Norwegian University of
Science and Technology

Efficient Spatial Search using Memory Resident R-trees

Truls Rustad Fossum

Master of Science in Computer Science
Submission date: June 2017
Supervisor: Kjetil Nørvåg, IDI
Co-supervisor: Sean Chester, IDI

Norwegian University of Science and Technology
Department of Computer Science

Problem Description

With the trend of large main memories capable of holding entire databases, we aim to investigate how existing techniques for spatial search can be tailored for memory resident databases. The task is to study existing techniques for spatial search, suggest improvements to these techniques for memory resident databases and provide an experimental evaluation thereof.

Supervisor: Kjetil Nørvåg
Co-supervisor: Sean Chester

Abstract

This thesis investigates the performance of memory resident spatial search, focusing on the R-tree. The characteristics of modern computer architectures are first visited to understand how they have changed since the invention of the R-tree, and what difference moving the database from disk to memory can be expected to make. The design of four well known R-trees are introduced, implemented and used to reproduce the results of Beckmann and Seeger.

Next, four optimizations for the R-tree search are suggested in an attempt to speed up search by improving the memory layout, through explicit parallelization using SIMD instruction, and by applying pruning at a lower level than classical search in R-tree does. The results show that the optimizations often have the intended effects and yield speedups in excess of 1.3 for all tested data sets, and almost reach 1.8 in specific cases.

More importantly, it is found during the analysis that there exists a tradeoff between sequential memory access, with an associated computational cost, and random memory access arising due to the tree structure. Improving one aspect alone may offsets this balance and improve performance, but the true challenge is to create the combined approach needed to fully exploit modern computer architectures during search in memory resident R-trees.

Sammendrag

Denne oppgaven utforsker ytelsen til søk i romlig data hvor søkestrukturen befinner seg i datamaskinens arbeidsminne. Først undersøkes kjennetegnene ved dagens datamaskinarkitekturer for å forstå hvordan de har endret seg siden R-treets oppfinnelse, og for å kunne forstå hvordan skiftet fra disk til minne påvirker ytelsen. Deretter introduseres fire eksisterende algoritmer for kostruksjon av R-trær, før de implementeres og brukes til å reprodusere resultatene til Beckmann og Seeger.

Til slutt foreslås fire forbedringer for søk i R-trær i et forsøk på å gjøre søket raskere ved å forbedre minnestrukturen, eksplisitt parallelisere søket med SIMD instruksjoner og ved å sortere ut hvilke data som er nyttige å søke igjennom på et lavere nivå enn det klassiske R-treet. Resultatene viser at forbedringene har de tilsiktede effektene og at de gir en speedup i overkant av 1.3 for samtlige dataset de testes på. I noen tilfeller oppnås en speedup på nesten 1.8.

I løpet av analysen konstateres det også at det finnes en balanse mellom sekvensiell minneaksess, ledsaget av en kostnad for beregning, og tilfeldig minneaksess som følge av trestrukturen. Å forbedre ytelsen til en av disse kan forskyve balansen og forbedre ytelsen, men utfordringen ligger i å finne en metode som forbedrer ytelsen på begge områdene slik at maskinvaren utnyttes fullt ut.

Acknowledgments

I wish to thank my supervisor Kjetil Nørvåg and co-supervisor Sean Chester for good advice, feedback and inspiring ideas. I would also like to thank Norbert Beckmann and Bernhard Seeger for providing data and query sets.

Contents

1 Introduction **1**
1.1 Related Work 2

2 Modern Computer Architectures **5**
2.1 Memory Access 6
2.2 Parallelism 8
2.3 Speculation 11
2.4 Micro Operations 12

3 Spatial Indexing **15**
3.1 Concepts 15
3.2 Queries 17
3.3 R-trees 18

4 R-tree Variants **25**
4.1 Prerequisites 25
4.2 Quadratic R-tree 26
4.3 R*-tree 29
4.4 Revised R*-tree 32
4.5 Hilbert R-tree 37

5 Optimizations **45**
5.1 Arrays of Fields Layout 45
5.2 SIMD Node Scans 47
5.3 Full Node Scans 48
5.4 Pruning Node Scans 49

6 Methodology **53**
6.1 Data and Query Sets 53
6.2 Run Time Measurement 54
6.3 Hardware Counters and Instrumentation 56
6.4 Implementation 57
6.5 Environment 57
6.6 Parameter Optimization 58

7 Experimental Results **61**
7.1 Reproduction of Results 61
7.2 Run Time Probability Distribution 63
7.3 Impact of Parameters 67
7.4 Behavior of the Naive Implementation 70
7.5 Results of Optimizations 71

8 Discussion **81**
8.1 Memory Behavior 81
8.2 Parallelism 83
8.3 Research Questions 85

9 Conclusion **87**
9.1 Future Work 88

A Implementation **89**
A.1 Framework 89
A.2 Indexes 90

B Probability Based R-trees **93**
B.1 Query Intersection Probability 93
B.2 Example Distributions 94
B.3 Expectancies 97
B.4 Implementation 98

List of Figures

2.1 Von Neumann Architecture 5
2.2 Example memory hierarchy 7
2.3 Example NUMA architecture 9
2.4 Strided memory access 12

3.1 Concepts of spatial indexing 16
3.2 An axis aligned box 17
3.3 Example range query 18
3.4 Example R-tree in 2 dimensions 19
3.5 Example R-tree data layout 20
3.6 Untraditional R-tree 22

4.1 Hilbert curves 37

5.1 Naive node layout 46
5.2 Arrays of fields layout 47
5.3 SIMD blocking 48
5.4 Full node scan layout 48
5.5 Query and MBB overlap types 50

7.1 Run time histogram 63
7.2 Run time plots 66
7.3 Run time as function of m and M 67
7.4 Run time as function of m and M for large values 68
7.5 Total run time and fill grade for fixed m 69
7.6 Naive revised R*-tree statistics 71
7.7 Histogram of instruction retirement for the naive implementation 72

7.8 Speedup for arrays of fields layout 73
7.9 Speedup using SIMD scans 74
7.10 Histogram of instruction retirement for SIMD scans 75
7.11 Speedup using full node scans 76
7.12 Histogram of instruction retirement for full node scans 77
7.13 Speedup using pruning scans 78

B.1 Decision tree for branch-and-bound 100

List of Algorithms

3.1 R-tree range search 21
3.2 Generic R-tree insert 23

4.1 Quadratic R-tree `ChooseSubtree` 27
4.2 Quadratic R-tree `SplitNode` 28
4.3 R*-tree `ChooseSubtree` 30
4.4 R*-tree `SplitNode` 31
4.5 Revised R*-tree `ChooseSubtree` 33
4.6 Revised R*-tree `SplitNode` 34
4.7 Hilbert Encoding 39
4.8 Hilbert R-tree `ChooseSubtree` 41
4.9 Hilbert R-tree `SplitNode` 42
4.10 Hilbert R-tree `Share` 43

6.1 Evaluation process 55
6.2 Simulated annealing 59

Chapter 1

Introduction

More and more data is being generated in the world, but for this data to be useful, it must be possible to extract the relevant parts in an efficient manner. This is often accomplished through an index, serving as a set of shortcuts for accessing the data over which it spans.

Depending on the application, the most appropriate indexing method may vary. For simple numeric values, a sorted list and a binary search can suffice, or using a B*-tree [7] may solve the problem. Much of the data does however present itself as points in 2 or 3 dimensional space. Several indexing techniques for this kind of data exists [5, 6, 10, 28], one of which is the R-tree, introduced by Guttman in 1984 [14].

Many changes have occurred in the computer hardware industry since then. The DRAM capacity been roughly doubled every 2–3 years [16] and parallelism has become the default for performance sensitive application after the breakdown of Dennard scaling [8]. This radically changes the task of rapidly accessing large amounts of data.

First of all, the increased main memory storage means entire databases can be stored in memory, something which has already been explored and applied in commercial database systems for relational databases [32, 33]. This also allows better utilization of the available parallelism in modern processors as the latency of disk access is no longer the bottleneck.

When accessing databases residing on disk, the top priority is to reduce the number of disk accesses. As shown by Beckmann and Seeger [4], this even holds true for the more computationally complex insert operation. As long as a small percentage of the nodes are disk resident, the time required for computation drowns in the latency of loading nodes from disk.

The assumption that only disk accesses needs to be considered leads to the conclusion that reducing the computations required for search is useless. This may not be the case for memory resident indexes where both memory access and computation contribute to the total execution time.

This thesis examines search in memory resident R-trees, proposes some possible improvements for a memory resident R-tree with the goal of improving its single core performance. The improvements are then tested and evaluated.

These improvements do indeed increase the performance of the memory resident R-tree by exploiting more of the instruction and data level parallelism available on a single processor core and reducing the number of cache misses.

More precisely, the following research questions are examined.

- What aspects of R-tree search are not suited for modern processors?
- How can range search in R-trees be optimized such that more of the available data and instruction level parallelism can be utilized?
- What are the main challenges in doing so?

This thesis starts off with an introduction to modern computer architectures in Chapter 2, before the concepts of spatial indexing is introduced in Chapter 3· A description of different construction techniques follows in Chapter 4 before possible optimizations for search in memory resident R-trees are presented in Chapter 5. Finally, the methodology used during evaluation is presented in Chapter 6 and the results thereof in Chapter 7. These are then discussed in Chapter 8 before a short conclusion follows in Chapter 9.

1.1 Related Work

Many have tried to improve the performance of R-trees, and most of these improvements focus on the algorithm used when constructing the tree structure, such as the Greene's R-tree, Hilbert R-tree, R*-tree and revised R*-tree [2, 4, 13, 24]. The idea is to create a tree structure that is better suited for search.

Greene's R-tree, often referred to as Greene's split as the only change from the algorithm introduced by Guttman is an improved node split al-

gorithm, tries to improve the performance by finding a dimension to split along and sort the children in the found dimension.

Furthermore, the Hilbert R-tree uses a space filling curve to map the multidimensional data to single dimension. This has the advantage that the nodes can be sorted such that full nodes can share some of their children with their neighbors instead of being split, similar to what is done in the B*-tree. This provides good performance during inserts and in some cases a better structure for the tree.

The R*-tree introduces two major improvements. First, it tries to reinsert some of the entries in an overfull node instead of splitting it. This allows restructuring the tree after construction, often improving performance. Second, the split algorithm also falls back to perimeter when nodes cannot be distinguished based on volume. In addition, the overlap caused by a node insert is used to determine where a new item should be inserted.

The revised R*-tree improves on the R*-tree by increasing the number of splits considered for internal nodes and introducing a more sophisticated algorithm for selecting where a new item should be inserted. Furthermore, the reinsertion is replaced by a weighting function that depends on the previously inserted elements.

Although all of the above trees improve on the performance of the R-tree for disk, these improvements are also mostly valid for memory resident R-trees. They do however miss a lot of cases where further improvements could have been made because they assume a disk based index for which lowering the number of disk accesses is the most important optimization. For memory resident indexes, this is not as crystal clear.

The CR-tree [25], on the other hand, focus on the memory performance of the R-tree and employs ideas from memory resident relational databases. Using quantization and relative coordinates, the memory footprint of the CR-tree is reduced by 60 % compared to a regular R-tree with improved search efficiency as a consequence.

Even though the reduced size of the CR-tree increases cache utilization and reduces the memory bandwidth requirement, this is only a good step in the right direction. Several other techniques goes untried and untested, such as searching using SIMD instruction and optimizing the search algorithm itself for the memory hierarchy found in modern computers.

Scans using SIMD instructions has on the other hand been implemented for table scans in memory resident databases [38], and other approaches using bit parallel methods have been proposed [12, 27]. Even though such

approaches may have potential when combined with R-trees, this has, to the author's knowledge, not been tried.

Speeding up one dimensional tree search using SIMD instruction has however already been explored by Steffen et.al. [40]. Using a k-ary search instead of the more classical binary search or node scan, they manage to use vector instructions to parallelize the comparisons performed during the search, and thereby speeding up the search.

The k-ary search with SIMD instructions does however only consider the case where each key fits into a single SIMD lane. This is unfortunately not the case for the bounding boxes used in the R-tree, and thus there is no obvious way to implement k-ary search in an R-tree for increased performance.

Other efforts to improve the spatial search performance in main memory include replacing the entire index with a highly optimized linear scan through the data objects, as done for the VA-file [36]. Due to what has become known as the curse of dimensionality, this is especially useful for high dimensional data sets. For data of lower dimensions it may fail to exploit structure in the data, such as for example pruning away most of the data early on when querying an empty region.

Other examples of efforts to speed up search in R-trees include parallelizing the search, both using several disks [23] and using several machines [35]. The first approach may be applicable in the cases of NUMA architectures with multiple memory banks, but since the memory bandwidth of other nodes is more likely to be better exploited by the related processor, this is not directly applicable to memory resident R-trees.

Parallelizing the search across machines can easily be combined with memory resident R-trees, but since the work performed by each node is essentially a search through an R-tree, speeding up the single core performance of searches will also improve the parallel performance. Parallelizing across machines and speeding up single core performance are thus orthogonal improvements.

Chapter 2

Modern Computer Architectures

This thesis assumes a computer architecture based on the von Neumann architecture [34]. In short, the architecture consists of a control unit, an arithmetic unit, a memory and input/output units, of which the first three are shown in Figure 2.1.

The control unit reads and sees to the execution of instructions from memory. Operations performed by instructions may include

1. transferring data between the memory and the arithmetic unit,
2. performing calculations using the arithmetic unit and
3. reading data from input devices or writing data to output devices.

While these basics remain the same, a wide range of performance improvements have left the modern computer architectures much more complicated. Section 2.1 explores the memory subsystem, Section 2.2 takes a

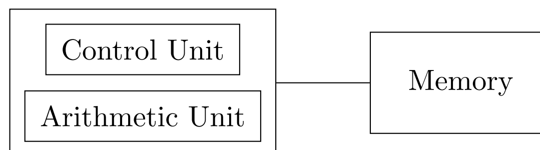


Figure 2.1: The central components of a von Neumann architecture, not considering input and output.

closer look at parallelism, before Section 2.3 introduces speculation. Finally, Section 2.4 introduces micro operations.

2.1 Memory Access

Already in the report by von Neumann, the memory is pointed out as the most challenging part of the machine due to the combination of speed and size requirements. Today, building fast memory can easily be done by placing the memory on the same chip as the processor, but because of practical limitation on die sizes, large memories are placed on separate chips with the disadvantage of higher access times and lower memory bandwidth.

The computational power of processors have increased faster than the bandwidth and speed of the memory system feeding them with data causing many computations to be bounded by the memory, a phenomena known as the memory wall [39]. Applications whose speed is limited by the memory bandwidth will be described as *memory bound*.

