On Evaluating And Parallelizing External Memory Data Structures

Stergios Charalampakis

Master's thesis

Contents

Chapter 1

Introduction

Many modern applications need to deal with massive datasets, that usually are much larger than the size of the main memory.Additionally, many massive dataset applications involve geometric data (for example points,lines or polygons) or data that can be interpreted geometrically. Such applications often perform queries that correspond to searching in massive multidimensional geometric databases for objects that satisfy certain spatial constraints. Typical queries include reporting the objects intersecting a query region, reporting the objects containing a query point, and reporting the objects near a query point.Examples of such applications are applications in computer graphics, computer vision, database management systems, computer-aided design, solid modeling,robotics, geographic information systems (GIS), image processing, computational geometry, pattern recognition , and other areas.

Solutions to proccess and store these massive datasets efficiently were needed. To address this problem, many data structures and algorithmic techniques were implemented. However, these early data structures were either slow or too difficult to understand and implement, thus impractical. In recent years, a number of data structures and algorithmic techniques have been developed, that improved and simplified previous approaches. Despite all these efforts, industry resists to adopt many of those algorithmic techniques. The main reason for that, is that there are still some deficiencies in practice and the data structures that were developed are still impractical to implement for industrial reasons.

1.1 Memory Models

At first, algorithms were developed to run efficiently in idealized computer models such as the **Random Access Machine** or the **Pointer Machine**. These early models described a two-level computer system were the proccessor fetches instructions from the memory and executes them . Even though they are widely used, they cannot accurately model contemporary computer systems that contain a multilevel memory hierarchy. Algorithms that are implemented under these models, are thus poor performing, especially in applications that need to deal with massive data.

For reasons of economy, contemporary general-purpose computer systems usually contain a hierarchy of memory levels, each level with its own cost and performance characteristics. At the lowest level, CPU registers and caches are built with the fastest but most expensive memory. At a higher level, inexpensive but slower magnetic disks are used for external mass storage and even slower but larger capacity devices, such as tapes and optical disks are used for archival storage.

The need to model memory and disk systems more accurately led to the development of the **External Memory Model** [AV88] . In the External Memory Model we have a two-level memory hierarchy which works with block transfers. Thus, we can describe more precisely contemporary computers and modern applications where data are so massive, that cannot fit in the main memory. In such cases the Input/ Output (or I/O) communication between internal and external memory can become a major performance bottleneck. Thus, the time needed to fetch data from the external storage device usually overwhelms the CPU time. For example, loading a register takes on the order of nanosecond (10^{-9} seconds) and accessing internal memory takes tens of nanoseconds, but the latency of accessing data from a disk is several milliseconds (10^{-3} seconds), which is about one million times slower. That's why the time complexity in this model is measured by the number of I/O operations needed.

The central aspect of the external-memory model is that transfers between cache and disk involves blocks of data. Specifically, the disk is partitioned into blocks of B elements each, and the cache can store up to M/B blocks for a total size of M elements. The main deficiency of this model is that external memory algorithms depend on the M and B parameters. In practice though, we may have several storage devices in our memory hierarchy with varying parameters.

The next attempt to define the memory hierarchy of contemporary computer systems even more accurately was the **Cache-Oblivious Model**, introduced by Frigo, Leiserson, Prokop and Ramachandran [FLPR99]. The principle is to design external memory algorithms without knowing the external memory parameters B

and M. The consequence of that idea is that, if a cache-oblivious algorithm performs well between two levels of the memory hierarchy, then it will automatically work well between any two adjacent levels of the memory hierarchy. This model thus, accurately represents even the common computer systems where the external memory is constituted by several storage devices with varying parameters, in contrast with the External Memory Model where we can model only one type of external memory.

1.2 Massive Data Applications

The aforementioned models were implemented due to the need to accurately model computer systems running applications that deal with data so massive that cannot fit in the main memory. One of the most common applications is spatial data structures. Spatial data consist of points, lines, rectangles, regions, surfaces and volumes. The challenge is to create efficient indexes for spatial data structures.

A fundamental database primitive in spatial databases is range search. For example consider a cartographic database consisting of a number of maps. A typical query would be to determine all cities within 50 kilometers of a well-known landmark. Other types of spatial queries include point location, ray shooting, nearest neighboor, and intersection queries.

Simple data structures such as linked lists or binary trees were simply inadequate to efficiently index spatial databases dealing with such kind of queries. For range-searching a number of advanced internal memory structures were implemented, like the elegant **priority search tree** [McC85b], that can be used to answer 3-sided range queries in optimal query and update time using linear space. Other internal memory structures that were implemented to answer efficiently range searching queries are the quadtree, the octree, the k-d tree and others. All these data structures are optimal in two-level memory models, but they fail in practice where the majority of data resides in external storage devices.

The most fundamental external memory data structure that was implemented is the B-tree [BM72]. The B-tree corresponds to an internal memory balanced search tree. One-dimentional range queries, asking for all elements in the tree in a query interval $[q_1, q_2]$, can be answered in optimal time using linear space. There are though, a number of problems, like multidimensional range searching that cannot be handled efficiently by the B-tree. A number of external memory model advanced data structures were implemented to cope with range-searching, like hB trees, various R-trees, external range trees, weight-balanced B-tree, O-tree, etc.

A usual method to implement an external memory data structure is to partition the nodes of the corresponding internal memory data structure into blocks. For example, to externalize the binary search tree, we just need to partition the nodes of the tree into blocks, thus creating the B-tree. Just partitioning the nodes of a tree into blocks to externalize it though, is not always I/O optimal, (as for example the problem of externalizing the priority search tree) because of the high fan-out of the tree. The **External Priority Search Tree**, an elegant solution for the long-standing problem of externalizing the priority search tree was given in [Sam01], utilizing advances techniques, such as bootstrapping and node buffering.

A very popular data structure for handling spatial queries that is based on the B-tree, is the R-Tree [Gut85]. This data structure uses heuristic algorithms to optimaly partition space into rectangles that cover data figures (points, lines, polygons). Even though the R-Tree is based on heuristic methods and we don't have any performance guarantees, it is used prevalently in modern database management systems. Because of it's great practical value, much research has been done to imporove it's efficiency. This research has resulted in many R-Tree variants. The most promising one seems to be the Hilbert R-Tree, which uses the idea of space-filling curves to optimize the partitioning of the space into rectangles.

1.3 Exploiting Parallelism

Very large indexes that don't fit in main memory and thus need a large number of I/Os to fetch data for every query, can become a major performance bottleneck in database management systems. Much research has been done to improve the efficiency of these data structures and even devise new clever ways to index massive data. However very little work has been done to exploit parallelism, as a means to improving the data structures' performance.

In the past few years commodity computing hardware has become cheaper and much more powerful. This fact along with the need to constantly process vast amounts of data, has driven large organizations and companies to build robust and highly scalable parallel systems. One recent effort that has attracted much attention due to it's scalability and efficiency is Google's MapReduce model. The MapReduce model uses the ideas of functional programing to parallelize programs, in a way that is transparent and painless to the end user. The MapReduce system has been used for many problems that require to deal with massive data such as sorting, distributed grep, reverse web-link graph and others. It would be thus, very conve-

nient to somehow exploit these systems for improving data structures' efficiency.

1.4 Our Thesis

All of the well-known memory models have some deficiencies in practice. The two-level memory models (RAM, pointer machine) are adequate when the data we are dealing with fits in main memory. However, when we are dealing with massive data these models fail to take into account the high latencies incurred by the I/O operations. This problem is handled by the other two models (cache-oblivious and external-memory), but both of these models totally disregard the CPU cost of the algorithms. There are practical cases though, where the CPU cost is not negligible compared to the I/O cost. Consider for example an external memory B tree where almost all the internal nodes are kept in main memory and just a percent of the leaves of the tree are kept on disk. These cases are very common with contemporary huge main memories. When we query this tree the search cost measured in I/O cost is just 1, compared to the CPU cost which is approximately $\log N$. Additionally, there are cases where only a small part of the tree is kept in external storage devices. In those cases a query may incur a small number of I/Os, or none at all. In all of the above cases we must take into account both the CPU cost and the I/O cost when we design an efficient algorithm. It is necessary therefore, to reexamine the CPU efficiency of many I/O optimal external memory algorithms.

Additionaly, the new technological trends cannot be accurately represented by these old models. With the introduction of multicore processors, the CPU speed has dramatically increased and the bottleneck seems to be the I/O between main memory and CPU. To minimize this I/O latency, a hierarchy of cache memories are built near the processor. The memory levels that are closer to the processor are faster, but of limited capacity. An important factor thus, in designing algorithms is the concept of locality, which is not taken into account by the old cost models. Another important technological aspect that is disregarded by the old cost models, is the introduction of cheap and high capacity flash disks. These flash disks still have a relatively large I/O cost, but compared to conventional magnetic disks they have zero seek time, making them especially useful for point queries.

In our thesis we implement well-known I/O optimal external memory data structures such as the B-tree, the R-Tree and variants and the External Memory Priority Search Tree (EPST), and conduct a series of experiments on them. These experiments are conducted under various datasets and different hardware schemes. We

performe the same experiments in the corresponding internal memory data structures (priority search tree, R-Tree) and we analyze the results.

The results taken from the external memory data structures and their corresponding internal memory data structures are compared. We examine the cases when it is optimal to use the external memory data structures and the cases when it is optimal to use the corresponding internal memory data structures. We prove that the CPU cost can be comparable and even higher than the I/O cost in some cases and vice versa. We also note the effect of varying the block size in the overall performance of external memory data structures.

In the second part we exploit parallelism to improve the bulk-loading performance of a packed Hilbert R-Tree. An algorithm is provided to adapt the process of bulk-loading a Hilbert R-Tree in the MapReduce model. The Hadoop open source implementation of the MapReduce model is used to experiment with this algorithm. The hardware we used was a five node computing cluster. Finally we conduct experiments that test the scalability and the overall efficiency of the system and the data structures.

Out results can be summarized as follows:

- We show that data structures with proven performance guarantees such as the priority, the EPST and the B Tree perform better, especially for certain datasets and queries, than the data structures that rely on heuristic algorithms, such as the R Tree and variants.

- We experimentally prove that heuristic data structures perform well only under certain datasets and queries which is very limiting in practise.

- We prove that both the CPU cost and the Disk I/O cost, can be of great importance for certain types of datasets, despite the assumptions made by the cost models.

- We also experimentally prove that the increase of the block size affects the data structures' performance negatively or positively, depending on the dataset size.

- We implement an algorithm that expresses the process of bulk-loading Hilbert R-Trees with the MapReduce model.

- We use the Hadoop system to experimentally evaluate the above algorithm in a cluster of five nodes.

- We prove that our algorithm is scalable and efficient.

- The experimental results, indicate that the trees produced by this parallel procedure, have almost the same query performance with the equivalent trees which are bulk-loaded in serial.

Chapter 2

Overview Of Memory Models

Data structures are always specified within a particular *memory model*. Space and time costs, that usually determine the quality of an algorithm, are therefore relative to the model under which an algorithm is analysed. As a consequence of that, some models are more suitable than others depending on the application.

The early memory models, like the *random acess memory*, or the *pointer machine* model, described an oversimplified two-level computer architecture. These models produce simple and efficient algorithms for a broad range of applications. Algorithms though, for applications that need to deal with massive data cannot be efficiently implemented with these simplistic models.

More sofisticated models were thus implemented, like the *external memory model* or the *cache-oblivious model*. These models managed to describe more accurately contemporary computer systems, by incorporating the notion of memory hierarchies. Algorithms that are implemented under these models, for applications that deal with massive data, are far more efficient than their internal memory counterparts. A major drawback however, of those sofisticated models is that algorithms are more complicated and difficult to implement.

2.1 The RAM Model

At first, algorithms were developed to run efficiently in a hypothetical computer called the **Random Access Machine**. Computations on this machine can only be performed on objects in the internal memory. The Random Access Machine has a proccessor that fetches instructions from the memory and executes them. The

Random Access Machine can perform "simple" operations such as addition, multiplication and substraction, in a single step. Instructions are executed one after another with no concurrent operations.

This two-level model of computation is called the **Random Access Machine Model**.Under the RAM model, we measure the run time of an algorithm by counting up the number of steps it takes on a given problem instance. By assuming that our RAM executes a given number of steps per second, the operation count converts to the actual run time easily. A great number of algorithms are still developed under this model.

The RAM model is simple and the algorithms constructed under this model are both easy to understand and implement. However, the RAM model of computation is obviously unrealistic and leads to the development of poor performing algorithms for modern applications. The main reason is that in the RAM model we do not attempt to model the memory hierarchy that is common in contemporary computers. Thus, the only practical use of the RAM model is in applications where data (or at least the majority of data) resides in the main memory. Whenever we have to deal with massive datasets, where the majority of data resides in the external memory storage, the RAM model fails to produce efficient algorithms.

2.2 The Binary Search Tree

Binary Search Trees and Search Trees in general are the most widely used data structures that are analyzed under the RAM model. Search trees are data structures that support many dynamic-set operations, including *Search, Minimum, Maximum, Predecessor, Successor, Insert* and *Delete*

The binary search tree is a linked data structure in which each node is an object (Figure 2.1). In addition to a key field and satellite data each node contains fields *left,right* and p that point to the node corresponding to its left child, its right child and its parent respectively. If a child or the parent is missing, the appropriate field contains the value NIL. The root node is the only node in the tree whose parent field is NIL.

The keys in a binary search tree are always stored in such a way as to satisfy the **binary-search-tree property:**

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $key[y] \leq key[x]$. If y is a node in the right subtree of x, then $key[x] \leq key[y]$. The most fundamental operations of a binary search tree are described below.

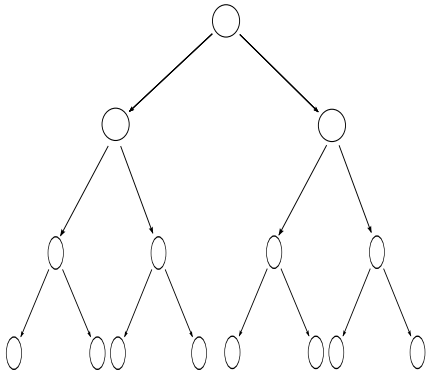


Figure 2.1: A typical binary search tree.

Searching

Searching a binary search tree for a specific key stored in the tree, can be implemented as a recursive or an iterative process. In the recursive case we begin by comparing the key of the root with the given value k. If the key of the root is equal to k or it is equal to NIL (the tree is empty), then we return with a pointer to root. If the key of the root is less than the given value then we traverse the right subtree of the root, otherwise we traverse the left subtree of the root. This procedure is continued recursively and it is depicted in the algorithm below.

Algorithm 1: SEARCH(x,k)

if $x = NIL$ or $k = key[x]$ **then**
  return x
end if
if $k \leq key[x]$ **then**
  return SEARCH($left[x]$,k)
else
  return SEARCH($right[x]$,k)
end if

Insertion

The first part of the insertion is to search for the approppriate node for the new value u to be inserted, so as the binary trees' invariants are not violated. We begin

by searching the root and if the root is not equal to u, then we proceed iteratively in the left and the right subtree of the root, exactly as we did in operation search. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value. Algorithm 2 depicts the procedure of inserting a node z for which $key[z] = u$, $left[z] = NIL$ and $right[z] = NIL$, in a binary search tree T.

Algorithm 2: TREE-INSERT(T,z)

```
y ← NIL
x ← root[T]
while x ≠ NIL do
   y ← x
   if key[z] < key[x] then
      x ← left[x]
   else
      x ← right[x]
   end if
end while
p[z] ← y
if y = NIL then
   root[T] ← z
else if key[z] < key[y] then
   left[y] ← z
else
   right[y] ← z
end if
```

Deletion

If we want to delete a given node z from a binary search tree we have to consider separately several cases. If z has no children, we modify its parent $p[z]$ to replace z with NIL as its child. If the node has only a single child, we "splice out" z by making a new link between its child and its parent. Finaly, if the node has two children, we splice out z's successor y, which has no left child and replace z's key and satellite data with y's key and satellite data. The procedure of deleting a node z from a binary search tree T is depicted in algorithm 3.

Algorithm 3: TREE-DELETE(T,z)

```
if left[z] = NIL or right[z] = NIL then
   y ← z
else
   y ← TREE − SUCCESSOR(z)
end if
if left[y] ≠ NIL then
   x ← left[y]
else
   x ← right[y]
end if
if x ≠ NIL then
   p[x] ← p[y]
end if
if p[y] = NIL then
   root[T] ← x
else if y = left[p[y]] then
   left[p[y]] ← x
else
   right[p[y]] ← x
end if
if y ≠ z then
   key[z] ← key[y]
end if
return y
```

2.2.1 Red-Black Trees

As we saw the main drawback of the binary search trees is that after a number of random insertions they may become unbalanced, resulting in poor performance due to the increase of the height h of the tree. To address this problem many solutions based on the binary search tree were implemented, that utilized techniques trying to avoid the loss of balance. One of the most popular data structures of this kind is the **Red-Black Tree**, invented in 1972 by Rudolf Bayer [Bay72]. The red-black tree is more complex than the simple binary search tree but it guarantees logarithmic time complexity in the worst case.

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either *RED* or *BLACK*. By constraining the way nodes can be colored on any path from root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately **balanced**.

Each node of the tree now contains the fields *color,key,left,right* and p. If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value *NIL*. In red-black trees, the leaf nodes are not relevant and do not contain data. To save memory, sometimes a single sentinel node performs the role of all leaf nodes. All references from internal nodes to leaf nodes instead point to the sentinel node.

Red-Black trees must confine to the following properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

We call the number of black nodes on any path from, but not including, a node x down to a leaf the **black-height** of the node, denoted $bh(x)$. By the last property, the notion of the black-height is well defined, since all descending paths from the node have the same number of black nodes. We define the black-height of a red-black tree to be the black-height of its root.

These constraints enforce a critical property of red-black trees: that the longest

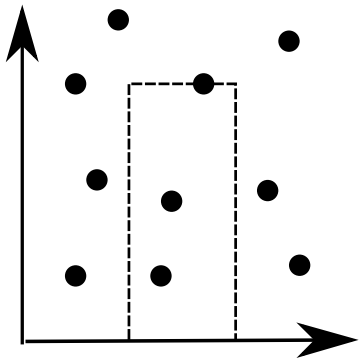


Figure 2.2: Three-sided query

path from the root to a leaf is no more than twice as long as the shortest path from the root to a leaf in that tree. The result is that the tree's height is no more than $2 \log n + 1$. Since operations such as inserting, deleting, and finding values requires worst-case time proportional to the height of the tree, this theoretical upper bound on the height allows red-black trees to be efficient in the worst-case, unlike ordinary binary search trees.

To see why these properties guarantee this, it suffices to note that no path can have two red nodes in a row, due to property 4. The shortest possible path has all black nodes, and the longest possible path alternates between red and black nodes. Since all maximal paths have the same number of black nodes, by property 5, this shows that no path is more than twice as long as any other path.

