Implementation and Evaluation of I/O Efficient Range Trees

Akif Ahmed Khan

IIIT-D-MTech-CSE
July 28, 2015

Indraprastha Institute of Information Technology
New Delhi

Thesis Committee
Dr. Rajiv Raman (Advisor)
Dr. Debajyoti Bera
Dr. Ojaswa Sharma

Submitted in partial fulfillment of the requirements
for the Degree of M.Tech. in Computer Science.

Keywords: External Memory(or EM), Input/Output(or I/O), Range Search Trees, Buffer Technique, Standard Template Library for Extra Large Data Set(or STXXL), External Memory Algorithms, I/O Algorithms, Out-Of-Core computations.

Certificate

This is to certify that the thesis titled **"Implementation and Evaluation of I/O Efficient Range Trees"** submitted by **Akif Ahmed Khan** for the partial fulfillment of the requirements for the degree of *Master of Technology* in *Computer Science & Engineering* is a record of the bonafide work carried out by him under my guidance and supervision at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Dr. Rajiv Raman
Indraprastha Institute of Information Technology, New Delhi

Abstract

Range Tree is a geometrical data structure used to store data in d-dimensions. In order to store larger amount of data in external memory, we need an I/O efficient algorithm. There are many I/O efficient algorithms designed like buffer trees but we need to implement these algorithm designs and evaluate its performance.According to our study, there is no implementation of 2-dimensional I/O efficient range trees available. In this work, we have implemented range trees using buffer technique in 1-dimension and 2-dimensions. Range Tree data structure is used to answer range search queries in an I/O efficient manner.
We have evaluated I/O Efficient Buffer Range Trees with different set of parameters like buffer size, number of input elements and block size. We did our experiments and evaluation with a randomly generated data set and geographical latitude-longitude data of Open Street Maps.

Acknowledgments

First and foremost, I offer my sincere gratitude to my advisor, Dr. Rajiv Raman, who has supported me throughout my thesis with his patience. I want to thank my advisor for the belief that he has shown in me and gave me sufficient time to work. I dedicate this thesis to my family who believed in me, supported me and motivated me all the time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern computational problems in many domains require processing of larger and larger volume of data. The data required to be processed far outstrips the capacity of the RAM on most machines. This requires us to store the data in external memory and retrieve small amounts at a time and store into RAM for processing. A major drawback is that EM is extremely slow compared to RAM. So the right measure is number of I/Os and not amount of computation. This suggests the need of EM algorithms that minimizes I/Os and assume computation comes for free. Due to a massive increase in the amount of data that we need to process, analyse, computer's internal memory is not enough to store a sufficiently large data. Large data set includes data in the field of telecommunication industry wherein lots of data get generated due to the continuous phone calls which is required to be processed for generating bills, phone call graph analysis to understand the customer behavior, geo spatial data, geometrical data , social network data and many more. All these sufficiently large data cannot fit into the computer's internal memory which is required to be swap in/out from external memory(or EM). Though, the generations of computer architecture are getting better with time but we do not want to wait for the hardware to improve instead use currently available limited resources by understanding and utilizing the fundamental constraints to the best of its capacity.

1.1 Motivation

The average latency growth in the design and manufacturing of EM disk hardware is 9% per year whereas for the processor it is 55% per year. In the absence of EM algorithms, the computer's internal memory makes use of virtual memory which is a swap memory residing on the hard disk making the address space look larger than its actual size but it is not efficient enough because it will not understand the complex data and it may reduce the efficiency of the system. The cost of disk I/O is empirically 10^6 times [11](approximately) than the cost of accessing the data through internal memory. So, the I/O cost for EM becomes a major bottleneck in the computation of data stored in EM and we need to design and implement I/O efficient algorithms that minimizes the I/O cost.

Many EM algorithms have been designed in the past that exploits and makes the most of locality of data in internal memory. In case of large data sets as well, there are two types of data set *viz.*batched data and on-line data. Batched problems have all the data available initially to get processed and get into the internal memory completely and On-line problems have dynamic data set which involves continuous flow in/out of the data causing insertions and deletions. However, these EM algorithms are supposed to be implemented so as to understand its behavior

on different parameters of system resources.

One such area which involves massive datasets is geometrical data structures in various dimensions, or the geospatial data in 2-dimensions. There are many data structures and algorithms that efficiently store the geometrical data such as KD-Trees [5], Quad Trees [12], Range Trees [5] which are efficient for specific purposes. Range Trees are used to do a range query on geometrical data in d-dimensions. In this thesis, we have implemented and evaluated Range Search Trees using the buffer technique which answers range queries on the given geometrical data set of points in 1-dimension and 2-dimension in external memory which is fast and scalable.

1.2 Problem Statement

A natural question for data set of points in 1-dimension or 2-dimension, we are always interested to know for the set of points that lies within a certain range. If we try to make range queries on points data set which comes as an input in unsorted order, it would be a very inefficient way to do range queries without using some data structure for it. Range Tree is one such data structure that can be used to do range queries on data set points efficiently in computer's internal memory in $O(\log_m^2 n)$ time in 2-dimension where m is maximum number of children for any internal node in the tree. However, this running time is an asymptotic time which applies in case the range tree data structure is stored in the computer's internal memory for its entire life time.

In case of data set being larger than the available internal memory size, we use EM data structure of Range Tree using a buffer technique which allows to insert data which is larger than the available internal memory size and do range queries on the data structure efficiently such that the I/O cost is minimized.

With this thesis work, we have tried to implement Buffer Range Tree in EM such that range queries gets answered correctly and efficiently. Moreover, we have evaluated its performance by setting different parameters for the tree.

1.3 Outline

In this thesis, we have briefly described about EM algorithm for range search trees along with a substantial set of experiments to evaluate the results. Chapter 2 describes EM model and algorithm for basic problem in external memory. Chapter 3 introduces an implementation of EM algorithms called the STXXL library [6], elaborating on its features, architecture and design followed with a brief introduction to *STXXL::vector*. STXXL [6] is an external memory implementation of C++ STL library [7] which supports parallel disk I/O, overlap of I/O, computation, and pipelining of EM algorithms. Chapter 4 contains a description about buffer range search trees which is an external memory implementation of range search trees. In this chapter, we have described about data structure of buffer range search trees, and construction, search procedures involved in it. Chapter 5 contains a set of experiments and results that we have performed to evaluate the performance of our implementation of buffer range search trees. We conclude in Chapter 6 with our findings, and experimental results followed with a future scope, and applications of our implementation in the area of geometrical data structures in external memory.

Chapter 2

Preliminaries

External Memory algorithms are studied based on Input/Output(I/O) Model [13]. I/O Model describes the parameters related to main memory, external memory, and block size along with a relation between them. The design of EM Algorithms makes use of parameters defined in I/O model for easier, and better understanding. With the help of I/O model, it is easy to compare, relate between different EM algorithms for it's running time, and space complexity.

2.1 Input Output Model

Input Output Model [13] has the parameters N, M and B as shown in Table 2.1 where $M<N$. In EM algorithms, transfer of B elements in internal memory to B consecutive elements in external memory and vice versa are considered as single I/O operation.

N	Number of Input elements
M	Number of elements in main memory
B	Number of elements in a block

Table 2.1: I/O Model parameters [13]

In this model, we define metric based on number of I/O operations and consider that internal memory computations comes for free. There are few more parameters defined as n= N/B which gives number of blocks containing N input elements, and m = M/B which gives number of blocks in main memory of size M. For defining the asymptotic notations, we consider O(n) as a linear time in terms of number of I/Os performed by a particular EM algorithm on a single disk model [13]. In case of parallel disk model or D(at most 100) disk model, if no two blocks comes from the same disk while performing the I/O operation then D blocks can be transferred in a single I/O reducing the scanning of n blocks of elements from θ(n) to θ(n/D). One of the basic paradigms is sorting algorithms in external memory which takes $\theta(n \log_m n)$ [11] time to sort N input elements in single disk model, and $\theta(n/D \log_m n)$ [11] in D disk model. In the recent times, researchers has designed many external memory algorithms in different areas like computational geometry [14, 16], and graph problems [15].

