NTNU
Norwegian University of
Science and Technology

R-trees with Overflow Blocks

Anders Oftebro Bjørnøy

Master of Science in Computer Science
Submission date: June 2017
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

A wide range of R-tree variants exists that competes to give the best performance for different uses[13]. This has gone on since the original R-tree was published in 1984[10]. A new wave of variants appeared to take advantage of new hardware such as SSDs, which has noticeable differences to HDDs.

The B^+-tree, an even older structure, has also had several different papers with goals to increase its performance. One of the newer ones (2015) describes the Bloomtree[11], which is a B^+-tree that makes use of three different leaf types. One of them utilises bloomfilters to reduce reads, hence the name. Unlike other B^+-tree variants that trades writes for reads to fit the characteristics of an SSD, this Bloomtree reports a decrease in both reads and writes. This would indisputably increase performance, given the structure doesn't have any special needs such as extreme CPU usage.

Given the performance increase the Bloomtree yields, it's interesting to see if its concepts can be applied to an R-tree. R-trees have strong similarities to B^+-trees, and is in some ways just an n-dimensional B^+-tree. Therefore, given its resemblance, it seems like a plausible task.

During the paper, only some parts of the Bloomtree was successfully applied to an R-tree. There was unfortunately not found a suited replacement for the most interesting leaf node, the one which uses bloomfilters. Without that node, it essentially then became an R-tree with overflow blocks. The B^+ has one such overflow structure, which performed poorly. Therefore no particular performance gain was expected, other than that it should perform better than it's overflow B^+-tree equivalent. Simulations done in this paper showed that overflow with the overflow linear R-tree did perform better than it's B^+-tree equivalent, and could actually rival the performance of the plain linear R-tree.

Sammendrag

Det eksisterer en rekke forskjellige R-trær som konkurrerer om å være best for forskjellige bruksområder [13]. Evolusjonen har foregått siden det originale R-treet ble presentert i 1984[10]. Etter at SSDer ble introdusert, så har det kommet en ny *bølge* med R-tree varianter som spesifikt utnytter fordelene slike SSDer har.

Den eldre indekseringsstrukturen B^+-trær har også mange artikler om forskjellige versjoner der ytelse er i fokus. En av de nyere (2015) er Bloomtree[11], som er et B^+-tree som bruker tre forskjellige løvnode typer. Ene av den bruker et bloomfilter til å redusere antall lesinger. Bloomtree artikkelen viser at strukturen deres reduserer både lesinger og skrivinger, i motsetning til andre varianter som endrer på lese og skrive ratio for å utnytte egenskapene til en SSD. Å redusere både lesinger og skrivinger gir naturligvis en økelse i ytelse, gitt den ikke har noen andre egenskaper som hemmer.

Med motivasjon i Bloomtreets økte ytelse i forhold til B^+-treet, så hadde det vært spennende å se om konseptene kunne brukes i R-trær også. R-trær og B^+-trær har mange likheter. R-treet er nesten et n-dimensjonalt B^+-tree, så å gjenbruke konsepter fremstår som en overkommelig utfordring.

I løpet at rapporten så ble bare deler av Bloomtreet overført til er R-tree. Den mest spennende delen ved Bloomtreet, den løvnoden som bruker bloomfilter, ble ikke overført til et R-tree. Dette fordi ingen god erstatning ble funnet for R-treet. Uten denne delen så ble treet i rapporten i grunn et R-tree med overflytsblokker. Tilsvarende finnes det et B^+-tree med overflytsblokker som viser seg å yte dårligere enn B^+-treet alene. Derfor var det ikke ventet noe spesiell ytelse fra R-treet med overflytsblokker, annet enn at det burde prestere bedre enn B^+-tree ekvivalenten. Simuleringer gjort viser at linear R-tree med overflytsblokker yter bedre enn B^+-tree ekvivalenten, og kan til og med utfordre ytelsen til det vanlige linear R-tree.

Preface

This report was written as a part of a master project for the Department of Computer and Information Science at the Norwegian University of Science and Technology. It was written during the spring semester 2017 under the supervision of Svein Erik Bratsberg whom I'd like to thank for excellent feedback and comments throughout the semester. Much of the report is based on work done during a specialization project the preceding semester by the same author and with the same supervisor.

Anders Oftebro Bjønøy

Trondheim, June 2017

Table of Contents

Abstract **i**

Sammendrag **iii**

Preface **v**

Table of Contents **x**

List of Tables **xi**

List of Figures **xv**

1 Introduction **1**

1.1 Goal 2

1.2 Findings 2

2 Theory **3**

2.1 B+tree 3

2.1.1 Algorithms 4

2.2 R-tree 5

2.2.1 Algorithms 6
2.2.2 R-tree types 9
2.2.3 Challenges 10
2.2.4 Optimising for modern hardware 11

3 Bloomtree **13**

3.1 Overview 13
3.2 Leafnodes 14
3.2.1 Normal 14
3.2.2 Overflow Node 14
3.2.3 Bloomfilter Node 15
3.3 Findings 16

4 Related work **17**

4.1 R*-tree 17
4.2 Generalized Bulk Insertion 20
4.3 Bulk insertion by seeded clustering 21

5 Implementation **23**

5.1 Overview 23
5.2 Challenges 24
5.2.1 Different type of data/storage 24
5.2.2 General optimisation 24
5.2.3 BF-Node 24
5.2.4 Overflow blocks 27
5.2.5 Variants implemented 28
5.2.6 Cache 29

5.2.7 Choice of language 29

6 Datasets and metrics **31**

6.1 Datasets 31

6.1.1 Bit02 31

6.1.2 Uni02 32

6.2 Metrics 33

7 Results **35**

7.1 How test where done 35

7.1.1 Default configuration 36

7.2 Changing cache size - Bit02 36

7.3 Changing fanout - Bit02 40

7.3.1 Linear R-tree 40

7.3.2 R*-tree 42

7.4 Changing fanout - Uni02 46

7.4.1 Linear R-tree 46

7.4.2 R*-tree 48

7.5 Changing fanout with cache size 1000 - Uni02 52

7.5.1 Linear R-tree 52

7.5.2 R*-tree 53

8 Discussion **55**

8.1 Unexpected results 56

8.2 Comparison to OR-tree 56

9 Conclusion **59**

10 Further work **61**

Bibliography **63**

A Appendix **65**

List of Tables

7.1 Default parameters for runs. 36

A.1 Bit02 distribution, fanout 50, p_grade 30%, min_fill 40%. 66

A.2 Bit02 distribution, p_grade 30%, min_fill 40%, cachesize 0. 67

A.3 Bit02 distribution, p_grade 30%, min_fill 40%, cachesize 0. 68

A.4 Uni02 distribution, fanout 50, p_grade 30%, min_fill 40%. 69

A.5 Uni02 distribution, p_grade 30%, min_fill 40%, cachesize 0. 70

A.6 Uni02 distribution, p_grade 30%, min_fill 40%, cachesize 0. 71

A.7 Uni02 distribution, p_grade 30%, min_fill 40%, cachesize 1000. 72

A.8 Uni02 distribution, p_grade 30%, min_fill 40%, cachesize 1000. 73

List of Figures

2.1 A small B$^+$-tree with fanout 4. 4

2.2 Two different MBRs for the same three points. 6

2.3 A simple R-tree with corresponding 2D visualisation at the bottom. 7

3.1 An overview of the three different leaf types. [11]. 14

3.2 Performance for B$^+$-tree and OB$^+$-tree. Figure from [11]. 15

4.1 Linear R-tree. Leaf level MBRs for 2000 points. 19

4.2 Linear R-tree. Leaf level MBRs for 2000 points. 19

4.3 Process when inserting points in a bulk insertion structure. Inspired by [6]. 21

5.1 A simple grid filter with $n = 3$. 26

6.1 The Bit02 dataset. $1.000.000$ 2D datapoints. 32

6.2 The Uni02 dataset. $1.000.000$ 2D datapoints. 32

7.1 Reads used to create different structures from Bit02 distribution. Fanout set to 50. Rstar is the R*-tree. Rstar-o3 is R*-tree with max_overflow as 3. Linear is the linear R-tree. Linear-o3 is the linear R-tree with max_overflow as 3. 37

7.2 Writes used to create different structures from Bit02 distribution. Fanout set to 50. Rstar is the R*-tree. Rstar-o3 is R*-tree with max_overflow as 3. Linear is the linear R-tree. Linear-o3 is the linear R-tree with max_overflow as 3. 38

7.3 Reads per query in q0 from Bit02 distribution. Fanout set to 50. Rstar is the R*-tree. Rstar-o3 is R*-tree with max_overflow as 3. Linear is the linear R-tree. Linear-o3 is the linear R-tree with max_overflow as 3. . . . 38

7.4 Reads per query in q3 from Bit02 distribution. Fanout set to 50. Rstar is the R*-tree. Rstar-o3 is R*-tree with max_overflow as 3. Linear is the linear R-tree. Linear-o3 is R-tree with max_overflow as 3. 39

7.5 Writes used to create different structures from Bit02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X. 40

7.6 Reads used to create different structures from Bit02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X. 41

7.7 Reads per query in q0 from Bit02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X. 41

7.8 Reads per query in q3 from Bit02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X. 42

7.9 Writes used to create different structures from Bit02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X. 43

7.10 Reads used to create different structures from Bit02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X. 44

7.11 Reads per query in q0 from Bit02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X. 44

7.12 Reads per query in q2 from Bit02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X. 45

7.13 Reads per query in q3 from Bit02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X. 45

7.14 Writes used to create different structures from Uni02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X. 46

7.15 Reads used to create different structures from Uni02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X. 47

7.16 Reads per query in q0 from Uni02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X. 47

7.17 Reads per query in q2 from Uni02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X. 48

7.18 Reads per query in q3 from Uni02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X. 49

7.19 Writes used to create different structures from Uni02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X. 49

7.20 Reads used to create different structures from Uni02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X. 50

7.21 Reads per query in q0 from Uni02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X. 50

7.22 Reads per query in q2 from Uni02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X. 51

7.23 Reads per query in q3 from Uni02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X. 51

7.24 Reads used to create different structures from Uni02 distribution, cache size 1000. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X. 52

7.25 Reads used to create different structures from Uni02 distribution, cache size 1000. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X. 53

7.26 Reads pr query in q0 from Uni02 distribution, cache size 1000. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X. 54

Chapter 1

Introduction

Since the R-tree was introduced, many papers have been released in search of increasing its performance. It fast became de facto for structuring spatial data, which gives it an extra relevant spot among data structures today. More and more data are being stored, and often with a spatial aspect attached to it.

Naturally, given its introduction in 1984[10], the underlying data storage, to begin with, was hard disc drives. Therefore these first papers describe structures that perform well for hard disc drives, and the workload fits well for the discs read and write times.

Recently the solid state drive has taken its share of the marked. It gives various benefits compared to the traditional HDD and one would naturally want to use SSDs over HDDs. With the price of SSDs decreasing, this has now become a viable choice for some actors. With the rise of SSD usage, there has in the last years been many papers on how to make R-trees that are well suited for an SSD storage.

Such papers show that it's useful to optimise for specific storage, and certainly possible to take advantage of a storage's strong sides to get increase performance. However, the structure should preferably be agnostic to storage type, and perform better than it's competing structures regardless of where it's stored. In the case of R-trees the possible primary storage types is HDDs and SSDs. With today's hardware prices, it is also plausible for some databases to keep it entirely in memory[14, 7], but that is outside the scope of this paper.

Computing cost is also a factor for R-trees. Some can have advanced and resource-intensive algorithms, while other have simple lightweight ones. There is a clear correlation between how advanced the algorithm is, and how well the structure is. Even tough computation cost is important, the bottleneck is often with reads and writes to permanent storage. Therefore computational cost and runtime are left outside the scope of this paper,

while i/o is considered.

1.1 Goal

The goal of this paper is to create an experimental R-tree structure that takes advantage of the concepts from the Bloomtree. The Bloomtree is a B^+-tree that uses three different leaf nodes in order to reduce the overall I/O. It is especially the use of bloomfilters in one of its leaf node that makes it special and interesting. In their paper [11] they report a significant reduce in i/o, compared to other B^+-tree structures. As an R-tree is in many ways an N-dimensional B^+-tree, it is interesting to see if its concepts can be transferred from the Bloomtree to an R-tree. If possible, does such an R-tree benefit the same improvement in i/o?

Research questions are therefore:

- **RQ1:** Is it possible to apply the concepts of the Bloomtree to an R-tree?
- **RQ2:** Does this new solution have the same performance achievement as the Bloomtree?

1.2 Findings

Throughout the paper, no equivalent to the Bloomtrees special leaf node was found. Although the R-tree and B^+-tree has many similarities, they also have many differences that make transferring certain concepts hard, if not impossible. Without a suited way to use the Bloomtrees special node for an R-tree, one was left with only applying the concept of overflow to the R-tree. Such a concept is not new, Wang et al [16] has done so with their OR-tree that they specialise for SSDs instead of being storage agnostic.

The implemented overflow structures, based on the linear R-tree [13] and on the R^*-tree [1], generally shows a decrease in writes but an increase in reads. It also shows that with a cache, it is plausible to utilise the Linear R-tree with overflow and reduce i/o.

Chapter 2

Theory

This chapter explains the basic workings of the B$^+$-tree and the R-tree. As we try to apply concepts from a B$^+$-tree, it's relevant to know how it works and what concepts its basic form is built upon. Of course, it's also necessary to explain the R-tree, which we, after all, is trying to optimise. After laying out this base of how the structures works, a handful of interesting variations of the R-tree is described to get a view of the possible variants to gain inspiration from. Much of the theory discussed here is from [8, 10, 13]

2.1 B+tree

The B$^+$-tree has evolved from the standard B-tree. Because the B$^+$-tree is far more popular and similar to the R-tree, the standard B-tree is left undescribed. The B$^+$-tree is a key-value storage, that uses a search-tree structure. Being a key-value storage means that one provides a key and the storage returns the value associated with that key if it exists. The search-tree structure means that it can search efficiently down through the tree, selecting the right path until it reaches the leaf node with the right key. This is made intuitive by having keys that are linear of one-dimension, meaning that there is a given order at the leaf level. This is, as we shall see, a significant difference from the R-tree.

With a base in Figure 2.1, the simple concepts of the B$^+$-tree can be explained. The figure shows a small tree with only two levels and five posts. Each block can contain $b = 4$ pointers, and $b - 1 = 3$ keys. This b is called fanout, and it determines how much the tree can "grow" each level. At the bottom level, the number represents the key for the post, and the pointer either contains the post or gives the exact place to find it. At the leaf level, the last pointer is reserved to point to the next block in the sequence. Because of this last pointer, the leaf level acts like a chained list over one dimension. This enables range

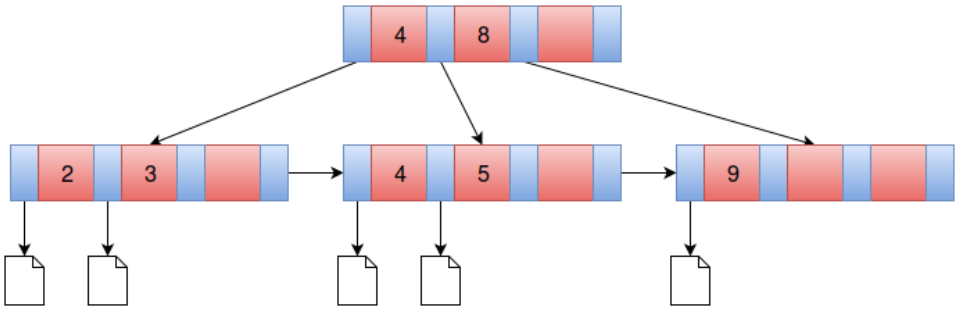

Figure 2.1: A small B^+-tree with fanout 4.

searches and scans to be done with ease.

2.1.1 Algorithms

Search

Usually, one searches the B^+-tree by providing a key and wanting the corresponding value/post back. The search is relatively straightforward and known in the B^+-tree as the structure has strong resemblances to the binary tree. The search is executed in a structured manner. When performing a search, it starts with the root node. Then beginning with the leftmost value in the node, one checks if the key that is being looked up is smaller than the value. If the key is smaller, then one follows the pointer on its left. If the key is equal or greater, one moves on to check against the next value in the block in the same manner. If one reaches the last value in the node and the key is still equal or bigger than the value, then one follow the rightmost pointer. The pointers take us to new nodes where we do the same action. Eventually one reaches a leaf node. In that case, each document has a ”dedicated” pointer, as the pointer to the left for a value will point to the record/document. If the key doesn't match any value in that leaf node, then it doesn't exist in the tree.

Because the tree is balanced and it chooses the right path for each step, few blocks have to be accessed during this kind of searches. This gives it with a logarithmic search time.

One can also do a range search, say wanting all the post that has keys between 'A' and 'B'. Then one follows the pointer, searching for the first legal key in the range. When reaching the leaf node, it follows the last pointer in the node in order to get to the next consecutive leaf node. When the keys in the leafs are beyond the one searched for, it's done.

Insert

Insertions use the same way to find the right leaf node to insert the record/document. Typically, insertions only require that the right leaf node inserts a pointer to the document and the corresponding key in the correct way.

However, there is also a possibility that the leaf node you're trying to insert into is full. In this case, one has to split the leaf node into two and update the parent node as well with value and pointer of the new leaf. This update might propagate upwards if the parent node is full. In a worst case, it propagates all the way back to the root node, which then also have to be split into two while adding a new root node.

Deletes

Deleting is also a simple process. One just deletes the value and pointer from the leaf node. Here it could be a problem that the nodes end up having too few elements after a delete. If the node ends up having less than the threshold m values, then it either has to redistribute documents from a sibling leaf node (sharing parent node) or merge with one.

2.2 R-tree

New technology, services and applications have driven the need for a search structure with multiple dimensions. The R-tree has become the default indexing structure for a range of such applications, especially for applications that utilise spatial data. After the R-tree was introduced in 1984 by Guttman [10], a wide range of improvements and alternatives has been made. This section explains how the standard R-tree works, and then mentions some variants in the end.

While the B^+-tree is great for one-dimensional data, it can't effectively deal with keys with more dimensions. Spatial data as coordinates is a highly relevant example of data that it can't index. This is where the R-tree appear, a structure that strongly resembles the B^+-tree, but can support n-dimensions. At least in theory. Too many dimensions invokes the "curse of dimensionality", but it's still supported.

The structure for the R-tree shares many characteristics with the B^+-tree. They are both search trees with values and pointers in order to efficient support a range of operations, like searches and insertions. Both are also self-balancing. One important difference is that there it makes use of Minimum Bounding Rectangles (MBRs) instead of keys to structure the tree. Also unlike the B^+-tree that had one more pointer than keys, the R-tree has one pointer designated for each MBR in the node.

