Three-sided Range Reporting in External Memory

Peter Gabrielsen, 20114179

Christoffer Holbæk Hansen, 20114637

Master's Thesis, Computer Science

June 2016

Project Advisor: Kasper Green Larsen

Formal Advisor: Gerth Stølting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

The world has become increasingly data driven and almost everything in our daily lives generate data. The generated data sets are typically larger than we can fit in a normal computer's internal memory. We therefore need clever techniques to handle and analyse the large data sets in order to extrapolate information. Much of the data that is gathered is spatial in nature, i.e. it contains e.g. GPS coordinates. Analysing coordinates require us to solve complicated problems within the field of computational geometry. One such problem is that of three-sided range reporting. In this problem we are asked to maintain a dynamic set, $\mathcal{S}$, of N points in $\mathbb{R}^2$. The set can be updated by inserting or deleting points. Any solution to the problem must be able to answer three-sided range queries, i.e. given a range of the type $[x_1, x_2] \times [y, \infty]$ we report points in $\mathcal{S} \cap [x_1, x_2] \times [y, \infty]$.

This thesis presents, implements, and experiments with several solutions to the problem of three-sided range reporting. The main focus will be to provide evidence to corroborate that the three-sided range reporting problem on huge data sets is best solved using data structures optimized for external memory.

We have implemented two external memory data structures that solves the problem: The External Memory Priority Search Tree by Arge et al. [ASV99] and the External Memory *Buffered* Priority Search Tree by Brodal [Bro15].

Besides comparing these structures we also compare them against our own implementation of the Priority Search Tree of McCreight [McC85], and wrappers around MySQL, libspatialindex R*-Tree, and Boost R-Tree.

Our results show that the External Memory Buffered Priority Search Tree outperforms all other structures as soon as data exceeds internal memory capacity. MySQL without an index was significantly faster at inserting but as both the experiments with deletion and querying showed, the cost of not having an index on the table is too high. The Boost R-Tree proved to be the fastest of the internal memory data structures.

Resumé

Flere og flere hverdagsapparater bliver koblet på internettet. Dette har til formål at personliggøre og bedre kunne målrette produkter til forbrugeren. Det skaber samtidig kæmpe mængder data — større mængder end en almindelig datamat kan håndtere i det interne hukommelseslager. Vi har derfor brug for teknikker og redskaber til at analysere det genererede data. Meget data indeholder information omkring vores position i form af GPS koordinater. For at håndtere GPS koordinater skal man typisk løse komplicerede problemer indenfor algoritmisk geometri. Et sådan problem kunne eksempelvis være det tre-sidede interval rapporteringsproblem. Dette problem går ud på at vedligeholde en dynamisk mængde, $\mathcal{S}$, indeholdende N punkter i $\mathbb{R}^2$. Mængden kan modificeres ved at indsætte eller slette punkter. En løsning på dette problem skal være i stand til at svare på forespørgsler af typen $[x_1, x_2] \times [y, \infty]$, dvs. rapportere alle punkter i $\mathcal{S} \cap [x_1, x_2] \times [y, \infty]$.

Vi vil i dette speciale præsentere, implementere og eksperimentere med mange forskellige løsninger til problemet. Hovedfokuset bliver at vise hvor stor forskel, der er på strukturer optimeret til at håndtere store datamængder og strukturer der er optimeret til at kunne være i det interne hukommelseslager. Dette gøres ved at sammenligne dem eksperimentielt.

Vi har igennem specialeforløbet implementeret to strukturer optimeret til det eksterne hukommelseslager: Prioritetssøgetræet til det eksterne hukommelseslager af Arge et al. [ASV99] og det *Bufferet* Prioritetssøgetræ til det eksterne hukommelseslager af Brodal [Bro15].

Vi har derudover implementeret et Prioritetssøgetræ til det interne hukommelseslager af McCreight [McC85], og lavet omslag til MySQL, libspatialindex R*-Træ og Boost R-Træ.

Resultater fra eksperimenter viser at det *Bufferet* Prioritetssøgetræ til det eksterne hukommelseslager præsterer bedre end de resterende data strukturer så snart de skal håndtere mere data end der er plads til i datamatens interne hukommelseslager. MySQL uden et tilhørende indeks var den eneste struktur der kunne følge med når det kommer til at indsætte. Denne struktur måtte dog betale dyrt for de hurtige indsættelser i form af langsomme slettelser og forespørgsler. Det var Boost R-Træet der viste sig at være den hurtigste af strukturerne for det interne hukommelseslager.

Contents

Preface

Writing this thesis has been been a long and often frustrating process with many ups and downs. The *struggle* has been very real[1] at some points during the four and a half months. We got off to a very shaky start to say the least. Only a few days before we were supposed to begin work on our thesis we decided, based on a gut feeling, that the topic we were initially set on covering was not going to lead to a good thesis; or at least not one that we thought would encapsulate the hard work we have been putting in to our courses in the 5 years we have studied Computer Science at Aarhus University. With much haste, we decided to try to set up a meeting with Kasper Green Larsen and Gerth Stølting Brodal. Gerth referred us to Kasper and to our surprise Kasper had a project in mind. The project sounded extremely challenging but also very interesting. It would allow us to really dig into some data structures of very high complexity and implement these in `C++` while at the same time make use of our deep understanding of algorithm analysis. This would allow us to utilize all of the hard work we had put into virtually all offered algorithmic courses. We gladly accepted and only a few days before start we now had ourselves a project.

`#BetterGutFeeling`.

After having spent a few days reading through articles we had a better idea of what we were up against. The coding challenge was very daunting. We initially estimated that we would need about 80% of the total time just to implement the structures. We were not much off that estimate. We have tried to summarize the main milestones throughout the project in Figure 1, and we believe it gives a fair overview of how much effort we have put into coding. In the end of the project we had a total of 19,009 lines of `C++` code.

Even though the project has not been a walk in the park we still managed to have a lot of fun throughout. As we were approaching the phase in the project where we were going to start experimenting, the summer was also nearing and the temperatures came close to the boiling point in our small office. With five machines working overtime to finish experiments it is safe to say that our indoor environment would not live up to human rights regulations, but as a computer scientist it was cozy, and it was our home for the time being. We put ♫ Nelly with *Hot in here* on the stereo and embraced the situation.

[1] Denotes a situation where the writers wishes to express that they are encountering some sort of undesirable difficulty, but dealing with it [Dic16].

The experiments often ran for more than a week at a time and in order for us to know exactly when an experiment had finished, we had the clever idea of using the beep function of the motherboard to alert us. As a simple beep would be too boring we spent some time writing a small piece of code which, by controlling the frequency and duration of the beep sound, would play the Star Wars song: The Imperial March[2]. All was good and it worked perfectly until one weekend where we did not come in early. As we walked down the corridors approaching our office the sound of Darth Vader became increasingly ear-wrecking. The sound was extremely annoying and we really hoped that all the other tenants in the building would not be working during the weekend. As we walked down the corridors toward our office we looked through the small window next to the doors to see if anyone were noticeably annoyed, but it did not seem to bother them. Just before our office we finally glanced into the office just next to ours and in there sat a very displeased lady with large headphones clearly very annoyed by the deafening sound of The Imperial March. We are truly sorry about this.

Another perk of The Imperial March was the ability to scare the crap out of Christoffer. Peter would be able to sit comfortably at home and randomly start the cacophony which would make Christoffer jump out of his seat and generally become very anxious about when the next sound would come.

After months of intense experimenting, documentation, and LaTeX'ing we concluded with a small out-of-sight victory dance, shut down our faithful test machines, and with all our new knowledge and insight we could finally have our happily ever after.

A number of people have been part to giving this thesis its life. First and foremost we would like to thank our primary advisor, Kasper Green Larsen, for our weekly meetings, constructive feedback, enthusiasm, and engagement in the project. We would also like to thank our formal advisor, Gerth Stølting Brodal, for his great work on the data structure we have implemented and sparring when we were in doubt. In more general terms we would like to thank Aarhus University for giving us 5 years worth of great courses and preparing us for the life to come within the field of Computer Science.

We have really enjoyed working on this thesis and we sincerely hope you enjoy reading it as much as we have enjoyed writing it!

Peter Gabrielsen and Christoffer Holbæk Hansen
Aarhus, Wednesday 22nd June, 2016.

[2]You know, the song that comes on when Darth Vader is there.

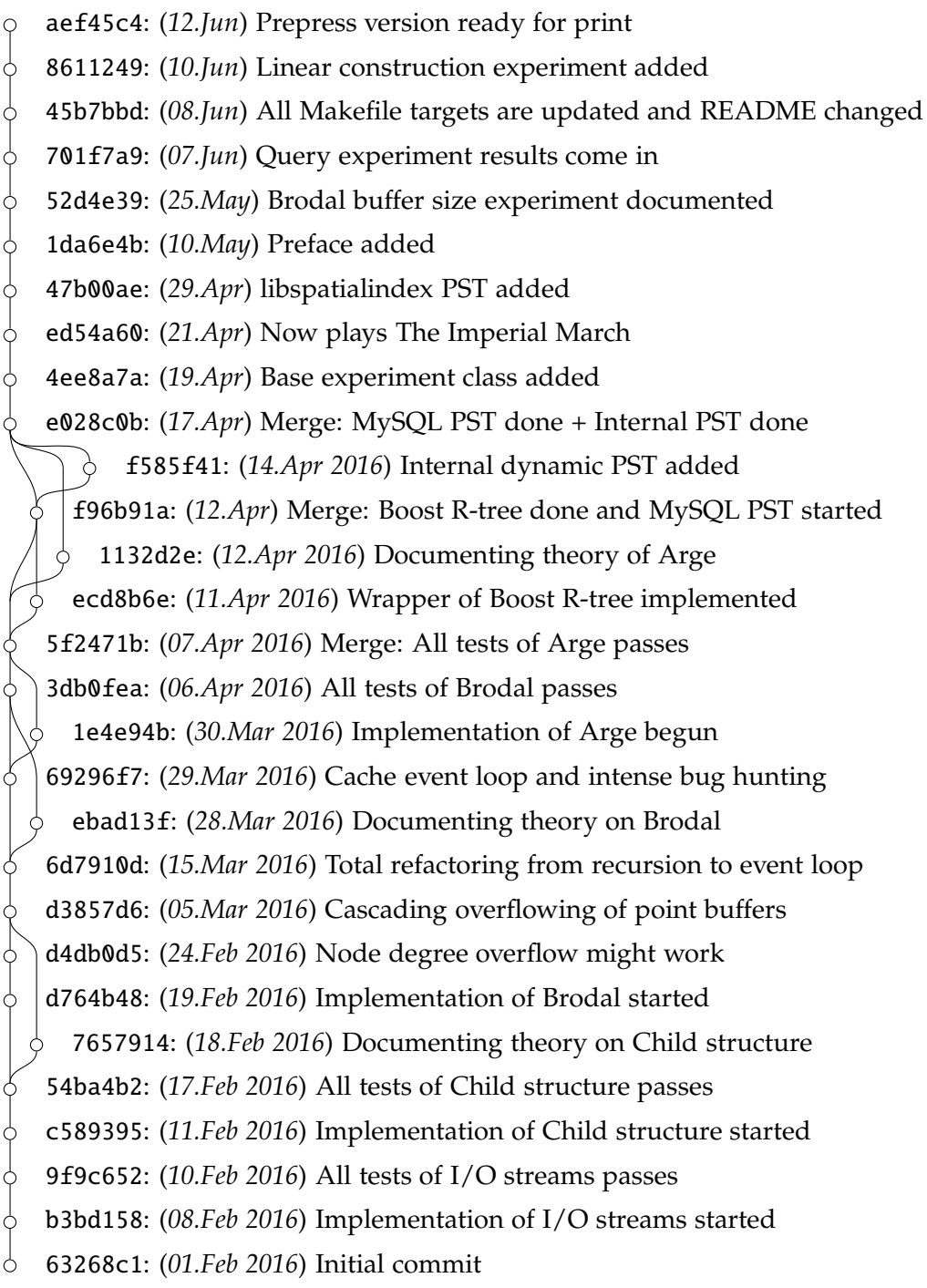

Figure 1: Excerpt of the git history for the entire project. The figure is read bottom up.

"The difference in speed between modern CPU and disk technologies is analogous to the difference in speed in sharpening a pencil using a sharpener on one's desk or by taking an airplane to the other side of the world and using a sharpener on someone else's desk."

— D. Comer

1 Introduction

In the early days of electronic computers disks were faster than processors. Since then processor technology has advanced at an incredible rate achieving annual speedups of 40 to 60 percent [RW94]. Although this is also true for disk capacity an entirely different story can be told for the speed-up of disk performance. The disparity between processor, internal memory, and external memory speeds have grown larger for each year and the gap is widening as seen in Figure 1.1. A back on the envelope analysis shows that a job that was 5% disk bound in 1999 is more than 70% disk bound on an average CPU in 2014. While the database community has always been involved in the development of practically efficient external memory data structures, most algorithm research has focused on worst-case efficient internal memory data structures [Arg05].

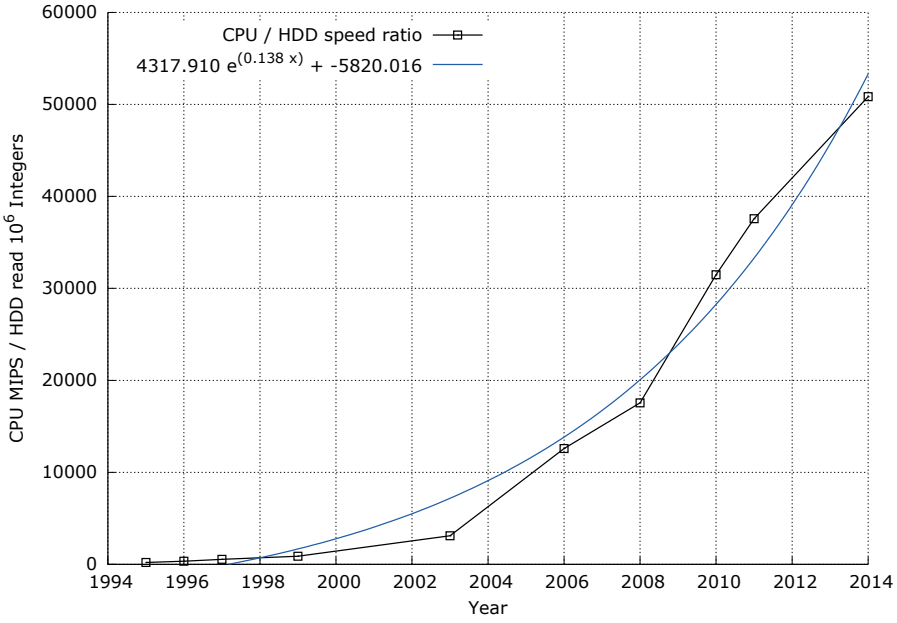

Figure 1.1: Growth of CPU and HDD speed ratio over time.
Data from `https://en.wikipedia.org/wiki/Instructions_per_second`.

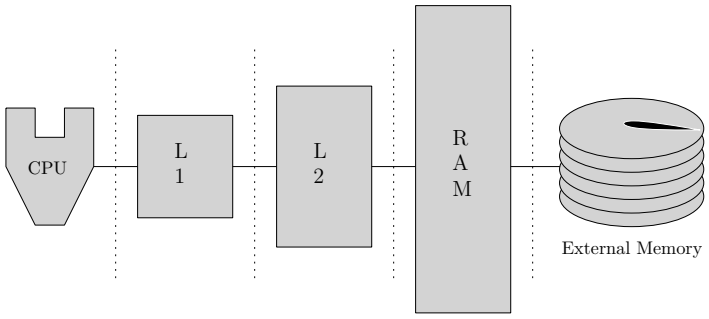

Figure 1.3: Hierarchical memory. Modern machines have complicated memory hierarchy consisting of registers in the CPU, multi-tier caches (here denoted L1 and L2), volatile internal memory and typically a mechanical or solid state disk as external memory.

With the advent of *Big Data* many industries have come to realise that adapting classic and well founded internal memory algorithms on large data sets is in many cases undesirable. They simply prove to perform much slower than the asymptotic bounds suggests. The algorithm community has found the reason to be the very same that ensured the success of the computer industry. The problem derives from the standard RAM-model of computation, where we assume an infinite memory and uniform access cost. See Figure 1.2.

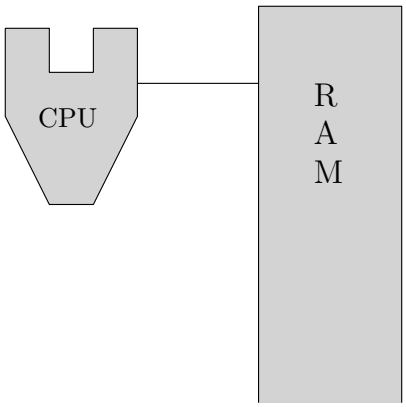

Figure 1.2: RAM-model. The standard model of computation. We assume an infinite memory with uniform access cost.

The RAM-model is as powerful as it is simple, but it ignores the more complicated memory hierarchy on modern computers. See Figure 1.3.

As we move away from the CPU the access time gets bigger. CPU registers can be accessed in a few nanoseconds. Accessing CPU caches add a small multiple to that time. Internal memory access are typically a few tens of nanoseconds. Now comes a big gap, as the time to access external memory is typically measured in milliseconds, i.e. more than 10^6 times slower than internal memory access. Also, the storage capacity increases. CPU registers are good for bytes of data, caches for a few megabytes, internal memory for gigabytes, and disks are good for terabytes of data [TG98].

Disk systems try to amortize the large access time by transferring large contiguous blocks of data and many modern operating systems utilizes sophisticated paging and pre-fetching strategies to move blocks to and from disk as needed [Tan07]. This is the main reason we still have many worst-case optimal internal memory algorithms performing well on large datasets. If the algorithms, however, relies on scattered access across data, even good

operating systems cannot take advantage of block access and we start to see severe scalability problems. It is these observations that give rise to the external memory model. The model encapsulates performance as the number of disk accesses as opposed to RAM accesses. For algorithms analysed in the I/O model it is of extreme importance to store and access data in such a way we can take advantage of data's locality in order to achieve good bounds.

In some industries, disk-based systems present too large of an obstacle, and in an attempt to close the gap they have moved towards developing internal memory big data processing algorithms. This move has been enabled by growing internal memory capacities but it comes with the price of issues such as fault-tolerance and consistency which are inherently more challenging to handle in volatile memory [ZCO+15].

Another price of the move to internal memory is the actual cost of running server farms and the cost of internal memory compared to that of external memory. The extra costs and increased complexity suggests that external memory data structures have some well defined advantages.

Along with pervasive use of computers and sensors, increased ability to acquire and store data, and the society being increasingly *data driven*, it seems that data is collected everywhere today. It is claimed in [Osb10] that the amount of generated data on a world-wide scale grew from 150 billion gigabytes to 1200 billion gigabytes from the year 2005 to 2010. This suggest that we, today, generate as much data in a single day as the entire mankind did until 2003.

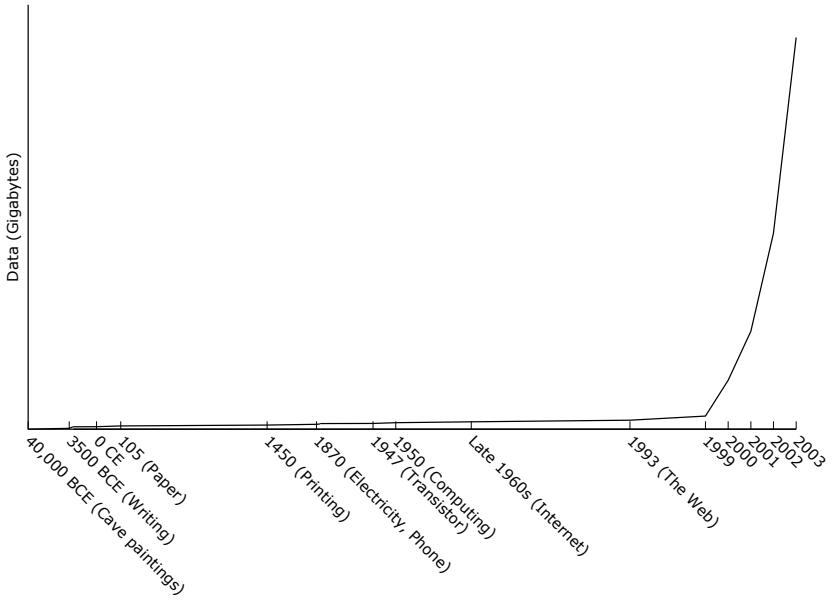

Figure 1.4: Total amount of data generated by Man over time.
Data from UC Berkeley.

An industry that has benefit severely from the continuous improvement of technology is that of processing geographical data. Such systems are also

known as geographical information systems (GIS). In the year 2000 the Shuttle Radar Topography mission set out to map Europe and North America into a 30-meter data set. Denmark alone consists of more than 46 million data points and is stored using gigabytes of data that can easily fit in a modern computer's internal memory. Today, the data set has improved to a 1/2-meter model of more than 168 billion data points. This amounts to terabytes of data that is unlikely to fit into internal memory of a standard personal computer in the coming years. Most GIS applications today use results from the field of computational geometry, and it is in this field we will focus our studies.

An integral problem of computational geometry is that of range searching. In addition to GIS applications, the problem arises in many different applications with huge data sets such as spatial databases and computer graphics. The problem can be formally described as follows. Let $\mathcal{S}$ be a set of N points in $\mathbb{R}^d$, and let $\mathcal{R}$ be a family of subsets of $\mathbb{R}^d$. Our objective is to preprocess $\mathcal{S}$ such that for a query range $r \in \mathcal{R}$, the points in $\mathcal{S} \cap r$ can be reported or counted efficiently [AE99]. The ranges can be anything from rectangles and halfspaces to balls.

In this thesis we consider the problem of maintaining a dynamic set, $\mathcal{S}$, of N points in $\mathbb{R}^2$ in external memory. The set of points can be updated by insertion and deletion. The set will be processed such that we are able to report three-sided range queries, i.e. given a range of the type $[x_1, x_2] \times [y, \infty]$, we report points in $\mathcal{S} \cap [x_1, x_2] \times [y, \infty]$.

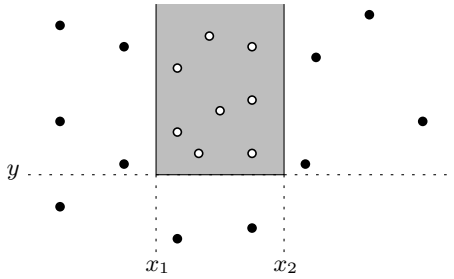

Figure 1.5: A query of the form $[x_1, x_2] \times [y, \infty]$, reporting all points in the gray area.

The thesis will present several solutions to this problem, implement, and experimentally compare solutions.

Some of the solutions will not be optimized for usage in external memory and they must pay a hefty price for accessing data not readily available in internal memory. This thesis will show the power of the I/O efficient data structures for large data sets and demonstrate what happens when internal memory data structures have to work overtime along with the virtual memory system.

1.1 Outline of thesis

This thesis is structured as follows:

In Chapter 2 we investigate the I/O model – a model of computation that encapsulates performance of the I/O bottleneck.

In Chapter 3 we give a preliminary overview of some of the techniques used in developing external memory efficient data structures.

Much work has been done on the three sided range queries and more general range queries. Some of the main ideas leading up to the main focus of this thesis is presented in Chapter 4.

Chapter 5 gives a detailed description and analysis of the Priority Search Tree for internal memory by McCreight [McC85].

Chapter 6 presents an external memory data structure for the three-sided range reporting problem by Arge et al. [ASV99] with optimal query bounds.

The main focus of this thesis, the External Memory Buffered Priority Search Tree of Brodal [Bro15] is presented in Chapter 7.

Leading up to our presentation of our experimental results in Chapter 11 we present other structures included in our experiments in Chapter 8, considerations throughout our implementation in Chapter 9, and our experimental setup in Chapter 10.

We conclude the thesis in Chapter 12 and present possible future work that could be very interesting to realise.

"All models are wrong but some are useful."

— George E. P. Box

2 Model of computation

We will argue asymptotic complexities in this thesis using the external memory model of Aggarwal and Vitter [AV88]. The external memory model (or I/O Model) measures the efficiency of an algorithm by counting the total number of reads and writes to and from disk. In detail the model consists of two levels of memory; a bounded internal memory of size M and an unbounded external memory. For a total of N records we define an I/O operation to be the process of transferring B contiguous records between the two levels of memory as depicted in Figure 2.1. We restrict all computations on records to internal memory. Throughout the thesis we will let K denote the total number of records in the output.

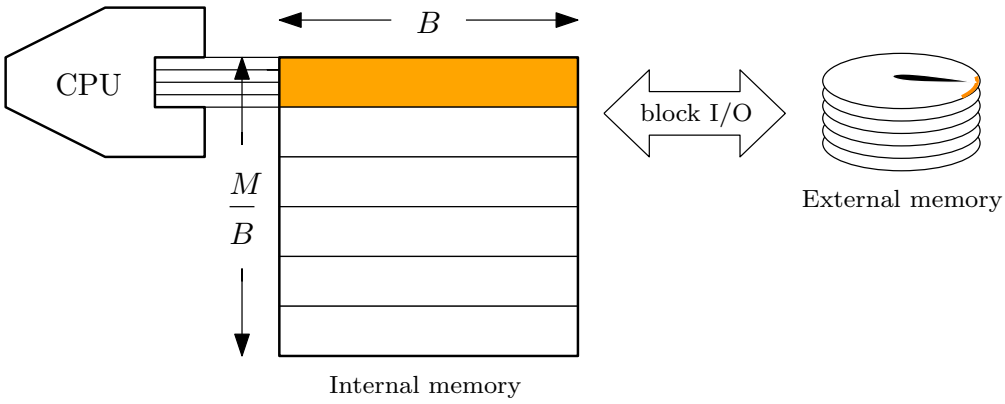

Figure 2.1: The I/O Model. Only reads and writes of contiguous blocks between internal and external memory are charged.

The fundamental bounds in the I/O Model are that we can scan N records in $\mathcal{O}(\text{Scan}) = \mathcal{O}(N/B)$ I/O's, sort N records in $\mathcal{O}(\text{Sort}) = \mathcal{O}\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ I/O's, and search for a single record between N records in $\mathcal{O}(\log_B N)$ I/O's. We denote $\mathcal{O}(N/B)$ as linear in terms of I/O's. Note that the B factor is very important as $N/B < \mathcal{O}\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right) \ll N$.

For convenience we will assume $M > B^2$. This assumption is known as the *tall-cache assumption* in the cache-oblivious model and basically states that the number of blocks M/B is larger than the size of each block B [Pro99].

"The way I see it, if you want the rainbow,
you gotta put up with the rain."

— Dolly Parton

3
Preliminaries

This chapter aims to give an overview of some of the techniques and structures used throughout the thesis. Some of the techniques are very rudimentary and may be skipped or revisited when encountered in later chapters.

3.1 Amortization

Amortization is an important algorithmic tool to argue about average performance of an operation in the worst case. In an amortized analysis, we average the time of a sequence of data structure operations. We can then show that, even though a single operation in the sequence is expensive, the average cost of an operation is small [CL09, p. 451-452].

The term was coined by Tarjan [Tar85] and describes two views of amortization. The first view is the banker's view, where we assume that a computer is running on coins. We can insert a coin and the computer will run for a fixed amount of time. An operation will pay a certain amount of coins and the goal of the analysis is then to show that all operations can be performed with the amount paid. We assume that we start without any coins, we are allowed to borrow coins, and coins can be carried over to later operations. Paying coins amounts to averaging forward over time and borrowing is the opposite.

Another view of amortization is that of the physicist. Instead of representing prepaid work as coins, the physicist represent work as potential energy, which can be released later to pay for future operations.

If we perform n operations, we will start with an initial data structure D_0. For each of the operations we let c_i be the cost of operation i and D_i be the data structure that results from that operation on the previous data structure. We define a potential function Φ to map a data structure D_i to a real number $\Phi(D_i)$. The cost of the i'th operation becomes

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

The total amortized cost becomes

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

If we can show $\Phi(D_i) \geq \Phi(D_0)$ for all i then we know that we always are able to build up enough potential in advance.

3.2 Global rebuilding

The term *global rebuilding* refers to the technique of making a (typically small) static data structure dynamic. We simply store all updates in an *update block* and once a certain threshold has been collected we rebuild the data structure [Arg05]. For data structures that does not allow the space for deleted records to be reoccupied we *mark* (or *weak delete*) the elements. Whenever αN elements have been marked, for some constant $\alpha > 0$, the entire data structure is rebuilt from scratch with only the non-marked elements. The cost of rebuilding is at most a constant factor higher than the cost of inserting αN elements, and so the amortized cost of global rebuilding is paid in advance when elements are inserted, i.e. elements are charged double such that they later can pay for being removed from the structure during a global rebuild [MSS03].

3.3 Filtering

The technique of *filtering* is used on retrieval problems where we query a certain data structure for a subset of data points. The technique is based on the fact that the complexity of the *search* and the *report* part of the algorithm should be made dependent upon each other such that we charge part of the query cost to output. In order to make filtering feasible, it is crucial that the problems specifically require the exhaustive enumeration of the objects satisfying the query.

3.4 Bootstrapping

It is often possible to develop dynamic external memory data structures by *externalizing* the equivalent internal memory data structure. In the case of trees this typically involves increasing the fanout from binary to multiway. This, however, results in problems when searching and reporting items from the tree, e.g. it might in worst case be that each subtree of a node only contributes one item to the query answer, each costing one I/O.

This problem can sometimes be solved by augmenting the data structure with several filtering substructures, i.e. smaller versions of the same problem.

This approach was first described by Arge and Vitter [AV03]. Each of the substructures typically holds $\mathcal{O}(B^2)$ elements (B elements from each subtree), and answers queries in $\mathcal{O}\left(\log_B B^2 + {}^K/_B\right) = \mathcal{O}\left(1 + {}^K/_B\right)$, where K is the size of the output. It can even be a static structure if it can be constructed in $\mathcal{O}(B)$ I/O's, since B updates can be stored in a separate buffer and applied using a global rebuild in amortized $\mathcal{O}(1)$ I/O's per update [Vit08].

3.5 B-Tree

The B-Tree of Bayer and McCreight [BM72] is to external memory what the balanced binary search tree is to internal memory. It supports insertions and deletions of points, and searching in $\mathcal{O}(h)$ where h is the height of the tree. The height of the tree depends on the branching parameter, i.e. the maximum number of children a node can have. This parameter typically depends on the characteristics of the disk used and the problem at hand. This gives the following definition of a B-Tree:

Definition 1 *$\mathcal{T}$ is a B-Tree with branching parameter b if*

- *All leafs have the same depth.*
- *All nodes store at most $b - 1$ elements.*
- *All nodes and leafs except for the root have degree between $\frac{1}{2}b$ and b.*
- *The root has degree between 2 and b.*
- *Elements are stored in non-decreasing order in the nodes.*
- *The keys of node x, $x.key_i$, separate the children's elements into ranges such that if k_i is a key stored in child c_i then $k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \cdots$*

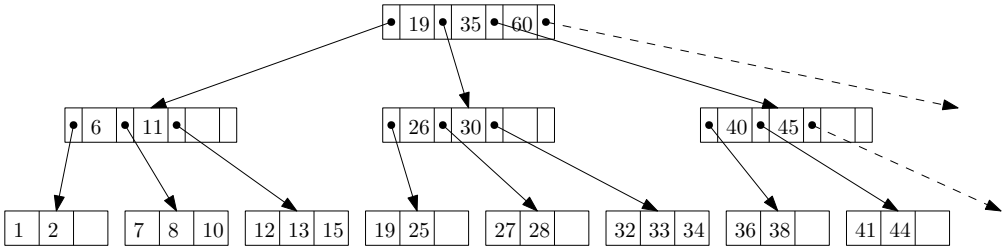

Figure 3.1: A B-tree with $b = 4$.

It follows from the definition that if $b = \Theta(B)$ then a B-Tree will have height $\mathcal{O}(\log_B N)$. Note that the B-Tree is a special case of the (a, b)-Tree in which the number of elements in leafs is a uniquely defined parameter.

Searching in a B-Tree is very much similar to searching in a binary search tree. Instead of making a binary decision at each node we instead have to make a multiway branching decision. If the element we are searching for is not contained in the current node, we find the smallest i such that the key we

are searching for is less than $x.key_i$. We then recursively search for the key in child c_i. It will in the worst case require $\mathcal{O}(\log_B N)$ I/O's to search for an element residing in a leaf.

Inserting in a B-Tree is not as simple as inserting into a binary search tree. Similarly to binary trees we search for the leaf node to insert the key, but we cannot simply create a new node for the key. Instead we insert the key into the found leaf, and if the leaf now contains too many elements we split the leaf into two each containing half the elements of the original leaf. The median of the elements are inserted in the parent. See Figure 3.2. Splitting a leaf might cause its parent to have too many children which causes the parent to similarly split. Searching for the leaf node to insert into takes $\mathcal{O}(\log_B N)$ and so does recursively splitting nodes from a leaf to root path as a split operation requires $\mathcal{O}(1)$ I/O's.

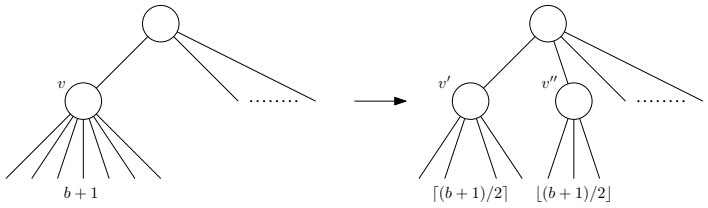

Figure 3.2: Splitting a degree $b + 1$ node v (or leaf with $b + 1$ elements) into nodes (or leafs) v' and v''.