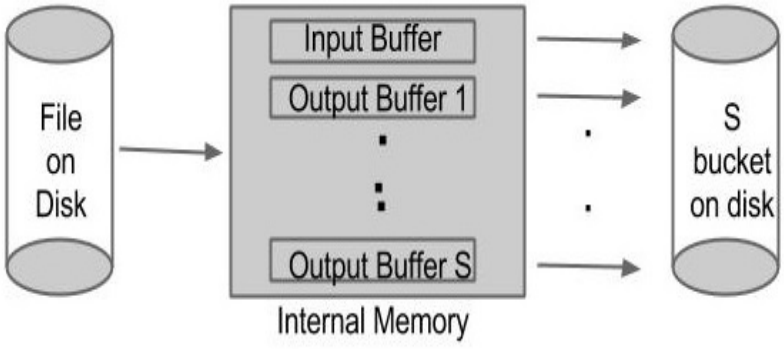

Figure 2.1: Distribution Sort Technique in External Memory [11]

2.2 External Memory Sorting

In external memory algorithms, external memory sorting [11] has always been a central problem and a paradigm in the design of efficient external memory algorithms. The asymptotic bound on the EM sorting is given by θ(n/D $\log_m$n) [11] in a D disk I/O model. Recently developed sorting algorithms in external memory are based upon *Distribution*(see Figure 2.1), and *Merge*(see Figure 2.2) paradigms.

2.2.1 Distribution Sort Technique in External Memory

In *Distribution* sort technique [11] in external memory, we assume that the input and output file are stored on two different disks to simplify the explanation of technique. In this technique, input data of N elements in the input file on disk is partitioned into S different sub files or buckets using partitioning elements defined as $\forall$ i=0 to (S-1), x_i such that for partition i, all elements are in the range [x_$i-1$, x_i]. Thus, S different buckets are sorted recursively, and independently in internal memory. Thus, S buckets stored back on the disk gives sorted N elements in θ(n $\log_m$n) [11] number of I/Os.

2.2.2 Merge Sort Technique in External Memory

In *Merge* sort technique [11] in external memory, input data elements on the disk is sorted in internal memory in N/M number of runs such that each run is sorted in internal memory, and output is written on the disk in series of stripes (see Figure 2.2). In the merge step, R runs such that R = θ(m) [11] are streamed through internal memory, merged, and written to disk on left as shown in Figure 2.2. We repeat the procedure until all N elements are merged, and written to the disk in O($\log_m$ n) number of pass [11]. Thus, merge sort paradigm in external memory takes θ(n $\log_m$n) [11] number of I/Os to sort n blocks of input elements.

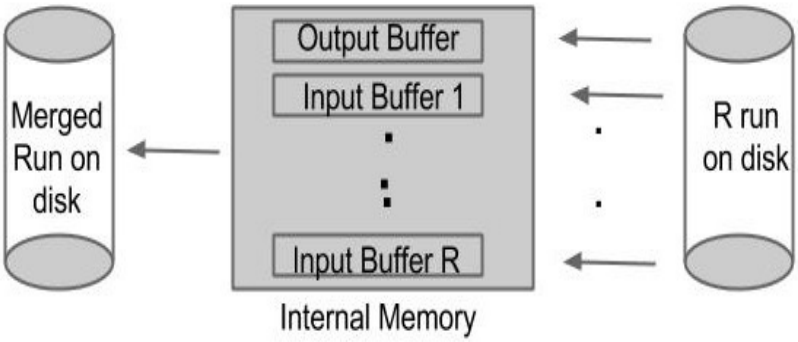

Figure 2.2: Merge Sort Technique in External Memory [11]

2.3 Memory Hierarchy

For implementing the EM algorithms, it is important to understand the memory hierarchy architecture. There has been always a speculation about the design, architecture, cost and performance of various devices in the memory hierarchy. Memory architecture will let us know about the essential need of EM algorithms. In the computer's memory hierarchy, each of the level has its own cost and performance such as cache memory is the fastest but costliest whereas EM is slowest but cheapest, so there is always a trade-off in between these two, cost and performance. The widely used and accepted architecture is *Parallel Disk Model(or PDM)* [1] which works on the specific level of Memory hierarchy in between the external memory and internal memory. However, for this reason, *PDM* can be ineffective or inefficient over the other levels and the solution to this problem is *Cache oblivious Algorithms* [2] which is not dependent on particular system parameter(Internal Memory Size, Block Size) and works equivalently over all memory levels unlike *PDM* which knows about the system's memory and block size beforehand and gives good efficient performance only at a particular level in between external memory and internal memory. However, *Cache Oblivious Model* have a big hidden constant in its *I/O* complexity unlike *PDM*. So, this is why, many EM algorithm libraries or framework [6] follows and uses *PDM* .

2.3.1 Parallel Disk Model

The *Parallel Disk Model* [1] describes memory hierarchy in an elegant way. As shown in Figure 2.3, the memory hierarchy consist of the processor(or CPU) which has its own set of registers, caches at different levels, main memory, and external memory in the form of parallel disks. As we see t_i denote transfer rate in Figure 2.3, t_i at level i is the transfer rate for that particular level in the hierarchy. Each level i has its own cost and performance. Cache is the fastest but costliest memory whereas external memory is slowest but cheapest memory. So, for storing larger amount of data, today's computers make use of external disks. However, the average access time for registers that are within the processor(or CPU) is 1 nanosecond [11] (approximately), while access time for cache memory is few nanoseconds [11], multiple nanoseconds [11] for internal memory whereas access time for external memory is multiple milliseconds [11] which is almost 10^6 slower than the access time for internal memory. Hence, in practice, it becomes very essential

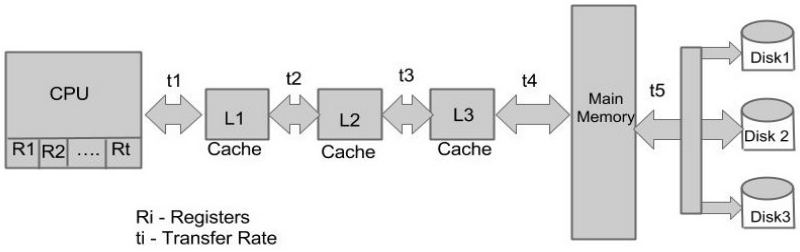

Figure 2.3: Parallel Disk Model in Memory Hierarchy [1]

and necessary to minimize to access to the external memory which could be achieved only with the help of design and implementation of EM data structures and algorithms. Based on the access time across different levels in memory hierarchy, we can conclude that transfer rate across different levels in memory hierarchy is related as :

$$\forall i, t_i > t_{i+1}$$

2.4 External Memory Data Structures and Algorithms

We can observe and understand the need of efficient EM algorithms or out-of-core algorithms so as to reduce the I/O cost. The EM algorithms that we are considering are for batched problems wherein all the data is available at the initial stage itself, and the complete data is required to be processed in one or multiple iterations in internal memory. There has always been a concern that why do we really need EM algorithms if the underlying operating system(or OS) have its own virtual memory scheme which makes the address space look larger using paging. The virtual memory mechanism in modern OS works on the principle of locality of data by caching and pre-fetching the more frequently visited pages or blocks in the internal memory cache and paging out the less frequently visiting pages or blocks. However, for more complex data, virtual memory can often turn out to be non-productive causing more swap access to the disk and as a result, more number of I/O [11] operations.

For having an explicit control over the number of disk I/O, we need EM algorithms. There has been a lot of work done in the area of EM algorithms.Many algorithms in the field of graph algorithms, string processing algorithms and computation geometry [3, 4] have been designed in the past. The implementation of these EM algorithms will have a long lasting tangible effect in the field of computer science because these EM algorithms have many pratical applications.

Implementation of EM algorithms hence, need libraries and frameworks that are capable of operating and working at system's low level, handling the I/O details efficiently and in an abstract manner. In next subsection, we briefly describe I/O efficient buffer tree technique [10] in external memory.

2.4.1 Buffer Tree Technique

The migration of internal memory tree data structures to external memory can be done using the buffer tree technique [10]. It involves grouping of internal nodes(nodes which do not have leaves as their children) and leaf nodes(parent nodes of leaves) such that maximum fanout of

nodes is θ(m) [10]. Hence, a buffer tree with n blocks of input elements has O($\log_m$n) height with n blocks of data elements stored in the leaves of the tree. Internal nodes has a buffer of size θ(m) [10] and buffer tree technique operates in a lazy update manner.
In lazy update technique, we do not update the leaves of the tree for each input element unlike internal memory tree data structures but we insert 'm' blocks(M elements) of input data into the buffer of root node in buffer tree. Once, the buffer of root node is full, we apply buffer emptying process [10] which involves sorting and inserting buffer elements from root node into the buffers of corresponding internal nodes one level down the tree followed with a recursive buffer emptying process for a child node with full buffer. The total cost of an arbitrary sequence of N insert or delete operations in an initially empty buffer tree takes O(n $\log_m$n) [10] number of I/O operations with an amortized cost of O($\log_m$n /B) I/O operations. Since, data elements are stored in the leaves in buffer tree, internal node stores an array of 'm' elements storing the largest element in its corresponding child node one level down the tree. Buffer tree stores the data in sorted manner in its leaves and a query to report the elements takes O(n) [10] I/O operations. We have used this buffer tree technique of lazy updates in our implementation of buffer range trees which is illustrated in Chapter 4.