The MBR is a way to define the smallest rectangle to enclose a set of objects. For two

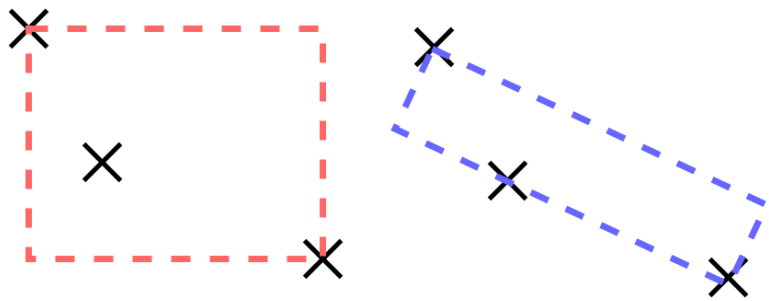

Figure 2.2: Two different MBRs for the same three points.

dimensions, X and Y, this would mean we define an MBR by determining four values, $(max\_X, min\_X, max\_Y, min\_Y)$, in practice two diagonal corners. One can add more dimensions to an MBR by adding a min/max for the new dimension. Because of this property of the MBR, the R-tree can support n-dimensions. Note that we don't use actual MBR, we only use MBR with horizontal and vertical lines. Image 2.2 shows the difference. While the blue one to the right is an actual MBR and more correct, we use the one on the left as its way more convenient and frequently used.

A simple R-tree is shown in Figure 2.3, with a corresponding 2D canvas that shows the points. Just as the B$^+$-tree, it consists of nodes and pointers. The nodes can hold up to a certain amount of posts containing a Minimum Bounding Rectangle (MBR) and a pointer. In the leaf nodes, the posts contain an MBR (in the image just a point) and possibly extra meta-data.

The observant reader will have noticed that the MBRs have to circumvent all the MBRs of its children. In other words, the MBRs works hierarchically. This means that the parent MBR must surround all the points and MBRs of its children. One would also have noticed that some posts actually fits within several MBRs. If the posts were inserted in a different order, the tree could look different. This is due to the way insertions is handled in the R-tree. This property makes the tree non-deterministic.

Furthermore, one will notice that some of the MBRs overlap. The fact that MBRs can overlap comes from the fact that there is no definitive way to order the elements as they are several dimensions, this is a big challenge that the B$^+$-tree don't have to deal with. This problem directly influences search speed, as much overlap can result in many search paths.

2.2.1 Algorithms

As mentioned above, this section only describes the basic/standard R-tree if otherwise isn't written.

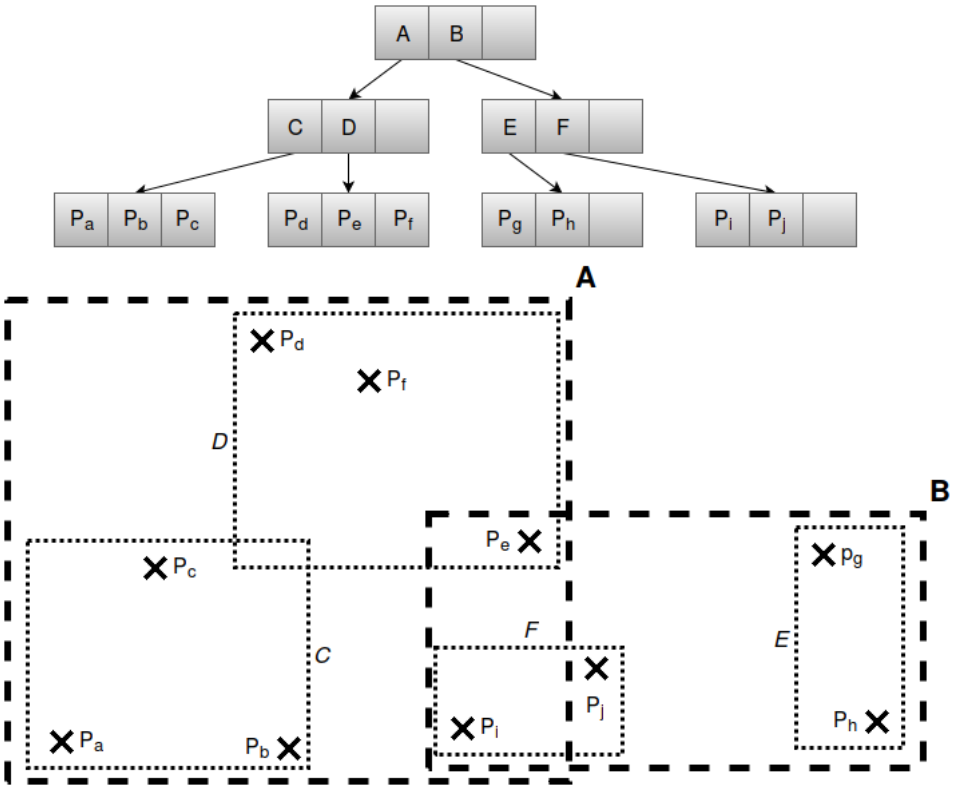

Figure 2.3: A simple R-tree with corresponding 2D visualisation at the bottom.

Searches

The typical thing to do in an R-tree is to perform *RangeSearches*, which means you want to retrieve all points that are within a query rectangle Q. In order to perform this search, the structure recursively searches the tree. Beginning at the root node, one finds all the children that have an MBR that overlaps with Q. For all those children, the same process is done until it reaches leaf nodes. There the posts that have an MBR that overlaps with Q is added to an answer set. We clearly see how the hierarchical structure aids the search.

Unfortunately, the MBRs can overlap, even in the same height, there can be several nodes that can contain the point. Therefore it can be several search paths that must be looked into for a query. Naturally, a search that has to consider many paths takes longer time, and therefore it is preferable to minimise the overlap. Having a good tree structure is critical in order to get good query performance. Many R-tree variants focus on improving MBRs to, among other things, reduce overlap. These improvements are primarily done in the insertion algorithm, and subsequently the split algorithm.

Another search form that occurs is K-nearest neighbour search. Such a search provide a point and asks for the K nearest post. An example of one such query "Give me restaurants around me", where the search point provided is the users' location. Such a search also requires some filtering to give the K nearest restaurant, instead of just the K nearest posts.

Insertions

As with the other algorithms, the R-tree is pretty similar to the B$^+$-tree here as well. It gets a post to insert, then starting from the root the algorithm selects one suited MBR and follows the corresponding pointer and then does the same in that node. The suited MBR is found by different metrics, but one simple metrics is to select the MBR that needs the least enlargement to accommodate the new post.

Eventually, it finds a suited leaf to insert the post. If the suited leaf has space for the element, it's an easy process to just add it. If the new element enlarges the leaf nodes MBR, one also has to remember to potentially update the parent nodes MBRs, possibly all the way to the top.

In the case where the selected node is full, a split is needed. Splits in R-trees aren't as easy as in B$^+$-trees because there are several dimensions and therefore a lot more to consider when splitting. In the R-tree, different metrics are used when splitting that results in different splits. One basic way to do a split is to divide the node's elements into two sets, where one want to minimise the total MBR area of the two sets. There are three ways to go about this calculation, Linear, Quadratic or Exponential split. There is a trade-off between the different algorithms is search-space and the result. Often Quadratic split is chosen as a compromise.

Different types of insertions and splits are some of the main differences between R-trees variants. For example, does some utilise more advanced and expensive algorithms that give better search performance. The R*-tree is one well know variant which has a more expensive insert algorithm but makes up for it in search performance. This R-tree variant is explained in deeper detail in Section 4.1.

Deletes

When deleting an element one has to first find the leaf node that contains the element. As this first part is a search, it also is affected by how good the MBR structure is. Then the element is deleted from the node, and one might end up updating several MBRs again.

A tree also has a min fill degree, for example, a threshold of $m = 40\%$. The tree isn't allowed to have nodes with fewer elements than m. So, if a delete results in a node having too few elements one have to restructure the tree. Again, depending on the tree type, different delete algorithms are used. The basic one either redistributes from a sibling,

merge with a sibling or reinserts all remaining elements.

2.2.2 R-tree types

As mentioned, many alternatives have appeared. They all have strong resemblances to the R-tree, but they are different in what properties and metrics they value. There are also two different types of R-trees to group them by. We have the dynamic R-trees and the static ones. R-trees are generally dynamic, or at least introduced as a dynamic structure, meaning they were suited for randomly occurring deletions, insertions and updates. On the other hand, static trees mainly focus on building up an R-tree once, without potentially degrading the tree structure by having insertions or deletions that can result in bad splits.

Both groups of R-trees have their use cases and applications that they are well suited for. Section 4, related work, describes some of the more interesting R-tree variants in detail.

Dynamic

R-trees are in ”nature” dynamic as they don’t need global reorganisation to handle changes in the tree. In this case, it means that they are suited for insertions and deletions. The tree described above is the original R-tree which is indeed dynamic. Also, the first variants of the R-tree to follow were of the dynamic sort, such as the R^+-tree and R^*-tree. The trees within the dynamic group vary most in what metrics they value.

Static

In contrast to the dynamic trees, which handles changes in the trees, the static ones need to know all the data beforehand of creating the tree. This is called *packing* or *bulk loading*. There are several advantages, with the distinct advantage being that one doesn’t have to restructure the tree because new insertions can’t occur. Having insertions one by one (OBO) is an ineffective way to build up a large R-tree from scratch if we have all the points in advance. One also avoids the problem where insertions ”damages” the structure of the tree due to simplified heuristics, and the fact that it would simply be too costly to globally restructure the tree each time.

This makes it much easier to make (closer to) optimal trees and one can also potentially achieve close to 100 % disk utilisation. The dynamic types would have potentially globally restructured the tree for each insert to achieve the same (in practice statically building it for each insertion). Depending on the tree and characteristics wanted, different techniques are used when building the tree. Among them are top-down and bottom-up approaches.

While there are suited applications for bulk loading trees, one would usually want to insert more data at some point. This can, of course, be done with the insertion command that we

know from the dynamic versions. The insertion command works fine for small amounts of data, but for large quantities of data, it requires time and resources. Taking up a lot of time means that applications using the structure are suffering, as they can't query it. Several R-tree versions address this need for inserting. Common for these approaches is that they insert larger branches, instead of points OBO. Among them are the Generalized Bulk Insertion(GBI) which is described in section 4.2.

2.2.3 Challenges

Insertion cost

With the dynamic versions, a problem is that the insertion cost scales with the size of the tree. A higher tree means that more nodes have to be accessed for queries and insertions. Changes also it might propagate longer. The approaches with bulk insertion solve this problem to some degree. Such approaches have shown to scale way better than the R-trees, and keep up with query performance.

Another similar problem is that the query cost scales with the size of the tree. In the same way, here a higher tree means a longer path to the desired elements. The height depends on the order of the tree, filling degree and number of elements. A tree typically reaches a broad span of elements in few step. Besides reads are generally much faster than writes. However, if a search need to traverse many paths and each path are long, it can take a long time. The answer is to have a good structure.

Partitioning

Partitioning data of one dimension can be intuitively and effectively done, as we see for the B^+-tree. Still, there are choices to make when partitioning one-dimensional data, for example between hash function or value range. Partitioning data that are in two dimensions isn't as intuitive, and for higher levels of dimensionality, it gets worse. In this paper, only two dimensions are considered but this is still a hard task. Also, because only points are considered in this article, which is easier than trying to partition large geographical objects. With geometric objects, an object can potentially overlap several partitions. With points, they can only belong to one partition. Note that several partitions can overlap one point.

This partition problem to some degree represented in R-trees as MBRs. There countless ways MBRs are built and the metrics they use, show that this partitioning isn't easy and there isn't one obvious way. Then the challenge reappears when you shall partition the tree over several servers in a good way. Who gets which part of the tree. A natural way is to give them each some of the top level MBRs, and the elements within it. Another problem that occurs is how to effectively restructure the tree.

2.2.4 Optimising for modern hardware

Solid state drive SSD

With today's prices on solid state drives (SSD), it's increasingly feasible and popular to use SSDs as primary storage. Some databases also go as far as to use RAM as primary storage [14, 7] as it is today also within reasonable cost and have certain benefits. When moving away from traditional hard disk drive HDDs, one should know the strengths and weaknesses of the new hardware in order to take full advantage of it. However, it is preferable with a structure that is storage agnostic and performs well regardless of storage type.

Besides the fact that there are no moving parts, the biggest difference is the read/write speed, as pointed out in [11]. When SSDs were new, it was a significant gap between read and write speed, meaning a read would be many times faster than a write. Earlier structures have focused on this characteristic and therefore traded writes in favour of reads. These structures typically had more reads and fewer writes than their original structure. Some such structures, as the OB^+-tree, is described in [11]. These structures would work poorly on an HDD. Today new SSDs have erased much of the gap between read and write speeds, though reads are still faster. The characteristics of the "new" SSDs are more similar to the HDDs. This means that the write/read load ratio is closer to one and new structures are better suited for to run on both storage types.

Parallelised tree

Another common approach today is to parallelise applications, making it run on several cores or machines. This is becoming more usual as the numbers of cores in a machine increase. It's also usual to distribute a database over several machines in order to scale up, know as *shared nothing* approach. Having a distributed database has benefits as being able to handle more requests and potentially better availability. Although this is an important field, it will not be the focus of this report. The focus is rather to optimise a structure within one machine, instead of creating a distributed R-tree. If one were to concentrate on distributing the tree, a good partitioning scheme would be necessary.

Chapter 3

Bloomtree

In this chapter, the Bloomtree is described, although a full description one should read their paper[11].

The Bloomtree was made in an attempt to optimise the B^+-tree for use with modern SSDs. Other papers have often had an approach where they trade of writes for reads, as SSDs often have had significantly faster reads than writes. With new SSDs, the difference between reads and writes are decreasing, thus such trade-offs aren't as beneficial as they once were. The Bloomtree they present is reducing both reads and writes. Although they focus on SSDs, it's intuitively good to reduce both reads and writes, regardless of storage medium.

3.1 Overview

In short, the Bloomtree is a B^+-tree that has modified the leaf nodes to morph between three different types. An illustrative example of the structure is shown in Figure 3.1. The two first, the normal leaf and the overflow-leaf, are both common and known. The third one is more unique, as it utilises a bloomfilter to reduce the number of reads. Besides these changes at the leaf level, the main concepts are left unchanged. The basic algorithms for the Bloomtree do need some changes to incorporate the different nodes.

Only the leaf node types will be described, as it's the only thing that has changed. Psudocode for their implementation of the different nodes can be found in their paper [11].

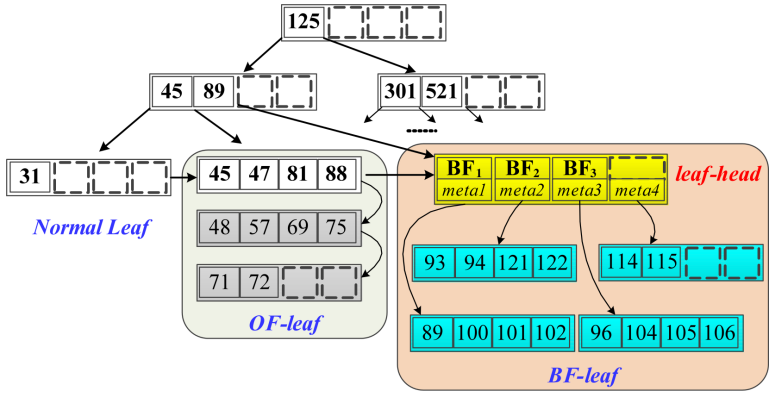

Figure 3.1: An overview of the three different leaf types. [11].

3.2 Leafnodes

The different types of leaf nodes work in a circular manner, where a leaf node can cycle between the three different types if it gets enough elements inserted. A leaf node starts as a normal leaf. When it's full, and a new element is inserted, it transforms to an overflow node. Similarly, when the overflow node is full, it transforms to a bloomfilter node. To complete the circle, the bloomfilter node transforms to several normal nodes and the transformation can start over.

3.2.1 Normal

This is just a normal node with nothing special about it. In practice, it's an overflow node that hasn't been overflowed yet.

3.2.2 Overflow Node

The concept of overflowing a node (or container) is a common concept. In this node, the overflow means one just extends the node instead of splitting it. The split would cause extra reads and writes, and could also propagate upwards and begin a larger operation. Therefore, by overflowing the node, one increases insertion performance.

On the downside, the search performance suffers. With overflow nodes, one would regularly have to access several of the overflow blocks to find the right key. As a mean, one would have to access half of the overflow blocks to find the key searched for. A worse case is when the key doesn't exist in the database, and one would have to chain through all of the nodes overflow blocks to confirm it.

When utilising only overflow blocks, one normally have a parameter that sets the max blocks it can overflow. Otherwise, one risk having to bad performance. Then eventually, the node become full and a larger operation has to be invoked to split it up and re-balance the tree. This operation can often be optimised, for example with bulk insertion, but one still trades short term insertion rate for larger operations later. The Bloomtree handles a full overflow leaf more elegant and changes the node into a BF-node.

They create a B^+-tree that only utilises the normal leaf and overflow block, called an OB^+-tree. Figure 3.2 shows the i/o difference between regular B^+-tree and the OB^+-tree. The OB^+-tree trades 290% more reads for 17% less writes. This is clearly not ideal, and we see that one would have an unrealistically skewed read to write ratio for it to be beneficial. Thus just adding overflow blocks aren't good enough for B^+-trees.

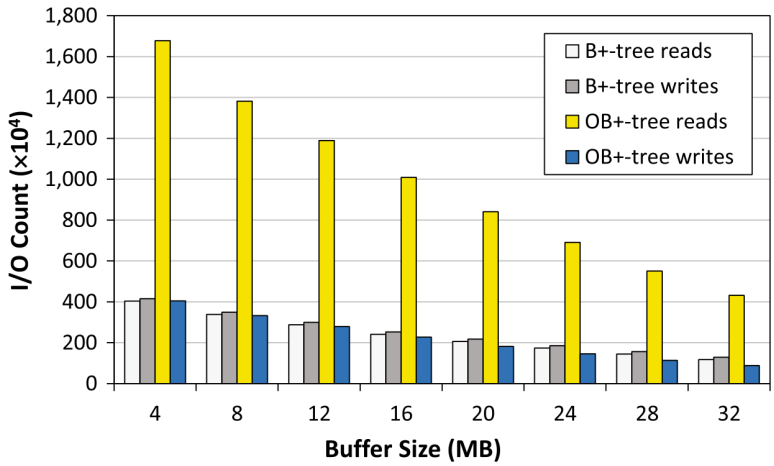

Figure 3.2: Performance for B^+-tree and OB^+-tree. Figure from [11].

3.2.3 Bloomfilter Node

The unique part of this tree is this Bloom Filter node (BF-Node), and it's where the bloomfilter is utilised. As figure 3.1 shows, it contains two type of leafs. It has exactly one head-leaf which then points to nodes that contain the posts.

