associates with a BCB-tree to answer the y-axis coordinate query. You can see from Figure 4 that each internal node associates with a BCB-tree to achieve the purpose. We categorize every B (let us use B here instead of the refined value, we will describe the refinement soon later) points into one category with y-axis coordinate as keys and their ranking as value (or say, label), when we have B categories, we build a BCB-tree on these B categories and make it be associated with an internal node of a B-tree built on the x-axis coordinate as routing

elements. By repeating until all points are processed, there are $\frac{N}{B}$ internal nodes in the level above the leaf nodes. For each internal node, the points in its BCBtree are categorized into one category. For every B internal nodes, build a BCB-tree on these B categories and form an internal node in one upper level. Repeat until one internal node remains, and we call it a root node.

There are $O(log_B N)$ BCB-trees, and each BCB-tree is bounded by $O\left(\frac{N}{Blog_B N}\right)$. The space complexity is bounded by $O\left(\frac{N}{B}\right)$, which is linear. And the structure can be built within $O\left(\frac{N}{B} log_B N\right)$ I/Os if input data are sorted in ascending order of x-axis coordinates.

To answer a 2-sided query, we first look into the root node and access its routing elements, for each corresponding routing elements preceding our x-axis query, we can safely sum up the number of points inside their categories those are below our y-axis query. We can query the BCB-tree that associated with the node to obtain such numbers. Then, we proceed to the corresponding internal node in the next level and repeat the procedure until reach the leaf node. In the leaf node, we can read through the whole block to count the number of points with x-axis coordinate and y-axis coordinate less or equal to our query. The time complexity of querying such structure is $O(log_B^2 N)$.

4.2 Refinement

Each leaf and internal node element requires two 32-bit integers, for corresponding x-axis coordinate plus y-axis coordinate or corresponding routing element plus

routing pointer. Thus, if we categorize B element to one category, it requires 8 blocks for storing one category. To improve the query performance, we limit B to $\frac{B}{8}$ for the number of elements in a node. Thus the fanout of B-tree is limited from B to $\frac{B}{8}$, but every time we query an internal or leaf node, we read only one block instead of eight blocks.

And, similar to the problem of BCB-tree, we need a block for storing the metadata at the beginning of structure. It includes the similar information of the metadata of a BCB-tree, the location of leaf and root nodes. But, instead of storing the number of categories; we store the total number of points in the structure to indicate when to stop if we are processing the last block of leaf nodes, which may not be fully filled.

There is a remark that has to be mentioned. We have to create some intermediate files when building the structure. We have to sort the first $\frac{B}{8} * \frac{B}{8}$ y-axis coordinates and store them in an intermediate file, for both passing to the function to build a BCB-tree or to merge with other $\frac{B}{8}$ - 1 intermediate files when proceeding to the next level of B-tree. For building each level of B-tree, it requires to scan through all the files once to merge the files, thus the building time complexity is $O\left(\frac{N}{B} log_B N\right)$.

5. API

The BCB-tree API provides functions for creating and querying BCB-tree structure, while the BCB-tree range count API provides functions for creating and querying BCB-tree range count structure.

5.1 BCB-tree API

Without loss of generality of the functions of BCB-tree in other variants of orthogonal range query, we provide the API of BCB-tree for further software development.

```
int build_BCBtree(const char* input_file_name, const char*
output_file_name, int block_offset);
```

Figure 5: Function prototype of building BCB-tree structure.