Chapter 3

Standard Template Library For Extra Large Data Sets- STXXL

The implementation of Input/Output(or I/O) algorithms need a platform, framework which will be able to handle the I/O explicitly, and a library which offers the implementation of the basic data structures in EM in I/O efficient manner.

One such library that has been used by many people in the field of I/O algorithms is Standard Template Library For Extra Large Data Sets(or STXXL) [6]. It is a framework library implemented over the low level details of the system's external memory and explicitly using the operating system details. The data structures defined in STXXL are analogous to C++ STL [7] such that data structures and algorithms in C++ STL directly applies and are implemented in STXXL. It consist of many basic data structures like *std::vectors*, *std::priority queue*, *std::set*, *std::multiset*, *std::deque* . So, STXXL has the implementation of all these basic data structures which can be used to build and develop other complex data structures like trees, graph on top by using these basic data structures as the basic entity. The major advantage of using STXXL algorithms is that they are not container bound which means that STXXL algorithms can be used with the data in any type of container *viz.* *stxxl::vectors*, *stxxl::priority queue*, *stxxl::set*, *stxxl::deque*. Moreover, it is a generic kind of library which has been widely used throughout the community and well accepted for the framework it has followed similar to C++ STL.

3.1 Design of STXXL

STXXL has been designed and implemented with an aim to bridge the gap between theoretical and practical implementation of the EM algorithms. Previously implemented library for out-of-core computations are LEDA [8] and TPIE [9] which have an implementation of basic data structures for EM algorithms so that the user need not worry and look into the details of how the system I/O are performed. However, STXXL provides support of parallel disk I/O and algorithm implementation unlike other libraries LEDA and TPIE. STXXL even supports overlapping of I/O and computation which makes it stand way ahead of other libraries in terms of I/O computation and efficiency.

STXXL library operates at low level between the operating system and applications running on the system(see Figure 3.1). As shown in Figure 3.1, STXXL has four major layers *viz.* STL-user layer, Streaming layer, Block management(BM) layer, and Asynchronous I/O primitives(AIO) layer. STXXL design depends on uncontrolled operating system I/O caching and buffering which

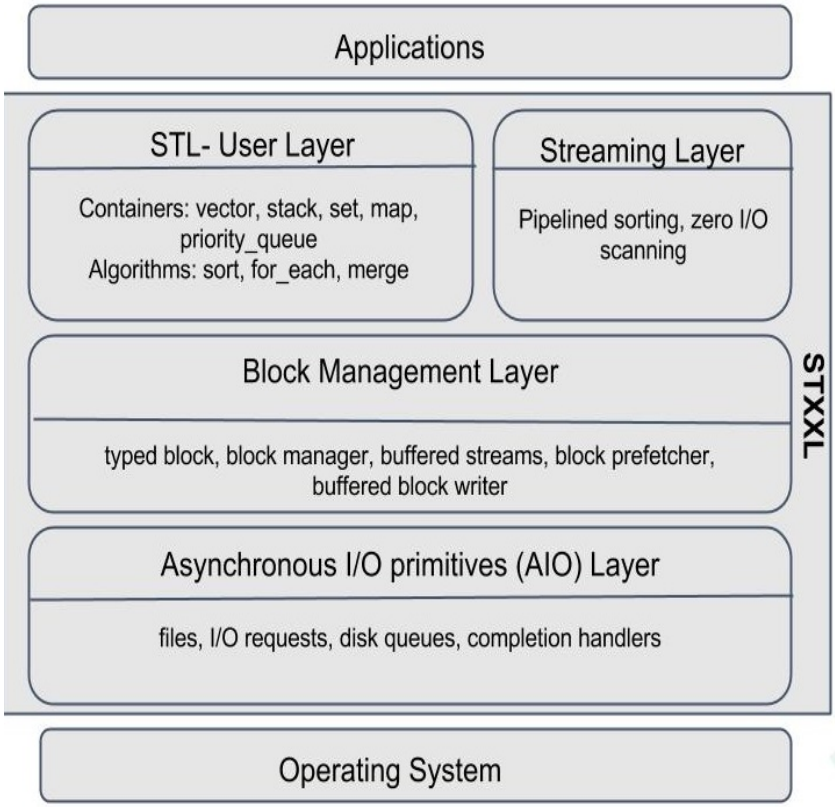

Figure 3.1: Structure of STXXL [6]

allows it to overlap I/O and computation. In Streaming layer, it uses the concept of Pipelining [6] which involves the coupling of the input and output of various algorithm components such that output from one component of the algorithm is passed as an input to another component without writing it back to the external memory and thus, saving the I/O cost. STL-user layer consist of containers like *vector*, *set*, *stack*, *map*, *priority_queue* and algorithms like *sort*, and *merge*. These containers and algorithms in STL-user layer has an implementation in STXXL which can be used in implementation of external memory data structures and algorithms. Block Management layer provides an interface imitating PDM. Block management(BM) layer manages the block of elements in every aspect. It controls the block allocation/deallocation in internal memory, writing block to external memory strategies. Thus, taking an explicit control over the I/O happening in the swap in/out of the data between internal memory and external memory. BM layer support overlapping of I/O and computation. The lower layer of Asynchronous I/O primitives handles the I/O requests, and communication of STXXL with the underlying operating system. Other EM libraries like TPIE [9], and LEDA [8] has a synchronous I/O API layer which makes the porting of the library difficult on different operating systems. However, unlike TPIE [9], and LEDA [8], STXXL's Asynchronous I/O layer makes the porting easy which requires only reimplementation of this layer on different platforms like windows, and linux. STXXL has been used in implementation of various data structures and algorithm problems in external memory like minimum spanning tree, connected components, breadth first search, suffix arrays, and social analysis matrices [6].

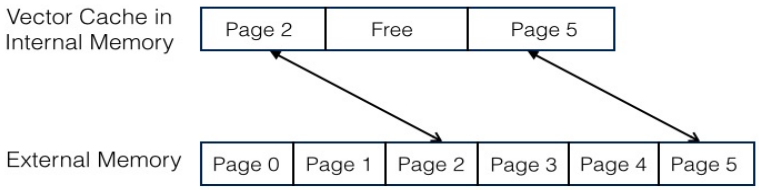

Figure 3.2: stxxl::vector Architecture [6]

3.2 STXXL Vector

One of the basic data structures implemented in STXXL is *stxxl::vector* which is an unbounded array with dynamic size. The architecture of *stxxl::vector* is as depicted in Figure 3.2.

stxxl::vector holds a fixed size of cache in internal memory. One random access causes $O(1)$ I/O in worst case. *stxxl::vector* is a container that is stored in the external memory in a collection of blocks. The cache in the internal memory follows a full associative strategy. As shown in the Figure 3.2, Page2 and Page5 are in the vector's cache in internal memory from total 6 pages which are stored in external memory. The page replacement strategy followed by default is Least Recently Visited(or LRU) . Whenever, any element in the *stxxl::vector* is accessed using *at()* function or *operator[]* then it first checks for that element amongst the pages available in the cache, if not available in the cache in internal memory then the element is required to be accessed through external memory where its corresponding page is bought into internal memory using the LRU (default strategy) or any other page replacement strategy specified. Each page in internal memory's *stxxl::vector* cache is associated with a dirty bit flag which is by default set to '0'. The dirty bit flag for any page is set to '1' only when a non-constant reference is made to that particular page and any element in that page is modified. The dirty bit flag set to '1' indicates that this particular page is supposed to be written back to disk when chosen for replacement by page replacement strategy because replaced page contains modified and updated data. Hence, a typical random access to any element takes 2*Blocks per page number of I/Os in the architecture of *stxxl::vector*.