Deleting in a B-Tree introduces an opposite to splitting, fusing. To delete a key from the tree we search the tree for the key, which now can reside in an internal node x. We then delete the key from the node x, which might cause x to have too few elements. To remedy this situation we will have to potentially fuse x with a neighbouring node. If x together with either its predecessor or successor contains less than b elements we can fuse the two nodes. See Figure 3.3.

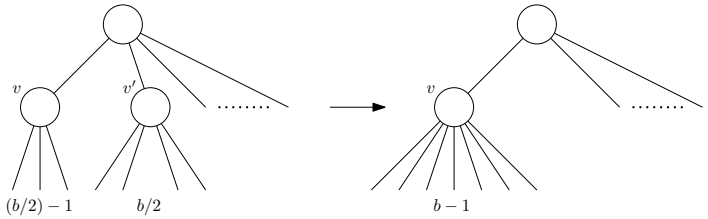

Figure 3.3: Fusing a degree $(b/2) - 1$ node v (or leaf with $(b/2) - 1$ elements) with sibling v'.

If this is not the case then we know that we are able to steal an element from a neighbouring node in order to satisfy the properties of the B-Tree. As in the case of insertion, fusing nodes might recursively cause the parent to fuse with one of its neighbours. Fusing two nodes require $\mathcal{O}(1)$ I/O's but a fuse can cascade on a leaf to root path causing $\mathcal{O}(\log_B N)$ I/O's.

A B-Tree on N elements are stored in $\mathcal{O}(N/B)$ blocks and can be constructed in the sorting bound by building the tree level-by-level bottom-up.

3.5.1 Weight-balanced B-Tree

A weight balanced B-Tree and a regular B-Tree differ with regards to how and when splitting and fusing of nodes take place. Constraints are imposed on the *weight* of a node rather than the number of children. The weight of a node v is the number of elements stored in the subtree rooted at v. Let a be the branching parameter and k the leaf parameter of the tree. An internal node on level l has weight between $\frac{1}{2}a^l k$ and $2a^l k$ and has at least one child. Inserting in a weight balanced B-Tree is similar to a normal B-Tree and when a leaf splits it might cause the weight of the parent to become too large and recursively split on a path from a leaf to the root.

The strength of the weight-balanced B-Tree is the crucial property described in the following lemma.

Lemma 1 *After a split of a node v_l on level l into two nodes v'_l and v''_l, at least $\frac{1}{2}a^l k$ inserts have to pass through v'_l (or v''_l) to make it split again. After a new root r in a tree containing N items is created, at least $3N$ inserts have to be done before r splits again.*

The lemma, simply put, states that a node v will not underflow or overflow unless $\Omega(weight(v))$ elements have been inserted or deleted in the subtree of v. A proof of the lemma can be found in [AV96].

3.6 Buffer Tree

The Buffer Tree of Arge [Arg95] combines the basic B-Tree described in Section 3.5 with a *buffer-technique*. The result is an external data structure supporting batched operations efficiently in terms of I/O's. The ideas introduced by Arge has proven especially useful when generalizing well-known internal-memory algorithms into efficient I/O algorithms. The main idea of the buffer-technique is to introduce *laziness* in the update algorithms and utilize internal memory to process a large number of updates simultaneously. For example, when inserting a point we do not search all the way down the tree to find the leaf. Instead, the point is inserted into a buffer of the root. Whenever the size of a buffer exceeds a certain threshold we push elements from the buffer one level down to buffers on the next level of the tree. This process of emptying full buffers is repeated recursively down the tree.

Formally the basic Buffer Tree is defined according to Definition 2. Refer to Figure 3.4 for an illustration of a Buffer Tree.

Definition 2 *A basic Buffer Tree is*

- *A B-Tree with leaf parameter B.*
- *All internal nodes, except for the root, have degree between $\frac{1}{4}\,{}^{M}/_{B}$ and ${}^{M}/_{B}$.*
- *The root has degree between 2 and ${}^{M}/_{B}$.*
- *Each internal node has a buffer of size M.*

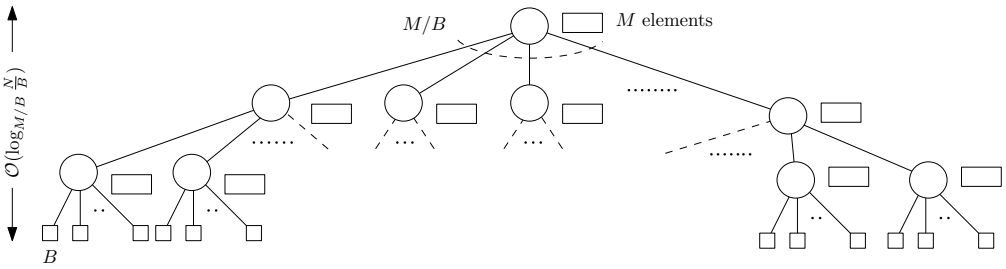

Figure 3.4: Buffer Tree.

Updates are handled by augmenting the element in question with information on whether we are inserting or deleting the element. Since an element can be represented in multiple buffers, we also augment elements with a time stamp. Whenever we have collected B elements we insert all of them into the root buffer of size M. Whenever the buffer overflows, i.e. have more than M points, we initiate a buffer-emptying process that distributes all elements in the buffers to the children.

For an *internal* node that does not have leafs as children this process is done as follows. First, we load the M unsorted elements into internal memory and sort them. Then we scan through the sorted updates while removing matching inserts and deletes with respect to the time stamps. Now we simply distribute the remaining elements one level down using a single scan. We

make sure to distribute the elements in sorted order, as this will guarantee that we leave no buffer of a child with more than M unsorted elements followed by a list of sorted elements. Thus, we are able to sort the resulting buffer in a linear number of I/O's as depicted in Figure 3.5.

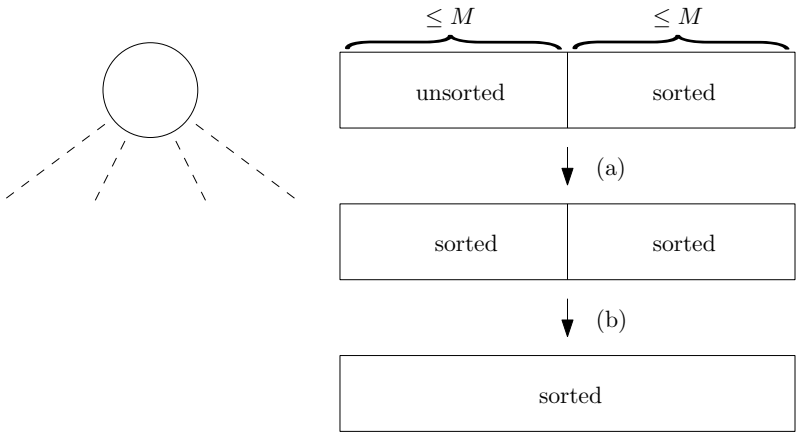

Figure 3.5: (a) First, the $\leq M$ unsorted points are loaded into internal memory in $\mathcal{O}(M/B)$ I/O's and are then sorted in internal memory (b) Then, the two lists of sorted points are merged in $\mathcal{O}(M/B)$ I/O's.

We then recursively empty full child buffers provided that the children are internal nodes that do not have leafs as children. Only when we have emptied buffers of all overflowing internal nodes which do not have leafs as children, we proceed the buffer-emptying process to leaf nodes. The reason is that a buffer-emptying process on a leaf may result in the need for rebalancing. By only emptying leafs after all internal node buffer-emptying processes have been performed we prevent rebalancing and buffer-emptying processes from interfering with each other.

We empty all relevant leaf buffers one-by-one while maintaining the *leaf-emptying invariant* that all buffers of nodes on the path from the root to a leaf with a full buffer are empty. Since we handle all internal nodes before emptying the leafs this invariant is true when we handle the buffer-emptying of the first leaf. To empty a node u with K elements in the leafs, we start by sorting the buffer and remove matching inserts and deletes. Then, we merge the buffer elements with the K leafs below, again removing matching inserts and deletes. The resulting set of K' sorted elements now needs to replace the K original leafs along with new routing elements of u reflecting the changes. If we end up with a resulting set of size $K' < K$, i.e. we do not have enough elements to fill the K leafs, we introduce $K - K'$ dummy elements and insert those in the remaining leafs. If we have $K' \geq K$ we place K elements in the leafs. The remaining elements (if any) are finally placed one-by-one while ensuring rebalancing when necessary. See Figure 3.6.

We can rebalance as in a normal B-Tree using splits as depicted in Figure 3.7, since the leaf emptying invariant ensures that all nodes from u to the root have empty buffers.

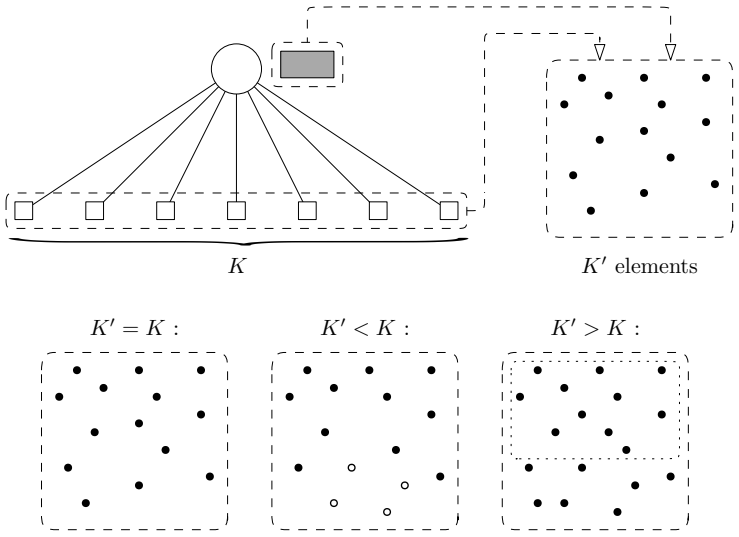

Figure 3.6: Buffer emptying process. First the K original leafs and the buffer are merged, and matching inserts and deletes are removed. This gives a set of size K′. If $K = K'$ then the original leafs are replaced with the K′ new ones. If $K' < K$ then we add dummy elements, here represented as circles, to the set such that we replace all of the original K leafs with elements from the new set. If $K' > K$ then we use a subset of size K to replace the original leafs, and the rest of the points are then inserted one by one.

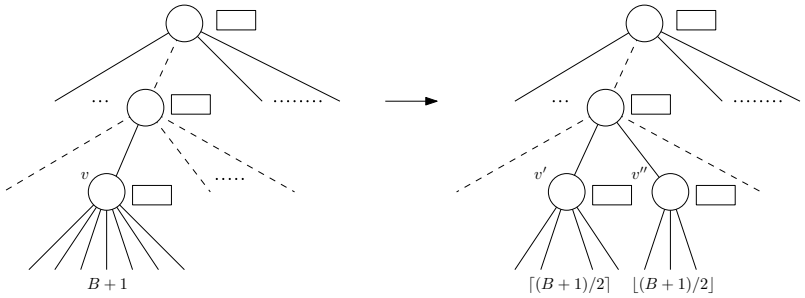

Figure 3.7: Split in a Buffer Tree. The leaf-emptying invariant guarantees that there are empty buffers on the root to leaf path ensuring splits can be done as in a normal B-Tree.

After we have emptied all leaf buffers we remove the place-holder elements one-by-one. The leaf-emptying invariant ensures that a node v on the path from u to the root has an empty buffer, but $v's$ sibling may not have an empty buffer. Therefore we cannot fuse in a normal B-Tree manner. Instead, we perform a buffer-emptying process on $v's$ immediate sibling before performing the actual fuse. The emptying of the buffer of a sibling node v' can result in buffers running full. We empty all such full non-leaf buffers before performing the actual fuse on v. See Figure 3.8.

The place-holder elements ensures we are always only in the process of handling a rebalancing caused by a single delete.

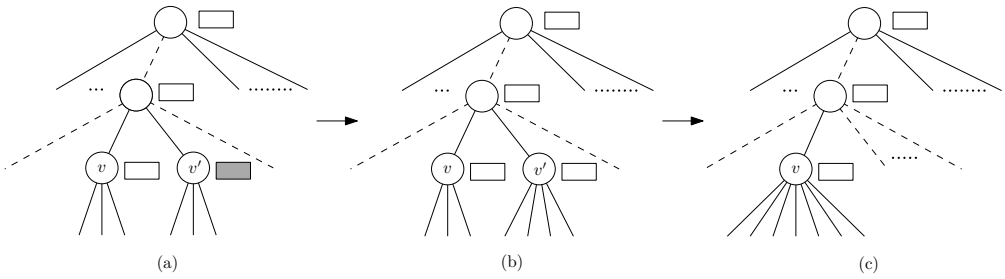

Figure 3.8: Buffer tree fusing. (a) node v already have an empty buffer guaranteed by the leaf-emptying invariant, but no such guarantee is given for the sibling node v', so we have to start a buffer-emptying process on v' (b) The buffer emptying of v' might cause a leaf split which is handled before we fuse v and v' (c) After the buffer-emptying process of v' and rebalancing has finished we perform the actual fuse of nodes. This is done as in a normal B-Tree.

3.6.1 Analysis

To empty an internal node buffer of size $X \geq M$ we need $\mathcal{O}(X/B)$ to scan the elements and $\mathcal{O}(M/B)$ to distribute them one level down. In order to empty a leaf node we have to scan the $\Theta(M)$ elements below it which gives an additional $\mathcal{O}(X/B + M/B)$ I/O's.

By letting the branching parameter equal M/B and the leaf parameter equal B, we can push all elements in a buffer of size M down to the next level in $\mathcal{O}(M/B)$ I/O's. This follows from the fact that all the elements fit into internal memory and we use $\mathcal{O}(1)$ I/O's to push one block one level down. Disregarding rebalancing of the tree, we can argue that we touch each block of elements a constant number of times on each of the $\mathcal{O}\left(\log_{M/B} \frac{N}{B}\right)$ levels. Thus, inserting N elements can be done in an optimal $\mathcal{O}\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/O's assuming no rebalancing of the tree.

It is showed in [Arg05] that an N-element B-Tree with branching parameter b and leaf parameter $k = \Omega(B)$ has an amortized number of internal node rebalancing operations (split/fuse) needed after an update equal to $\mathcal{O}\left(\frac{1}{b \cdot k} \log_b \frac{N}{B}\right)$ I/O's. It follows directly that the total number of internal node rebalancing operations performed during N updates is $\mathcal{O}\left(\frac{N}{b \cdot M/B} \log_{M/B} \frac{N}{B}\right)$. Since each operation takes $\mathcal{O}(M/B)$ I/O's to empty a non-empty buffer, the total cost of the rebalancing is also $\mathcal{O}\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$.

We conclude that the total cost of a sequence of N update operations on an initially empty Buffer Tree is $\mathcal{O}\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ amortized.

"Nanos gigantum humeris insidentes."

— Bernard de Chartres

4
Related work

McCreight introduced the Priority Search Tree for internal memory [McC85]. The Priority Search Tree is basically a combination of a binary search tree on the x-coordinate and a heap on the y-coordinate, where the root of every subtree stores the maximum y-value in that subtree and points are distributed according to the median x-coordinate. This allows updates in $\mathcal{O}(\log N)$ time and three-sided range queries in $\mathcal{O}(\log N + K)$ time, where K is the number of points reported. The Internal Memory Priority Search Tree is explained in greater detail in Chapter 5.

The study of adapting the Priority Search Tree to external memory was initiated by Icking et al. [IKO88]. They achieve a static external Priority Search Tree that uses $\mathcal{O}(N/B)$ space and answers three-sided range queries in $\mathcal{O}(\log_B N + K/B)$ I/O's. The data structure uses a blocked B-Tree with pointers to full buckets of data points. The idea is depicted in Figure 4.1. In order to make the data structure dynamic the underlying B-Tree is replaced with a Red-Black tree. This change of underlying search tree results in a solution that answers queries in $\mathcal{O}(\log_2 N + K/B)$ I/O's and handles updates in $\mathcal{O}(B \log_2 N)$ I/O's.

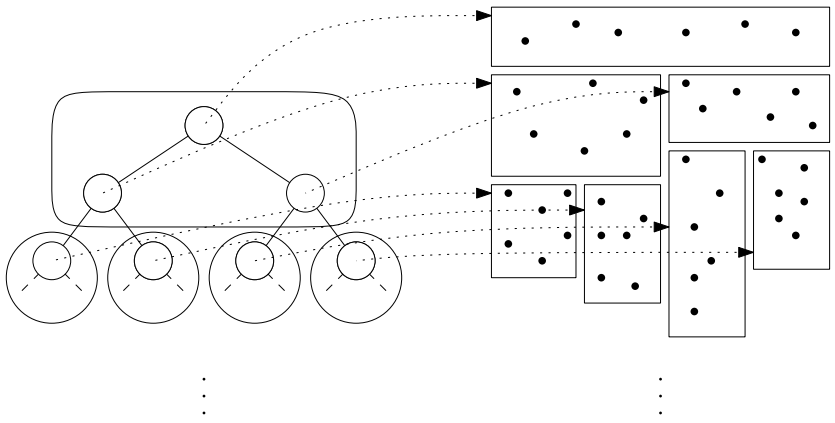

Figure 4.1: Illustration of the solution of Icking et al. Blocked B-Tree with pointers to full buckets of data points.

Table 4.1: Previous dynamic external-memory three-sided range reporting data structures. All query bounds except for [SR95] are optimal. Amortized bounds are marked †, and ε is satisfying $0 < \varepsilon \leq 1$. All data structures require $\mathcal{O}(N/B)$ space, except for [RS94] requiring space $\mathcal{O}(N/B \log_2 B \log\log B)$. $\mathcal{IL}^*(x)$ denotes the number of times $\log^*$ must be applied before the result becomes ≤ 2

Reference	Update	Query	Construction
[RS94]	$\mathcal{O}\left(\log N \log B\right)^\dagger$	$\mathcal{O}\left(\log_B N + K/B\right)$	
[SR95]	$\mathcal{O}\left(\log_B N + (\log_B N)^2/B\right)^\dagger$	$\mathcal{O}\left(\log_B N + K/B + \mathcal{IL}^*(B)\right)$	
[Arg95]	$\mathcal{O}\left(\log_B N\right)$	$\mathcal{O}\left(\log_B N + K/B\right)$	
[Bro15]	$\mathcal{O}\left(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N\right)^\dagger$	$\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N + K/B\right)^\dagger$	$\mathcal{O}\left(\text{Sort}(N)\right)$

Kanellakis et al. presents a linear space and partially dynamic solution in [KRVV96]. The data structure answers three-sided queries in $\mathcal{O}(\log_B N + K/B + \log_2 B)$ I/O's and supports inserts in $\mathcal{O}(\log_B N + \log_B^2 N/B)$ I/O's. The result is fairly involved and is unlikely to perform well in any practical manner. Please refer to Appendix A for a presentation of the overall ideas of their solution.

Ramaswamy and Subramanian presents a suboptimal space data structure that answers three-sided queries with an optimal query bound in [RS94]. They use the same basic blocked B-Tree with pointers to full buckets of data points as introduced by Icking et al. [IKO88] and illustrated in Figure 4.1. In addition they introduce the idea of *path caching*. Please refer to Appendix B for a presentation of the main ideas of their solution.

Ramaswamy and Subramanian continues their work and brings down the space usage in [SR95]. This is achieved by constructing a search tree that divides the points into smaller regions and using a slightly modified caching scheme. Further details of the main ideas can be found in Appendix B.1.

Arge et al. presented the first linear space dynamic data structure with optimal query bounds and suboptimal update bounds in [ASV99]. The data structure supports queries using $\mathcal{O}(\log_B N + K/B)$ I/O's and updates using $\mathcal{O}(\log_B N)$ I/O's. Please refer to Chapter 6 for a detailed description of the solution.

Brodal [Bro15] introduced an amortized solution that improves the update bound of [ASV99] by a factor $\varepsilon B^{1-\varepsilon}$ by adding ε^{-1} to the query bound. This gives a data structure supporting updates in amortized $\mathcal{O}\left(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N\right)$ and three-sided range queries in amortized $\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N + K/B\right)$ for $0 < \varepsilon \leq 1$. The data structure adapts ideas of the Buffer Tree described in Section 3.6 to the External Memory Priority Search Tree of Arge [ASV99]. The solution is presented in detail in Chapter 7.

Please refer to Table 4.1 for a summary of the results.

4.1 Lower bounds

Solving the problem of three-sided range queries is very closely related to that of the 1D-dictionary problem. The 1D-dictionary problem asks us to

maintain a dynamic set of keys such that we can answer membership queries, i.e. whether or not a key is contained in the set. We can reduce the three-sided range reporting problem to the 1D-dictionary by restricting elements to be of the form (x, x) and test membership by a query of the form $[x, x] \times [-\infty, \infty]$. By reduction we must have that the lower bounds of the 1D-dictionary problem also applies to that of the three-sided range queries.

The 1D-dictionary problem has been a popular topic in the case of internal memory. It has been proved by a simply adversary argument that a query can be forced to cost $\log_2 N$ comparisons no matter the cost of updates, and more generally it has been proved that if an insertion performs at most $\mathcal{O}(k)$ comparisons then queries can be forced to cost at least $\max\left\{\log_2 N, N/2^{\Theta(k)}\right\}$ comparisons [BGLY81].

There has also been much work on the lower bounds of the 1D-dictionary problem in external memory. We here give an adversary argument which shows that for any comparison based dictionary storing N elements, there exists a query requiring at least $\log_B \frac{N}{M} - \mathcal{O}(1)$ I/O's, i.e. a lower bound for queries in external memory dictionaries. The argument goes like this:

Assume we are at a position in our dictionary where the elements that can still be equal to our query are denoted *candidate elements*. These elements form a consecutive subsequence in the partial ordering of the N elements in the dictionary. Initially we can have at most M elements in internal memory. The adversary will now select a partial ordering of these M elements, i.e. select answers to each comparison between these M elements, such that there are at least $\frac{N-M}{M+1} > \frac{N}{M+1} - 1$ candidate elements left. Each I/O will bring in B elements. If we have k candidate elements before this I/O then the adversary will choose a partial ordering such that there are at least $\frac{k-B}{B+1} > \frac{k}{B+1} - 1$ candidate elements left. An argument by induction will show that after i I/O's there are at least $\frac{N}{(M+1)(B+1)^i} - 2$ candidate elements left. As a consistent answer to the membership query cannot be given before we have only one candidate element left we must have that $\frac{N}{(M+1)(B+1)^i} - 2 \leq 1 \Rightarrow i = \log_{B+1} \frac{N}{M} - \mathcal{O}(1)$.

As mentioned in Section 3.5, the B-Tree is the external memory version of a binary search tree. The query bounds of the B-Tree, $\mathcal{O}(\log_B N + K/B)$, are optimal, i.e. equal to the lower bound, but this is not the case for the update bounds.

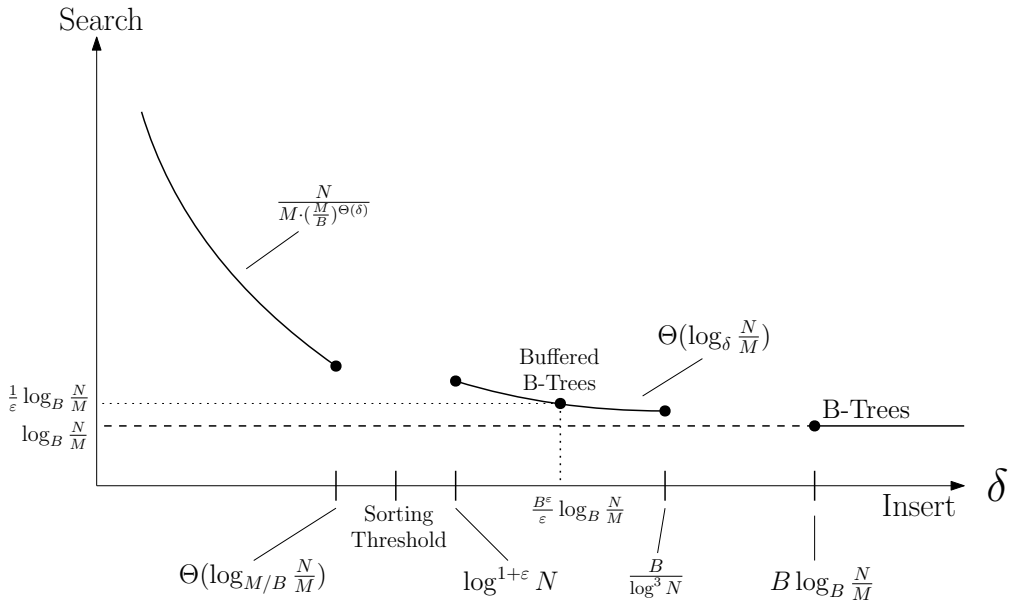

Figure 4.2: A summary of the results of Brodal and Fagerberg [BF03]. It depicts the trade-off between insert and search/query. On one end we can achieve really fast insert operations but pay the price with slow queries. In the other end we have the B-Tree which query bound matches the optimal but has suboptimal insertions. No results exists for the gaps.

Brodal and Fagerberg [BF03] studied two lower bound trade-offs between the I/O complexity of membership queries and updates. They arrive at the following theorem:

Theorem 1 *If N insertions perform at most $\delta \cdot N/B$ I/O's, for $1 \leq \delta \leq B \log_B N$ then*

1. *There exists a query requiring at least $\log_{B+1} \frac{N}{M} - \mathcal{O}(1)$ I/O's.*
2. *There exists a query requiring $N/\left(M \cdot \left(\frac{M}{B}\right)^{\mathcal{O}(\delta)}\right)$ I/O's for $N > M$.*
3. *There exists a query requiring $\Omega\left(\log_{\delta \log^2 N} \frac{N}{M}\right)$ I/O's for $N > M$.*

The first is essentially just the result proved by the above adversary argument saying that B-Trees have an optimal query bound. This result is summarized in Figure 4.2.

The results by Brodal and Fagerberg assume a comparison based model and that keys are indivisible. Iacono and Pătraşcu [IP12] looks at what happens when we remove this assumption and are thus allowed to use hashing. They improve the update time of the Buffer Tree by roughly a logarithmic factor. More precisely they arrive at the following theorem:

Theorem 2 *For any $\max\{\log\log N, \log_M N\} \leq \lambda \leq B$, we can solve the dictionary problem by a Las Vegas data structure with update time $t_u = \mathcal{O}(\frac{\lambda}{B})$ and query time $t_q = \mathcal{O}(\log_\lambda N)$ with high probability.*

This means that in one end of the trade-off they can for $\lambda = B^{\varepsilon}$ obtain an update time of $\mathcal{O}(1/B^{1-\varepsilon})$ and query time of $\mathcal{O}(\log_B N)$. This matches the bounds of the Buffer Tree of Arge.

If we instead set out to obtain fast updates, they are able to achieve an update bound very close to the optimal disk transfer rate of $1/B$ namely they obtain $t_u^{\min} = \mathcal{O}\left(\frac{1}{B} \cdot \max\left\{\log\log N, \log_M N\right\}\right)$ but at the cost of a query time of $t_q^{\max} = \mathcal{O}\left(\log_{\max\left\{\log\log N, \log_M N\right\}} N\right)$.

These results suggest that there are still many possibilities to improve external memory structures if we abandon the comparison and indivisibility paradigms.

"All my best thoughts were stolen by the ancients."
— Ralph Waldo Emerson

5
Internal Memory Priority Search Tree

In this chapter we present an internal memory data structure for the three-sided range reporting problem. The data structure was originally presented by McCreight [McC85] and is denoted a Priority Search Tree. The Priority Search Tree can be constructed in linear time and is a combination of a binary search tree on the x-coordinate and a heap on the y-coordinate. A formal definition of a Priority Search Tree on a set of N points, P, is as follows. We assume that all points have distinct coordinates, though this assumption can be removed by using the normal lexicographical ordering of points.

- If $P = \varnothing$ then the Priority Search Tree is an empty leaf.
- Otherwise, let $p_{\max}$ be the point in the set P with the largest y-coordinate.

  Let x_{mid} be the median of the x-coordinate of the remaining points.

  Now let

  $$P_{\text{below}} := \{p \in P \setminus \{p_{\max}\} : p_x \leq x_{\text{mid}}\}$$

  $$P_{\text{above}} := \{p \in P \setminus \{p_{\max}\} : p_x > x_{\text{mid}}\}$$

  The Priority Search Tree consists of a root node v where the point $p(v) := p_{\max}$ and the value $x(v) := x_{\text{mid}}$ 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}

What is important to note is that the specific construction method allows the data structure to be indexed in two different ways. First, the tree can be searched as a binary search tree based on the x-coordinate. Second, the tree operates as a max-heap based on the y-coordinate. Please refer to Figure 5.1 for an illustration of a Priority Search Tree constructed on points $P = \{A, B, C, D, E, F, G, H\}$.

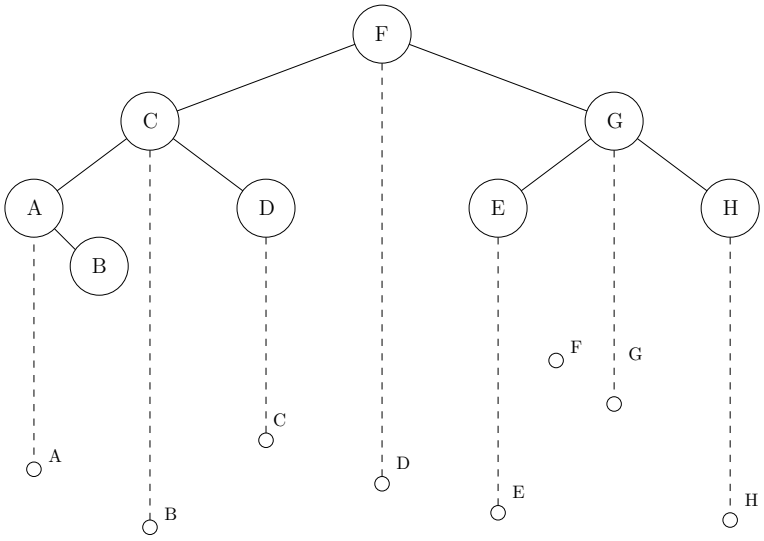

Figure 5.1: The letter given in the node represents the y_{max} stored as a key, while the dashed line represents the median x value stored in the node.

5.1 Three-sided range query

When answering a query of the form $[x_1, x_2] \times [y, \infty]$ we begin at the root of the tree and first check the y-coordinate of this node against the query-y. As long as the y-coordinate of the current node is greater than the query-y we will continue down the tree. We can use the fact that the tree is a basic binary search tree on the x-coordinate to only visit the part of the tree that is within our search range. Please refer to Figure 5.2 for an illustration of the general query pattern.

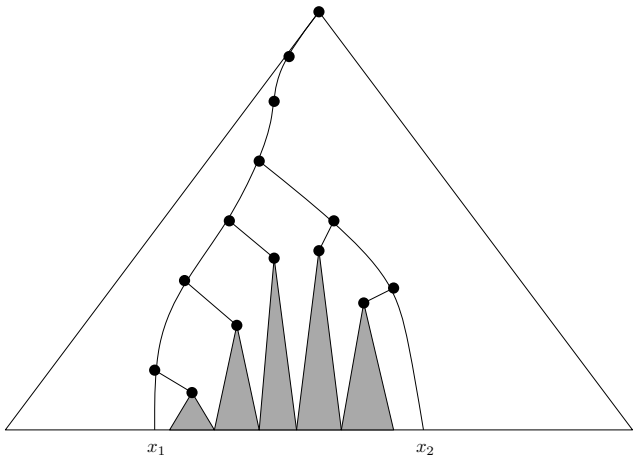

Figure 5.2: The shaded subtrees in the figure store only points with x-coordinate within the correct range. This property ensures we can search the subtrees based on y-coordinate only.

Analysis

The search operation follows two root to leaf paths each of length $\mathcal{O}(\log N)$. Using the search paths and heap property of the tree we are guaranteed to visit only nodes containing points that are reported. This gives a total running time of $\mathcal{O}(\log N + K)$.

5.2 Dynamic Priority Search Tree

The key difference between the static solution presented by McCreight and a dynamic solution is that we always ensure that each point is placed in exactly one leaf and the order of the leafs from left to right corresponds to the order of the x-coordinate of the points. An internal node stores the point with greatest y-coordinate in its subtree that is not already stored by an ancestor. Whenever we store a point in an interior node, then the leaf node which corresponds to this point is considered a *place-holder*. Please refer to Figure 5.3 for an illustration of a dynamic Priority Search Tree over the points $P = \{A, B, C, D, E, F, G\}$.