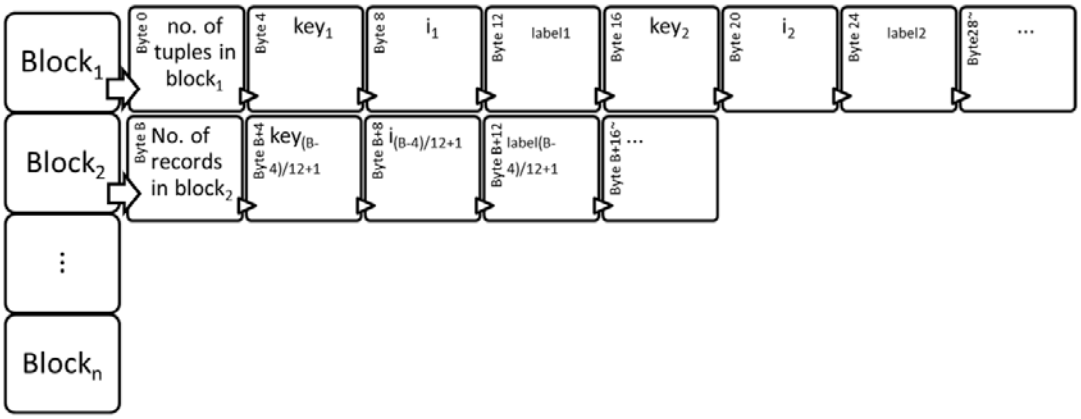

Figure 6: The input file format of building BCB-tree structure, it should be in ascending order of keys.

The parameter *input_file_name* is the directory of input file and *output_file_name* is the directory of output file. The parameter *block_offset* is indicating how many disk blocks should be jumped over before the first block of BCB-tree structure. The return value of the function is the number of blocks occupied for the built structure, return -1 if fail. The input data should be in the form of triple described in section 3. Thus each tuple in the input data occupies 12 bytes (three 32-bit long integers). You may see this in Figure 6. The function is reading the input data in a block by block manner. Consider if some block is not fully filled, especially the last block of the whole input data file, this may cause the function to read some data that is not intended to read.

Thus, the first 4 bytes (32-bit) of each block in input data file is being used to indicate the number of tuples being contained in that block to solve this problem.

```
int query_BCBtree(const char* input_file_name, int
block_offset, int** output_list, int query);
```

Figure 7: Function prototype of querying BCB-tree structure.

Similar to the build function, the parameter *input_file_name* is the directory of the built BCB-tree structure, and the parameter *block_offset* is indicating how many disk blocks should be jumped over before looking into the first block (metadata block) of BCB-tree structure. The parameter *query* is not more than the value wanted to query. The query result will be stored in a newly created 32-bit integer array, which the address of the first element of the array is stored in input parameter *output_list*. And the return value is the number of elements in the array pointed by *output_list*, return -1 if fail.

5.2 Range count structure API

```
int build_BCBtree_range_count(const char* input_file_name,
const char* output_file_name);
```

Figure 8: Function prototype of building our range count structure.

Input file

x_1[\space]y_1[\n]

x_2 y_2

⋮

x_N y_N

Figure 9: The input file format of building BCB-tree structure.

The API of BCB-tree range count is similar to that of BCB-tree. The parameter *input_file_name* is the directory of input file and the *output_file_name* is the directory of output file. The input file format is just a plain text file containing the x-axis coordinate and y-axis coordinate of the data points. Between two x-axis coordinate and y-axis coordinate, there is a space character, and between two data points, there is a new line character (also known as, '\n'). You may see this in Figure 9. Since the whole output file contains only one structure to answer range count query, thus unlike the API of BCB-tree, it takes no block offset, or say block offset is a default value zero. It will return the number of blocks occupied for the built structure, return -1 if fail.

```
int query_BCBtree_range_count(const char* input_file_name,
int query_x, int query_y);
```

Figure 10: Function prototype of querying our range count structure.

The function takes the directory of input file as input parameter *input_file_name*. The input parameter *query_x* is x of 2-sided query and *query_y* is y of it. Unlike the BCB-tree query, it will return only one value, which is the answer of the query. It will return -1 if failed in query. As mentioned, the query time is $O(log_B N)$.

5.3 Remarks

The interface is totally C language compitable, but the implementation of the function is written in C++. Thus, to compile our code, we use g++ instead of gcc.

6. Performance evaluation

We first measure BCB-tree's performance with building cost, space consumption and query cost with different size of datasets. We then measure the range count structure on a few benchmark datasets.