2.2.2 Priority Search Tree

Binary search trees and variants are optimal when dealing with simple one-dimensional queries. However, when we need to perform multi-dimensional queries like finding all items in a confined space, then more advanced data structures like interval trees, range trees, segment trees etc. are more suitable. The most efficient data structure for solving three-sided queries (report all points that lie in a region of the form $[x_1, x_2] \times [-\infty, y]$ Figure 2.2) is the elegant **priority search tree** by R. McCreight [McC85a].

The priority search tree is the combination of a binary search tree and a heap. A heap is a binary tree defined as follows. The root of the tree stores the point

with minimum y-value. The remainder of the set is partitioned into two subsets of almost equal size, and these subsets are stored recursively in the same way. We can do a query with $(-\infty : q_y]$ by walking down the tree. When we visit a node we check if the y-coordinate of the point stored at the node lies in $(-\infty : q_y]$. If it does, we report the point and continue the search in both subtrees; otherwise, we abort the search in this part of the tree.

The priority search tree is consisted from a base binary search tree on x-coordinates and a heap on y coordinates. A formal definition of a priority search tree for a set P of points is as follows. We assume that all points have distinct coordinates whith no loss of generality.

- If $P = \emptyset$ then the priority search tree is an empty leaf.
- Otherwise let P_{min} be the point in the set P with the smallest y-coordinate. Let x_{mid} be the median of the x-coordinates of the remaining points. Let

$$P_{below} = \{p \in P\{p_{min}\} : p_x < x_{mid}\},$$
$$P_{above} = \{p \in P\{p_{min}\} : p_x > x_{mid}\},$$

  The priority search tree consists of a root node v where the point $p(v) = p_{min}$ and the value $x(v) = x_{mid}$ are stored. Furthermore,

- the left subtree of v is a priority search tree for the set P_{below},
- the right subtree of v is a priority search tree for the set P_{above}.

The running time for building a priority search tree is optimal $O(nlogn)$. Priority search trees can be even built in linear time, if the points are already stored on x-coordinate. The idea is to construct the tree bottom-up instead of top-down, in the same way heaps are normally constructed. The priority search tree requires optimal $O(n)$ space for storing n points, since there is one node for each point

Searching A Priority Search Tree

Performing a three-sided query on a priority search tree involves reporting all points that lie in the region $[x_1, x_2] \times [-\infty, y]$. The following is the algorithm for performing this query:

- If the tree is $NULL$ we return without reporting any points.

- Let R be the root of the tree, R_x be its X-coordinate, R_y be its Y-coordinate and $X(R)$ be the value of the axis separating the X-ranges of R's child subtrees.
- Compare R_y to y. If $R_y > y$ then return without reporting any points (all the other nodes of the tree will have even larger Y-coordinate).
- Else if $x_1 \leq R_x \leq x_2$ then report the root point.
- If $x_1 < X(R)$, the X-range of the left subtree must overlap with the X-range of the query. Recursively search the left subtree of R.
- If $X(R) < x_2$, the X-range of the right subtree must overlap with the X-range of the query. Recursively search the left subtree of R.

The complexity of a query of size k in a priority search tree of size $O(n)$ is optimal $O(k + logn)$.

Dynamic Operations

Insertion and deletion in a priority search tree are also optimal with an overall cost of $O(logn)$ in the worst case. The procedure for inserting a point P in a priority search tree with root R is described in the algorithm below.

- Compare P_y with R_y.
- If it is bigger recursively insert P in the subtree on path to P_x
- If it is smaller replace root with P and recursively insert root in the proper subtree.

2.3 The Pointer Machine Model

Another popular early model for analyzing algorithms, similar to the RAM model is the **Pointer Machine Model**. Pointer Machines correspong roughly to high-level programming languages without arrays. There are many types of pointer machine models but the most popular is the Storage Modification Machine (SMM) introduced by Schönhage in 1970 [Sch80].

The pointer machine model resembles the RAM model in having a stored program and a similar flow of control. However, instead of operating on registers in memory it operates on a single storage structure, called a Δ-*structure*, where Δ is a finite alphabet of at least two symbols. Typical is the binary alphabet $0, 1$. A Δ structure S is a finite directed graph in which each node has $k = \#\Delta$ outgoing edges, which are labeled by the k elements of Δ. A path along nodes and edges of the structure, represents every word (string of symbols e.g. 101101) "the machine" can accept.

Similar to the RAM model, the program of the pointer machine consists of a flow of control instructions (**goto**, *Accept, Halt,...*), transput instructions (*Read* and *Print* where a *Read* will input a single bit and act like a conditional jump depending on the value of the bit read), and instructions which operate on memory, in this case a Δ structure S. There exists three types of instructions of the latter type:

1. **new** w: creates a new node which will be located at the end of the path traced by w. All outgoing edges of the new node will be directed to the former node $p^\star(w)$.

2. **set** w **to** u: redirects the last pointer on the path labeled by w to the former node $p^\star(u)$.

3. **if** $u = w$ (**if** $u \neq w$) **then** ...: the conditional instruction. Here it is tested whether the nodes $p^\star(w)$ and $p^\star(u)$ coincide or not.

The pointer machine model is rather more complicated and difficult to understand than the RAM model. Aditionally, it is equally inefficient with the RAM model when we have applications that deal with massive data, because it doesn't model the memory hierarchy of a typical contemporary computer system.

2.4 Memory Hierarchies

The memory of contemporary computer systems is structured under a hierarchy of levels that are successively bigger in size and cheaper, but significantly slower. At the top level of the hierarchy we have the inexpensive but low-speed external storage devices, with access times that are many orders slower than the access time of high-speed memory. In many applications therefore, external storage device access time is the speed bottleneck. That's why efficient algorithms must focus on minimizing

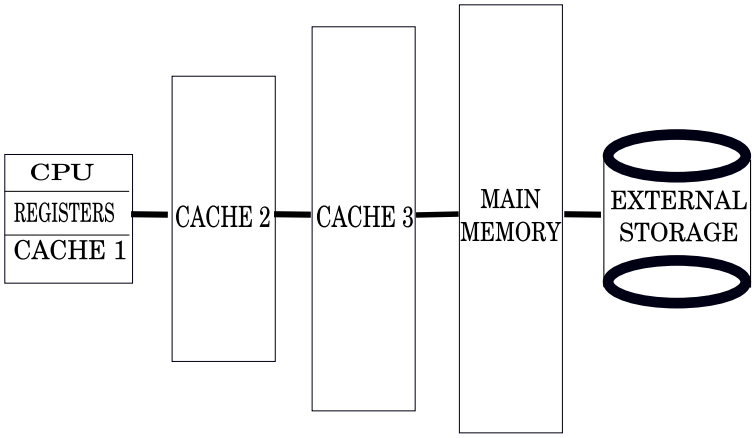


Figure 2.3: Memory hierarchy of a typical contemporary computer.

as much as possible the I/O accesses.

The design of increasingly faster Central Processing Units during the latest years, was not accompanied by a similar boost in speed for computer memory. To deal with the problem of speed difference between CPUs and memories, memory hierarchies were introduced (Figure 2.3). According to this approach, small but fast memory is kept at the first level close to the CPU. At the next levels we have memory devices, that are larger in size and cheaper, but significantly slower. Memory in level i, acts as a buffer for the memory at the next level $i + 1$.

The adoption of the memory hierarchy model is based on the principle of the locality of reference. This principle states that, when a program references a particular memory address, it is very likely that it will reference it again many times in a short period of time (*time locality*). Additionaly it is higly possible that a program referencing a particular memory address, will reference neighbouring memory addresses as well (*space locality*).

When a program requests a data item from the memory, CPU examines if the data item resides in the first-level memory, which is closer to the processor. When the data item cannot be found in the first-level memory, a cache miss is marked and the processor tries to fetch the requested data item from the next memory level. Caching and prefetcing heuristics have been developed to reduce the number of occurences of a cache miss.

Caching and prefetching methods though, are typically designed to be general-purpose and thus cannot be expected to take full advantage of the locality present

in every computation. Additionaly some computations themselves are inherently nonlocal and are doomed to perform large amounts of cache misses. All these facts are totally disregarded in the oversimplistic RAM model (§2.1) and the pointer machine model (§2.3) , resulting in poor performing algorithms for applications that deal with massive data. That's why even though most data structure research in the algorithms community has focused on worst-case efficient internal memory data structures, several authors have considered more accurate and complex multi-level memory models than the two-level model.

2.5 The External Memory Model

Aggarwal and Vitter at 1988 [AV88], introduced the **external memory model** or the **I/O model**. This level describes a computer system with a two-level memory. The first level memory is the fast but limited in space main memory, with a capacity of size M. The next level memory is the external storage device, which has a huge capacity, but is extremely slow compared to the main memory. Transfers between these two, are done via blocks of size B (Figure 2.4). The main memory thus, can hold at most M/B blocks of the external memory. Whenever we read or write a block from (or into) the external memory, an I/O operation is charged. We therefore, use the term I/O to designate the communication between the internal memory and the disks.

The primary feature of disks we want to model in the external memory model, is their extremely long access time relatively to that of the internal memory. In order to amortize the access time over a large amount of data, typical disks read or write large blocks of contiguous data at once. Computation can only be performed on objects in the internal memory. The measures of performance in this model are the numbers of I/Os used to solve a problem, as well as the amount of space (disk blocks) used. That's why the primary concern when we design algorithms under the external memory model is to minimize the number of I/Os, ignoring the CPU time needed for the computations of the algorithm.

2.5.1 Fundamental External Memory Data Structures

In this section we will ilustrate some of the most fundamental external memory data structures and the techniques and ideas used in their development. Several of the

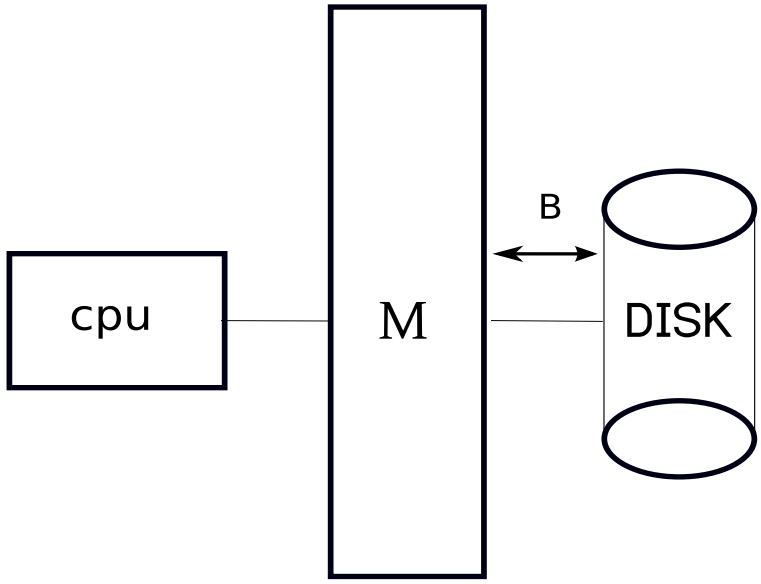


Figure 2.4: In the external memory model transfers between disk and internal memory are done via blocks of data, containing B contiguous elements.

worst-case efficient data structures we consider are simple enough to be of practical interest. Still, there is the need ,mainly in the database industry, to develop even simpler data structures for practical purposes.

The B-tree

Tree-based data srtuctures arise naturally in applications, where data can be updated and queries must be processed immediately.The B tree is propably the most famous external memory data strucure [BM72, Com79]. The B-tree corresponds to an internal memory balanced search tree.Many database systems use B-trees, or variants of B-trees, to store information.

B-trees differ from internal memory binary trees, in that B-tree nodes may have many children, from a handful to thousand. That is, the branching factor of a B-tree can be quite large, although it is usually determined by the hardware characteristics of the disk unit used.

The degree of each node in the B-tree (with the exception of the root) is required to be $\Theta(B)$, which guarantees that the height of a B-tree storing N items

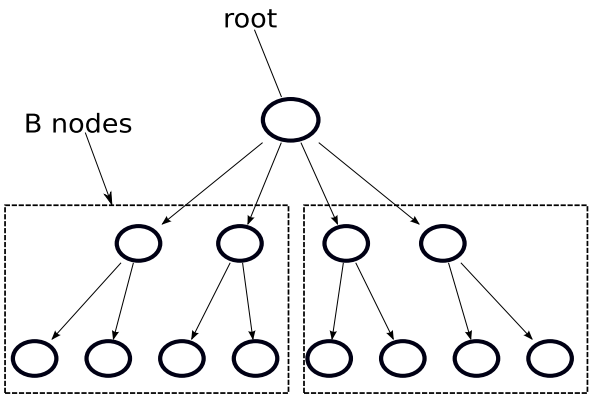


Figure 2.5: Externalization of an internal memory search tree. B nodes are partitioned together, to form a node of the B tree.

is $\Theta(log_B N)$. The procedure of turning an internal memory data structure into its equivalent external memory, is called externalization. To externalize the balanced tree and thus create the B tree, we use a standard and simple technique, which partitions the nodes of the tree into blocks of size B (Figure 2.5). The parameter B represents a block of data and is chosen to be equal with the hardware storage device block parameter to optimize the B-tree in terms of I/O disk accesses.

B-trees support dynamic dictionary operations and one-dimensional range search. B-tree uses linear space ($O(N/B)$ disk blocks) and supports insertions and deletions in $O(log_B N)$ I/Os, which is optimal. One-dimensional range queries, asking for all elements in the tree in a query interval $[q_1, q_2]$, can be answered in $O(log_B N + T/B)$ I/Os, where T is the number of reported elements. The space, update, and query bounds obtained by the B-tree are the bounds we would like to obtain in general for more complicated problems. The bounds are significantly better than the bounds we would obtain if we just used an internal memory data structure. Note that the query bound consists of an $O(log_B N)$ search-term corresponding to the familiar $O(log_2 N)$ internal memory search-term and an $O(T/B)$ reporting term accounting for the $O(T/B)$ I/Os needed to report T elements.

The B tree is formaly defined in [CLR90] as follows:

A **B-tree** T is a rooted tree (whose root is $root[T]$) having the following properties:

1. Every node x has the following fields:

(a) $n[x]$, the number of keys currently stored in node x

(b) the $n[x]$ keys themselves, stored in nondecreasing order, so that $key_1[x] \leq key_2[x] \leq \ldots \leq key_n[x]$,

(c) $leaf[x]$, a boolean value that is TRUE if x is a leaf and $FALSE$ if x is an internal node.

2. Each internal node x also contains $n[x+1]$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.

3. The keys $key_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$, then

   $k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq key_{n[x]}[x] \leq key_{n[x]+1}$.

4. All leaves have the same depth, which is the tree's height h.

5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer $t \geq 2$ called the **minimum degree** of the B-tree:

   (a) Every node other than the root must have at least $t-1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

   (b) Every node can contain at most $2t-1$ keys. Therefore, an internal node can have at most $2t$ children. We say that a node is full if it contains exactly $2t-1$ keys.

When a node overflows during an insertion, it splits into two half-full nodes and if the splitting causes the parent node to overflow, the parent node splits and so on. Splitting can thus propagate up to the root, which is how the tree grows in height. Deletions are handled in a symmetric way by merging nodes. During a deletion the tree's height may decrease.

The B+ Tree

An important variant of the B tree is the B+ tree [BM72]. The major difference of the B+ tree is that all the records are stored in the leaves of the tree. The internal

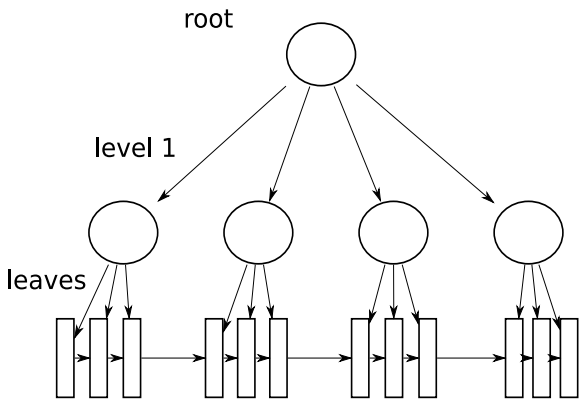


Figure 2.6: A typical three-level B+ tree. Leaves are linked together, to optimize range search.

nodes of the tree contain only keys and pointers and thus can have a higher branching factor. The leaves are linked together (usually in a linked list), in symmetric order to facilitate range queries and sequential access (Figure 2.6).This does not substantially increase space consumption or maintenance on the tree.

The fact that in the internal nodes of a B+ tree we need to store only keys and pointers, allows for compression on these nodes. For example we can compress the pointers stored in the internal nodes, with the following simple technique: if we suppose that some consecutive blocks $i, i+1 \ldots i+k$ are stored contiguously, then it will suffice to store only a pointer to the first block and the count of consecutive blocks.

The ReiserFS filesystem (for Unix and Linux), XFS filesystem (for IRIX and Linux), JFS2 filesystem (for AIX, OS/2 and Linux), and NTFS all use this type of tree for block indexing. B+ trees are also the index structure of choice in most database implementations.

The Multiversion B-tree

In some database applications we need to be able to update the current database while querying both the current and earlier versions of the database (data structure). One simple but very inefficient way of supporting this functionality is to copy the whole data structure every time an update is performed. Another and much more efficient way is throuh the (partially) persistent, or multiversion method

[BGO+96].

In this method, instead of making copies of the structure every time an update is performed, we maintain one structure at all times, but for each element we keep track of the time interval at which it is really present in the structure. This can be done by augmenting each data element with a time interval. Every time an update is performed to the data, a new version is created. We say that an element is alive in it's existence interval. In this structure we can search any element at any version, but we can update only the most recent version, that's why we call it a partially persistent structure. In a fully persistent data structure instead, we can make updates (inserts or deletes) in the history as well.

The Multiversion B-tree supports all the operations that the B-tree supports, such as insert, delete, query. With the Multiversion B-tree we can also perform a range query which returns all records whose keys lies between the given lowkey and the given highkey in the given version. If we think of each data element as an interval, where the interval's endpoints are the time stamps of the element, then we can also perform a time-stabbing query. This type of query, which is very useful in computational geometry, returns all the data elements that are alive in a given version i (Figure 2.7).

The Multiversion B-tree is asymptotically optimal concerning the query time, achieving a bound of $O(log_B N + T)$ I/Os , where T is the number of the reported elements. It is also optimal concerning the space needed, achieving a spatial bound of $O(N/B)$. To achieve this kind of linear space requirement, certain constraints are imposed on the number of elements that a node must contain.

More specifically, for each version i and each block A except the roots of the versions, we require that the number of entries of version i in block A, is either zero or at least d, where $b = kd$ for block capacity b and some constant k; this is called the *weak version condition*. After each structural change certain invariants must also be maintained, which ensure the linear space bound of the multiversion B tree. A structural change is triggered in two ways. A block overflow occurs as the result of an insertion of an entry into a block that already contains b entries. A block underflow, can never occure because entries are never removed from blocks. However, the weak version condition may be violated in a non-root block as a result of a deletion. This situation occurs if an entry is deleted in a block with exactly d live entries and is called a *weak version underflow*.