2.1.1 Memory Hierarchies

The low bandwidth and high latency of external memory can be mitigated, although not solved, by introducing a memory hierarchy. At the top of the hierarchy is the small and fast memory, which is closely tied to the processor. Since more storage is needed, a larger and slower memory is added. More levels can be added to the hierarchy to improve performance further.

In some cases, a level only holds data present at some lower level to speed up access and deliver more bandwidth, in which case the level is a cache. Several caches may appear at adjacent levels in the hierarchy.

As an example, consider the memory hierarchy in Figure 2.2. When the processor requests data from memory, the level 1 (L1) cache is first consulted. If the data is not present, the request is passed down the hierarchy to the level 2 (L2) cache. Similarly, the request is forwarded to memory if the requested data is not present in the L2 cache.

Since caches only store data present at minimum one other level in the memory hierarchy, they do not contribute to the total amount of memory, but merely decrease the latency for frequently accessed data. Because the cache is taking up space on the die that may otherwise have been used for processor cores, this makes determining the cache size a tradeoff between processing power, price and cache size [18].

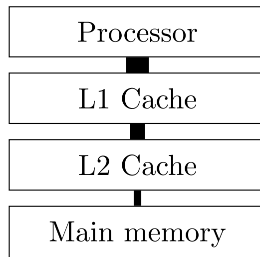


Figure 2.2: An example memory hierarchy consisting of level 1 cache (L1), level 2 cache (L2) and a main memory. A thicker line between the levels signals a higher bandwidth bus with lower latency access.

It should be noted that the access latency differences between the levels in the hierarchy may be quite significant. For example, the latency of the L1 cache in an processor based on the Intel Haswell architecture processor has been measured to around 1.6 ns [31]. The latency to L2 and L3 was measured to 4.8 ns and 21.2 ns respectively, but the memory clocked in at around 96.4 ns. At 2.5 GHz, this is equivalent to around 4, 12, 53 and 241 cycles respectively. Similar values have also been measured for the Intel Nehalem and Intel Sandy Bridge processors [29, 30].

Since the latency for data in memory is around 60 times longer than for data in L1 cache, programmers cannot ignore data locality when programming for performance. Placing and accessing data without considering the cache behavior can incur repeated performance hits of hundreds of cycles, which may slow down a program considerably.

Due to the large difference in access time for the cache immediately above memory and the memory itself, this cache is often referred to as the *last level cache* (LLC) regardless of how many caches exist above it.

Note that these numbers form main memory access are only valid for dynamic random access memory (DRAM) connected via a high speed bus to the processor. Some computer systems also have one or more disk drives for long term storage of large amounts of data. These tend to have much higher latencies than DRAM, which makes the latency of memory access negligible when data is stored on disk.

2.1.2 Data Units

Processors usually operate on a given number of bits at a time, which gives the natural width of registers and logic in the architecture. In the report by Von Neumann, this unit is referred to as a *minor cycle* and set to be 32 bits. In this report, the more modern term *computer word* will be used.

Although the processor is scaled to handle one computer word at a time, it may implement operations for larger or smaller units as well. An example of this is SIMD instructions, which will be covered in Section 2.2.2.

Furthermore, other components in the computer may be using other sizes. For example, the transfer unit between the main memory and caches tends to be larger than a computer word. This unit is often also the unit used by the caches to keep track of what data is currently in the cache, thus it will be referred to as a *cache line*.

In addition, disk drives are often partitioned into yet another unit, here referred to as the *disk page size*.

2.1.3 Non-Uniform Memory Access

For systems consisting of several processors or cores, it is not uncommon to include several memory banks to speed up memory access. Each memory bank may be more closely associated with a given set of cores than other. As a result, a given core may experience different memory access performance for different parts of the memory. This is referred to as *non-uniform memory access* (NUMA).

An example is illustrated in Figure 2.3 where two processors with a memory bank each can both access each other's memory. Since the latency (and possibly bandwidth) of access to the memory available is not uniform, this is an example of an architecture with NUMA.

A *NUMA aware* program is constructed to take advantage of NUMA architectures. As for the cache utilization, the NUMA architecture must be considered when programming for performance to avoid performance hits on a large number of memory accesses.

2.2 Parallelism

The ever increasing computational power of processors was for a long time partly a consequence of increasing clock rates, but this is no longer the

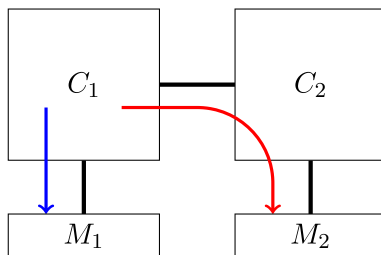


Figure 2.3: Example of a NUMA architecture where memory requests from the processor C_1 to the memory bank M_2 must be routed through another processor, C_2, as illustrated by the red arrow. Access to the memory bank M_1, on the other hand, goes directly and is thus quicker as illustrated by the blue arrow.

case [9]. Shrinking components have increased the power leakage to the point at which the heat simply cannot be transported away quickly enough, neither can the wires be made short enough to accommodate data transfer at higher frequencies. Instead, parallelism is harvested to squeeze more computational power out of the increasing number of transistors delivered by the industry as predicted by Moore's law.

2.2.1 Instruction-Level Parallelism

In many cases, some of the instructions in a program can be run in parallel. This is heavily exploited on modern CPUs where several instructions are processed simultaneously through *pipelining* and *out-of-order*.

Pipelining splits the execution of an instruction into several steps and executes one step after another until all steps have been executed. Each step normally maps on to a set of hardware resources, and since only one step of an instruction is executed at any time, the remaining hardware resources may be used to execute other instructions simultaneously.

For pipelining, branches thus pose a problem because the branch must be evaluated before the next instruction can be determined. This renders some of the hardware unused while the branch is being evaluated, limiting the potential performance gains of pipelining when the program contains a large proportion of branches.

Although less sever, instructions that uses some other instructions result as an input may also have to wait for other instructions to complete before

they can be started. Hardware solutions for forwarding the results of instruction before they are actually completed are often used. These solutions mitigate the problem, but do not eliminate it completely.

One way to increase the utilization of the hardware is to execute instructions *out-of-order*. When the next instruction cannot be executed because the results it require is not yet available, one of the following instructions may be executed instead as long as this does not change the result of the computations. To further boost performance, some of the hardware may be duplicated to allow several instructions to be in progress at the same step in the pipeline simultaneously.

In order to exploit the full potential of the processor, enough independent instructions must be available to fill the pipeline and exploit the possibly duplicated hardware. Since the time required to complete an instruction determines the number of independent instruction required to maximize the hardware utilization, this is an easier task for instructions completing quickly. For long-running memory requests, this is not so easy because the following instructions are likely to be dependent on the loaded value.

Additionally, since each instruction currently being executed requires some hardware resources, there is also a limit to the number of instructions that can be in progress at any given time.

Another technique for filling the pipeline found in modern processors is the use of *simultaneous multi-threading*. Instead of expecting a single thread of execution to provide enough instruction level parallelism, the processor executes two threads simultaneously on the same core. Instructions from one thread can thus be used to fill the gaps in the other thread's execution.

2.2.2 Data-Level Parallelism

Many processors today have support for SIMD (Single-Instruction Multiple-Data) instructions. These instructions, in combination with extra wide registers, allow operations to be performed on several values using a single instruction. This can reduces instruction overhead and increase performance as the operations can be performed in parallel at the hardware level.

Since several values are combined into a single register, the register is divided into *SIMD lanes*. The number of such lanes in a register may be configurable and depend on the data type. Instructions commonly perform some operation with the corresponding lanes of two source registers as operands, and stores the result in the corresponding lanes of a third register.

An example set of such instructions is the SSE and AVX extensions for Intel processors. The main challenge with SIMD instructions is designing software capable of exploiting all the available computational power. Automatic vectorization is a hot topic of research, but some algorithms and problems are inherently sequential or perform different operations on each data item.

2.3 Speculation

As one may have guessed from the previous sections, filling the pipeline is one of the largest challenges when working with modern processors. One way to make the unused resources useful, is to make a guess as to what is needed later and start he calculations ahead of time. This is termed *speculation*.

2.3.1 Prefetching

One example of such speculation is prefetching. By examining the memory access pattern, a hardware unit referred to as the *prefetcher* may be able to find a pattern that makes it possible to predict, with reasonable accuracy, which cache line will be required in the future. By issuing a speculative load for the cache line before it has been requested, the latency can in some cases be hidden.

As an example, modern Intel architectures can detect strided memory access patterns [21]. When memory is accessed with regular and sufficiently small intervals as illustrated in Figure 2.4, the processor starts fetching the next reference from memory to cache before it has been requested. This causes some memory access patterns to give significantly better performance than others.

On the flip side, the extra loads issued by the prefetcher may in some cases occupy resources needed by the running program, and thus reduce performance.

2.3.2 Branch prediction

Another source of speculation is typically *branch prediction*. The idea is to try to predict the result of a branch before it has been executed. After making a guess as to which way the branch goes, the instructions for the

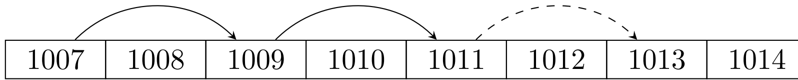


Figure 2.4: Memory locations with a strided access pattern. Assuming the memory locations 1007, 1009, 1011 has been accessed in order, it is reasonable to assume the next location accessed is going to be 1013.

result may be loaded into cache. If speculative execution is used, they may also start execution. This way, the pipeline can be kept busy while evaluating a branch.

As with prefetching, this may also reduce performance if the wrong result was predicted because the pipeline needs to be cleared before the correct instructions can be loaded and executed.

Usually, the results of speculatively executed instructions are stored until it can be determined whether the instructions should actually be executed, in which case the instructions are *retired*. When retired, the stored results become the new program state.

This also means that the retired instructions only include the instructions that would be executed without any speculation at all. The number of retired instructions may be very different from the number of instructions executed, as these also include the ones executed speculatively, but where the result was not used.

Unfortunately, the exact design of commercial branch predictors are often not published, but rather kept as a trade secret. This makes the behavior observed related to branch predictors hard to explain in some cases.

For example may the branch history be recorded using a small buffer indexed by the lower bits of the branch instruction address. In such cases, small changes in code may degrade or improve branch prediction performance other places in the code by changing the distance between branch instructions, which determines whether the instructions share their lower bits.

2.4 Micro Operations

The available instructions, resources and their behavior for a processor must be clearly defined such that programmers know what to expect of their pro-

grams. These definitions, often referred to as an *instruction set architecture* (ISA), is an abstraction useful for freeing the programmer from the burden of having to know the inner workings of the processor. In addition, it allows the same programs to be run on several different processors, as long as they implement the same ISA.

Unfortunately, the ISA may be designed for a different use case or be so old that the once reasonable assumptions are no longer valid. One such example is the x86 family of ISAs, often used in today's desktop and server computers, which has been around since 1978 [22].

To implement this efficiently on modern hardware, the instructions are decomposed into several *micro operations*. These can be seen as instructions only used internally by the processor, and their execution is commonly subject to the optimizations mentioned previously, such as pipelining and out-of-order execution.

The ISA does however rarely say anything about the performance of the operations it allows, thus the performance is implementation defined. The same instruction may perform differently on two different processors, both implementing the same ISA. When programming for performance, it is therefore often necessary to look through the abstraction to understand the performance of the code.

Translation from instructions to micro operations may not be straight forward. When mixed with out-of-order execution, pipelining and other optimizations, it may be hard to tell which instructions are blocking the processor, let alone what causes the blockage, as there may be several reasons.

Chapter 3

Spatial Indexing

Spatial indexing is concerned with speeding up search in spatial data. This section starts by presenting basic concepts of spatial search in Section 3.1 and spatial queries in Section 3.2. Then a closer look at the R-tree follows in Section 3.3.

3.1 Concepts

Several spatial indexing methods have been proposed to quickly look up data in a space [5, 6, 10, 28, 37]. All of these consider *spatial objects* within a real coordinate space of d dimensions, $\mathbb{R}^d$. Examples of such objects may be points, line segments, polygons or other shapes, as displayed in Figure 3.1. Ω is drawn with a dashed line, but it is just as much an object in the same way as A-E.

Definition 1 (Spatial object)**.** A point, line or other shape in space. The object X is a potentially infinite set of points such that $X \subseteq \mathbb{R}^d$.

Note that normal set operations apply to objects and have a geometric interpretation. For example, given two objects X and Y, the intersection $X \cap Y$ is another object in the same space including all the points where both X and Y occupy the same area. In the case where X and Y do not intersect, $X \cap Y = \emptyset$.

Because the objects are typically defined by points stored with a limited range for each coordinate, they often reside within a defined subset of $\mathbb{R}^d$, from here on referred to as the *data space*. Since the limited range of each

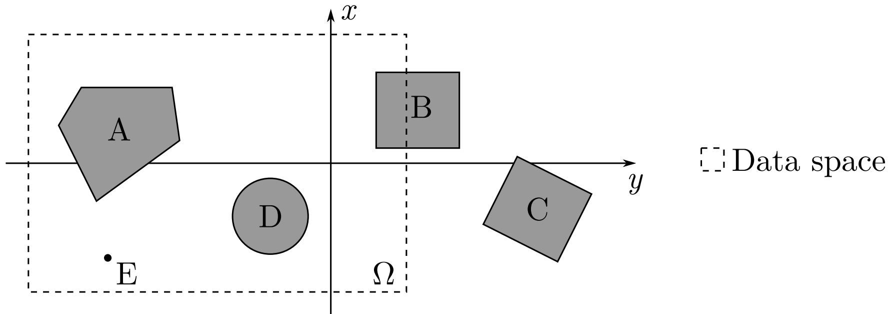


Figure 3.1: Illustration of a set of objects in $\mathbb{R}^2$. Ω and B are also axis-aligned boxes, while C is not axis-aligned, even though it is a box. The data set $\mathcal{D}$ consists of the three data objects A, D and E.

coordinate gives an upper and a lower bound in each dimension, this space has the shape of an *axis-aligned box* in $\mathbb{R}^d$. The data space and axis-aligned box is therefore defined as follows.

Definition 2 (Data space)**.** A data space Ω is a finite subspace of $\mathbb{R}^d$, where d is the dimensionality of Ω, thus $\Omega \subseteq \mathbb{R}^d$. It is assumed to be an axis-aligned box as defined by Definition 3.

Definition 3 (Axis-aligned box)**.** An axis aligned box is an object with the shape of a box aligned with the axes, defined by two points. Given two points $b, t \in \mathbb{R}^d$ such that

$$\forall i \in \{1, \ldots, d\} : b^{[i]} \leq t^{[i]},$$

the corresponding axis-aligned box $B = (b, t)$ is a spatial object consisting of the points in the set

$$B = \left\{ q \in \mathbb{R}^d \mid \forall i \in \{1, \ldots, d\} : b^{[i]} \leq q \leq t^{[i]} \right\}$$

where $p^{[i]}$ denotes the ith coordinate of the point p.