6.1 BCB-tree performance

Number of key and value pairs	Building cost (I/Os)	Occupied blocks (B)	Query cost (I/Os)
10	7	6	6
100	7	6	6
1000	9	6	6
10000	40	10	6
100000	356	62	6
1000000	3507	574	6
10000000	35031	5705	7
100000000	350302	57046	7

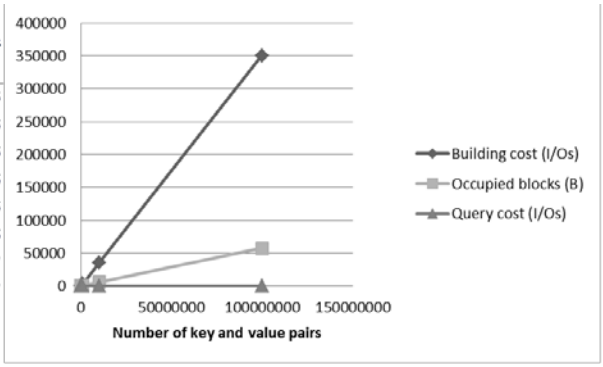

Figure 11: The table shows the experiment result with varies size of evenly distributed dataset. The graph shows the trend of growth according to the size of dataset. Each block is of the size B = 4096 words.

When the dataset is too small, the overhead of BCB-tree is obvious in all the three aspects, building cost, space consumption and query cost. The explanation of this overhead is that we at least have a metadata block, a fat block (which occupies 4 blocks) and a root node of B-tree, totally 6 blocks at minimum storage and building I/Os. When building the structure, extra I/O cost is needed to read the input data.

Both building cost and space consumption increase linearly with different coefficients when dataset size grows. The query cost increases by one I/O when dataset size grows to 10 million because the B-tree structure grows one more level than the smaller

dataset, thus 1 extra I/O is required. And it confirms the query cost to be logarithmic.

6.2 Range count structure performance

Dataset name	Number of 2-dimenional points	Building cost (I/Os)	Occupied blocks (B)	Query cost (I/Os)
minimal.dat	2	11	9	8
LA_original.dat	131461	1205	433	9
TCB_original.dat	556696	13751	2229	16
CAR.dat	2249727	41005	8009	16
CAR_original.dat	2249727	41245	8249	16
RAIL.dat	3350387	63852	14713	17
RedList.dat	5126830	91538	16348	17
UAC.dat	7980981	150666	33617	17
BLOCK.dat	11496067	217394	48796	17
NE.dat	19909725	376205	84216	17
NE_original.dat	19909725	379122	87133	17

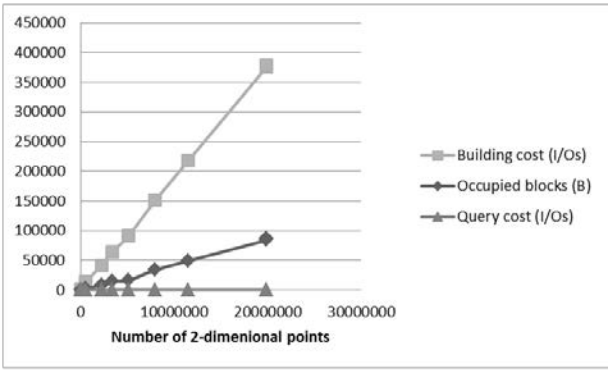

Figure 12: The table shows the experiment result with different benchmark datasets. The graph shows the trend of growth.

Similar to BCB-tree, the overhead cost is obvious when dataset is small. Let us look at the smallest dataset which contains only 2 points. In our structure there must be a root node, and a root node is associated with a BCB-tree, thus 6 blocks are occupied for this. And the remaining 3 blocks are metadata block, leaf block and the root block mentioned. And for the worst case of query cost, we have to query the BCB-tree to know the total number of points in the left branches of B-tree, thus 6 I/Os are spent on this, another 2 I/Os are for reading metadata block and root block. When building the structure, since the input is a plain text file, we do not count the I/Os required to read the input file, but we can expect it is being scanned through once only.

For LA_original.dat, it illustrated the situation of a B-tree with many leaf node, but no internal nodes. Since we defined B as 4096 words, the fanout of B-tree is $\frac{B}{8}$, which is 512. This dataset has no internal nodes because it occupies less than 512 leaf nodes.