After a block overflow occurs, the block is copied and all but the live entries are removed. This operation is called a *version split*. In general, a copy produced by this version split may be an almost full block. In that case, a few subsquent insertions would again trigger a version split, resulting in a space cost of $\Theta(1)$ block

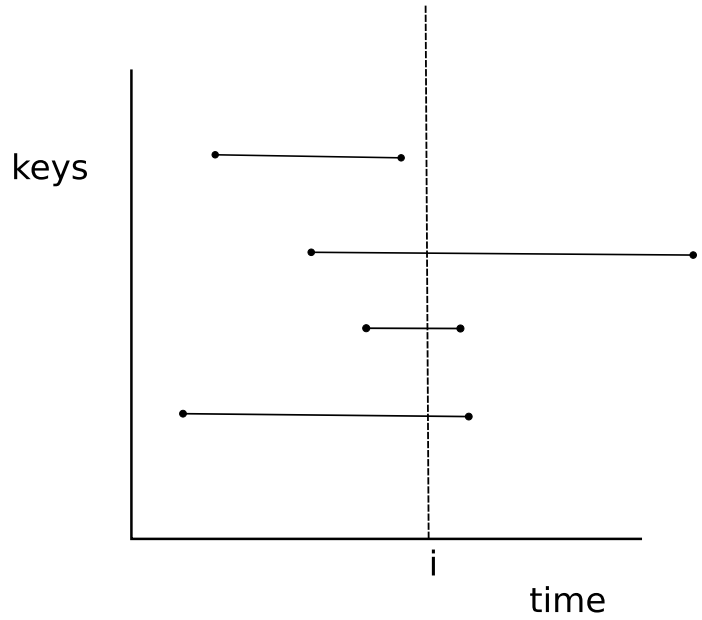


Figure 2.7: Time-stabbing query at time i.

per insertion. To avoide this and reduce the space cost of the MVBT, the following invariant is imposed: After a version split, at list $\epsilon d + 1$ insert or delete operations are necessary to arrive at the next block overflow or version underflow in that block, where ϵ is a constant chosen to reflect the fraction of the data entries that are guaranteed to be in a new node in the underlying data structure (for the B-tree $\epsilon = 0.5$). As a consequence, the number of current entries after a version split must be in the range from $(1+\epsilon)d$) to $(k-\epsilon)d$. This condition is called the *strong version condition*. If a version split leads to less than $(1 + \epsilon)d$ entries, a *strong version underflow* is marked and a merge is attempted with a copy of a sibling block containing only its current entries. Similarly, if a version split leads to more than $(k - \epsilon)d$ entries in a block, a *strong version overflow* is marked and a key split is performed.

R-Tree

The need to index multidimensional data led to the development of new advanced data structures that stem from the B tree. One of the most widely used data strcutures for spatial databases indexing multidimensional information is the **R-Tree**

[Gut85] and its variants. A common operation on spatial data handled by the R-Tree is a search for all objects in an area, for example to find all countries that have land within 20 miles of a particular point. The R-Tree can be used for efficiently storing and retrieving a variety of geometric objects, such as points, segments, polygons and polyhedra, using linear disk space.

The R-Tree partitions the space whith hierarchically nested, and possibly overlapping, minimum bounding rectangles of the spatial objects that are represented (MBR, otherwise known as bounding boxes, i.e. "rectangle", what the "R" in R-tree stands for). The internal nodes contain the minimum bounding rectangles of rectangles below each child.

The basic rules for the formation of an R-tree are similar to those for a B-tree. All leaf nodes appear at the same level. Each entry in a leaf node is a 2-tuple of the form (R, O) such that R is the smallest rectangle that spatially contains data object O. Each entry in a non-leaf node is a 2-tuple of the form (R, P) such that R is the smallest rectangle that spatially contains the rectangles in the child node pointed at by P. An R-Tree of order (m, M) means that each node in the tree, with the exception of the root, contains between $m \leq \lceil M/2 \rceil$ and M entries. The root node has at least two entries unless it is a leaf node.

Searching An R-Tree

Searching with a query rectangle and reporting the R-Tree's rectangles that intersect it, is done in a similar manner with the B Tree. The search starts from the root node of the tree and proceeds to the children determining if every rectangle in a node, overlaps the search rectangle or not. For all the internal nodes whose entries overlap the search rectangle, recursively call the search procedure for the subtree rooted at these nodes. For the leaf nodes a search is performed in their entries to check if they overlap the search rectangle and if they do, they are reported.

The only problem with this search procedure, is that a large number of nodes have to be examined since a rectangle may be contained in the covering rectangles of many nodes while its corresponding record is contained only in one of the leaf nodes. Due to this deficiency the search complexity of the R-Tree can be even linear in the worst case. The efficiency of the R-Tree relies on the assumption that for the most kinds of data the update algorithm will maintain the tree in a form that allows the search algonthm to elimimate irrelevant regions of the indexed space, and examime only data near the search area. Variants of the R-Tree have been implemented to cope with this deficiency, such as the *packed R-Tree* by Royssopoulos and Leifker

[RL85].

Dynamic Operations In An R-Tree

The algorithm for inserting an object (i.e., a record corresponding to its enclosing rectangle) in an R-Tree is analogous to that used for B-trees. New rectangles are added to leaf nodes. The appropriate leaf node is determined by traversing the R-tree starting at its root and at each step choosing the subtree whose corresponding rectangle would have to be enlarged the least. Once the leaf node has been determined a check is made to see if insertion of the rectangle will cause the node to overflow. If yes, the node must be split and the records of the node must be distributed in the two new nodes. Splits are propagated up the tree.

Deletion of a node, say R, from an R-tree proceeds by locating the leaf node, say L containing R and removing R from L. Next, adjust the covering rectangles on the path from L to the root of the tree while removing all nodes in which underflow occurs and adding them to the set U. Once the root node is reached, if it has just one son, the son becomes the new root. The nodes at which underflow occured (i.e., members of U) are inserted at the root. Elements of U that correspond to leaf nodes correspond in the placement of their constituent rectangles in the leaf nodes, while other nodes are placed at a level so that their leaf nodes are at the same level as those of the whole tree.

The Hilbert R-Tree

Even though the R-Tree is widely adopted by the database industry and has proven to perform well in practise, it is still a heuristic method with no well-defined quality guarantees. In particular, when multiple dynamic operations are performed, the space utilization of the R-Tree severely deteriorates, so as the time to answer queries.To address these deficiencies several variants of the R-Tree have been proposed, like the R^+-tree, the R^*-tree etc. One very popular variant is the Hilbert R-tree [KF94].

The Hilbert R-tree is based on space filling curves (or fractals), and specifically, the Hilbert curve to achieve better space utilization and thus better overall perfomance. The Hilbert curve is used to impose a linear ordering on the data rectangles and thus achieving a better clustering of the rectangles in the nodes of the tree.

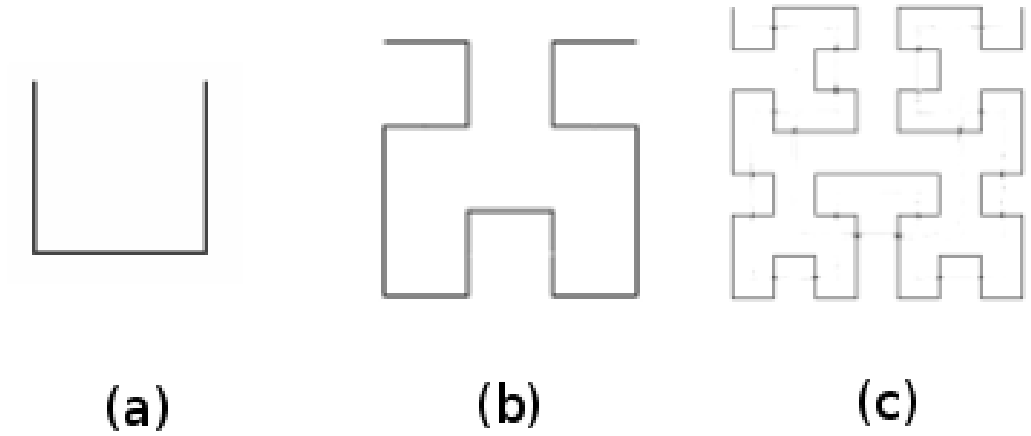


Figure 2.8: Hillbert curves of order 1,2 and 3

Space-filling curves

Space-filling curves are used in database context whenever we want to map from multiple dimensions to one dimension in a way that preserves locality. A space filling curve visits all points in a k-dimensional grid and never visits itself. Besides the Hilbert curve, we have the Z-order (or Peano curve), the Gray-code curve etc. It has been proven experimentally, that the Hillbert curve achieves the best clustering. The Hilbert curve is used extensively in a wide variety of applications.

To construct the Hilbert curve we begin with the basic Hilbert curve on a $2X2$ grid denoted by H_1, which is shown in figure 2.8(a). To construct the second order Hilbert curve we replicate the first order curve in four quadrants. When replicating the lower left quadrant is rotated clockwise 90^o, the lower right quadrant is rotated antl-clockwlse 90^o, and the sense (or direction of traversal) of both lower quadrants is reversed. The two upper quadrants have no rotation and no change of sense Thus we obtain figure 2.8(b). Remembering that all rotation and sense computations are relative to previously obtained rotation and sense in a particular quadrant, a repetition of this step gives rise to figure 2.8 (c). We can obtain higher order Hilbert curves by repeating this process. It is straightforward to obtain the linear coordinate along the curve for any given X and Y coordinate value in the 2-D grid.

Packed Hilbert R-Trees

Static data appear in several applications. For example, in cartographic databases, insertions and deletions are rare; the same is true for databases that are published on CD-ROMs; databases with spatio-temporal metereological and environmental data are seldom modified too. The volume of data in these databases is expected to be enormous, in which case it is crucial to minimize the space overhead of the index.

Bulk-loading a tree with a static dataset is a well-known and very important technique to create indexes, that has many advantages compared to the method using individual insertions. First of all, it is much easier and faster to create the index via bulk-loading methods. More importantly, the space utilization is much better when we use bulk-loading techniques to create indexes(usually 100%), improving the overall efficiency by minimizing the number of nodes visited when we query the index. Finally, when we bulk-load R-tree like structures, we can minimize the overlap between the nodes, that severely degrades the performance of the tree.

Kamel and Faloutsos [KF93] developed algorithms for bulk-loading hilbert R-trees with static spatial datasets. Their experimental results shows that the packed Hilbert R-tree performs better in practise than a tree constructed using individual insertions.

2.5.2 The External Priority Search Tree

Due to its great practical importance the priority search tree has been the target of many externalization attempts [IKO87], [BG90], [RS94]. All off these solutions have either suboptimal query time, or they use non-linear space. Samoladas and independently Arge and Vitter [ASV99], [Sam01], solved this long-standing problem optimally in both space $O(N/B)$ and query time $O(\log_B N + T/B)$ I/Os, with optimal worst-case update cost $O(\log_B N)$ I/Os.

The **External Priority Search (EPS) Tree** is applied to the 3-sided query problem for 2-d points as it's internal memory counterpart, but ,unlike the simple priority tree, it is optimal concerning the I/O access. The natural idea for constructing an EPS tree is to partition the blocks of the internal memory priority search tree into "supernodes" of fan-out $\Theta(B)$. However, this solution is not optimal concerning the I/O search cost, since one has to pay non-optimal cost $(O(\log_B N + T))$ for searching and reporting T points. Better results can be achieved if a technique callled bootstrapping is used. The EPS tree uses this technique by employing a B+

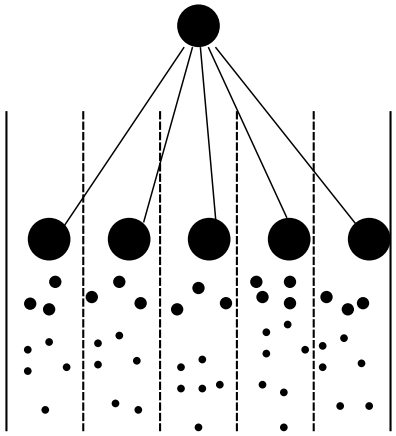


Figure 2.9: An internal node of the EPS tree. The Y-sets of the children of the node indicated by the bold points are the points within the x-range of the children, which have the highest y-coordinates.

Tree (or a Weight Balanced B Tree) as the underlying base tree and associating each node of the base tree with a small substructure called a child cache that supports three-sided queries. Each node in the base tree corresponds to a one-dimensional range of x-values, called its x-range, and its $\Theta(B)$ children correspond to subranges consisting of vertical slabs as depicted in Figure 2.9. The child cache associated with each node w contains $O(B^2)$ nodes, $O(B)$ corresponding to each child u of the node, called the Y-set of the u node. The Y-Set of the node u contains all the points with the highest y-coordinate among the points within its x-range that are not already stored in ancestors of w. Y-sets of the leafs of the base tree must be either empty or contain at least $B/2$ points.

Querying The EPS-tree

Performing one or two-dimensional queries in the EPS-tree can be done in exactly the same manner as the simple B-tree. The primary function of the EPS-tree is performing I/O efficiently, three-sided queries $Q(a, b, c)$. A three-sided query starts at the root of the tree and descends to its children where their child caches are queried. A child u of a node v is visited if its full Y-set was reported, or its x-range contains the x-restriction points, a and b, of the query (a, b, c). This is done because if one or more points of the Y-set corresponding to u isn't reported, then all the subtree

rooted at u won't be reported either, due to the construction of the EPS-tree (all the nodes of the subtree rooted at u will have smaller y-coordinates than u).

The time complexity for querying the EPS-tree is optimal $O(\log_B N + T/B)$ I/Os. This upper bound can be explained as follows. The complexity of visiting an internal node v is $O(1 + T_v/B)$ I/Os, where T_v is the number of points reported in node v. The nodes on the path to the leaf containing the boundary points a, b are reported adding up to a complexity bound of $O(\log_B N + T/B)$. The other internal nodes v_i are reported only if their parent's full Y-sets corresponding to v_i and containing $\Theta(B)$ points are reported adding up $O(T/B)$ to the overall complexity.

Updating The EPS-tree

To insert a point $p(x, y)$ into an EPS-tree with root u, we are first inserting it's x coordinate into the base tree data structure. Then, a proper update of the appropriate child caches is needed, called a "bubble down" operation. During this operation we first query the child cache of the root to find the Y-set ,corresponding to the child u_i with the proper x-range which includes p_x. If p_y is smaller than the y-coordinates of all the $\Theta(B)$ points of the Y-set, then we insert recursively p into u_i. Otherwise we insert p into the Y-set and we insert the point with the smallest y-coordinate of the Y-set, recursively into u_i. If u is a leaf we simply store p into the associated block.

To delete a point p from an EPS-tree we perform a search and we delete p from the proper Y-set of node u corresponding to it's child u_i. Because the invariant that a Y-set must contain $\Theta(B)$ points has been violated with this operation, we must promote a point from u_i, with a recursive operation called "bubble up". During this operation we recursively promote the point p' with the highest y-coordinate from the child u_i belonging to a Y-set corresponding to the slab containing p. Afterwards, we delete the point p' from it's original Y-set.

To calculate the cost of inserting a point in the EPS we first calculate the cost of finding and inserting the point in the base tree which is $O(\log_B N)$. To query the child caches we use $O(\log_B B^2 + B/B) = O(1)I/Os$ amortized assuming that we are performing queries that return at most B points. When a node u of the base tree splits due to the insertion of the point p, then the corresponding child cache also splits. To maintain the invariant that a child cache contains $\Theta(B^2)$ points, we have to perform a rebalancing "bubble up" operation, which promotes $\Theta(B)$ points with the highest y-coordinate from the children of the node u to the child cache of u. In the worst case an insertion can cause $O(\log_B N)$ splits, so the cost of

the rebalancing operations will be $O(B \log_B N)$ in the worst case. However, these splits happen only one in every $B/2$ insertions, so by amortizing this cost we can conclude that the cost of the rebalancing operations is $O(\log_B N)$. Thus, the overall cost of inserting a point p into the EPS-tree is the optimal $O(\log_B N + T/B)$ I/Os.

2.6 The Cache-Oblivious Model

As computer systems become more complex, the memory hierarchy that is incorporated in them grows in the number of levels that it contains. As a result of that, the external memory model cannot accurately represent the contemporary computer systems, because it considers only a two-level memory hierarchy (main memory and external storage) and the external memory algorithms depend on the M and B parameters of the internal memory and the external storage. In practice though, we may have several levels in our memory hierarchy with varying parameters.

The need to describe contemporary computer systems even more accurately, led to the adoption of the **Cache-Oblivious Model** by Frigo et al. [FLPR99]. The cache-oblivious model, assumes no knowledge about the memory hierarchy. In essence, a cache-oblivious algorithm is an algorithm formulated in the RAM model but analyzed in the I/O model, with the analysis required to hold for any B and M. Memory transfers are assumed to be performed by an on-line optimal replacement strategy. The beauty of the cache-oblivious model is that since the I/O-model analysis holds for any block and memory size, it holds for all levels of a multi-level memory hierarchy. In other words, by optimizing an algorithm to one unknown level of the memory hierarchy, it is optimized on all levels simultaneously. Thus the cache-oblivious model is effectively a way of modeling a complicated multi-level memory hierarchy using the simple two-level I/O-model.

Another benefit of the cache-oblivious model is self-tuning. Typical cache-aware algorithms require tuning to several cache parameters which are not always available from the manufacturer and often difficult to extract automatically. Parameter tuning makes code portability difficult. Perhaps the first and most obvious motivation for cache-oblivious algorithms is the lack of such tuning: a single algorithm should work well on all machines without modification.

The complexity of the cache-oblivious algorithms is studied under the **ideal cache model** (Z, L), which enables us to reason about a two level memory model like the external memory model but prove results about a multilevel memory model. This model, which is illustrated in Figure 2.10, consists of a computer with a two-

level memory hierarchy consisting of an ideal (data) cache of Z words and an arbitrarily large main memory. The cache is partitioned into **cache lines**, each consisting of L consecutive words which are always moved together between cache and main memory. Cache designers typically use $L > 1$, banking on spatial locality to amortize the overhead of moving the cache line. We shall generally assume in this paper that the cache is tall:

$$Z = \Omega(L^2)$$

which is usually true in practice. The model is built upon some basic assumptions.

The following four assumptions are key to the model.

- **Optimal replacement** The *replacement policy* refers to the policy chosen to replace a block when a cache miss occurs and the cache is full. In most hardware, this is implemented as FIFO, LRU or Random. The model assumes that the cache line chosen for replacement is the one that is accessed furthest in the future. The strategy is called optimal on-line replacement strategy.