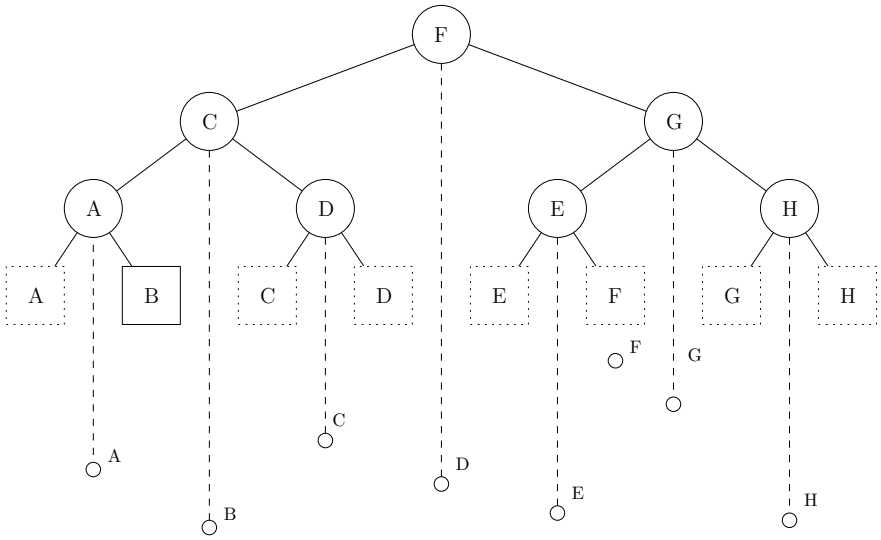

Figure 5.3: 8 data points and the corresponding dynamic Priority Search Tree. Dotted leafs are place-holders for a key higher in the tree.

5.3 Construction

Assuming we are given a set of x-sorted points we can construct a balanced dynamic Priority Search Tree using a bottom-up construction method similar to the bottom-up construction of a heap. In the first phase of construction we associate each point with a place-holder in the Priority Search Tree. These place-holders will become the leaf level of the data structure. Next

we select neighbouring pairs of place-holders and compare them to one another in terms of their y-coordinate. We denote the point with the highest y-coordinate as the *winning* point and the comparison between points as a *tournament round*. It is the winning point that will be represented by a new internal node at one level higher in the tree. Please refer to Figure 5.4 for an illustration of the construction of the leaf level. At the next level of the tree, we perform the same comparisons as before to determine which nodes will advance to the third level. At most $N/2$ points are compared at this level.

Every tournament round will leave an empty interior node behind as the wining point is moved one level up. We thus have to check if any previous tournament losers are now eligible to be represented higher in the tree.

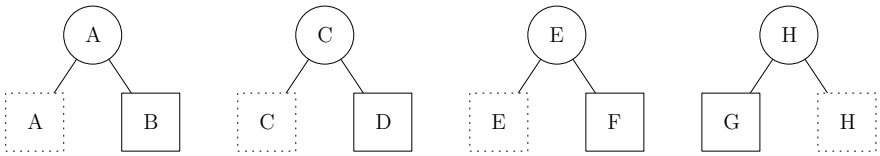

Figure 5.4: The first phase of the bottom-up construction "tournament".

Analysis

First, all points are inserted as leafs in the bottom layer of the tree. At a layer above $N/2^1$ nodes are created and the tournaments are played one level down. For the i'th level we create $N/2^i$ nodes and we play tournaments i levels down. The total steps to build the dynamic Priority Search Tree of size N is thus:

$$\sum_{i=0}^{\log(N)} \frac{N}{2^i} i = N \cdot \sum_{i=0}^{\log(N)} i \left(\frac{1}{2}\right)^i \leq N \cdot \sum_{i=0}^{\infty} i \left(\frac{1}{2}\right)^i$$

The solution to the last summation can be found by taking the derivative of both sides of the well known geometric series:

$$\frac{\partial}{\partial x}\left(\sum_{i=0}^{\infty} x^i\right) = \frac{\partial}{\partial x}\left(\frac{1}{1-x}\right) \Rightarrow \sum_{i=1}^{\infty} i x^i = \frac{x}{(1-x)^2}$$

For $x = \frac{1}{2}$ we get

$$\frac{1/2}{(1-1/2)^2} = 2$$

Plugging this in to the above sum we get that the total number of steps to build a dynamic Priority Search Tree on N points is $\mathcal{O}(N)$.

This implies that we can construct the balanced dynamic Priority Search Tree in linear time assuming we are given a sorted input. The tree is balanced since we place all leafs at the same level. If the size of the input is not a perfect power of 2, the tree will be unbalanced by a single level.

5.4 Insertion

In order to insert a point into the data structure, we add a leaf (place-holder) for the point and perform a push-down operation starting at the root.

We can determine where to add the place-holder as the dynamic Priority Search Tree is a binary search tree on the x-coordinate of the points. When we reach an existing leaf, we add a new internal node in place of this, and make the existing leaf one of the children of the new internal node. Then we add a new leaf to the tree as the other child, and store the new point in this leaf. See Figure 5.5.

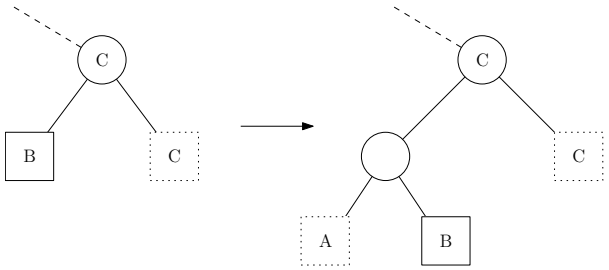

Figure 5.5: First step of the insertion algorithm. A place-holder is created for the point to be inserted and an internal node is created as its parent. Here point A is inserted.

In order to maintain the heap order of the Priority Search Tree we now perform a push-down operation, where we at each level compare the y-coordinate of the point to be inserted with the y-coordinate of the point represented by the given internal node. If the y-coordinate of the point to be inserted is less than that of the point stored in the internal node, then we push the point to be inserted further down the tree. Otherwise, we store the point to be inserted in this internal node, and take the point which was formerly represented by this internal node and continue the push-down operation with this point instead. See Figure 5.6.

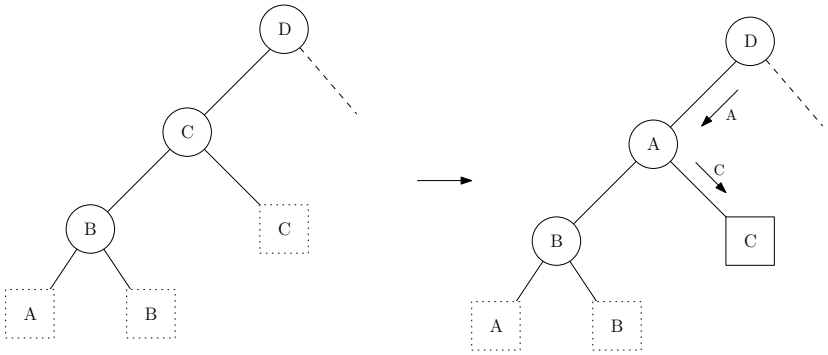

Figure 5.6: Second step of the insertion algorithm. The push-down operation of point A starts at the root. Here point A has smaller y-coordinate than point D. Point A has larger y-coordinate than point C and so it occupies the internal node and pushes point C further down the tree to its place-holder.

Analysis

The first step of the insertion algorithm is a binary search on the x-coordinate of the new point. The path is of length $\mathcal{O}(\log N)$. Adding a new internal node in place of the old leaf takes a constant amount of operations. The push-down operation follows a single root to leaf path of length $\mathcal{O}(\log N)$ and uses a constant amount of work in each node. We conclude insertion of a point can be done in $\mathcal{O}(\log N)$.

5.5 Deletion

When deleting a point we must locate the node representing the point we wish to delete. After we have removed the point from the dynamic search tree, we must replay a portion of the tournament among the points below this node in order to replace it. Finally we must delete the leaf which is the place-holder for the point.

Analysis

Locating the node representing the point to be deleted can be done in $\mathcal{O}(\log N)$ using the binary search property. Once we have located the interior node we can remove this point in $\mathcal{O}(1)$. The deletion of the point of a node leaves a hole in the tree that we fill by replaying tournaments following a node to leaf path of length $\mathcal{O}(\log N)$. We conclude the deletion algorithm requires $\mathcal{O}(\log N)$ per deletion.

5.6 Rebalancing

If we use the operations as stated above we could end up with a highly unbalanced tree. We fix this using global rebuilding when a linear number of updates have been performed. We can collect all points in sorted order in linear time by visiting leafs from left to right using an in-order tree walk. On the collected points we now use the linear construction algorithm to rebalance the tree. Using this strategy yields a data structure that handles updates in $\mathcal{O}(\log N)$ amortized. By using a Red-Black tree as the heart of the tree we can achieve a data structure that is $\mathcal{O}(\log N)$ worst case by performing rotations to rebalance the tree.

5.7 Bounds in the I/O model

The above bounds translate directly to the I/O model as we cannot guarantee that nodes on the search path are perfectly placed in blocks, which in the worst case means that each visit to a node will equal 1 I/O.

"Truth, like gold, is to be obtained not by its growth,
but by washing away from it all that is not gold."

— Leo Tolstoy

6
External Memory Priority Search Tree

In this chapter we present a result on dynamic three-sided range reporting due to Arge et al. [ASV99]. The result is a weight-balanced B-Tree where each node is augmented with a bootstrapped structure for storing the top $\Theta(B^2)$ points w.r.t. the y-value of the subtree rooted at that node. The bootstrapped structure is described in Section 6.1 and the main data structure of Arge et al. that proves Theorem 3 is described in Section 6.2.

Theorem 3 *An external memory data structure exists supporting insertion and deletion of points in amortized $\mathcal{O}(\log_B N/B)$ I/O's and reporting of three sided range queries in $\mathcal{O}(\log_B N/B + K/B)$ I/O's, where N is the input size and K is the size of the output. The structure uses $\mathcal{O}(N/B)$ space.*

6.1 Dynamic 3-sided queries on $\Theta\left(B^2\right)$ points

In this section we describe a data structure that supports the operations stated in Theorem 4.

Theorem 4 *There exists a dynamic data structure for storing $\mathcal{O}(B^{1+\varepsilon})$ two dimensional points for $0 \leq \varepsilon \leq 1$. Insertion and deletion of s points requires amortized $\mathcal{O}\left(1 + \frac{s}{B^{1-\varepsilon}}\right)$ I/O's. The data structure supports reporting of all points inside a query range of the form $[x_1, x_2] \times [y, \infty]$ in $\mathcal{O}(1 + K/B)$ I/O's. The structure uses linear space. Finally the structure can be constructed using $\mathcal{O}(B^{1+\varepsilon}/B)$ I/O's given an x-sorted set of $B^{1+\varepsilon}$ points.*

The structure consists of a static structure $\mathcal{L}$ storing $\mathcal{O}(B^{1+\varepsilon})$ points and two buffers $\mathcal{I}$ and $\mathcal{D}$ storing at most B points each. The buffers $\mathcal{I}$ and $\mathcal{D}$ store delayed insertions and deletions, respectively, and are initially empty. A point can appear in either $\mathcal{I}$ or $\mathcal{D}$ but not both as updates from either cancel each other out.

Let L be the points stored in $\mathcal{L}$ and let $\ell = \lceil |L|/B \rceil$. When $\mathcal{L}$ is fully constructed it will consists of $2\ell - 1$ blocks of B points in each block. The points in L are first partitioned into blocks $b_1, \ldots, b_\ell$ sorted by x-value. The last block may have size less than B.

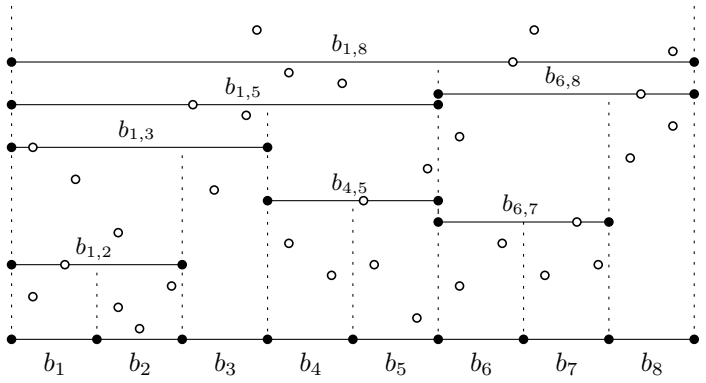

Figure 6.1: The structure for $B = 4$. The points are represented by circles. The sweep line has merged blocks b_1 and b_2 at the point where the blocks contain 4 points on or above the line. This is represented by a line segment with black endpoints and the $b_{1,2}$ label. The same goes for the other merged blocks created.

To construct blocks $b_{\ell+1}, \ldots, b_{2\ell}$ we make a vertical sweep over the points in increasing y-order. When the sweep line reaches a point in a block b_i that together with an adjacent block, i.e. either b_{i-1} or b_{i+1}, contains exactly B points on or above the sweep line, we replace the two blocks by a single block containing the B points on or above the sweep line. The merged block is denoted $b_{i,j}$ if it contains points from the initial blocks in the range from, and including, i to j. The two merged blocks are then excluded from the sweep and the newly created merged block is included in the continued sweep. Every merge of adjacent blocks causes the sweep line to intersect one block less resulting in at most $\ell - 1$ blocks created from the sweep.

A catalogue structure stores in $\mathcal{O}(1)$ disk blocks a reference to each of the $2\ell - 1$ blocks. For block b_i we store the minimum and maximum x-values for the points contained in the block. For a merged block $b_{i,j}$ we store the interval $[i, j]$ and the minimum y-value of the points in the block. Note, this minimum y-value is also the point where the sweep line created the block $b_{i,j}$.

Insertions and deletions are stored in $\mathcal{I}$ and $\mathcal{D}$ respectively. When a point is inserted in $\mathcal{I}$ or $\mathcal{D}$ we make sure to remove any existing occurrence of the point in $\mathcal{I}$ and $\mathcal{D}$ such that the new update overwrites any previous updates. Whenever $\mathcal{I}$ or $\mathcal{D}$ overflows, i.e. $|\mathcal{I}| > B$ or $|\mathcal{D}| > B$, the stored updates are applied to the set of points in $\mathcal{L}$. This is done by scanning L in increasing x-order while applying insertions and deletions, i.e. for each point in L we check whether we should insert a new point from $\mathcal{I}$ before it or if the point should be deleted. This process results in a new set of points L' which once again is partitioned into blocks $b_1, \ldots, b_{\ell'}$ and a vertical sweep similar to the previously described is performed to rebuild the merged blocks and catalogue structure. This reconstruction is done in $\mathcal{O}(\ell')$ I/O's. Since $\ell' \leq \lceil (|L|+1)/B \rceil$ it requires $\mathcal{O}(\lceil |L|/B \rceil) = \mathcal{O}(B^\varepsilon)$ I/O's to rebuild $\mathcal{L}$. If we amortize this cost over the $> B$ updates that caused the overflow the cost becomes $\mathcal{O}\left(B^\varepsilon/B\right) = \mathcal{O}\left(1/B^{1-\varepsilon}\right)$ amortized I/O's per delayed update.

Queries are of the form $[x_1, x_2] \times [y, \infty]$ and can be answered by scanning

the catalogue to find the blocks intersected by the sweep when it was at y. This corresponds directly to the t line segments immediately below the line segment imposed by the bottom of the query range. These blocks will contain a superset of the points contained in our query.

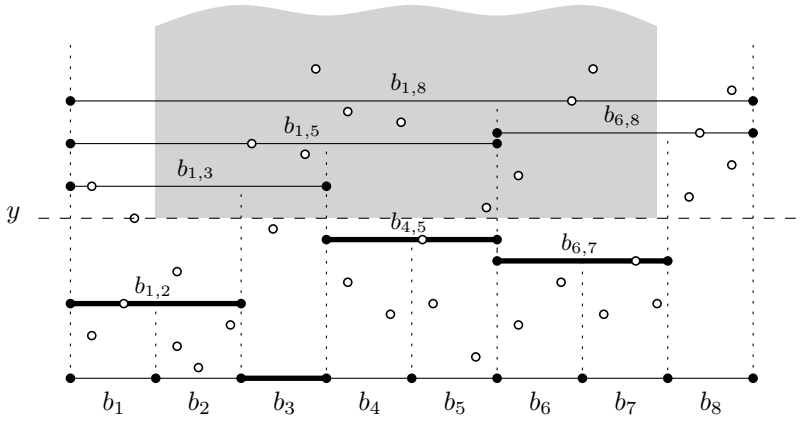

Figure 6.2: The gray area is our query and we should report the points within. This is done by finding the fat line segments which is the segments just below the sweep line at y. The segments can be found using the catalogue. Now the blocks can be scanned and relevant points can be reported in $\mathcal{O}(1 + K/B)$.

We know from construction that the blocks intersected contains B points on or above the sweep line. The left most and right most of these blocks are not necessarily fully contained in the query range and do not necessarily contain any points to report. We know that blocks must contain at least $B\lfloor (t-2)/2 \rfloor$ points since two adjacent blocks in the query range at the sweep line would otherwise have been merged to a single block containing just B points, i.e. if we force merge all adjacent blocks two and two we would end up with $(t-2)/2$ blocks each with at least B points on or above the sweep line. It follows that the output is at least $K \geq B\lfloor (t-2)/2 \rfloor$.

The t relevant blocks are scanned and the points contained in the query are reported. The total number of I/O's required becomes $\mathcal{O}(1+t) = \mathcal{O}(1 + K/B)$ as $t \leq 2\frac{K}{B} - 2$ from the previous observation.

We have now showed that we are able to construct a dynamic data structure with the bounds stated in Theorem 4.

■

6.2 Main structure

As mentioned earlier, the main structure is a weight-balanced B-Tree [AV96] on the normal lexicographical ordering of points w.r.t the x-coordinate. Each internal node of the structure stores an instance of the bootstrapped structure described above for answering three-sided queries on $\Theta(B^2)$ points. Arge et al. denotes this structure the *query data structure* as it allows for fast queries which will be explained later. Points are stored in the query data structure according to the following rules.

- An internal node stores at most B^2 points in the associated query data structure.
- For a child w of internal node v the Y-set of w denoted $Y(w)$ is the points of the query data structure of v that is associated with the range of w. See Figure 6.3.
- An internal node stores at most B points for each child of the node, i.e. for all children w of an internal node v we have that the size of $Y(w)$ is at most B.
- A leaf stores at most $2k$ points in its query data structure where k is the leaf parameter of the B-Tree.
- If a node or leaf v stores points in its query data structure then $Y(v)$ in $parent(v)$ must contain at least $B/2$ points.

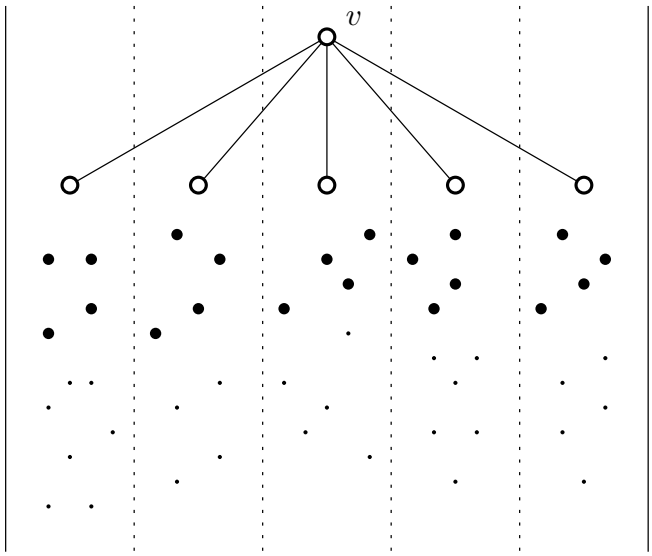

Figure 6.3: An internal node v of the base B-tree. For each child w of v, the Y-set $Y(w)$ consists of the $\Theta(B)$ highest points stored in the subtree of v that are within the x-range of w. The Y-sets of the five children of v are indicated by bold points. They are stored collectively in the query structure of v.

The base B-Tree uses linear space and since each point is stored only once in a query data structure that also uses linear space, we can conclude that the structure stores N points in $\mathcal{O}(N/B)$ blocks.

6.3 Updates

6.3.1 Insertion

Inserting a point in the structure involves two steps. The first is to insert the point in the base B-Tree. This is done as described in Section 3.5 and may result in nodes splitting which in turn might result in the splitting of query data structures. Let v be a node in the tree that has just been split into v' and v'' as depicted in Figure 6.4.

As a result $Y(v')$ and $Y(v'')$ may contain fewer than $B/2$ points. This is remedied by promoting points of v' (resp. v'') into $Y(v')$ (resp. $Y(v'')$). Promoting a point from v' to $parent(v')$ is done by finding the top-most point p' stored in the query data structure of v' in $\mathcal{O}(1)$ I/O's using the block structure of $Q_{v'}$. The points found are as shown in Figure 6.5. Now, p' is deleted from $Q_{v'}$ and inserted into $Q_{parent(v')}$. This process might cause one of the Y sets of the children to become too small and we thus need to recursively promote a point. This recursion might in the worst case be on a path from v down to a leaf. The process is called *bubble-up*.

After inserting in the base tree and appropriate reorganization we need to insert the point in the correct query data structure. The search starts in the root. The child w responsible for the x-range of the point is found and its Y-set is found by a degenerate query on the form $[x\text{-range of } w] \times [-\infty, \infty]$ on the query data structure. If the number of points is $\geq B/2$ and the point is below all of them then the point is recursively inserted into the found child. Otherwise the point joins the query data structure of the root. If the Y-set of the found child is now too large we recursively insert the lowest of these points into the child's query data structure. If we reach a leaf we simply insert the point into the query data structure of that leaf. See Figure 6.6.

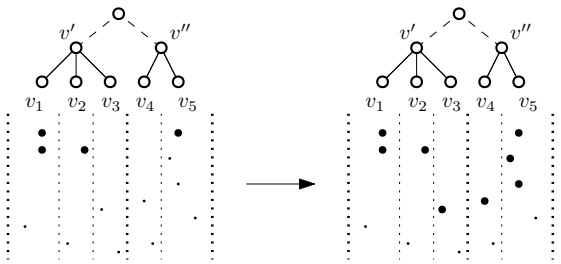

Figure 6.5: Too small Y-sets are remedied by promoting the topmost points from the children.

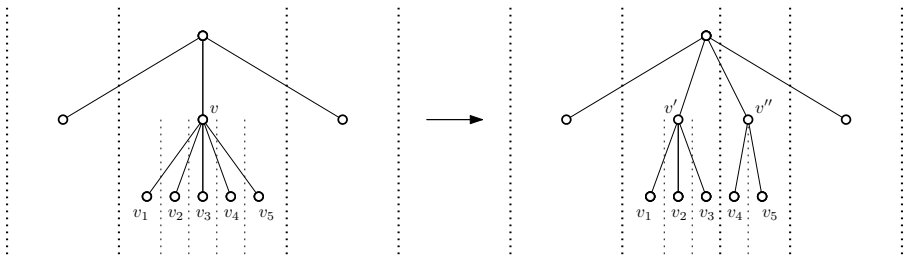

Figure 6.4: v is split into v' and v''. As a result $Y(v')$ and $Y(v'')$ may contain fewer than $B/2$ points.

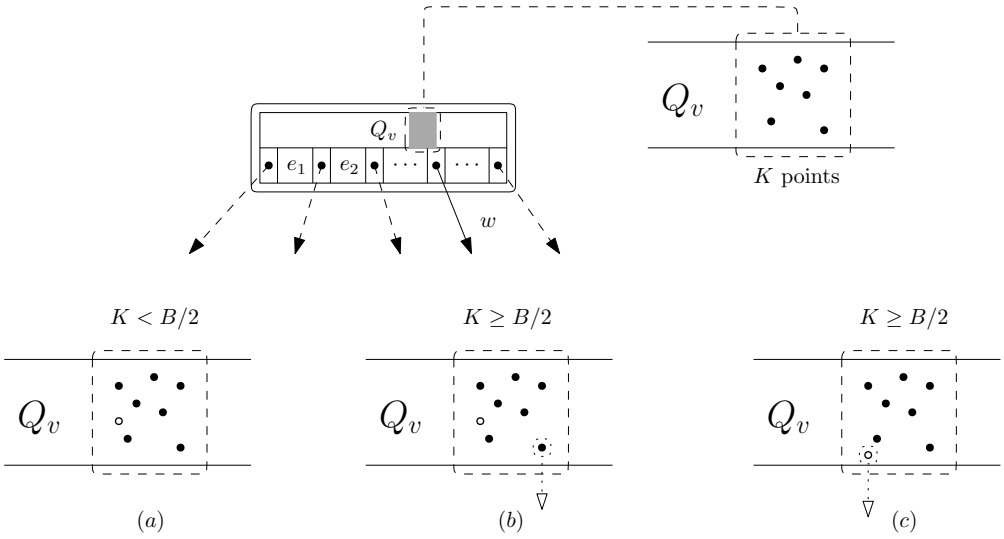

Figure 6.6: Bubble down. First we compute the Y-set of the found child w by a degenerate query on the form [x-range of w] × [−∞, ∞] resulting in a set of points with size K (a) If $K < B/2$ then we simply insert the point (here marked as a circle) (b) If $K \geq B/2$ and the point is larger than the smallest y-value then the point is inserted and the smallest y-value is recursively sent down (c) If $K \geq B/2$ and the point is smaller than the smallest y-value then the point itself is sent recursively down.

6.3.2 Deletion

Rebalancing the tree after handling a delete is done by global rebuilding instead of using fusion of nodes. To delete a point we search down in the base tree for the point and mark it as deleted without actually removing the point. The next step is to remove the point from the query data structure that it resides in. This is done similar to finding the query data structure to insert the point into. The Y-set is recursively found on the search path of the point and if the Y-set contains the point, then the point is removed. If the Y-set becomes too small as a result we perform a *bubble-up* operation.

6.3.3 Analysis

Inserting in the base tree can be done in $\mathcal{O}(\log_B N)$ I/O's by Section 3.5 and can cause as many splits on the path from a leaf to the root. Each split might cause $B/2$ *bubble-up* operations. Each *bubble-up* at v costs $\mathcal{O}(1)$ I/O's and might recurse all the way to a leaf for a total of $\mathcal{O}(\log_B weight(v))$ where $weight(v)$ is the size of the subtree rooted at v. In the worst case $B/2$ of these operations are performed totalling $\mathcal{O}(B \log_B weight(v)) = \mathcal{O}(weight(v))$ I/O's.

It follows from Lemma 1 in Subsection 3.5.1 that the cost of splitting a node can be amortized over the insertions and thus each of the $\mathcal{O}(\log_B N)$ splits cost $\mathcal{O}(1)$ I/O's amortized.

Deleting can be done in $\mathcal{O}(\log_B N)$ I/O's as it is just a search for the point in the base tree and query data structure.

Rebalancing of the tree is done using global rebuilding. After $\Theta(N)$ delete operations the tree is rebuilt using $\mathcal{O}(N \log_B N)$ I/O's which is paid for by double charging the $\Theta(N)$ delete operations.

6.4 Query

Querying the data structure with $Q = [x_1, x_2] \times [y, \infty]$ consists of two steps. The first step is to identify which nodes to visit and the second consists of reporting points in Q from the query data structures of the identified nodes. We identify which nodes to visit by searching on a path from the root to leaf along paths corresponding to x_1 and x_2 and by visiting nodes in between the two paths. As the tree is a search tree on the x-coordinate we know that nodes in between the search paths for x_1 and x_2 will be in the query range of Q. We will only proceed to visit a child of v if we report all points from the query data structure of v in the Y-set of that child, with the exception of the leftmost and rightmost paths which are always visited all the way to a leaf. We report all points in Q, since, by the rules, a point in Q cannot be in an unvisited subtree as this would have been visited if all points were reported and no points in the subtree has lower y-values.

6.4.1 Analysis

In every internal node v visited we spend $\mathcal{O}(1 + K_v/B)$ I/O's. There are $\mathcal{O}(\log_B N)$ nodes on the search paths from root to the leftmost and rightmost leaf and thus the number of I/O's used on these paths is $\mathcal{O}(\log_B N + K/B)$. All other internal nodes visited are visited because all points were reported in the parent. If we do not report all points from a Y-set we can charge the $\mathcal{O}(1)$ I/O's of visiting the child to the parent which must have reported $\Theta(B)$ points. As the cost of reporting from the query data structures is $\mathcal{O}(1 + K/B)$ the total cost amounts to $\mathcal{O}(\log_B N + K/B)$.

"Daring ideas are like chessmen moved forward: they may be beaten, but they may start a winning game."

— Johann Wolfgang von Goethe

7 External Memory Buffered Priority Search Tree

In this chapter we present an external memory data structure introduced by Brodal [Bro15]. The structure supports updates in amortized $\mathcal{O}\left(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N\right)$ I/O's, three sided range queries in $\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N + K/B\right)$ I/O's for $0 < \varepsilon \leq 1$, and can be constructed on N sorted points in $\mathcal{O}(N/B)$ I/O's. The parameter ε determines the size of the fanout and in turn the size of a bootstrapped substructure for storing $\mathcal{O}(B^{1+\varepsilon})$ points in every internal node. The substructure is very similar to that of Arge et al. [ASV99, Section 3.1] for handling $\Theta(B^2)$ points with the main difference being that we reduce the capacity to allow an amortized constant number of I/O's per update. The bootstrapped structure is in [ASV99] used to store the top $\Theta(B^2)$ points w.r.t. the y value for the subtree rooted at the given node. This structure uses it in a slightly different way to store the top $\mathcal{O}(B^{1+\varepsilon})$ points of the children of the given node. The bootstrapped structure is described further in Section 6.1 and will be referred to as the *child structure* in the rest of the chapter.

The External Memory Buffered Priority Search Tree is a combination of the External Memory Priority Search Tree of Arge et al. [ASV99] described in Chapter 6, and the buffered updates of the Buffer Tree described in Section 3.6 also thanks to Arge [Arg95]. The main data structure is described in Section 7.1.

7.1 Main data structure

This section presents the main data structure achieving the results introduced in Theorem 5.

Theorem 5 *An external memory data structure exists supporting insertion and deletion of points in amortized $\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N)$ I/O's and three sided range queries in amortized $\mathcal{O}(\frac{1}{\varepsilon} \log_B N + K/B)$, where ε is a constant, $0 < \varepsilon \leq 1$, N is the number of points in the structure, and K is the size of the output. The structure can be constructed in amortized $\mathcal{O}(N/B)$ I/O's on an x-sorted set of points and stored in $\mathcal{O}(N/B)$ blocks.*

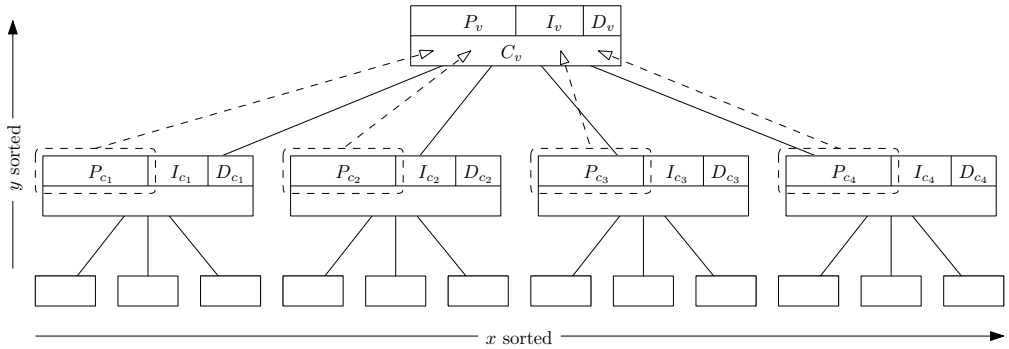

Figure 7.1: External Memory Priority Search Tree with buffers. The points stored in the point buffers P_{c_i} of the children are also stored in the child structure C_v of the parent. This allows for fast queries.

The structure is a slightly modified version of the B-Tree over the x-values of the points in the tree. Each internal node, except for the root, has a degree between $\Delta/2$ and Δ, with $\Delta = \lceil B^\varepsilon \rceil$. The root has degree between 2 and Δ.

Each node v stores three buffers containing $\mathcal{O}(B)$ points each, namely a point buffer P_v, an insertion buffer I_v, and a deletion buffer D_v which purpose will be described shortly.

As in the Internal Memory Priority Search Tree of McCreight [McC85] described in Chapter 5, the points with highest y-value resides in the top of the tree, i.e. we have a heap ordering among the nodes of the tree on the y-value. This means that for a child c of v, there are no points in the point buffer P_c of the child with larger y-value than any y-value in the point buffer P_v of v.

The buffers I_v and D_v stores delayed insertions and deletions on their way down to a point buffer of a descendant. Using the basic ideas of the Buffer Tree of Arge [Arg95] described in Section 3.6, buffers are handled recursively whenever an invariant is broken. We will introduce the invariants in Section 7.1.1.

For each internal node v we also store an instance of the child structure C_v containing a copy of all points stored in the point buffers P_c of every child c of v. See Figure 7.1.

Finally, for each internal node, v, we store, in $\mathcal{O}(1)$ blocks, information about the minimum y-value of the points in the point buffers of each of v's children. If a child is empty we will mark this by storing ∞ as the minimum y-value.

All information at the root is kept in internal memory except for the child structure.

7.1.1 Invariants

For a node v the following invariants must be true:

- P_v, I_v, and D_v are disjoint and points in the buffers have x-values spanned by the subtree rooted at v.

- All points in $I_v \cup D_v$ have y-value less than the points in P_v.
- An update in a buffer at v is more recent than updates in descendants of v, and thus, should overwrite any updates in descendants of v.
- A leaf in the tree has empty insertion and deletion buffer and the size of its point buffer is less than $B/2$.
- An internal node in the tree has $B/2 \leq |P_v| \leq B$, $|D_v| \leq B/4$, and $|I_v| \leq B$.