As per the architecture of *stxxl::vector*, it is very much visible that it takes allocation size of *BlockSize*CacheSize*PageSize* size in the internal memory for each vector cache. While using *stxxl::vector*, a proper care is supposed to be taken such that we do not make any references to the elements in it because the pages in the vector swaps in/out from the external memory, the reference made previously will be invalid and it will throw a memory reference error or an invalid access error causing memory leak.

Chapter 4

Buffer Range Trees

In this chapter, we describe construction and query on buffer range trees. Range tree [5] is a data structure which can store data points in one or more dimensions and answer range queries. The range tree data structure is defined such that it can answer rectangular or box range queries in d-dimension. Range queries has applications in the area of geometrical points data, geographical data based on latitude-longitude in maps.
Consider a dataset of employees in a certain organization, each employee's details contains name, age, salary, and years of work experience. We have a query to report list of employees with their age in range [a1, a2], salary in range [s1, s2], and years of work experience in range [y1, y2]. A naive solution to answer such a range query would be to sort the dataset on the basis of age, salary and years of work experience respectively and answer the intersection of the result obtained in each of the three range searches. This technique do not gives us a data structure to answer queries in two or more dimensions. Range trees can answer such query giving employees list with their age in a range [a1, a2], salary in a range [s1, s2] , and years of work experience in a range [y1, y2] in O($\log^3$ n) [5] running time in 3-dimensions. This is one of the examples for which range trees can be used to answer range queries in one or more dimensions. Geographical dataset points based on latitude-longitude can be stored in range trees and range queries to find a particular label like hospitals, restaurants in a given range can be answered by buffer range trees in O(n) [10] number of I/O operations.

4.1 2-Dimensional Range Trees

The range trees in 1-dimension is constructed with data points stored in its leaves. The internal nodes in the range tree stores an array(size of array is equal to number of children for current internal node) of values which contains largest value in its corresponding child nodes. Thus, for a given range query, we can traverse down the tree, starting at root to find split node and answer range queries in O($\log_p$ n+k) [5] running time in 1-dimension where p is fanout in range tree and k is number of elements reported in range search. If we use range trees in two or more dimensions, a range query is answered in O($\log_p^d$ n+k) [5] running time in 1-dimension where d is number of dimensions, p is fanout in range tree and k is number of elements reported in range search. However, for a range search in further dimension, we search in the associate tree at split node [5] which has less than N number of elements in the tree unlike naive implementation which would involve sort and range search for each dimension of N elements.
The construction of range trees is also based on the same principle of balanced binary search trees like AVL trees or Red-Black trees but in range trees, data is stored in the leaves unlike

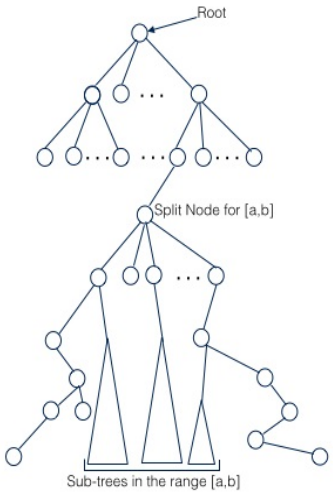

Figure 4.1: Range Tree in 1-Dimension [5]

other balanced binary search trees wherein internal/non-leaf nodes also contains the data. In range trees, thus, data being at the leaves, sub trees at each of the internal node in the tree stores the information about its corresponding leaves, like it can store the smallest and greatest value in that particular sub tree or only a maximum value in its left sub tree for all of its children. Range Tree in 1-dimension is as shown in Figure 4.1.

1-dimensional range tree as shown in Figure 4.1, once we get to the split node where search for 'a' and 'b' in [a,b] splits into left and right sub tree of the split node. While searching in the left sub tree for 'a', all the elements in the right sub tree along the path down the tree starting to the left of split node belongs to the given query range of [a,b] whereas the elements in left sub tree along the path down the tree starting to the left of split node do not belong the given range of [a,b]. And an exact mirror image of it is applicable for right sub tree of 'b' starting at the split node in which all the elements in the left sub trees along the path down the tree starting to the right of split node belongs to the range in the query whereas elements in right sub tree along the path down the tree starting to the right of split node do not belong to the given range in the query of [a,b].

In Figure 4.1, we have shown a 1-dimensional range tree with a fan out that can range from 2 to p where p is some positive integer. In case, it is greater than 2, at each internal node, we can store maximum value in its p children in an array of p elements. Thus, range tree can also be used to find number of elements in a particular sub tree by storing the count at each internal node when data elements pass through that node down the tree. One of the greatest advantage of using range trees is that it can be easily generalized to d-dimensions and n-fan out unlike balanced binary search tree which is used in 1-dimension.Thus, range trees can answer the range query in $O(\log_p n + k)$ [5] where k is the number of elements in the output in 1-dimension.

As mentioned, range trees can be further expanded to d-dimensions. For that purpose, we will define and study the case of 2-dimensional range trees.

In Figure 4.2, data points are in 2-dimension and a query range is defined in [(x1,y1) , (x2,y2)] and we need to report all the points in this range as an answer to the query. In such a scenario, range tree can be used to answer such queries. For the construction of range tree, first the data

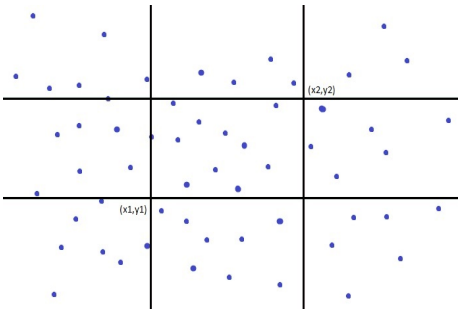

Figure 4.2: Data Points in 2-Dimension

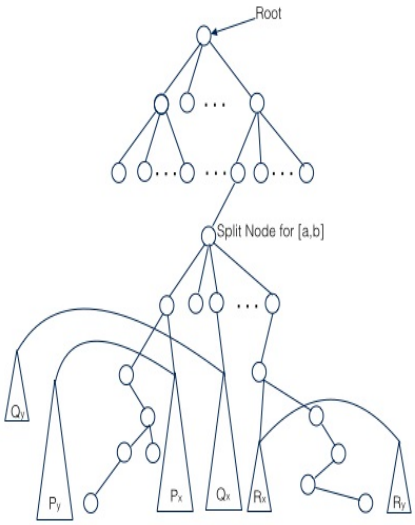

Figure 4.3: Range Tree in 2-Dimension [5]

points are stored in the range based on their x-coordinates.

Once, range tree is constructed in 1-dimension based on x-coordinates, we can look up at each and every internal node in the tree, and at each internal node in the tree, construct a range tree in 1-dimension based on y-coordinates for the data elements in sub tree of x-coordinate based tree and link the new y-tree's (or y-associate tree) root node to the corresponding internal node in the x-tree. Thus, given an internal node, it will have a linked y-associate tree containing all the data points stored based on y-coordinates in sub tree of the given internal node in x-tree. The structure of 2-dimensional range tree is depicted and visualized in the figure 4.3. However, range tree being stored in further dimensions consumes lots of space and it takes θ(n $\log_p$ n) [5] space in 2-dimension because each point is stored in ($\log_p$n) number of y-trees where p is the fan out of the range tree. Further, searching in a particular x-tree's internal node's y-associate tree takes O($\log_p$n) time and thus, total search time in 2-dimension for basic range search tree is O($\log_p^2 n$+ k) [5] where k is the number of data points in the result set. The dimension can be extended further such that constructing third dimension associate trees on each internal node in y-trees and it goes on till d-dimension. But for now, we are interested in and will look into 2-dimensional range search trees.