The head-leaf is pointed by an internal node and is an intermediate node before reaching the posts. It is within this head-leaf that the bloomfilter is used. The head-leaf contains pairs of/with bloomfilters and pointers to nodes. The head-leaf has one of these pairs marked as active, the node that corresponds to this pair receives new elements that are assigned to the BF-node. When the active node becomes full, it becomes solid, and a new node takes the role as active. Also, the pair that points to the newly solid node gets its bloomfilter made from the keys contained in the node.

When an OF-node transforms into a BF-node, it is initially filled with overflow blocks

from the OF-node. Each block becomes a solid node and gets a pair of bloomfilter and pointer in the head-leaf. This is an elegant transition from the OF-leaf.

By having this bloomfilter for each solid node, a search that arrives in the BF-node checks the bloomfilters to see if there is a match. In that chase, it follows the corresponding pointer and reads the value. If no bloomfilter matches, the active node is looked up, as it might contain the wanted post. Also, a bloomfilter is prone to false positives, meaning that sometimes nodes is searched only to find that the key isn't there. However, one can tweak the false-positive rate by changing length and number of hash functions, so the number of false positives is low. It appears to be a smart way to reduce the number of reads without balancing the tree, and therefore one of the main factors that motivated this paper. It also seems to be a good idea to introduce the bloomfilter at leaf-level, instead of one for the whole tree. If one have a varied workload with a good portion deletes, a bloomfilter for the whole tree would have suffered because one can't delete from a bloomfilter, so one would have to accept a higher rate of false positives.

When a BF-node is all filled up with solid nodes, it can buy some extra time by reactivating solid nodes if elements has have deleted. However, eventually, it gets full and can't receive more elements. Then the contents of the BF-node is converted into several normal leafs. This happens by gathering all elements the BF-Node has, sorting them, and then splitting them in chunks that fit a normal leaf. All of these normal nodes gets inserted to the internal node that held the BF-node that got split, and of course, the new key-value pair/post gets inserted. If the internal node can't fit all of them, then it splits in normal B^+-tree fashion.

3.3 Findings

When the tree is compared to different other trees, across different workloads, it performs very well. They conclude that their Bloomtree can reduce overall runtime for both SSDs and HDDs. Several of the other structures use overflow pages as well, and they have significantly fewer splits than those that don't use it. However, just having overflow pages isn't enough. Their results show that some of those perform worse than the standard B^+-tree, which actually performs quite well compared with the different variants.

From this one concludes that the BF-node makes a huge difference, and therefore one should attempt to make a BF-node equivalent for R-trees in search for similar improvements. This is the focus of this paper, which tries to use this BF-node in R-trees.

Chapter 4

Related work

R-trees have become a well-studied structure. Just the amount of R-tree variations alone confirms this. The book ”R-trees: Theory and Applications” [13] from 2006 finds 70 R-trees worth mentioning in the period 1984 to 2004. Many have made improvements on the structure, using different metrics and valuing different properties. To my knowledge, no papers have directly focused on insertion speed. They often include a correlated measurement, I/O needed for operations, but not insertion speed directly. The closest paper is probably the bulk insertions papers such as [6, 12], where it can be strongly argued that they are somewhat onto this problem though their focus doesn't lie directly on insertion speed.

4.1 R*-tree

This variant came in 1990 [1] and is among the first variants of the R-tree. It is a dynamic tree which differs in what metrics it values. By using other metrics, it creates a better structure than the original tree and outperforms it in query performance. The metrics that the R*-tree values are:

1. Minimise MBR size.
2. Minimise overlap among MBRs.
3. Minimise circumference of MBR.
4. Optimise storage utilisation.

These metrics comes in play when splitting nodes and insertion elements. For example, when determining how to distribute elements among the two new nodes, the split is optimised according to these metrics. The metrics are also used to find the best place for an element to be inserted. Compared to the basic R-tree, the R*-tree uses slightly more computational resources to handle these more advanced metrics. This extra computation results in a better tree structure. In addition to these values, the R*-tree uses something called forced reinsert, which further improves the structure. The idea is that re-insertions can restructure the tree and make up for bad structures that have occurred due to the order the elements were inserted.

The reinsertion is issued when a node is becoming overfull. When this happens, the elements in the node get sorted by their distance from the nodes' MBR. Then the p (30% works well) elements that are furthest away are getting reinserted. If the same level becomes overfull again, then a split is done on the node.

By having this reinsertion, the query performance is drastically improved. But it comes at the price of a little higher computational usage. The R*-tree outperforms the standard R-tree (with quadratic split) in all queries the article presents and declares it-selves as better than the standard R-tree.

Figure 4.1 and 4.2 show the leaf level MBRs for the same 2000 randomly distributed points. The MBRs are shown as red rectangles, and areas where they overlap has a darker red. We see that the R*-tree has a superior structure, that just barely overlaps a couple of places. In contrast, we have the linear R-tree with a lot of overlap, which results in more search paths for a query Q. In Chapter 7 we see just how much this affects nodes accessed during searches.

Later, a Revised R*-tree[3] has been introduced by the same authors. This revised version re-engineers some of the core algorithms, making it better. Improving search performance and reducing i/o usage, CPU and creation time. One of the key changes is to remove re-insertion, which also makes the R-tree more suited for DBMS.

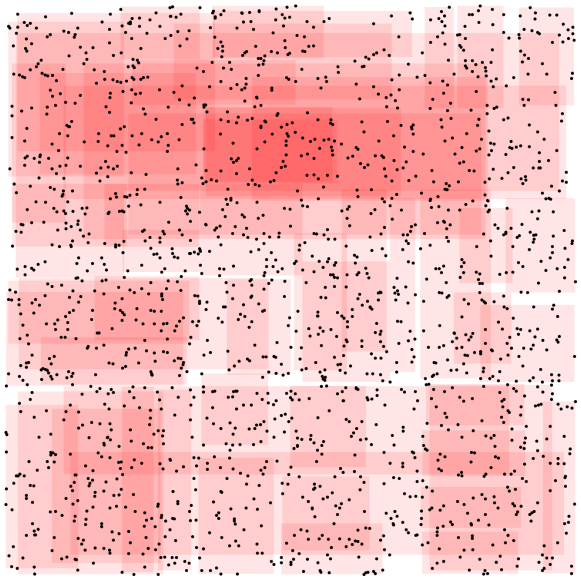

Figure 4.1: Linear R-tree. Leaf level MBRs for 2000 points.

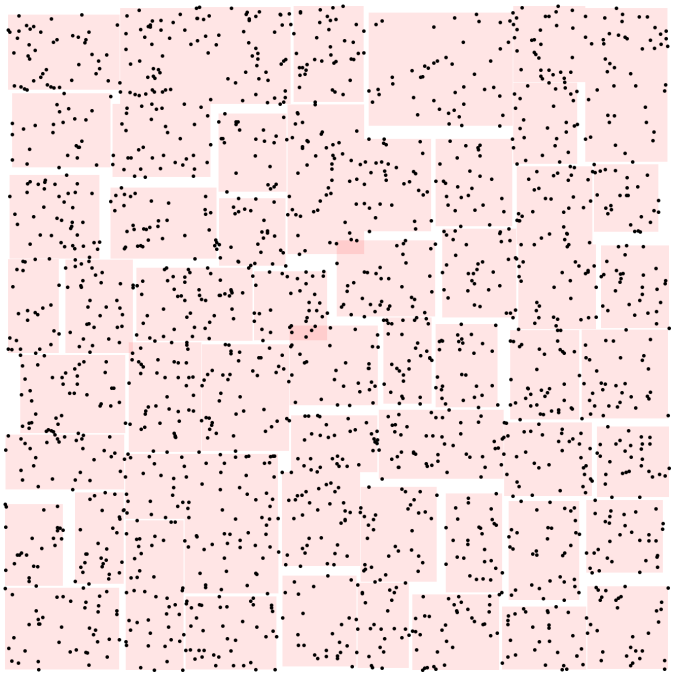

Figure 4.2: Linear R-tree. Leaf level MBRs for 2000 points.

4.2 Generalized Bulk Insertion

The article on Generalized Bulk Insertion (GBI) [6] is about effectively inserting larger amount of data without having a significant ”downtime” or significantly degrade the structure. The motivation behind GIB is quite similar to the motivation behind this paper. Previously the same authors have released a paper about Small Tree Large Tree (STLT) bulk insertion [5]. This method worked well for highly skewed data, but significantly degraded the tree for other distributions. Their paper on GBI improves the former STLT technique and makes the method work for all distributions.

The central principle behind GBI is to group up insertions and making new small R-trees that in the end are inserted into the existing large R-tree. This way insertions are handled is what differs from the original tree, as insertions consist of a ”clustering phase” and an ”insertion phase”. The process of inserting a point to it becoming part of the large R-tree is illustrated in Figure 4.3.

Insertions are processed before eventually getting inserted in the large R-tree. This way the R-tree won't have to do a bunch of disk IO for each element inserted. The insertions are instead grouped by clustering methods and from each cluster an R-tree is created, called small R-tree, which then is inserted to the existing R-tree. This clustering is essential to maintaining a good structure in the large R-tree as it makes sure that the elements in the small R-tree don't are too sparse. For the previous STLT method the input wasn't clustered, and the small R-tree could contain highly sparse data points. In the GBI approach, some elements might not be included in any clusters, and these are called outliers. If they were to include these outliers in the clustered R-trees, the structure of the tree would be significantly damaged, as the MBRs would have been enlarged and the content would be more sparse. Therefore outliers are inserted one by one into (OBO) the large R-tree in traditional fashion. Figure 4.3 shows both insertions of small R-trees and OBO.

In the article, they don't implement any clustering method on their own but uses MacQueens k-means method as it takes parameters to control favourable behaviour. Such parameters are how close elements in a cluster has to be and max and min size of the clusters. Tweaking these parameters results in variations in the clusters compactness and size. Demanding to compact clusters results in few R-trees and most elements will be inserted one by one (OBO). In this case, the GBI won't give much performance improvement. On the other hand, too loose clusters will give a highly increased input performance. This is because most elements will be inserted as part of as small R-tree. The downside with too loose clusters is that the R-tree gets a bad structure and queries suffers.

During an insertion, the small R-tree is treated as one element with MBR equal to the MBR of its root node. Then one searches the large R-tree for a suitable place to insert it. This search uses the principles of the standard insertion algorithm with some small adjustments. It searches for a suited internal node at the correct height ($h_{largetree} - h_{smalltree}$) so the tree remains balanced after the insertion. If this node is full, different techniques can be used to make space for the small tree. Etc merging, splitting and reinsertion.

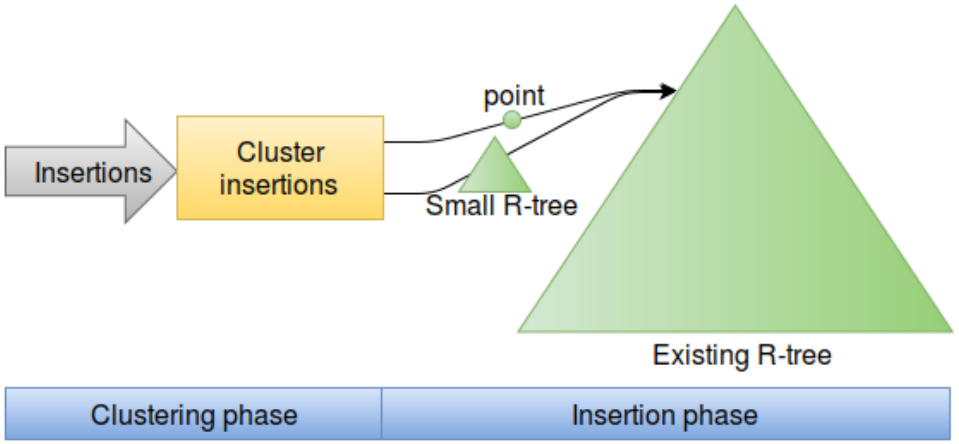

Figure 4.3: Process when inserting points in a bulk insertion structure. Inspired by [6].

Even with the improvements from STLT to GBI, there are issues with this approach. One is that the small R-trees are made without considering the structure of the large R-tree. Therefore the MBRs of the large R-tree might be expanded quite a bit. In some cases, this could be prevented if the elements would fit better in other nodes in the large R-tree. This can, for example, happen if the small R-tree is inserted near the edges of an MBR and the MBR has to be expanded. This issue results in a degraded structure in the R-tree.

Another issue is that the large R-tree might become invalid after inserting a small R-tree. This happens when the root node in the small R-tree doesn't have enough elements, meaning it has less than the threshold m elements. When building the small R-tree it is legal to have less than m elements in the root node, but it isn't allowed in an internal node. This isn't a critical issue, but can to some degree damage query performance and disk utilisation.

Results show that the insertion cost with GBI is significantly lower than OBO insertions. The GBI also scales better with the tree size in regards to insertion cost. The downside is that query performance from an R-tree using GBI is slightly worse than the one using OBO. The trade-off between insertion cost and query performance is dependent on the parameters for the clusters. The paper means that it's within an acceptable performance, trading slightly worse query performance against significantly lower insertion cost.

4.3 Bulk insertion by seeded clustering

The bulk insertion by seeded clustering method [12] strongly resembles the GBI from the previous section and can be viewed as an improvement. It has the same idea of gathering insertions in clusters and building R-trees before bulk inserting them. Still, there are some

significant differences.

The key difference is that this methods take into account the structure of the existing R-tree when the clusters are made. This structure awareness comes from copying the top k-levels of the existing R-trees, where k is a chosen parameter. This copied tree holds the MBRs of the top nodes and guides where the new elements should be inserted, and call this tree the *seed tree*. When a new element arrives, it gets "placed" in a suited leaf node in the seed tree. This selection of a suited node is done in the same manner as regular insertions. Thus the elements are getting inserted closer to where they would have been inserted if in OBO, although elements are only ensured to follow "right" path for k levels. If an element doesn't fit in any of the leaf nodes, it gets inserted into the existing R-tree in a regular OBO fashion. By considering the structure of the existing tree, the MBRs in the tree are expanded less than with GBI. This results in a better tree structure.

All the elements that end up in the same leaf node in the seed tree are assumed to form a cluster. Each of the clusters is then made into a small R-tree, and then the small R-trees are inserted one by one in the existing R-tree. Just as in GBI, it's important to insert the small R-trees at the right height. Because the clusters were made using the seed tree, it already knows which node is the designated insertion point in the large R-tree for each small R-tree. Still, with the designated insertion point, there are three different scenarios that need to be dealt with. One is that the root node in the small can create an illegal R-tree if inserted into the large tree. This is because it not necessarily meets the min_fill degree an internal node requires. While the GBI approach ignores this, the seeded clustering approach deals with it. The two others are if the small R-tree is too high or too low for the selected node. These cases are dealt with. Therefore, one never get an illegal R-tree as in the GBI approach.

Another important thing to take into consideration from this approach is that it locally reduces overlap. As one can expect when using the *seed tree* there can still be much overlap between the small R-tree and existing entries in the designated insertion point. To decrease the overlapped area, this approach uses repacking which locally minimises the overlap. In order to achieve this, elements from the designated node that overlaps with the incoming R-tree are recursively repacked bottom-up. This improves query performance drastically, actually to the point that it provides better performance that an R-tree created in OBO fashion. It is really impressive that it beats the performance one gets from R-tree made in OBO fashion, in both insertion and query performance. Although it beats the regular R-tree, it would be interesting if they had experiments to show how it held up against an R*-tree with OBO insertion, which in short is a better structured R-tree. The R*-tree is explained in section 4.1.

They don't discuss the durability aspect of having elements grouped in memory before insertion. Nor do they say if the elements in memory are retrievable, or only elements in the large tree can be retrieved. I assume that both of these remarks are handled as they are quite common. One could, for example, do as in Googles LevelDB [9]. Meaning having a log file for durability and make queries look up the part in memory for consistency.

Chapter 5

Implementation

This chapter describes the implementation done in relation to this paper. The goal was to make an R-tree, with properties from the Bloomtree described in Chapter 3. Specifically, it was the use of three different leaf nodes that was found appealing. Unfortunately, no solution on how to apply the BF-Node to an R-tree was found, and the challenges that the BF-Node imposed is described as well. Without the BF-Node the paper ended up with a structure that only utilises the normal and the overflow node. Therefore, this structure strongly resembles other research conducted[16] although the angle is different.

5.1 Overview

The purpose of this paper is not to create a full featured R-tree. The idea is to check out if the concepts from the Bloomtree can be applied to an R-tree, and if they give the same performance increase. Therefore there is no attempt here to optimise the tree, and for the same reason, no advanced variant of the R-tree is used. Simply the basic R-tree and the R^*-tree is used to give a wide range of performance within the dynamic structures. To test with more advanced structures would be a task for further work.

As mentioned above, only normal and overflow leafs were possible to be transferred to the R-tree. Therefore cutting the leaf transformation from Normal $\rightarrow$ OF $\rightarrow$ BF $\rightarrow$ Normal down to Normal $\rightarrow$ OF $\rightarrow$ Normal.

5.2 Challenges

Creating an R-tree with the Bloomtree characteristics turned out to be harder than expected. Because the R-tree has so much resemblance to the B^+-tree, one would think that it should be well manageable. However, there are many differences when digging into it. This section cast light on those differences that makes the implementation challenging. Note that it's, in particular, the BF-node that is challenging to implement, if not impossible.

5.2.1 Different type of data/storage

The B^+-tree is a key-value storage, where one typically asks for a value that corresponds to a key. The R-tree on the other hand is often used as a spatial access structure where the search is defined in several dimensions. The R-tree search can return an arbitrary number of elements. This difference causes the challenges in the following subsections. One shouldn't expect less when trying to adapt some features tailored for one structure, into another type of structure.

5.2.2 General optimisation

This problem is in general covered when making an n-dimensional structure from the B^+-tree, where the solution is to make use of MBRs in a hierarchy. Along with the MBRs, different techniques/ideas on how to manage them best. We don't look into this challenge in this paper, as it's not part of the scope. Section 2.2 describes how the R-tree works, it doesn't focus on *how* the challenge was solved.

5.2.3 BF-Node

This node poses the biggest challenge in this paper. As the part that really differentiates the Bloomtree from other B^+-trees, one would think much of the optimisation can be credited this node. It's therefore truly unfortunately that no suited equivalent for it was found for the R-tree.

The problem is that a bloomfilter for the type of data in an R-tree is completely useless. This is due to the way it's searched and the data it contains. In the B^+-tree, the keys are excellent things to search for. They are discrete and unique within the tree. When used in the BF-node in the Bloomtree, at most one data-node can contain the post. Such use is ideal for a bloomfilter, that excels at use-cases where it quickly can state with certainty that a node doesn't contain a key. The number of false positives it gives can be tweaked to be non-significant.

In the R-tree one doesn't search for specific values, instead one queries an area that returns a set of posts. Searching one and one value simply doesn't fit the R-trees use-case. When searching for posts within an area Q, it's not feasible to check every possible value in Q against the bloomfilter. For an R-tree with coordinates it would be too many possible values to test, depending on Q and the granularity of the coordinates. This makes a bloomfilter completely useless for an R-tree.