7.1.2 Updates

We update the structure with insertions and deletions by adding points to the root's insertion or deletion buffer respectively, while maintaining the above invariants. During an update the insertion or deletion buffer might overflow, i.e. get larger than B or $B/4$ respectively. This is handled in the following five steps: (i) handle overflowing deletion buffers (ii) handle overflowing insertion buffers (iii) split leafs with overflowing point buffers (iv) split nodes of degree $\Delta + 1$ (v) fill underflowing point buffers.

We will in the following look at each step individually and argue their complexity.

(i) A deletion buffer at node v overflows when $|D_v| > B/4$. As the structure is a B-Tree on the lexicographically ordering of points, we must have by the pigeon-hole principle, that there exists a child c such that we can push $\mathcal{U} \subseteq D_v$ of $\lceil |D_v|/\Delta \rceil$ deletions to c. This is illustrated in Figure 7.2. Points in $\mathcal{U}$ are removed from D_v, I_c, D_c, P_c, and C_v. Any point p in $\mathcal{U}$ lexicographically larger than the minimum point in P_c (w.r.t. y) is removed from $\mathcal{U}$ as the deletion cannot cancel any updates further down in the tree. See Figure 7.3.

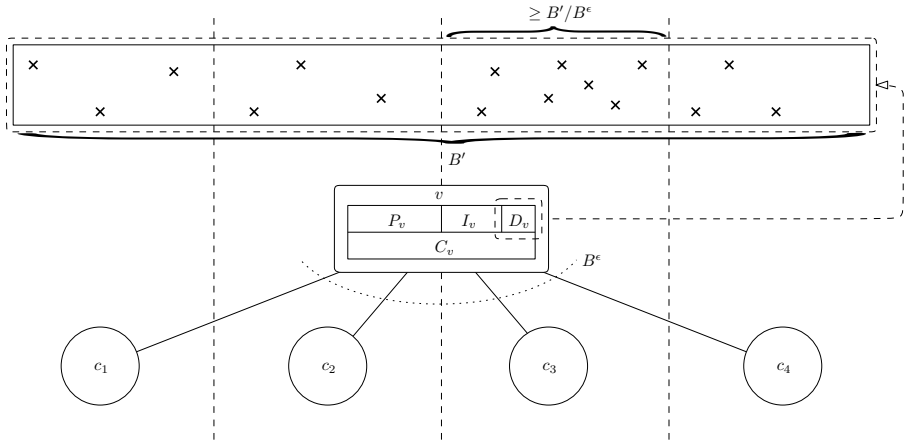

Figure 7.2: Pigeon hole principle. Each cross represents a deletion stored in the deletion buffer of v. Here $B' = B/4 = 16$ and $\varepsilon = 1/3$. This gives $B^\varepsilon = 4$. By the pigeon hole principle there must exist an x-range containing at least B'/B^ε points that can be sent down to the child responsible for storing these points. Here the child c_3 will receive a subset of the deletions.

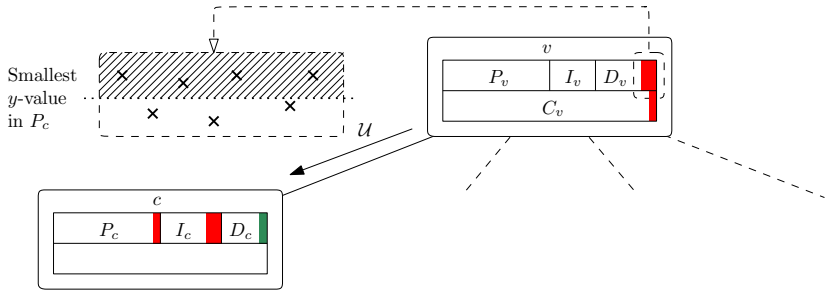

Figure 7.3: Deletion buffer overflow. A total of $\lceil |D_v|/B^\varepsilon \rceil$ deletions are moved from D_v to $\mathcal{U}$. Deletions larger than the smallest y-value in P_c are removed from $\mathcal{U}$ since they cannot cancel points further down due to the heap order of the tree. Finally all points in $\mathcal{U}$ are removed from P_c, I_c, D_c, and C_v before $\mathcal{U}$ is inserted into D_c.

If v is a leaf we are done. If not, the remaining points in $\mathcal{U}$ are inserted in D_c which might recursively overflow. In the worst case we might recursively overflow along a path from the root to a leaf each time causing $\mathcal{O}(\lceil B/\Delta \rceil)$ deletes to be pushed one level down. Updating C_v with $\mathcal{O}(\lceil B/\Delta \rceil)$ updates takes amortized $\mathcal{O}(1 + (B/\Delta)/B^{1-\varepsilon}) = \mathcal{O}(1)$ I/O's.

(ii) An insertion buffer at v overflows when $|I_v| > B$. Similar to handling a deletion buffer overflow we find a child c such that we can push $U \subseteq I_v$ of $\lceil |I_v|/\Delta \rceil$ insertions to c. Points in $\mathcal{U}$ are removed from I_v, I_c, D_c, P_c, and C_v. Any point p in $\mathcal{U}$ lexicographically larger than the minimum point in P_c (w.r.t. y) is removed from $\mathcal{U}$ and inserted into P_c and C_v. If P_c overflows, the lexicographically smallest points w.r.t. y are moved from P_c to $\mathcal{U}$ and removed from C_v until P_c no longer overflows. If c is a leaf then all points are inserted into P_c and $\mathcal{U}$ is now empty. Otherwise, the remaining points in $\mathcal{U}$ are added to I_c which might overflow and cause a similar overflow along a path from the root to a leaf in the worst case as in the case of the deletion buffer overflow. See Figure 7.4. The analysis and bounds are similar to the deletion buffer overflow.

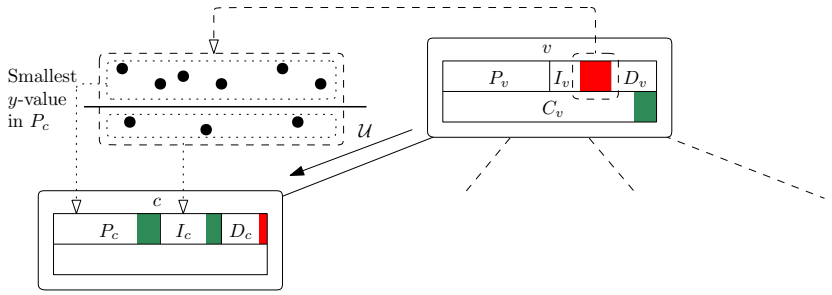

Figure 7.4: Insert buffer overflow. A total of $\lceil |I_v|/B^\varepsilon \rceil$ points are moved from I_v to $\mathcal{U}$. All points from $\mathcal{U}$ are removed from D_c since they cancel the deletions. Points larger than the smallest y-value in P_c are inserted into P_c and points smaller than the smallest y-value in P_c are inserted into I_c. This ensures the tree is heap ordered. Finally the newly added points to P_c are also inserted into C_v to ensure that C_v contains a copy of all points in P_c.

(iii) A point buffer overflows at a leaf v when $|P_v| > B/2$. If this is the case then we split the leaf into two and evenly distribute the points from P_v among the two leafs v' and v'' using $\mathcal{O}(1)$ I/O's. See Figure 7.5. The splitting of a leaf might cause the parent to get a degree of $\Delta + 1$.

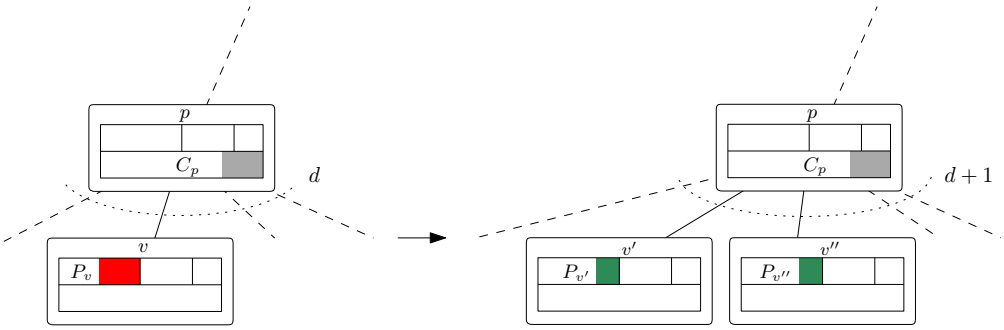

Figure 7.5: Point buffer overflow. The point buffer P_v is evenly distributed between v' and v''. After the split the parent might have too high degree.

(iv) An internal node v with a degree larger than Δ is split into two nodes v' and v''. I_v, D_v, and P_v are distributed among v' and v'' according to the x-values of the points. Finally the child structures of v' and v'' are rebuilt from the children's point buffers. See Figure 7.6. The split might cause the parent of v to have a degree overflow and in the worst case we need to split along a path from a leaf to the root. The splitting of a single node costs $\mathcal{O}(\Delta)$ I/O's due to the reconstruction of the child structures.

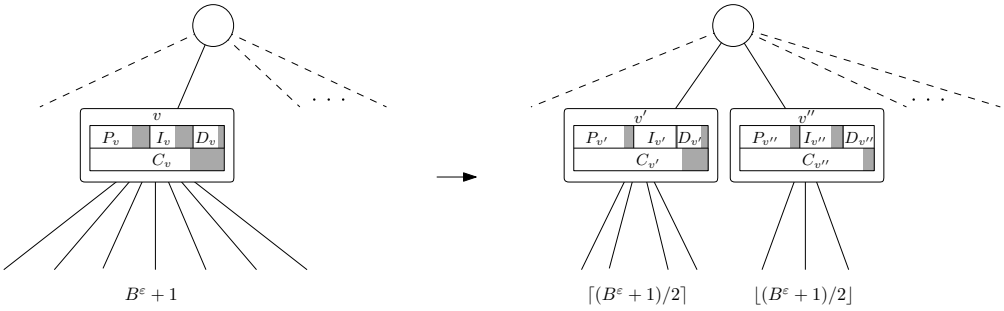

Figure 7.6: Node degree overflow. The buffers I_v, D_v, and P_v are distributed among nodes v' and v''. After the split we have to ensure the child structures mirrors all points of the children's point buffer.

(v) A point buffer underflows at v when $|P_v| < B/2$. In that case we try to *pull up* the highest $B/2$ points from the children of v into P_v. If v's subtree does not store any points then we remove all points from D_v and move points from I_v to P_v until $|P_v| = B$ or $I_v = \varnothing$. Otherwise we use $\mathcal{O}(\Delta)$ I/O's to identify the set X of the top $B/2$ points from the children of v and remove the identified points from the point buffers of the children and the child structure C_v of v.

If a point buffer of a child becomes empty before having identified all of the top $B/2$ points we have to recursively fill that child before continuing as the subtree might contain points with larger y-value than points in the remaining children of v.

All points in $X \cap D_v$ are then removed from X. This might cause $|X| < B/2 - |P_v|$ resulting in a repeated run of the procedure to guarantee X contains enough points to ensure P_v is no longer underflowed.

The remaining points of X are inserted into P_v and the child structure of the parent of v. Please refer to Figure 7.7 for an illustration of the main ideas of the pull up procedure.

The points of X inserted into P_v might have a smaller y-value than the points in I_v. We solve this problem by swapping the highest point in I_v with the lowest point in P_v while there exists a point in I_v that is higher than a point in P_v, and make sure to maintain the child structure of the parent to reflect the changes made to the insert and point buffer.

If the subtree of v becomes empty as a result of pulling points up to v we must remove all points of D_v and move points from I_v to P_v. This might cause P_v to overflow.

Finally, after having pulled points from the children, we check if any of the children's point buffers underflows and should be refilled.

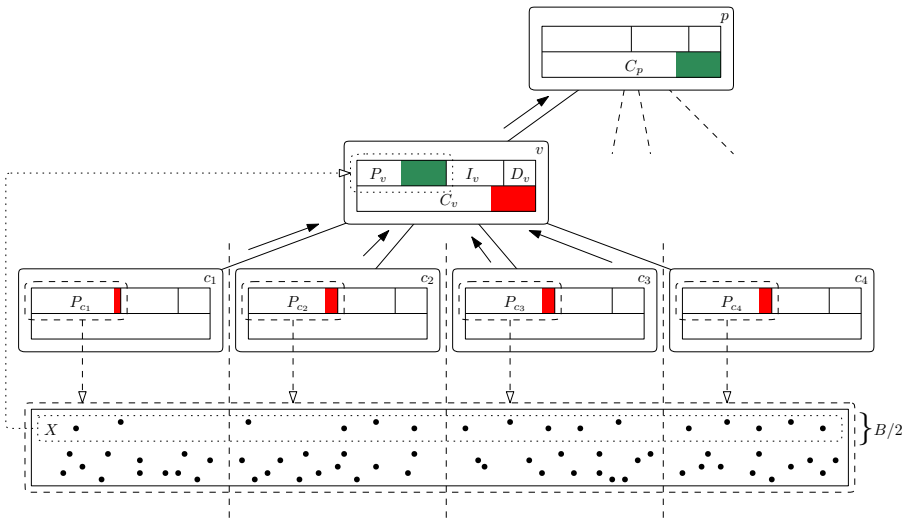

Figure 7.7: Point buffer underflow. Here $B = 32$, $\varepsilon = 2/5$ giving $B^\varepsilon = 4$. The point buffer P_v contains less than $B/2$ points, i.e. it is underflowed. The point buffers of the children are considered and the top $B/2$ points are added to X. Now deletions from D_v cancels points in X and the remaining of X is inserted into P_v and the child structure of the parent C_p. Finally the heap order is maintained by swapping points between P_v and I_v, and by reflecting these changes in C_p.

Analysis

The tree remains balanced during insertions as the tree only increase in height whenever the root splits, which causes every path from root to leaf to increase by one. In the B-Tree we handle rebalancing using fusion of nodes. We do not apply this method here. Instead we apply global rebuilding when a linear number of updates have been performed. By (iii) it follows that the total number of leafs created during N insertions can be at most $\mathcal{O}(N/B)$ implying that at most $\mathcal{O}(\frac{N}{\Delta B})$ internal nodes can be created by splitting internal nodes. From this it follows that the tree has height $\mathcal{O}(\log_\Delta \frac{N}{B}) = \mathcal{O}(\frac{1}{\varepsilon}\log_B N)$.

We can now argue that every update in (i) and (ii) requires amortized $\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B N)$ I/O's. As every $\Theta(B/\Delta)$ update require $\mathcal{O}(1)$ I/O's on every layer of the tree we get the correct amortized bound:

$$\begin{aligned}\mathcal{O}\left(\frac{1}{B/\Delta}\log_\Delta \frac{N}{B}\right) &= \mathcal{O}\left(\frac{B^\varepsilon}{B}\log_{B^\varepsilon}\frac{N}{B}\right)\\ &= \mathcal{O}\left(\frac{B^\varepsilon}{B}\frac{1}{\varepsilon}\log_B N\right)\\ &= \mathcal{O}\left(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B N\right)\end{aligned}$$

In (iii), we know that at most $\mathcal{O}(N/B)$ leafs are created each requiring $\mathcal{O}(1)$ I/O's giving amortized $\mathcal{O}(1/B)$ I/O's per update.

In (iv), we know that at most $\mathcal{O}(\frac{N}{B\Delta})$ internal nodes are created. The creation of such a node costs $\mathcal{O}(\Delta)$ giving an amortized cost of $\mathcal{O}(1/B)$ I/O's per update.

In (v) each refilling might trigger a cascaded recursive refilling of one or more of the children. Every refilling takes $\mathcal{O}(\Delta)$ I/O's and moves $\Theta(B)$ points one level up through the tree's point buffers. Each point can at most move $\mathcal{O}(\log_\Delta \frac{N}{B})$ levels up, as this is the tree's height. This means that the total number of I/O's for the refillings during the course of N operations is amortized $\mathcal{O}(\frac{1}{B/\Delta}\log_\Delta \frac{N}{B}) = \mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B N)$ per point.

This argument ignores the fact that when pulling up points some points might swap positions from I_v to P_v. This swap does not change the fact that the number of points we pull up remain the same and therefore it does not affect the amortized accounting.

Another fact that we ignore is what happens if we are not able to pull up $B/2$ points from the children. This is solved by a simple amortization argument. We double charge the operation responsible for pushing points to a child. This way we can ensure each node with non-empty point buffers always has an I/O saved for being emptied by a recursive pull up.

7.1.3 Global rebuilding

Since we do not fuse nodes with too low node degree we might end up with an unbalanced tree. We use global rebuilding as described in Section 3.2 to guarantee the tree never gets *too* unbalanced which would disprove our

amortized bounds. This is done by partitioning updates into epochs. After a rebuild a new epoch begins and if the data structure at this points stores $\widetilde{N}$ points, then the next epoch will begin after $\widetilde{N}/2$ updates, i.e. a global rebuild will be performed. Having a new epoch after every $\widetilde{N}/2$ updates ensures that the tree never grows higher than $\mathcal{O}\left(\frac{1}{\varepsilon}\log_B \frac{3\widetilde{N}}{2}\right) = \mathcal{O}\left(\frac{1}{\varepsilon}\log_B N\right)$ as the size of the tree is $\frac{1}{2}\widetilde{N} \leq N \leq \frac{3}{2}\widetilde{N}$.

Global rebuilding works by constructing an empty structure and then reinserting all the points of the old structure that has not been deleted.

The points to reinsert are found by doing a top-down traversal of the tree while flushing insertion and deletion buffers to children. The points to reinsert are then found in the point buffers after flushing. This might cause buffers to temporarily overflow but we will allow this as the old structure will be deleted.

Once the set of points to reinsert have been found we simply insert the points in an initially empty tree.

Analysis

Elements at level i (leaf layer being level 0) can at most be flushed i levels down. The structure holds at most $\frac{3\widetilde{N}}{2B}$ nodes in total and at level i the structure has at most $\frac{3\widetilde{N}}{2B}\frac{1}{\Delta^i}$ nodes. The cost of flushing all the buffers at level i becomes $i \cdot \frac{3\widetilde{N}}{2B}\frac{1}{\Delta^i}$.

By summing over all layers of the tree we get the total cost of flushing all buffers to be:

$$\begin{aligned}
\sum_{i=0}^{\mathcal{O}(\log_\Delta \frac{3N}{2B})} i \cdot \frac{3\widetilde{N}}{2B}\frac{1}{\Delta^i} &< \sum_{i=0}^{\infty} i \cdot \frac{3\widetilde{N}}{2B}\frac{1}{\Delta^i} \\
&= \frac{3\widetilde{N}}{2B}\sum_{i=0}^{\infty}\frac{i}{\Delta^i} \\
&= \frac{3\widetilde{N}}{2B}\sum_{i=0}^{\infty}\left(\frac{1}{\Delta}\right)^i \\
&= \frac{3\widetilde{N}}{2B}\frac{1}{1-\Delta} \\
&= \mathcal{O}(N/B)
\end{aligned}$$

This gives an amortized cost of $\mathcal{O}(1/B)$ per update to flush all buffers.

The $\mathcal{O}(\widetilde{N})$ reinsertions into the new, initially empty, tree can be done in amortized $\mathcal{O}(\frac{\widetilde{N}}{\varepsilon B^{1-\varepsilon}}\log_B \widetilde{N})$ I/O's which is paid for by the $\widetilde{N}/2$ updates during the epoch.

7.1.4 Three sided range queries

Reporting a three-sided query $Q = [x_1, x_2] \times [y, \infty]$ consists of three steps. Namely, identifying the nodes to visit, push down delayed insertions and deletions between the identified nodes, and finally reporting the points contained in Q.

We identify the nodes to visit in a breadth first manner. Starting from the root we identify, from the query's x-range, the children that are relevant to the query and push all insertions and deletions belonging to those children. This is done without handling possible overflows. After this we know that the point buffers of the children do not change further and we can thus report all points in the query range from the child structure and the point buffer. The children worth visiting are then added to the back of the breadth first search queue. We can decide whether a child is worth visiting without reading the node by comparing the query-y with the minimum y-value of that child's point buffer. This follows from the heap-order of point buffers. If the query-y lies above the minimum y-point of a child, then by the heap-order invariant, we know that no relevant points are to be found in the subtree rooted at that child. The minimum y-value for every child is stored in the parent. All nodes, except for the root, do not need to report from their point buffers as the parent of the node has already reported the relevant points from the child structure.

After all points have been reported we might have some buffers that are temporarily overflowed. This is now handled in a bottom up fashion using the update operations described in Subsection 7.1.2. We will handle a single subtree at a time and make sure that the entire subtree has no broken invariants, i.e. buffers that overflow or underflow or any nodes with too high node degree.

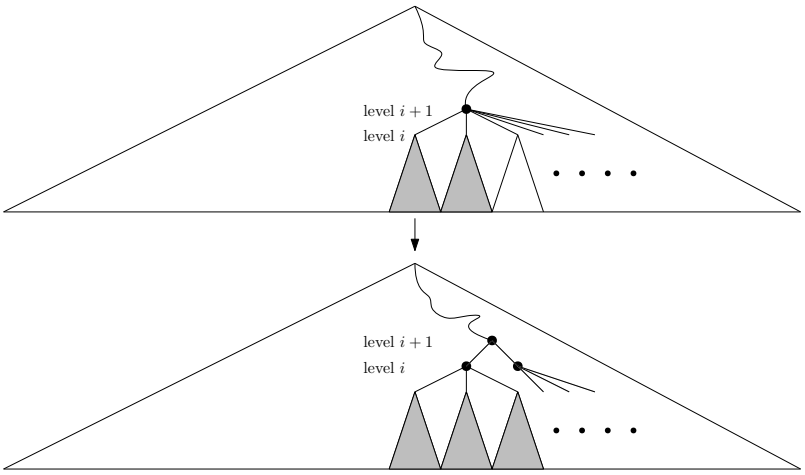

Figure 7.8: Fixup of the tree after having reported a query. All broken invariants of subtrees at level i are handled before handling broken invariants at level $i+1$. Here, the gray subtrees respects all invariants and we are about to handle the white subtree. The node degree overflow at level $i+1$ is handled after all subtrees at level i are valid w.r.t. the invariants.

We do this by applying the update operations (i)-(iv)[1] while we disallow any recursion leaving the subtree, i.e. no splits may recursively split nodes outside of the subtree. Only when the entire subtree has no overflows or underflows can we remedy a potential underflowed point buffer using update operation (v)[2], and only then are we allowed to continue to a subtree one level higher. See Figure 7.8. This way of handling the fixup procedure ensures we never get interleaving update operations interfering with each other.

Analysis

During a query assume we visit V nodes *not on* the search paths for x_1 or x_2 and $\mathcal{O}\left(\frac{1}{\varepsilon}\log_B N\right)$ nodes *on* the search paths. We know that the V nodes must have at least $VB/2$ points in their point buffers before updates are pushed down. The number of deletions we push down to visited nodes can at most be $\left(V + \mathcal{O}\left(\frac{1}{\varepsilon}\log_B N\right)\right) B/4$.

It now follows that the number of points we report, K, must be at least the number of points in the point buffers before pushing down minus the deletions we push down:

$$VB/2 - \left(V + \mathcal{O}\left(\frac{1}{\varepsilon}\log_B N\right)\right) B/4 = VB/4 - \mathcal{O}\left(\frac{B}{\varepsilon}\log_B N\right) = K$$

By isolating V it follows that $V = \mathcal{O}\left(\frac{1}{\varepsilon}\log_B N + K/B\right)$. The worst case bound now becomes the sum of visiting the V nodes, the nodes on the search paths for x_1 and x_2, and the output: $\mathcal{O}(V + \frac{1}{\varepsilon}\log_B N + K/B) = \mathcal{O}(\frac{1}{\varepsilon}\log_B N + K/B)$. On top of this comes the cost of pushing down update buffer elements and handling overflowing update buffers and overflowing point buffers.

This cost of pushing $\Omega(B/\Delta)$ points to a child is however already paid for by the update operations and is thus covered by the analysis of Subsection 7.1.2. It is only when we push down $\mathcal{O}(B/\Delta)$ updates to a child, with an amortized cost of $\mathcal{O}(1)$ that this cost is covered by the cost of visiting the child.

Handling the overflowing update buffers and underflowing point buffers are also paid for by the update operations described in Subsection 7.1.2.

This all adds up to a total amortized cost of $\mathcal{O}(\frac{1}{\varepsilon}\log_B N + K/B)$ I/O's for a three-sided range query.

7.1.5 Construction

The structure can be initialized with a set of N points using $\mathcal{O}(\text{Sort})$ I/O's. If the points are sorted on the x-coordinate we only need $\mathcal{O}(\text{Scan})$ I/O's. For the remainder of this section we assume that the points are initially sorted w.r.t. to the x-coordinate.

[1](i) overflowing deletion buffers (ii) overflowing insertion buffers (iii) split leafs with overflowing point buffers (iv) split nodes of degree $\Delta + 1$

[2](v) fill underflowing point buffers

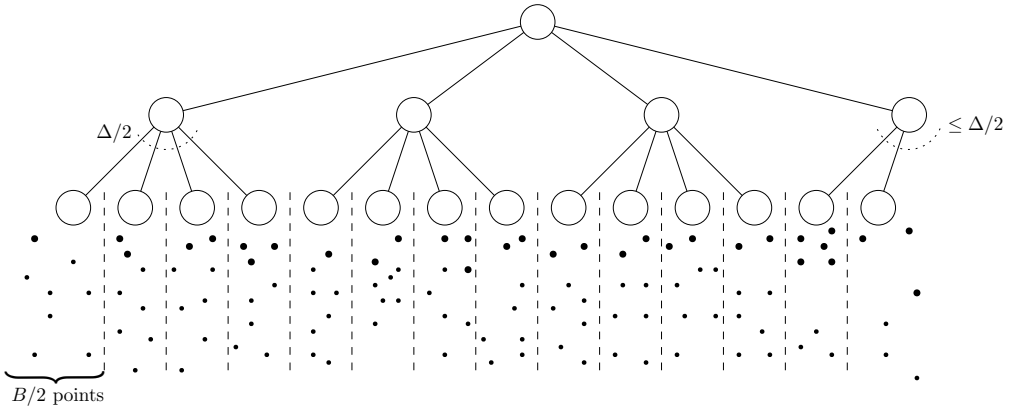

Figure 7.9: A B-Tree with each internal node have a degree of $\Delta/2$. Each leaf stores $B/2$ points with the possible exception of the rightmost. The bold points are the top $B/2$ points that is pulled up from the leafs into the layer above.

The first step of the construction is to construct a B-Tree over the x-values. We let each internal node have a degree of $\Delta/2$ and each leaf stores $B/2$ points, with the exception of the rightmost leaf which might contain less than $B/2$ points, and the rightmost internal nodes having a degree less than $\Delta/2$.

The point buffers of the internal nodes are now filled bottom up by pulling up the top $B/2$ highest y-value points. If this results in a child having an underflowing point buffer we recursively fill that child before proceeding. In a second iteration we do the same but in a top-down fashion.

All insertion and deletion buffers are initially empty and the child structures are constructed from the point buffers of the children.

This construction algorithm could also have been used for global rebuilding giving a matching amortized cost.

Analysis

We know level i of the tree contains at most $\frac{N}{B\Delta^i}$ nodes. It follows that the number of points stored at or above level i is $\mathcal{O}\left(\sum_{j=i}^{\infty} B\frac{N}{B\Delta^j}\right) = \mathcal{O}\left(\frac{N}{\Delta^i}\right)$. It follows that we cannot move $B/2$ points to level i from $i-1$ more than $\mathcal{O}\left(\frac{N}{\Delta^i}/\frac{B}{2}\right)$ times. We know that we can move $B/2$ points using $\mathcal{O}(\Delta)$ I/O's and thus the total number of I/O's to fill the point buffers becomes:

$$\mathcal{O}\left(\sum_{i=1}^{\infty}\Delta\frac{N}{B\Delta^i}\right) = \mathcal{O}\left(\frac{N}{B}\sum_{i=1}^{\infty}\Delta\frac{1}{\Delta^i}\right) = \mathcal{O}\left(\frac{N}{B}\sum_{i=0}^{\infty}\frac{1}{\Delta^i}\right) = \mathcal{O}\left(\frac{N}{B}\right)$$

Another aspect we have to look at is what happens to the amortized analysis when we initialize our data structure using this construction method, i.e. we have to argue that the amortized cost of the remaining operations remain unchanged during the epoch started by the construction.

In order to argue this we consider a sequence of operations containing N_{ins} insertions and N_{del} deletions and a newly constructed tree of N points.

Let us first consider the cost of creating new nodes in the tree. Each leaf has initially at most $B/2$ points and it follows that we can at most create $2N_{ins}/B$ new leafs. Each new leaf is created in $\mathcal{O}(1)$ I/O's and it thus cost at most $\mathcal{O}(N_{ins}/B)$ I/O's to create new leafs during N_{ins} insertions. By a similar argument it follows that at most $\mathcal{O}\left(\frac{N_{ins}}{\Delta B}\right)$ new internal nodes can be created since each internal node initially has a degree of $\leq \Delta/2$. Each new internal node is created in $\mathcal{O}(\Delta)$ I/O's and it thus costs $\mathcal{O}(N_{ins}/B)$ I/O's to create new internal nodes without the cost of refilling point buffers. The refilling of point buffers will be accounted for in the following.

An insertion has to be moved at most from the top to the bottom of the tree before it is cancelled or moved into a point buffer. Since the height of the tree is $\mathcal{O}\left(\frac{1}{\varepsilon}\log_B N\right)$ it follows that the cost of handling overflowing insertion buffers during the course of N_{ins} insertions becomes $\mathcal{O}\left(\frac{N_{ins}}{B/\Delta}\frac{1}{\varepsilon}\log_B N\right)$ I/O's. A similar argument can be given for the case of deletions.

Each deletion leaves behind a hole which needs to be filled. This hole is filled by recursively pulling up points which effectively moves down the hole. Each split of internal nodes also potentially creates up to B holes. In total we need to handle $\mathcal{O}\left(N_{del} + \frac{N_{ins}}{\Delta B}\right)$ holes. We can move up $B/2$ points using $\mathcal{O}(\Delta)$ I/O's and they at most need to be moved to the top of the tree, i.e. they need to be at most moved the height of the tree levels up. This gives a total cost of $\mathcal{O}\left(\left(N_{del} + \frac{N_{ins}}{\Delta B}\right)\frac{\Delta}{B}\frac{1}{\varepsilon}\log_B N\right)$ I/O's.

All this adds up to $\mathcal{O}\left(\frac{N_{ins}+N_{del}}{B/\Delta}\frac{1}{\varepsilon}\log_B N\right) = \mathcal{O}\left(\frac{N_{ins}+N_{del}}{\varepsilon B^{1-\varepsilon}}\log_B N\right)$ I/O's. This matches the amortized bounds of the structure.

"The problem with quotes found on the internet is that they are often not true."

— Abraham Lincoln

8 Other structures

8.1 R-Tree

The R-Tree was introduced by Antonin Guttman in [Gut84]. The structure is heuristic in nature and does not provide any close to optimal worst case search bounds. Arge et al. has, however, provided strong evidence for R-Trees outperforming several theoretical optimal data structures in practice [ABHY08]. We will introduce the important ideas of the R-Tree.

Let P be a set of points. An R-Tree stores all points from P in leaf nodes, each of which contains $\Theta(B)$ points. Each non-leaf node u has $\Theta(B)$ children, except for the root which must have 2 children unless it is the only node in the tree. For each child c, u stores a *minimum bounding rectangle* (MBR), which is the smallest rectangle that *tightly* encloses all the points in the subtree of c. Note that there is no constraint on how points should be grouped into leaf nodes. Also, there is no constraint on how non-leaf nodes should be grouped in higher level nodes. Since each point is stored only once, the entire data structure consumes linear space. Please refer to Figure 8.1 for an illustration of an R-Tree structure.

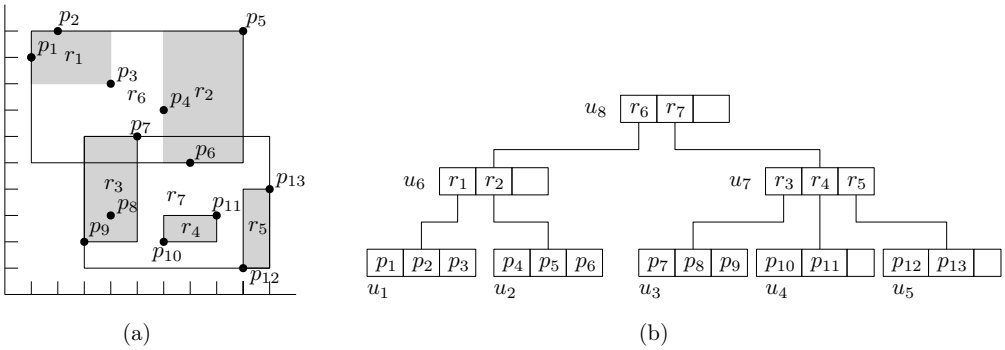

Figure 8.1: (a) Data and MBRs (b) The R-Tree structure

8.1.1 Query

Given a query $Q = [x_1, x_2] \times [y_1, \infty]$ we want to find all the points in P covered by Q. The relation to the R-Tree is that we only need to visit those nodes whose MBRs intersect Q. Intuitively, this means we desire as *small* MBRs as possible, as this directly implies that a query spans fewer MBRs, again implying fewer nodes are visited. A good heuristic is therefore to minimize the perimeter of each MBR as this directly implies MBRs covers smaller areas.