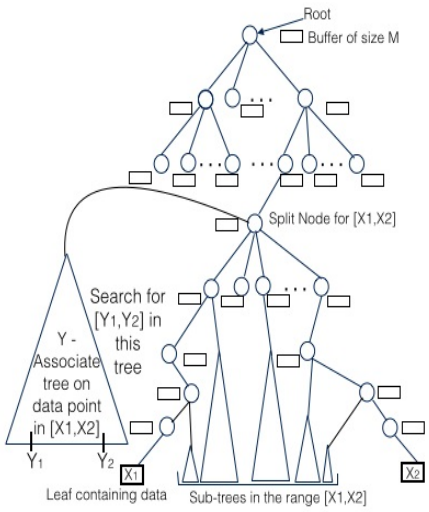

Figure 4.4: 2-Dimensional Buffer Range Tree [5, 10]

Theorem [5] 4.1.1 *For a set of points in d-dimensions and fanout p, range trees can answer range queries in O(log_p^d n+k) time with a space of θ(n log_p^{d-1} n) where d >1 and k is the number of data points reported in range search.*

4.2 2-Dimensional Buffer Range Trees

2-Dimensional Buffer Range Trees involves the use of lazy update buffer technique to implement the tree in external memory. Since, we have now learnt about the range tree data structure in internal memory, if we want to pull in more data which cannot fit into internal memory then we need external memory data structures. In external memory data structure techniques, buffer tree technique [10] is a widely used technique for various I/O algorithms. In case of range trees, we have tried to incorporate this technique for the update of data in the leaves of the range tree such that buffer range tree will be able to hold out-of-memory data and do out-of-core computations. The buffer tree technique includes buffer of size M (size that can fit in internal memory) attached with each of the internal node in the tree. In case of external memory algorithms where I/O cost is the bottleneck, we are interested in storing larger size of data which cannot fit into usable internal memory.In our implementation of buffer range search tree, we basically have a set of operations involved in the construction of a tree. The basic operations includes SplitRootNode, SplitInternalNode, EmptyInternalBuffer, Search in a range of [x1 , x2] and the construction of y-associate trees for each of the internal node in x-tree. In case of our implementation, it should be noted that internal nodes along with empty buffers are stored in internal memory pointing to caches of data in leaves. These caches of data in internal memory are part of the larger set of data in external memory which is accessed through disk I/Os.

In construction of the tree, we do not flow down the path of the tree to the leaf for each and every single element in an input unlike internal memory version of range trees, however, we wait till we have collected M number of elements in buffer of root node in our tree, once buffer is full then we call Empty Internal Buffer procedure to empty the buffer and move its block elements

into their corresponding child buffers one level down the tree. And we recursively repeat the process till we reach the leaves of the tree and empty the buffer into its corresponding leaves at the leaf node. Using this technique, we are able to avoid access to leaves for each and every single input element. It takes O(n log n) number of I/O operations [10] in one buffer for 'n' blocks of input data in 1-dimension. Our implementation includes the use of this technique. So, once we construct the buffer range tree, range search queries can be made to the tree which theoretically takes running time of $O(\log^2 n + k)$ [5] in its search operation where 'k' is number of elements in the result set. And in case of external memory implementation, it takes O(n) number of I/O operations [10] to report sorted elements in a certain range and in our case of 2-dimensional implementation, we need to run this operation twice, one for x-tree and another for y-associate tree respectively.

The basic operations in our implementation can be mentioned in brief. These operations are involved and part of construction of buffer range search tree [5] [10] that we have used. First operation of Split Root Node involves an increase in level of the tree by one if non of its child is empty. The procedure for the same is given in Algorithm 1.

Algorithm 1 Procedure : Split Root Node

SplitRootNode()
1. If non of the child of Root node is empty
   then request a new node and make that new node as the parent of current root node.
   Since, we have a new Root node, split the old root node such that half of its child nodes are passed to its sibling.
   Return
2. Else
   Return

Now, in case of Split Internal Node, unlike Split Root Node, here we do not need a new root node or a new parent node but it includes passing half of the child nodes of current node to its new empty sibling. Procedure for Split Internal Node is as given in Algorithm 2.

Algorithm 2 Procedure : Split Internal Node

SplitInternalNode(n)
1. If node n's child pointers are full then
   Add an empty node as a sibling of current node n by shifting its old siblings to the right by one position.
   Pass half of the child nodes of current node n to new empty node
   Update current node n and new node's details.
   Return
2. Else
   Return

We need Split Internal Node only when the corresponding child node in which buffer is going to insert data is full. One of the major steps in the construction of buffer range tree is an empty internal buffer process which involves emptying the buffer elements to its corresponding buffer once it is full. The procedure for Empty Internal Buffer is given in Algorithm 3. The procedure defined in Algorithm 3 describes the steps involved in emptying the internal buffer when it becomes full. With these set of procedures we can construct a buffer range tree in 1-dimension.

Algorithm 3 Procedure : Empty Internal Buffer

EmptyInternalBuffer(Node n)
1. Base Case: Empty Leaf Buffer if n is Leaf Node or its buffer is full
2. If n is an internal node then for each block in its full buffer of size M
    2.1] Search corresponding child of node n to which block elements should be inserted
    2.2] If corresponding child not found
        2.2.1] If Node n's last child is full then call Split Internal Node
        We can insert block data in last child of Node n
    2.3] Else if child found
        Repeat step 2.2.1 for correct Node child's buffer
    2.4] If the corresponding child's buffer is full now
        then call recursively Empty Internal Node(n.Node Child)
    2.5] Else
        Delete last block from n's buffer and move to next block
3. Return

For, extending it to another dimension we need to construct the associate tree for each internal node in currently constructed x-tree. Traversing of each node in x-tree can be done using preorder traversal and constructing associate y-tree for each internal node that comes across in traversing the path.

Algorithm 4 Procedure: Search in range[(x1,y1) , (x2,y2)]

Search(x1, x2, y1, y2)
1. Find Split Node V for [x1, x2]
    1.1] Traverse Sub tree at V using Postorder traversal
    and report all the elements which are in range [x1, x2]
2. Find Split Node U for [y1 , y2] in y-associate tree at V
    2.1] Traverse Sub tree at U using Postorder traversal
    and report all the elements which are in range [y1, y2]
3. Report intersection of elements reported in step 1.1 and 2.1
    which are the elements in the range query in 2-dimension
4. Return

If our buffer range tree is constructed in 2-dimension, we are interested in making a range query to the tree which includes the search procedure for the correct split node and then all the elements in that sub tree are reported to be in range of [x1, x2]. And as shown in Algorithm 4, for the search in y-dimension, we call y-associate tree at split node.

Chapter 5

Experiments and Results

We have done a set of experiments and evaluation to understand and interpret the behavior of an implemented buffer range tree. We did 68 different iterations of experiments in 1-dimension and 2-dimension. In our implementation, since, tree nodes except the leaves which contains the data are stored in internal memory, we need an internal memory to store these internal nodes. In worst case, it may happen that for a particular node 't' , all its children's buffer of size 'M' are one block short for getting full and thus, being recursively called for emptying that buffer, In such a scenario, we need O(Number Of Child of t * M) size of internal memory. And for the leaves which have its data in external memory, contains a small block sized cache in internal memory. We have observed through our experiments that there is always a trade-off in between the leaf cache block size and the number of I/Os which are inversely proportional to each other. However, we have a limitation for setting the leaf cache block size as well because it needs space in internal memory and as number of leaves increases in the tree, it occupies more space in internal memory though it can store much more data in its external memory. We have used std::random_device random number generator function in C++ in our implementation to generate random numbers in a defined range of $1{:}10^7$ for both 1-dimensional and 2-dimensional tree experiments. This random generator uses std::uniform_int_distribution<> to generate a random number in a defined range. For 2-dimension, since we call the random generator function independently, the value generated for x,y in a pair of [x , y] are also independent. We did our experiments on the dataset of open street map(or OSM) where we have used the latitude-longitude details in the dataset of OSM input files to construct EM data structure.

The statistics that we have mentioned in construction time of both 1-dimension and 2-dimension also includes the stack in external memory that we used in our implementation, external memory sort for buffer elements but since all these are part of preprocessing in constructing the tree, we have not ignored them . For each input of y-associate tree, we are storing the input data for next y-associate tree in a file and file I/O itself is very much time consuming. So, our statistics includes the time for writing a file as well. We did our experiments for $M=4*10^6$ and different block sizes of 512KB, 768KB and 1MB. The configuration of the system on which we did all our experiments is with an usable(or available) internal memory of size 768MB and an *Intel(R) Core(TM) i3-4030U CPU@1.9 GHz.* processor.

5.1 1-Dimensional Buffer Range Tree

For 1-dimensional buffer range tree , we ran iterations to find the break point here the data cannot fit in internal memory which is shown in Figure 5.1 . It shows that internal memory

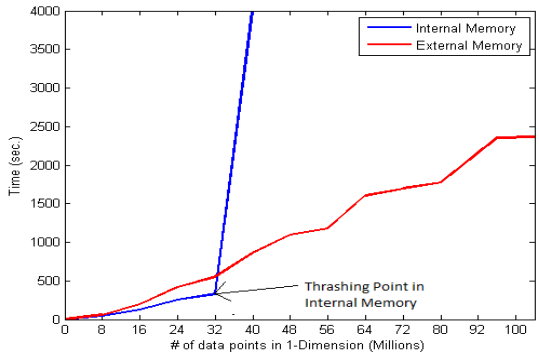

Figure 5.1: 1-Dimension: Internal Memory Thrashing point

cannot store data elements beyond 32 million and we did our external memory experiments up to 104 million input data elements on same system configuration. It can be observed in Figure 5.1 that running time for tree construction in internal memory is less than running time in external memory which have a valid reason of the disk I/O occurring in external memory implementation. In next two sections, we will evaluate the construction and search in range search tree in 1-dimension.