- **Two levels of memory** There are certain assumptions in the model regarding the two levels of memory chosen. They should follow the *inclusion property* which says that data cannot be present at level i unless it is present at level $i + 1$. In most systems, the inclusion property holds. Another assumption is that the size of level i of the memory hierarchy is strictly smaller than level $i + 1$.

- **Full associativity** When a block of data is fetched from the slower level of the memory, it can reside in any part of the faster level.

- **Automatic replacement** When a block is to be brought in the faster level of the memory, it is automatically done by the OS/hardware and the algorithm designer does not have to care about it while designing the algorithm. Note that we could access single blocks for reading and writing in the external memory model, which is not allowed in the cache oblivious model.

Unlike various other hierarchical-memory models in which algorithms are analyzed in terms of a single measure, the ideal-cache model uses two measures. An algorithm with an input of size n is measured by its **work complexity** $W(n)$, –its conventional running time in a RAM model– and its **cache complexity** $Q(n; Z; L)$ –the number of cache misses it incurs as a function of the size Z and line length L of the ideal cache. When Z and L are clear from context, we denote the cache complexity simply as $Q(n)$ to ease notation.

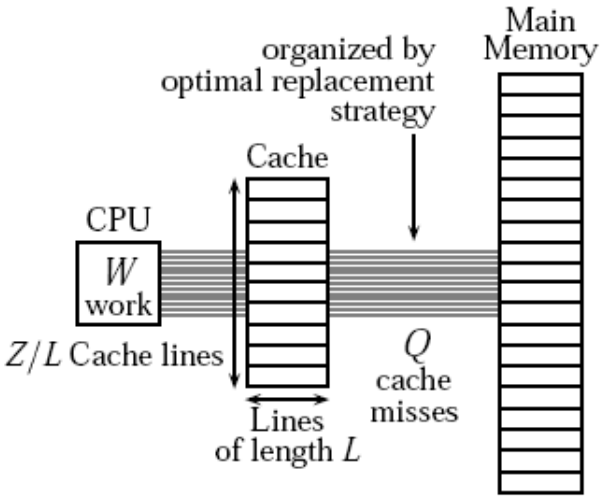


Figure 2.10: The ideal-cache model

2.6.1 The Van Emde Boas Layout

A fundamental data structure that is optimal for the cache-oblivious model is the **van Emde Boas** layout. This data structure is merely a layout of data in a memory array, such that answering a search of an element among N comparable elements, can be answered in optimal $O(\log_B N)$ memory transfers. What makes this data structure optimal for the cache-oblivious model, is that we don't have to know about the **B** parameter. This data structure corresponds to a static binary search tree in the RAM model, but with much better performance for memory hierarchies.

To construct the Van Emde Boas layout for N items, we just construct a binary search tree for these items. This tree will be stored sequentially in memory according to a recursive layout (Figure 2.11). Conceptually split the tree at the middle level of edges, resulting in one top recursize subtree and roughly $\sqrt{N}$ bottom recursive subtrees, each of size roughly $\sqrt{N}$. Recursively lay out the top recursive subtree, followed by each of the bottom recursive subtrees. Each recursive subtree is laid out in a single segment of memory, and these segments are stored together without gaps.

The van Emde Boas layout is a kind of divide-and-conquer algorithm, except that just the layout is divide-and-conquer, whereas the search algorithm is just the usual tree-search algorithm: look at the root and go left or right appropriately. One way to support the search navigation is to store left and right pointers at each node.

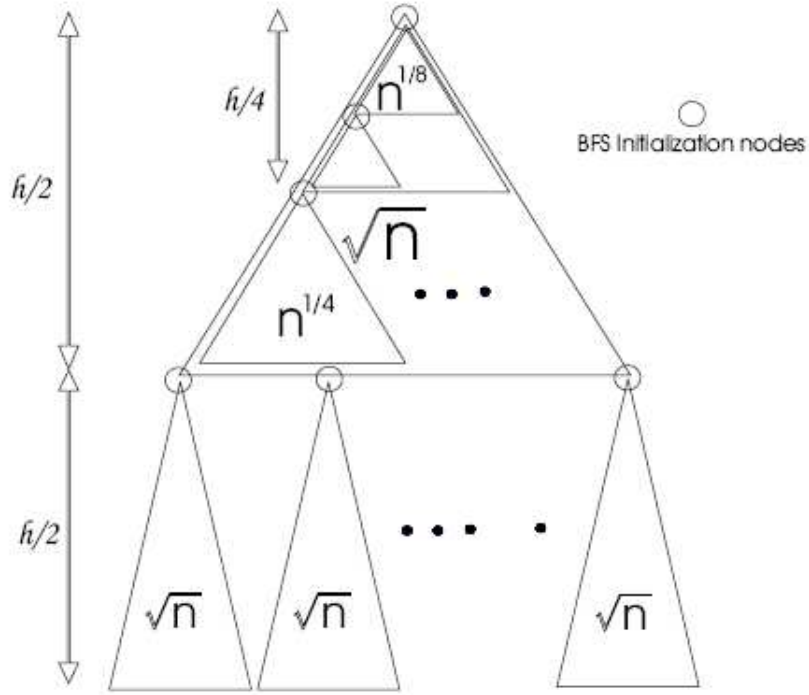


Figure 2.11: The van Emde Boas layout.

Chapter 3

Experimental Evaluation Of Data Structures

The performance analysis of external memory data structures has mainly concentrated on their I/O efficiency. However, as main memory cost decreases, it is usual in practice to load a data structure almost entirely in main memory. The CPU cost thus in these cases, overwhelms the I/O cost. In internal memory data structures on the other hand, disk latencies are almost totally disregarded when analyzing them with the RAM model, even though data intensive applications need to access secondary data storage frequently.

In this chapter we try to address this issue, by experimentally evaluating both the I/O cost as well as the CPU cost of querying and loading external memory data structures. Internal memory data structures query times are also evaluated by altering the data set size and noticing the effects of the introduction of high disk latencies. Various data structures such as the R-Tree and variants, the EPS tree and the priority tree are compared and conclusions are drawn from their relative performance under various data sets.

3.1 Implementation Details

3.1.1 The EMIL Library

All of our external memory algorithm implementations make use of the External Memory Infrastructure Library (EMIL) written by Vasilis Samoladas. EMIL is

a C++ library that abstracts external memory management primitives. The programmer can thus implement disk-oriented data structures almost as easy as writing main-memory data structures. My contribution was to port the documentation to the Doxygen documentation tool [Dox].

External memory is organized in **blocks** in EMIL. The size of the block is a user defined multiple of the operating system's page size. The effect of the choice of the block size in the overall performance of the external memory data structures, is examined experimentally. A set of blocks in EMIL is organized in **pools**, which is an abstraction for files in external storage. Blocks are accessed in EMIL through **pins** which contain information about the block's pool and it's unique id. Pins are much like pointers in regular programming, which makes it easy for programmers to implement external memory data structures. EMIL has also a buffer manager for the blocks, which uses an LRU replacement policy, minimizing the I/Os needed to fetch the blocks from disk to internal memory and thus making it more efficient.

3.1.2 Experiment Setup

All of the data structures were implemented in C++ using the EMIL library for the external memory data structures. An External Priority Search Tree (EPST) was implemented using the algorithms from [Sam01]. Two implementations of a simple R-tree, one using the EMIL library and one main-memory, were implemented as a straightforward application of the algorithms in [Gut85]. The Linear-Cost algorithm was used as a choice for handling node overflows, due to it's decreased complexity. A Hilbert R-Tree was constructed, because it is a popular R-tree variant. A bulk-loaded version of the Hilbert R-tree was also implemented, using algorithms from [KF93]. From the different methods proposed, we chose to compute the Hilbert values for the center of the rectangles.

To evaluate the external memory data structures, various three-sided queries were performed. We chose two different synthetic datasets and evaluated the data structures by querying them with different query sizes and aspects. We also varied the block size in pages to measure its effect in query performance.

We constructed two kinds of synthetic datasets for the experiments; a Fibonacci dataset and a uniform one with varying sizes (1,000,000 , 10,000,000 and 100,000,000 keys). Using these datasets a number of trees were constructed and queried by a set of uniformly distributed queries. The expected query size was varied from 10 to 50,000 keys. The block size was varied from one to four operating system pages. We also measured the time to load the datasets on the trees.

The quantities that were measured to evaluate the performance of the external memory data structures were the total running (real) time, the CPU time and the number of disk I/Os. To neutralize the effect of the operating system buffer, that would alter the number of disk I/Os, we turned the operating system's swap memory off. Because EMIL has also got a buffer and we needed to make sure that the pages requested were from the disk and not from the main memory buffer of EMIL, we reserved a large amount of main memory within our programs.

Our primary concern is to identify the hidden constant in the asymptotic analysis of the external memory data structures. We also want to examine by varying the dataset size, at which point the I/O cost overwhelms the CPU cost and for relatively small datasets, where does CPU cost dominate the total running time. We also want to study the performance of loading the data structures with a relatively large dataset. We expect that the bulk-loaded data structures will greatly outperform the data structures built with individual insertion. Bulk-loading is also expected to enhance the quality of the data structures and its overall performance, which is to be experimentally asserted. Finally, we expect that by choosing small block sizes, the performance will be better for small data sets and worse for large.

3.1.3 Results

Bulk-Loading Data Structures

The loading time of an online data structure is arguably less important than the query time for most applications. Still, the construction time must be reasonable for a data structure to have any practical use, and can be used to further discriminate between solutions that have similar query performance. We expect that the data structures that are bulk-loaded with a static dataset, will perform much better than the data structures that are built with individual insertions. Additionaly, we expect that the data structures that are built with heuristic algorithms, such as the R-Tree and variants will pay an extra overhead for visiting irrelevant nodes for some insertions. The main memory data structures will perform well until the dataset gets big enough so that it can't fit in main memory. In that point the disk I/Os will dominate the total running time.

We experimented with the bulk-loading efficiency of the EPST, the Hilbert R-Tree, the Packed Hilbert R-Tree and the Priority tree. We constructed several synthetic Fibonacci datasets with sizes ranging from 100,000 points to 30,000,000 points. Each point is a struct holding two unsigned integers. The block size for all

the data structures was set to one operating system page. For each dataset we constructed the data structures and we measured their efficiency in terms of CPU and real time in nanoseconds, needed to complete the bulk-loading. The EPST, priority tree and the Hilbert R-Tree were built with individual insertions using straightforward implementations of the algorithms described in chapter 2. The paked Hilbert R-Tree was bulk-loaded with the algorithm Hilbert-Pack as in [KF93]. To sort the rectangles on ascending hilbert values we used an external merge-sort algorithm.

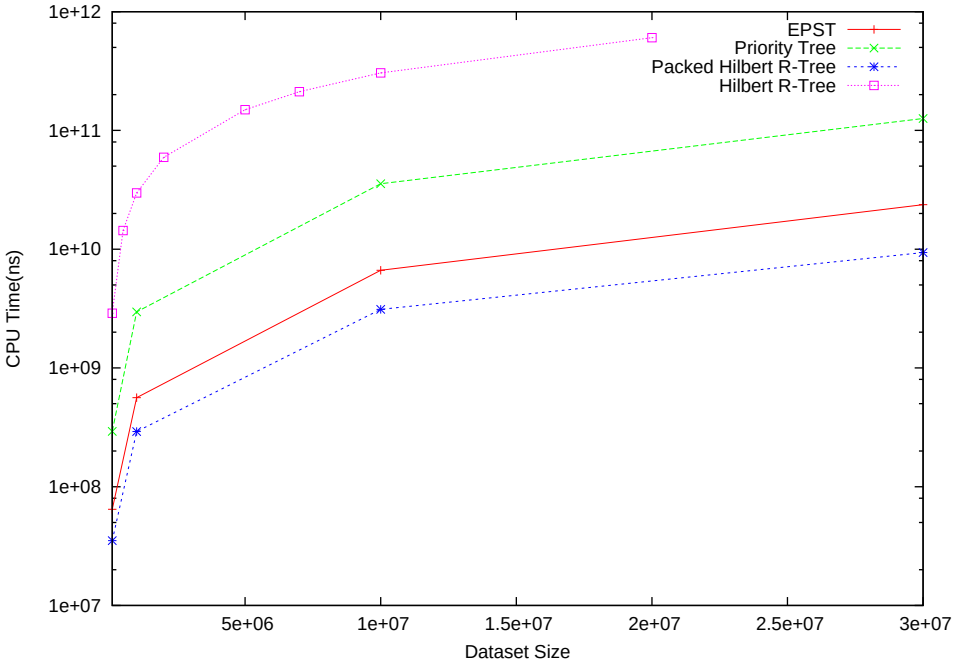
The results are shown in Figure 3.1 where we can see the CPU and the real time in nanoseconds to bulk-load the EPST, the Priority tree, the Packed Hilbert and the Hilbert R-Tree for various dataset sizes. As we expected the Packed Hilbert R-Tree outperforms the data structures that are loaded with individual insertions. The priority tree performs worse than the EPST due to the increased number of I/Os needed for a main memory data structure when loaded with a large dataset. The simple Hilbert R-Tree performs poorly mainly because many irrelevant nodes are visited at each insertion. We also notice that for large datasets the total running time for the priority tree increases dramaticaly. This is because of the disk I/Os that greatly affect the performance.

Query Performance

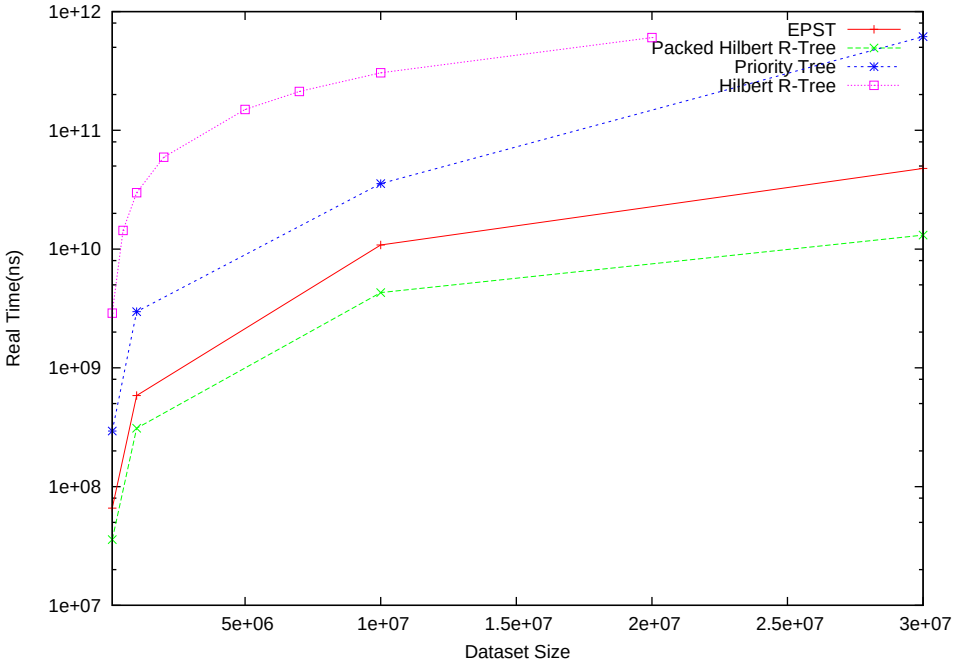
Measuring The Performance Of Data Structures In Terms Of Disk I/Os

The query performance is the most important factor when we want to examine the quality of a data structure. That's why we are going to analyze every factor affecting the query performance of the data structures. When dealing with massive datasets, the disk I/Os can become a major performance bottleneck affecting query performance. The total time needed to fetch a block from the disk can be orders of magnitude higher than the time needed to fetch a page from the main memory. That's why a data structure that deals with massive data needs to be I/O efficient.

We will query and examine the disk I/O efficiency of the EPST, the R-Tree and the Hilbert R-Tree. The trees are loaded with various datasets and the queries are rectangles and three-sided queries of various sizes and aspects. We expect that the Hilbert R-Tree will outperform the simple R-Tree for some queries, because of it's improved insertion algorithms. Because of these algorithms, the Hilbert R-Tree achieves better space utilization which affects the query performance. Many of the queries that were conducted have a high aspect ratio. We expect that the R-Trees will perform poorly under these queries because they will have to visit many redun-



(a) CPU Time



(b) Real Time

Figure 3.1: Plot showing CPU and real time(ns) needed to bulk load various data structures, against the dataset size.

dant nodes. Additionally, because of the heuristic nature of the R-Trees, we expect that they will perform well only for certain datasets and queries.

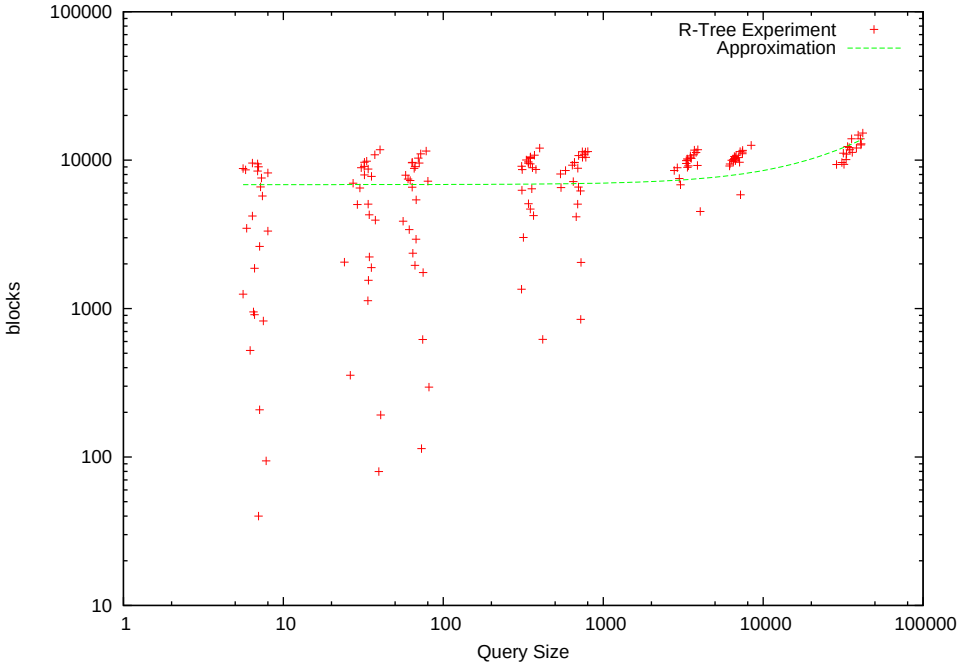
The trees were constructed with a uniform and a Fibonacci dataset, both of them of size 10,000,000 points. The block size was set to two operating system pages. The queries were a set of rectangles for the R-Trees and variants and a set of three-sided queries for the EPST of various sizes. The query algorithms that were used, are straightforward implementations of the algorithms described in chapter 2. To measure raw disk I/Os and not the blocks that were fetched by the operating system's or the Emil's buffer, we reserved a large amount of main memory for our programs and we turned the operating system's swap memory off.