Obviously the bloomfilter has to be replaced with a suited structure, but still, the different type of search is a problem. Even with a head-leaf that uses suited filters, one can expect access several data nodes for a regular search. This is because the R-tree returns an answer set instead of at most one answer. One can't stop after the first match but instead continue the search through all of the filters, and their data-nodes if the filter matches. Because of this, the BF-node might not yield a significant performance boost for the R-tree, if any at all. There also exists a spatial bloomfilter [15], but it doesn't quite fit the needs.

Another part that one would need to be sorted out, is the transformation from BF-node to a normal leaf. Where one can find inspiration in various R-tree variants, especially those using bulk insertion to insert branches into an existing R-tree. For example is the Large-tree/Small-tree a place to take inspiration, there a small tree is constructed in memory and inserted into the large tree as a branch at the right height. This variant is described in Section 4.2.

Naive filter

Ideally, a new filter should be able to handle n dimensions, even tough this paper looks at 2D points. Figure 5.1 shows a simple approach. It uses a grid that divides the points into sections that are represented as a bitstring. If a point is in a section, the corresponding bit is set. Given the parameter $n = 3$ and the length and width of the MBR as l and w, respectively, all of the n^2 sections are of equal size with width w/n and height h/n. The n parameter is up for exploration. This filter will also be created when a node becomes solid, just like with the Bloomtree. Creating the sections relative to data-nodes MBR means that the MBR can't be allowed to change afterwards. To deal with this restriction, it's best to use the MBR of the node that became solid. This also makes the sections for the bitstring as tight a possible. The filter-pointer pair will then have to store this MBR as well. If one were to use the MBR of the head-leaf, the BF-leaf would not be able to accept new points that expand the MBR.

Depending om the overlap between the query Q and the MBR of the node, some sub-rectangle of the filter has to be checked for occupied sections. This is done by checking the bitstring and requires some extra CPU computation as it must calculate which bits to check. If there's a match on the filter, then one has to explore the pointer that corresponds to the filter. There is no need to check the bitstring if the overlap is so large that all the sections would have to be checked. In this case, it's guaranteed that there's a match in the filter, and the pointer is explored.

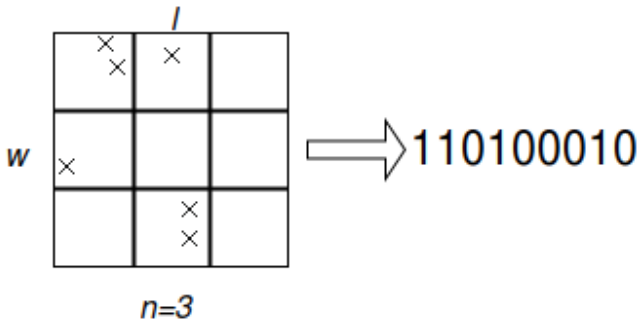

Figure 5.1: A simple grid filter with $n = 3$.

This filter is also prone to false positives, in the scenario where a post is deleted without updating the filter. There are some problems with this filter. The first is that several filters in the head-leaf can have a match, meaning we have to access a set of extra nodes in addition to the head-leaf. Without quantifying it, this set can be the majority of the nodes. With more points in a node, the chance that the filters are full of 1's increase. This could kill performance as it basically just forces and extra read with the head-leaf and in it selves gives a strong case for not using BF-leafs.

It's clear that having matches in many filters and accessing many nodes defeats the purpose of having the BF-leaf. The algorithm described up until now uses an active node designated for writes. Then a uniform distribution of points would have the least use of the filter. This is simply because uniformly distributed points in each node would give more or less similar filters (depending on the filters granularity given by n). To deal with this, one could try to remove the idea of active nodes. Then introduce a metric to select which node to insert a new point in. Such metric would use ideas from the R-tree, such as potentially expand the filter the least. In practice, this will try to fill each node with points that cover a few sections in the filters grid. By using filters in this way the overlap between the filters will decrease, and as the usefulness will increase. A few problems with this is that when many of the nodes are full, then the new points have to be inserted where there is space, regardless of how it affects the filter. Another issue could be low storage utilisation in the event that one have many nodes that only contains a few points. It's not given that this will be a huge issue as they get filled up over time, and have the benefit of accepting insertions without reconstruction. A third issue is that the overflow blocks can't be transformed into data-nodes as they are, this could cause bad filters. This third issue would be solvable by sorting the posts, for example by clustering or tree-construction, but this would give many extra writes in the transformation. The advantage is that these sorted data-leafs might be good enough to transform directly to normal leafs. Effectively changing the "restructure step" from BF $\rightarrow$ Normal to OF $\rightarrow$ BF, and the total number of operations might not differ too much.

It would seem like this naive filter would perform badly, and one would not want to implement it. Also, the approach looks fairly similar to an R-tree, almost like having a fixed

grid instead of nicely fitted MBRs. As they look pretty much the same, it seems saner to just drop the BF-leaf and continue to use the regular approach of R-trees with MBRs. That beeing said, if a suited filter is found, it would be highly interesting to explore.

5.2.4 Overflow blocks

This isn't really a new concept for an R-tree. One can almost just use the one from the Bloomtree directly, with minor adjustments to make it work with an R-tree. Using an overflow blocks sort of violates the R-tree principles of min_fanout, as the overflow blocks get created with one post in them.

Search performance

Compared to the Bloomtree more blocks has to be searched on average, indicating worse search performance. For an R-tree one has to search through all of the overflow blocks, while the Bloomtree has to search through half of the blocks on average. This isn't that bad as one average would have to access some extra nodes anyway, in comparison to the Bloomtree. That some extra nodes have to be searched is "normal" for the R-tree, and isn't necessarily that bad.

Insertion performance

The main reason to use overflow blocks is to avoid or at least postpone restructuring of the tree. This saves a lot of writes as one can just add a new block and start filling it. The alternative is to split the node into two half-full nodes, meanwhile also possibly causing some restructuring. A downside is that overflow blocks do increase reads. It increases the number of reads during a search and also the number of reads during an insert. This is explained as having to read through full overflow blocks before inserting into the available one.

One could try to make a system to avoid reading through full overflow blocks during inserts, for example by having active nodes such as in the BF-node. However, this means more bookkeeping and it needs an equivalent to the BF-nodes head-leaf.

Transforming to a BF-node

This wasn't an actual problem as the BF-node wasn't realised. However, if it was, challenges would occur. It's not given that it would be as be as easy as just make each block from the overflow node into a data leaf and create a leaf-node for them. Say we make a

filter like the naive described above, one would want to distribute the posts differently between the data-nodes. It would be good to be inspired by clusters, so that the posts within a node is clustered. To make the filters less sparse, and in turn making the BF-node more effective. Although the clustering can be done in memory, the overflow blocks has to be updated and that cost a write each.

Transforming to normal node

This challenge is a result of the BF-node not being implemented. It is similar to what a transition from a BF-node to normal leaf would have been. There are minor considerations on what to do.

One could use the Small Tree, Large Tree method mentioned above. The method chosen in this paper is almost the same. First, it makes a tree in memory and then inserts the leaf nodes from it, one by one. This lets the existing structure decide where to put them, instead than forcing in a branch that might cause much overlap. For the sake of the structure, it would, of course, be better to insert one by one, but it comes at a high i/o cost.

5.2.5 Variants implemented

In this paper, four R-trees are implemented. Two regular ones, each with their overflow counterpart. These two regular ones represent two different *tiers* of R-trees.

The first one is the basic R-tree, with the linear algorithm used. It's not known to be a highly performing R-tree search wise, but being fast at insertions. Fast in the sense of runtime, not i/o. The second one was the well know R*-tree, described in Section 4.1. These mainly differ in the way insertions and splits are handled.

Both of these trees have an overflow version as well. The only difference is that an in-memory tree is made during the transaction from overflow node to normal node. They use their respective tree, meaning the linear overflow tree builds a linear tree, and the R* overflow tree uses a R*-tree. This gives an additional edge to the R*-tree.

The overflow R*-tree also have one significant difference from the regular R*-tree. The overflow version doesn't have forced re-insertions at the leaf level, as it simply overflows when it gets full. And when the overflow node is full, it gets transformed to normal leafs. In the transformation, a normal R*-tree that uses re-insertions are used, but that doesn't affect the read or write count. The re-insertions are something the paper [1] emphasises on as making sure the tree doesn't suffer from early inserted posts. It's therefore interesting to see how the overflow R*-tree performs when it loses this key behaviour. It still uses more advanced metrics, so it should outperform the linear version.

5.2.6 Cache

Today, practically all systems have some form of caching that saves the system from unnecessary reads. Due to the forced re-insertions in the R*-tree, it is more prone to a high read count. Most of these re-insertions pretty much follows the same path, so a cache would reduce the reads significantly.

In order to get a more real simulation, a virtual cache is implemented. Simply as a structure that holds an LRU queue, in addition to counters for reads and writes. The structure has a read method, that checks if the node asked for is in the LRU queue. If it's not, the read count is incremented. Either way, the node is placed at the beginning of the queue.

During a write on a node, a read is first done in the fashion above, and then the write count is incremented. This way, written nodes ends up first in the LRU queue.

For the overflow nodes, each overflow block is treated as a node in the cache. This means that when an overflow block with size 6 is searched, one will potentially get 6 new reads. The overflow blocks are in the same way stored in the cache.

During this paper, the root node is assumed to always be cached. The root node is accessed all the time during insertions and searches, regardless of tree type. Therefore this doesn't give skewed results but helps normalise them. Furthermore, it would be natural in a real application to force the root node, and perhaps a few more levels, to always stay in cache. In some papers, they have gone further and assumed that all intermediate nodes are stored in the cache.

5.2.7 Choice of language

Implementation is done in Python. Measuring run-time in the python environment isn't that constructive. If it was to be used in a real application, one would probably implement it in a more efficient language as C/C++. Python works fine for measuring simulated discs accessed, though it is a bit slow. Much of the insertion is "number crunching" which it isn't really suited for, and one notices a significant difference in runtime for the different trees.

Chapter 6

Datasets and metrics

6.1 Datasets

There are many ways to create datasets. It is possible to make synthetic ones, or use a subset of real data. With the synthetic ones, it's possible to make specialised sets with certain distributions. Some of these distributions can be crafted to evaluate how the structure performs for some unusual distributions. It's easiest to use already made datasets, such as [4].

The data sets that are used is in this paper is from 'A Benchmark for Multidimensional Index Structures' [2]. The same data sets are used when evaluating the Revised R^*-tree [3]. In the paper, all the distributions and their corresponding query sets are explained. With 28 datasets containing a variety of real and synthetic data, points and rectangles and up to 26 dimensions, it's able to suit many needs. Even with its vast selection of datasets, only some of them fit because the experimental structure only handles points in 2D. Bit02 and Uni02 distributions are used.

The datasets include insertions and searches, which cover the majority of R-tree usage. It left as further work to see how well the structures hold up when the dataset is a mix of insertions, deletions, searches and updates.

6.1.1 Bit02

Bit02 is a point distribution in two dimensions. It also comes at bit03 and bit09 which are 3 and 9 distributions, respectively. Three query sets follows for each dataset, where they

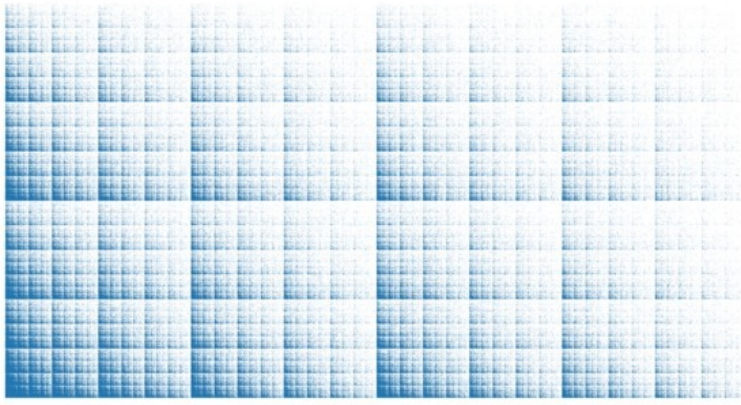

Figure 6.1: The Bit02 dataset. 1.000.000 2D datapoints.

are different in that they return about 1, 100 and 1000 answers per query. The number of queries for the querysets decrease as the number of expected answers increase. This is done to limit the query cost. Also, the points are listed in the query set randomly. This means that the insertion of points is random instead of starting from one corner.

6.1.2 Uni02

Same as with Bit02, only that the distribution of the points are uniform.

Figure 6.2: The Uni02 dataset. 1.000.000 2D datapoints.

6.2 Metrics

In this paper, only reads and writes are measured. This is done entirely simulated in the program, and everything is contained in RAM. Therefore one count reads and writes as if they would have interacted with permanent storage. This is done in the manner where accessing a node produces a read, while updating or creating a node causes a write in addition to a read.

In some ways, just monitoring reads and writes can be viewed as unfair to some of the structures. Especially since one often has some kind of cache layer that reduces the amount of disk reads required. For example, the reinsert function in the R*-tree will cause it to have extra reads. In reality, one would have a cache that would be able to eliminate many of those reads. For example, the Revised R*-tree paper [3] assume that all non-leaf nodes fit in memory. This eliminates a big amount of the reads. Therefore a cache is implemented to simulate such behaviour.

Due to the language which the structures are implemented in, runtime isn't a viable measure. Python isn't known for its speedy performance. It's a general purpose high-level language that has been widely adopted the last years, also in scientific areas. Python does provide excellent performance for certain intensive tasks trough bindings to C/C++ libraries, meaning the heavy lifting is done natively in C/C++. In this paper no such libraries were suited, so the runtime isn't much to look at. If implemented for production, the structure would probably be implemented in a more bare-metal language, where a natural choice is C/C++.

Other measures besides runtime and i/o do exist. Some are mentioned in previous sections as metrics different R-tree variants value. Many of these are easy enough to simulate with software, one could, for example, measure storage utilisation. Other more structure oriented metrics is also possible, for example, total overlap and circumference. These metrics all correlates with each other in varying degree. The read and write count provide a good and easy measure for the structure's performance. Also, the bottleneck often is at the i/o stage, therefore the only measurement in this paper is read and write count.

Chapter 7

Results

7.1 How test where done

The different structures were run several times with different datasets and parameters. Although the R-tree generally isn't deterministic for a dataset, the posts in the datasets were inserted in the same order. Therefore only one run per configuration was needed. It also makes the results reproducible and more correct, not risking that any of the runs are lucky or unlucky. Tables with exact values for runs, which the figures in this chapter are based on, can be found in the Appendix A. In the figures, the labels "$type - oX$" reefer to an overflow version of $type$ with max allowed overflow blocks X. Linear and Rstar is the linear R-tree and R*-tree, respectively.

The computer used was running *Ubuntu 16.04* with *i7-4770 CPU @ 3.40GHz* and *16GB RAM*. Without measuring runtime, no special percussion was needed while running. The different configuration could run in parallel without affecting each other results. Although runtime isn't measured, for reasons described in 6.2, the R*-tree and its overflow variants took significantly longer time to run.

In Section 6.1 the distributions used here are described. Each distribution have three query sets, $q0$, $q2$ and $q3$. Reads for queries are measures as average per query executed, and not for the whole set. This makes it easier to compare since the sets have a different number of queries. The buffer is reset for each query set that are executed so that none of them has any advantages from a pre-filled cache.

7.1.1 Default configuration

Different parameters are changed for the runs. When nothing else is specified, the default parameters apply. These default parameters are shown in Table 7.1. Not all parameters affect every variant. The P grade is set as the recommended value in [1].

Note that even with cache size as 0, the root node is still cached. This mainly affects the R*-tree which would have an even higher write count, in addition to the 1.000.000 extra read count all of them would have.

Table 7.1: Default parameters for runs.

Configuration	Value	Note
Fanout	50	
Min node fill	40 %	
Max overflow	3	Applies to overflow variants
P grade	30 %	Applies to R*-tree
Cache maxlen	0	Affects read count

7.2 Changing cache size - Bit02

Here we see how much the cache size impacts the read and write count. The overflow structures are set to accept overflow up to three blocks, meaning that the overflow node in practice can contain 3*fanout elements. Except from the cachesize, the parameters are set to the default ones in Table 7.1. We use the bit02 distribution for this. Using the other distribution, uni02, while changing cache size shows the same trends, where the biggest difference is that the query sets start with fewer writes for cache size 0.

Figure 7.1 shows how many reads are needed to build the different trees when varying the cache size. The figure shows results as expected, the number of reads decrease as the cache size increases and evens out the read count. It's also very noticeable to see that even a small cache size helps dramatically in reducing the reads. For example, the R*-tree reduce needed reads by around 80 % when getting a small cache. This is probably due to re-insertions, which without a cache gives a loot more reads.

Further, we can see that with moderate cache sizes, the two with overflow have higher read counts than those without. Also, we see that the linear R-tree type beats the R*-tree respectively in normal and overflow variations.

When cache size is the only changed parameter, the write count stays constant for the runs. In Figure 7.2 we see that the overflow types cuts the write count by more than half. Another interesting thing to notice is that the write count for R*-tree and Linear isn't that much. One could expect it to be clearly different from the linear, due to the re-insertions, but the difference isn't that big.

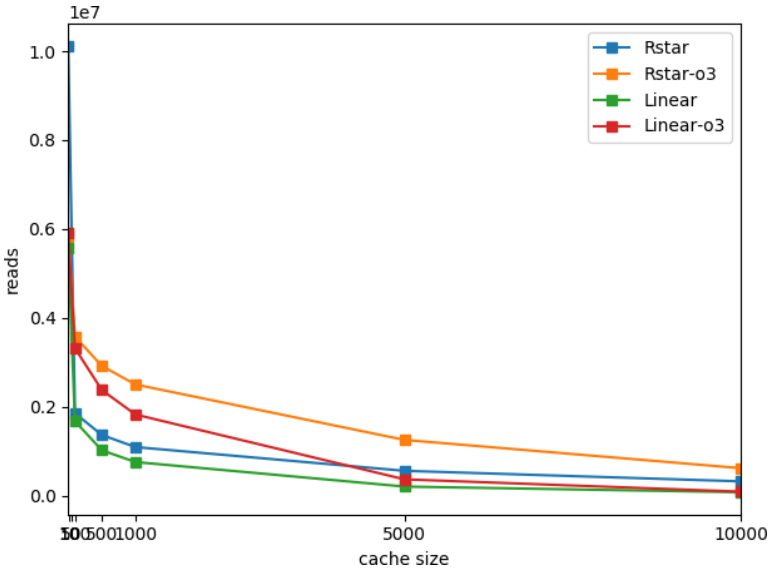

Figure 7.1: Reads used to create different structures from Bit02 distribution. Fanout set to 50. Rstar is the R*-tree. Rstar-o3 is R*-tree with max_overflow as 3. Linear is the linear R-tree. Linear-o3 is the linear R-tree with max_overflow as 3.