Figure 3.2 illustrates the definition of an axis-aligned box in $\mathbb{R}^2$, and Ω in Figure 3.1 is defined as a data space. The latter is however a bit arbitrary, since B also satisfies the requirements set forth by Definition 2. The objects B, C and Ω in Figure 3.1 are all boxes, but only B and Ω are axis aligned.

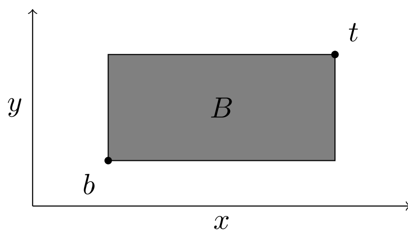


Figure 3.2: The shaded area is the set of points making up the axis-aligned box $B = (b, t)$.

When searching it is assumed only a specific set of the objects in the data space are considered. These objects form the *data set* $\mathcal{D}$ for the search and are referred to as *data objects*. All such objects in a data set must reside within the same data space to be searchable.

Definition 4 (Data set)**.** A data set is a set of object residing in the same data space. Given a data space Ω, the data set $\mathcal{D}$ in Ω is a set of spatial objects such that

$$\forall X \in \mathcal{D} : X \subseteq \Omega$$

Definition 5 (Data object)**.** A spatial object X is a data object with respect to a data set $\mathcal{D}$ if, and only if, $X \in \mathcal{D}$.

As an example, consider Figure 3.1, where A, D and E may be data objects with respect to the data space Ω. Since the definition does not require all objects within the data space to be in the data set, the set $\mathcal{D} = \{A, D\}$ is also a possible data set in this case. The set can however not include B or C, since they do not reside in Ω.

3.2 Queries

The main task of the spatial index is to speed up searching, which is the retrieval of data objects matching a given query.

A common and conceptually simple query is the *range query*, which requests all objects within certain bounds on each coordinate. Example use cases include finding all restaurants in the vicinity of a user with a known position, finding all images similar to another using features extracted from the images, or finding the visible elements for rendering in a map application.

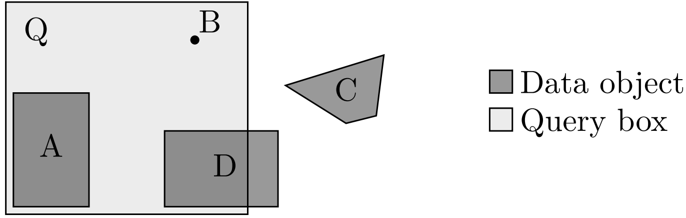


Figure 3.3: Example of a range query with Q. The result set consists of objects A, B and D as they intersect Q. C is outside Q and not included.

In practice, a range query is the same as requesting all data objects intersecting an axis-aligned box Q, referred to as the query box. Although the definition below includes all objects intersecting Q, the search can be restricted to objects enclosed in Q by filtering the results.

Definition 6 (Range search)**.** Given a data set $\mathcal{D}$ and an axis-aligned box Q, a range search returns a set of objects $\mathcal{R}$ given by

$$\mathcal{R} = \{X \in \mathcal{D} \mid X \cap Q \neq \emptyset\}$$

An example of a range query can be seen in Figure 3.3. Note that the query box Q is not a part of the data set.

3.3 R-trees

One data structure for speeding up search in spatial data is the R-tree [14], which can be seen as the multidimensional version of the popular B-tree.

Conceptually, the R-tree partitions the space it covers into a hierarchy of *minimum bounding boxes* (MBBs) as defined below. At the bottom of the hierarchy are the data objects.

Definition 7 (Minimum bounding box)**.** The minimum bounding box (MBB) of a spatial object X is the smallest axis-aligned box M that contains all points in X. In other words, given a spatial object X, the minimum bounding box $M = \text{MBB}(X)$ is such that

1. M is an axis-aligned box,
2. $X \subseteq M$, and

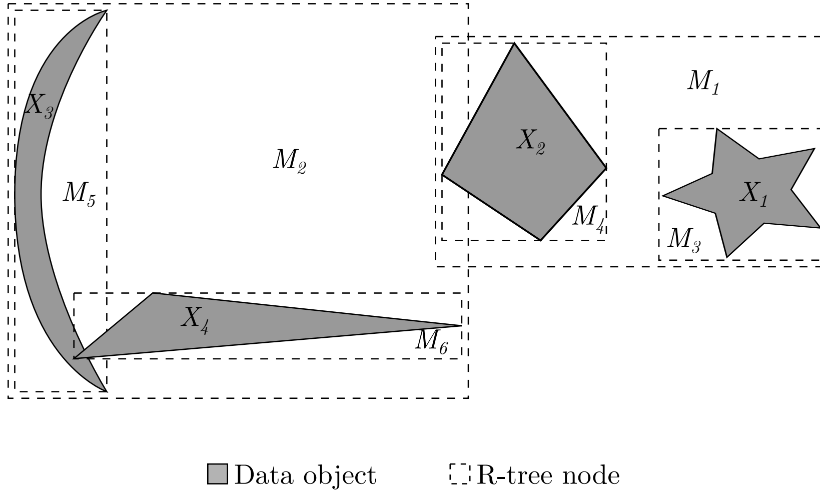


Figure 3.4: Example R-tree in 2 dimensions. In this case, M_1 through M_6 are the MBBs in the hierarchy and X_1 through X_4 are the data objects.

3. no M' satisfying Items 1 and 2 exists such that $|M'| < |M|$.

Each MBB in the hierarchy is required to contain all child MBBs. As an example, consider the visualized R-tree in Figure 3.4. M_1 and M_2 are the MBBs at the top of the hierarchy, each containing two other MBBs. Note that the MBBs may overlap even when one is not a parent of the other, as is the case for M_1 and M_2, and M_2 and M_4.

From a more practical perspective, the R-tree is a tree structure where each node is a collection of several entries, each consisting of an MBB and a pointer, as visualized in Figure 3.5. For internal nodes, the pointers point to child nodes in the tree; for leaf nodes, they point to data objects. Furthermore, the MBB associated with an entry is always set to the MBB containing whatever the pointer points to.

To keep the tree balanced and node sizes reasonable, the number of children in each node, except from the root node, is always between m and M, where $m \leq \frac{M}{2}$. Analogous to the B-tree, a node is split into two should an insertion be in conflict with this requirement, and a split may propagate to the top of the tree, adding another level. This keeps all leafs of the tree at the same depth at all times.

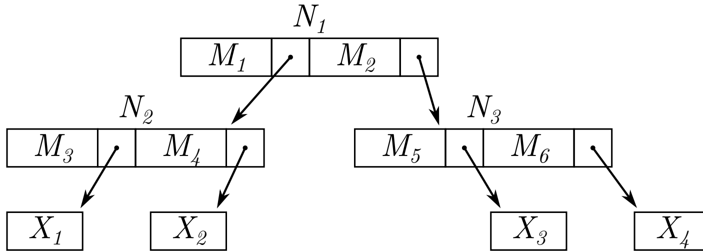


Figure 3.5: Example structure of an R-tree where each node has 2 children. The data objects are illustrated at the bottom denoted X_1 through X_4, and M_1 through M_6 are the MBBs as depicted in Figure 3.4.

More rigid definitions of entries and nodes follow, such that the following discussions on R-tree search and construction can be simplified.

Definition 8 (R-tree entry)**.** An entry in the R-tree is a either an R-tree node (Definition 9) or a data object (Definition 5).

Definition 9 (R-tree node)**.** A node N in the R-tree is a set of entries as defined in Definition 8. Unless N is the root node, it also satisfies the restriction

$$m \leq |N| \leq M,$$

where m is the minimum node fill grade and M is the maximum node size.

For the MBB of an entry E, MBB(E), to be well defined in all cases, a definition for the MBB of a node N is needed. This definition is recursive and therefore ends up including all leafs below N.

Definition 10 (Minimum bounding box of node)**.** In the case where N is an R-tree node, MBB(N) is the MBB containing all children of N,

$$\text{MBB}(N) = \text{MBB}(\bigcup_{E \in N} \text{MBB}(E))$$

Note that the definition of an entry omits the MBB, simply because the MBB can be generated based on the other information stored in the tree, and therefore works more like a cache. As this is uninteresting from a theoretical perspective, this caching is assumed to be implicit and is therefore not mentioned in most algorithms. Do however keep in mind that retrieving the MBB of an entry is usually a low cost operation in practice due to the caching.

3.3.1 Range Search

A range search is performed by traversing the tree, pruning away nodes where the respective MBB does not intersect the query object Q, as described in Algorithm 3.1. This is based on the insight that any node N' in the sub tree below a node N is contained within MBB(N). This is what makes searching through an R-tree faster than a linear scan in many cases.

In contrast to range searches in B-trees, the algorithm for R-trees may visit the entire tree, and the complexity is therefore linear in the number of nodes in the worst case. The actual performance depends heavily on the strategy used to group objects into nodes when constructing the tree, and many strategies have been developed [2, 6, 14], as will be covered in more detail in Chapter 4.

As can be seen in Algorithm 3.1, each call to `RangeSearch` results in the entries of a node being iterated through while checking whether each entry intersects the query box. For the remainder of this thesis, this will be referred to as a *node scan*. A search can then be viewed as a series of such node scans, possibly interrupted by another node scan before continuing as indicated by the recursive call in the algorithm.

```
Function RangeSearch(N, Q)
    forall E ∈ N do
        if MBB(E) ∩ Q = ∅ then
            continue;
        end
        if E is node then
            RangeSearch(E, Q);
        else
            report E;
        end
    end
end
```

Algorithm 3.1: Range search in R-tree. N is usually the root node for the first call, and Q is the query box.

As a side note, one may wonder why the R-tree has not been structured as the alternative version in Figure 3.6, where the MBBs have been pushed

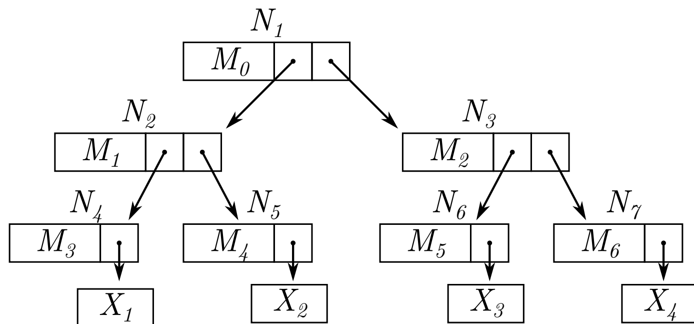


Figure 3.6: A less traditional version of the R-tree where each node only has one MBB or a data object.

one level down the tree, resulting in only a single MBB in each node. This would arguably simplify the concept, since each node contains an MBB enclosing its children, obviating the need for a definition of an entry.

The first observation may be that the alternative version has an extra level, which causes it to have more pointers than the original R-tree. The number of MBBs does however not increase considerably.

More importantly, observe that during a node scan, an intersection test will be performed between the query box and the MBB of each child. For the alternative version, this means loading every child of the visited node, including the pointers to the grandchildren. Loading the pointers to the grandchildren could probably be avoided, but would result in random memory access, which is not as preferable as sequential access when performance matters, as explained in Chapter 2.

3.3.2 Generic Insert Algorithm

Many of the algorithms for inserting objects into an R-tree structure are based on the same basic idea, presented in Algorithm 3.2. First, the algorithm drills down the tree to find a suitable node into which the new data object should be inserted. Next, the changes are propagated up the tree, possibly splitting nodes on the way.

The observant reader will notice that O is actually an entry in Algorithm 3.2, and not a data object as would probably be more expected. The reason for this will become apparent when the R*-tree is introduced. On

```
Function Insert(O, N)
    if O belongs at the level of N then
        add O to N;
    else
        E ← ChooseSubtree(N, O);
        (N, N′) ← Insert(O, E);
        if N′ ≠ ∅ then
            add N′ to N;
        end
    end
    if N is overfull then
        return SplitNode(N);
    end
    return (N, ∅);
end
```

Algorithm 3.2: Generic insert algorithm for dynamic construction of an R-tree, where N is a node and O is an entry. `ChooseSubtree` and `SplitNode` depends on the implementation.

the first call, $O = (M, p)$ will always be an entry where M is the MBB of, and p is a pointer to the data object to insert. Similarly, N will always be the root node.

The `ChooseSubtree` method selects a child from the given node, which determines which sub tree the new object should reside in. Thus this method is also ultimately responsible for selecting the leaf node in which the new object should be placed.

`SplitNode` accepts a node and divides its entries into two groups, which will be referred to as a *split*. The new node groups are returned and replace the node that was split in its parent.

Should the algorithm return a pair where the second element is a non-empty set, the root node is replaced by a node containing both elements from the returned pair. This is how the tree gains height. Otherwise, the returned node is the root of the new tree.

Chapter 4

R-tree Variants

As mentioned in Section 3.3, the actual performance of the R-tree depends heavily on the method used to group data objects into nodes as the worst case performance is linear in the number of nodes.

This section considers several algorithms for constructing an R-tree. Two classes of such algorithms exist. The first is the static approaches, where all the data objects are known before construction begins. The second is the dynamic class, where new items can be inserted at any time. Only the latter will be considered here. Assuming that deletions are rare, the scope is also limited to insertions only.

This section starts off with a few prerequisites in Section 4.1, before the Quadratic R-tree, R*-tree, Revised R*-tree and Hilbert R-tree are introduced in Sections 4.2 to 4.5.

4.1 Prerequisites

Since constructing an R-tree is largely an optimization problem, it is often useful to compare elements based on the value of a function. Many of the algorithms will however also require some sort of a tie breaker. To simplify the notation of such tie breakers, a partial order over tuples is defined.

Definition 11 (Partial order over tuples)**.** Given two tuples, $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$, the one with the lowest value in the first element where they differ is the lowest. More formally,

$$A \leq B \Leftrightarrow \exists i \in [1, n] : a_i \leq b_i \ \wedge \ \forall j \in [1, i] : a_j = b_j$$

To see how this can be useful, consider the following example. Let f be a function, and assume the set of values $X \subseteq D$ minimizing a function f is sought. This can easily be expressed using the common argmin operator as

$$X = \underset{x \in D}{\operatorname{argmin}} f(x)$$

Extending this concept to also include a second function g used as tie breaker can conveniently be done using the partial order over tuples by defining a third function

$$f'(x) = (f(x), g(x))$$

which allows the direct application of argmin, giving

$$Y = \underset{x \in D}{\operatorname{argmin}} f'(x)$$

where $Y \subseteq D$ is the set of values minimizing f using g as tie breaker.