The results are shown in Figure 3.2 where we can see scatter plots that depict the disk I/Os against the query size for a simple R-Tree, a Hilbert R-Tree and in Figure 3.3 for the EPST. The trees are loaded with a Fibonacci synthetic dataset. There is also a least square approximation of the results depicted in these figures. From these results we notice as we expected, that the R-Trees perform poorly compared to the EPST. We notice also that there is a tight correlation between the query size and the disk I/Os in the EPST, which we don't see in the R-Tree and variants. This is because, even though some queries return a small number of points, the R-Trees still have to visit a great number of irrelevant nodes. The type of these queries simply don't abide with the assumptions made in the heuristic R-Tree algorithms. We also observe that, as we expected, the Hilbert R-Tree indeed outperforms the simple R-Tree in terms of disk I/Os.

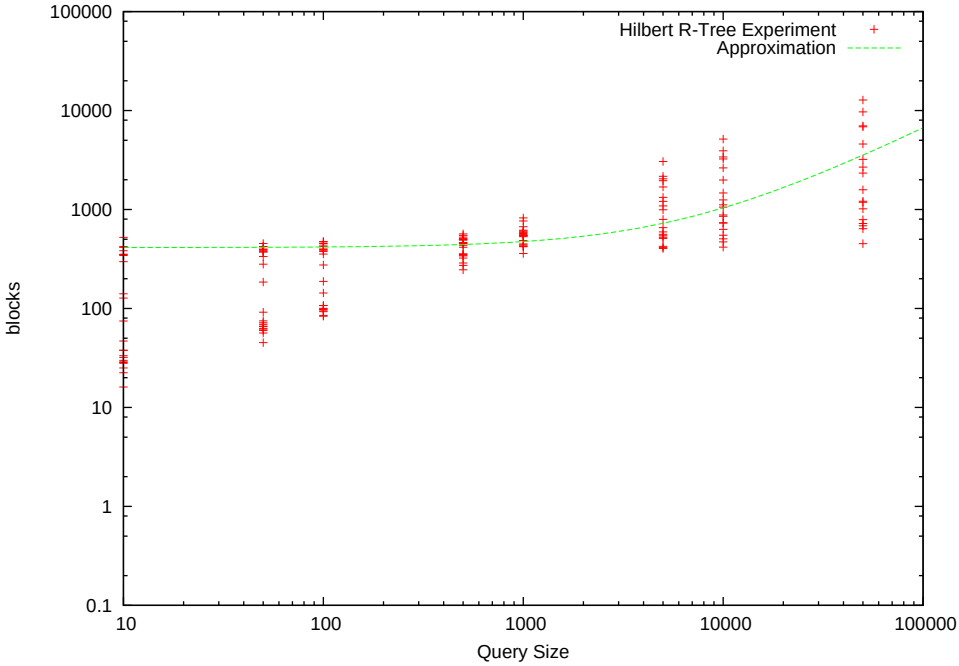
In Figure 3.4 we can see a scatter plot of disk I/Os vs the query size for a simple R-Tree loaded with a uniform synthetic dataset.The block size was set to four operating system pages. We observe that, even though the performance is still rather poor, the I/Os are correlated with the query size. A "good" dataset thus, greatly affects the performance and the quality of the resulting R-Tree which is not the case for the EPST.

Measuring The Performance Of Data Structures In Terms Of CPU Time

When the dataset is small enough to fit in the main memory of the computer the major factor that affects the query performance of a data structure is the CPU time needed. But with the introduction of cheap and large capacity main memory, even for massive datasets we can have all, or a great part of the tree fit in the main memory. In these cases, the CPU time will play a significant role in the overall performance of the data structure and thus we cannot afford to ignore it in our



(a) R-Tree



(b) Hilbert R-Tree

Figure 3.2: Plot showing Disk I/Os vs the query size for the R-Tree and the Hilbert R-Tree.

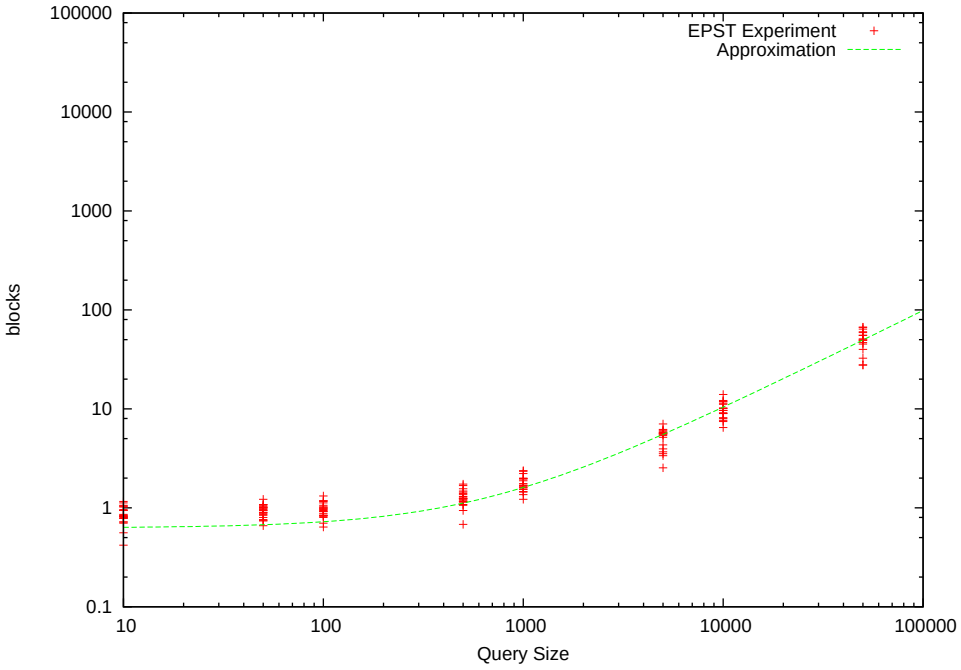


Figure 3.3: Plot showing Disk I/Os vs the query size for the EPST.

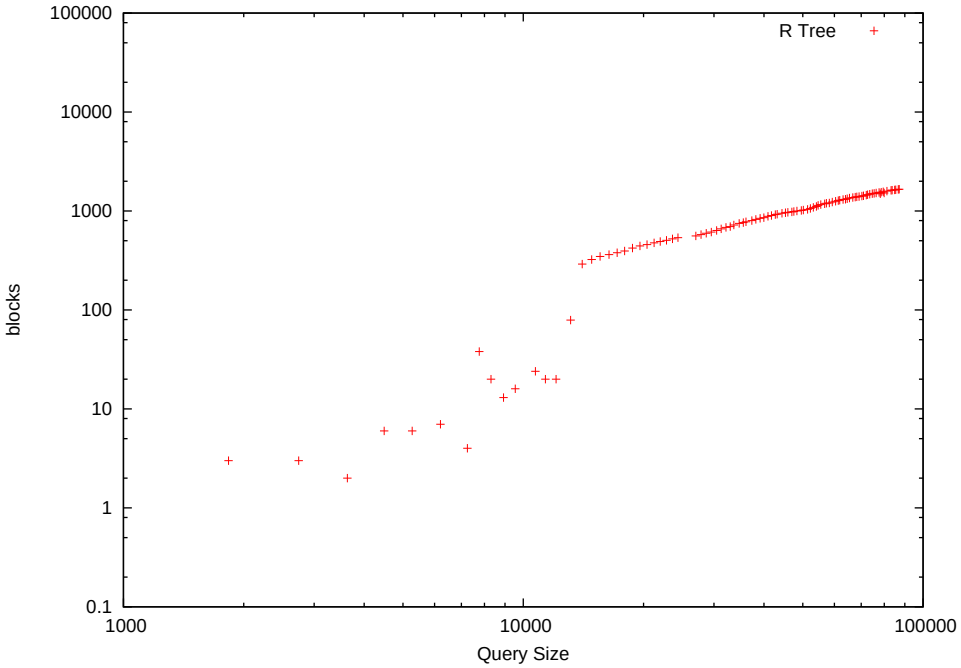


Figure 3.4: Plot showing Disk I/Os vs the query size for the R-Tree loaded with a uniform dataset.

analysis.

We will evaluate experimentally in terms of CPU time the Packed Hilbert R-Tree, the Hilbert R-Tree, the EPST, the R-Tree, the priority tree and the main memory R-Tree. We expect that data structures that have performance guarantees such as the EPST and the priority tree, will perform overall better than the heuristic data structures such as the R-Tree and variants. Additionally, we expect that the bulk-loaded data structures such as the packed Hilbert R-Tree will perform better than the equivalent data structures that were built with individual insertions, because of the better space utilization that they exhibit. Again we expect that for "suitable" datasets, such as the uniform, and certain types of queries, the R-Trees and variants will perform much better.

All the trees were loaded with two datasets of size 10,000,000 points. The one was a Fibonacci and the other a Uniform dataset. The block size for the external memory data structures loaded with the Fibonacci dataset was set to one operating system page. For the data structures loaded with a uniform dataset, the block size was configured to two operating system pages. The data structures were queried by a set of three-sided queries for the EPST and the priority tree and rectangles for the R-Trees and variants, of varying size and aspect. The time was measured in nanoseconds

Figures 3.5, 3.6 show the CPU time in nanoseconds against the query size for the EPST, the Packed Hilbert R-Tree and the R-Tree. We notice, as we expected, that the R-Trees perform poorly compared to the EPST due to the nature of the dataset and the queries. Again we can see in the R-Tree and variants, that even for small query sizes we have a wide spread in the CPU time for different queries, which is because of their aspects. We also notice that the Packed Hilbert R-Tree indeed outperforms the simple R-Tree in all kinds of queries

In 3.7 we can see how a Hilbert R-Tree and an EPST, both loaded with a uniform dataset, perform in terms of CPU time. Again the EPST outperforms the Hilbert R-Tree, but we observe that there is a tight correlation between the query size and the CPU time that doesn't exist for the Fibonacci dataset for the R-Trees. The type of dataset thus, severely affects the R-Tree's and variants' performance.

For the main memory data structures, we can see in 3.8 the Priority and the R-Tree's CPU performance against the query size, for a Fibonacci dataset. From these diagrams, we notice that the main memory data structures perform better than their external memory counterparts when the query size is small. This is because of the overhead that is introduced by the external memory framework. We also notice that the priority tree is not affected by the queries' aspect and exhibits a tight correlation between query size and CPU time, which is not the case for the

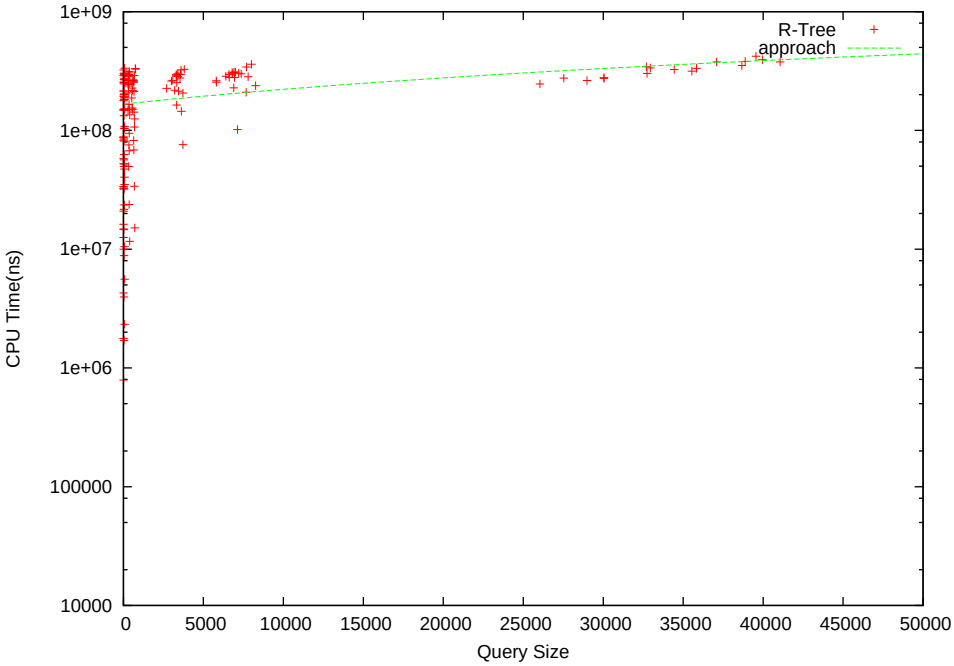
R-Tree.

Effect Of Diks I/Os And CPU Time In The Total Running Time Of A Query

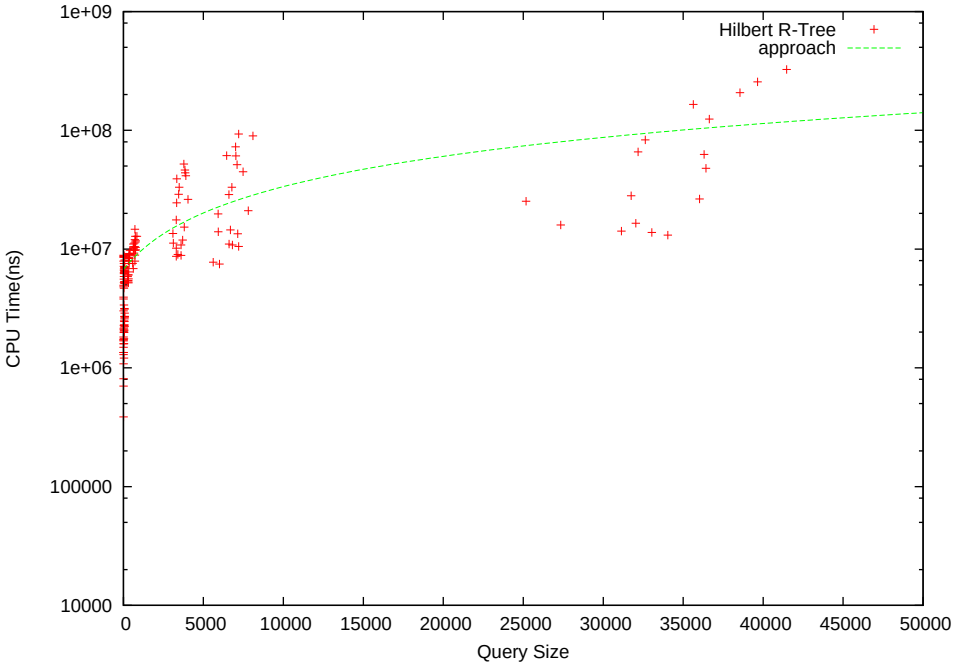
Main memory data structures are analyzed and implemented in a way that the CPU cost is minimized, disregarding any other factor. It is certain however, that when the dataset size exceeds the size of the main memory, disk I/Os contribute significantly in the total running time of a query and in fact, they sometimes dominate it. External memory data structures on the other hand are implemented in a way such that the number of disk I/Os will be as small as possible, disregarding any other factor. However, for certain datasets and queries the CPU cost contributes significantly in the total running time of a query. In this section we are going to experimentally investigate the correlation between disk I/Os and CPU time with the total running time, when querying data structures. We are expecting, that for main memory data strucures the CPU cost will be the dominant factor for small dataset sizes, but after a certain dataset size the introduction of disk I/Os will severely affect the trees' performance. For external memory data structures, we are expecting that for small to medium dataset sizes the CPU cost will not be negligible as their model imply, but as the dataset size increases, the contribution of the disk I/Os to the total running time will be significantly bigger.

In this section we measured the total running time and the CPU time in nanoseconds of a query returning approximately 50,000 points for the priority tree. We also measured the disk I/Os needed to query a Packed Hilbert R-Tree where the query size is again approximately 50,000 points. We loaded the data structures with Fibonacci datasets of various sizes, from 10,000 to 30,000,000 points for the priority tree and from 10,000 to 60,000,000 points for the Packed Hilbert R-Tree. The block size for the Packed Hilbert R-Tree was configured to one operating system page.

In Figure 3.9 we can see a plot of the total time and the CPU time needed for a query that returns approximately 50,000 points, against the dataset size for the priority tree. As we can see from the diagram, the total time is almost linearly correlated to the dataset size, until the size of 20,000,000 points when we notice a sharp increase of the query time. From this point on the disk I/Os introduce a high overhead, making this main memory data structure impractical. We also notice how little does the CPU time contribute in the total running time for large datasets. In Figure 3.10 we can see the disk I/Os needed to query packed Hilbert R-Trees of various sizes. We notice that for small but not negligible dataset sizes, the number

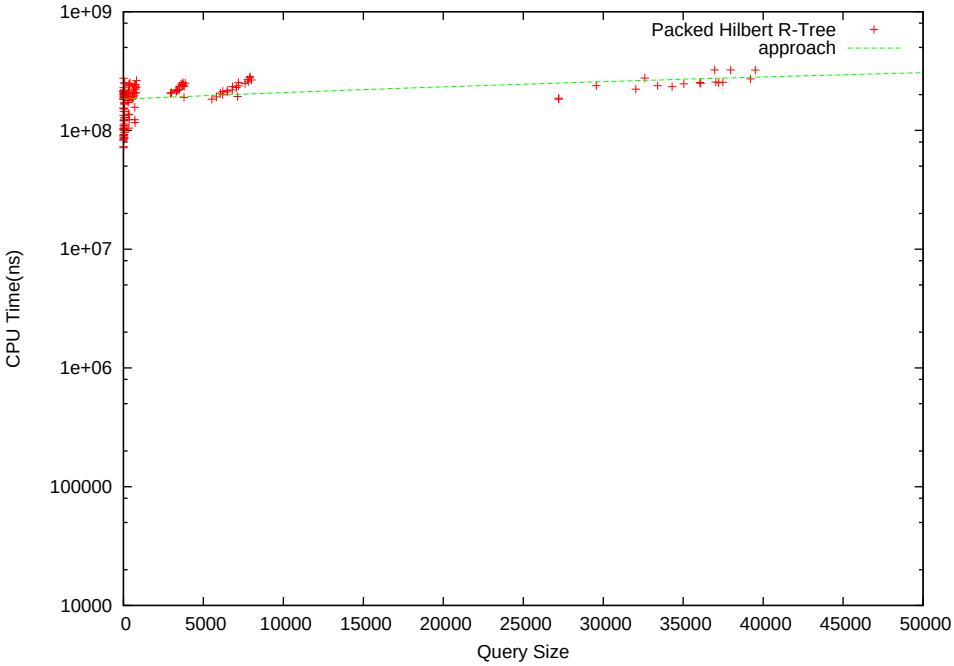


(a) R-Tree

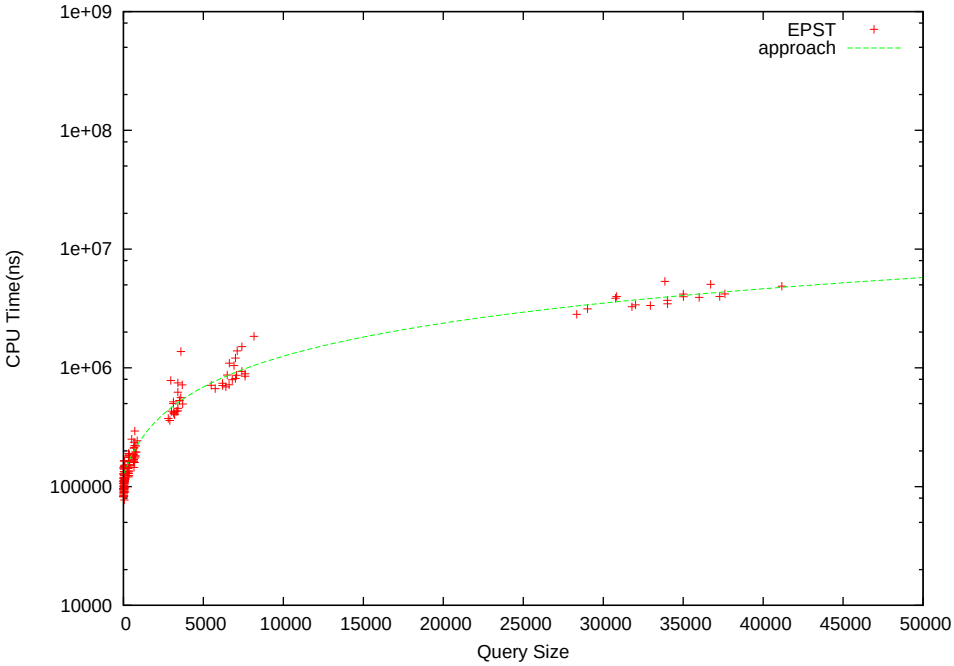


(b) Hilbert R-Tree

Figure 3.5: Scatter plot showing CPU time vs the query size for the R-Tree and the Hilbert R-Tree.