8.1.2 Insertions

To insert a point p, we add p to a leaf node u by following a single root-to-leaf path. If u overflows we split it, which creates a new child of $parent(u)$. This could cause $parent(u)$ to overflow which is handled in a recursive manner. Finally, if the root split, then a new root is created. Note that it is legal to insert a point p into *any* leaf, after which, the data structure will still be considered legal. This is the main property that differs R-Trees from standard B-Trees. There are several heuristics for choosing a subtree to insert into. It is these heuristics that gives rise to the different R-Tree variants. We will cover the original R-Tree heuristic and the R*-Tree heuristic.

The formal definition of inserting a new point p is as follows. Given a non-leaf u with children $c_1, c_2, \ldots, c_{\Theta(B)}$, we need to pick the best child c^* such that the new point p is best inserted into the subtree of c^*.

Choosing a subtree to insert into in an R-Tree. The standard R-Tree chooses the best child in a greedy manner. Specifically, c^* is simply the child c_i whose MBR requires the *least increase* of area in order to cover p.

Choosing a subtree to insert into in an R*-Tree. The problem in the original R-Tree is that certain types of data points may create small areas but large distances which will initiate a bad split. To overcome this, a mixed heuristic is employed. At leaf level we try to minimize the overlap and in case of *ties* the MBR that requires the *least increase* of perimeter is chosen. If this again yields a tie the MBR that increases the least in area is chosen. At the higher levels, it behaves similar to the R-Tree.

Node split in an R-Tree was by Guttman originally proposed handled using two different heuristics. The *linear method* chooses far apart nodes as ends. Randomly nodes are then chosen and assigned such that they require the smallest MBR enlargement. The *quadratic method* chooses two nodes such that the dead space between them is maximized. Nodes are then assigned such that the MBR area is minimized.

Node split in an R*-Tree is more involved but the main idea is to always split point set S using an axis-orthogonal cut. This means that points of S are sorted w.r.t. their x- and y-coordinate respectively. Then, the first $B/2$ points are inserted into S_1 and the rest is inserted into S_2 for the x-sorted points and into S_1' and S_2' for the y-sorted points. The final split is the better one, i.e. the split that have the least combined MBR perimeter and least combined MBR area. The above applies to splitting of a leaf node. The case of a non-leaf node is a bit different because the items split are MBRs. The strategy is however

the same and involves sorting the MBRs by their centroids. Please refer to Figure 8.2 for an illustration of the splitting of a leaf.

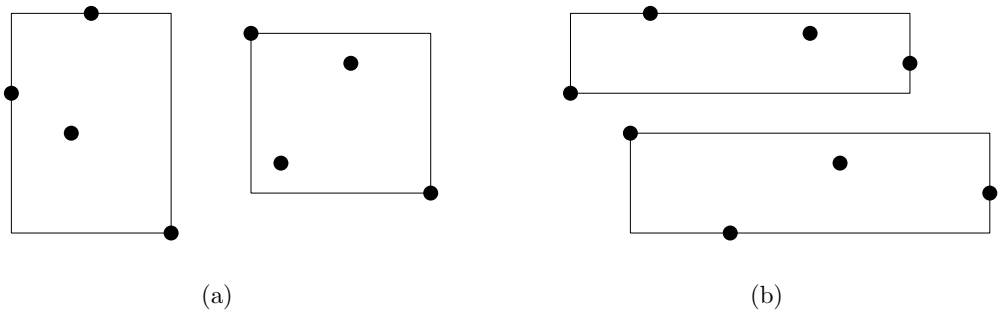

Figure 8.2: (a) Split by cutting the x-dimension (b) Split by cutting the y-dimension.

8.1.3 Deletions

Let p be the point to be deleted. First the leaf node u which stores p is found using p as search region. Then p is removed from u. The deletion is done if the node has more than λB points, where λ denotes the minimum node utilization. Otherwise, u *underflows*, which is handled by first removing u from its parent, and then re-inserting all points in u using the insert algorithm described earlier. Now, removing u from $parent(u)$ may cause $parent(u)$ to underflow too. In general, the underflow of a non-leaf node u' is also handled by re-insertions, with the only difference that the items re-inserted are MBRs, and each MBR is re-inserted to the same level of u'.

8.1.4 Analysis

It follows trivially from construction that a point can be inserted in $\mathcal{O}(\log_B N)$ by simply searching after the MBR that is responsible for the update.

While much work has been done on evaluating the practical query performance of the different variants of the R-Tree, very little is known about their theoretical worst-case performance. Most theoretical work on R-Trees is concerned with estimating the expected cost under hard assumptions on the distribution of input and on the queries that is to be answered.

Since we cannot guarantee the heuristics for constructing the R-Tree choosing all MBRs not to overlap, we believe a worst case analysis for querying must be $\mathcal{O}(N)$.

8.2 MySQL

MySQL is a popular open-source relational database management system. We will not give an in-depth description of how relational databases work but we will describe how to very simply adapt MySQL to answer three-sided range queries. A very minimal table was constructed with just two columns. One for the x-coordinate and one for the y-coordinate. These two columns

are base for a primary key on the table. This eliminates duplicates in the table and allows for faster range queries by building some variant of a B-Tree on the concatenation of the x- and y-coordinate yielding a single key, i.e. MySQL does nothing extraordinary to utilize the two elements of the key. Inserting in the table was done using simple `INSERT IGNORE INTO table VALUES { values }` SQL queries. In order to make this work efficiently we buffer inserts in memory and bulk insert whenever the buffer overflows. This gave a significant speedup. Deletion was done using `DELETE FROM table WHERE (x,y) IN { values }` queries and with buffers on top.

Queries of the form $[x_1, x_2] \times [y_1, \infty]$ were answered using a `SELECT * FROM table WHERE` x_1 `<= x AND x <=` x_2 `AND y >=` y_1 query.

8.2.1 Analysis

It is important to keep an open connection to the server at all times; if not we will end up spending a lot of time reconnecting to the server.

Inserting points in the database involves sending the query to the server, parsing the query, and finally inserting the rows. Due to the primary key on the table, we know that MySQL will build a B-Tree on the data. This will give an insertion time of $\mathcal{O}(\log_B N)$ I/O's for some MySQL implementation specific B.

Deleting points is similar to inserting and is also done in $\mathcal{O}(\log_B N)$.

It is a little harder to argue about the complexity of a three sided query. A B-Tree can answer normal two sided range queries in $\mathcal{O}(\log_B N + K/B)$. We can however not guarantee that all points in the two sided range should be reported and thus we cannot properly attribute any I/O's to the output, i.e. use filtering. We will have to settle with a complexity of $\mathcal{O}(\log_B N + T/B)$ where T is the size of the output within the x range of the query.

If we enforce no index on the MySQL table, then we believe the standard implementation will simply append all insertions to a continuous stream, which can be done in $\mathcal{O}(1)$ I/O's. We believe deletions and queries can be handled in $\mathcal{O}(N/B)$ I/O's by scanning the entire stream.

"There is nothing more deceptive than an obvious fact."

— Sherlock Holmes

9 Implementation

Throughout our implementation of the Internal Memory Priority Search Tree, External Memory Priority Search Tree, and the External Memory Buffered Priority Search Tree we noted down important considerations. In this chapter we present these considerations together with a short presentation of how we wrote wrappers around MySQL, Boost R-Tree, and libspatialindex R*-Tree. We end the chapter with a description of the experimental framework we developed to significantly simplify the experimental phase of the project.

All code can be cloned from the following git repository or downloaded from the mirrors listed below. Instructions on how to compile and run the code can be found in the accompanying `readme` file.

`https://github.com/gabet1337/speciale`
`http://cs.au.dk/~peterg/three_sided.zip`
`http://cs.au.dk/~chrha22/three_sided.zip`

9.1 General

Almost all code was developed using pair programming and we strongly believe this technique, though slow and cumbersome, eliminated many mistakes that would otherwise had slowed us down later on. Everything we implemented was unit tested and large parts of the project implemented using test driven development. We made sure to make sound design choices to enable easy extension and reuse of our code. Stubs and mocks were used to ease the integration of substructures into larger structures. We also wrote checkers to automatically check validity of a tree. In the case of the External Memory Buffered Priority Search Tree we wrote a checker that would iterate the entire tree and check that no invariants were broken. Furthermore we included a method to print the trees in a DOT format (graph description language) allowing us to visualize every step of the algorithm. This really made debugging easier as it supplied us with a quick overview of the structure.

We optimized the code as much as time allowed using the profiling tool `valgrind`.

In order to fully test that there were no errors in our implementation we wrote a test that would insert thousands of random points and test validity of the tree for every insert, delete, and query. We would then repeat this test for several thousand iterations over several days while continuing on other parts of the code.

Using all these methods and tools we feel confident that our code works as intended.

9.2 Stream

The implementations we present make use of the concept of *streams*. A stream gives access to reading and writing from disk to memory and vice versa. A stream typically manages an internal buffer which is a mirror of a small portion of the external memory allowing for fast interaction with that small piece of data. Although the `C++` standard library provides several streams that allow for the internal buffer to be managed we introduce an implementation of our own. We denote this stream `buffered_stream`. This design choice was made because of the nature of our experiments in which it is of extreme importance that we are able to argue about the exact number of I/O's being used. By introducing a stream of our own we avoid that any undefined behaviour in the standard library implementations gives rise to a potential I/O overhead. Any such I/O overhead would be reflected directly in the overall running time of our implementation giving us a hard time to reason about the behaviour. A stream of our own would further more allow us to count the exact number of I/O's being used.

The stream we introduce makes use of the `read` and `write` system calls and is equipped with a buffer of size B that is maintained on all operations.

There are many different ways to construct a stream. In order to substantiate our choice of using buffering on top of the `read` and `write` system calls we conducted some experiments with different types of streams:

- Direct invocation of the operating system calls `read` and `write` that reads/writes one item using no buffering mechanism.
- The standard library streams `fread` and `fwrite` that use a built-in buffering mechanism that we do not manage.
- Direct invocation of the operating system calls `mmap` and `munmap` that makes use of the operating system's virtual memory mechanism through demand paging.

It is clear that direct invocation of the `read` and `write` system calls cannot be better than adding buffering on top, which early experiments without doubt showed. The results were so slow that we had to exclude them.

The results of Figure 9.1 and 9.2 show that the `buffered_stream` performs similarly to using `mmap` when reading 5Gb of data while using `fread` and `fwrite` is significantly slower. Figure 9.2 surprisingly shows that it is faster

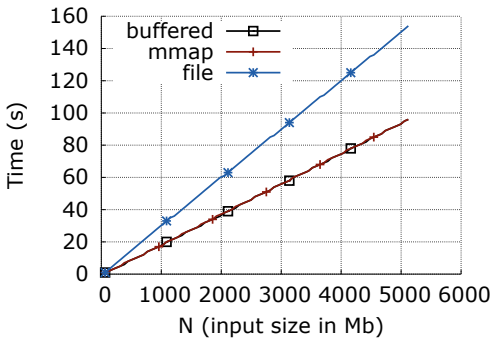

Figure 9.1: Reading 5Gb.

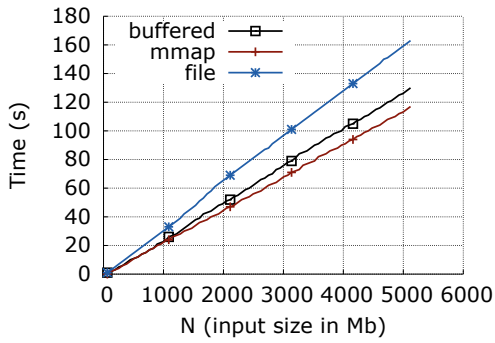

Figure 9.2: Writing 5Gb.

to use the virtual memory mechanism to write data. Using virtual memory however also means that we lose some control of when and how data is written to disk, and we lose the control to accurately measure disk I/O's as we have no control of how the operating system schedules I/O's. We would also have a harder time measuring the number of page faults caused by our structures' internal work as every I/O will cause a page fault which will interfere with our measurements. With these considerations in mind we conclude that it makes most sense to use the `buffered_stream`.

9.3 External Memory Buffered Priority Search Tree

This section presents important implementation specific design choices for the main data structure of the External Memory Buffered Priority Search Tree by Brodal presented in Section 7.1.

We decided to implement a **dependency mechanism** on each node that allows for the node to be partly flushed and loaded. This allows us to optimize the number of I/O's needed as we can restrict the load and flush to required data. The data for each node naturally groups into a separate file for the insert buffer, delete buffer, point buffer, meta data on children, and meta data for the node itself.

As described in Section 7.1 we have subroutines calling each other. If we simply let each subroutine call each other using a naïve call stack we would have to manage that a subroutine can be both caller and callee. This would require a complex logical mechanism that does nothing but manage the control flow of the recursion and handling of data load and flushing. In order to overcome this we have introduced an **event loop mechanism** that uses a stack of events to control flushing, loading, and calling proper subroutines. Using this mechanism we are able to predict exactly what data is needed. Furthermore, it becomes an easy task to flush all required data before taking further steps in the recursion ensuring optimal use of available internal memory.

As it is not uncommon to see consecutive events for the same node we have added a **caching mechanism** that makes sure not to flush data on any nodes used in the previous event if the same nodes and data is required in

the current event. This idea of using a simple caching scheme dramatically reduces the number of I/O's required compared against the naïve solution.

In order to further reduce the number of I/O's required we make sure to maintain a detailed *view* of the state of each node. This enables us to test, in only a single I/O, whether a node is internal or a leaf, and if it has any broken invariants.

As both event loop and buffer over-/underflow thresholds depends on whether we are currently global rebuilding, linear constructing, reporting, or handling updates, we introduce a **state switch** that is used throughout our implementation to decide which path the recursion should follow.

The **general representation** of buffers makes use of Red-Black search trees (`std::set` from the `C++` standard library) on totally ordered points. This design choice allows us to retrieve the minimum and maximum element in each buffer in constant time using `iterators`. Furthermore we are able to traverse each buffer in sorted order in linear time. We are aware that using a search tree comes with the price of a space blow-up. We maintain the point buffer as two separate search trees totally ordered on the x-coordinate and y-coordinate respectively. This is needed as we frequently need access to the minimum y-value when deciding whether we should insert a point into the point buffer or insert buffer. The **catalogue structure** containing information about the children of each node is also represented as a Red-Black search tree. In the **info file** of each node we maintain information on whether the node is a leaf, a virtual leaf[1], or an internal node, whether the node currently has an overflowing or underflowing point buffer, insert buffer, delete buffer, and whether the node is currently node degree overflowed. This allows the event loop to identify whether we should remedy any broken invariants using only 1 I/O.

9.4 External Memory Priority Search Tree

Drawing from the experiences gained while implementing the External Memory Buffered Priority Search Tree of Section 9.3 we decided to once again make use of the event loop to handle our recursion allowing for full control of loading and flushing. This choice also allowed for simple adaptation of the caching mechanism previously described.

The elements of the base B-Tree is a simple type containing a point, a reference to a child, and a boolean telling us whether the point has been deleted. The elements are stored in nodes which is just a collection of the point type, a reference to a query data structure, and some booleans used in the loading and flushing mechanisms. The collection used for the point type is a Red-Black tree (`std::set`). This collection allows us to find the child responsible for a point in logarithmic time using binary search and naturally keeps the points in sorted order w.r.t. the x-coordinates.

Updates and reporting are done as described in Section 6.3 and 6.4 with no remarkably deviations.

[1] An internal node with an empty subtree.

9.5 Other structures

We also implemented wrappers around MySQL 5.7.12, Boost 1.60.0 R-Tree using the quadratic method, and libspatialindex 1.8.5 external R*-Tree such that we could use these structures as a Priority Search Tree. The structures are described further in Chapter 8. We made sure to implement functionality to disallow duplicates of points on top of the basic functionality.

9.6 Experimental framework

We developed an extensive framework for running experiments in order to automate the process as much as possible.

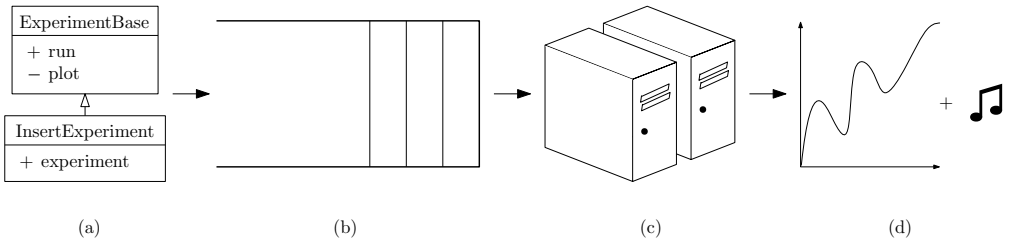

Figure 9.3: (a) An experiment is created by extending a base class (b) The experiment is added to an experiment queue (c) When a machine becomes available the experiment is run (c) Measures on statistics and time are automatically plotted and finally the motherboard beeper plays a tune to signal that an experiment has been processed.

Figure 9.3 depicts the flow of an experiment from thought to result. In order to achieve this flow we wrote a framework that would do all the hard work for us such that all we had to do was to describe the actual experiment. As an example, to test the time it takes to insert in all the structures we only had to tell the framework to insert into the Priority Search Tree and measure for each 10 megabytes. The framework would then automatically run the experiment on all the structures one-by-one, measure time, I/O's, page faults, and other data structure specific statistics (overflows, splits, etc.), plot all the gathered data as a function of the input size, and finally play a small tune to signal that the experiment had finished. To make sure that no experiment leaves any ungathered memory behind we start each experiment

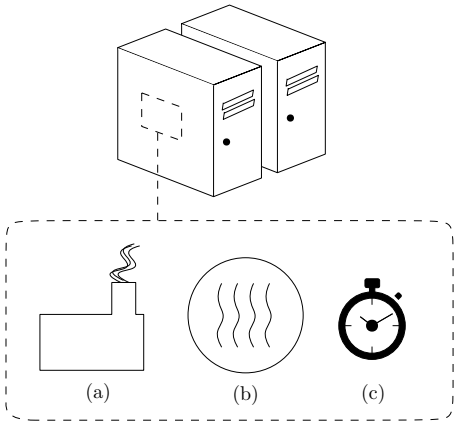

Figure 9.4: (a) A factory produces an instance of a Priority Search Tree (b) Each experiment is run in a delegated thread (c) Measures are recorded on statistics and time.

in its own thread such that the memory would be automatically recollected upon termination of the thread. All caches are dropped between each experiment.

The running time was measured using the `high_resolution_clock` of the `chrono` library. It was measured in both seconds and milliseconds to make sure we had all the required data. In most cases it suffices to measure in seconds.

In order to obtain a deeper understanding of the results we chose to measure the number of major page faults generated when using the data structures. This was done using `perf` - a performance analysis tool for Linux. Finally we measured the number of I/O's by extending our stream with a counter on calls to `read` and `write`.

As the machines run on a 32 bit operating system it is very important to define `_FILE_OFFSET_BITS=64`. This forces the system to use 64 bit pointers when seeking in files allowing us to handle data sets larger than 2^{32} on the machines.

"We must never make experiments to confirm our ideas, but simply to control them."

— Claude Bernard

10 Experimental setup

Conducting experiments on I/O algorithms is extremely time consuming. In order to compare I/O efficient algorithms against internal memory algorithms we need input sizes that force the internal memory algorithms to store and load data to and from the disk (swapping). The input size to the algorithms can be severely minimized if run on machines with a small amount of internal memory which in turn decrease the running time severely as well.

It is important to mention some considerations when it comes to choice of persistent storage media. In recent years solid state drives (SSD's) have grown increasingly more desirably in terms of price per gigabyte and storage capacity, but there is still a significant gap in price between electronic SSD's and mechanical HDD's. Disregarding this gap in price, solid state drives clearly outperforms the mechanical hard disk in every notable aspect. Solid state drives allow random access to blocks and diminish the seek time which is considered to be the culprit of the mechanical disks due to the rotational latency. The rotational latency makes it very important to store data on the disk in consecutive blocks as scattering data blocks across the disk would be detrimental to the performance as each block would have to wait for the rotational latency. As long as the mechanical disk is still as widespread as it is, these culprits of the mechanical disk have to be taken into consideration when designing I/O efficient algorithms.

With these considerations in mind we acquired two very old Dell machines with the specifications outlined in Figure 10.1.

We used newer machines for some experiments where it made sense. An example of a situation where it made sense was when we experimentally compared the actual running time with the theoretical running time of a single data structure, i.e. confirm whether the actual running time divided by the asymptotic complexity would give a horizontal line. For these experiments a newer machine would not change the results but rather provide faster results and not take up scarce time on the two Dell machines.

CPU	Intel(R) Celeron(R) CPU 3.06GHz
CPU L1 cache	16Kb
CPU L2 cache	256Kb
RAM	512Mb DDR 553MHz
Disk	Seagate ST3160828AS
Disk capacity	160Gb
Disk number of disks	2
Disk number of heads	4
Disk RPM	7200
Disk rotation time	8.33ms
Disk seek time	8.5ms
Disk buffer size	8192Kb
Disk sector size	512bytes
Operating system	32 bit Ubuntu 14.04
Kernel version	4.2.0-27-generic
Compiler	gcc 4.8.4 with optimization level 2

Figure 10.1: Specifications of the two Dell machines used for running experiments.

"Big results require big ambitions."
— Heraclitus of Ephesus

11 Our results

The performance of the structures were evaluated and compared through meticulous experimentation. In this chapter we present and discuss the most interesting results. Some of the experiments are limited to run only for a fixed amount of time. This was a necessary restriction as the internal memory structures are severely limited when data input is greater than the available internal memory. This will be very apparent in many of the presented results.

11.1 Parameter tuning

Both the original and buffered external memory Priority Search Tree by Arge et al. (Chapter 6) and Brodal (Chapter 7) respectively are parametrised with fanout and buffer size. These parameters are of a very machine dependent nature as larger internal memory allows for larger buffer sizes. It is our goal to utilize as much internal memory as available in the machine running the data structure. We decided to focus the parameter tuning sorely on the `insert` operation, since this would allow us to construct data structures with a large amount of data, which again would give rise to interesting experiments for the remaining operations. Put in other words; it is of no interest to achieve a data structure that queries *really* fast if we are unable to construct it with a decent amount of data within a decent amount of time.

In Subsection 11.1.1 we present our tuning parameters for the structure by Brodal and in Subsection 11.1.2 we present our findings for the structure by Arge et al.

11.1.1 External Memory Buffered Priority Search Tree

The experiments consisted of inserting 5Gb of data, i.e. for a 8 byte point $5 \cdot 1024 \cdot 1024 \cdot 1024/8 \approx 670$ million data points. The coordinates of the data points were chosen uniformly distributed among the positive integers.

Buffer size

To tune the buffer size we fixed the fanout to two and varied the buffer size. Theoretically the running time should decrease with ever increasing buffer sizes as shown in Figure 11.1. We are, however, limited by the internal memory size and have to be careful not to cause swapping of internal memory to external memory as this greatly decreases performance.

The results as running time per insert are depicted in Figure 11.2. The actual running time follows the same tendency as the theoretical number of I/O's per insert. In Figure 11.3 we plot the actual number of I/O's used per insert. Again it seems we have a good alignment between the actual, and the theoretical, number of I/O's. In order to verify the running times to be truly bound by the number of I/O's we plot the actual time per insert divided by the theoretical number of I/O's per insert in Figure 11.4. We expect a close relation between running time and the theoretical number of I/O's and produce a plot with close-to horizontal lines. For buffers of sizes 1Mb, 2Mb, 4Mb, 8Mb, and 16Mb we believe this to be the case.

For buffer size 32Mb we see more fluctuations and we can hardly argue the plot follows a horizontal line. We believe this is caused by the limited amount of internal memory forcing us to utilize more physical memory than available, which again causes the operating system to swap out internal memory to external memory. We verified this by examining the measured number of page faults generated from which it became apparent that the data structure started to cause page faults using this buffer size. See Figure C.1.

In Section 9.3 we argued about the high space overhead of using Red-Black trees as buffers. When we underflow a point buffer we must load 2 child structures along with all of the related buffers giving a huge space blow up. These two facts together explains the relatively low threshold at which we begin to generate page faults.

We can conclude that while the buffers fit in internal memory the running time improves with larger buffer sizes. The results shown in this section was not generated from the machines described in Chapter 10, but a similar experiment on those machines showed that a buffer size of 8Mb performed best. Going forward we will stick to using a buffer size of 8Mb.

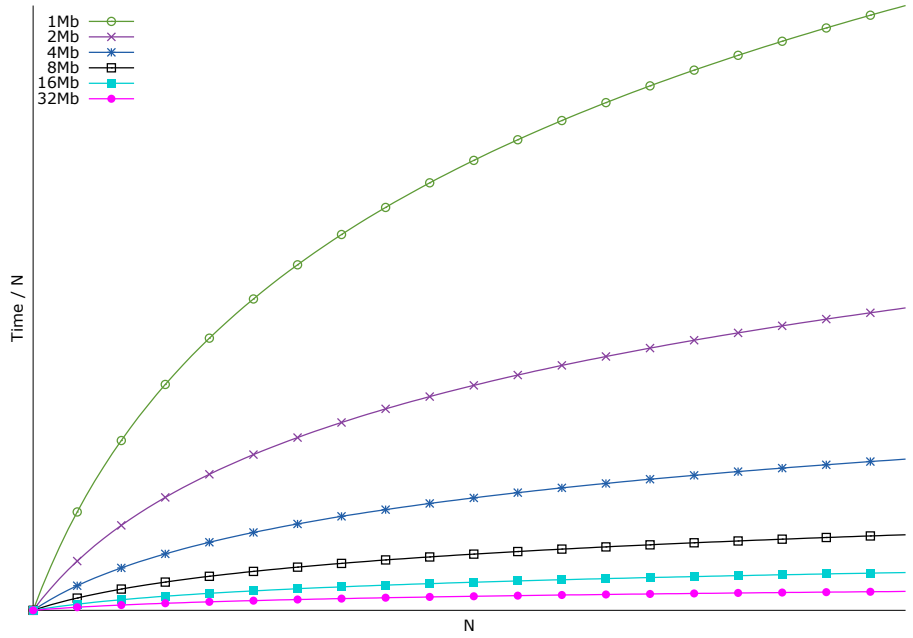

Figure 11.1: Theoretical asymptotic update time per operation for buffer sizes $B \in \{1Mb, 2Mb, 4Mb, 8Mb, 16Mb, 32Mb\}$. *Each graph is on the form* $f(N) = \frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N$ *for* $\varepsilon = \log(2)/\log(B)$. *An epsilon on this form guarantees a fanout* $B^\varepsilon = 2$.

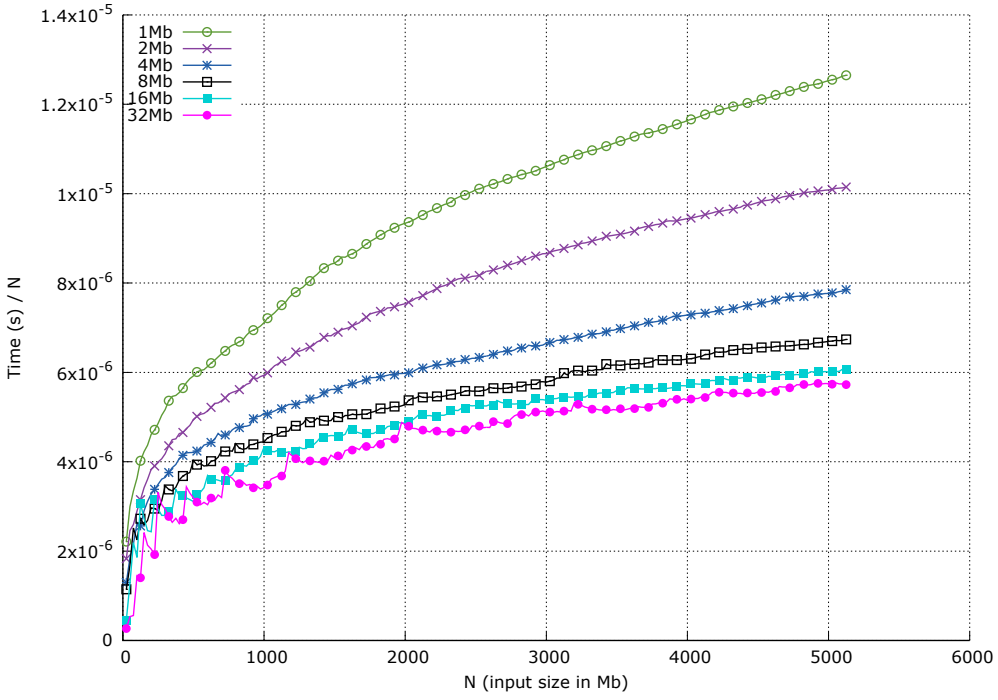

Figure 11.2: Experimentally measured update time per operation for buffer sizes $B \in \{1Mb, 2Mb, 4Mb, 8Mb, 16Mb, 32Mb\}$ *with fanout 2. The tendencies align with the theoretical update bounds depicted in Figure 11.1*

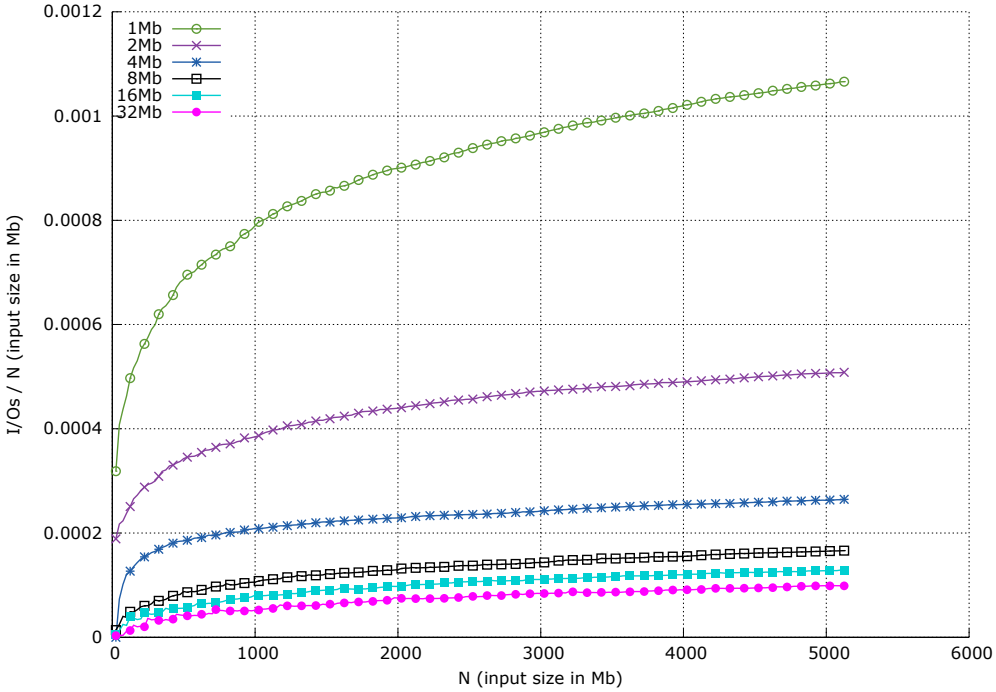

Figure 11.3: Experimentally measured I/O's per insert for buffer sizes $B \in \{1Mb, 2Mb, 4Mb, 8Mb, 16Mb, 32Mb\}$ with fanout 2. The tendencies align with the theoretical update bounds depicted in Figure 11.1 and the actual update times depicted in Figure 11.2.

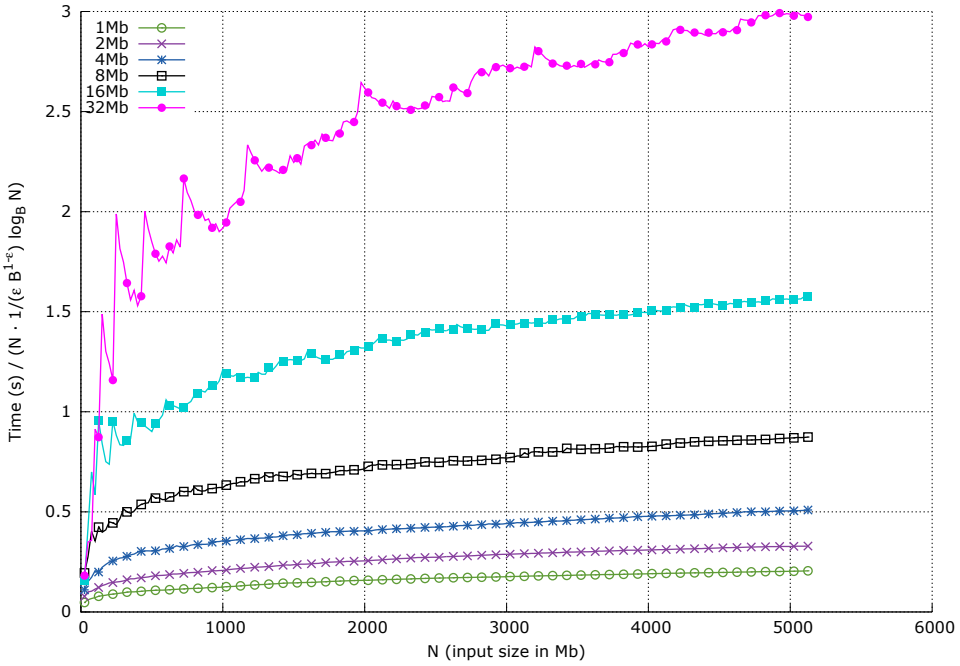

Figure 11.4: Actual running time per update divided by the theoretical number of I/O's per update. A horizontal line suggests a close relation between the two measures. The fluctuations for buffer size $B = 32Mb$ can be explained by the operating system starting to swap out internal memory.