4.2 Quadratic R-tree

When Guttman published the first paper on R-trees [14], he also included three different construction algorithms. They have been named based on the computational complexity of the `SplitNode` method; one is linear, one is quadratic and the last checks all possible combinations and is thus exponential in the number of entries in the node.

Since the linear algorithm generally performs very poorly, and the exponential algorithm tends to be too expensive, the quadratic version is arguably the most popular of the three and is therefore described here. All three variants concentrate on the volume of the MBBs in the tree, as defined below.

Definition 12 (Volume)**.** For an axis-aligned box $B = (b, t)$, $\operatorname{vol} B$ is the product of its extent in all dimensions. Thus

$$\operatorname{vol} B = \prod_{i=1}^{d} \left(t^{[i]} - b^{[i]} \right)$$

The volume of X, where X is not an axis aligned box and $\operatorname{MBB}(X)$ exists, is given by $\operatorname{vol}(\operatorname{MBB}(X))$.

When selecting the sub tree, the quadratic algorithm simply selects the node whose MBB needs the least volume *enlargement* to include the new object, as is visible in Algorithm 4.1. Should there be a tie, the one with the least volume is selected.

Definition 13 (Enlargement operator)**.** Given two MMBs G and M, and a function f, the enlargement required to include M in G is the change in the value of $f(G)$, when M is added to G. This enlargement, denoted $\Delta_M f(G)$, is thus given by

$$\underset{M}{\Delta} f(G) = f(G \cup M) - f(G)$$

Function `ChooseSubtree(`N, O`)`

  return $N' \in \underset{E \in N}{\operatorname{argmin}} \left(\underset{O}{\Delta} \operatorname{vol} E, \operatorname{vol} E \right)$;

end

Algorithm 4.1: The `ChooseSubtree` algorithm used by the quadratic R-tree.

For the `SplitNode` method, which can be found in Algorithm 4.2, the two entries that would *waste* the most space when placed in the same node are selected and assign to different groups. This waste is loosely defined as the volume in the MBB of both entries that is not occupied by any of the entries. More strictly, given two MBBs M_1 and M_2, the wasted space $W(M_1, M_2)$ is given by

$$W(M_1, M_2) = \operatorname{vol}(M_1 \cup M_2) - \operatorname{vol} M_1 - \operatorname{vol} M_2$$

which may very well be negative when M_1 and M_2 overlap.

Next, the algorithm repeatedly selects the entry among the remaining where the difference between placing it in one group compared to the other is the largest, as measured using volume enlargement. Should there be a tie, the group with the lowest volume is selected, followed by the group with the fewest elements.

One interesting observation is that the quadratic R-tree mainly tries to minimize the volume of the MBBs. This makes sense because a large volume implies a greater chance of intersection with the query box, thus increasing the number of nodes that has to be scanned during a search.

Function `SplitNode`(N)

$(E_1, E_2) \leftarrow \operatorname*{argmax}_{(E_1,E_2)\in N\times N} W(E_1, E_2)$;

$S \leftarrow N \setminus \{E_1, E_2\}$;

$G_1 \leftarrow \{E_1\}$;

$G_2 \leftarrow \{E_2\}$;

while $S \neq \emptyset$ **do**

$E \leftarrow E \in \operatorname*{argmax}_{E\in S} \left| \underset{E}{\Delta} \operatorname{vol} G_1 - \underset{E}{\Delta} \operatorname{vol} G_2 \right|$;

if $(\Delta_E \operatorname{vol} G_1, \operatorname{vol} G_1, |G_1|) < (\Delta_E \operatorname{vol} G_2, \operatorname{vol} G_2, |G_2|)$ **then**

$G_1 \leftarrow G_1 \cup \{E\}$;

else

$G_2 \leftarrow G_2 \cup \{E\}$;

end

$S \leftarrow S \setminus \{E\}$;

end

return (G_1, G_2);

end

Algorithm 4.2: The `SplitNode` algorithm used by the quadratic R-tree.

Unfortunately, a MBB may be very likely to intersect the query box without having any volume at all, which may happen if the MBB covers the entire data domain in all dimensions with the exception of one. A more rigid justification for this can be found in Appendix B.

4.3 R*-tree

The R*-tree, introduced by Beckmann and Seeger [2], introduces the concept of reinsertion to the R-tree, borrows some ideas from Greene's split [13] and also takes the *perimeter* and *overlap* of the resulting MBBs into account.

Definition 14 (Perimeter)**.** Given an axis-aligned box $B = (b, t)$, the perimeter of B is the sum of its extension in all dimensions. More formally, the perimeter $\operatorname{perim} B$ is given by

$$\operatorname{perim} B = \sum_{i=1}^{d} \left(t^{[i]} - b^{[i]} \right)$$

The perimeter of X, where X is not an axis-aligned box and $\operatorname{MBB}(X)$ is defined, is given by $\operatorname{perim}(\operatorname{MBB}(X))$.

Definition 15 (Overlap operator)**.** Given a set of axis-aligned boxes $\mathcal{X}$ and a function $f : MBB \to \mathbb{R}$, the overlap of an axis-aligned box B with $\mathcal{X}$ is the sum of the intersections between B and the other axis-aligned boxes in $\mathcal{X}$ as calculated by f. In other words, the overlap $\Omega_{\mathcal{X}} f(B)$ is given by

$$\underset{\mathcal{X}}{\Omega} f(B) = \sum_{B' \in \mathcal{X}} f(B \cap B')$$

More precisely, the `ChooseSubtree` method of the R*-tree, featured in Algorithm 4.3, first checks whether the entry to insert is contained within any of the children. If so, the one with the least volume is selected. Otherwise, the algorithm minimizes volume enlargement as in Gutmann's quadratic algorithm when given an internal node, while the child having the least volume overlap enlargement with respect to the other children in the node is used when given a leaf node.

Since calculating the overlap enlargement is quadratic in the node size, Beckmann and Seeger suggests selecting a group of p children with the lowest

Function `ChooseSubtree(`N, O`)`

  $S \leftarrow \{E \in N \mid \mathrm{MBB}(O) \subseteq \mathrm{MBB}(E)\}$;

  if $S \neq \emptyset$ **then**

    return $\operatorname*{argmin}_{E \in S} \operatorname{vol} E$;

  end

  if N *is internal* **then**

    return $\operatorname*{argmin}_{E \in S} \left(\underset{O}{\Delta} \operatorname{vol} E, \operatorname{vol} E \right)$;

  end

  return $N' \in \operatorname*{argmin}_{E \in N} \left(\underset{O}{\Delta} \underset{N}{\Omega} \operatorname{vol} E, \underset{O}{\Delta} \operatorname{vol} E \right)$;

end

Algorithm 4.3: The `ChooseSubtree` algorithm used by the R*-tree.

volume, and to only consider these when minimizing the overlap enlargement. They found $p = 32$ to be a good value, and claim this has negligible impact on the query performance, while reducing the computations required to insert new data objects.

During splits, for which the algorithm is show in Algorithm 4.4, the R*-tree evaluates several *split candidates*, as defined below. Since the number of split candidates is exponential in the node size, the R*-tree only considers a well selected handful.

Definition 16 (Split candidate)**.** A split candidate is a pair of sets (S_1, S_2) that partition the set of entries S such that S_1 and S_2 would both make legal nodes in the R-tree. In other words, the pair (S_1, S_2) is a split candidate for set of entries S if

- $S_1, S_2 \subseteq S$,
- $m \leq |S_1|, |S_2| \leq M$, and
- $S_1 = S \setminus S_2$,

where m is the minimum node fill grade and M is the maximum fill grade.

```
Function SplitNode(N)
    if this is not a reinsert then
        while |N| > M − p + 1 do
            E ← argmax_{E∈N} d_c(E, N);
            N ← N \ {E};
            schedule Insert(E, R);
        end
        return (N, ∅);
    end
    i ← argmin_{j∈{1...d}} ( min_{(S_1,S_2)∈L_i(N)} (perim S_1 + perim S_2) );
    (N, N') ← argmin_{(S_1,S_2)∈L_i(N)} vol (S_1 ∩ S_2);
    return (N, N');
end
```

Algorithm 4.4: The `SplitNode` algorithm used by the R*-tree, where R is the root node. The **schedule** keyword schedules its operand for execution in a first-in first-out manner.

For each dimension i, the entries are sorted by the lower coordinate of their MBBs in dimension i. Then the entries are partitioned by index such that the two halves constitute a split candidate.

More precisely, the split set considered by the R*-tree for dimension i and a set of entries S, is the set of pairs $L_i(S)$ given by

$$L_i(S) = \left\{ (S_1, S_2) \in C(S) \mid \max_{E_1 \in S_1} b_1^{[i]} \leq \min_{E_2 \in S_2} b_2^{[i]} \right\}$$

where $C(S)$ is the set of all possible split candidates for S and $\mathrm{MBB}(E_i) = (b_i, t_i)$ for $i \in \{1, 2\}$.

Presumably to lower the cost, the R*-tree first selects a dimension by calculating the perimeter of each split candidate and adding them together. The dimension associated with the candidate yielding the lowest sum is selected. Next, one of the split candidates in $L_i(N)$ for the decided dimension is selected by minimizing the intersection volume.

Moreover, the R*-tree actually avoids splitting a node at all in many cases by reinserting a set of entries instead of splitting a node. When reinserts are performed, the p entries having their center farthest from the center of the node are selected for reinsertion. The distance between the centers of two MBBs $M_1 = (b_1, t_1)$ and $M_2 = (b_2, t_2)$ is given by

$$d_c(M_1, M_2) = \sqrt[d]{\sum_{d \in [0,1)} \left(\frac{1}{2}(b_1 + t_1) - \frac{1}{2}(b_2 + t_2) \right)^2}$$

To avoid endless reinsertion, a reinsert is not allowed to trigger a new reinsert at the same level.

Not only does reinsertion increase the fill grade of the tree, it also allows the tree to adapt to a changing data distribution, which may be the case when the first entries inserted do not reflect the distribution of the remaining entries. This comes at the cost of a more expensive insert operation.

4.4 Revised R*-tree

In their next revision of the R*-tree [4], Beckmann and Seeger drops the reinsert strategy and instead records the change in a nodes MBB from its creation time to the time it is split. This allows splits to be skewed in such a way that more space is available in the spots where many new objects

are expected to appear in the future. Other improvements include better handling of MBBs without volume.

The ChooseSubtree algorithm used by the revised R*-tree shown in Algorithm 4.5 starts out in a similar manner to that of the R*-tree, by checking whether the new object is contained within any of the children, but where the R*-tree only uses volume to distinguish between several such children, the revised R*-tree uses perimeter whenever at least one of the relevant children has no volume.

```
Function ChooseSubtree(N, O)
    S ← {E ∈ N | O ∈ MBB(E)};
    if S ≠ ∅ then
        if ∃E ∈ S : vol(E) = 0 then
            return N ∈ argmin_{E∈S} perim(E);
        else
            return N ∈ argmin_{E∈S} vol(E);
        end
    end
    S′ ← { E ∈ argmin_{E′∈N} ( Δ_O perim(E′) ) : ΔΩ_{O N} perim(E) = 0 };
    if S′ ≠ ∅ then
        return N ∈ S′;
    end
    E_0 ← E′ ∈ argmin_{E∈N} ( Δ_O perim E );
    B ← { E ∈ N : Δ_O Ω_{{E_0}} perim E ≠ 0 };
    P ← { E ∈ N : ∃E′ ∈ P_0 : Δ_O perim E ≤ Δ_O perim E′ };
    return N ∈ argmin_{E∈P} ( ΔΩ_{O P} perim(E) );
end
```

Algorithm 4.5: The ChooseSubtree algorithm used by the Revised R*-tree.

Should no child contain the data object, the child having the lowest perimeter enlargement is select if this child also suffers no overlap enlargement, as measured using perimeter.

Finally, if no child has been selected so far, the entry with the lowest perimeter enlargement is set as E_0 and a limited set of children P is chosen to be all children E where there exists an entry E', with non-zero overlap enlargement with respect to E_0, whose perimeter enlargement is greater than the overlap enlargement of E. The child in P with a minimal overlap enlargement with respect to P is returned.

To cut the cost of this procedure in the case where a child $E \in P$ with no overlap enlargement with respect to P exists, the entries are traversed in a depth-first manner, where an edge is present between two children if they overlap. This search computes the overlap of all children in P simultaneously by accumulating the overlaps found during the traversal.

For the `SplitNode` algorithm, the revised R*-tree reduces the problem to a single optimization problem of a goal function. The set of split candidates considered does however differ depending on whether the node to be split is internal or not. All dimensions are considered for internal nodes, but only the dimension with the lowest sum of perimeters is used for leaf nodes.

Function `SplitNode(`N`)`
 if *N is a leaf* **then**
  $i \leftarrow \underset{i \in \{1 \ldots d\}}{\operatorname{argmin}} \left(\sum_{(S_1, S_2) \in P_i} (\operatorname{perim}(S_1) + \operatorname{perim}(S_2)) \right);$
  $S \leftarrow P_i;$
 else
  $S \leftarrow \bigcup_{i \in \{1 \ldots d\}} P_i;$
 end
 return $(N, N') \in \underset{(S_1, S_2) \in S}{\operatorname{argmin}} w(S_1, S_2);$
end

Algorithm 4.6: The `SplitNode` algorithm used by the Revised R*-tree.

Given a dimension i, the revised R*-tree uses the same set of split candidates as the R*-tree, but also includes all splits yielded by sorting by the

upper coordinate, in addition to the lower. The additional set U_i can be defined in a similar manner to L_i,

$$U_i(S) = \left\{ (S_1, S_2) \in C(S) \mid \max_{E_1 \in S_1} t_1^{[i]} \leq \min_{E_2 \in S_2} t_2^{[i]} \right\}$$

where $\mathrm{MBB}(E_i) = (b_i, t_i)$ for $i \in \{1, 2\}$, such that the full set of split candidates for dimension i is given by

$$P_i(S) = L_i(S) \cup U_i(S)$$

Finally, a split candidate minimizing a goal function is selected from the set of split candidates considered. The goal function $w(S_1, S_2)$ is a combination of two other functions, $wg(S_1, S_2)$ and $wf(S_1, S_2)$, and is given by

$$w(S_1, S_2) = \begin{cases} wg(S_1, S_2) \cdot wf(S_1, S_2) & \text{if } wg(S_1, S_2) < 0 \\ wg(S_1, S_2) / wf(S_1, S_2) & \text{otherwise} \end{cases}$$

The function $wg(C)$ evaluates the split by measuring the size of the overlap, where less is better. This is similar to the R*-tree, but with two major changes. First, perimeter is used instead of volume whenever at least one of the split candidates lacks volume. Second, the combined perimeter of the two partitions is used to distinguish between split candidates when the MBBs of the two partitions do not intersect. This gives

$$wg(S_1, S_2) = \begin{cases} \mathrm{perim}(S_1) + \mathrm{perim}(S_2) - p_{\max} & \text{if } \mathrm{MBB}(S_1) \cap \mathrm{MBB}(S_2) = \emptyset \\ f(\mathrm{MBB}(S_1) \cap \mathrm{MBB}(S_2)) & \text{otherwise} \end{cases}$$

where $f = \mathrm{perim}$ if at least one split candidate lacks volume and $f = \mathrm{vol}$ otherwise, and $p_{\max}$ is the maximum possible value of $\mathrm{perim}(C_1) + \mathrm{perim}(C_2)$ as judged by the size of $\mathrm{MBB}(N)$. Beckmann and Seeger shows this value is given by

$$p_{\max} = 2 \, \mathrm{perim} \, N - \min_{i \in \{1, \ldots, d\}} \left(t^{[i]} - b^{[i]} \right)$$

where $\mathrm{MBB}(N) = (b, t)$.