The trends for the three query sets are just like for $q0$ in Figure 7.3, with the same ordering. In fact, the lines don't cross at any point. The only difference is that the $q0$ benefits slightly more from the cache. We see this when comparing it to $q3$ in Figure 7.4 by $q0$ having a larger drop in the beginning. Another thing to notice is that both the R*-tree and the overflow R*-tree outperforms the linear versions, although the R*-tree is unrivalled at searches. The R*-tree doesn't in the beginning suffer from not having a cache, in contrast to the reads needed to construct the tree. This is because there is no re-inserts when searching.

From Figure 7.3 and 7.4 it's clear that the number of reads increases when the query size increases. This is of course expected, but we see how much a large Q can request in terms of reads.

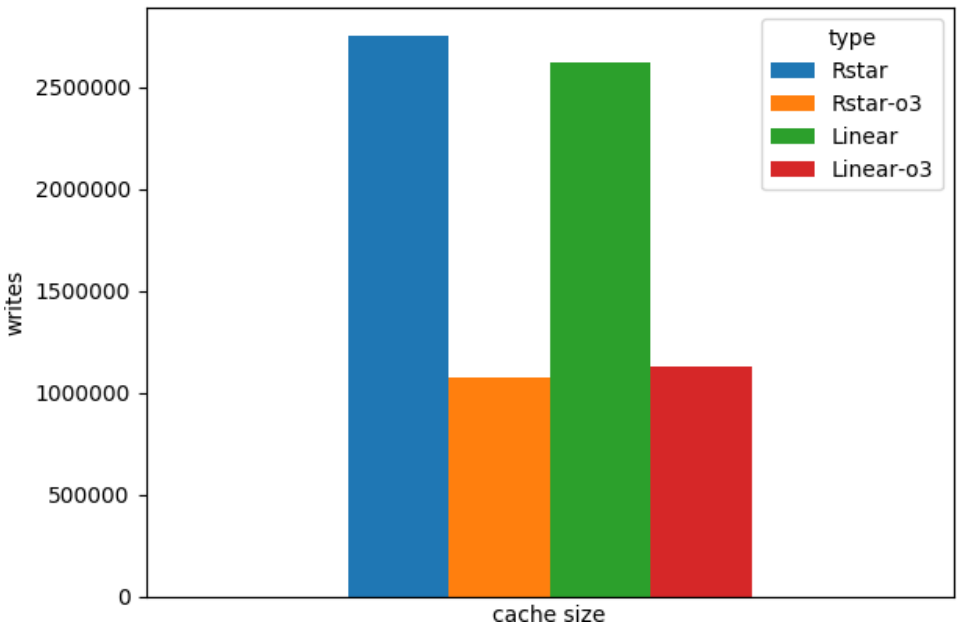

Figure 7.2: Writes used to create different structures from Bit02 distribution. Fanout set to 50. Rstar is the R*-tree. Rstar-o3 is R*-tree with max_overflow as 3. Linear is the linear R-tree. Linear-o3 is the linear R-tree with max_overflow as 3.

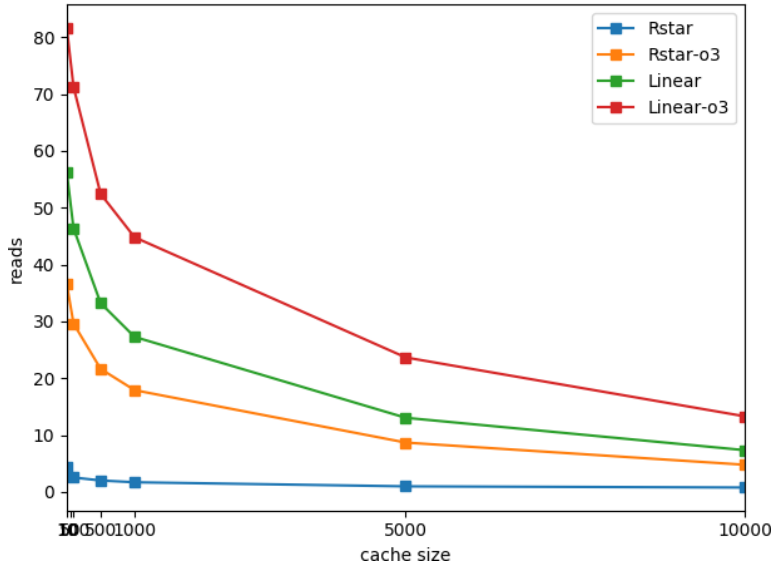

Figure 7.3: Reads per query in q0 from Bit02 distribution. Fanout set to 50. Rstar is the R*-tree. Rstar-o3 is R*-tree with max_overflow as 3. Linear is the linear R-tree. Linear-o3 is the linear R-tree with max_overflow as 3.

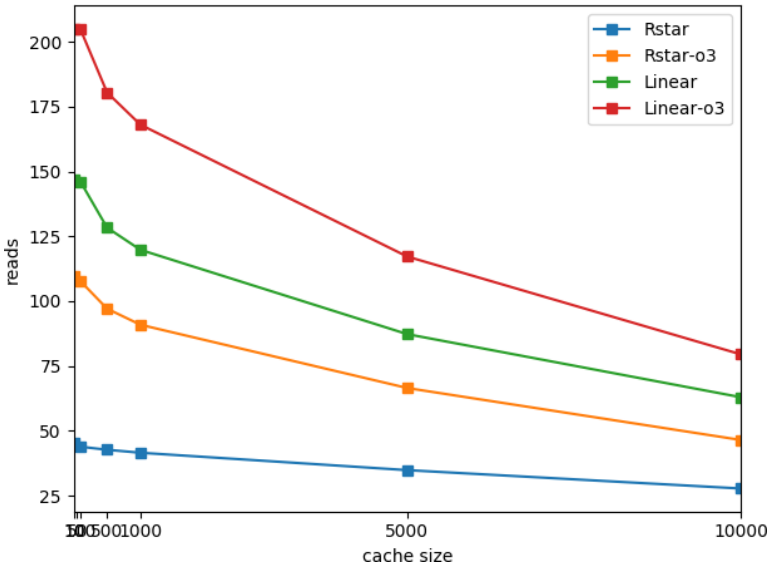

Figure 7.4: Reads per query in q3 from Bit02 distribution. Fanout set to 50. Rstar is the R*-tree. Rstar-o3 is R*-tree with max_overflow as 3. Linear is the linear R-tree. Linear-o3 is R-tree with max_overflow as 3.

7.3 Changing fanout - Bit02

7.3.1 Linear R-tree

In this section, we change the fanout for different R-tree variants. The different lines represents the tree variants, the normal one and overflow variants with different max overflow parameter. This shows how the combination of max overflow and fanout affects read and write count.

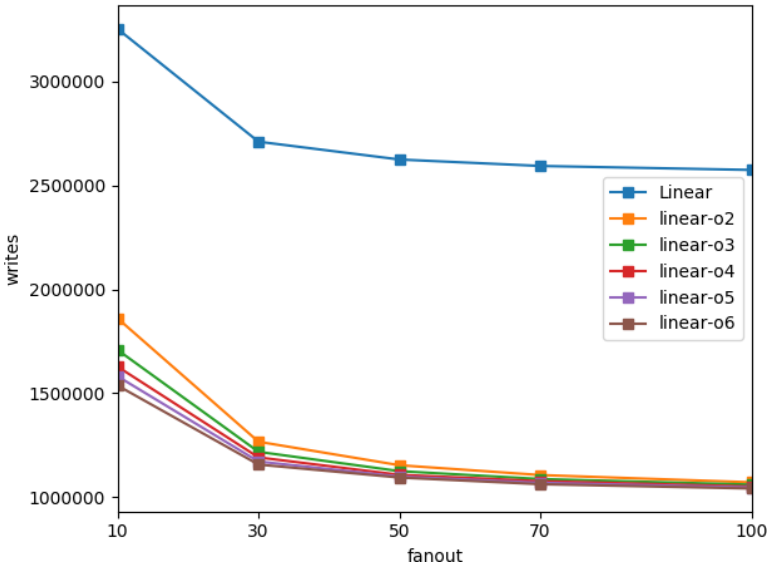

Figure 7.5: Writes used to create different structures from Bit02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X.

Figure 7.5 shows that all the overflow versions of the Linear R-tree have significantly less writes than the original one. This is what we also saw in Figure 7.2, only that we here see that it applies for all the versions and for all fanouts. We also note that the most significant change in writes is from fanout 10 to 30.

When it comes to reads needed to create the tree, they all follow the same trend. And it's also clear that the more overflow blocks give an increased read count. Figure 7.6 shows this.

From Figure 7.7 we see that one sometimes can be lucky and unlucky with the structure. An example is that linear is expected to consume the least amount of reads for all of the fanouts, but it has a spike at 70. Such spikes make it more difficult to read into the results. The graph nicely layered as in Figure 7.6, which fits the theory well. It's expected that reads would increase as one has to search through long overflow block chains.

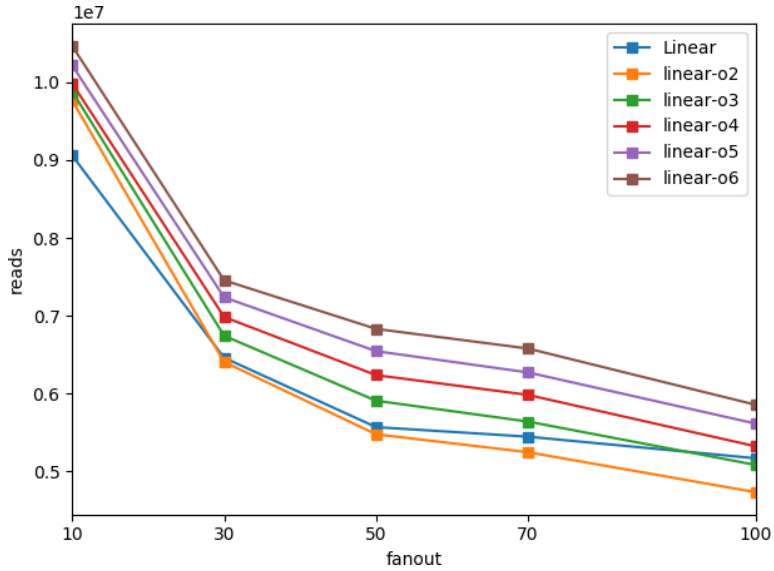

Figure 7.6: Reads used to create different structures from Bit02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X.

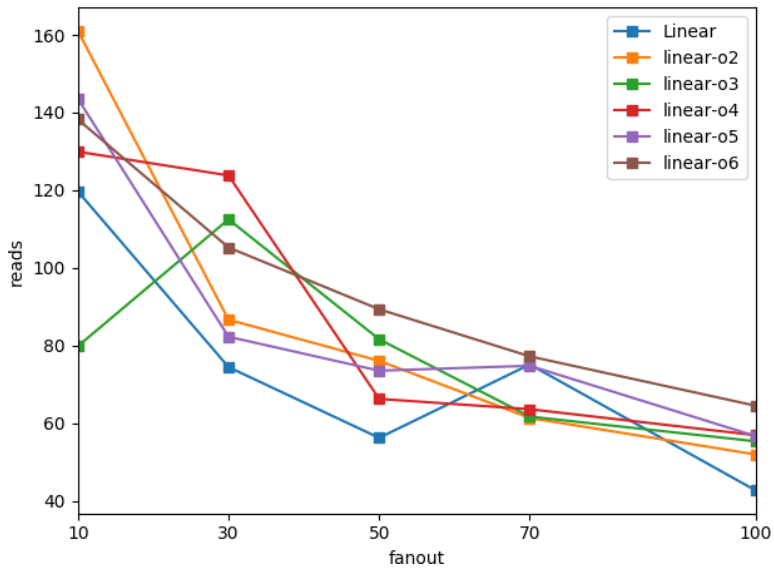

Figure 7.7: Reads per query in q0 from Bit02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X.

The same trend is also found in $q2$ and $q3$. The number of reads increases, but the relative variance among the lines decrease. Figure 7.8 shows the reads for q3, there we see that the

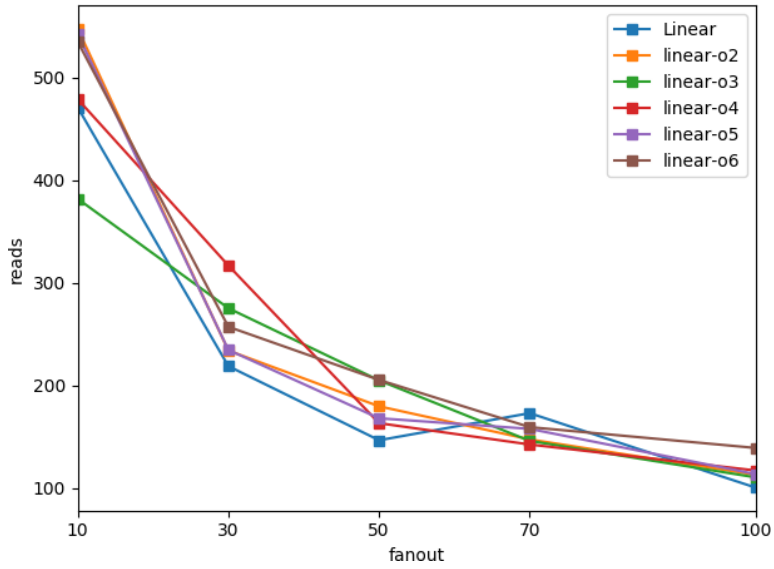

Figure 7.8: Reads per query in q3 from Bit02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X.

lines are more compressed together. Naturally, we still see the same spike for linear as we saw in Figure 7.7, as it's the same structure with just a different query set.

7.3.2 R*-tree

With the R*-tree versions, we again start by looking at the reads and writes required to create the tree.

Figure 7.9 shows the number of writes needed. We notice at once that there is one outlier, the standard R*-tree without overflow blocks. This one has significantly more reads than the rest which is grouped together lower down. We also see that the pattern follows our expectations that more overflow blocks mean less writes. When the fanout increases the difference between the overflow versions practically vanish. Thus the big difference lies in the fact that the standard R*-tree has re-insertions that cause a lot of more writes. It also helps a long way that the overflow version creates the new nodes in memory after an overflow leaf is to be transformed to a normal leaf.

Furthermore, in Figure 7.10 we also see that a pretty similar trend for the reads required to create the R*-tree. Again the standard R*-tree requires a significantly more i/o than the ones with overflow. We also see that the overflow lines are reversed compared to Figure 7.9, meaning that more overflow blocks result in more reads. This is expected, and the reasons are explained in Chapter 5.

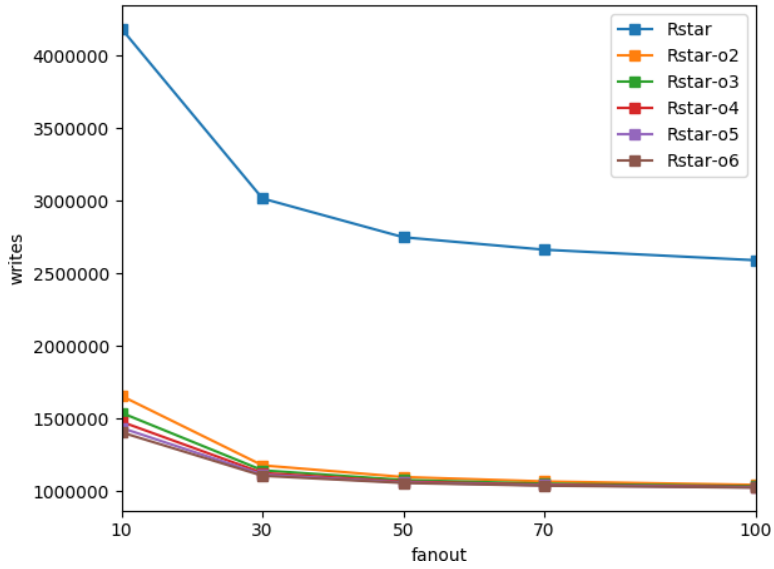

Figure 7.9: Writes used to create different structures from Bit02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X.

Keep in mind that we saw a big decrease in Figure 7.1 for reads by the R*-tree when it was aided by a cache. In Figure 7.10 the cache size is 0, so a similar improvement is likely to apply here given a cache.

The graphs in Figure 7.11 are unexpected. The standard R*-tree acts as normal and shows that an increase in fanout greatly increases the performance of the searches. This is at least the case when going from 10 and up to 50, after 50 there isn't any further improvement. The performance is great after 50. In fact, for $q0$, the Linear R-tree uses same roughly reads when fanout is 100, as the R*-tree uses when fanout is 10. This is a huge difference.

The overflow versions don't have any performance gain when increasing the fanout. In fact, it the performance is generally worse when fanout is increasing. We also find the trend that the overflow versions are stacked/sorted in the order of their overflow blocks, meaning Rstar-o2 at the bottom and Rstar-o6 at the top, this is expected. However it's surprising to see that the performance is worse for increased fanout, the overflow versions of the linear do benefit from increased fanout. This could come from the chaining of overflow block resulting in higher reads as one has to read through them all when an overflow leaf is accessed. Note that the performance is better for all fanouts than the linear versions have at fanout 100.

For $q2$, shown in Figure 7.12, the overall reads naturally goes up. Here the standard R*-tree still massively outperform the overflow versions. However, in contrast to $q0$ in Figure 7.11, the overflow versions actually reports an increase in performance with higher fanout. Not by far the same performance gain as the standard tree sees. It's interesting to see

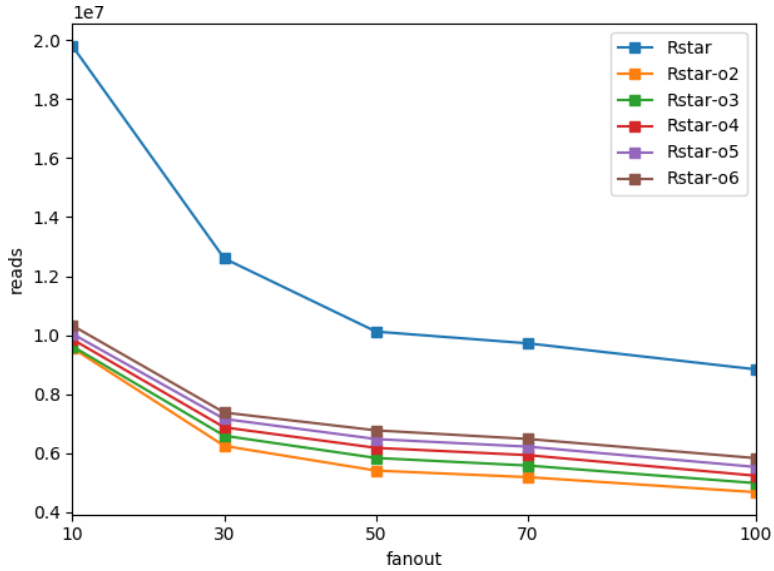

Figure 7.10: Reads used to create different structures from Bit02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X.