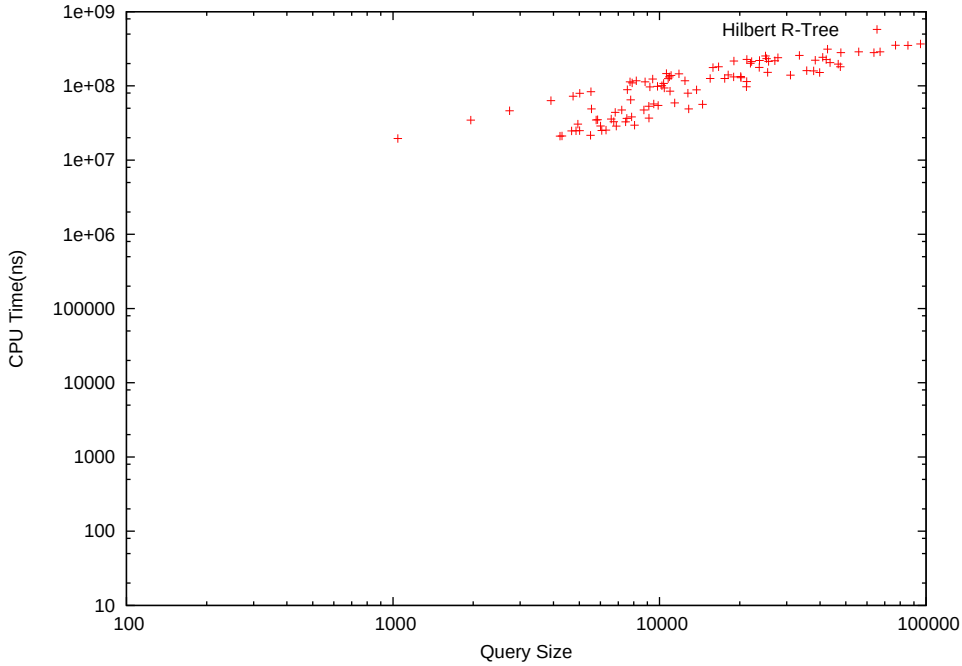


(a) Packed Hilbert R-Tree

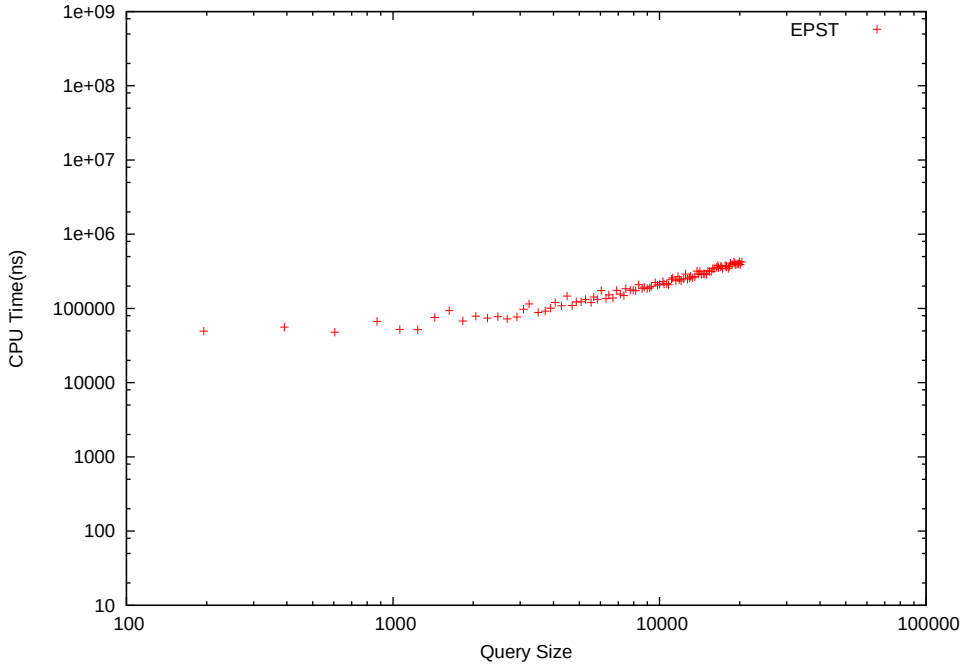


(b) EPST

Figure 3.6: Scatter plot showing CPU time vs the query size for the Packed Hilbert R-Tree and the EPST.

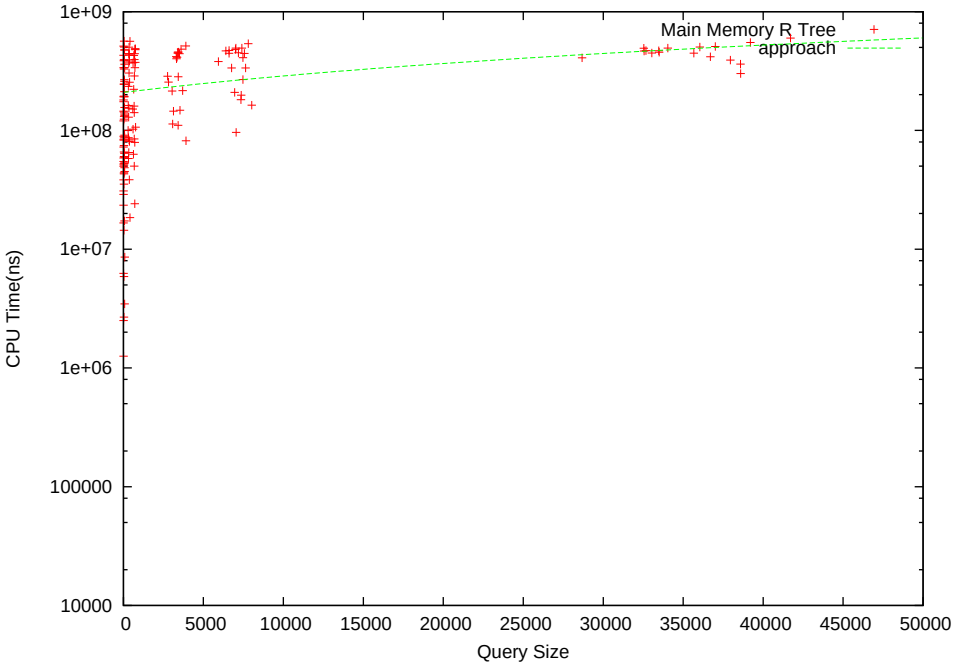


(a) Hilbert R-Tree

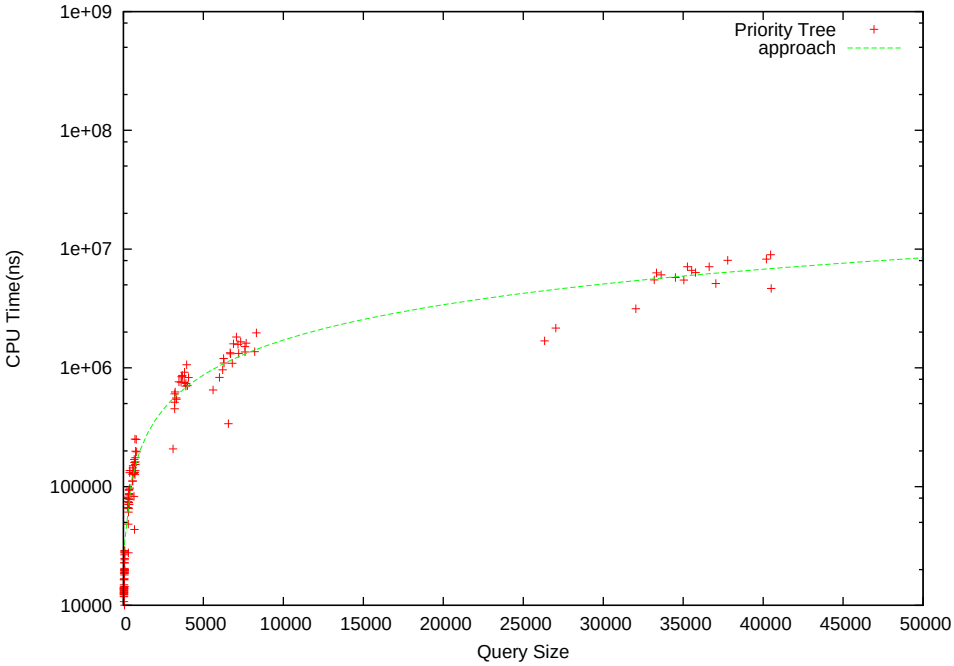


(b) EPST

Figure 3.7: Scatter plot showing CPU time vs the query size for the Hilbert R-Tree and the EPST loaded with a uniform dataset.



(a) Main Memory R-Tree



(b) Priority Tree

Figure 3.8: Scatter plot showing CPU time vs the query size for the Priority Tree and the Main Memory R-Tree.

of disk I/Os is rather small and thus they play a small role in the overall tree's performance. After the dataset size of approximately 10,000,000 points the number of disk I/Os sharply increases and so do their contribution to the total running time of the query.

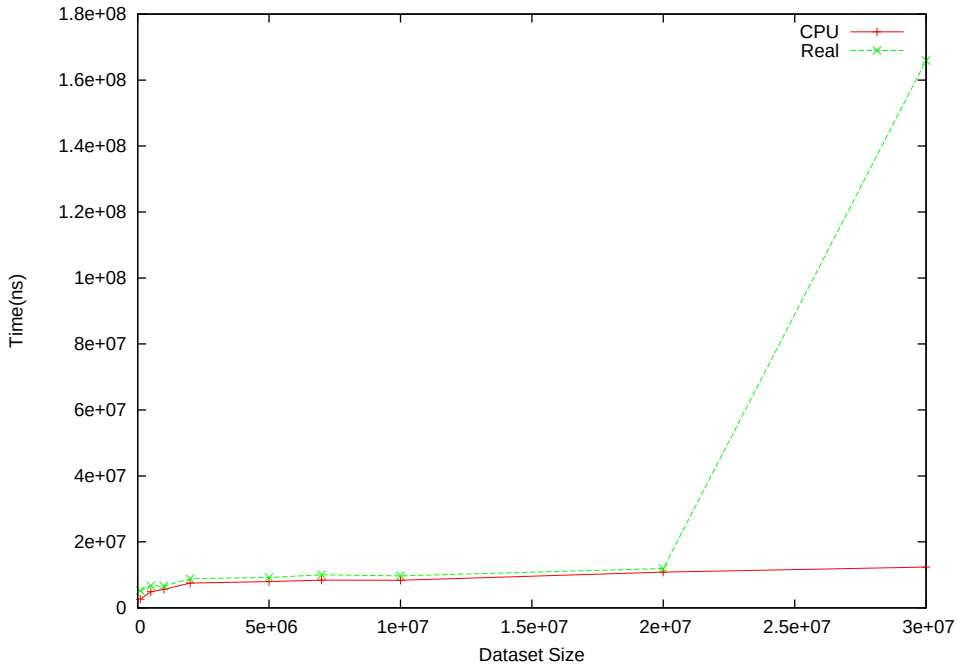


Figure 3.9: Total time against dataset sizes for main memory priority search trees.

Effect Of The Block Size In Query Performance

External memory data structures transfer data between the main memory and the disk via blocks of user defined size B. By transfering data in chunks, they are able to reduce the number of I/Os needed for a query and improving their performance. We are expecting thus, that by increasing the block size for an external memory data structure, it's query performance will be improved. However, by increasing the block size the CPU cost is also increased. There must be therefore, a limit where the increase of the block size will actually not be beneficial for the overall query performance of a data structure.

In this section we experimented with EPSTs loaded with Fibonacci datasets of

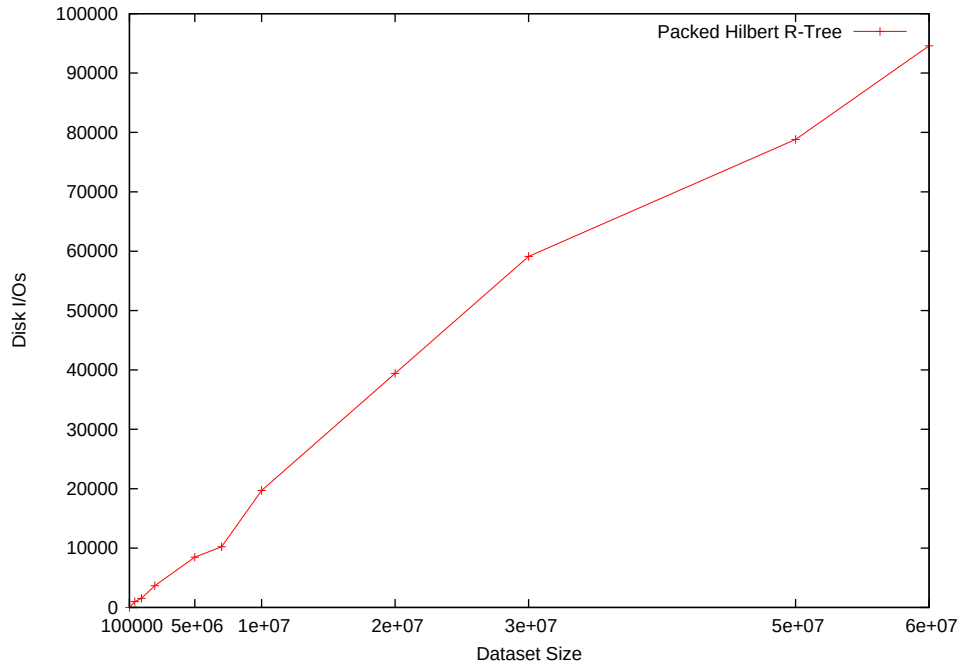


Figure 3.10: Disk I/Os against dataset sizes for Packed Hilbert R-trees.

sizes 1,000,000 points and 100,000,000 points with varying block sizes (1,2,4 pages). We measured the query performance of these trees, by running a set of three-sided queries and measuring the CPU time and the total running time in nanoseconds and the disk I/Os.

In Figures 3.11, 3.12, 3.13 we see the effects of varying the block size (1,2,4 pages) on the EPST performance (CPU time, disk I/Os, total time). The trees are loaded with a relatively small Fibonacci dataset of 1,000,000 points that fits in the main memory. We notice that a small increase of the block size is beneficial for the overall tree performance. Of course, the increase in the tree's performance is not linearly correlated with the block size, as we can also see from the diagrams, and we expect that for a very large block size, the performance would actually deteriorate.

In Figures 3.14, 3.15, 3.16, we can see two EPSTs loaded with a large dataset of 100,000,000 points that doesn't fit in the main memory. The one has block size set to one page and the other set to four pages. We notice that here the overall performance deteriorates when we increase the block size. As we can see from Figure 3.15, even though the number of disk I/Os decreases as we increase the block

size, the overall query time (Figure 3.16) increases. This result directly disproves the assertion that is made in the external memory model, where only the disk I/Os are taken into account and not the CPU time, in the analysis of data structures.

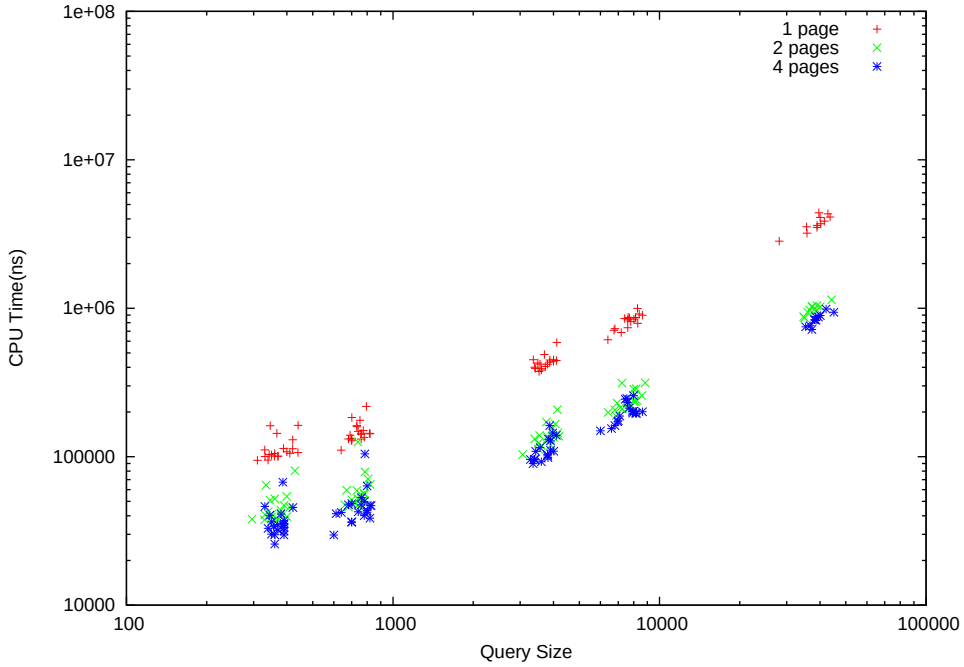


Figure 3.11: Scatter plot showing the effect of the block size in CPU time for the EPST loaded with a small dataset.