And thus, the cost is nearly the same as the dataset that has only 2 points. The worst case requires one more I/O because we have to process a leaf node to obtain an accurate result while the previous one does not.

For the rest of the experiment results, the $O\left(\frac{N}{B} log_B N\right)$ building cost, linear space consumption and poly-logarithmic query cost can be observed. The query cost changes from 16 to 17 when the dataset size grows. Instead of the structure grows for another level of B-tree, the BCB-tree associated with the root node grows for one level, it matches to our experiment result on BCB-tree in section 6.1 that a growth of BCB-tree requires one extra I/O to query. Besides, the query cost is very stable across different query value, which is valuable to be highlighted.

Here is some statistic of the file size. The size of RAIL.dat is 63MB, while the built structure is 58MB. The size of UAC.dat is 152MB, while the built structure is 132MB. The size of NE.dat is 378MB, while the built structure is 338MB. As a contrast, the size of RedList.dat is 63MB, same as RAIL.dat, while the built structure is 64MB. It is due to the reason that most of the dataset have much more big numbers, while RedList.dat has less. Big numbers stored in plain text format are space inefficient, thus the compression ratio is better for these kinds of datasets. If we let each entry in the datasets to be two 32-bit integers, we can calculate that our structure is around 1.5 ~2 times the dataset file size. Since the leaf node of our range count structure is doing the same thing to store all the entry in 32-bit integers, the minimal file size will not be less than the input file size in this case if we do so.

7. Sample application

We have built a sample application with graphical user interface (GUI) to demonstrate how our structure can be used in real life applications. Our sample application is built with gtk+ [4], which is a nice cross-platform widget toolkit to create GUI.

7.1 GUI design

We make the GUI layout design as simple as we can. A simple layout has several advantages. It is intuitive for the user to use, resources efficient and easier for the software developer to program, especially for this small sample application.

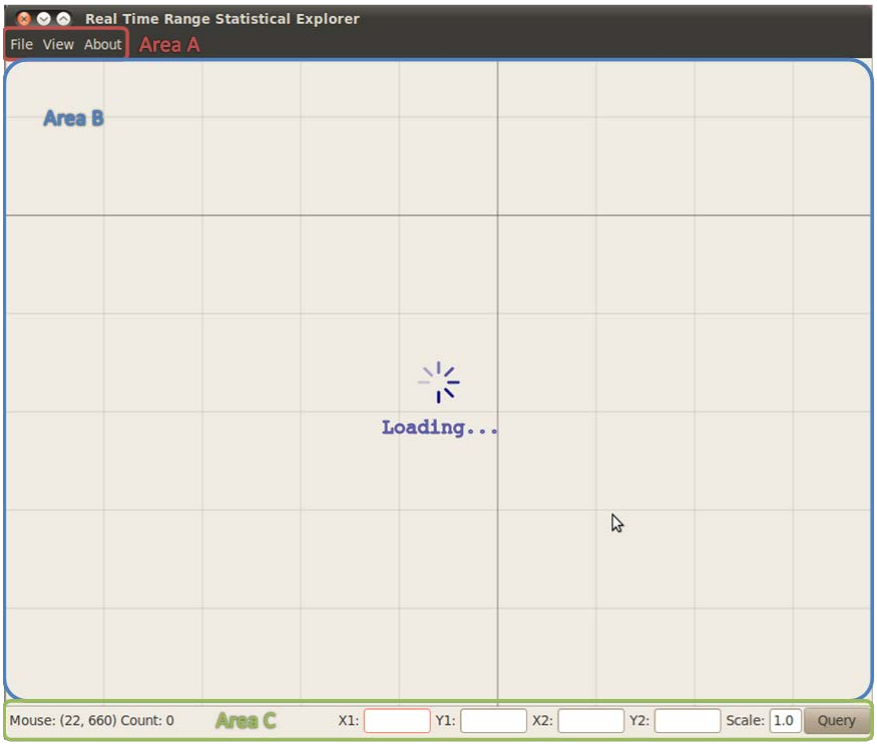

Figure 13: GUI layout of sample application program: Real Time Range Statistical Explorer.