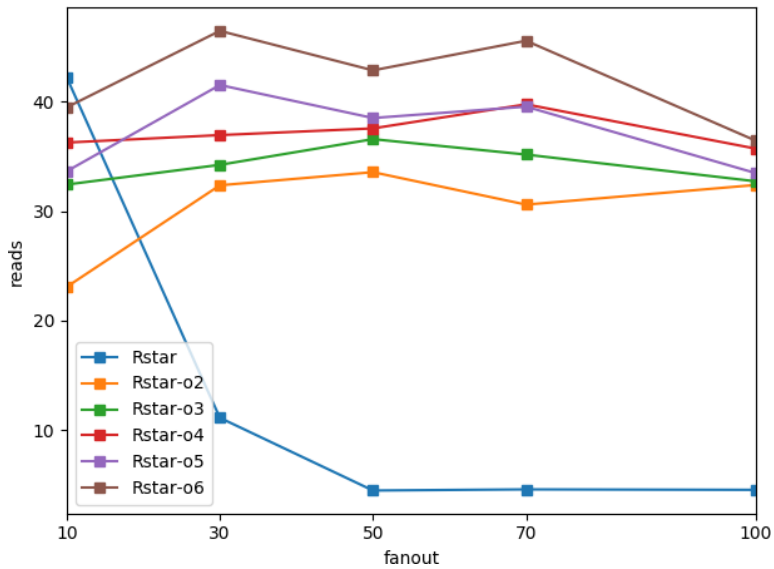

Figure 7.11: Reads per query in q0 from Bit02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X.

that for high fanout, the overflow versions has pretty similar performance. Reads between $40 - 50$ for $q2$ and $30 - 40$ for $q0$.

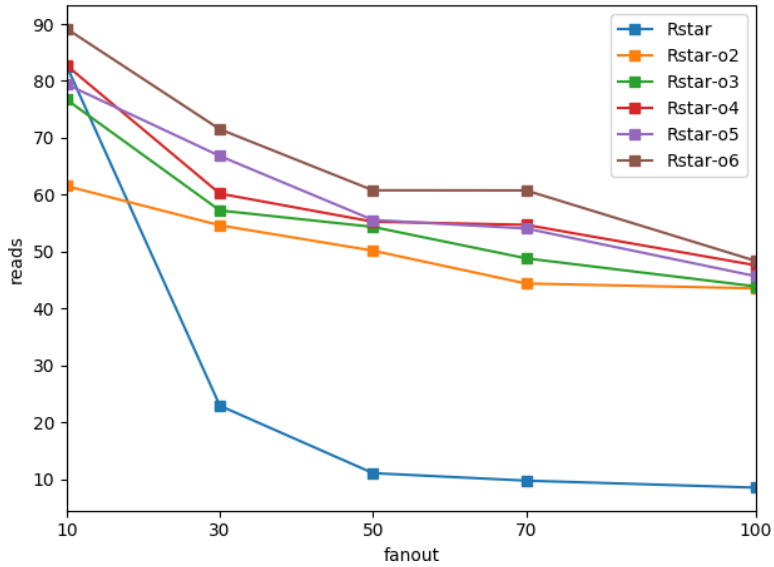

Figure 7.12: Reads per query in q2 from Bit02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X.

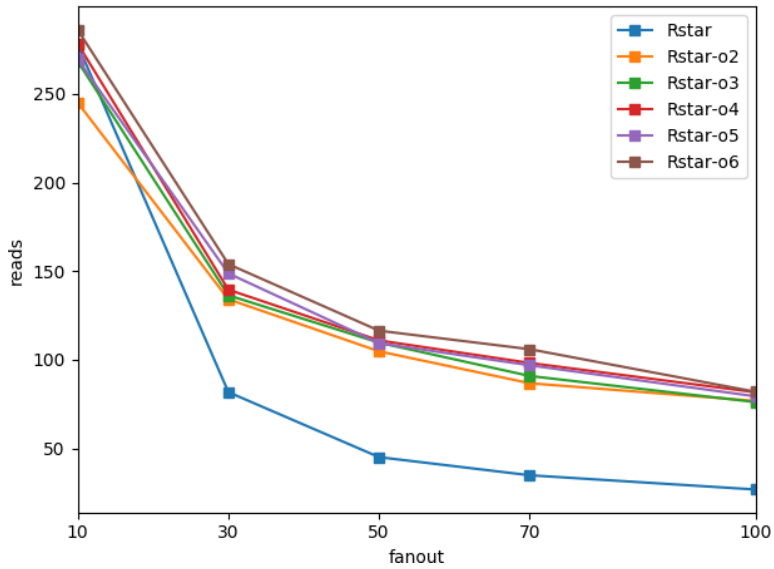

Figure 7.13: Reads per query in q3 from Bit02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X.

For the last query set, $q3$ shown in Figure 7.13, we see that the gap between standard R*-tree and the overflow versions are closing even more. This gap is still about 50 reads for

fanouts above 40, a little more than for $q2$. But the gap is relatively smaller compared to the improvement all the structures see from 10 to 30 fanout.

7.4 Changing fanout - Uni02

7.4.1 Linear R-tree

In this section we do the same as the last, but for the Uni02 distribution. This gives a clue about problems that might only occur in some distributions.

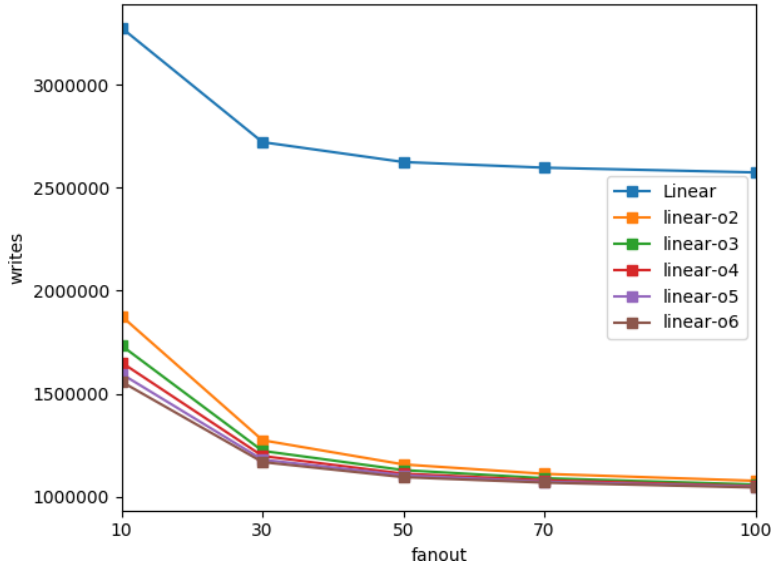

Figure 7.14: Writes used to create different structures from Uni02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X.

Figure 7.14 shows writes needed to build the structure. It is almost identical to the one for bit02, and there's hardly any difference. This also goes for the read count in Figure 7.15. This is probably because the same number of posts are added, which roughly produces the same number of splits. Also, a split causes less extra reads and writes in the linear tree than the R*-tree. This could explain why the figures are almost the same for the two distributions. Again it would highly benefit from a cache.

For the performance of $q0$, shown in Figure 7.16, one sees hints of linear being the best and linear-06 the worst. It also generally performs better than for the Bit02 distribution, but it also shows unexpected behaviour. The performance for $q0$ is again unexpected in that the performance improves until fan-out 50, and then gets worse. As we remember from Bit02, the performance had a spike for the linear version at fanout 70, but the all over

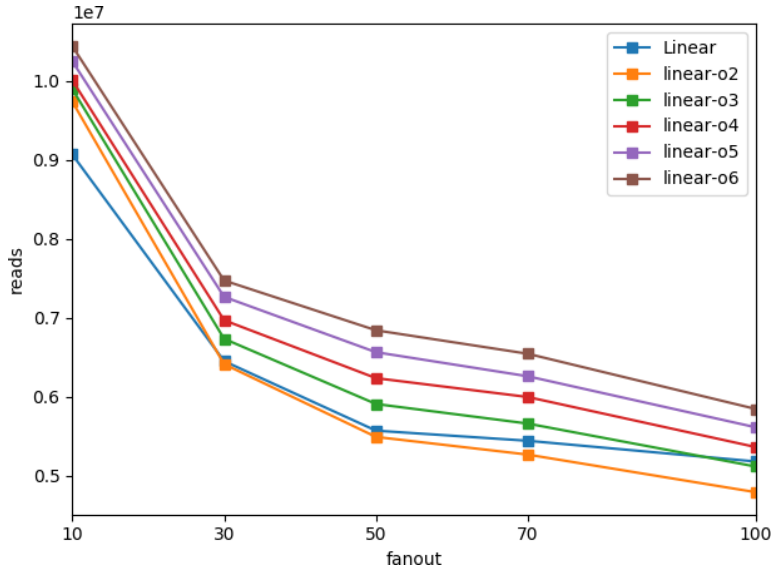

Figure 7.15: Reads used to create different structures from Uni02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X.

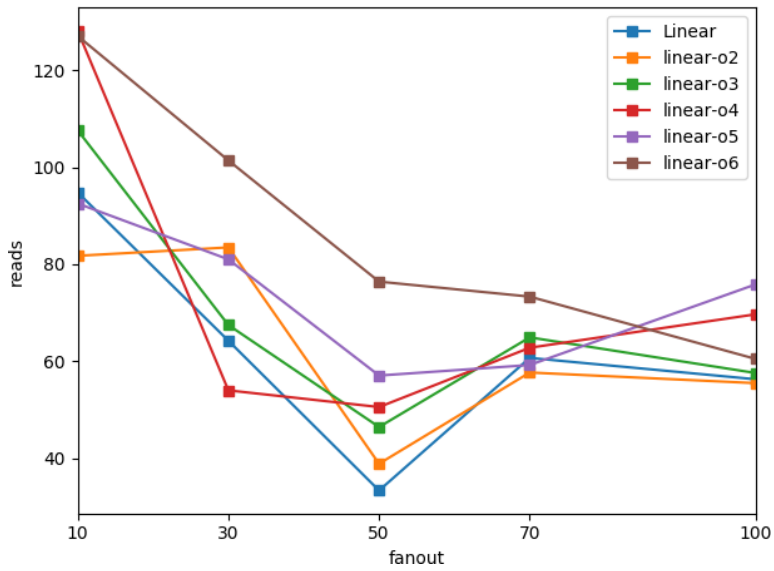

Figure 7.16: Reads per query in q0 from Uni02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X.

performance was increasing. It could have something to do with the nodes getting bigger and resulting in much overlap, and therefore also several paths one has to read.

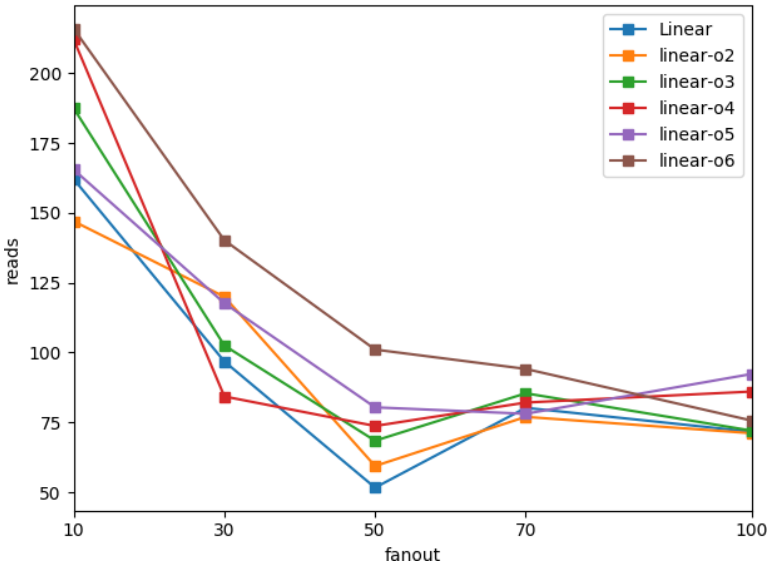

Figure 7.17: Reads per query in q2 from Uni02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X.

With $q2$ the trend from $q0$ continues. Only linear-o6 stands out as it has a performance increase throughout the graph. For $q3$ it seems more like it spikes at fanout 70, and then have a slight performance increase to 100 again.

7.4.2 R*-tree

Again, shown in Figure 7.19 and 7.20, one can hardly see any difference in construction cost between the Bit02 and the Uni02.

The query performance for the overflow R*-tree variants is interesting. The standard R*-tree does a great job for all the sets, and is clearly better than the overflow versions. This applies for $q0$, $q2$ and $q3$. For the overflow versions, they seem to have the same waypoint at fanout 50 as the linear versions also had for Uni02. For $q0$, in Figure 7.21, it seems like the overflow versions are varying around the same performance until fanout 50. Beyond that, they all have decreased performance.

For $q2$, in Figure 7.22, we see a clear improvement until fanout 50. But again the performance decrease afterwards. In $q3$, the structure has a performance gain from 10 to 50 but flattens out with only a slight performance gain until 100. Figure 7.23 shows this.

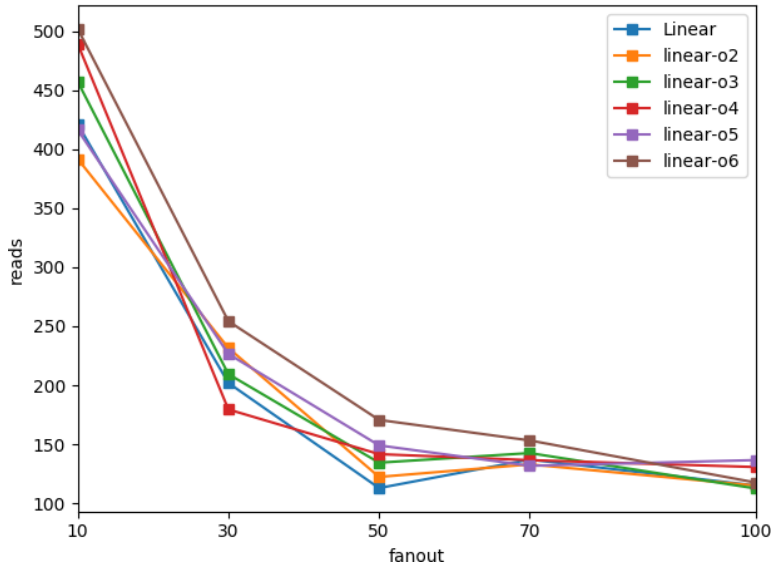

Figure 7.18: Reads per query in q3 from Uni02 distribution. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X.

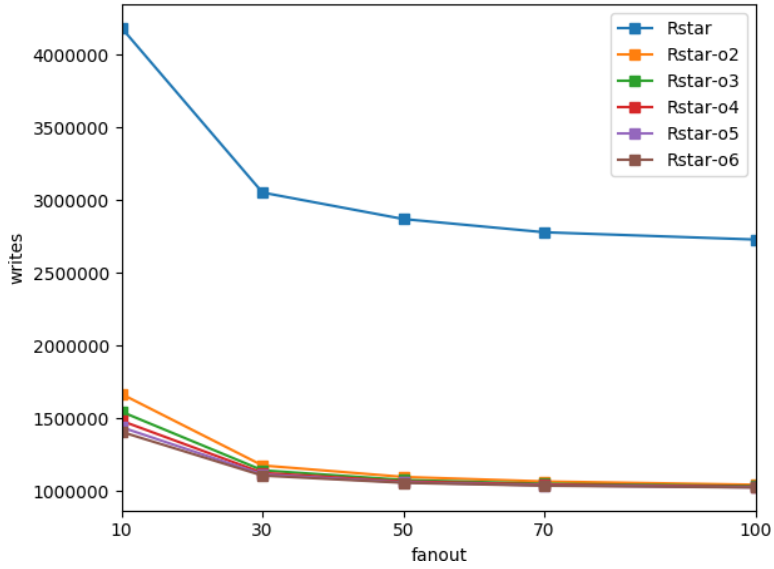

Figure 7.19: Writes used to create different structures from Uni02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X.

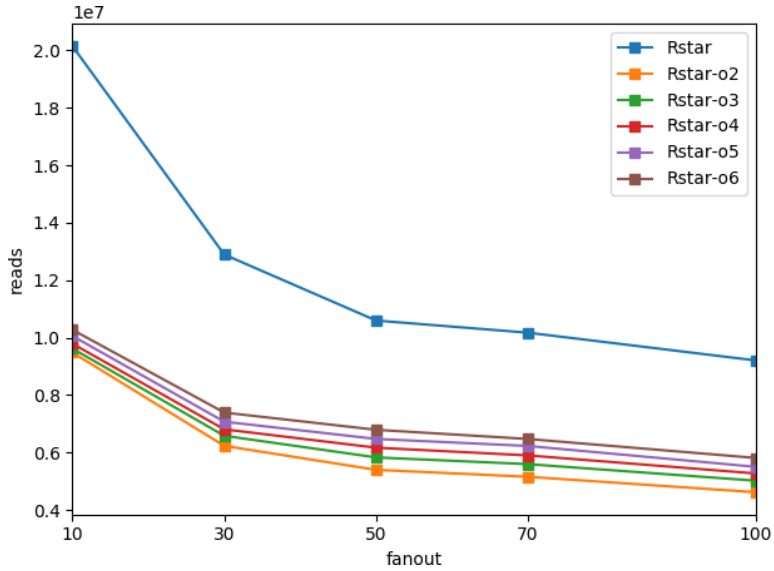

Figure 7.20: Reads used to create different structures from Uni02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X.

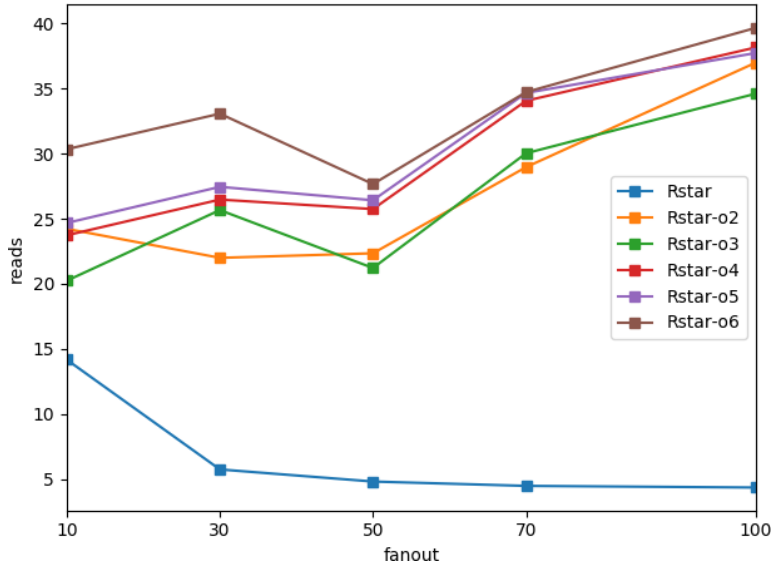

Figure 7.21: Reads per query in q0 from Uni02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X.