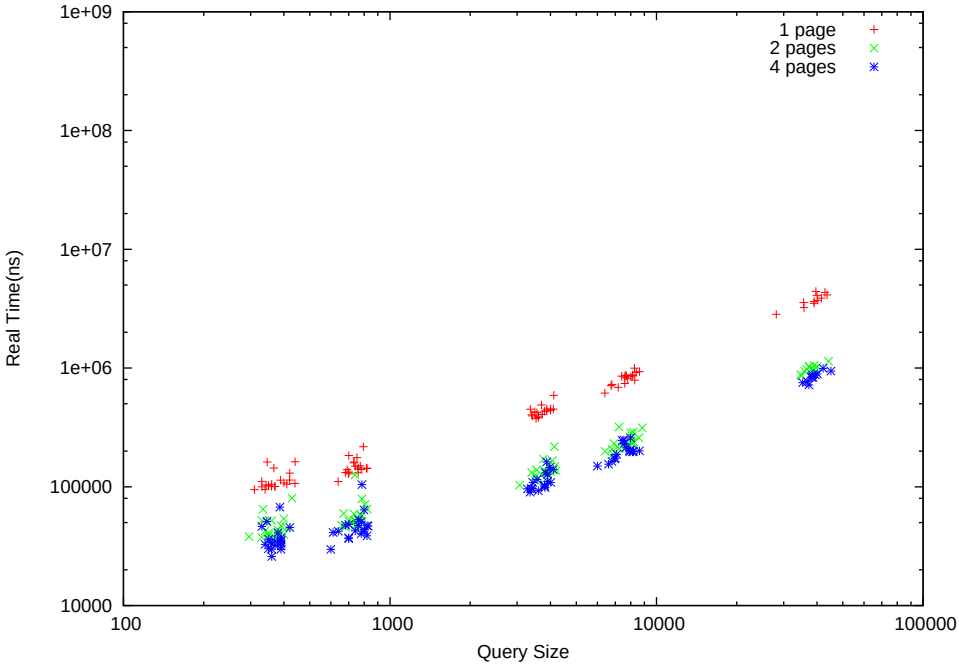


Figure 3.12: Scatter plot showing the effect of the block size in disk IOs for the EPST loaded with a small dataset.

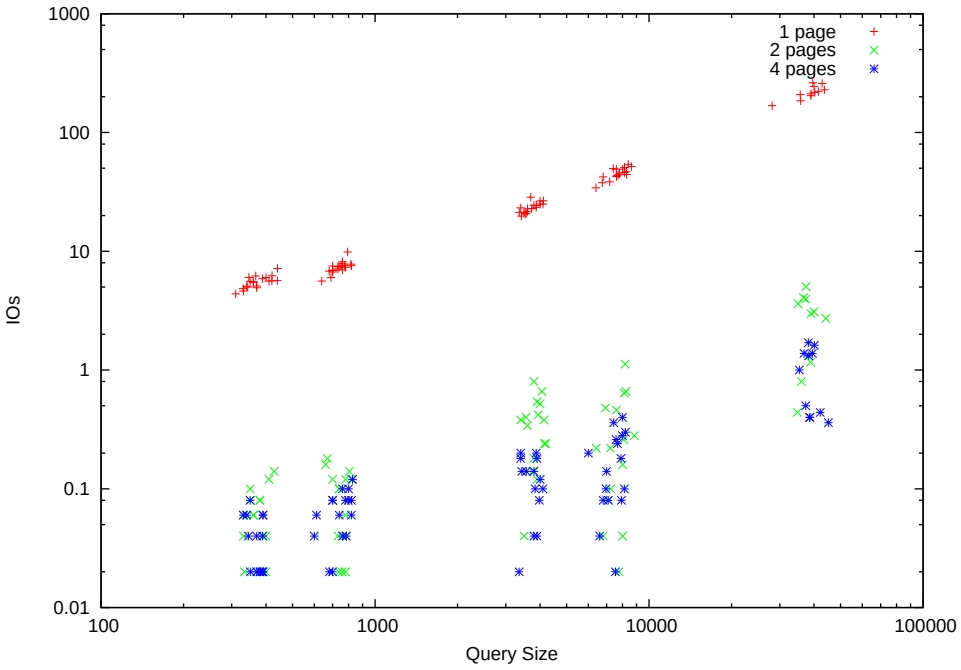


Figure 3.13: Scatter plot showing the effect of the block size in real time for the EPST loaded with a small dataset.

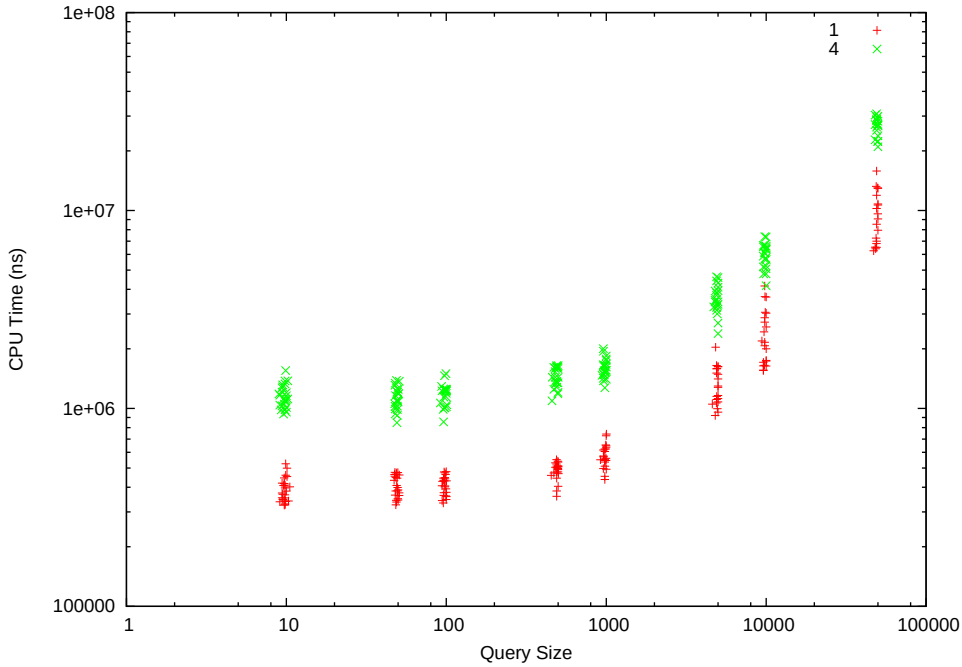


Figure 3.14: Scatter plot showing the effect of the block size in CPU time for the EPST loaded with a large dataset.

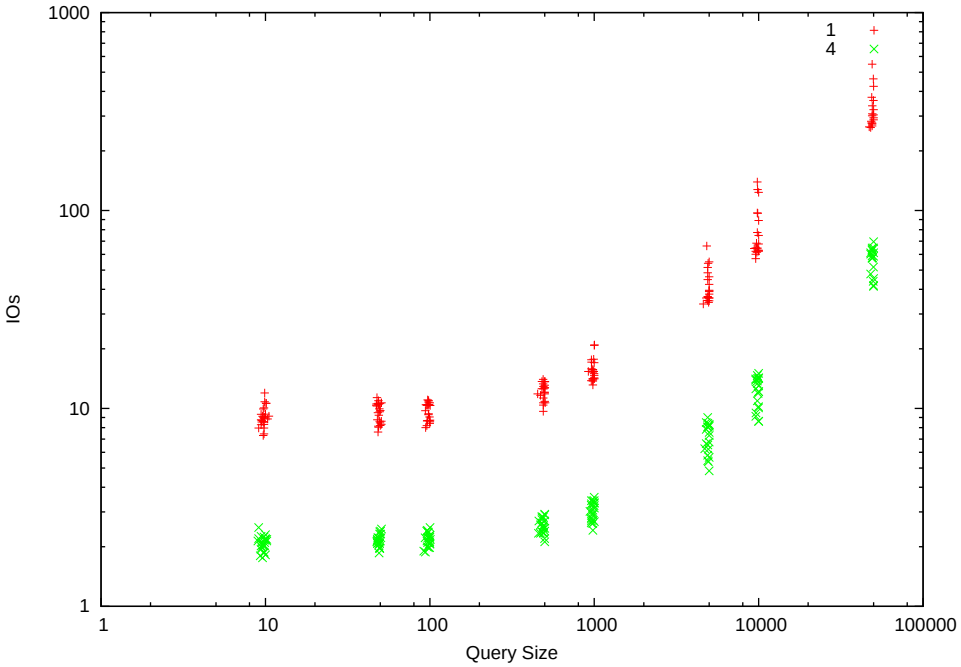


Figure 3.15: Scatter plot showing the effect of the block size in disk I/Os for the EPST loaded with a large dataset.

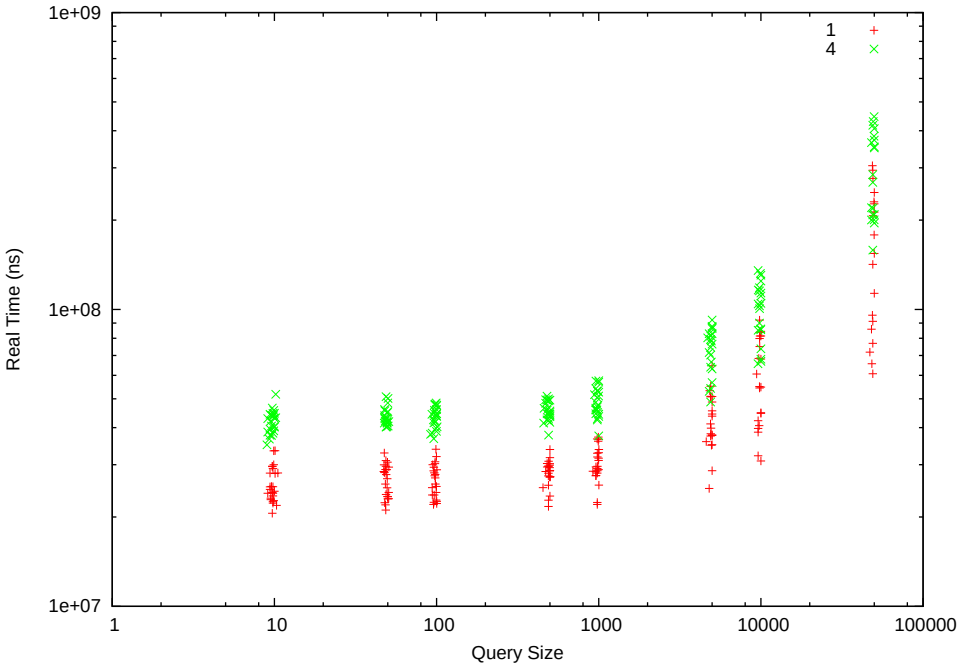


Figure 3.16: Scatter plot showing the effect of the block size in real time for the EPST loaded with a large dataset.

Chapter 4

Parallel Bulk-Loading Hilbert R-Trees

4.1 The MapReduce Model

The need to process vast amounts of data in large organizations like Google or Yahoo!, has led to the development of highly scalable and efficient parallel systems. The majority of the computations that take place in such large organizations are conceptually straightforward. However, the input data is usually large and have to be distributed across hundreds of thousands of machines in order to finish in a reasonable amount of time. Unfortunately, the majority of the distributed systems that are available, obscure the original simple computation with large amounts of complex code that deals with issues of parallelism.

As a reaction to this complexity, a new abstraction was designed to separate the messy details of parallelization, such as fault tolerance, data distribution and load balancing, from the actual computations. This abstraction is called the **MapReduce** [DG04] model and was inspired by functional programming primitives. Computations with the MapReduce model, involve applying a *map* operation to each logical record in our input, in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. All computations that can be expressed with this model, can be parallelized easily, regardless of their size.

The computations in the MapReduce model take a set of key/value pairs as input, and produce a set of output key/value pairs. The user must provide only two functions: *Map* and *Reduce*. Map function provided by the user, must take

an input pair and provide a set of *intermediate* key/value pairs. The MapReduce library will group together all intermediate values associated with the same key, and pass them to the user defined Reduce function. The Reduce function accepts an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set of values. All the other issues of parallelization are handled by the MapReduce library and are abstracted from the user. Many interesting data intensive applications can be expressed in the MapReduce model, such as *distributed grep*, *reverse web-link graph*, *distributed sort* and many more.

4.1.1 Apache Hadoop

A widely used open source implementation of the MapReduce interface is **Hadoop** [Had] by the Apache Software Foundation. Hadoop is a framework that allows running applications on large clusters built of commodity hardware. Clusters can consist of hundreds or thousands of machines, and therefore machine failures are common. Node failures are automatically handled by the Hadoop framework. In addition, it provides a distributed file system (HDFS) that stores data on the compute nodes, providing very high aggregate bandwidth across the cluster. All the messy details of parallelism are abstracted from the user.

The HDFS filesystem stores large files across multiple machines. It achieves reliability by replicating the data across multiple hosts, and hence does not require RAID storage on hosts. The filesystem is built from a cluster of data nodes, each of which serves up blocks of data over the network using a block protocol specific to HDFS. They also serve the data over HTTP, allowing access to all content from a web browser or other client. Data nodes can talk to each other to rebalance data, to move copies around, and to keep the replication of data high.

HDFS requires one unique server, the name node. This is a single point of failure for an HDFS installation. If the name node goes down, the filesystem is offline. When it comes back up, the name node must replay all outstanding operations. This replay process can take over half an hour for a big cluster. The filesystem includes what is called a Secondary Namenode, which regularly connects with the namenode and downloads a snapshot of the primary Namenode's directory information, which is then saved to a directory. This Secondary Namenode is used together with the edit log of the Primary Namenode to create an up-to-date directory structure.

Above the file systems comes the MapReduce engine, which consists of one Job Tracker, to which client applications submit MapReduce jobs. The Job Tracker pushes work out to available Task Tracker nodes in the cluster, striving to keep

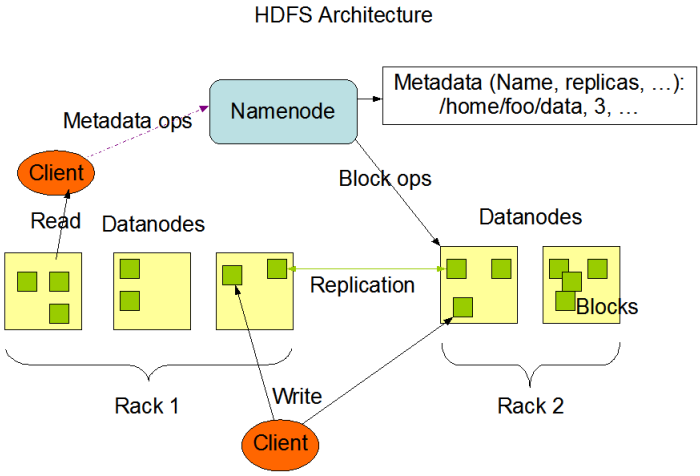


Figure 4.1: HDFS Architecture.

the work as close to the data as possible. With a rack-aware filesystem, the Job Tracker knows which node contains the data, and which other machines are nearby. If the work cannot be hosted on the actual node where the data resides, priority is given to nodes in the same rack. This reduces network traffic on the main backbone network. If a Task Tracker fails or times out, that part of the job is rescheduled. If the Job Tracker fails, all ongoing work is lost.

4.2 Previous Work

Because of the great importance of the R-tree and its prevalent use in modern database systems for indexing spatial data, there is a lot of research for building efficient indexes in comparatively short time. Bulk-loading an R-Tree in parallel, would thus seem as a natural idea to dramatically decrease the time consumption of this process. However, even though a lot of work has been done for devising efficient serial bulk-loading techniques of spatial data structures, little has been done to address the issue of bulk-loading in parallel.

Papadopoulos et al [PM03] have created an algorithm that partitions the space in a way similar to the construction of a kd-tree and assign one partition to each processor. Each processor creates a local subindex. All of the subindices are finally

merged to a global index. The authors claim that their method is efficient and exploits parallelism to a certain degree. Another recent work on parallel bulk-loading R-trees has been done by Cary et al [CSHR09]. In their work they use Google's MapReduce model to bulk load a simple R-tree. Their algorithm, computes a partitioning function f, that partitions the given dataset D into R parts. This function uses the idea of space-filling curves, specifically the z-order curve to map the multidimensional data objects into an ordered sequence of single-dimensional values. The *mappers* use this function to divide the given dataset into R partitions and assign them to R reducers. Then, the reducers build R independent R-tree indices. At the final stage the local R-tree indices are merged into a single global R-tree. The authors tested experimentally these methods on a grid and showed that for a high level of parallelization, there is a significant gain in the time needed to bulk-load an R-tree.

4.3 Bulk-Loading Hilbert R-Trees With MapReduce

To parallelize the process of bulk-loading Hilbert R-Trees using the MapReduce framework, we just need to devise an algorithm that follows the functional programming primitives of MapReduce. The input of our problem is a set of tuples of the form $< x_1, y_1, x_2, y_2 >$ that represent the minimum bounding rectangles (MBRs) of our spatial data. We want each node of our cluster to process a subset of these tuples. The first problem we encounter is to find an optimal way to distribute those records to the processors, so that we can achieve maximum load-balancing between the nodes, and achieve the highest possible locality.

What we need therefore, is an optimal partitioning of the data into k equal parts, where k is the number of the nodes. The obvious way, is to sort the input data according to their Hilbert value, and then pick the k quantiles. However, sorting the data introduces a significant overhead, especially for large input datasets. We can avoid sorting the data, by using data sampling as in [Mal08] the TeraSort Hadoop application.

We perform a MapReduce operation to calculate the optimal partitioning of the data. A uniform random sample of size M is extracted from the input dataset and is fed as input to our Map function. The Mapper application receives the input data and calculates in parallel the Hilbert values of the centers of the input MBRs.

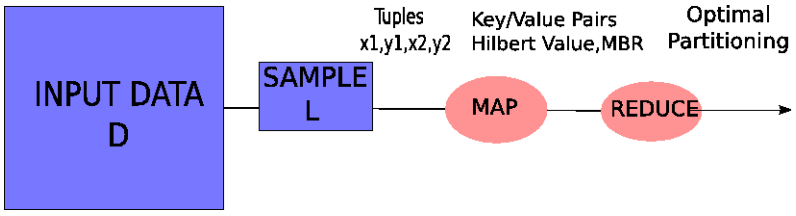


Figure 4.2: Calculating the optimal partitioning of the data.

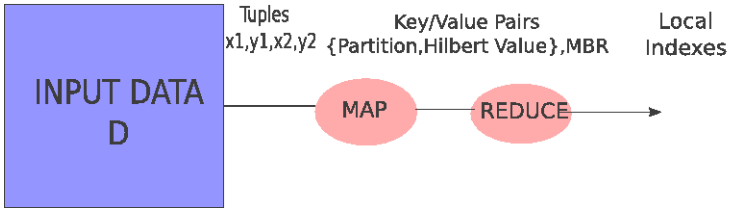


Figure 4.3: Bulk-loading a Hilbert R-Tree.