Referring to Figure 13, area A is a menu bar, it contains command like open a file, exit the application, show the information about this application, and additional control on the view of area B. Area B, which is a drawing area, we will discuss it in the next paragraph. Area C is a status bar to show the result of query, coordinate currently pointing at, and some textboxes and a button for query with keyboard input.

Area B is a drawing area, user do control on this area most frequently. The control is simply dragging, dropping and scrolling. Points in the dataset are shown in this area. As long as a point is fall inside the current window view, it is being shown in this area. By the way, the background can be changed according to the meaning of the dataset. If the dataset is the trees locations in CUHK, then a map of CUHK can be shown in the background instead of grids.

7.2 Features provided

Users can do query, zooming and exploring in the drawing area.

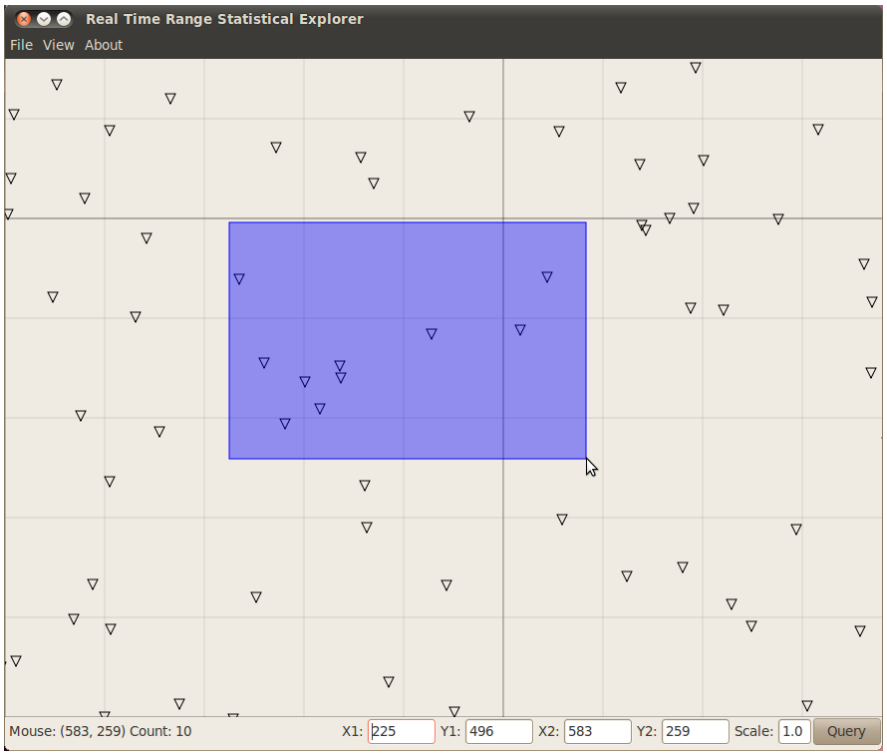

Figure 14: Dragging a window to query the structure.

To do a query, it is simply dragging a rectangle (a window), the system will query the structure simultaneously when the window size is being changed, i.e. user does not need to release the button to have query result. Since the queries may fix at a starting point, but with different width and height, this feature makes it convenient to do such kind of queries.