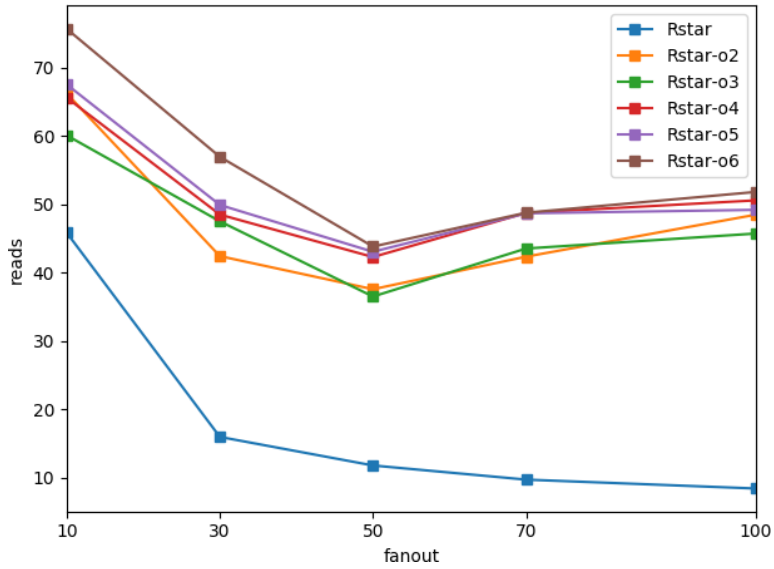

Figure 7.22: Reads per query in q2 from Uni02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X.

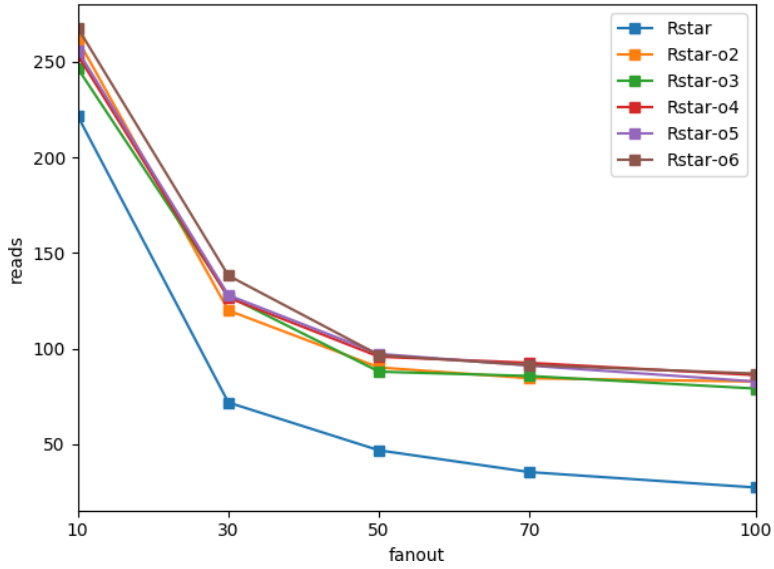

Figure 7.23: Reads per query in q3 from Uni02 distribution. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X.

7.5 Changing fanout with cache size 1000 - Uni02

As suggested in Figure 7.2, having a cache will dramatically reduce the reads needed. By the figure, 1000 seems to be a good choice as it's a reasonable size, and the improvement seems to decrease afterwards. Uni02 is used as there were the most unexpected results with that dataset.

7.5.1 Linear R-tree

The writes are of course the same as in Figure 7.14. The reads in Figure 7.24 has decreased a lot, to the point that they are all better than one without cache had at fanout 100. The trend is similar, but now the linear R-tree is noticeable better than the overflow versions. This makes sense because one has to read through the full overflow blocks before finding an available one to insert the post into.

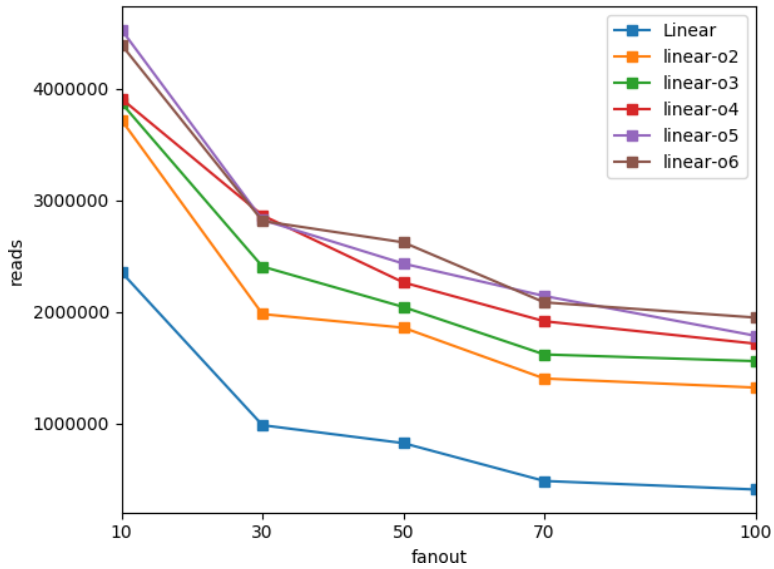

Figure 7.24: Reads used to create different structures from Uni02 distribution, cache size 1000. Linear is the linear R-tree. Linear-oX is the linear R-tree with max_overflow as X.

For the query sets, the read count is reduced by about 30-40 reads on average and the increase from fanout 50 to fanout 70 is less apparent.

7.5.2 R*-tree

Like the previous section, the number of writes is unchanged. The read count is another story. The noticeable thing here where cache size is 1000, in contrast to when it's 0, is that the R*-tree uses fewer reads than the overflow ones. Also, the read count is significantly reduced, for the R*-tree by up to 17.000.000 or down to 15 %. This is shown in Figure 7.25.

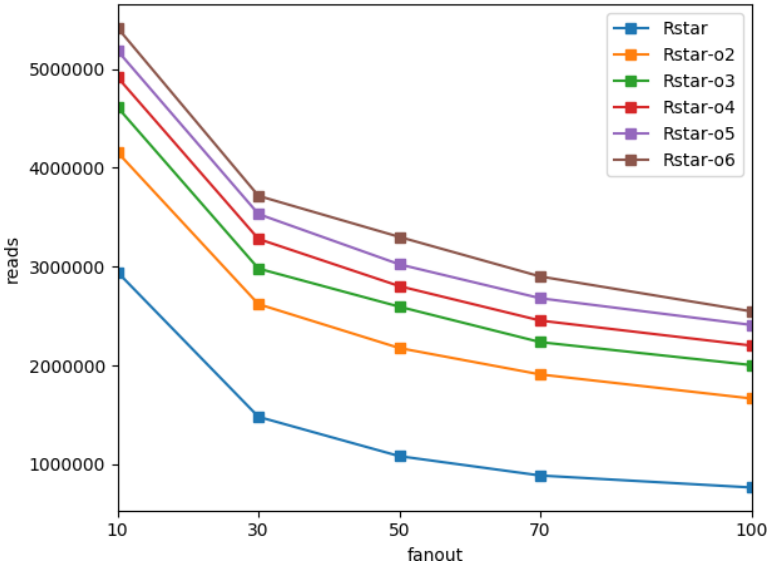

Figure 7.25: Reads used to create different structures from Uni02 distribution, cache size 1000. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X.

Figure 7.26 shows that the query performance for $q0$ has improved quite a bit with the use of a cache. Besides the lower read count, the performance doesn't decrease after fanout 50. It rather just keeps the same performance level, with a slight improvement as fanout increases. The same applies for $q2$. The last query set, $q3$, isn't that affected by the larger query size. The only difference is that the overflow versions perform better for high fanouts.

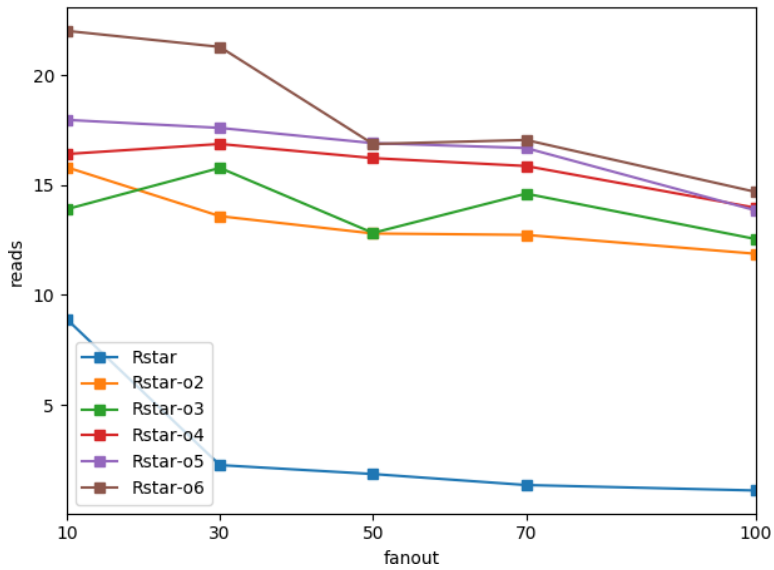

Figure 7.26: Reads pr query in q0 from Uni02 distribution, cache size 1000. Rstar is the R*-tree. Rstar-oX is the R*-tree with max_overflow as X.

Chapter 8

Discussion

From the results, it can be seen that the R*-tree and the linear R-tree presents quite different results. All overflow versions outperform the original ones in regard of writes to create the trees, for both distributions and regardless of cache size. The overflow versions also require far fewer reads to create the structure when the cache size is 0, but this flips when a cache i utilised. This flip is perhaps because that with overall less reads, the reads when inserting in an overflow node becomes noticeable. The cache also greatly reduces the overall reads for creating the tree.

When it comes to the queries, the non-overflow versions performs better. Especially is the R*-tree great and clearly better. The linear often has quite similar performance as its overflow versions, but tend to be in the lower level of plotted lines. Using a cache reduces their reads, especially for $q0$ and $q2$ where small query rectangles are requested. The largest queries, those in $q3$, don't benefit as much from the cache. All over we see that we greatly benefit from having a cache, even a moderate one for 1000 nodes.

Depending on the workload, the results would vary. Here the tree is first constructed and then queried. Other types of workload included deletions and updates, as well as queries and insertions. Arguably, the $q0$ performance from our results is most important for workload work deletions and insertions. Simply because updates and deletes require that one search and finds the post first, and then potentially trigger a restructure. Another argument to prioritise good query performance is that a tree usually handles more queries than insertions.

The types of trees tested represent two different *tiers* of R-tree, one good and one less god. The well-developed R*-tree would not benefit from implementing an overflow node. It would be an expensive trade between reads and writes.

For the linear R-tree, it is plausible. The non-overflow version tends to have better read count, but they are more grouped together compared the R*-tree versions. With a max_overflow of 2 and fanout of 50, the reads for queries are about the same, but one saves some writes. Unfortunately the non-overflow requires fewer reads when constructing the tree. Never the less, this is better than the OB+-tree that had 270 % more reads to save 17 % writes. This proves that overflow blocks alone work better for R-trees than for B+-tree, at least for some R-tree variants. In the end, one would have to look at the workload for the tree to see if it's beneficial.

8.1 Unexpected results

In this simulation, we don't take into account that all nodes might not fit into a block in storage. For example is typically one block in storage $4KB$, and if the fanout is large, one might need several such blocks to store a node. Therefore it is not realistic to increase fanout indefinitely, without increasing number of nodes accessed. In a real system, one would make sure to select a fanout that fits the filsystem's blocksize.

Because this simulation doesn't take storage blocks into account, it can be presumed that it would be beneficial to increase fanout as much as possible. A large fanout will increase the number of post or new nodes accessed from a node, and therefore decreasing the number of reads needed. The results on the other hand, shows that this isn't the case. Several of the figures show that the number of reads is reduced until fanout 50, and then increased after 50 until fanout reaches 70. This is not expected.

It is uncertain why this happens. One reason could be that with the nodes gets a larger MBR and overlap, which gives more paths to search. In the Revised R*-tree[3], they experiment and find that a fanout of about 100 is best for that tree. Although it's not the same structure as those in this paper, it's unexpected that the difference is that high.

8.2 Comparison to OR-tree

Because only the overflow leaf was made, the versions in this tree is pretty similar as the OR-tree[16]. Although they have a different focus, it's interesting to see if the findings are about the same. It is not possible to directly compare as they have a buffer to delay writes to a permanent storage, while this paper caches the latest lockups and writes to the permanent storage. Furthermore, the workloads and R-tree versions are different.

They don't explicitly say which R-tree they use as a base, but based on their *ChooseLeaf* algorithms it looks like like the one from Gutmann[10]. With their setup, they find that the OR-tree outperforms their standard R-tree by far when creating the tree. On the contrary, our results say that with a cache of 1000, the Linear R-tree uses fewer reads when con-

structing the tree, but that it's all over plausible for a linear R-tree. The R*-tree doesn't all over benefit from the overflow nodes in our paper, and it would be interesting to see them compare their OR-tree with such a structure.

Our tree and simulation don't show the same performance increasement. But they make it specialised against flash storage such as SSDs, and uses a buffer in their simulation. Perhaps our structures would have shown the same performance given a buffer instead.

Chapter 9

Conclusion

It is clear that the B^+-tree and the R-tree is similar in many ways, but different enough to make some concepts work on one and not the other. No equivalent for the Bloomtrees BF-node was found, as there was no suited filter to replace the bloomfilter and the bloomfilter wouldn't work on an R-tree. This is due to the differences in data stored and how they are searched, which effectively makes the boomfilter useless for an R-tree even tough it's highly effective for a B^+-tree.

Unfortunately, it's the BF-node that was the most interesting part about the Bloomtree. Without that part figured out for the R-tree, the concepts left from the Bloomtree to use was the overflow node. This is a known strategy, that has also been successfully applied to an R-tree before.

By implementing the overflow strategy for a couple of R-tree variants, and varying the amount of overflow blocks allowed, we confirmed that the utilising overflow blocks decreases number of writes needed. Not unexpected, it also increases the number of reads. This means one trades reads for writes, as it's often more suited for SSD characteristics. For the less advanced Linear R-tree, it seems like a plausible ting to do. And that just utilising overflow blocks works better than on the OB^+-tree. Although we preferably would like to be storage agnostic, it shows that overflow strategy could work.

- **RQ1:** Is it possible to apply the concepts of the Bloomtree to an R-tree?
- **RQ2:** Does this new solution have the same performance achievement as the Bloomtree?

To summarise in terms of the research questions. **RQ1** was partly successful, parts of the Bloomtree was applied to an R-tree, although it only was the overflow part. **RQ2** isn't possible to answer, without the BF-node the structures aren't quite comparable. However, the

linear version seemed to in some cases save both reads and writes by just using overflow. The decrease in reads was not was not at all indicated for R^*-trees with overflow.

Chapter 10

Further work

There is no doubt that good R-tree variants exist for various applications and storage mediums. But as technology moves along there are possibilities to make it even better. As there wasn't found an replacement for the Bloomtrees BF-node in this paper, it would be interesting to find a replacement and check if there's a significant performance improvement. There are many challenges here as highlighted in Chapter 5.

Based on this paper one could try to incorporate the overflow node in other R-tree variants, as well as test it further on more distributions and vary the workload. It's possible to experiment with the overflow node. For example with inspiration from the head-leaf in the Bloomtree, and try and freeze some blocks in order to save reads on insertions. Another part is to improve the transformation from overflow node to normal node, given of course that no replacement for the BF-node is found. For this part, inspiration could be taken from different R-trees that uses bulk insertion [5, 12, 6].

Another path that can be explored is to a larger degree defy the R-tree structure, in regards to min-fanout and height. One could, for example, add levels further than what the tree has elements for. This is sort of like starting a new R-tree at leaf level. Further one could try to allow splits to be divided into more than two partitions. This can be done trough k-means clustering or an arbitrary number through hierarchical clustering. The last thing is to let new elements get their own leafs if they otherwise would require too much enlargement on existing MBRs.

Additionally, one could try and utilise parallelism in new hardware. This is more like adding resources rather than improving the structure, but it should give improved results. This could enable even more advanced algorithms for insertions and splits, which again improves the structure. This would probably mainly improve runtime, which isnt the focus of this paper.

Bibliography

[1] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, May 1990.

[2] Norbert Beckmann and Bernhard Seeger. A benchmark for multidimensional index structures. `http://www.mathematik.uni-marburg.de/~seeger/rrstar/index.html`. [Online; accessed June-2017].

[3] Norbert Beckmann and Bernhard Seeger. A revised r*-tree in comparison with related index structures. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 799–812, New York, NY, USA, 2009. ACM.

[4] Juan Castaeda. Sequoia 2000. `http://meteora.ucsd.edu/s2k/s2k_home.html`. [Online; accessed 14-December-2016].

[5] Li Chen, Rupesh Choubey, and Elke A. Rundensteiner. Bulk-insertions into r-trees using the small-tree-large-tree approach. In *Proceedings of the 6th ACM International Symposium on Advances in Geographic Information Systems*, GIS '98, pages 161–162, New York, NY, USA, 1998. ACM.

[6] Rupesh Choubey, Li Chen, and Elke A. Rundensteiner. Gbi: A generalized r-tree bulk-insertion strategy. In *In Proceedings of International Symposium on Spatial Databases*, pages 91–108, 1999.

[7] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. Anti-caching: A new approach to database management system architecture. *Proceedings of the VLDB Endowment*, 6(14):1942–1953, 2013.

[8] R Elmasri and S Navathe. *Fundamentals of Database Systems, ed 7*. Pearson Education, 2015.

[9] Sanjay Ghemawat and Jeff Dean. Leveldb, a fast and lightweight key/value database library by google. `https://github.com/google/leveldb`. [Online; accessed 13-December-2016].

[10] Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[11] Peiquan Jin, Chengcheng Yang, Christian S Jensen, Puyuan Yang, and Lihua Yue. Read/write-optimized tree indexing for solid-state drives. *The VLDB Journal*, pages 1–23, 2015.

[12] Taewon Lee, Bongki Moon, and Sukho Lee. *Bulk Insertion for R-Tree by Seeded Clustering*, pages 129–138. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[13] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N Papadopoulos, and Yannis Theodoridis. *R-trees: Theory and Applications*. Springer Science & Business Media, 2010.

[14] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[15] Paolo Palmieri, Luca Calderoni, and Dario Maio. Spatial bloom filters: enabling privacy in location-aware applications. In *International Conference on Information Security and Cryptology*, pages 16–36. Springer, 2014.