5.1.1 Preprocessing and Construction

For preprocessing and construction of buffer range tree in 1-dimension, we did 32 iterations of experiments for different number of input elements with different block sizes of 512KB, 768KB and 1 MB with buffer size M(# elements in the buffer) $4*10^6$. Figure 5.2 shows the number of I/Os comparison for range tree construction in 1-dimension with different block sizes of 512KB, 768KB, 1MB and 2MB (we recorded only number of I/Os with 2MB of block size). As we can observe in the figure that the # of I/Os is inversely proportional to block size because larger block can transfer more number of elements between internal memory and external memory in one I/O cost. Table 5.1 and 5.2 shows the statistics for tree construction with different block sizes of 512KB, 768KB and 1MB.

#Data Points	Total I/O Time(sec)			Rate Of I/O(MBps)		
(Millions)	512KB	768KB	1MB	512KB	768KB	1MB
8	3.74	3.55	5.81	106.92	1133.4	68.14
16	12.701	14.41	15.13	133.77	103.17	96.88
24	38.62	28.76	32.78	79.49	107.63	93.2
32	38.72	40.03	42.73	110.22	107.5	99.45
40	108.3	53.05	75.85	59	110.94	76.64
48	84.55	106.63	86.64	93.03	74.38	90.44
56	100.7	86.24	86.22	91.2	102.75	101.57
64	239.8	128.39	118.26	46.52	90.73	97.33
72	192.6	118.11	143.05	66.6	109.6	89.43

Table 5.1: Construction Statistics 1: 1-Dimension

#Data Points	Total Time(sec)			Dirty Writes(Dirty Bit=1)		
(Millions)	512KB	768KB	1MB	512KB	768KB	1MB
8	61.406	63.5	70.64	220	180	84
16	199.63	205.34	218.95	1100	600	280
24	420.8	405.09	435.24	2400	1200	560
32	552.47	593.26	565.84	3240	1620	756
40	867.56	797.4	799.38	4800	2160	1008
48	1099.2	1141.9	1041.4	5760	2880	1344
56	1180.3	1175.6	1137.3	6600	3120	1456
64	1607.6	1540	1488.8	8040	4200	1960
72	1721.1	1629.7	1696.2	9120	4560	2138

Table 5.2: Construction Statistics 2: 1-Dimension

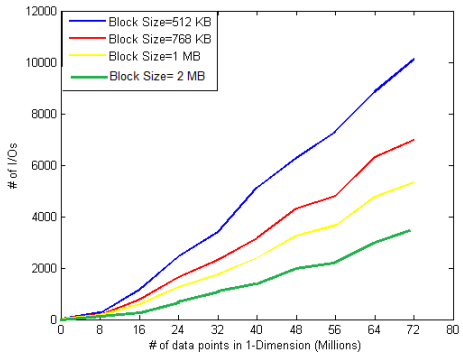

Figure 5.2: Buffer Range Tree Construction in 1-Dimension

We have observed that though # of I/Os reduces and shows a visible change with the change in block size with exactly same parameters, the I/O time depends on the rate of I/O between external and internal memory which we are not controlling through our EM algorithm. Another table 5.2 shows the dirty writes which is writing back of modified data block back to external memory if we are reading a new block from external memory. And design of stxxl::vector is such that in worst case it requires 2*I/Os [6] for each and every single access to the elements for this reason. The total time for all block sizes can be observed as almost same which is because of file I/Os that we are performing and it is a major contributor to the total running time.

5.1.2 Range Search in 1-Dimension

After construction of buffer range trees, we made range search queries on the constructed tree. Figure 5.3 shows the # of I/Os for search in 1-dimension. It takes O(n) [10] # of I/Os for reporting elements in constructed buffer range tree in a given range. In Figure 5.3, maximum I/Os represent the upper bound on # of I/Os allowed for particular number of elements and it can be observed that we have achieved to get the # of I/Os within the upper bound. The steep rise and fall in the curve is because of the range of query or we can conclude that higher the level of split node, higher will be the number of I/Os because we will access more number of leaves.

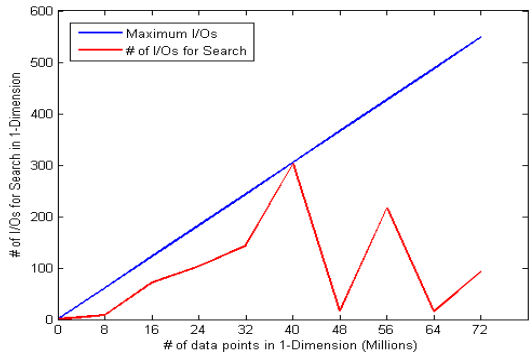

Figure 5.3: Search in 1-Dimension: Block Size: 1MB

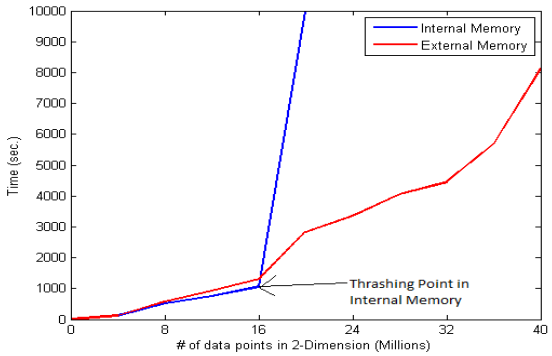

Figure 5.4: 2-dimension: Internal Memory Thrashing Point

5.2 2-Dimensional Buffer Range Tree

Construction statistics of 2-dimensional buffer range trees is going to vary than statistics of 1-dimensional buffer range search tree. The thrashing point in internal memory is as shown in Figure 5.4 which shows that internal memory cannot store data points beyond 16 million.

Further, we did 30 iterations of experiments in 2-dimension with different block sizes to evaluate the behavior and performance of the 2-dimensional buffer range tree.

5.2.1 Preprocessing and Construction

Figure 5.5 shows the # of I/O comparison for range tree construction in 2-dimension with different block sizes.As we can observe in the figure that the # of I/Os is inversely proportioanal to block size because larger block can transfer more number of elements between internal memory and external memory in one I/O cost.

We can also observe that there is an abrupt increase in # of I/Os at some points for all block sizes which occurs whenever the level(or height) of the tree increases because with increase in number of level, number of y-associate trees also increases. Table 5.3 and 5.4 shows the statistics for tree construction with different block sizes.The statistics in tables 5.3 and 5.4 shows that the I/O time is dependent on the rate of I/O and the parallel I/O architecture of stxxl::vector that has been used. Total time for the construction of buffer range tree has a major contribution of file I/O that we do to collect the input data for y-associate tree construction.

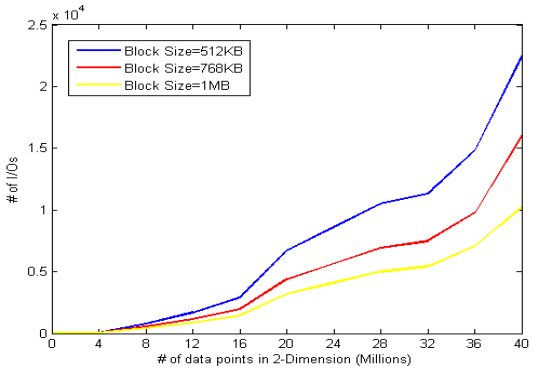

Figure 5.5: Buffer Range Tree construction in 2-Dimension

#Data Points	I/O Time(sec)			I/O Rate(MBps)		
(Millions)	512KB	768KB	1MB	512KB	768KB	1MB
4	2.1	1.73	1.84	87.49	106.17	97.7
8	11.12	16.47	12.12	99.6	67.85	90.76
12	32.53	23.8	23.96	67.08	92.41	90.55
16	38.41	34.86	43.06	92.7	103.14	83.39
20	121.02	98	83.74	64.5	80.4	92.95
24	120.08	102.49	104.78	83.15	98.43	95.03
28	167.95	129.56	134.82	72.47	94.87	90.04
32	175.44	130.88	135.62	75.32	101.94	97.16
36	193.18	197.47	190.19	89.07	87.99	90.22
40	328.83	304.46	295.03	78.15	92.28	83.18

Table 5.3: Construction Statistics 1: 2-Dimension

5.2.2 Range Search in 2-Dimension

Range search [(x1,y1) , (x2,y2)] in 2-dimension involves the search for data elements in the range of [x1, x2] first in buffer range tree constructed on x-coordinate of the data elements. Once, we have data elements in x-range, we can now search for data elements in y-range [y1, y2] on the y-associate tree at split node in x-tree. The intersection of reported points in the range [x1, x2] and [y1, y2] gives us the result of all the data elements in range [(x1,y1), (x2,y2)] . The range search needs O(n) number of I/O [10] in 1-dimension where n is number of blocks required to hold all N input elements and since, we do a range search in 2-dimension, we still need O(n) number of I/Os with a constant factor of 2.