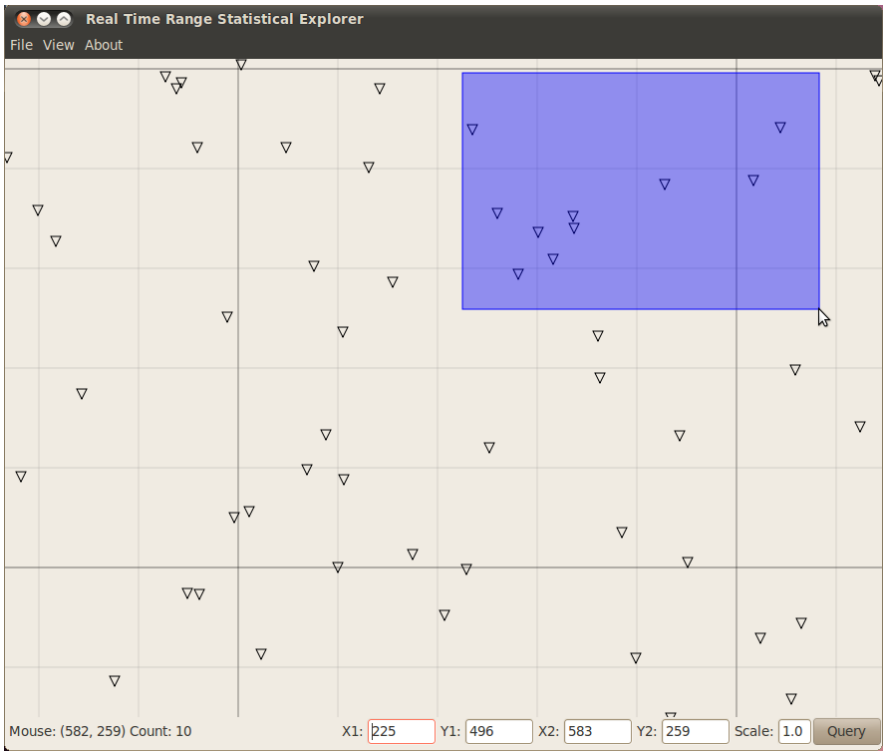

Figure 15: Dragging while pressing down control key, user can explore around the dataset space.

Because the size of drawing area is not large enough to browse the whole dataset in most of the cases, so we provide a way that user can explore the dataset space by dragging with pressing down the control key. While pressing down the control key, the coordination system of dataset space will change according to the user's dragging action. It is intuitive enough to drag left and right, up and down.

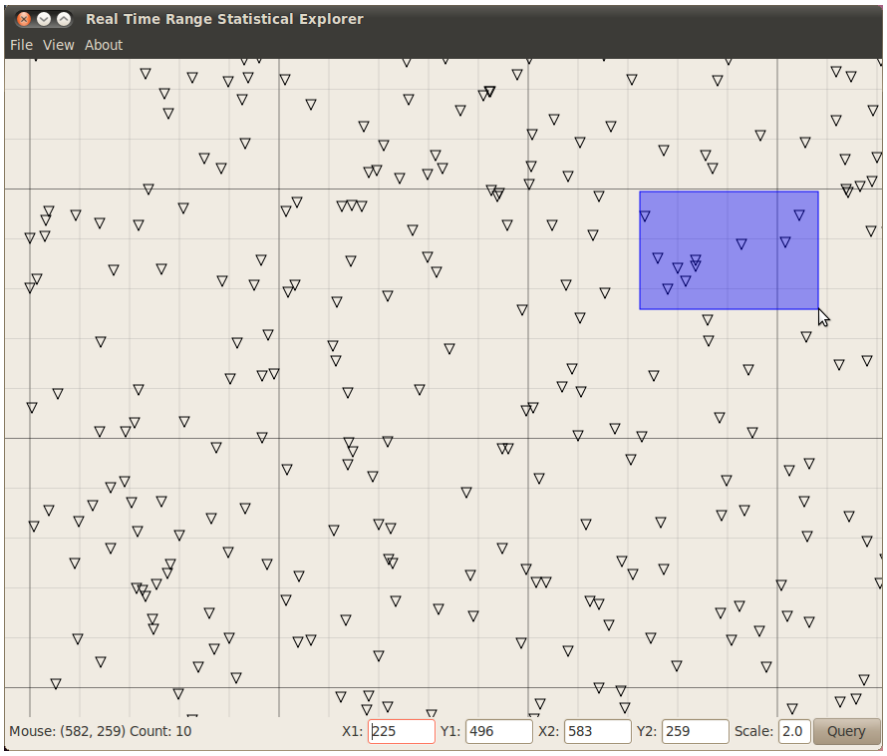

Figure 16: Scrolling up and down to change the scale multiplier.