[16] Na Wang, Peiquan Jin, Shouhong Wan, Yinghui Zhang, and Lihua Yue. *OR-Tree: An Optimized Spatial Tree Index for Flash-Memory Storage Systems*, pages 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

Appendix A

Appendix

Table A.1: Bit02 distribution, fanout 50, p_grade 30%, min_fill 40%.

cache size	type	reads	writes	q0	q2	q3
0	linear	5569859	2625244	56.26826	79.1493	146.841024
100	linear	1675831	2625244	46.37226	71.6431	146.156448
500	linear	1020208	2625244	33.19202	57.4635	128.362200
1000	linear	755872	2625244	27.26763	50.9939	119.802781
5000	linear	203981	2625244	13.02740	31.0321	87.276233
10000	linear	77311	2625244	7.32850	20.3362	62.943110
0	linear-o3	5909057	1125393	81.64936	115.5343	205.062895
100	linear-o3	3316349	1125393	71.24374	112.0136	204.931732
500	linear-o3	2374747	1125393	52.42113	87.8072	180.516751
1000	linear-o3	1825636	1125393	44.79918	78.9271	168.087547
5000	linear-o3	367031	1125393	23.64641	48.9442	117.213970
10000	linear-o3	89638	1125393	13.28587	30.2860	79.486726
0	Rstar	10116073	2750565	4.47062	11.0540	45.121997
100	Rstar	1855035	2750565	2.51984	9.5213	43.859039
500	Rstar	1365074	2750565	1.99600	8.6195	42.664981
1000	Rstar	1092867	2750565	1.68114	8.2110	41.516435
5000	Rstar	558736	2750565	0.96158	6.3816	34.804994
10000	Rstar	321507	2750565	0.75701	4.9919	27.716498
0	Rstar-o3	5827054	1078056	36.57871	54.3434	109.686157
100	Rstar-o3	3578425	1078056	29.60913	49.0000	107.729140
500	Rstar-o3	2917999	1078056	21.62898	40.3325	97.098925
1000	Rstar-o3	2501355	1078056	17.89286	35.8912	90.900126
5000	Rstar-o3	1254042	1078056	8.66609	21.9703	66.490202
10000	Rstar-o3	618212	1078056	4.76164	14.1448	46.441846

Table A.2: Bit02 distribution, p_grade 30%, min_fill 40%, cachesize 0.

fanout	type	reads	writes	q0	q2	q3
10	Linear	9058221	3254008	119.71418	196.4575	470.588180
30	Linear	6460917	2710951	74.48847	109.8951	218.920986
50	Linear	5569859	2625244	56.26826	79.1493	146.841024
70	Linear	5447202	2594201	75.16231	102.3342	173.204804
100	Linear	5169252	2574937	42.71482	57.8384	100.736410
10	Linear-o2	9764845	1860806	161.03629	254.3142	547.089128
30	Linear-o2	6406846	1267011	86.68284	124.0763	234.090708
50	Linear-o2	5477820	1153759	76.09833	103.8144	179.859671
70	Linear-o2	5247548	1106590	61.39588	84.5964	147.655815
100	Linear-o2	4733694	1071580	51.98674	68.7182	113.207016
10	Linear-o3	9881357	1708670	79.87055	145.4671	381.722819
30	Linear-o3	6743517	1218341	112.51546	155.1897	275.286979
50	Linear-o3	5909057	1125393	81.64936	115.5343	205.062895
70	Linear-o3	5641343	1087554	61.73784	85.4172	146.210809
100	Linear-o3	5083905	1060363	55.38132	70.0531	110.717446
10	Linear-o4	9984747	1628232	129.93709	212.8188	478.647282
30	Linear-o4	6984242	1191536	123.82194	177.5075	316.802781
50	Linear-o4	6238002	1107626	66.30311	91.3740	163.447219
70	Linear-o4	5984030	1077287	63.64435	85.1466	142.713654
100	Linear-o4	5322936	1051306	56.95458	73.4091	117.412137
10	Linear-o5	10215342	1581930	143.55743	242.3223	541.724083
30	Linear-o5	7236764	1171662	82.29556	122.6947	234.703540
50	Linear-o5	6545941	1099269	73.56321	97.8469	168.211125
70	Linear-o5	6272716	1069153	74.82256	97.6633	157.931416
100	Linear-o5	5614720	1046993	56.71347	72.2457	113.870417
10	Linear-o6	10464908	1537358	138.24662	237.7460	534.501264
30	Linear-o6	7455782	1157045	105.27937	144.7285	257.209861
50	Linear-o6	6831889	1094601	89.37022	120.2977	205.510746
70	Linear-o6	6579708	1062519	77.24877	98.9290	159.611568
100	Linear-o6	5856562	1041762	64.53639	85.4897	139.336283

Table A.3: Bit02 distribution, p_grade 30%, min_fill 40%, cachesize 0.

fanout	type	reads	writes	q0	q2	q3
10	Rstar	19807416	4186447	42.21144	82.5903	277.697851
30	Rstar	12599282	3017522	11.11538	22.9060	81.785082
50	Rstar	10116073	2750565	4.47062	11.0540	45.121997
70	Rstar	9718206	2664771	4.57287	9.7390	34.902655
100	Rstar	8834829	2592094	4.53152	8.5046	26.862832
10	Rstar-o2	9559546	1657153	23.08091	61.5187	245.060683
30	Rstar-o2	6237711	1178833	32.36142	54.6197	134.161820
50	Rstar-o2	5399330	1098588	33.56215	50.1702	104.801201
70	Rstar-o2	5177115	1068201	30.59455	44.3923	86.811315
100	Rstar-o2	4665981	1044586	32.39007	43.5375	76.814159
10	Rstar-o3	9617100	1540841	32.42972	76.7028	268.352718
30	Rstar-o3	6583101	1144755	34.22906	57.2301	136.405499
50	Rstar-o3	5827054	1078056	36.57871	54.3434	109.686157
70	Rstar-o3	5571218	1052870	35.16848	48.8205	90.896650
100	Rstar-o3	4974699	1034913	32.72449	43.8847	76.155815
10	Rstar-o4	9854350	1479295	36.26631	82.7097	278.311315
30	Rstar-o4	6867610	1126454	36.94776	60.1964	139.584387
50	Rstar-o4	6170000	1066725	37.55931	55.2580	110.964918
70	Rstar-o4	5924434	1045910	39.76116	54.7054	98.242099
100	Rstar-o4	5225833	1030290	35.71028	47.5940	81.712073
10	Rstar-o5	10041506	1437156	33.64381	79.4025	269.488622
30	Rstar-o5	7153616	1115247	41.52387	66.8034	148.763590
50	Rstar-o5	6469111	1061043	38.50851	55.5593	109.402023
70	Rstar-o5	6212572	1041938	39.54909	54.0668	96.940265
100	Rstar-o5	5523205	1027442	33.47554	45.6820	79.411188
10	Rstar-o6	10321594	1406416	39.48453	89.1686	286.137800
30	Rstar-o6	7370285	1107203	46.47203	71.5068	153.992099
50	Rstar-o6	6764149	1057801	42.86227	60.7798	116.438053
70	Rstar-o6	6473314	1039054	45.56908	60.7618	106.004741
100	Rstar-o6	5822793	1025430	36.43809	48.3650	81.966182

Table A.4: Uni02 distribution, fanout 50, p_grade 30%, min_fill 40%.

cache size	type	reads	writes	q0	q2	q3
0	linear	5567629	2625124	33.32062	51.6585	112.801201
100	linear	1713579	2625124	28.47964	47.9682	111.702276
500	linear	1092693	2625124	21.40804	40.7989	103.449115
1000	linear	822435	2625124	17.66346	36.5491	97.696587
5000	linear	249297	2625124	8.75367	23.6015	73.836915
10000	linear	94364	2625124	5.19335	16.2683	55.564791
0	linear-o3	5905723	1128400	46.42080	68.4458	134.314791
100	linear-o3	3401815	1128400	41.41405	64.5387	133.767067
500	linear-o3	2548261	1128400	33.09963	55.9720	123.385588
1000	linear-o3	2041788	1128400	28.59644	50.8372	116.520228
5000	linear-o3	524394	1128400	15.51414	33.0710	86.136852
10000	linear-o3	110970	1128400	9.03077	21.7644	61.634640
0	Rstar	10597720	2869685	4.80199	11.8211	46.864728
100	Rstar	1849782	2869685	2.78740	10.2455	45.519595
500	Rstar	1357947	2869685	2.19529	9.2594	44.214918
1000	Rstar	1081686	2869685	1.84892	8.8267	43.055626
5000	Rstar	541874	2869685	1.05472	6.7903	36.044248
10000	Rstar	302529	2869685	0.81263	5.2359	28.302781
0	Rstar-o3	5833230	1077685	21.21346	36.5103	87.991466
100	Rstar-o3	3607128	1077685	18.67578	34.4471	86.609987
500	Rstar-o3	2993719	1077685	15.05041	30.9178	82.927939
1000	Rstar-o3	2594242	1077685	12.82499	28.3221	79.172882
5000	Rstar-o3	1358494	1077685	6.65358	18.9719	60.726296
10000	Rstar-o3	695057	1077685	3.82365	12.8729	44.563527

Table A.5: Uni02 distribution, p_grade 30%, min_fill 40%, cachesize 0.

fanout	type	reads	writes	q0	q2	q3
10	linear	9069467	3277016	94.80629	162.0423	421.250948
30	linear	6448826	2721875	64.21984	96.8134	201.900442
50	linear	5567629	2625124	33.32062	51.6585	112.801201
70	linear	5439213	2597714	60.67503	80.2289	136.981037
100	linear	5176336	2574645	56.26316	71.6406	114.986726
10	linear-o2	9741011	1876363	81.73628	146.9883	391.851138
30	linear-o2	6410609	1273691	83.46712	119.9680	231.536030
50	linear-o2	5488075	1155967	38.83874	59.4032	122.242099
70	linear-o2	5263778	1110310	57.66208	76.9900	132.873894
100	linear-o2	4787005	1076378	55.46535	71.1288	114.888748
10	linear-o3	9900381	1733343	107.61058	187.3924	457.333123
30	linear-o3	6732237	1221917	67.45773	102.5159	209.404867
50	linear-o3	5905723	1128400	46.42080	68.4458	134.314791
70	linear-o3	5657149	1089139	64.90886	85.4205	142.480405
100	linear-o3	5112249	1058338	57.55166	72.0528	112.420354
10	linear-o4	10010962	1651104	128.13627	212.0036	489.326485
30	linear-o4	6967829	1197044	53.98089	84.2970	179.453224
50	linear-o4	6233362	1111199	50.53421	73.7319	141.563527
70	linear-o4	5993841	1079243	62.77230	82.1085	136.479772
100	linear-o4	5359247	1050211	69.67185	86.0559	130.615676
10	linear-o5	10252164	1595636	92.52488	165.4559	416.383059
30	linear-o5	7263952	1178967	81.00162	117.6659	226.529393
50	linear-o5	6562375	1103347	57.03313	80.3948	148.979140
70	linear-o5	6256803	1072085	59.21163	78.0904	131.667509
100	linear-o5	5609956	1047995	75.84640	92.3425	136.517383
10	linear-o6	10444100	1557824	127.07788	215.8604	502.136220
30	linear-o6	7471471	1167466	101.40986	140.1994	254.298040
50	linear-o6	6840199	1094162	76.40820	101.0748	170.578382
70	linear-o6	6542297	1067647	73.31299	94.1469	153.195006
100	linear-o6	5842202	1043884	60.45503	75.6635	117.478192

Table A.6: Uni02 distribution, p_grade 30%, min_fill 40%, cachesize 0.

fanout	type	reads	writes	q0	q2	q3
10	Rstar	20149782	4182491	14.16658	45.8342	221.705120
30	Rstar	12893930	3051549	5.73245	15.9959	71.880215
50	Rstar	10597720	2869685	4.80199	11.8211	46.864728
70	Rstar	10174206	2778737	4.46807	9.7527	35.479772
100	Rstar	9208529	2728672	4.34282	8.4530	27.424463
10	Rstar-o2	9485711	1667828	24.22458	66.1803	261.706068
30	Rstar-o2	6236219	1177205	21.99437	42.4173	120.006637
50	Rstar-o2	5400807	1098599	22.34350	37.5805	90.202276
70	Rstar-o2	5160115	1067465	28.96481	42.3382	84.530025
100	Rstar-o2	4622502	1044871	36.99519	48.4766	82.834703
10	Rstar-o3	9623188	1546149	20.23576	60.0678	246.613780
30	Rstar-o3	6586706	1144011	25.66453	47.5054	127.472819
50	Rstar-o3	5833230	1077685	21.21346	36.5103	87.991466
70	Rstar-o3	5599838	1051825	30.02909	43.5289	85.740834
100	Rstar-o3	5023820	1035338	34.61061	45.7189	79.153919
10	Rstar-o4	9794191	1483744	23.72543	65.5773	253.547408
30	Rstar-o4	6811118	1126260	26.47182	48.4907	126.693742
50	Rstar-o4	6171867	1067860	25.74115	42.2845	95.816372
70	Rstar-o4	5903839	1046519	34.06239	48.7367	92.603034
100	Rstar-o4	5274501	1031352	38.16421	50.5683	86.174779
10	Rstar-o5	10061301	1439602	24.67238	67.5022	255.804046
30	Rstar-o5	7076391	1115682	27.44996	49.8969	127.986410
50	Rstar-o5	6477036	1061124	26.41920	43.0549	97.300885
70	Rstar-o5	6232179	1041055	34.66613	48.6770	91.165297
100	Rstar-o5	5505374	1027321	37.73480	49.1837	82.841972
10	Rstar-o6	10275960	1407892	30.34049	75.6355	267.775601
30	Rstar-o6	7394619	1107797	33.07367	56.9320	138.225348
50	Rstar-o6	6794787	1058198	27.66279	43.8124	96.646650
70	Rstar-o6	6475515	1038313	34.73121	48.7595	91.424779
100	Rstar-o6	5815017	1025874	39.66822	51.8009	87.002212

Table A.7: Uni02 distribution, p_grade 30%, min_fill 40%, cachesize 1000.

fanout	type	reads	writes	q0	q2	q3
10	linear	2352626	3277016	61.80118	134.1018	401.297408
30	linear	983235	2721875	35.33720	69.7154	178.301833
50	linear	822435	2625124	17.66346	36.5491	97.696587
70	linear	483514	2597714	27.34881	47.0475	103.661188
100	linear	406998	2574645	19.61997	34.6277	76.565424
10	linear-o2	3714898	1876363	53.58556	123.4500	374.884640
30	linear-o2	1978326	1273691	47.90072	86.0013	200.717762
50	linear-o2	1856598	1155967	22.28456	43.0949	105.858407
70	linear-o2	1400976	1110310	27.49231	46.9261	102.094185
100	linear-o2	1320014	1076378	20.24887	35.4606	76.872946
10	linear-o3	3867434	1733343	74.07750	158.8701	436.232301
30	linear-o3	2404041	1221917	40.97399	77.3137	186.065740
50	linear-o3	2041788	1128400	28.59644	50.8372	116.520228
70	linear-o3	1616805	1089139	32.53368	53.0234	109.211441
100	linear-o3	1556880	1058338	22.58639	36.4424	75.490202
10	linear-o4	3906613	1651104	95.06424	183.7231	468.229140
30	linear-o4	2862677	1197044	33.98237	65.0779	161.554678
50	linear-o4	2262031	1111199	31.25243	54.5518	122.132743
70	linear-o4	1914163	1079243	31.18691	50.4684	104.134640
100	linear-o4	1713137	1050211	25.27841	40.8677	82.921934
10	linear-o5	4524282	1595636	70.19999	147.0206	402.436157
30	linear-o5	2822455	1178967	52.87477	90.5635	201.116625
50	linear-o5	2430417	1103347	35.45598	58.9407	127.594817
70	linear-o5	2140275	1072085	29.76042	48.5811	101.056574
100	linear-o5	1783920	1047995	28.47678	43.9321	85.628635
10	linear-o6	4389376	1557824	97.53142	190.9307	482.925411
30	linear-o6	2812068	1167466	66.64524	106.7407	223.467446
50	linear-o6	2620836	1094162	42.29066	67.2091	137.548989
70	linear-o6	2083465	1067647	38.83697	59.9610	118.359987
100	linear-o6	1946987	1043884	25.88781	40.3385	79.693110

Table A.8: Uni02 distribution, p_grade 30%, min_fill 40%, cachesize 1000.

fanout	blocks	reads	writes	q0	q2	q3
10	Rstar	2943592	4182491	8.89929	41.6935	217.877686
30	Rstar	1479612	3051549	2.25572	12.6663	67.961441
50	Rstar	1081686	2869685	1.84892	8.8267	43.055626
70	Rstar	885719	2778737	1.34795	6.6459	31.287611
100	Rstar	764251	2728672	1.09844	5.0449	22.902971
10	Rstar-o2	4156785	1667828	15.81597	59.5731	256.143805
30	Rstar-o2	2620078	1177205	13.58819	34.6105	112.001580
50	Rstar-o2	2175828	1098599	12.79928	28.2758	80.326802
70	Rstar-o2	1908715	1067465	12.73541	25.9931	67.409292
100	Rstar-o2	1664865	1044871	11.87691	22.6489	55.188685
10	Rstar-o3	4612693	1546149	13.90535	55.0545	241.976296
30	Rstar-o3	2980154	1144011	15.78626	38.4503	118.327750
50	Rstar-o3	2594242	1077685	12.82499	28.3221	79.172882
70	Rstar-o3	2235614	1051825	14.61380	27.9332	69.310051
100	Rstar-o3	2003128	1035338	12.54585	22.9132	54.759166
10	Rstar-o4	4916454	1483744	16.41732	59.8344	248.421618
30	Rstar-o4	3280079	1126260	16.87487	39.5525	117.703856
50	Rstar-o4	2802611	1067860	16.23155	32.8525	85.909608
70	Rstar-o4	2453962	1046519	15.87195	30.2775	73.105247
100	Rstar-o4	2200734	1031352	13.96745	25.8016	59.526865
10	Rstar-o5	5185281	1439602	17.97141	62.0853	250.713338
30	Rstar-o5	3532095	1115682	17.60663	40.6034	118.684260
50	Rstar-o5	3022948	1061124	16.91273	33.5543	86.934260
70	Rstar-o5	2679897	1041055	16.68984	30.4561	71.783502
100	Rstar-o5	2409379	1027321	13.85018	24.7118	56.565107
10	Rstar-o6	5416855	1407892	22.02417	68.8334	261.889381
30	Rstar-o6	3716035	1107797	21.29285	45.8050	127.209861
50	Rstar-o6	3300552	1058198	16.87536	33.0274	85.361568
70	Rstar-o6	2900158	1038313	17.05607	30.9403	72.691846
100	Rstar-o6	2546402	1025874	14.69152	26.1860	59.171302