When it comes to $wf(C)$, it takes the trend of the MBB expansion into account for dimension i and acts more like a scaling factor for wg. Given that the MBB of the node has mainly expanded in one direction since the last split, it is reasonable to assume that it will continue to expand in that

direction. Or in other words, that the distribution of new entries will be skewed in the very same direction.

Thus, to avoid extra splits and improve the structure of the tree, the revised R*-tree does a skewed split so that there is more space for new entries where they are expected to appear. This is accomplished by storing the MBB of each new node in the node itself at creation time, and calculating an asymmetry score for dimension i during splits,

$$a_i = \frac{(t_1^{[i]} + b_1^{[i]}) - (t_0^{[i]} + b_0^{[i]})}{t_1^{[i]} - b_1^{[i]}}$$

where $\mathrm{MBB}(N) = (b_1, t_1)$ is the current MBB of N and $MBB_0(N) = (b_0, t_0)$ is the MBB of N at creation time.

Assuming only insertions, or in other words that $\mathrm{MBB}_0(N) \subseteq \mathrm{MBB}(N)$, then $a_i \in [-1, 1]$. Thus, a_i can be transformed into an index that can be used for splitting by linear scaling. This index, μ_i, can be written

$$\mu_i = \left(\frac{1 - 2m}{M + 1}\right) a_i$$

which must be rounded to yield an actual index.

Since using the asymmetry score alone to select a split would result in a severely reduced number of split candidates, possibly leaving out other good splits, the function $wf(C)$ instead yields a score where μ_i gets the best score. More precisely, this function is an adjusted Gaussian function and is given by

$$wf(C) = y_s \left(\exp\left(-\left(\frac{x(C) - \mu_i}{s(1 + |\mu_i|)} \right)^2 \right) - y_1 \right)$$

where

$$y_1 = \exp\left(-\frac{1}{s^2}\right), \quad y_s = \frac{1}{1 - y_1}, \quad x(C) = \frac{2|C_1|}{M + 1} - 1,$$

and s is a parameter controlling the shape of the curve, for which Beckmann and Seeger found $s = 0.5$ to be a decent value. The dimension i is the dimension used to obtain the split (S_1, S_2).

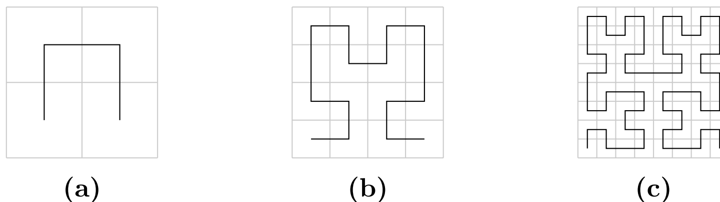


Figure 4.1: The first 3 orders of a Hilbert curve in 2 dimensions.

4.5 Hilbert R-tree

In contrast to the other R-tree variants, the Hilbert R-tree does not try to devise a good split strategy in the d dimensional space directly. Instead, the MBBs are mapped to a single dimension using *Hilbert encoding* as described below. This allows the nodes of the R-tree to be sorted, which simplifies the splitting algorithm and makes the R-tree function more like a B*-tree on updates. Since the MBBs are still a part of the tree, searching can still be done using the same algorithms as for the regular R-tree.

Before describing the insertion algorithm used by the Hilbert R-tree in Section 4.5.3, Hilbert encoding is introduced in Section 4.5.1 and an algorithm useful for performing the actual encoding is given in Section 4.5.2.

4.5.1 Hilbert Encoding

David Hilbert described a curve capable of filling a square [17], also known as a space filling curve. He stresses the fact that this curve gives a unique mapping between the points on a continuous curve and the points of a square.

As can be seen in Figure 4.1, the curve is developed in steps. Starting with a square and dividing it into 4 quadrants with a predefined curve giving the order of the squares, gives a first order Hilbert curve, as visible in Figure 4.1a.

The next step in generated by copying the first square four times to fill the four quadrants of a new square and rotating the bottom two before connecting the ends, giving the curve in Figure 4.1b. By further copying, rotation and connection of the pattern at the step before, a Hilbert curve of order 3 emerges, as seen in Figure 4.1c, and so forth. By using a curve of a sufficiently high order, the mapping between the two dimensional space and a line can be made arbitrarily accurate.

Although Hilbert never explores the possibility of using the curve for mapping the points in a hyper cube of an arbitrary dimension onto a line, this has also shown to be possible [1]. Even though the mapping is unique given a set of rules to generate the curve, several sets of rules is possible for the generalized version of the curve [15].

More formally, the generalized Hilbert encoding maps a point in a d dimensional space to a single value in one dimension. The Hilbert encoding thus be expressed as a function $H_d : [0,1\rangle^d \rightarrow [0,1\rangle$, such that $H_d(p)$ is the distance along the Hilbert curve before reaching the square in which point p resides, starting at the end of the curve residing in the lower half of the square along the x axis.

One important property of the Hilbert curve is its locality preserving behavior. That is, points that are close to each other in the original domain tend to be close in the co-domain. This property can obviously not hold for any point because the number of points within a given distance grows with the number of dimensions, but the reverse is true; if the images of two points are close in the co-domain, the points are also close in the domain. This follows directly from the triangle inequality since the curve is continuous,

$$l(p_1, p_2) \leq |H_d(p_1) - H_d(p_2)|$$

where $p_1, p_2 \in \mathbb{R}^d$ are points, and $l(p_1, p_2)$ is the distance between them.

4.5.2 Encoding Algorithm

Given a point $p \in \mathbb{R}^d$, finding the image $H_d(p)$ can be done in an iterative manner such that the accuracy of the result improves with every iteration. A modified version of the algorithm given by Lawder [26] can be found in Algorithm 4.7. It drills down through the hierarchy of boxes, one step each iteration, and each iteration thus adds d bits to the result.

As shown in Figure 4.1, the Hilbert curve partitions the space into 2^d partitions by slicing the original data space in two equal parts for each dimension. Since the value of $H_d(p)$ depends on which of these partitions p resides in, the first step of the algorithm is a set of comparisons to extract a set of dimensions B where p resides in the upper half.

At the end of the algorithm, the set B is used to generate the output for the current iteration. This generation is based on the observation that the curve needs to go back where it came from for all axes, except the first. This can be seen in Figures 4.1a to 4.1c where the curve first goes from low

```
Function HilbertEncode(p)
    B ← {i ∈ {1,...,d} | p_i > 1/2};
    /* Calculate transform                                                  */
    r ← { max_{i∈B}(i) − 1   if B ≠ ∅
        { d                  otherwise ;
    B′ ← { B ⊖ {d}           if |B| is odd
         { B ⊖ {d} ⊖ {r}     otherwise ;
    /* Transform p to q                                                     */
    for i ∈ {1...d} do
        q^[i] ← { p^[i]       if i ∈ B
                { p^[i] − 1   otherwise ;
    end
    for i ∈ B′ do
        q^[i] ← 1 − q^[i];
    end
    q = (q^[(1−r) mod d+1], q^[(2−r) mod d+1], ..., q^[(n+1−r) mod d+1]);
    /* Convert to Hilbert encoding                                          */
    for i ∈ {1,...,d} do
        h_i = { 1   if |{j ∈ B | j ≤ i}| is odd
              { 0   otherwise ;
    end
    return h_1 ... h_d HilbertEncode(q);
end
```

Algorithm 4.7: Recursive version of the Hilbert encoding algorithm presented by Lawder [26]. The endless recursion is usually halted when enough bits has been collected.

to high coordinates along the y axis, but the order is opposite after crossing the x axis.

Since d bits are needed to specify the box in which p resides, the output from each iteration is d bits. The output bit for dimension i is generated by checking whether $i \in B$ and then inverting the result if the number of earlier dimensions where p resides in the upper half is odd. This can be simplified by including i in the count and set the bit if the count is odd.

Finally, the point p is transformed to create a new point q which is rotated to the local coordinate system of the Hilbert curve inside the partition in which p resides. This is done by first constructing a set B' and an integer r.

The integer r defines a rotation of the axes, both from a symbolic and geometric perspective. From a symbolic perspective, the rotation is applied by left shifting the axes, such that what was previously the ith axis instead becomes the $((i - r) \mod d)^{th}$ axis. This does however correspond to a rotation also from a geometric perspective. The purpose of this rotation is to ensure the curve travels along the correct dimension.

Finding r is done by finding the highest dimension for which p resides in the upper half. This dimension gives the last position in the output at which the bit value is different from the corresponding bit in the output from the previous iteration. By following the lines in Figure 4.1, one can observe that the line travels in the direction of dimension i every time the bit corresponding to i in the output changes from unset to set.

The set B' is used to mirror p through a hyper plane perpendicular to the dimensions present in B'. This ensures the output of one partition is aligned with the input of the next, and that the values are increasing along the line.

4.5.3 Insert Algorithm

As stated in the introduction, the Hilbert R-tree acts more like a B*-tree when the tree is constructed, since it only takes the Hilbert value of a data object into account when deciding where it should be placed. The Hilbert value of a data object O is calculated as

$$H_d(O) = H_d(p)$$

where p is the center point of MBB(O).

For internal nodes, the MBB is not used at all. The Hilbert value is instead calculated as the maximum of the Hilbert values assigned to the children. In other words, given a node N, the Hilbert value is given by

$$H_d(N) = \max_{E \in N} H_d(N)$$

For the `ChooseSubtree` method as shown in Algorithm 4.8, the Hilbert R-tree simply picks the node where the Hilbert value is as low as possible without being lower than that of the new data object, in other words the lower upper bound. Should no such node exist, the node with the maximum Hilbert value is selected (upper lower bound).

```
Function ChooseSubtree(N, O)
    S ← {E ∈ N | H_d(E) ≥ H_d(O)};
    if S = ∅ then
        return N′ ∈ argmax_{E∈N} H_d(E);
    end
    return N′ ∈ argmin_{E∈S} H_d(E);
end
```

Algorithm 4.8: The `ChooseSubtree` algorithm used by the Hilbert R-tree.

As Algorithm 4.9 shows, the algorithm for splitting a node is slightly more complicated. Instead of actually splitting a node, the entries may instead be *shared* with the *neighbor* of the node. The neighbor is the next node in the sequence of sibling nodes sorted by Hilbert values. When the current node is the last in the sequence, the previous node may be selected instead. It should be noted that implementations may also support larger neighborhoods, but this is not considered here.

The sharing itself is done by sorting all entries and partitioning the sorted sequence such that each node gets an equal share as far as possible. An algorithm for this is displayed in Algorithm 4.10.

When all nodes in the neighborhood are full, an actual split occurs. This is simply done by creating a new node, and sharing entries between the current node, its neighbor and the new node.

As is easy to imagine, the sorting described in these algorithms are often simplified by keeping the nodes themselves sorted. This reduces the `Choose-`

```
Function SplitNode(N)
    P ← parent of N;
    F ← {E ∈ P | H_d(E) > H_d(N)};
    B ← {E ∈ P | H_d(E) < H_d(N)};
    S ← (argmin_{E∈F} H_d(E)) ∪ (argmax_{E∈B} H_d(E));
    if S ≠ ∅ then
        N′ ← argmax_{E∈S} H_d(E);
        if |N′| < M then
            (N_2, N′_2) ← Share(N, N′);
            replace N and N′ with N_2 and N′_2 in the tree;
            return (N, ∅);
        else
            (N_2, N′_2, N″_2) ← Share(N, N′, ∅);
            replace N and N′ with N_2 and N′_2 in the tree;
            return (N_2, N″_2);
        end
    end
    return Share(N, ∅);
end
```

Algorithm 4.9: The `SplitNode` algorithm used by the Hilbert R-tree. Algorithm 4.10 displays the `Share` method.

```
Function Share(N_1, ..., N_n)
    ℰ ← ⋃_{i∈{1,...,d}};
    m(i) = { ⌊|ℰ|/n⌋ + 1   if |ℰ| mod n ≤ i
           { ⌊|ℰ|/n⌋       otherwise          ;
    for i ∈ {1, ..., n} do
        N'_i ← ∅;
        for j ∈ {1, ..., m(i)} do
            E ← E ∈ argmin_{E'∈ℰ} H_d(E');
            N'_i ← N_i ∪ {E};
            ℰ ← ℰ \ {E};
        end
    end
    return N'_1, ..., N'_n;
end
```

Algorithm 4.10: The `Share` algorithm used by the Hilbert R-tree.

`Subtree` algorithm to a binary search and eradicates the sorting from the `SplitNode` algorithm as the order is known.

Chapter 5

Optimizations

This chapter introduces possible optimizations for search in memory resident R-trees. All suggested improvements and whether they are expected to improve memory or compute performance can be found in Table 5.1, even though this distinction is not entirely clear cut.

This chapter starts off with the arrays of fields layout in Section 5.1, moves on to SIMD node scans in Section 5.2 and full node scans in Section 5.3, before the pruning node scan is introduced in Section 5.4.

5.1 Arrays of Fields Layout

Due to cache lines being much smaller than disk pages, the data layout of a node in the R-tree must be considered at a more fine grained level than the node level used for disk based trees. Mapping the concepts from the original R-tree [14] directly into memory results in nodes consisting of entries, where

Name	Memory	Compute
Array of Fields	x	
SIMD Node Scan	x	x
Full Node Scan	x	
Pruning scan	x	x

Table 5.1: Summary of algorithms suggested in this chapter and whether they improve memory performance, compute performance or both.