Fanout

To tune the fanout parameter we fixed the buffer size to 8Mb and varied the fanout parameter. The reader's intuition might deceive him into concluding an ever increasing fanout would cause an ever worsening of the update time. This was certainly what we had expected as we pass down an ever decreasing B/B^ε fraction, and each such fraction charges an I/O to the total running time. In Figure 11.5 we present the theoretical update time per insert. We see a tendency of ever increasing fanouts causing an ever decreasing performance for $B^\varepsilon \in \{4, 5, 6, 8, 16, 32, 64\}$. When zooming in on the graphs for fanouts $B^\varepsilon \in \{2, 3, 4, 5\}$ in Figure 11.6 we see that no theoretical gain is to be expected when decreasing the fanout from 3 to 2. In fact it seems we achieve the exact same update time per insert for fanout 2 and fanout 4. This is, however, not as surprising as one might think. See Equation 11.1 and Equation 11.2 where the theoretical bound is shown to be the exact same for fanouts 2 and 4.

Let $B^\varepsilon = 2$:

$$\frac{1}{\varepsilon \cdot B^{1-\varepsilon}} \log_B N = \frac{2 \log_2(B)}{B \log_2(2)} \log_B N \tag{11.1}$$

Let $B^\varepsilon = 4$:

$$\begin{aligned} \frac{1}{\varepsilon \cdot B^{1-\varepsilon}} \log_B N &= \frac{4 \log_2(B)}{B \log_2(4)} \log_B N \\ &= \frac{2 \log_2(B)}{B \log_2(2)} \log_B N \end{aligned} \tag{11.2}$$

What Equation 11.1 and Equation 11.2 essentially states is that the amortized cost of sending double the amount of points down per overflow in a tree of double the height yields no performance gain. Drawing from this conclusion we expect the performance of fanout 2 to be equal to that of fanout 4.

The results showing the experimentally measured update time per insert on varying fanout are depicted in Figure 11.7. We see that the actual running time aligns with what the theoretical number of I/O's per insert from Figure 11.5 suggests. If we are truly I/O bound we expect the theoretical number of I/O's and the actual running time to align well with with the actual number of I/O's per insert. We are pleased to see this is in fact the case in Figure 11.8.

To support our claims of the measured running time and number of I/O's aligning well with theory, we have divided the actual result with the expected in Figure 11.9 for the I/O's and in Figure 11.10 for the running time. We conclude there is a close relation between the actual measures and what the theory suggests, as we see close-to horizontal lines in both plots.

If we zoom in on the measured running time for fanouts 2, 3, 4, 5, and 6 we see minor inconsistencies from what the theory suggests. Comparing the measured running time per update in Figure 11.11 to the expected number of

I/O's per update in Figure 11.6 we see the tendencies align, but the order of the fanouts are inconsistent. For example we would have expected fanout 2 to perform the same as for fanout 4, and surely we had expected fanout 3 to perform the best of them all. This is not the case.

We believe these inconsistencies can be explained by the one parameter not encapsulated in the I/O model; the amount of internal work being done. Theory suggest we see a decreasing number of node degree overflows on increasing fanouts, and in turn we expect less node degree overflows to produce fewer point buffer underflows. These expectations align well with our measures on the total number of point buffer underflows depicted in Figure 11.12. But the I/O model does not account for the internal work needed to handle the actual point buffer underflow. In fact we regard this exact operation to be the most expensive with regard to internal work. Running a profiler on the structure for different fanouts showed us the exact distribution of time spent in different parts of the code. The profiler showed that for smaller fanouts we spent more time doing point buffer underflows, relative to the other operations, than we do for larger fanouts. Refer to Figure 7.7 for a detailed description of the underflow procedure. We claim that in the area of fanouts from 2-6 we are highly influenced by the internal work needed for handling the many point buffer underflows to a point where we do not benefit from the large B/B^{ε} fraction of points we pass down on each overflow.

The results suggest that a fanout of size 5 give the best performance. The results presented was, however, not generated from the machines presented in Chapter 10. The results of these machines show that with a buffer size of 8Mb it would be best to use a fanout size of 2. As the rest of the experiments will be run on the machines described in Chapter 10 we will use a fanout of size 2 going forward.

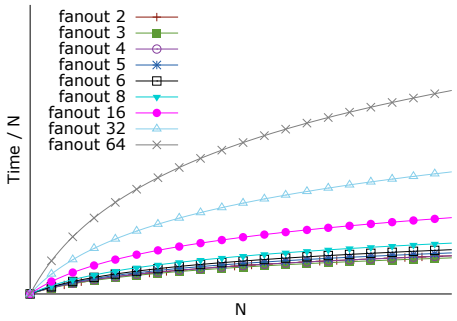

Figure 11.5: Theoretical asymptotic update time per operation for fanouts $B^\varepsilon \in \{2,3,4,5,6,8,16,32,64\}$. Each graph is on the form $f(N) = \frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N$ for $\varepsilon = \log(fanout)/\log(B)$. An epsilon on this form guarantees the desired fanout.

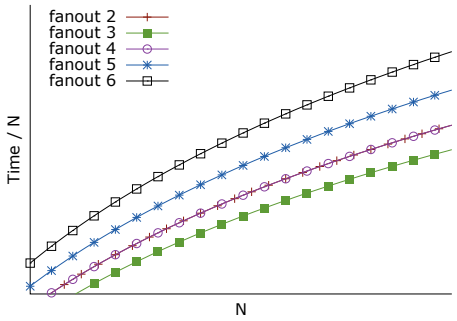

Figure 11.6: Zoomed-in plot of Figure 11.5 depicting the asymptotic update time per operation for fanouts $B^\varepsilon \in \{2,3,4,5,6\}$. Note that fanouts 2 and 4 gives rise to the exact same graphs. This can be explained by Equation 11.1 and 11.2.

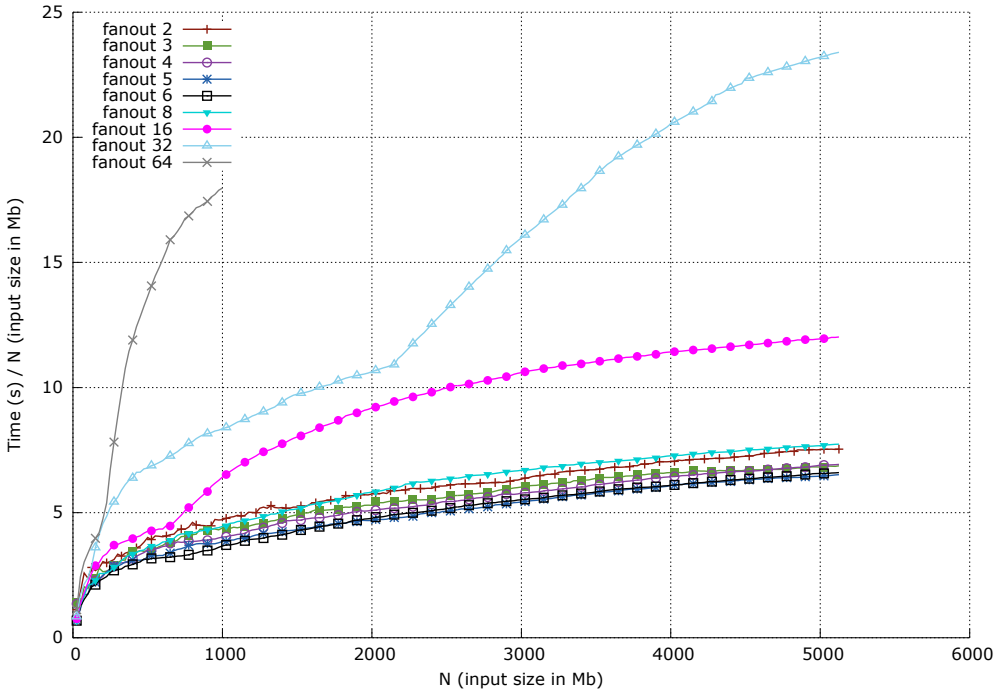

Figure 11.7: Experimentally measured update time per insert for fanouts $B^\varepsilon \in \{2,3,4,5,6,8,16,32,64\}$ with buffer size $B = 8Mb$. The tendencies align with the theoretical update bounds depicted in Figure 11.5.

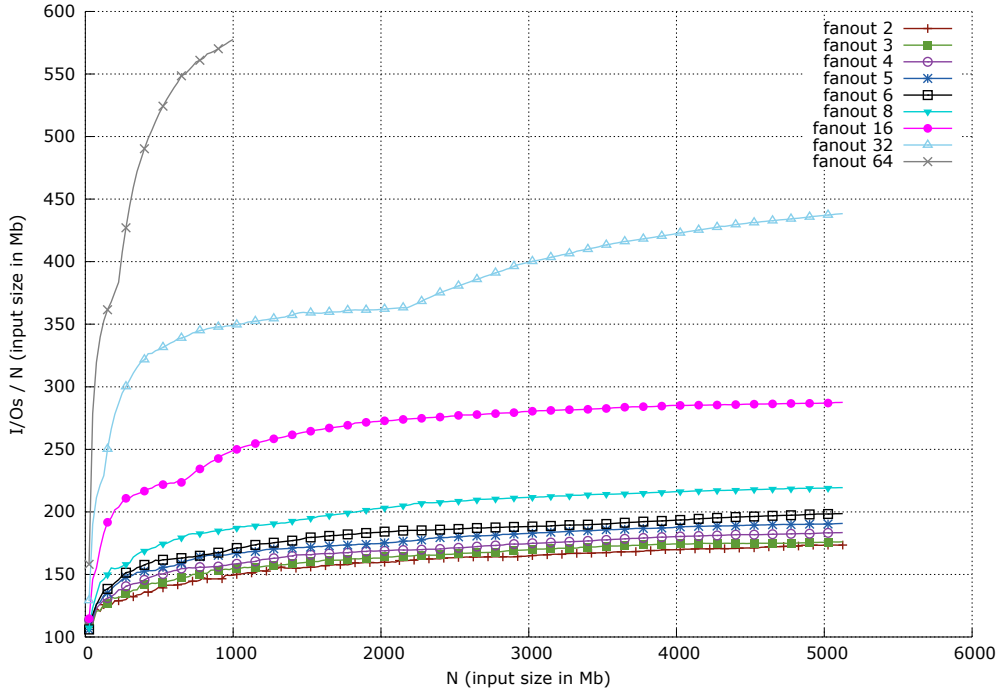

Figure 11.8: Experimentally measured number of I/O's per insert as a function of input size with a fixed buffer size of 8Mb and varying fanouts $B^\varepsilon \in \{2,3,4,5,6,8,16,32,64\}$. The tendencies align with the theoretical update bounds depicted in Figure 11.5.

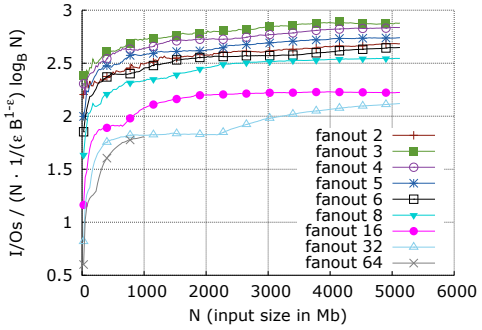

Figure 11.9: Experimentally measured number of I/O's divided by the theoretical number of I/O's per insert. A horizontal line suggests a close relation between the two measures.

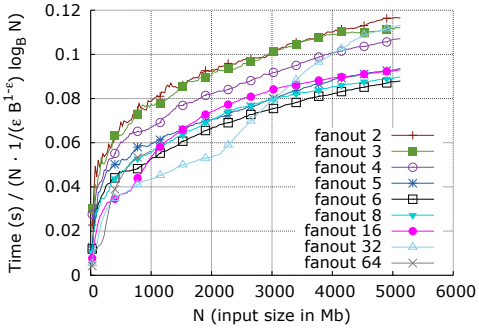

Figure 11.10: Experimentally measured running time per insert divided by the theoretical number of I/O's per insert. A horizontal line suggests a close relation between the two measures.

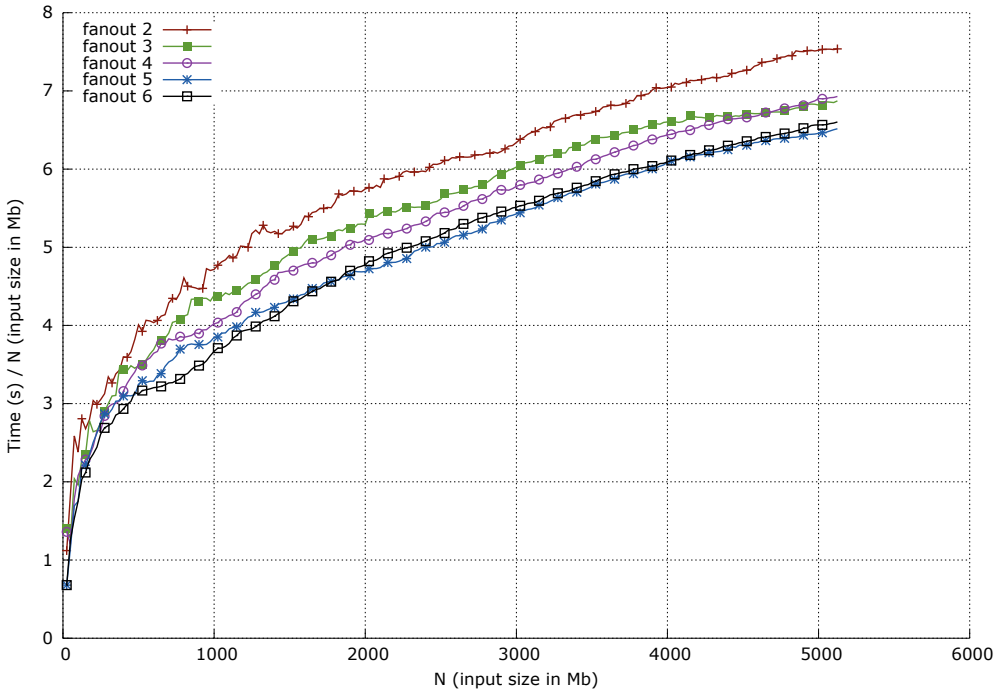

Figure 11.11: Zoomed-in plot of experimentally the measured update time per insert for fanouts $B^\varepsilon \in \{2,3,4,5,6\}$. Comparing these to the theoretical number of I/O's per insert depicted in Figure 11.6 we see minor divergences. This can be explained by the internal work done on point buffer underflows.

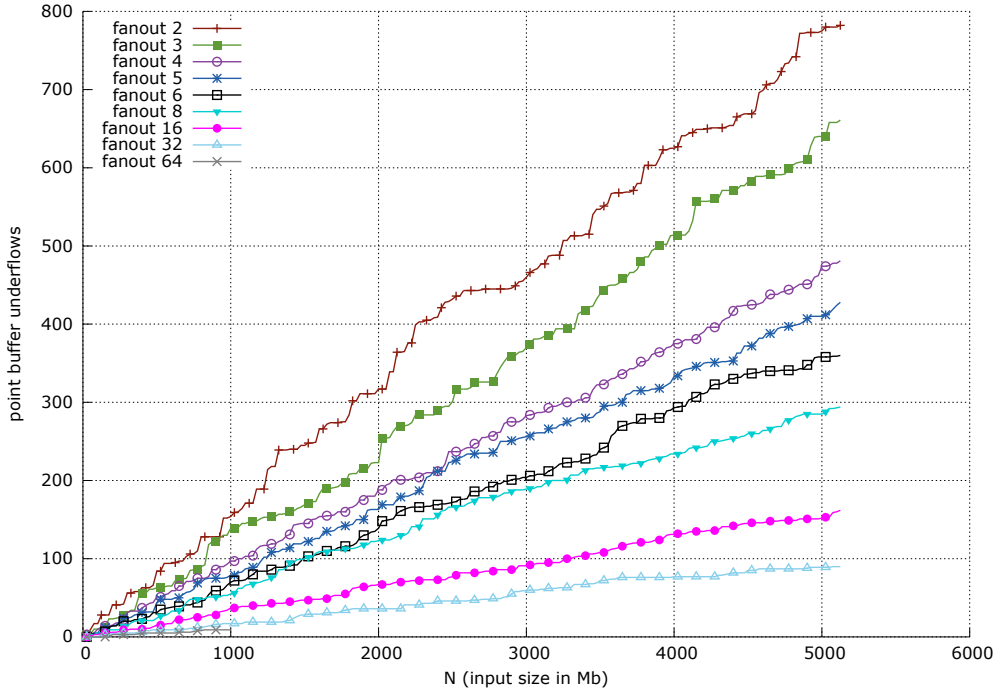

Figure 11.12: Experimentally measured number of point buffer underflows. We see a tendency of an ever decreasing number of point buffer underflow for ever increasing fanouts.

Summary

We conclude from our above experiments with different fanouts and buffer sizes that in order to get as fast an update time as possible we need a fanout of between 2-6 and as large a buffer size as internal memory allows.

It is worth noting that, since the experimental results are very much in line with the theoretical results, it can be concluded that the data structure is I/O bound.

Going forward we will use a buffer size of 8Mb together with a fanout of 2.

11.1.2 External Memory Priority Search Tree

The experiment consisted of inserting 50 Mb of data, i.e. around 6.55 million data points. The coordinates of the data points was chosen uniformly distributed among the positive integers.

The results of the experiment are depicted in Figure 11.14. We are very far away from the expected update time per insert depicted in Figure 11.13. The main reason behind this is most likely that more data is needed in the structure for the theoretical $\mathcal{O}(\log_B N)$ I/O per update to be apparent. It would seem we are dominated by the fact we need to load and store B data points on each node visited in order to handle an update. Since the experiment was very time consuming, we decided to elect a buffer size of 4Kb as the winning buffer size. We will elaborate more on the main bottlenecks of the structure in the insert experiment found in Section 11.2.

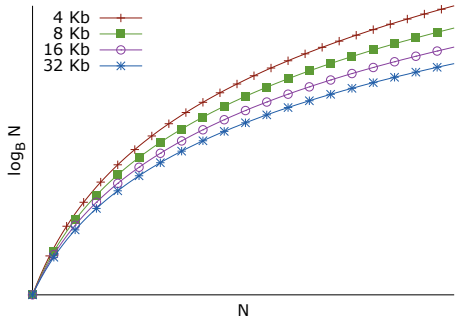

Figure 11.13: Theoretical asymptotic update time per insert for buffer sizes $B \in \{4Kb, 8Kb, 16Kb, 32Kb\}$. Each graph is on the form $f(N) = \log_B N$.

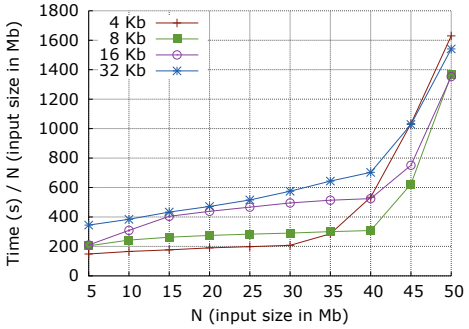

Figure 11.14: Experimentally measured update time per insert for buffer sizes $B \in \{4Kb, 8Kb, 16Kb, 32Kb\}$. Note the relatively long running time needed for inserting 50 Mb data.

11.2 Insertion

The goal of this experiment is to ascertain how the different structures compare when it comes to inserting points. The experiment consists of inserting as much data as possible within 24 hours. The data is uniformly distributed in the positive integer range.

Figure 11.15 shows the theoretical complexities of inserting an element into the different structures for different input sizes. We expect the data structures not optimized for external memory to align close to the theoretical bounds while completely contained in internal memory. Only when the operating system is forced to swap data between internal and external memory, do we expect a significant decrease in performance. When the data structures are no longer able to fit into internal memory, and since they all rely on scattered data access, we expect an amount of page faults close to the theoretical asymptotic complexity. For the external memory data structures we expect the MySQL implementation without an index to outperform all of the other structures, since inserting essentially just appends points to a file. We expect the effective buffering of points in the External Memory Buffered Priority Search Tree by Brodal to outperform the R-Tree variants and MySQL with an index on coordinates.

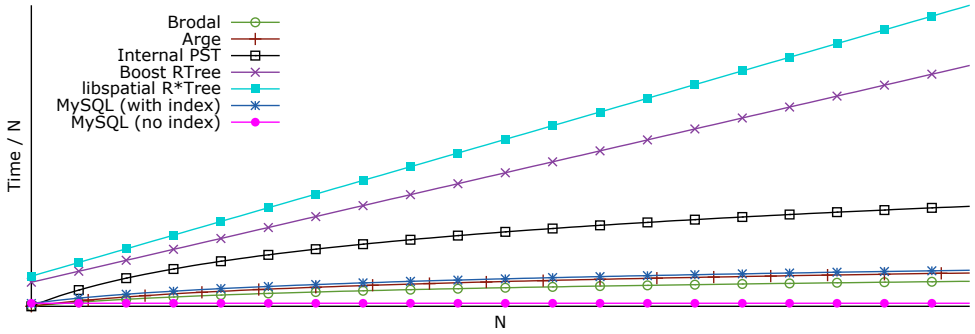

Figure 11.15: Theoretical asymptotic insert time for all tested structures. Brodal's is on the form $f(N) = \frac{1}{B^{1-\varepsilon}} \log_B N$ *for* $\varepsilon = \frac{\log(B)}{\log(2)}$*. Arge's and MySQL (with index) are on the form* $f(N) = \log_B N$*. The Internal Memory Priority Search Tree is on the form* $f(N) = \log_2 N$*. The MySQL (no index) is on the form* $f(N) = 1$*, and the Boost R-Tree and Libspatial R*-Tree are on the form* $f(N) = N$*.*

Figure 11.16 shows the actual running time per inserted megabyte in all of the tested structures. The figure is cropped at $N = 600$ to better present the relation between the internal memory and the external memory structures. MySQL without an index was able to insert more than 10Gb in less than 3 hours. We decided to stop the experiment prior to the time limit since there was no change in running time per inserted megabyte. The External Memory Buffered Priorty Search Tree by Brodal was able to insert around 3.5Gb worth of data within the time limit.

The internal memory data structures performs very well while contained in memory. It can be seen in Figure 11.17 that there is a close relation between

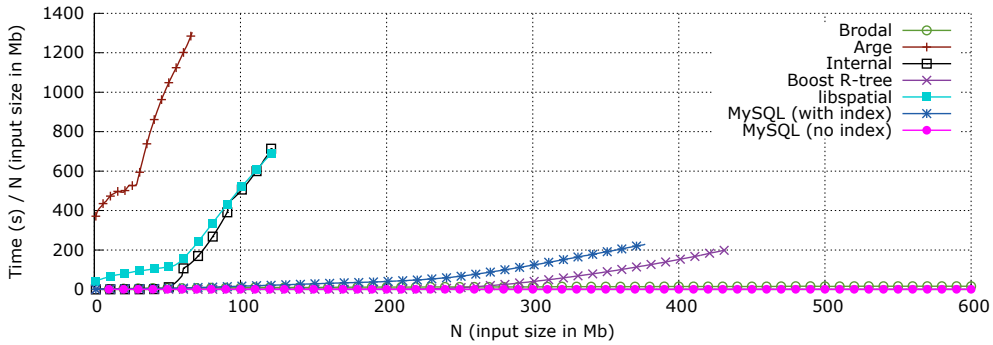

Figure 11.16: Actual running time when inserting into the tested structures. The figure shows the running time per insert for the first 600Mb.

the running time and the number of page faults. We see the running time increase significantly at the same input size as where the number of page faults increases.

It is obvious that the data structure of Arge performs the worst when it comes to inserting data. We believe this is due to the huge amount of data we need to load and store for inserting just a single point. We will in the following argue this in a more precise manner. It follows from the theory that a single insert requires $\mathcal{O}(\log_B N)$ I/O's. Each data point uses 8 bytes of space, 4 bytes for each coordinate. This means that in order to insert 50Mb data or equivalently 6,553,600 points in an initially empty structure with a buffer of size 4Kb we need roughly

$$\sum_{i=0}^{6,553,600} \log_{4096}(i) \approx 11.5 \cdot 10^6 \text{ I/O's}$$

Comparing this against the structure of Brodal with the same buffer size and a fanout of size 2, we get that Brodal's requires roughly a factor B less:

$$\sum_{i=0}^{6,553,600} \frac{1}{\varepsilon \cdot 4096^{1-\varepsilon}} \log_{4096}(i) \approx 67.8 \cdot 10^3 \text{ I/O's}$$

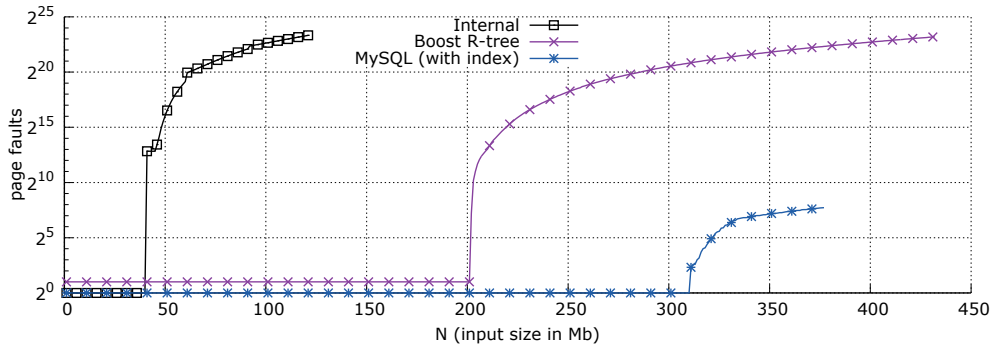

Figure 11.17: Number of page faults generated by the internal memory data structures when inserting points.

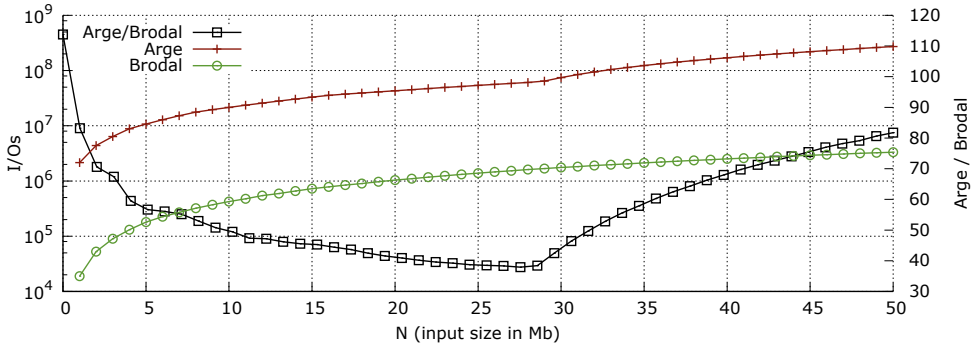

Figure 11.18: The number of I/O's for the structure of Arge and Brodal respectively. The y-axis on the right shows the relative difference in terms of I/O's between Arge's and Brodal's. It shows that Arge's generates 81 times more I/O's than Brodal's when inserting 50Mb.

The calculations shows that in order to insert 50Mb of data, we need to load/store $11.5 \cdot 10^6 \cdot 4096 = 47.1$Gb in the case of Arge's, and $67.8 \cdot 10^3 \cdot 4096 = 0.278$Gb in the case of Brodal's. This gives a relatively difference of around 170 times fewer I/O's performed in Brodal's over Arge's.

To corroborate these observations we conducted a single experiment measuring the I/O's used when inserting 50Mb of data. The results can be found in Figure 11.18. The results shows that the amount of I/O's performed is substantially larger for Arge's than for Brodal's. In fact we can see that in order to insert 50Mb we must perform I/O's equal to moving around 1Tb of data in Arges' and around 13Gb of data in Brodal's. This is a relative difference of around 81 times fewer I/O's in Brodal's compared to that of Arge's. This result is close to what we would suspect from the theoretical reasoning above, however with a constant factor between theory and reality.

Figure 11.19-11.23 shows the time per insert divided by the theoretical asymptotic bound of each of the tested structures. For the external memory data structures we also display the number of I/O's per insert divided by the theoretical asymptotic bound and for the internal memory structures we include the page faults per insert divided by the theoretical asymptotic bound.

If the actual running time align with the theoretical bounds then we expect to see the graphs form horizontal lines. The caveat is that the internal memory data structures will start to swap out data when they have no more free internal memory to use, which again causes the running time to increase severely. If we see a spike in both running time and number of page faults at the same mark then we feel confident the two measures are closely related.

In Figure 11.19 we see that the time per insert in the structure of Brodal aligns with the theoretical bound. This can be seen by the flattening of the graph as the input size goes up, which again suggests the structure becomes more and more I/O bound on increasing input size. The structure follows the same trend in terms of I/O's which further strengthens this hypothesis.

In Figure 11.20 we see that the time per insert and number of I/O's per insert follows the theoretical bound very well up to the $N = 28$ mark. We in-

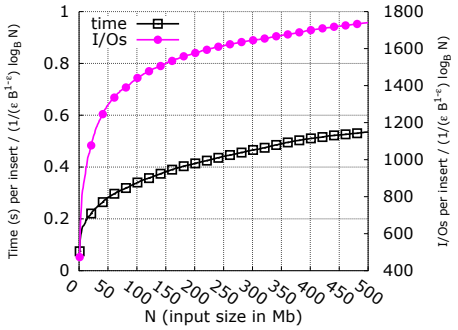

Figure 11.19: Time and I/O's per insert divided by $\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N$ for the structure of Brodal.

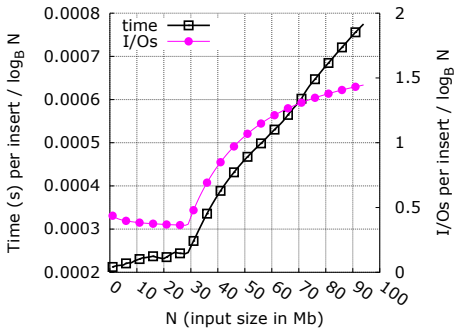

Figure 11.20: Time and I/O's per insert divided by $\log_B N$ for the structure of Arge et al.

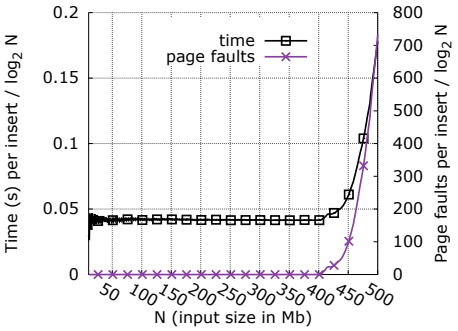

Figure 11.21: Time and page faults per insert divided by $\log_2 N$ for the Internal Memory Priority Search Tree of McCreight.

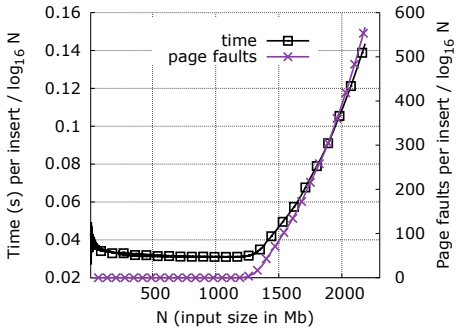

Figure 11.22: Time and page faults per insert divided by $\log_{16} N$ for the Boost R-Tree.

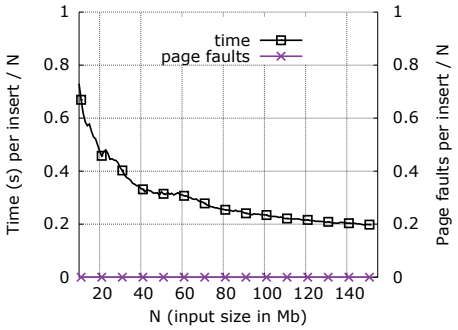

Figure 11.23: Time and page faults per insert divided by N for the libspatial external R-Tree.*

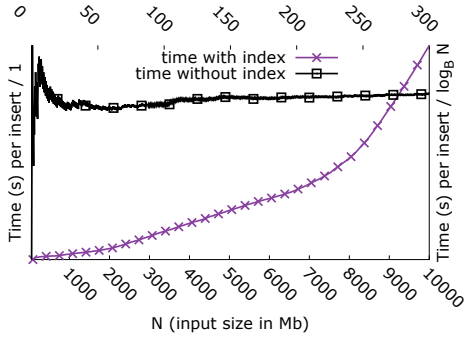

Figure 11.24: Time per insert divided by 1 for MySQL without an index and time per insert divided by $\log_B N$ for MySQL with an index.

vestigated this further and found this was the exact moment when the structure went from a B-Tree of height 2 to one of height 3. It would seem that adding a layer significantly increased the amount of I/O's. Further investigation of the trends for I/O's we see the graph flattening again after the $N = 28$ mark, which provides evidence for the number of I/O's once again becomes a constant factor of the theoretical bounds. The same cannot be concluded for the actual running time. We would have expected a flattening along the same lines as for the number of I/O's but this is clearly not the case. Once again we conclude this is caused by the amount of data we need to load and store to handle just a single insert.

Figure 11.21 shows that the Internal Memory Priority Search Tree of McCreight performs exactly as expected as long as the structure is contained in internal memory. At the exact moment the structure starts to swap out internal memory we see, just as expected, a significant increase in running time.

It is almost the same situation for the results of the Boost R-Tree depicted in Figure 11.22. The insertion time in an R-Tree heavily depends on heuristics which explains the decreasing graph and far from worst case behaviour. Again we see the same behaviour as for the structure by McCreight when we run out of internal memory – the running time suffers tremendously.