We did range search in 2-dimension experiments with different block size of 512 KB, 768 KB and 1 MB. The number of I/Os to do range search in 2-dimension with an upper bound of O(n) is given in Figure 5.6, 5.7 and 5.8 . The number of I/Os in range search fluctuates which is because of the range of query data elements. If the search range is large enough then it will access more number of leaves thus causing more number of I/Os. However, we are more interested to see that the number of I/O remains within the upper bound defined [10].

#Data Points	Total Time(sec)			Dirty Writes(Dirty Bit=1)		
(Millions)	512KB	768KB	1MB	512KB	768KB	1MB
4	152.34	134.69	131.43	242	120	56
8	578.55	594.06	572.34	960	480	224
12	936.56	953.6	922.13	1800	900	420
16	1303.42	1304.31	1314.17	2880	1440	672
20	2887.58	2864.9	2844.84	6000	3000	1400
24	3494	3355.09	3371.34	7560	3780	1764
28	4223.38	3956.3	4063.25	9120	5560	2128
32	4373.9	4430.7	4455.36	9840	4920	2296
36	5747.19	5762.6	5701.09	12960	6480	3024
40	8477.36	8638.2	8116.61	18730	10140	4144

Table 5.4: Construction Statistics 2: 2-Dimension

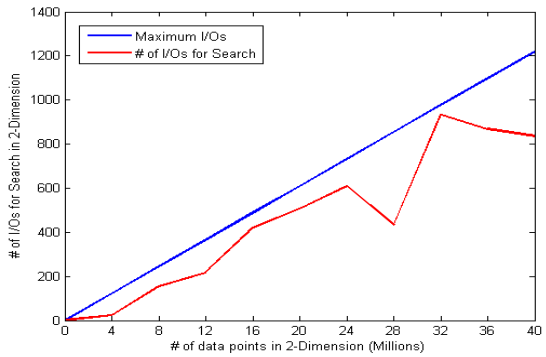

Figure 5.6: 2-Dimension Range Search: Block Size: 512 KB

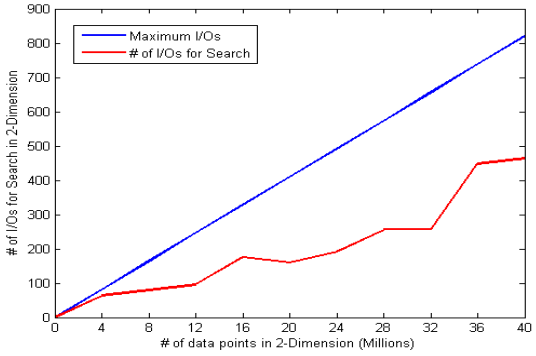

Figure 5.7: 2-Dimension Range Search: Block Size: 768KB

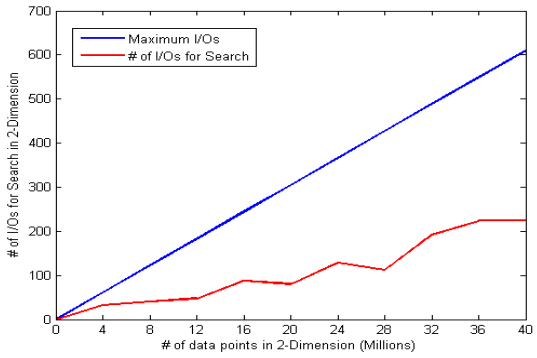

Figure 5.8: 2-Dimension Range Search: Block Size: 1MB

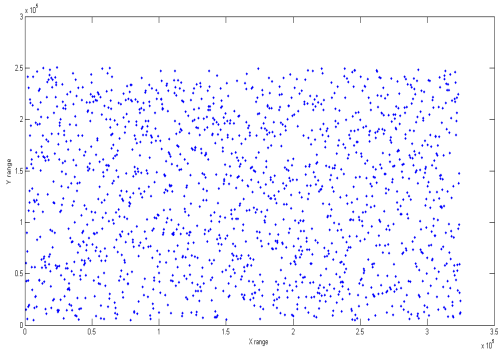

Figure 5.9: Elements in Range Search [(750,4500) , (325000,250500)] with 40 Million data elements

As a proof of concept, for range search in [(750,4500) , (325000,250500)] with 40 Million number of Data elements in 2-dimension, the range search result that we obtained are as shown in Figure 5.9 .

5.3 Preprocessing,Construction and Query on Open Street Map DataSet

We did our experiments with an open street map(or OSM) [17] latitude-longitude based dataset to observe the working of our buffer range tree data structure. Open street map [17] has collected its dataset from scratch over the years since its inception in 2004 with the help of crowd sourcing of around 2 million users across the globe. The data is generally collected using various GPS devices, aerial photography,etc. The software that can be used to store and edit the collected data by OSM volunteer users is named iD written by MapBox. The major contributors in collecting the data for OSM are cyclists, GIS professionals and many more. OSM supports route planning using services of open street route map(or OSRM), Graph Hopper and Map Quest. The data stored in OSM data set is in the form of topological data structures in xml files with an extension of *.osm* . The data of OSM mainly consists of nodes to determine latitude and longitude, tags to store the meta data of a particular node, ways with an ordered list of nodes to determine the path and relations to show the relation between different nodes and ways. The

Country	Size parameters		
(Millions)	Block Size(KB)	Buffer Size(Millions)	Maximum Memory used(MB)
Monaco (0.016)	0.5	0.004	50
Serbia (3.6)	32	0.4	100
India (36)	1024	4	768

Table 5.5: OSM Data Statistics:1

Country	Construction Statistics		
(Millions)	# of I/O	Total Time(I/O Time)(sec)	Dirty Writes
Monaco (0.016)	1024	8.18 (4.141)	1152
Serbia (3.6)	8486	673.83 (49.775)	7792
India (36)	6160	5427.36 (168.917)	2576

Table 5.6: OSM Data Statistics:2

dump of data in OSM is available for each country in a separate OSM file and a complete dump of global data is available in *planet.osm*. In all our experiments, we used different sizes of blocks, buffer as per the amount of data we are storing and these details are explicitly mentioned for each experiment.

We did our first and foremost experiment with a small data set of country *Monaco*. We constructed buffer range tree by extracting node data of latitude and longitude from the OSM file of Monaco. We stored latitude-longitude and node details in our buffer range tree. Statistics for construction of Monaco, Serbia and India are mentioned in table OSM Statistics. We chose different block, buffer sizes as per the amount of data available in that particular country's dataset.

Once, we have constructed buffer range tree on the maps of Monaco, Serbia and India, we did range queries and range search statistics are as shown in table 5.7. Since, the number of I/O in range search depends on the depth of range, the statistics for range search are independent for different countries. With respect to number of I/O for range search, we have observed that higher the level of split node, more will be the number of I/Os for range search because here we access more number of leaves. For Monaco data, we did a range query in the range of [(43.72,7.386) , (43.75,7.43)] as shown in Figure 5.10 and red colored dots shows the hospitals present in given range query.

In the map of Serbia, we did a range query in [(43,19) , (44,20)] as shown in Figure 5.11 and for India in the range of [(77 , 21.2) , (78 , 22.2)] as shown in Figure 5.12 where green colored area shows the result of range query.