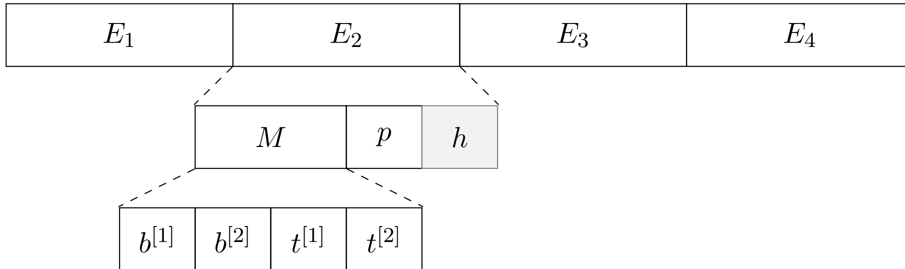


Figure 5.1: Naive layout of a node with the entries $E_1, \ldots, E_4$, with the layout of each entry and MBB below the node itself, where $M = (b, t)$ is assumed to be of 2 dimensions. The value h is only present in some trees.

each entry consists of an MBB and a pointer, as shown in Figure 5.1.

For other versions of the R-tree, extra fields may be included in each entry, as is the case for the Hilbert R-tree, where a Hilbert value is stored in each entry. Alternatively, an extra field may be included in the node, as is the case for the R*-tree, where the MBB of the node during construction must be stored.

The drawback of the naive layout from Figure 5.1 is however that several fields are likely to share cache lines. Since the smallest unit loaded from memory is a cache line, unused fields are likely to be loaded from cache just because they share cache line with a field that is used.

As an example, consider a search through a Hilbert R-tree, where the Hilbert value is included in every entry. Assume the Hilbert value resides in the same cache line as the MBB. Now, the Hilbert value must be loaded from memory when the MBB is required, regardless of whether it is needed or not.

This makes no difference for a disk based R-tree as the bottleneck is usually the disk access itself. Should the fields not share a disk page, several disk accesses must be made to access all fields. Due to the smaller size of cache lines, several cache lines will have to be loaded in most cases anyway.

One way to reduce the number of cache lines accessed is to rearrange the data into *arrays of fields*, as can be seen in Figure 5.2. This strategy places each field of an entry into a separate array in the node. Since adjacent fields now are of the same type, they are more likely to be required simultaneously, thus this may mitigate the effect of cache line sharing.

This idea is not a new idea, and has indeed been tried by Hwang et al.

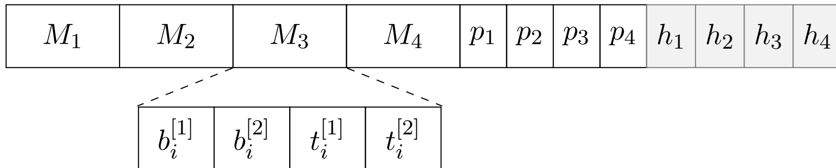


Figure 5.2: Arrays of fields layout for node where $M_i = (b_i, t_i)$, p_i and h_i is the MBB, pointer and extra value for entry i respectively.

during their evaluation of memory resident R-trees [19]. The general strategy is also presented in Intel's optimization reference [21] under the name `STRUCT_OF_ARRAY`.

5.2 SIMD Node Scans

When data is loaded sequentially from memory, it is likely the prefetcher kicks in to hide memory latency, and thus the available computational resources may become a bottleneck. Since a search through the R-tree consists mainly of node scans with sequential memory access, optimizing this operation is a good place to start reducing the computational cost, in accordance with Amdahl's law.

Since a node scan consists of the same operations applied to several entries, this is a perfect opportunity for parallelism using a SIMD approach. Good candidates for exploitation of the available data level parallelism are the SIMD instructions. These can be used to, for example, compare several coordinates simultaneously.

SIMD instructions can be used with several storage layouts, but some are better suited than others. For example would the naive layout displayed in Figure 5.1 be a poor choice, especially if $2d$ is not be a multiple of w, in which case some lanes will go unused, or more complex logic must be implemented to correctly handle all possible offsets. Both cases end up wasting computational resources, either through unused lanes or extra control logic.

A better alternative is to use *SIMD blocking*, where a group of w objects are combined into a single block, for which an arrays of fields layout is used. As shown in Figure 5.3, this results in a node where the corresponding fields can easily be loaded into a single SIMD register. In many ways, this can be

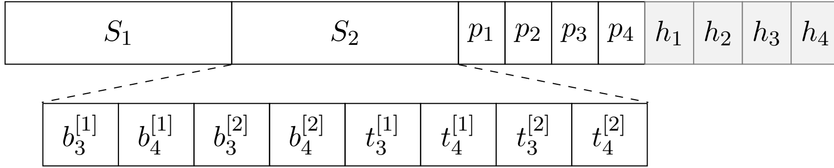


Figure 5.3: Node with SIMD blocking for $w = 2$ lanes, where S_1 and S_2 are SIMD blocks containing $M_1 = (b_1, t_1), \ldots, M_4 = (b_4, t_4)$. Note that corresponding coordinates are adjacent and can be loaded simultaneously into a SIMD register with w lanes.

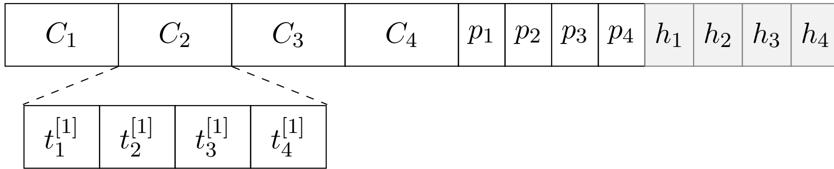


Figure 5.4: Node layout suitable for full node scans. Each $C_1, \ldots, C_4$ contains the lower and upper coordinates for the first and second dimension.

viewed as grouping w objects and using an arrays of fields layout recursively.

The drawback of SIMD blocking is generally that the algorithm needs to handle w objects simultaneously, and that arranging the fields in this way requires slightly more logic during construction. However, since w is usually constant, this is often quite simple.

In addition to increasing the computational performance of the search by increasing parallelism, SIMD instructions may also improve memory performance as more data is requested in one go using the specialized SIMD load instruction.

5.3 Full Node Scans

When searching through a tree, a series of node scans is performed, as explained in Section 3.3.1. As one node scan may be paused in order to explore a node further down the tree, this reassembles a depth first traversal through the tree of nodes overlapping the query box.

This approach makes sense when a scan can continue where it left off with negligible cost, as is the case for a disk based R-tree, where the nodes

in which a search is currently in progress are stored in memory. Pausing a search will therefore not incur an extra disk access.

For memory resident R-trees however, the search can benefit from scanning an entire node till the end before stopping. This reduces the number of hard to predict memory accesses that may stop the hardware prefetcher from successfully fetching the required cache lines.

The drawback of this approach is the need to store the results of each scan at the current path down the tree. Using bit sets for each level of the tree, this totals to a maximum of Mh bits, where h is the height of the tree and M is the maximal node size.

Since scanning the entire node in one go means all coordinates are required at the same time, the logic can be simplified by collecting all corresponding coordinates in separate arrays, as is illustrated in Figure 5.4. This is similar to SIMD blocking using $w = M$, and requires loading each coordinate of the query rectangle only once.

This can be combined with SIMD instructions to speed up the scan of each coordinate list.

5.4 Pruning Node Scans

During a search through an R-trees structure, an intersection test is used to prune away entire sub trees where such sub trees can be determined not to overlap the query box. This scenario is illustrated in Figure 5.5a and referred to as a *no overlap* scenario.

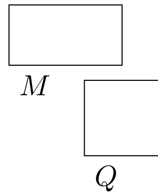
More theoretically, assume a node N, and let $E_i \in N$ be the entries of N for $i \in \{1, \dots, |N|\}$. Due to the structure of the R-tree, and by the definition of an MBB, it is already known that $\text{MBB}(E_i) \subseteq \text{MBB}(N)$.

Assuming that Q is disjoint from M, which is the case for the no overlap scenario, gives

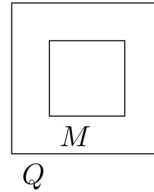
$$\text{MBB}(N) \cap Q = \emptyset \;\wedge\; \text{MBB}(E_i) \subseteq \text{MBB}(N) \;\Rightarrow\; \text{MBB}(E_i) \cap Q = \emptyset$$

which can be used recursively through the sub tree below N. Thus the sub tree below N can be pruned away in a no overlap scenario where $\text{MBB}(N) \cap Q = \emptyset$.

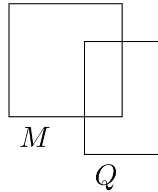
Less utilized however, is the fact that the entire sub tree below N can be assumed to intersect the query given that M is contained within Q, as



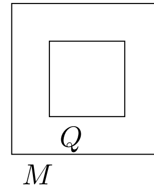
(a) No overlap. The sub tree below N can be pruned.



(b) MBB inclusion. All leafs in the sub tree below M overlaps with Q.



(c) Partial overlap. The child MBBs of N must be partially checked.



(d) Query inclusion. All child MBBs of N must be checked.

Figure 5.5: Four possible scenarios for the MBB M of a node N and a query box Q and how this affects pruning.

is the case in Figure 5.5b. In mathematical terms, this is similar to the no overlap scenario, and can be expressed as

$$\text{MBB}(N) \subseteq Q \;\wedge\; \text{MBB}(E_i) \subseteq \text{MBB}(N) \;\Rightarrow\; \text{MBB}(E) \subseteq Q$$

which will be referred to as a *MBB inclusion* scenario.

Another, perhaps less obvious situation is a *partial overlap* scenario, as seen in Figure 5.5c, where the Q overlaps M, but neither is included in the other. In contrast to the two previous cases where all children of N could be determined to be within or outside Q, this scenario does not allow any assumption as to which children are included or excluded.

What it does allow, is an assumption on some of the coordinates of the children of N. For a more useful description, some definitions must be in place. First, note that the overlap condition between two MBBs $A = (b_a, t_a)$ and $B = (b_b, t_b)$ can be expressed as

$$A \cap B \neq \emptyset \;\Leftrightarrow\; \forall i \in \{1, \ldots, d\} : b_a^{[i]} \leq t_b^{[i]} \;\wedge\; t_b^{[i]} \geq b_a^{[i]}$$

Now assume $b_q^{[i]} \leq b_m^{[i]}$ for some dimension i, where $Q = (b_q, t_q)$ and $\text{MBB}(N) = (b_m, t_m)$, as is the case in Figure 5.5c. Let $\text{MBB}(E_i) = (b_e, t_e)$ where E_i is a child of N as before. Since $\text{MBB}(E_i) \subseteq \text{MBB}(N)$, the following conclusion can now be drawn

$$b_q^{[i]} \leq b_m^{[i]} \;\wedge\; b_m^{[i]} \leq b_e^{[i]} \;\;\Rightarrow\;\; b_q^{[i]} \leq b_e$$

where the right side turns out to be one of the comparisons needed to do the intersection test above.

In other words, simply comparing the coordinates of Q with the corresponding coordinate of $\text{MBB}(N)$ may make it unnecessary to check any of the corresponding coordinates in the sub tree below N. This will be referred to as *partial pruning* since an entire set of coordinates can be pruned for the remaining search through the sub tree.

Note that this degenerates to the MBB inclusion scenario when all coordinates of Q are outside M, in which case none of the coordinates need be checked. In other words, implementing partial pruning also handles MBB inclusion.

In order to make this strategy effective, a memory layout where it is possible to skip the coordinates that need not be checked is desirable. This is a good match for the node layout used for full node scans, as seen in Figure 5.4. In addition, it can be easily combined with both full node scans from Section 5.3 and SIMD instructions as used in Section 5.2.

Chapter 6

Methodology

Evaluation and analysis of the proposed optimizations for search in the R-tree is done by measuring the run time as the short cut of counting disk accesses is no longer possible for memory resident R-trees. The tree itself is constructed using the insert algorithm for the revised R*-tree.

More details are given in this chapter, starting with the data and query sets in Section 6.1. The strategy for measuring run time is presented in Section 6.2, before the tools used for analysis is introduced in Section 6.3. The actual implementation is briefly described in Section 6.4 before the details of the environment in which the run time was measured is presented in Section 6.5. Finally, Section 6.6 describes how the parameters used were selected.

6.1 Data and Query Sets

The data sets used were provided by Beckmann and Seeger and are the sets they use when they evaluate the revised R*-tree [4]. They also provided a more detailed description [3]. In summary, there are 28 data sets, of which 21 are artificially generated and the remaining are data sets form real applications.

Three query sets have been derived from each data set. The first gives result sets of around 1 result, the second yields around 100 results and the thirds give 1000 results. Because the query cost tends to increase with result set size, there are fewer queries in the query sets yielding a large number of results.

Name	Type	Peculiarities	Order
`abs02`	rectangles	overlap	Row order
`bit02`	points	none	Random
`rea02`	mix	overlap duplicates	Ordered sub-regions inserted in random order
`abs03`	rectangles	overlap	Row order
`uni03`	points	none	Random

Table 6.1: The set of 2 and 3 dimensional data sets used for testing and their key properties.

To identify the data sets, a three letter code combined with the zero padded dimension is used, for example `abs02`. The query sets will mostly be referred to in the context of a data set, and thus it is sufficient to mention its size only. Be aware that the size refers to the number of results, not the number of queries.

Since this thesis focuses on search in 2 and 3 dimensions, data sets of higher dimensions are ignored. Furthermore, to avoid over fitting the solutions to the data, the suggested improvements for memory resident R-trees are tested on only a subset the data sets.

As can be seen in Table 6.1, the data sets have been selected to be representative for the entire collection, both with respect to order, type and peculiarities, while at the same time leaving a set of distributions untouched. The remaining data sets will be used for a final comparison between the chosen improvements and the baselines.

6.2 Run Time Measurement

The query performance of the index is measured as the run time of a given query set. The index is first constructed from the data set, then the query set is loaded into memory and finally the wall clock time required to generate the results is measured and reported as the run time. An overview of the evaluation process can be found in Algorithm 6.1.

Since data already present in cache may have an impact on the run time, the cache is cleared using the `CLFLUSH` instruction available on the x86 architecture. This instruction flushes and invalidates a cache line, and is used repeatedly to flush all cache lines in the memory regions mapped

```
Function Benchmark(I, 𝒬)
    R ← ∅;
    for i ∈ {1, ..., r} do
        L ← ∅;
        for j ∈ {1, ..., 10} do
            invalidate cache;
            r ← execution time of Q using I;
            L ← L ∪ {r};
            shuffle 𝒬;
        end
        R ← R ∪ { (1/10) ∑_{r∈L} r };
    end
    return min_{r∈R}(r);
end
```

Algorithm 6.1: The process performed when evaluating an index I given a query set $\mathcal{Q}$ using r runs. Note that I should already be filled with data.

into the process' virtual memory space.

Unfortunately, since all queries in the query set are executed in rapid succession without any cache clears in between, the order of the queries may shift the results in positive or negative direction for some query sets. To mitigate this effect, the run time for each query set is measured 10 times, in between which the query sets are shuffled using the standard template library function `std::shuffle`. To ensure reproducible results, the random number generator used is initialized with a constant seed.