The results for the libspatial external R*-Tree depicted in Figure 11.23 shows the running time follows the expected theoretical bounds very well. We would have liked to measure the number of I/O's for this structure but since it is a library implementation this was infeasible. We are pleased to see that the implementation does not generate any page faults. This is what we would expect from a sound external memory data structure.

Figure 11.24 shows both the running time of MySQL with an index and without an index both divided by the asymptotic running time, which for the case of the indexed version is $\mathcal{O}(\log_B N)$ and the non-indexed version is $\mathcal{O}(1)$. The results show that MySQL without an index follows the asymptotic running time very closely while the indexed version becomes steeper and steeper. Digging into the MySQL implementation of how indices and B-Trees are used we found that MySQL uses a Red-Black tree to store data in internal memory and then bulk insert the sorted data into a B-Tree whenever the Red-Black tree overgrows internal memory. We believe the increases in running time are caused by this bulk unloading into a B-Tree[1]. We deemed it out of scope for this thesis to look closer into the inner workings of the MySQL index structure.

[1] `https://dev.mysql.com/doc/internals/en/bulk-insert.html`

11.3 Deletion

It was difficult to come up with a good experiment to compare the different structures when it comes to deletion. In order to delete data we need to insert it first, and we need to insert equally much data in all structures to compare fairly. As described in the previous section it was not possible to insert much data in the internal structures as well as the structure of Arge et al. To come around this problem we decided to insert nothing more than 50 megabytes worth of uniformly distributed points in all the structures. We could then completely empty the structures while measuring for each megabyte. This way we hope we can exclude structures from further analysis.

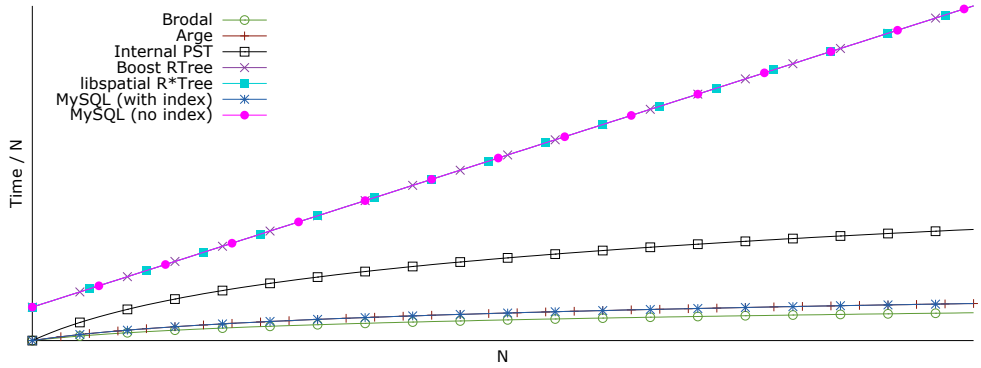

Figure 11.25: Theoretical asymptotic delete time for all tested structures. Brodal's is on the form $f(N) = \frac{1}{B^{1-\varepsilon}} \log_B N$ *for* $\varepsilon = \frac{\log(B)}{\log(2)}$*. Arge's and MySQL (with index) are on the form* $f(N) = \log_B N$*. The Internal Memory Priority Search Tree is on the form* $f(N) = \log_2 N$*. The MySQL (no index), Boost R-Tree and Libspatial R*-Tree are on the form* $f(N) = N$*.*

Figure 11.25 shows the theoretical complexity of deletion in the structures. We expect the internal memory algorithms to outperform the external structures significantly while still in internal memory. As soon as we outgrow internal memory we expect them to become obsolete. In external memory we expect the structure by Brodal to be the best due to the effective buffering of deletes.

First results

The results of the above described experiment are depicted in Figure 11.26. We have left out the results from the structure of Arge as we were unable to delete more than 2Mb worth of data within 24 hours, and thus concluded it would be infeasible to finish the experiment within reasonable time. We believe the poor performance of Arge's structure is due to similar reasons as described in Section 11.2.

In Figure 11.26 we see some, at first glance, strange behaviour on the running time of the Internal Memory Priority Search Tree. It seems we are achieving ever increasing running time on the first 9Mb worth of deletions, and then, after seeing a significant spike, we are suddenly achieving very

good running times on the emptying of the rest of the data structure. This is, however, not as surprising at it might seem. Remember, we are making use of global rebuilding in our implementation of the Internal Memory Priority Search Tree. We do this by *marking* any deleted place-holder, instead of *removing* the actual place-holder from the data structure. This is also known as a *weak* delete of an element. Only when we initiate a global rebuild on the non-deleted points are we freeing the occupied memory of deleted elements[2]. What we are observing are measures on an ever decreasing data structure in terms of non-deleted data. But the deleted data is still part of the tree, and thus it affects space usage, and again running time of our delete procedure. We are global rebuilding exactly around the spike at the 41Mb mark. We claim that the Internal Memory Priority Search Tree on 50Mb data is unable to fit in internal memory, and what we see is the effect of the swapping of internal memory to external memory. Only when we global rebuild to a data structure holding around 40Mb worth of data are we able to process the data structure entirely in internal memory. We believe Figure 11.27 depicting the number of measured page faults supports this claim.

Narrowing the field

Arge's structure, the Libspatial R*-Tree, and the Internal Memory Priority Search Tree performs the worst of all the data structures when it comes to handling deletions. Excluding these gives rise to Figure 11.28. It is obvious that, even with an index on coordinates, the MySQL implementations are performing much worse than the Boost R-Tree and the data structure of Brodal. We suspect the good performance of the Boost R-Tree is due to the effective space-usage and thus the data structure is able to process the deletions entirely in internal memory. In order to verify this, we re-ran the experiment only on Brodal's data structure and the Boost R-Tree on larger input.

Finding the winning data structure

The result of emptying Brodal's and the Boost R-Tree from a size of 400Mb down to a size of 350Mb worth of data is depicted in Figure 11.29. We now see the data structure of Brodal's handling deletions several orders of magnitude faster than the Boost R-Tree. We claim this is caused by the fact that the Boost R-Tree cannot fit into internal memory, and thus have to rely on the operating system handling swapping of data. Figure C.2 depicting the number of page faults for the experiment supports this claim.

Explaining fluctuations

Finally we zoom in on the result of completely emptying Brodal's data structure from a size of 400Mb of data in Figure 11.30. We see severe fluctuations

[2]We could, without having broken correctness of the implementation, have deleted place-holders, but that was an observation made in hindsight, and we did not have the time to make this change in code.

in running time during the emptying of the data structure. We believe these fluctuations are perfectly explained by the number of point buffer underflows depicted in Figure 11.31 and the number of delete buffer overflows depicted in Figure 11.32.

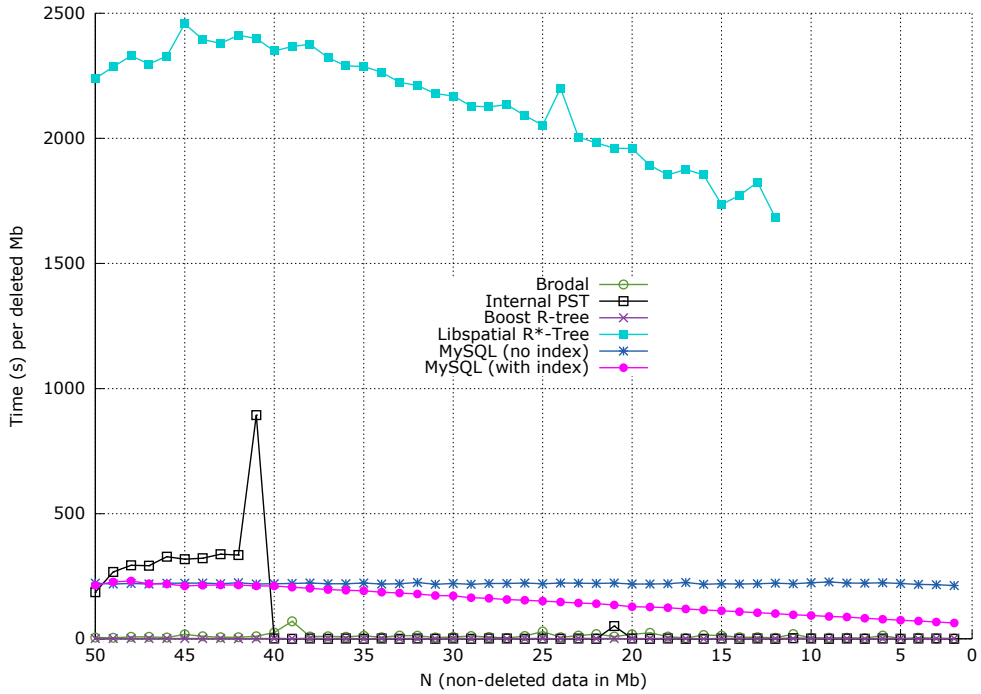

Figure 11.26: Experimentally measured running time when deleting 1Mb worth of data as a function of the remaining data in the structure going from 50Mb to 0Mb.

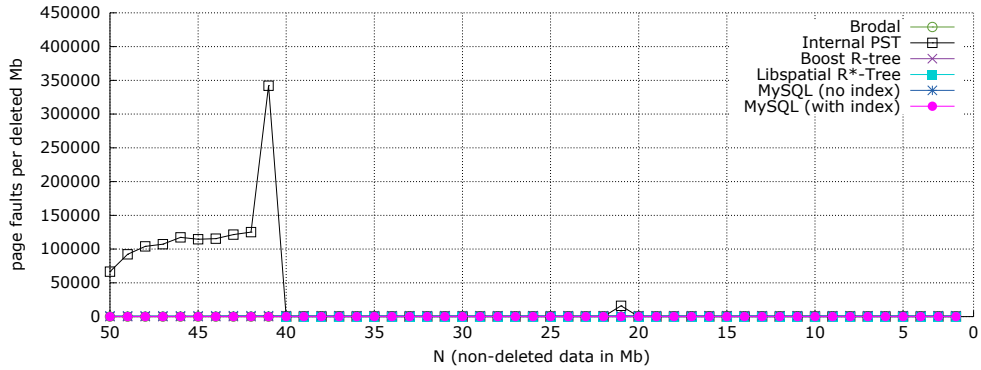

Figure 11.27: Experimentally measured page faults when deleting 1Mb worth of data as a function of the remaining data in the structure going from 50Mb to 0Mb.

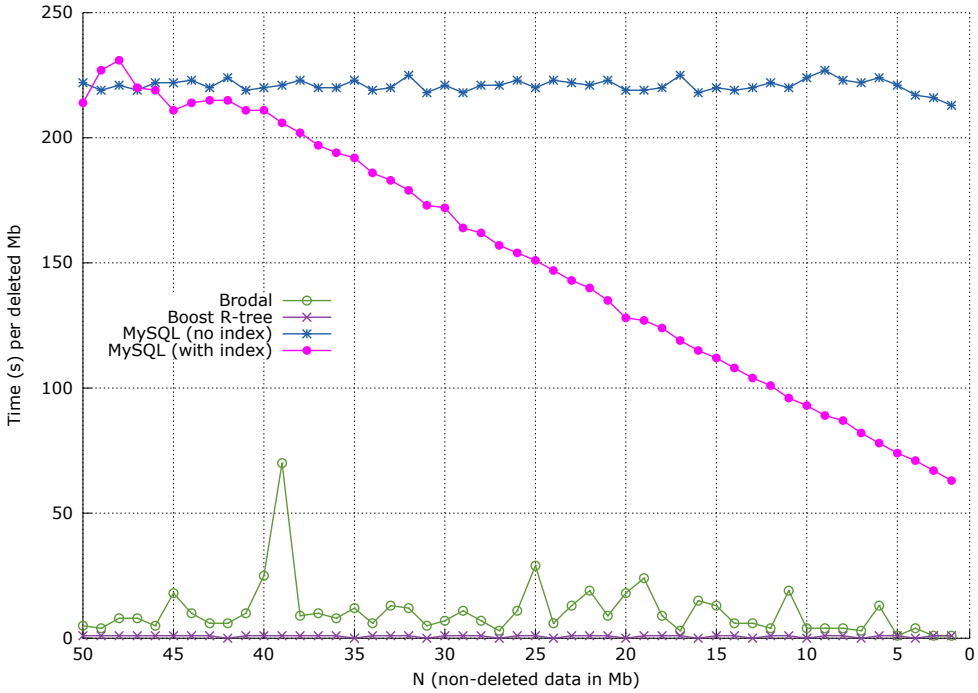

Figure 11.28: Experimentally measured running time when deleting 1Mb worth of data in a data structure holding from 50Mb to 0Mb non-deleted data. The Boost R-Tree performs the best as it can handle all deletions in internal memory.

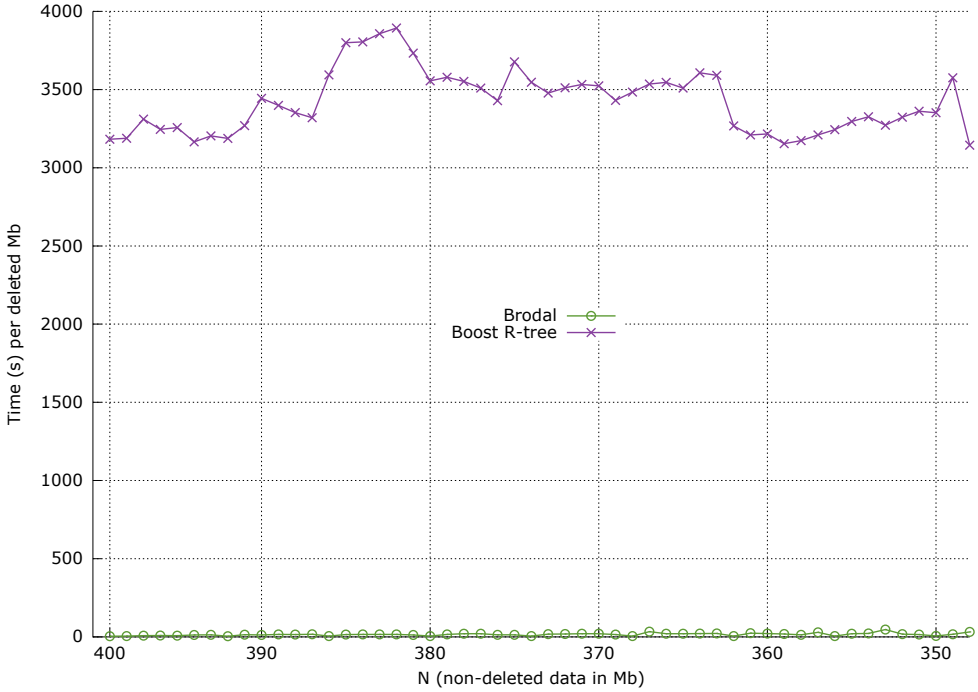

Figure 11.29: Experimentally measured running time when deleting 1Mb worth of data in a data structure holding from 400Mb to 350Mb of non-deleted data. The Boost R-Tree now suffers from swapping of data (see Figure C.2).

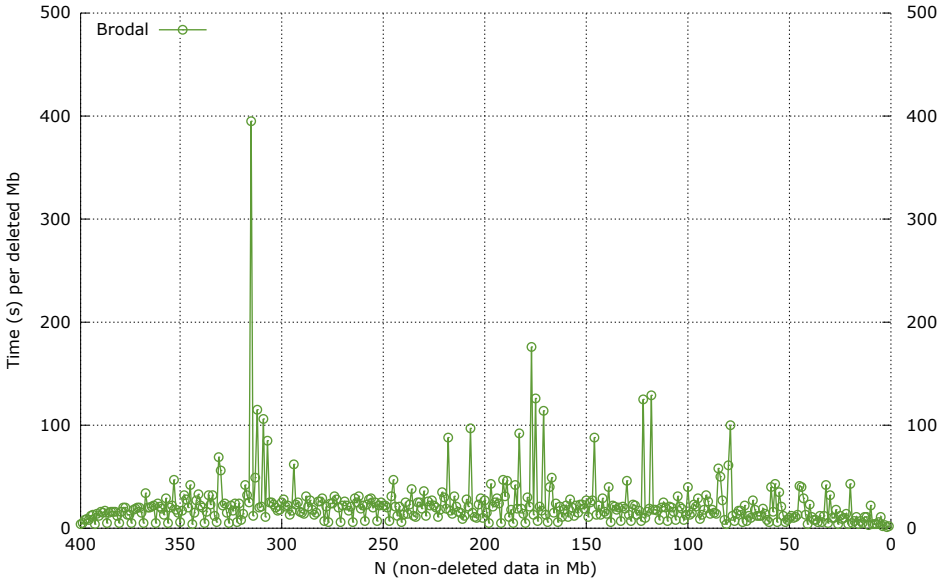

Figure 11.30: Experimentally measured running time when emptying Brodal's from a size of 400Mb data. We see fluctuations in the running time. These are perfectly explained by the number of point buffer underflows depicted in Figure 11.31 and the number of delete buffer overflows depicted in Figure 11.32.

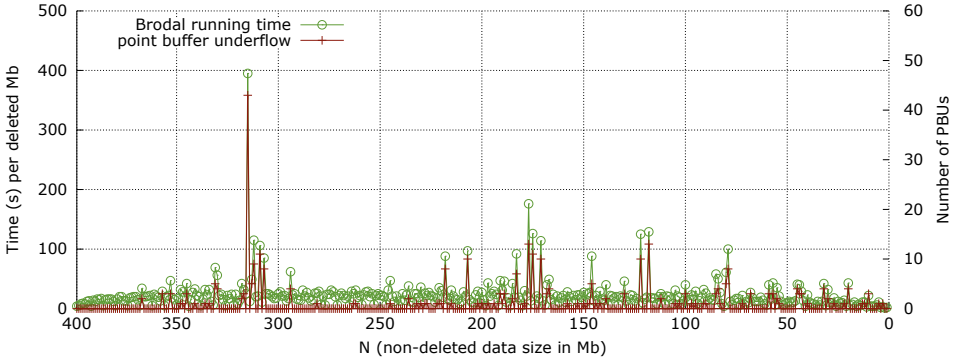

Figure 11.31: Experimentally measured number of point buffer underflows when emptying Brodal's from a size of 400Mb data.

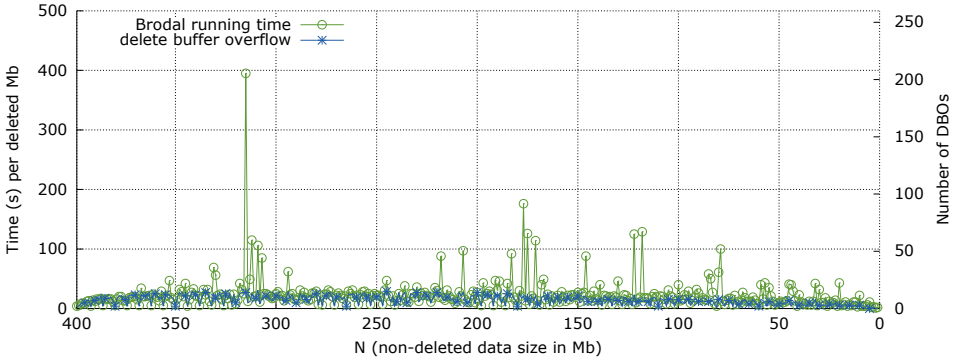

Figure 11.32: Experimentally measured number of delete buffer overflows when emptying Brodal's from a size of 400Mb data.

11.4 Three-sided range queries

As the theoretical query bounds are analysed using the method of filtering, we conduct experiments that focus on both the *search* and the *report* part of the algorithms. Please refer to Section 3.3 for a description of the method of filtering.

11.4.1 Focus on searching

In this subsection we try to focus on searching, i.e. remove the reporting part of the query complexity, by fixing the output to a fixed constant number of points.

First experiment

The first experiment on the **search** part of the algorithms was conducted by first inserting data *inside* 5 fixed query windows as shown in Figure 11.33 of 5Mb data each, i.e. each query window contains 655,360 points. We use query windows such that we report on both small and large ranges and high and low in the tree. This was followed by inserting uniformly distributed data *outside* the query windows. For every 10Mb of data inserted we report all of the 25Mb points inside the query windows. The experiment was limited to run for 24 hours.

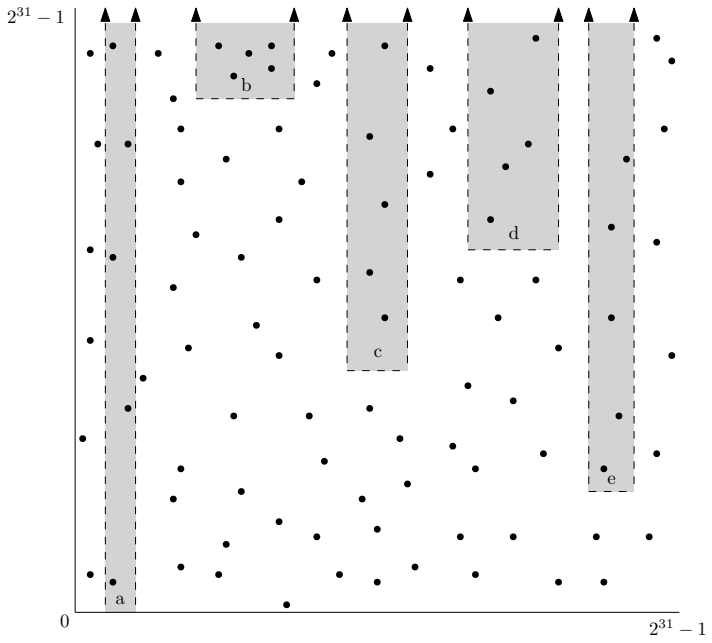

Figure 11.33: Distribution of points for the uniform reporting experiment (search). The gray areas (a) - (e) contains 5Mb data each. Points are distributed uniformly random. The query window (a) spans 5% of the x-axis and 100% of the y-axis (b) spans 17% of the x-axis and 20% of the y-axis (c) spans 11% of the x-axis and 60% of the y-axis (d) spans 14% of the x-axis and 40% of the y-axis (e) spans 8% of the x-axis and 80% of the y-axis.

All queries are performed once before the actual measuring is done. This is to remove fluctuations on the internal data structures and to pay the amortized cost of flushing buffers in the External Memory Buffered Priority Search Tree. The results of the experiment are depicted in Figure 11.35.

The theoretical search complexity on each of the data structures are depicted in Figure 11.34. Even though the theoretical asymptotic bound suggests the R*-Tree to be linear in the size of the input, we expect it to have an actual average running time closer to $\mathcal{O}(\log_B N)$ guaranteed when no bounding boxes overlap each other. We expect the external memory data structures to perform much better than the internal memory data structures when the operating system starts swapping out internal memory.

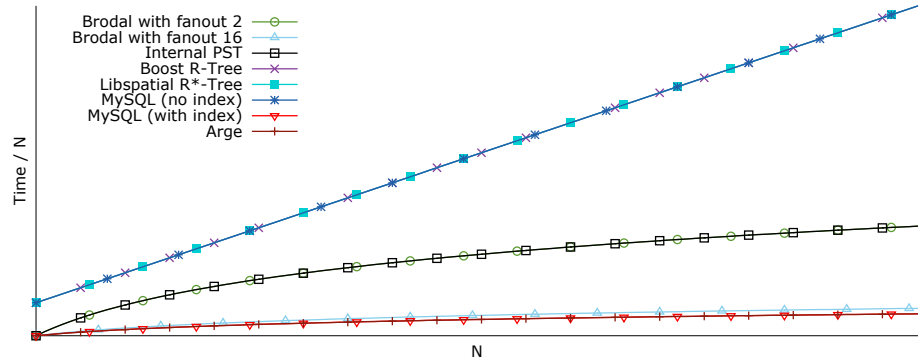

Figure 11.34: Theoretical search complexity for the query operation on all structures. Brodal's is on the form $f(N) = \frac{1}{\varepsilon}\log_B N$ for $\varepsilon = \log_B(fanout)$. The data structure of Arge and MySQL with an index are on the form $f(N) = log_B N$. The Internal Memory Priority Search Tree is on the form $f(N) = \log_2 N$. Both the Boost R-Tree, the non-indexed MySQL implementation and the Libspatial R-Tree are worst case linear, i.e. $f(N) = N$.*

We see that both the Internal Memory Priority Search Tree, the Boost R-Tree, and MySQL with an index suffer when they cannot fit into internal memory. We argue in the following our observations according to the zoomed-in plot depicted in Figure 11.36. It is difficult for us to argue that the libspatial R*-Tree follows our expectations of an average $\mathcal{O}(\log_B N)$ search time in the depicted results. We believe we would have had a better chance of explaining the tendencies on the R*-Tree had we been able to construct a data structure on a larger input size. This was not feasible within a reasonable amount of time. To our surprise we see that the structure of Arge et al. performs best of all the structures, but since we are unable to insert more than around 70Mb of data in the data structure within 24 hours, we conclude the implementation to be of no practical use when querying large data sets. We see that the non-indexed MySQL's running time is linear as expected. It is interesting to see that the data structure of Brodal performs very well even for a small fanout. The data structure aligns with the tendencies suggested by the theory except for minor fluctuations. These are, however, perfectly explained by the node degree overflows needed during fixup as shown in Figure C.3.

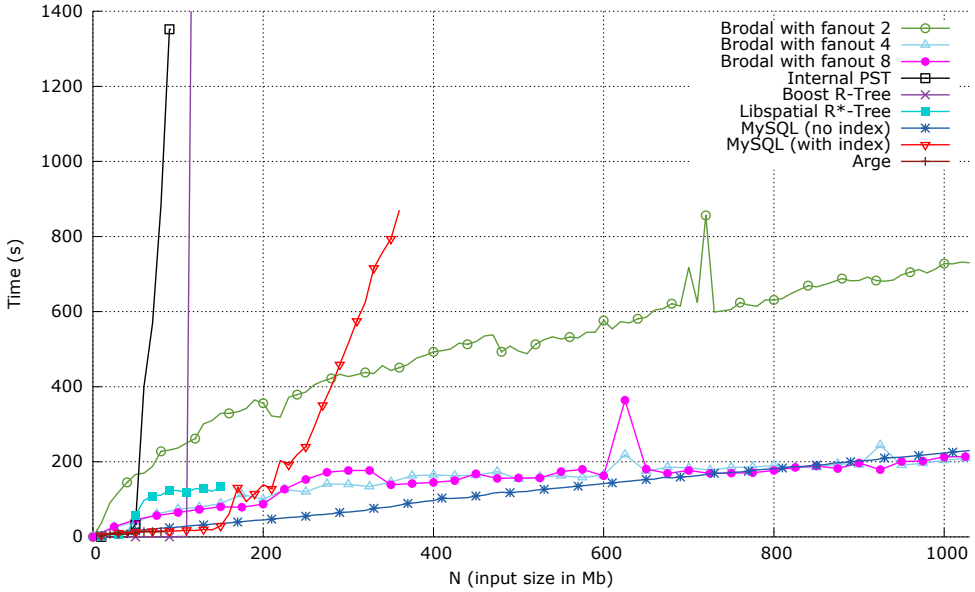

Figure 11.35: Experimentally measured running time for querying using the query windows of Figure 11.33 on all data structures of size N.

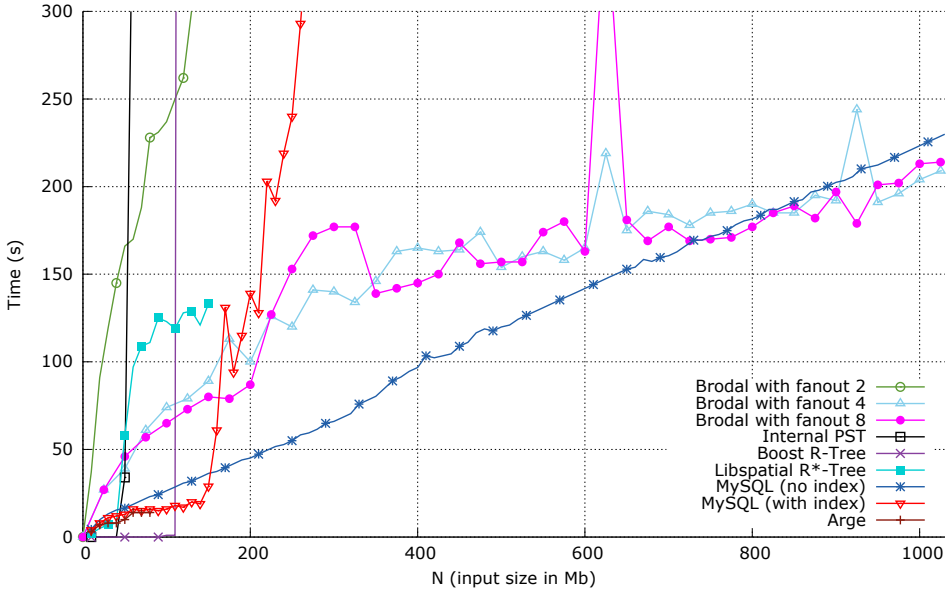

Figure 11.36: Zoomed-in version of the experimentally measured running time for querying using the query windows of Figure 11.33.

Second experiment

The second experiment on the **search** part of the algorithms was conducted by fixing x_1 and x_2 and $y_1, \ldots, y_n$, and inserting data points such that there is exactly 1Mb of data points in the range $[x_1, x_2] \times [y_i, y_{i+1}]$. In the range *outside* $[x_1, x_2] \times [-\infty, \infty]$ we distribute data points uniformly. Now we report points in ranges $[x_1, x_2] \times [y_1, \infty]$, $[x_1, x_2] \times [y_2, \infty], \ldots, [x_1, x_2] \times [y_n, \infty]$. The idea is that the number of reported points K only grows with 1Mb when reporting in the range $[x_1, x_2] \times [y_{i+1}, \infty]$ compared to reporting in the range $[x_1, x_2] \times [y_i, \infty]$.

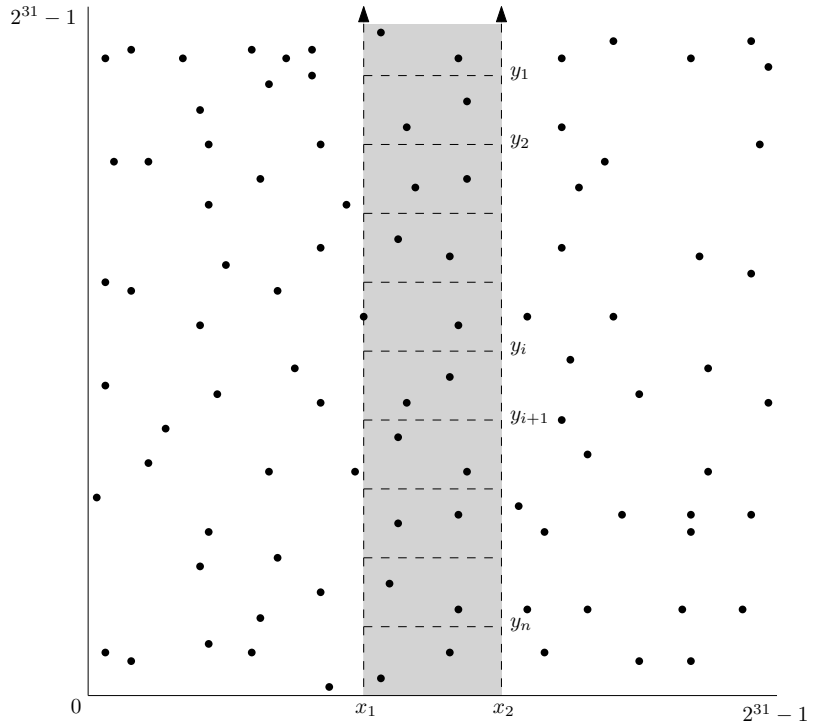

Figure 11.37: Distribution of points for the uniform reporting experiment (searching). The points x_1 and x_2 and $y_1, \ldots, y_n$ are fixed. There is 1Mb data in all ranges $[x_1, x_2] \times [y_i, y_{i+1}]$.

We first conducted the experiment with 50Mb data distributed *inside* the query windows and an additional 50Mb of data distributed *outside* the query windows. The result of this experiment is depicted in Figure 11.38. We see the Internal Memory Priority Search Tree of McCreight suffers severely from swapping of memory, and so we rule this data structure out after having deleted only 2Mb worth of data. Also, it is obvious that the Boost R-Tree is still able to fit entirely in internal memory and achieves a superior running time. The data structure of Arge achieves decent running times, but since the data structure took more than 24 hours to construct we have to rule it out as a candidate for solving huge data sets. Based on this observation we added an additional 300Mb data *outside* the query windows giving a total of 400Mb data in Figure 11.39. We now see that the Boost R-Tree can no longer fit into internal memory which results in a much worse running time than achieved

by the external memory data structures. What is also clear is that MySQL using no index is still superior compared to the data structure of Brodal. The last experiment therefore added an additional 2,100Mb data *outside* the query windows yielding a data structure of a total of 2,500Mb worth of data. The experiment was then run on MySQL with no index and the data structure by Brodal with a fanout of 2 and 4 respectively. The result of this experiment is depicted in Figure 11.40. We now see that the data structure of Brodal with fanout 4 outperforms MySQL which is explained by the fact that it is able to exclude searching in the parts of the tree with no points to report based on the y range of the query. We also see that the External Memory Buffered Priority Search Tree just barely overtake MySQL with a fanout of 2. These results support the tendency of increasing performance on query when the fanout goes up. This aligns well with the theoretical analysis.