Besides exploration, user can zoom in and zoom out to have different scaling on the coordination system. Zooming can be done by scrolling the middle button of mouse, scroll up to zoom in (just like you walk closer to the screen) and scroll down to zoom out (just like you walk beyond from the screen). This allows user to see more points on the screen. However, there is a concern for zooming feature. While keep zooming out to a certain level, the points shown in the screen will be a large amount (datasets are always over million/billion points), this will take a large cost for rendering the points. To solve this problem, we use a technique called consistent sampling. When zooming out to a certain level, some points in the dataset are randomly selected to show in the screen instead of showing all. If zooming out furthermore, we randomly select points from the set of points that we randomly selected previously. This largely reduces the rendering cost and gives a consistence on the points showing on the screen.

To implement the exploration and zooming features, we need a conversion of coordination system. We define x_{offset} and y_{offset} for how far the boundary of the currently view is from the origin (0, 0) of dataset space. Since window applications have origin (0, 0) at the top left corner with x-axis pointing right and y-axis pointing down, it makes the conversion of coordinate a little bit complex because the dataset space coordinate system has y-axis pointing up. But this still can be solved. We combine the converted coordination system with a scale multiplier, the exploration and zooming features are completed and being provided.

7.3 Real time query response

Since querying the structure involved I/O function calls, which are blocking function calls, we thus create a thread to build the structure and query the structure. Every time the main thread received a command to build or query a structure, the main thread will signal this thread to handle the blocking function calls and thus the main thread can avoid blocking and responds to user input on GUI. To present the functions of our structure with such a multi-thread mechanism, user does not feel any unpleasant delay while querying the structure.

8. Conclusions

We have described BCB-tree structure and how it is being used to solve the range count problem. We provided the API for developing further applications using either BCB-tree or the range count structure. BCB-tree provides logarithmic I/O query cost, linear space consumption and linear I/O building cost to solve bundled predecessor

problem. And range count problem can be solved with poly-logarithmic (can be improved to logarithmic) I/O query cost, linear space consumption and $O\left(\frac{N}{B} log_B N\right)$ I/O building cost.

The range count structure uses BCB-tree to solve range count problem has successfully met the real time response requirement on an application. Real Time Range Statistical Explorer demonstrates the potential ability of adopting BCB-tree in geostatistical application software. And this meets our purpose to build a real time system to answer range count queries on large dataset.

Remark: The following is the optimization of query cost of our current structure. To remedy the drawback of querying the B-tree of BCB-tree at each level of range count structure, we may add additional information in the fat blocks of BCB-tree. Let us consider a case of range count structure having two levels in its B-tree. For the lower level, we record down the address of the fat blocks as well as the largest key in that fat block. And then, we can make use of this information in the upper level BCB-tree to locate the corresponding fat block in lower level by only querying the B-tree in the upper level BCB-tree. That is, for each key in the BCB-tree, it will associate with two labels, one is its rank, and another is its fat block location in one level lower. Referring to section 3, define the location of fat block as *addr*(k) in one single BCB-tree. We make sure that, for each k, it has only one *addr*(k), if it has multiple *addr*(k), we take the largest one to keep the consistence of the structure. When we combine B (actually $\frac{B}{8}$) BCB-trees into one, we rename *addr*(k) to *addr*(k', i) where i $\in$ [b] and i is the category id of the set it belongs to. For each key k, it is now associated with two labels, one is ℓ(k, i), which is its rank, and an address *addr*(k', i), for the k which does not match k' exactly, we take the smallest k' that larger than k in the same category i, we treat the address in the same way as the labels and expend one

relay set to two for each fat block for recovering ranks and addresses correspondingly. Now, we can recover two things in one single BCB-tree, after we retrieved the ranks, we can jump to a fat block in a lower level directly without querying the B-tree in another BCB-tree. By doing so, we remove the square of our current poly-logarithmic query cost, and it becomes $O(log_B N)$ without affecting the space complexity and building time complexity.

References

[1] Sheng, C., Tao, Y. Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), pages 25-35, 2011.

[2] Elias, P. Universal Codeword Sets and Representations of the Integers. IEEE Trans. Inform. Theory 21, 2 (Mar.), 194-203. 1975.

[3] H. Edelsbrunner and M. H. Overmars. On the equivalence of some rectangle problems. Information Processing Letters (IPL), 14(3):124–127, 1982.

[4] http://www.gtk.org/