Each execution of the query set with the associated cache invalidation and query shuffling will be referred to as a *run*. The run time of a run is the average time required to generate result sets for the entire query set.

Even though the evaluation is done on a quiet system, there may be occasional disturbances causing the run times to vary slightly between each run. Therefore, the runs are repeated several times and the minimum among the averages is reported.

For the first query set, the number of repetitions is 60, but for the following it is only 10. The number of repetitions for the first query set

is higher to allow the branch predictor to record the history of branches and thus predict the branches more accurately during measurement. The results of the first runs could have been discarded, but since the minimum is selected, leaving them can only make the result more accurate.

6.3 Hardware Counters and Instrumentation

To provide a more in depth analysis of the performance observed, some extra information may be gathered using the performance counters present in the processor. These counters can be set up to register a set of predefined events and do not affect the performance. Examples of such events include cache misses, memory accesses, stalls and the number of instructions retired.

Unfortunately, only 4 such counters exist, and thus only 4 events can be recorded at any one time. To provide more statistics, each set of runs is repeated several times, each time with a different set of events. Since no two runs are exactly equal, two event counts from different runs may not correspond exactly. To increase the chance of comparable numbers, the value from the run with the minimum run time within one set of runs is reported.

In some cases, these hardware counters may also be used to provide more detailed information about what lines of code triggers certain events through sampling. This is done using the `perf` Linux tool, which sets up an event counter for the desired event. Each time the event counter overflows, an interrupt occurs and the operating system records the instruction pointer and lets the program continue. Finally, the `perf` tool reports the frequency of interrupts at each assembly instruction.

These numbers are unfortunately not always entirely exact. First of all, the use of sampling means a lot of events are disregarded. Second, the instruction pointer may be at a different instruction than the source of the event due to pipelining. In addition, it is sometimes hard to map the assembly code back to the source, even with source annotation, due to optimizations performed by the compiler.

Events related to the higher level behavior of the code, such as the number of nodes accessed, can obviously not be captured using hardware events. These are instead collected using instrumentation. Since the collection has an impact on the run time and performance counters, this is never done during run time measurements. In other words, the statistics are collected

during separate runs.

6.4 Implementation

All indexes have been implemented from scratch, using the original articles as reference for the existing variants of the R-tree. The code is written in C++11 with heavy use of templates, which allows the compiler to do several optimizations not otherwise possible, such as loop unrolling when looping through all dimensions. Each index is, together with its parameters, compiled in to a separate library with the highest level of compiler optimizations enabled.

A common executable is responsible for loading the dynamic library, reading data sets and query sets from files and passing them on to the index in a suitable format. Since some measurements may affect each other, command line parameters are used to specify what should be measured.

For R-trees, the search algorithm itself has been separated from the actual construction of the tree, such that no R-tree variant gains the advantage of a better search algorithm. Parts of the search algorithm are however specialized for some of the improvements suggested in Chapter 5. For a more complete overview of the implementation, see Appendix A.

Since the Hilbert R-tree requires the bounds of the data domain to correctly encode coordinates, this is calculated from the data set and passed to the index at construction time. The Hilbert values themselves are limited to 64 bits.

6.5 Environment

Run time measurements were done on a system with two Intel Xeon E5-2683 processors with 128 GB and 64 GB of memory. The maximum memory bandwidth is 68 GB s^{-1} [20] and each processor has 14 cores with 2 hardware threads each, a split L1 cache with a total of 64KB equally distributed between instructions and data, and a 256KB L2 cache shared with one other core on the same chip. All cores on a chip shares a 35MB L3 cache, and all cache lines are 64 bytes.

To increase the chance of reproducible results, only one of the available cores is used. Since the cores share many of the same resources, running several experiments simultaneously may render the results hard to predict

Parameter	R-tree variant	Description
M	All	Maximal number of entries in each node
m	All	Minimum entries in each node
p	R*-tree	Number of nodes to reinsert on overflow

Table 6.2: Parameters that must be determined for different R-tree variants. Note that m is essentially determined for the Hilbert R-tree as a part of the split strategy.

and reproduce. To stop the operating system from migrating the process between cores, which may results in varying run times due to cache effects, the process is also pinned to a specific core and the memory allocated on the corresponding node using the Linux `numactl` command.

For compilation, the GNU Compiler Collection (GCC) version 6.2.0 is used. The operating system running beneath was Ubuntu 14.04.5 LTS.

6.6 Parameter Optimization

The R-trees evaluated herein requires a set of parameters as listed in Table 6.2. Since the parameters influence the performance of the indexes, and because the optimal set of parameters may vary between the indexes, an optimal set of parameters must be devised for each index individually. Due to the large search space, finding the optimal parameters by enumerating all possibilities is intractable in the scope of this project.

First, the parameter p is fixed to $\lfloor M/3 \rfloor$, which is approximately 30 % as suggested by Beckmann and Seeger [2]. *Simulated annealing* [11], a probabilistic global optimization technique, is then employed to find values for M and m that can be used during evaluation.

As can be seen in Algorithm 6.2, the optimization algorithm starts out with an initial set of parameters P_0. During iteration i, a candidate solution C is selected randomly from the neighbors of the current solution. The candidate solution is used as the solution P_i for the current iteration if it is better than the previous solution P_{i-1}, or otherwise according to some probability that decreases with time. This is repeated until the current solution has not been updated for 20 subsequent iterations.

Due to the decreasing probability of selecting a solution that is worse than the current one, the algorithm explores the solution space during the

first iterations, but acts more like a hill climbing algorithm near the end.

Function `Anneal`(P_0)
- $i \leftarrow 1,\ j \leftarrow 0$;
- $T = 1$;
- **while** $j < 20$ **do**
  - $C \leftarrow$ `SelectNeighbor`(*solution*);
  - **if** $S(C) < S(P_{i-1}) \lor$ `Random()` $< \frac{1}{T(i)} \frac{S(P_{i-1}) - S(C)}{S(P_0)}$ **then**
    - $P_i \leftarrow C$;
    - $j \leftarrow 0$;
  - **else**
    - $P_i \leftarrow P_{i-1}$;
    - $j \leftarrow j + 1$;
  - **end**
  - $T \leftarrow f(T)$;
  - $i \leftarrow i + 1$;
- **end**
- **return** $\operatorname*{argmin}_{k \in \{0,\ldots,i\}} S(P_k)$

end

Algorithm 6.2: The basic simulated annealing algorithm given an initial set of parameters P_0. $S(P)$ calculates the run time using the set of parameters P, and `Random` returns a random number $r \in [0, 1\rangle$, drawn from a uniform distribution.

In the theoretical version of simulated annealing, the current solution would be returned. Many implementations, including this one, do however return the best solution visited during the course of the algorithm. This increases the chance of finding a good solution with only a negligible increase in cost.

When searching for parameters, the neighbor selection function `Select-Neighbor` simply draws each parameter of the new solution from a Gaussian distribution centered on the previous value for the same parameter, rounded to the closest integer. The standard deviation of the distribution is set to half the value of the parameter from the initial solution. Should the selected solution be invalid (e.g. $M < 2$), a new set of parameters is chosen until a

valid solution is found.

Assuming the parameters are bounded and given enough iterations, the algorithm can be shown to converge to the globally optimal solution [11]. The number of iterations required depends on the function to be optimized. If deep local minima exist, more iterations are required. Thus, assuming the run time varies smoothly with the parameters, this should not pose a problem.

Chapter 7

Experimental Results

This chapter presents the results of the experiments and starts off with the reproduction of earlier results in Section 7.1, before investigating the run time distribution to establish the error in run time measurements in Section 7.2. The impact of the parameters used is explored in Section 7.3. Finally, the behavior of a naive R-tree implementation is examined in Section 7.4 before the results from applying the optimizations suggested in this thesis are presented in Section 7.5.

7.1 Reproduction of Results

To validate the indexes, and provide a humble review of the work by Beckmann and Seeger [4], their results have been attempted reproduced using the original data sets, query sets and parameters.

First of all, it should be pointed out that since the data and query sets are the exact same used by Beckmann and Seeger, and because the results should be independent of the machine on which the tests are run, the results should be possible to reproduce exactly.

With that said, since most of the algorithms relay on floating point arithmetic, the order of calculation may have an impact on the result. This is however too specific to be included in the original papers. If the algorithm is sound, however, this should not make a large difference.

The average number of leaf accesses performed by the different indexes relative to Beckmann and Seeger's numbers can be found in Table 7.1. Judging from the averages, the results for the revised R*-tree have been repro-

Index	2–3D	2–9D	2–16D	2–26D
R*-tree	0.93	0.72	0.64	0.45
Quadratic R-tree	0.75	0.75	0.75	0.40
Revised R*-tree	1.00	0.98	0.99	0.98
Hilbert R-tree	0.91	0.97	0.96	0.59

Table 7.1: The average number of leaf accesses performed by the indexes over all data and query sets, the number of leafs and the number of splits using perimeter. All numbers are relative to the implementations by Beckmann and Seeger [4].

duced almost exactly. This is not so for the other indexes, which performed better than found by Beckmann and Seeger.

For the R*-tree, the description given in their later article [4] seems to be slightly different from the original [2] when it comes to tie resolution. The original version was followed in this case.

In addition, the cost limitation for the `ChooseSubtree` method has not been implemented, thus the better numbers may be due to more resources being spent during insertion. According to the original article, this should not make a large difference.

For the Hilbert R-tree, the original papers [4, 24] are not very precise when it comes to the algorithm. For example does none of them describe the Hilbert encoding used. Neither is there any explicit mention on whether sharing is actually implemented, and in that case how the neighborhood is chosen. It is thus expected that the numbers are slightly different.

The algorithm for the quadratic R-tree is however quite concise, and it is therefore slightly surprising that the numbers are off by such a large factor.

One interesting observation is how much better all indexes with the exception of the revised R*-tree do on average for dimensions 2 to 26, meaning that they do significantly better than expected in dimensions 16 to 26. Even so, the revised R*-tree is still way ahead of the others. The quadratic R-tree, Hilbert R-tree and R*-tree do approximately 3.4, 2.8 and 1.7 times as many leaf accesses on average across all dimensions.

A more detailed table with leaf accesses for the revised R*-tree each data and query set combination can be found in Table 7.2, also this time relative to the original numbers. This reveals that most of the data and

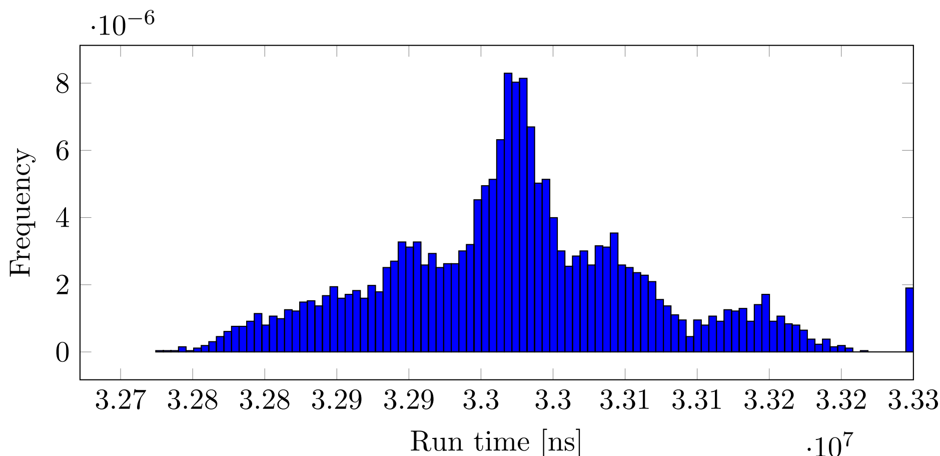


Figure 7.1: Run time histogram for 5000 runs. Around 1 % of the samples are located above the selected range and included in the last bar.

query sets yield very similar results, with the exception of a few data sets.

The discrepancies may be due to floating point arithmetic, as discussed earlier, or perhaps because the algorithm has not been implemented exactly as specified. As Table 7.1 show, the average is still lower than the original numbers.

7.2 Run Time Probability Distribution

The run time is expected to vary slightly between each run due to external events, such as context switches or translation look aside buffer flushes performed by the operating system, or cache interaction with other processes running on the system. This section therefore examines the probability distribution of the run times, which makes it possible to later determine whether differences in run times should be deemed significant or not.

To get a picture of the probability distribution, the benchmark used to determine the run time for the `abs02` data set has been run 5000 times in a row using the query set with a medium size result set. This gives the distribution in Figure 7.1. Note that the index structure was constructed only once, after which the 5000 runs followed.

The histogram acquired from the run times can be found in Figure 7.1. A small number of runs required significantly longer time than the other to

benchmark	Q0	Q2	Q3	Leafs	Perim. splits
abs02	1.00	1.00	0.99	0.99	1.00
abs03	1.00	1.01	1.01	1.01	1.00
abs09	1.00	0.99	0.99	1.00	1.00
bit02	0.96	0.99	0.99	1.00	1.00
bit03	1.00	1.01	1.00	1.00	1.00
bit09	1.00	0.97	0.99	1.00	1.00
dia02	1.00	1.00	1.01	1.00	1.00
dia03	1.00	1.00	1.00	1.00	1.00
dia09	1.00	1.00	1.00	1.00	1.00
par02	1.03	1.02	1.00	1.00	1.00
par03	1.01	1.00	1.00	1.00	1.00
par09	0.93	0.95	0.98	1.01	1.00
ped02	0.98	1.00	1.00	1.00	1.00
ped03	0.98	0.99	1.00	1.00	1.00
ped09	0.61	0.63	0.67	0.99	1.00
pha02	1.00	1.00	1.00	1.00	1.00
pha03	1.00	1.00	1.00	1.00	1.00
pha09	1.13	1.00	1.00	1.00	1.00
uni02	1.00	1.00	1.00	1.00	1.00
uni03	1.02	1.00	1.01	1.00	1.00
uni09	0.98	1.01	1.00	1.00	1.00
rea02	1.01	1.00	0.99	0.98	1.00
rea03	1.05	1.02	1.01	1.00	—
rea05	1.08	1.04	1.03	1.00	0.90
rea09	1.28	1.06	1.04	0.99	1.00
rea16	1.05	0.99	0.99	1.00	1.12
rea22	1.01	0.96	0.98	1.01	1.00
rea26	0.83	0.94	0.98	1.01	0.99

Table 7.2: The average number of leaf accesses performed by the indexes for all data and query sets, relative to the numbers given by Beckmann and Seeger [4]. The two last columns show the relative number of leafs in the tree and relative number of perimeter splits done during construction.

complete and have been collected in the last group of the histogram. As is visible, the run time distribution seems to be multi modal. A possible explanation for this may be that certain events occur in some percentage of the runs, always adding an approximately constant amount to the run time.