In our experiments, though our focus is on the implementation of buffer range search tree at low level, we did few set of experiments to check correctness and evaluate its behavior with real world dataset. This also helped us to understand that our implemented data structure

Country	Range Search Statistics		
(Millions)	# of I/O(I/O upperbound)	I/O TIme(sec)@I/O Rate(MBps)	Dirty Writes
Monaco (0.016)	480 (512)	2.381 @ 0.34	454
Serbia (3.6)	650 (2048)	53.39 @ 34.42	495
India (36)	32 (576)	0.706 @ 88.3	18

Table 5.7: OSM Range Search Statistics

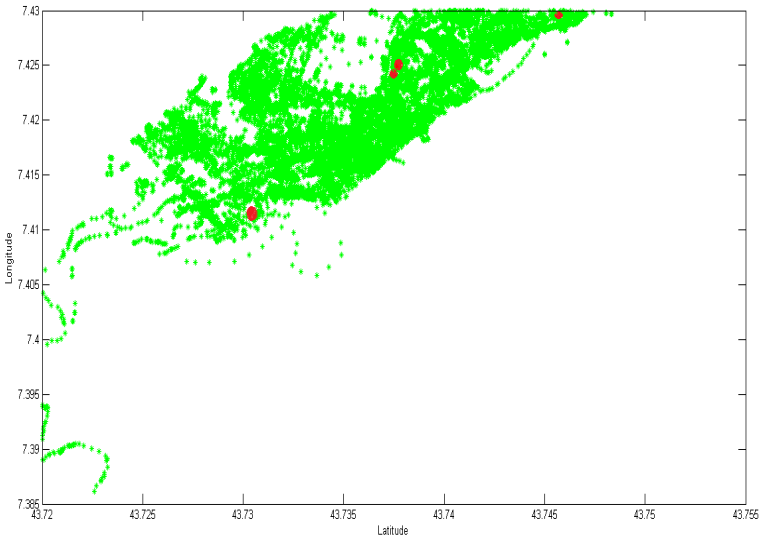

Figure 5.10: Hospitals in Range Search of Monaco(Red colored dots indicates hospitals)

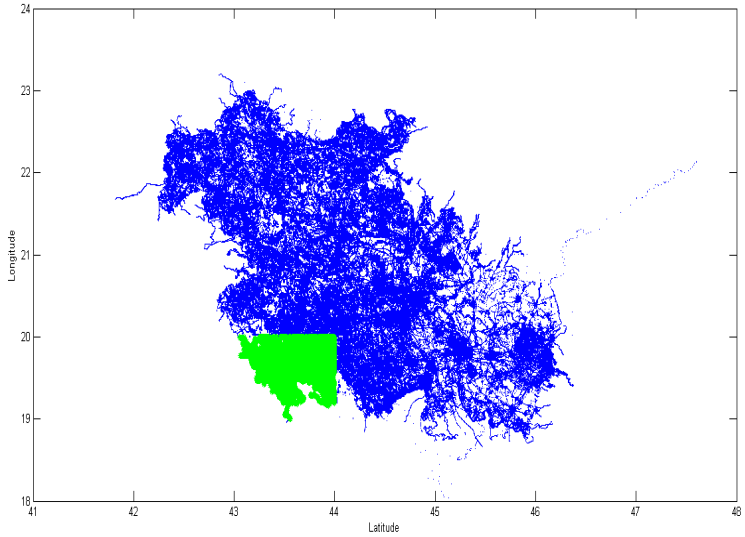

Figure 5.11: Range Search in Serbia(Green colored area is the result of range query)

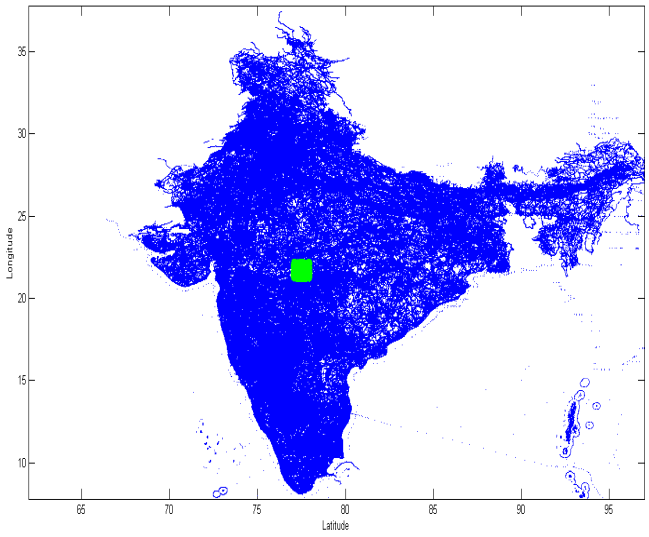

Figure 5.12: Range Search in India(Green colored area is the result of range query)

is applicable and scalable in such real world practical applications. In order to cover different ranges of number of data elements we chose data set of Monaco, Serbia and India such that the ratio in their data sizes is approximately 100.

Chapter 6

Conclusion and Future Work

We have implemented Buffer Range Tree in 2-dimension in external memory which is capable of storing out of memory data. We did our experiments with different parameters so as to evaluate the performance and correctness of the data structure. Few of the key observations in terms of number of I/O for the construction of buffer range tree is that in 2-dimension, the number of I/O shows a steep rise with an increase in number of input data elements which occurs with an increase in the level of the tree because with an increased level, number of y-associate trees also increases. We have also observed through our experiments that there is always a trade-off in between the size of block and number of I/O because larger block size can transfer more data in 1 I/O, so the choice of block size should always be such that it utilizes computer's internal memory to full of its capacity. For a search in given range, higher the level of split node, more will be the number of I/O because we need to access more number of leaves whose data is stored in external memory.

Our thesis work can have good applications in the real world area of geometrical data sets as we have shown through our experiments on open street map (or OSM). The implemented underlying buffer range tree can be used in real time map range search. We did an implementation of a basic range tree, more evaluation and experiments can be done by implementing Layered Range Trees, and Fractional Cascading technique. Many other geometrical data structures like Kd-trees [5], Quad Trees [12], etc can be implemented in external memory and evaluation of their performance, comparison with other geometrical data structures implemented in external memory will give a good insight on the overall performance and behavior of geometrical data structures in external memory.

Bibliography

[1] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I/II. Algorithmica, 12(2/3):110–169, 1994.

[2] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In 40th Symposium on Foundations of Computer Science, pages 285–298, 1999.

[3] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. ACM Computing Surveys, 33(2):209–271, 2001.

[4] U. Meyer, P. Sanders, and J. Sibeyn, editors. Algorithms for Memory Hierarchies, volume 2625 of LNCS Tutorial. Springer, 2003.

[5] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, Computational Geometry: Algorithms and Applications: 5. Orthogonal Range Searching, Springer, 3rd edition, 2008.

[6] Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: Standard Template Library for XXL Data Sets, Fakult at f ur Informatik Universit at Karlsruhe,August 9, 2005.

[7] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, Silicon Graphics Inc., Hewlett Packard Laboratories, 1994.

[8] A. Crauser and K. Mehlhorn. LEDA-SM, extending LEDA to secondary memory. In 3rd International Workshop on Algorithmic Engineering (WAE), volume 1668 of LNCS, pages 228–242, 1999.

[9] Lars Arge, Rakesh Barve, David Hutchinson, Octavian Procopiuc, Laura Toma, Darren Erik Vengroff, and Rajiv Wickeremesinghe. TPIE: User manual and reference, November 2003.

[10] Lars Arge. The Buffer Tree: A New Technique for Optimal I/O Algorithms, Copyright all rights reserved , BRICS, Department of Computer Science University of Aarhus, 1996.

[11] Jeffrey Scott Vitter. Algorithms and Data Structures for External Memory, Foundations and TrendsR in Theoretical Computer Science Vol. 2, No. 4 (2006) 305–474, Department of Computer Science, Purdue University, West Lafayette, Indiana, 47907–2107, USA, 2008.

[12] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, Computational Geometry: Algorithms and Applications: 14.Quad Trees, Springer, 3rd edition, 2008.

[13] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. Communications of the ACM, 31(9):1116 –1127, 1988.

[14] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In Proc. Annual European Symposium on Algorithms, LNCS 979, pages 295 –310, 1995.

[15] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In Proc. ACMSIAM Symp. on Discrete Algorithms, pages 139 –149, 1995.

[16] M. T. Goodrich, J.-J. Tsay, D. E. Vengrof, and J. S. Vitter. External memory computational geometry. In Proc. IEEE Symp. on Foundations of Comp. Sci., pages 714 –723, 1993.

[17] $OpenStreetMap - Wiki$