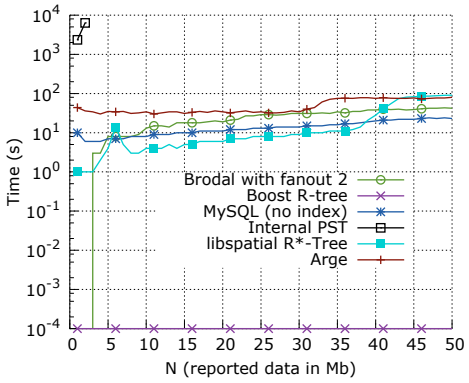

Figure 11.38

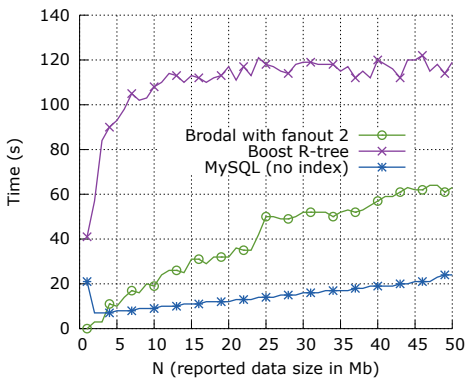

Figure 11.39

Reporting using the strategy of Figure 11.37 on data structures of size 100Mb (Figure 11.38). Here the Internal Priority Search Tree of McCreight suffers from swapping of memory. Running on data structures of size 400Mb (Figure 11.39). Here the Boost R-Tree suffers from swapping of memory.

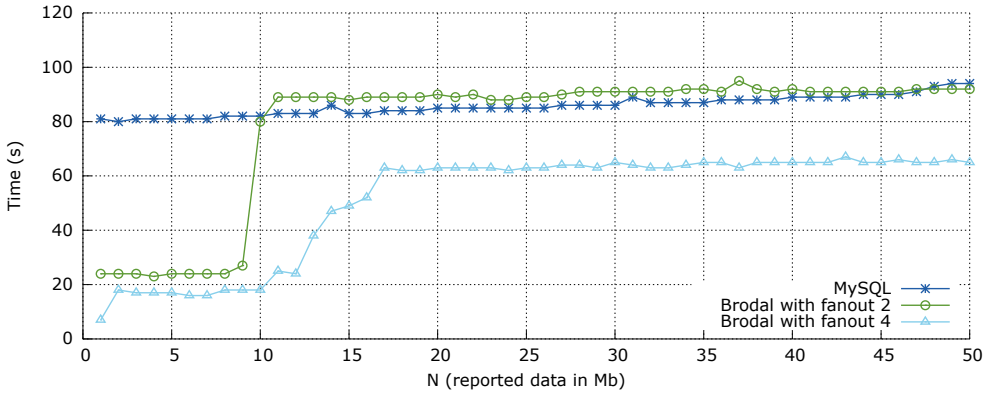

Figure 11.40: Reporting using the strategy of Figure 11.37 on a data structure of size 2,500Mb. The data structure of Brodal on fanout 4 now outperforms MySQL with no index.

11.4.2 Focus on reporting

This experiment focuses on the **report** part of the query algorithm. We limited the experiment to only include Brodal's since we only had limited time, and the nature of the experiment required a large input size. The experiment reused the idea of distributing data *inside* fixed query windows. The main difference was that we distributed 100Mb uniformly random data *inside* each query window and no data was added *outside* of the query windows. The idea is that we report in an x-range that spans all points and on increasing query-y values such that the number of reported points, K, grow with 100Mb when reporting in the range $[x_1, x_2] \times [y_{i+1}, \infty]$ compared to reporting in the range $[x_1, x_2] \times [y_i, \infty]$. We see from the results in Figure 11.42 that the query time follows a slowly decreasing line going towards a limit. This is what we would expect as

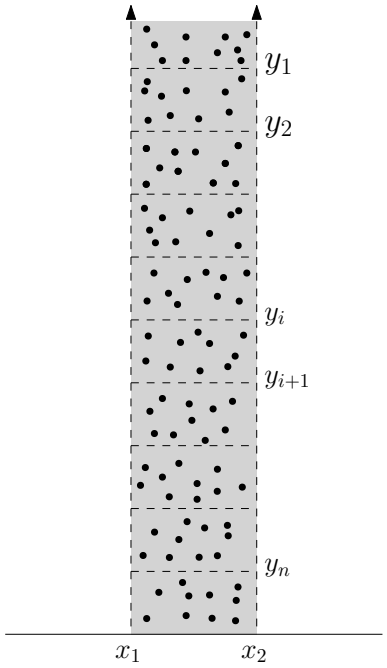

Figure 11.41: Distribution of points for the uniform reporting experiment. The points x_1 and x_2 and $y_1, \ldots, y_n$ are fixed. There is 100Mb data in all ranges $[x_1, x_2] \times [y_i, y_{i+1}]$.

$$f(K) = \frac{\frac{1}{\varepsilon}\log_B N + K/B}{K/B} \approx \frac{c}{K} + 1$$

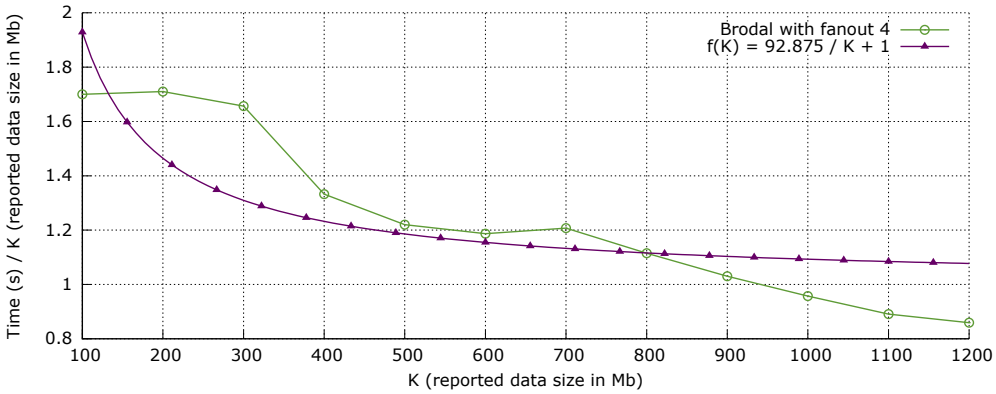

Figure 11.42: Experimentally measured running time divided by K/B when reporting increasingly more data on Brodal's of input size 1,200Mb. We fitted $f(K)$ to the experimentally measured data.

11.5 Construction

An interesting feature of the External Memory Buffered Priority Search Tree is that it supports construction in $\mathcal{O}(\text{Scan})$ on sorted input. We decided to compare this method on N points against the one-by-one insertion of N points. As we mention in Subsection 7.1.5 we could use the linear construction method together with a sorting algorithm to global rebuild the structure. The purpose of this experiment is to get an indication of which method is best when it comes to global rebuilding.

The experiment consists of constructing a data structure using the linear construction method and inserting one-by-one for increasing input sizes, N, and different fanouts to see whether this has an impact. We know from previous experiments that inserting becomes slower for larger fanouts but it is not clear what happens when we use the linear construction method.

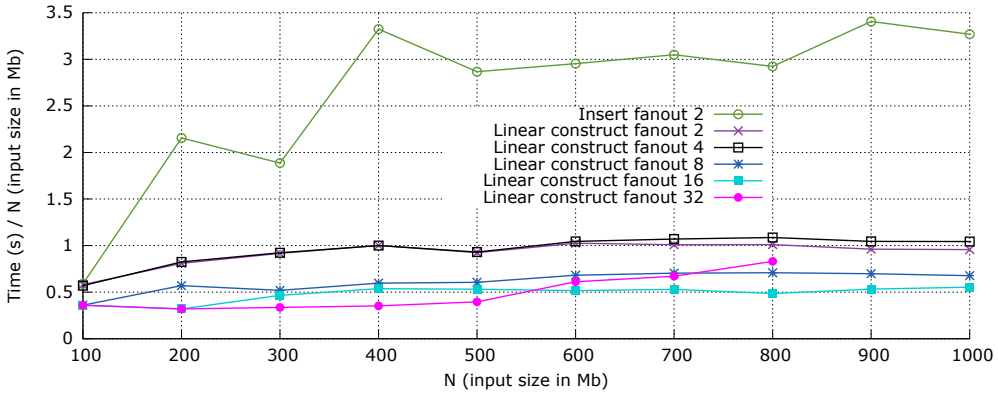

Figure 11.43: Running time of the linear construction method for sorted input divided by the input size in megabytes. The structure started to generate page faults on fanout 32.

Figure 11.43 depicts how the linear construction method performs for different fanouts. The running time is divided by the input size. Since the construction method is linear we are very pleased to see the graphs are horizontal when divided by N. This suggests we can construct the structure in linear time on sorted input. We see the running time improves for larger fanouts. This is explained in the decreased number of nodes in the tree which implies a decreased number of point buffer underflows that needs to be handled. Figure C.4 supports this claim.

Since linear construction is faster than inserting one-by-one we can conclude that it is indeed beneficial to construct the structure using the linear construction method on sorted input data. In order to conclude that this method should be used for global rebuilding we have to argue about the time required to sort the data. From an earlier course we have experimentally found that we can indeed sort fast enough for the linear construction method to be viable. The results from this course are depicted in Figure C.5. Based on these observations we feel confident to conclude that the linear construction method is superior when it comes to global rebuilding.

"To succeed, jump as quickly at opportunities as you do at conclusions."

— Benjamin Franklin

12 Conclusion

We have implemented three different data structures for solving the three-sided range reporting problem: The Internal Memory Priority Search Tree by McCreight [McC85], the External Memory Priority Search Tree by Arge et al. [ASV99], and the External Memory *Buffered* Priority Search Tree by Brodal [Bro15]. We have also implemented wrappers around MySQL 5.7.12, Boost R-Tree, and libspatialindex external memory R*-Tree, such that the implementation match the interface of a Priority Search Tree. All structures have been thoroughly described and analysed in the I/O Model, and for the internal memory structures we have argued that the complexity in the RAM Model can be adopted directly to the I/O Model due to the oblivious and inefficient access to data not available in internal memory.

We have conducted several experiments to compare the structures on the operations of the Priority Search Tree interface: `insert`, `delete`, and `report`.

Our results show that the External Memory Buffered Priority Search Tree outperforms all other structures except for MySQL without an index when it comes to inserting uniformly distributed data. We found that the Boost R-Tree performed the best in internal memory due to its efficient use of space.

When we looked at deleting uniformly distributed data, MySQL without an index expectedly fell short, and as soon as the structures not optimized for external memory outgrew internal memory we saw that the External Memory Buffered Priority Search Tree outperformed all other structures significantly.

We experimented with querying in two different ways to encapsulate the fact that the asymptotic complexity depends on both search and reporting. We isolated the search part by fixing several query windows of different size each with few megabytes worth of data, and we then inserted uniformly distributed data outside the windows. The results shows that the internal memory data structures once again perform very well while still contained in internal memory, but as soon as we outgrow internal memory the only real contestant to the External Memory Buffered Priority Search is MySQL with an index, which is once again outperformed for large enough input sizes. We isolated the reporting part of a query by reporting increasingly more data from a fixed query window. Our results were limited to the External Memory Buffered Priority Search Tree, and we showed that the structure follows the

expected behaviour.

Our experiments also showed that our implementation of the External Memory Priority Search Tree by Arge et al. proved inferior in all aspects which we explain by the large number of I/O's and heavy internal work.

As an extra experiment we found that the linear construction method of the External Memory Buffered Priority Search Tree proved to be a superior global rebuilding method over the one-by-one reinsertion of all points in an initially empty tree.

12.1 Future work

The structure of Brodal described in Chapter 7 is able to report top-k queries. A top-k query reports the highest k points in the query range. The structure does this in $\mathcal{O}(\frac{1}{\varepsilon}\log_B N + K/B)$, i.e. in the same bound as three-sided range queries, but here K is the minimum of k and the number of points in the query range. It would have been interesting to see how the structure performs these top-k queries compared to other results.

Many of the structures discussed in this thesis also describes a construction algorithm. In future work it could be interesting to see how the structures compare in terms of construction time.

Furthermore it could be an interesting future work to optimize the space usage of the structures allowing for larger buffers, and to see the effect of such optimizations on the running time.

Bibliography

[ABHY08] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Trans. Algorithms*, 4(1):9:1–9:30, March 2008.

[AE99] Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives, 1999.

[Arg95] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms (extended abstract). In *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, WADS '95, pages 334–345, London, UK, UK, 1995. Springer-Verlag.

[Arg05] Lars Arge. External memory geometric data structures(lecture note). Lecture note, 2005.

[ASV99] Lars Arge, Vasilis Samoladas, and Jeffrey Scott Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, pages 346–357, New York, NY, USA, 1999. ACM.

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[AV96] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, FOCS '96, pages 560–, Washington, DC, USA, 1996. IEEE Computer Society.

[AV03] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32(6):1488–1508, 2003.

[BF03] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 546–554, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[BFP+73] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[BGLY81] A. Borodin, L. J. Guibas, N. A. Lynch, and A. C. Yao. Efficient searching using partial ordering. In *Information Processing Letters, Vol 12, num 2*, 1981.

[BM72] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[Bro15] Gerth Stølting Brodal. External memory three-sided range reporting and top-k queries with sublogarithmic updates. *CoRR*, abs/1509.08240, 2015.

[CL09] Thomas H Cormen and Charles E Leiserson. *Introduction to algorithms, 3rd edition*. MIT Press, 2009.

[Dic16] Urban Dictionary. Urban dictionary - the struggle is real, 2016.

[FJKT99] R. Fadel, K.V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999.

[For64] Algorithms. *Commun. ACM*, 7(6):347–349, June 1964.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[Hoa61] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[IKO88] Ch. Icking, R. Klein, and Th. Ottmann. *Graph-Theoretic Concepts in Computer Science: International Workshop WG '87 Kloster Banz/Staffelstein, FRG, June 29 – July 1, 1987 Proceedings*, chapter Priority search trees in secondary memory (extended abstract), pages 84–93. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988.

[IP12] John Iacono and Mihai Pătraşcu. Using hashing to solve the dictionary problem. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 570–582. SIAM, 2012.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[KRVV96] Paris Kanellakis, Sridhar Ramaswamy, Darren E. Vengroff, and Jeffrey Scott Vitter. Indexing for data models with constraints and classes. *Journal of Computer and System Sciences*, 52(3):589 – 612, 1996.

[McC85] Edward M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.

[MSS03] Ulrich Meyer, Peter Sanders, and Jop Sibeyn, editors. *Algorithms for Memory Hierarchies: Advanced Lectures*. Springer-Verlag, Berlin, Heidelberg, 2003.

[Osb10] Jeff Erickson Scribe: Chris Osborn. The economist print edition (eu). feb 2010. http://www.economist.com/printedition/2010-02-27.

[Pro99] Harald Prokop. Cache-oblivious algorithms, 1999.

[RS94] Sridhar Ramaswamy and Sairam Subramanian. Path caching (extended abstract): A technique for optimal external searching. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '94, pages 25–35, New York, NY, USA, 1994. ACM.

[RW94] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[SR95] Sairam Subramanian and Sridhar Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, pages 378–387, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[Tar85] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[TG98] Andrew S. Tanenbaum and James R. Goodman. *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 1998.

[Vit08] Jeffrey Scott Vitter. *Algorithms and data structures for external memory*. Published, sold, and distributed by now Publishers, 2008.

[ZCO+15] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 27(7):1920–1948, July 2015.

A

Indexing for Data Models with Constraints and Classes

Kanellakis et al. presents a linear space and partially dynamic data structure in [KRVV96]. The data structure answers three-sided queries in $\mathcal{O}(\log_B N + {}^K/_B + \log_2 B)$ I/O's and supports inserts in $\mathcal{O}(\log_B N + \log_B^2 {}^N/_B)$ I/O's. The result is fairly involved and is unlikely to perform well in any practical manner.

We will omit the full details and only present the overall ideas. As a first step all points are shifted such that they lie above the line $y = x$. The basic building block of the data structure is the *metablock tree*; a B-ary tree of *metablocks*, each of which represents B^2 data points. The root represents the B^2 data points with the largest y-values. The remaining $N - B^2$ data points are divided into B groups of $(N - B^2)/B$ data points each based on the x-coordinate. The first group contains the $(N - B^2)/B$ data points with the smallest x-values and so on. A recursive tree of the exact same type is constructed for each such group of data points. This process continues until a group has at most B^2 data points and can fit into a single metablock. Refer to Figure A.1 for an illustration of a metablock tree.

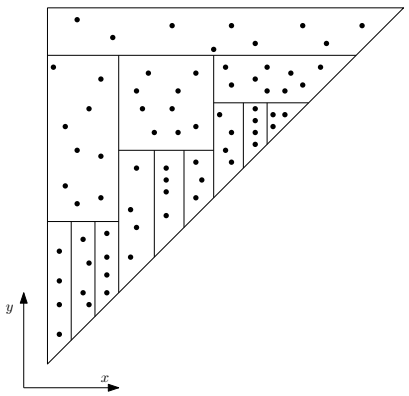

Figure A.1: A metablock tree for $B = 3$ and $N = 70$. All data points lie above the line $y = x$. Each region represents a metablock. The root is at the top. Note that each non-leaf metablock contains $B^2 = 9$ data points.

Points from each *metablock* are copied into new blocks that are both vertically and horizontally oriented as depicted in Figure A.2.

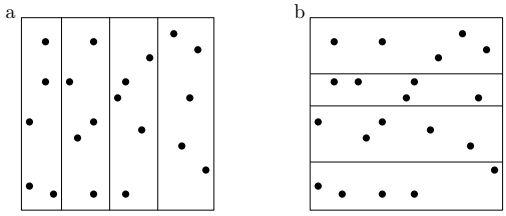

Figure A.2: Vertically and horizontally oriented blockings of data points. Each rectangle represents a block: (a) vertically oriented (b) horizontally oriented.

Finally, each metablock M contains pointers to B blocks that represents the set $TS(M)$ that is obtained by examining the left siblings of M and taking the B^2 largest points according to the y-value. Depending on how the query spans the metablock tree we can query the auxiliary data structures in such a way that we can achieve the promised $\mathcal{O}(\log_B N + {}^K/_B + log_2 B)$ I/O's. The five different query cases are depicted in Figure A.3.

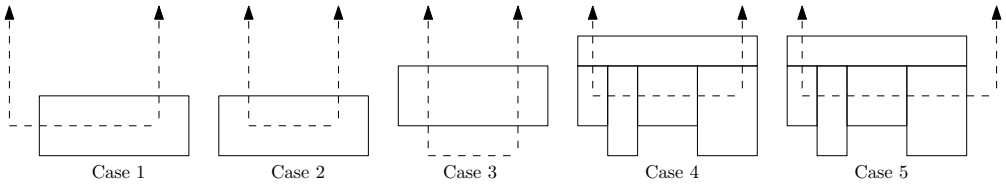

Figure A.3: The three-sided queries can span the metablock tree in five different ways.

B
Path Caching

Ramaswamy and Subramanian presents a suboptimal space data structure that answers three-sided queries with an optimal query bound in [RS94]. They use the same basic blocked B-Tree with pointers to full buckets of data points as introduced by Icking et al. [IKO88] and illustrated in Figure 4.1. In addition they introduce the idea of *path caching* that we will explain shortly. It can be seen that by using a B-Tree we are able to answer two-sided queries in $\mathcal{O}(\log N + {}^{K}/_{B})$ I/O's by classifying points inside the query into four categories as follows:

- *Corner*: this is the node whose region contains the corner of the query.
- *Ancestors of the corner*: These are nodes whose regions are cut by the left side of the query. There can be at most $\mathcal{O}(\log N)$ such nodes.
- *Right siblings of the corner and the ancestors*: these are nodes whose parents' regions are cut by the left side of the query. There can be at most $\mathcal{O}(\log N)$ such nodes.
- *Descendants of right siblings*: there can be an unbounded number of them, but for every such node, its parent's region has to be completely contained inside the query. That pays for the cost of looking into these nodes. That is, for every J descendant blocks that are partially cut by the query, there will be at least ${}^{J}/_{2}$ blocks that lie completely inside the query.

Please refer to Figure B.1 for an illustration of the categories.

Querying is done by locating the nodes intersecting the left side of the query using the B-Tree. Points from the nodes are reported by examining the associated buckets. Next, right siblings of the nodes and their descendants are examined in a top-down fashion until the bottom boundary of the query is crossed. It is crucial to note that the corner, ancestor, and sibling nodes can cause wasteful I/O's. Thus there are $\mathcal{O}(\log N)$ wasteful nodes as every parent of a visited node would have contributed an useful I/O. From this analysis we can conclude that we can answer two-sided queries in $\mathcal{O}(\log N + {}^{K}/_{B})$ I/O's.

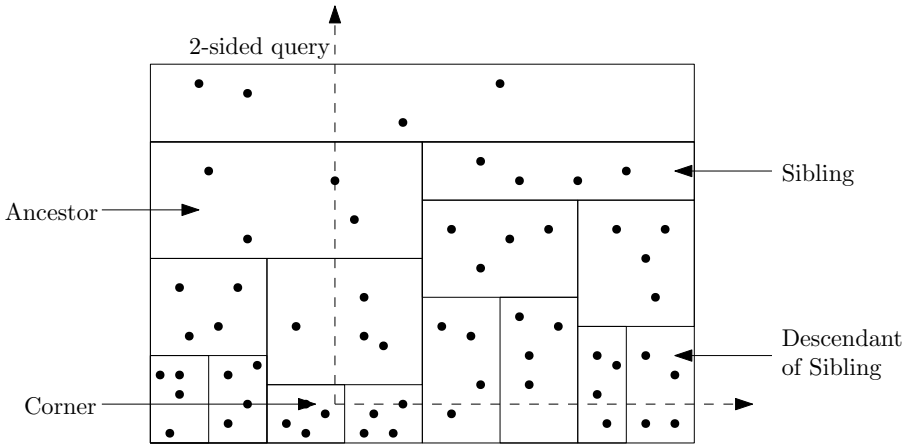

Figure B.1: Binary tree implementation of a Priority Search Tree in external memory showing a corner, ancestor, sibling, and descendant of sibling on a query. Here, B is 4.

Now, we are able to avoid the wasteful I/O's by caching the data in the ancestor and sibling nodes. By associating two caches with the corner that contain all data in the siblings sorted by x-coordinate and y-coordinate respectively, we are able to answer queries in $\mathcal{O}\left(\log_B N + {}^K/_B\right)$ I/O's by simply locating the corner in $\mathcal{O}(\log_B N)$ I/O's and report using the cache. The storage usage is $\mathcal{O}({}^N/_B \log N)$ disk blocks of size B each. The idea can be extended to handle three-sided queries by adding additional caches that cover point sets sorted from right to left. This gives a space usage of $\mathcal{O}({}^N/_B \log^2 N)$.

B.1 Better space bounds

Ramaswamy and Subramanian brings down the space usage in [SR95]. This is done by building a search tree that divides the points into smaller regions of size $B \log B$ instead of B giving a total of ${}^N/_B \log B$ regions. Now a slightly modified caching scheme is used with an additional secondary level structure for each region giving a total space usage of $\mathcal{O}({}^N/_B \log \log B)$. Reusing this idea in a multilevel scheme brings down the data structure to an $\mathcal{O}({}^N/_B \log B \log^* B)$ space solution that answers queries in $\mathcal{O}(\log_B + {}^K/_B + \mathcal{IL}^*(B))$, where $\mathcal{IL}^*(x)$ denotes the number of times $\log^*$ must be applied before the result becomes ≤ 2.

C
Data

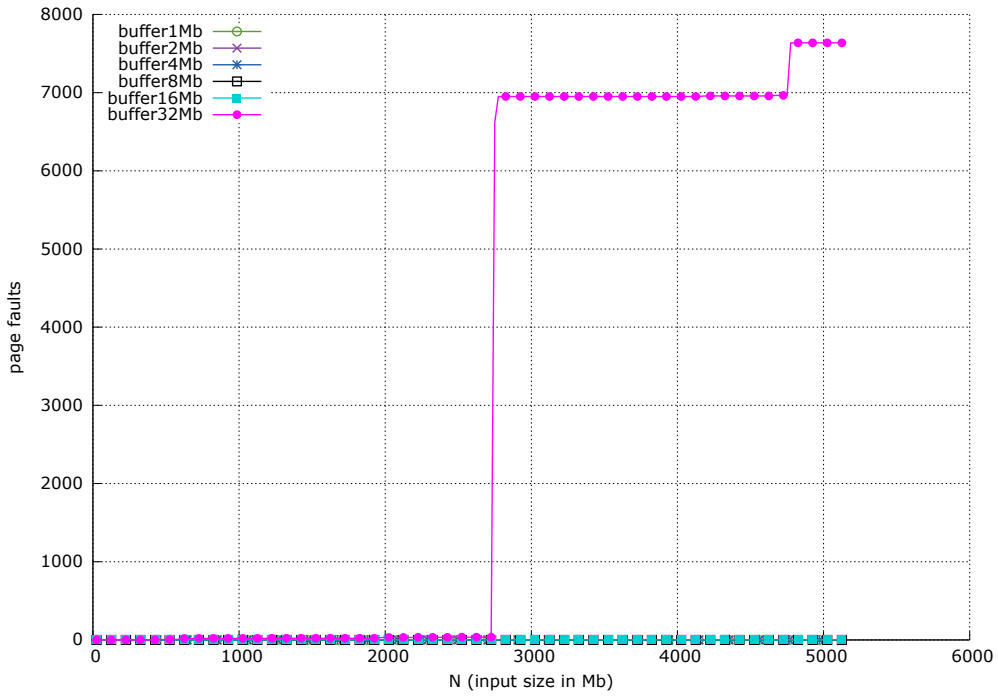

Figure C.1: The measured number of page faults when inserting 5Gb of data in the External Memory Buffered Priority Search Tree. We see that a buffer size of 32Mb will generate page faults.

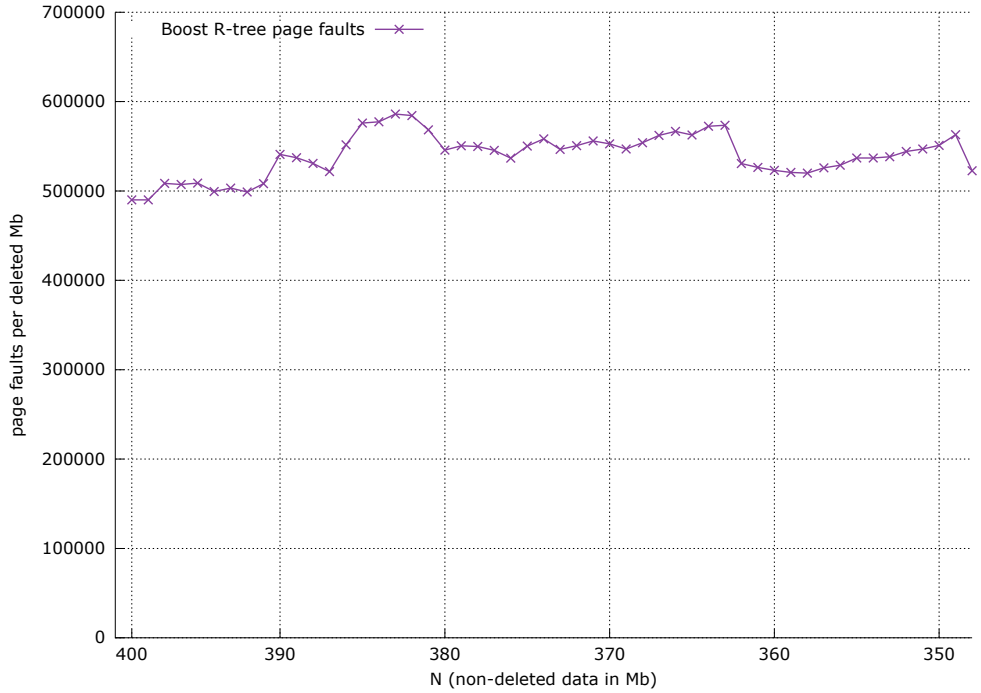

Figure C.2: Experimentally measured number of page faults when deleting points in the Boost R-Tree from a size of 400Mb down to a size of 350Mb.

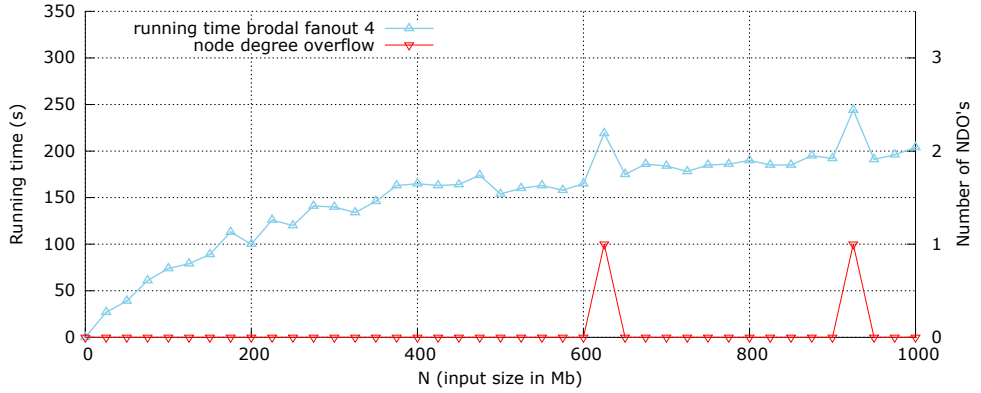

Figure C.3: Experimentally measured number of insert buffer overflows, node degree overflows, and point buffer underflows on the data structure of Brodal with fanout 2 depicted in Figure 11.36

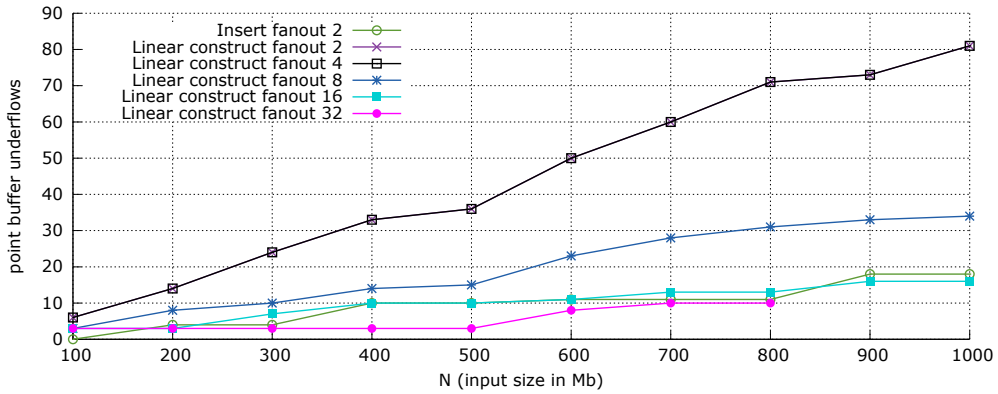

Figure C.4: The number of point buffer underflows for different fanouts when linear constructing. We see that the number of point buffer underflows decreases with increasing fanouts.

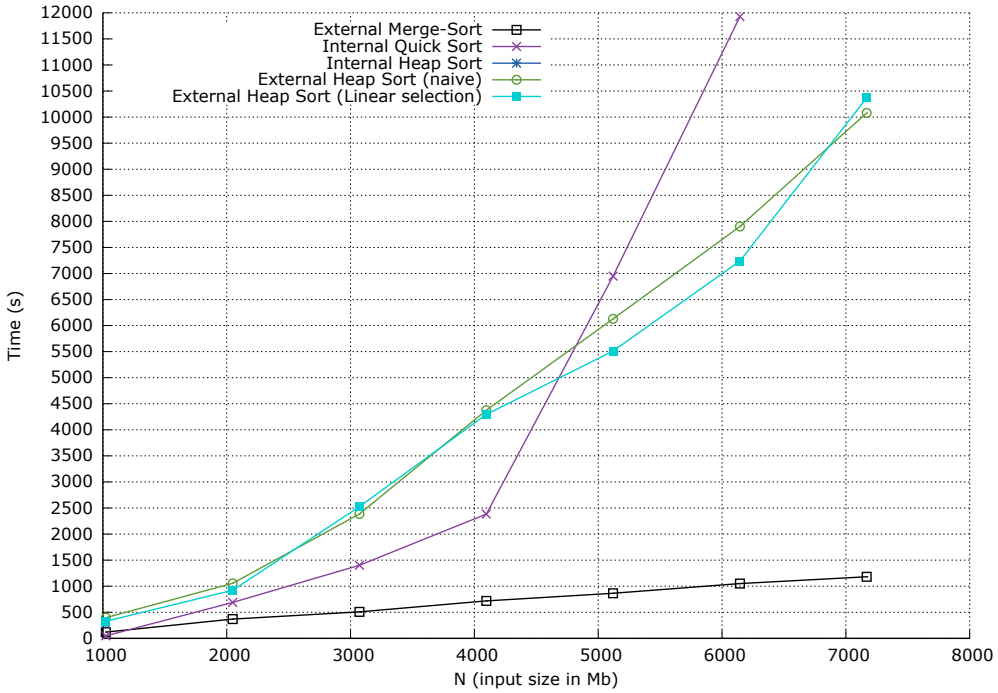

Figure C.5: The running time for different sorting algorithms. The tested algorithms include the External Heap sort by Fadel et al. [FJKT99] both naive and with linear selection [BFP+73], The External Merge Sort as described by Knuth in [Knu97], Internal Quick Sort by Hoare [Hoa61], and finally the Internal Heap Sort by Forsythe [For64].