Then, it emits key value pairs, where the key of each record is it's Hilbert value, and the value is the MBR. The records are sorted automatically according to their keys by the Hadoop system, and are fed to a **single** reducer, which calculates the partition intervals by simply splitting the dataset into k equal parts. We expect that the size M of our sample will have an effect in the quality of the partitioning. The optimal intervals are written in a text file and are persisted in the HDFS. Note that no actual partitioning or data moving happens at this point. The next face utilizes the optimal intervals to assign each node a proper subset of the records.

In the next phase we perform a second MapReduce application. The Mappers receive the input data and read the text file with the optimal partitioning of the data. Then, they emit a key wich consists of the partition number r that each tuple belongs ($1 <= r <= k$) and the hilbert value of the tuple. They also emit a value for each record, which is the MBR of the spatial data. The partioner of the Hadoop framework assigns to each of the r reducers a proper subset of the data. The reducers built a local Hilbert R-Tree from the subsets. The trees are then copied in one of the node's hard drive. In the final step, which is done serially because of it's low complexity, a global tree is created, by constructing a root that has the local indices' roots as children.

4.4 Experiments

In this thesis we used a cluster of the Softnet laboratory, which contains five nodes running open source software, including the Linux operating system and Apache's Hadoop. Each node has approximately half terrabyte of storage and a quad core Intel Xeon CPU of 2.50 GHz. Access to the cluster is provided by the secure shell protocol.

Interaction with the cluster is done through Hadoop's shell scripts. To upload the input data on the cluster, a special script is used, which saves data as a set of files in HDFS. These files are in turn stored as a sequence of blocks (typically of 64 MB in size) that are replicated on multiple nodes to provide fault-tolerance. The language that is mainly used with Hadoop is Java, but because we used the EMIL framework, which is written in C++, MapReduce application was written in C++. We used the **Streaming** utility that comes with the Hadoop distribution and works in a way similar to Unix pipes. The utility allows us to create and run map/reduce jobs with any executable or script as mapper or reducer. The data are fed as input from standard input to the mapper script, which outputs the intermediate values. The reducer script, then receives the intermediate values as input from standard input and emits the output key/value pairs.

The experiments were conducted with real and synthetic uniform datasets of varying size. We ran the experiments with one node (pseudoparallelism), and with all the nodes. The number of mappers was configured to 10 and the number of reducers to 5 in most of the experiments. Because load-balancing is handled in our application, the number of reducers was defined to be the optimal minimum of approximatelly 0.95 multiplied by number of nodes.

In table 4.1 we can see the total time in seconds for bulk-loading Hilbert R-Trees in parallel with various dataset sizes. As expected, there are great differences in performance between full parallelism and pseudo-parallelism for large datasets, where we have the additional benefit that by splitting our dataset in smaller sizes, much less I/Os are needed to bulk-load the local R-trees. This is because each local subindex is much smaller than the overall tree and can thus fit entirely or partially in the main memory of the nodes. For small dataset sizes, because of the replication of data and the general overhead that the Hadoop distributed system introduces, the pseudo-distributed mode seems to outperform the parallel mode. We also run the experiments with a small real dataset (MBRs of the Greek roads Figure 4.4 of size approximately 1 MB), for full parallelism and without using hadoop at all. Because the size of the dataset is so small, the serial version actually performed better than the parallel version, again because of the large overhead that the Hadoop and

Table 4.1: Job completion times for bulk-loading Hilbert R-Trees with MapReduce measured in seconds, for various dataset sizes, exploiting full parallelism and pseudo-parallelism.

Dataset Size	Total Time	Reducers
10,000	20	5
100,000	22.612	5
1,000,000	35.725	5
10,000,000	88.197	5
30,000,000	201.632	5
50,000,000	340.136	5
Greek Roads	19	5
10,000	11.2	1
100,000	12.58	1
1,000,000	27.37	1
30,000,000	499.44	1
50,000,000	839.42	1
70,000,000	1022.67	1
Greek Roads	0.89	serial

the HDFS system introduce. It is also interesting, to compare the running times for bulk-loading Hilbert R-Trees with synthetic datasets of different sizes in parallel and serially without using the Hadoop system. Figure 4.5 depicts the user time in seconds, needed to bulk-load Hilbert R-Trees of various sizes, in parallel and serialy, against the dataset size, where we can draw similar conclusions.

We expect that by removing nodes from our parallel system, the performance will deteriorate, at least for a relatively large dataset. To prove this, we conducted experiments where we measured the total time in seconds to bulk-load Hilbert R trees with a dataset of 20,000,000 points, and altered the number of working nodes in Hadoop. The results can be seen in Figure 4.6. As can be observed from the results, even though linear scalability is not achieved, there is a significant improvement in performance, as we increase the number of nodes.

We also experimented with really large datasets that exceed the main memory of

Figure 4.4: Real dataset containing the minimum bounding rectangles of the greek roads [Rtr].

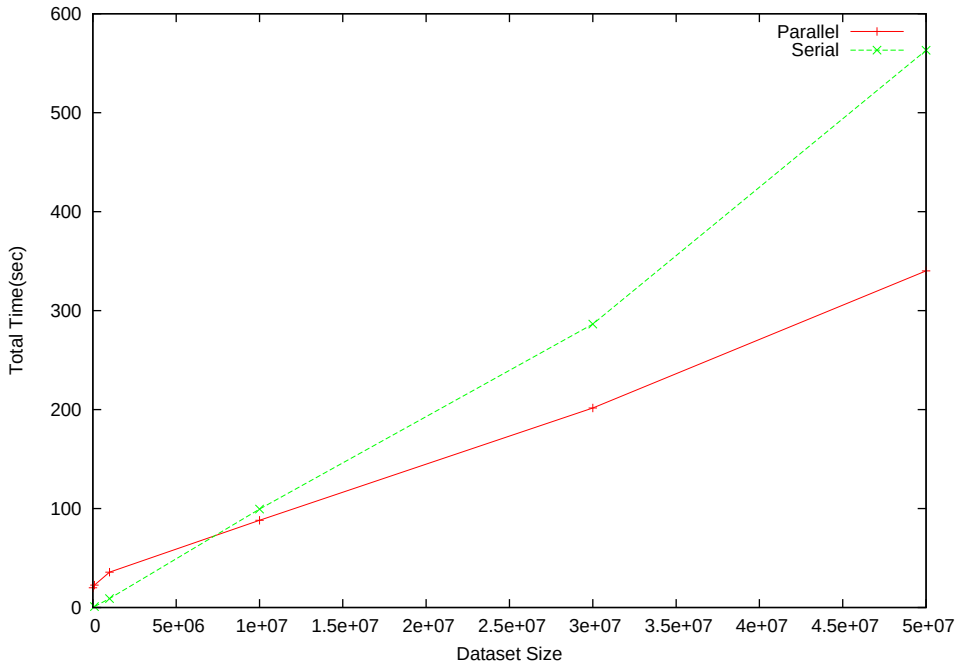


Figure 4.5: User time to load Hilbert R-Trees of various sizes in parallel and serially.

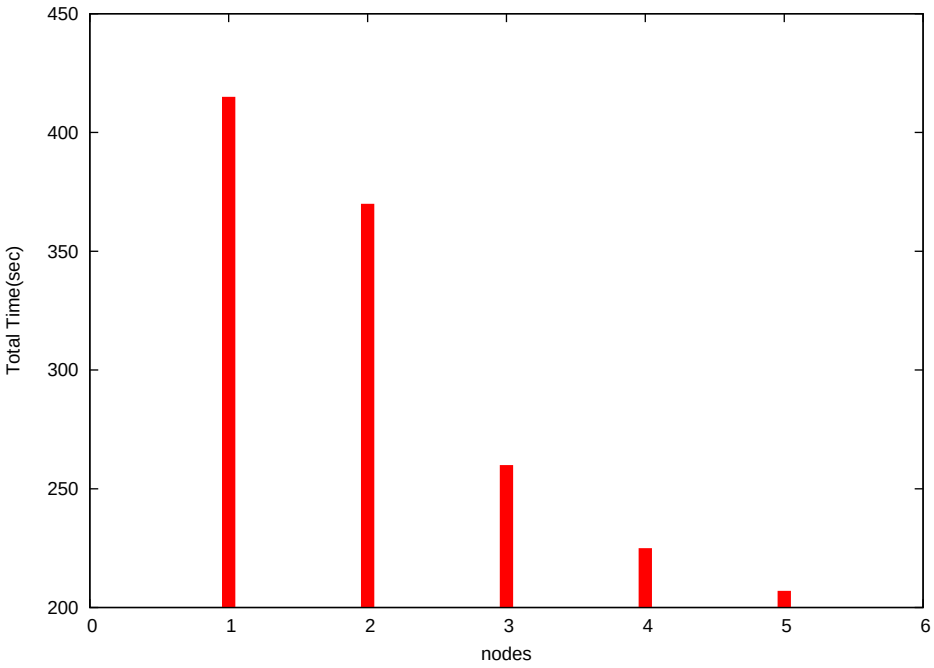


Figure 4.6: Total time to bulk-load Hilbert R-Trees for various number of nodes.

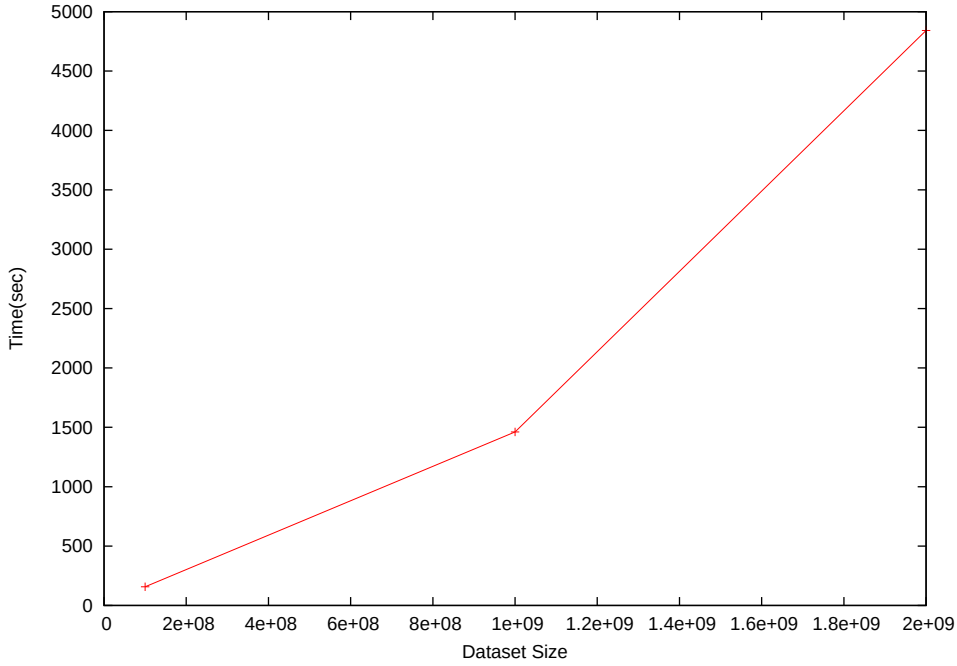


Figure 4.7: Total time to bulk-load Cube Trees in parallel for various dataset sizes.

all the nodes combined (20 GB). For that end we constructed a cubetree [RKR97], which is a simple packed R-Tree, that sorts the data points first by y coordinate and then by x coordinate. Three datasets were used, one relatively small (1.8G), one that is about the size of nodes' main memory(18G) and one that exceeds the size of nodes' main memory(36G). As we can see in 4.7 that depicts the total time to bulk-load cubetrees with the Hadoop parallel system for various dataset sizes, there is a steep increase in time as we exceed the main memory size. This is explained mainly by the introduction of disk I/Os, that greatly affect the overall performance.

The first part of our algorithm, computes the optimal partitioning of the data, by using a random sample and calculating the k quantiles, where k is the number of the nodes. Because we only sample a small part of the data, we expect a slight miscalculation of the optimal partitions. This would result in imperfect load-balancing and the locality of the data would also be affected. We investigated these hypotheses, by conducting queries on the trees that were created in the previous step. We expect that there will be a slight deterioration of the tree's performance, when we

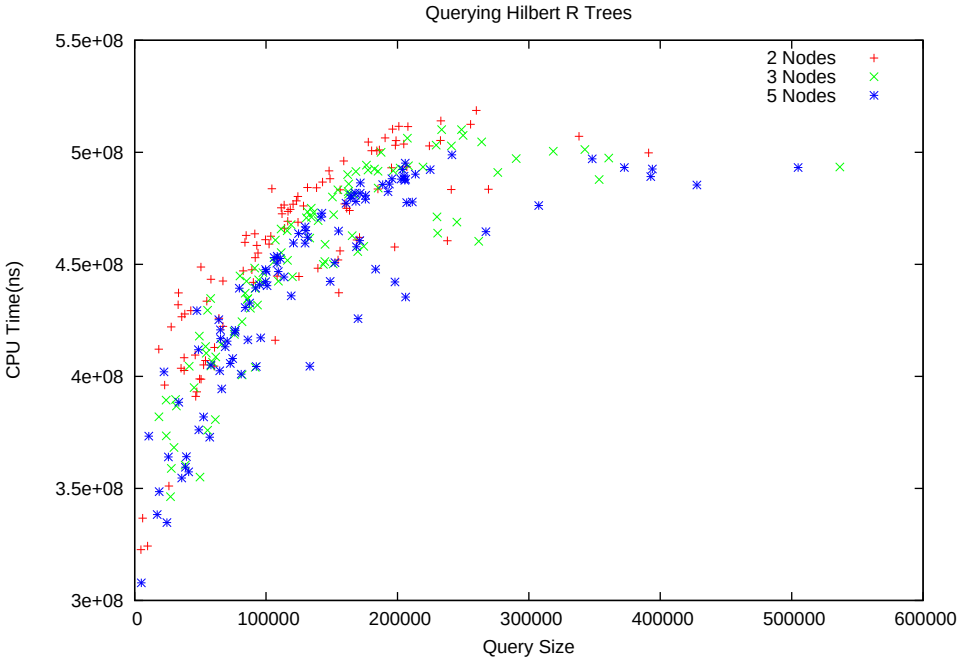


Figure 4.8: Query performance of Hilbert R Trees, created in parallel with various number of nodes.

increase the number of nodes, due to imperfect load-balancing. Figure 4.8 is a scatter plot of the CPU time in nanoseconds, against the query size, for three Hilbert R Trees, bulk-loaded with the Hadoop system, using two, three and five nodes. The results show that the difference in the tree's performance is insignificant, proving that a small change in the number of nodes doesn't result in deterioration of the tree's performance.

Chapter 5

Conclusions

In this dissertation we experimented with several well-known data structures and noticed their overall efficiency for various types of datasets and query types. We showed that data structures with proven performance guarantees such as the priority, the EPST and the B Tree, perform better, especially for certain datasets and queries, than the data structures that rely on heuristic algorithms, such as the R Tree and variants. These data structures perform well only under certain datasets and queries which is very limiting in practise.

Another contribution of this thesis is that we proved that both the CPU cost and the disk I/O cost, can be relevant for certain types of datasets. These results directly contradict the assumptions made in both the external and the main memory model. We also experimentally proven that the increase of the block size affects the data structures' performance negatively or positively, depending on the dataset size.

We also devised an algorithm that expresses the process of bulk-loading Hilbert R-Trees with the MapReduce model. The very promising Hadoop implementation of the MapReduce system was used to experimentally evaluate our algorithm. The algorithm has proven to be scalable in a cluster of five nodes. The trees produced from the parallel system have almost the same query performance, whith trees produced with serial bulk-loading.

This work can stimulate research for a unified memory model that will take in a proper manner under consideration both the CPU and the I/O cost of an algorithm. On the practical side, there are many promising data structures with theoritically proved optimal performance, such as the Priority R-Tree [AdBHY04], which need to be experimentaly evaluated. This evaluation will decide about the practical usefulness of these data structures and will assist their adoption by the database industry.

There is also much future work to be done for fully parallelizing external memory data structures with the MapReduce model. Very important work can be done on parallelizing the query process for an index, or maybe executing parallel joins and scans.

Bibliography

[AdBHY04] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The priority r-tree: A practically efficient and worst-case-optimal r-tree. In Lars Arge, Michael A. Bender, Erik D. Demaine, Charles E. Leiserson, and Kurt Mehlhorn, editors, *Cache-Oblivious and Cache-Aware Algorithms*, volume 04301 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2004.

[ASV99] L. Arge, V. Samoladas, and J.S. Vitter. On two-dimensional indexability and optimal range search indexing. 1999.

[AV88] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[Bay72] R. Bayer. Symmetric binary b-trees: Data structures and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[BG90] G. Blankenagel and R. H. Güting. XP-trees—External priority search trees. Technical report, FernUniversität Hagen, Informatik-Bericht Nr. 92, 1990.

[BGO+96] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass., 1990.

[Com79] D. Comer. The ubiquitous B-tree. 11(2):121–137, 1979.

[CSHR09] Ariel Cary, Zheengguo Sun, Vagelis Hristidis, and Naphtali Rishe. Experiences on processing spatial data with mapreduce. 2009.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce:simplified data processing on large clusters. *OSDI 2004*, 2004.

[Dox] http://www.stack.nl/ dimitri/doxygen/.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[Gut85] A. Guttman. R-trees: A dynamic index structure for spatial searching. pages 47–57, 1985.

[Had] http://hadoop.apache.org.

[IKO87] Ch. Icking, R. Klein, and Th. Ottmann. Priority search trees in secondary memory. In *, LNCS 314*, pages 84–93, 1987.

[KF93] I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. 2nd International Conference on Information and Knowledge Management (CIKM)*, pages 490–499, 1993.

[KF94] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings 20th International Conference on Very Large Databases*, pages 500–509, 1994.

[Mal08] Owen O' Maley. Terabyte sort on apache hadoop, 2008.

[McC85a] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.

[McC85b] E.M. McCreight. Priority search trees. 14(2):257–276, 1985.

[PM03] Apostolos Papadopoulos and Yannis Manolopoulos. Parallel bulk-loading of spatial data. *Parallel Comput.*, 29(10):1419–1444, 2003.

[RKR97] Nick Roussopoulos, Yannis Kotidis, and Mema Roussopoulos. Cubetree: Organization of and bulk incremental updates on the data cube. In *Proceedings of the 1997 ACM SIGMOD Conference*, pages 89–99, 1997.

[RL85] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. pages 17–31, 1985.

[RS94] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. pages 25–35, 1994.

[Rtr] http://www.rtreeportal.org/.

[Sam01] V. Samoladas. *On Indexing Large Databases for Advanced Data Models*. PhD thesis, University of Texas at Austin, August 2001.

[Sch80] A Schonage. Storage modification machines. *SIAM Journal on Computing*, 9:490–508, 1980.