One takeaway from the histogram is that the 99 percentile lies within a distance of 2 % of the minimum value, which is very acceptable for later evaluation of the indexes. This is of course under the assumption that the remaining results follow the same distribution.

A plot of the run times over time was also made, and the resulting plot for the first 500 runs can be seen in Figure 7.2a. One would perhaps expect each run to be independent of the others, but that seems not to be the case. In fact, the run times seems rather unstable for the first 50 runs. To explain this phenomenon, hardware counters were enabled and the data collected is shown in the remaining sub figures of Figure 7.2.

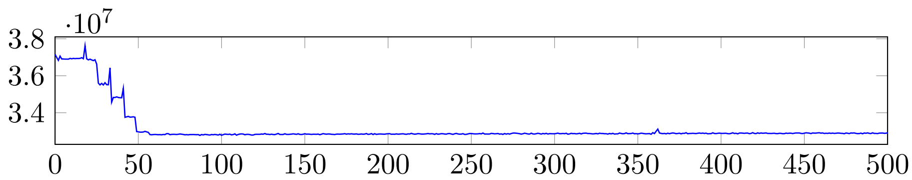
First of all, it may be tempting to explain the dropping run time from Figure 7.2a as a cache effect, especially when comparing he run times with the number of number of last level cache misses as visible in Figure 7.2b. In other words, it may seem like the cache clearing code is not doing its job, and thus the code performs better after some iterations as the most used memory references are served from cache.

This does however seem unlikely when taking the number of memory references arriving at the last level cache, which can be seen in Figure 7.2c, in to account. The plot seems to fit very well with the number of misses, meaning the increased number of cache misses is likely due to the increased number of references.

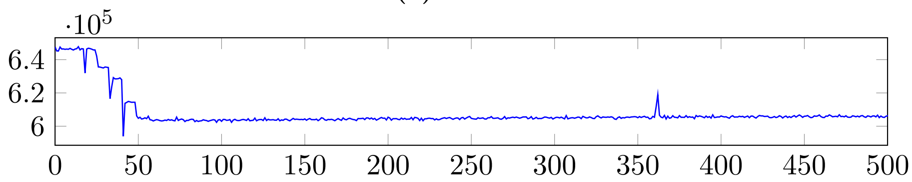
And indeed, the number of cache misses relative to the number of cache references is almost constant and always within 38 % to 40 %.

The large number of cache references during the first runs is accompanied by a large number of micro instruction issues, as seen in Figure 7.2d. Certainly, assuming that some of these micro instructions are memory references, this can explain both the increased number of last level cache references and misses.

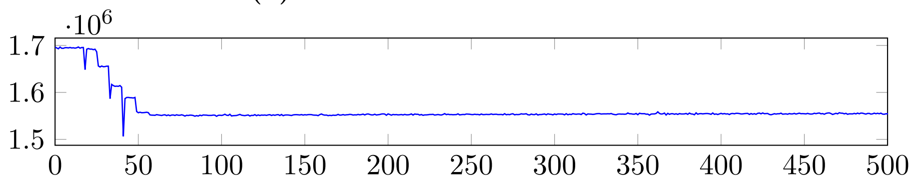
One could suspect that the varying number of micro instructions is due to varying amounts of work performed by the algorithm benchmarked, but measuring instruction retirement reveals a constant number of instructions being retired every run, which suggests the code is performing the same amount of work.



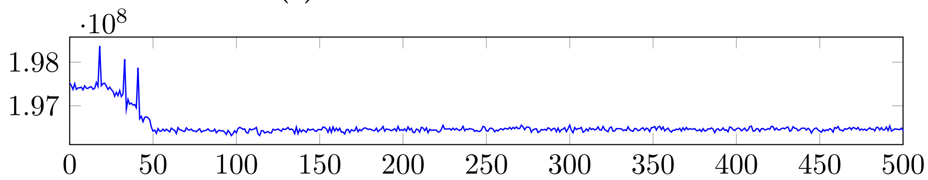
(a) Run time



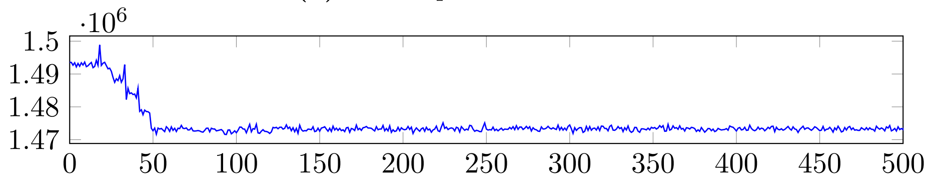
(b) Cache misses from last level cache



(c) References for last level cache



(d) Micro-operations executed



(e) Mispredicted branches executed

Figure 7.2: Plot of different statistics in the order they were collected. Note that only the first 500 runs are shown as the remaining ones are relatively uninteresting.

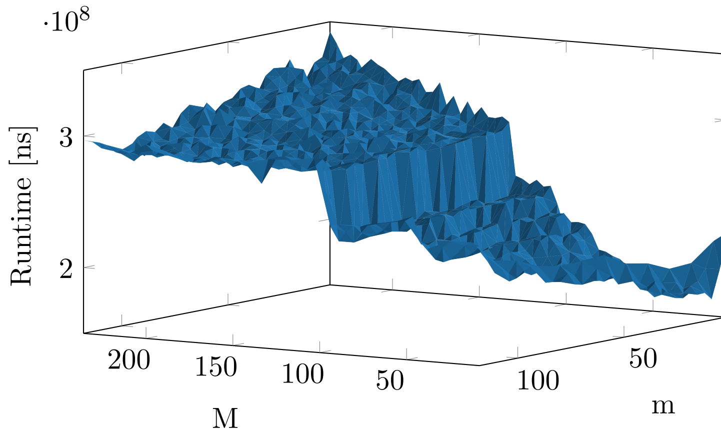


Figure 7.3: The total run time of all query sets on `abs02` as a function of M and m, sampled with a grid size of 4.

What may possibly explain the high run times during the first runs is perhaps speculation going wrong. As shown in Figure 7.2e, the number of times times the processor executes a branch speculatively the wrong way correlates well with the other measurements. This suggests that the branch predictor needs some time before it finds the best approach to predict the outcome of branches. When it does so, it does however seem to stick with the choice and thus the performance levels out after an initial warm up time.

7.3 Impact of Parameters

This section investigates the impact of the parameters on the performance of the memory resident revised R*-tree, and checks the assumptions made in Section 6.6.

The parameters m and M were chosen and the run time for the revised R*-tree sampled for the `abs02` data set, the total run time for all query sets can be found in Figure 7.3.

As seen, there is a cliff when the node size is increased beyond around 120–150 entries. A lower resolution plot of a larger domain shown in Figure 7.4 reveals that a similar cliff appears around node sizes of 1200–1400 entries.

These cliffs are most likely due to changes in the tree structure. Assume

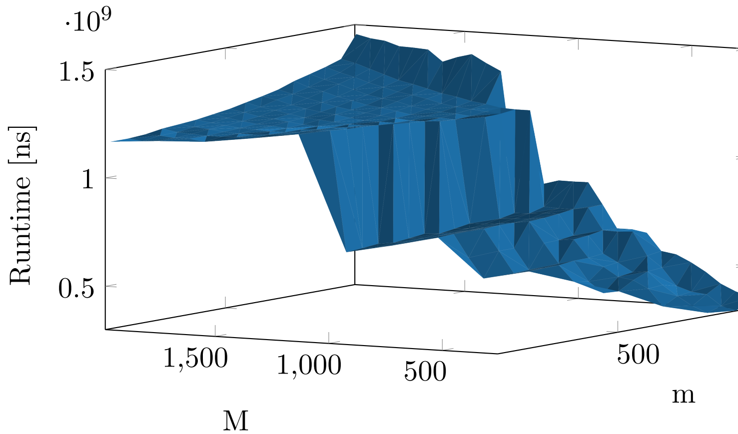


Figure 7.4: The total run time of all query sets on `abs02` as a function of M and m, sampled using a grid size of 64.

f is the average fill grade of a node in the tree. Let L_i be the number of nodes in the tree at depth i. Since the tree increases in height when the root node is split, the root must be full prior to a split and thus contain M entries. This gives

$$L_1 = M$$

Generally, each node at depth $i > 1$ in the tree contains fM entries, thus the number of nodes at the level below is given by

$$L_i = fML_{i-1},$$

for $i > 1$, and solving the recurrence relation for M yields

$$M = \sqrt[i]{f^{1-i}L_i}$$

Assuming a tree height of h, L_h is the number of data objects in the tree, M is given by

$$M_h = \sqrt[h]{f^{1-h}L_h}$$

which gives the values in Table 7.3, assuming $f = 0.6$ and $L_m = 1000000$, as is approximately the case for `abs02`. As can be seen, this matches very well for both cliffs identified in Figures 7.3 and 7.4. In addition, the lowest value recorded appears at $M = 48$, with $m = 4$, which is just around where the tree gains a level under the above assumptions.

h	M_h
2	1291
3	140
4	46
5	23
6	15

Table 7.3: The node sizes around which the height of the tree changes assuming a fill grade of $f = 0.6$ and $L_m = 1000000$ data objects.

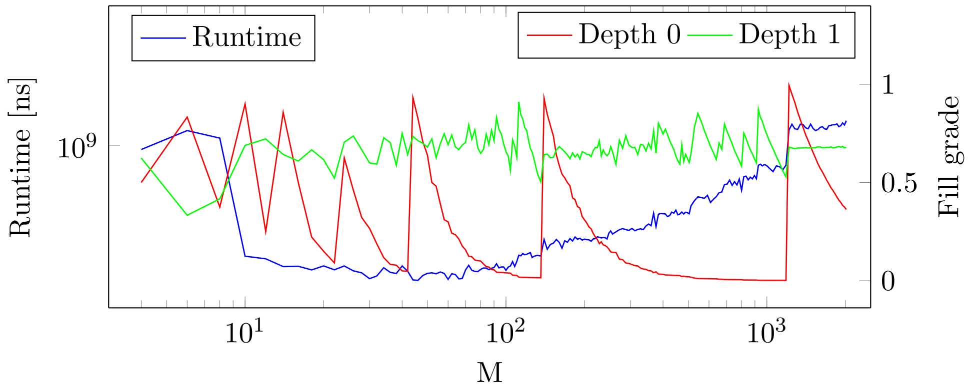


Figure 7.5: Total run time and average fill grade at depths 0 and 1 as a function of the node capacity M for the `abs02` data and query sets. The minimum node fill grade m is set to $\max\left(\left\lfloor \frac{M}{5} \right\rfloor, 1\right)$.

Plotting the actual run time and node fill grade at depths 0 and 1 yields the plot in Figure 7.5. Note that the fill grade for depth 0 is simply the fill grade of the root node, and thus jumps from almost 0 to 1 when the tree height decreases. This seems to correspond well with the jumps in run time for the two cases examined above.

As for the optimization algorithm, the run time function seems to have few deep local minima that are far from the true global minimum, which is consistent with the assumptions made at the end of Section 6.6. It should however be noted that this is only a case study for the revised R*-tree on the `abs02` data set.

During optimization, the initial values used as input for the simulated annealing algorithm were set to $M = 50$ and $m = 40\,\%$ based on the land-

Optimization	D	M	m	Iterations
None	2	35	20 %	104
	3	33	33 %	99
Arrays of fields	2	69	36 %	123
	3	37	23 %	138
SIMD	2	63	7 %	113
	3	63	24 %	152
Full Scan	2	68	7 %	150
	3	68	21 %	160
Pruning	2	76	21 %	100
	3	51	13 %	120

Table 7.4: Configuration parameter for the R*-tree for various optimizations found during search. The values for m are relative to M.

scapes presented in Section 7.3. The parameters found during parameter optimization and used throughout this chapter can be found in Table 7.4.

7.4 Behavior of the Naive Implementation

A quick look at the behavior of the naive implementation of the revised R*-tree running in memory is required to be able to interpret the optimization results. Figure 7.6 shows the values of some key metrics; the ratio of mispredicted branches that were executed to the number of retired branches, the number of last level cache misses relative to last level cache references, and the number of micro operations that never retired relative to the number of micro operations issued.

The number of mispredicted branches seems to decrease with the query set size. This is perhaps a bit surprising since the small query sets nearly only visits a single node at each level, meaning the outcome of the intersection test is highly predictable. For the larger query sets, more matches are expected, which results in a more even mix of taken and untaken branches, making them harder to predict.

A quick inspection of the performance profile using `perf` reveals that the large number of branch mispredictions seems to be related to the intersection

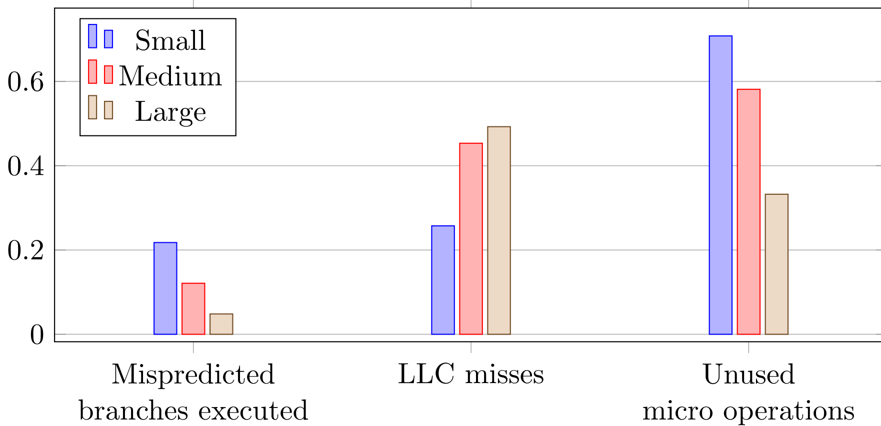


Figure 7.6: Some statistics for the naive memory resident revised R*-tree by query set size.

tests used to prune away sub trees.

For the number of unused micro operations, a similar trend can be found. These operations were executed speculatively, but the result was never used. As expected, these are tightly coupled with the number of mispredicted branches executed.

For the last level cache misses, the opposite trend can be observed. This can be explained by the more sequential memory access performed by the search for small query sets. Since this gives the prefetcher a chance to predict which cache lines are required next, the number of misses is lower.

When it comes to the throughput of instructions, a histogram of the number of retired instructions each cycle can be found in Figure 7.7. As can be seen, the larger query sets are better at retiring more instructions every cycle. This is also likely to be a consequence of better branch prediction as this allows the speculative operations to actually retire, thereby also retiring more instructions.

7.5 Results of Optimizations

In this section, the improvement in run time provided by each optimization suggested in Chapter 5 for memory resident R-trees will be presented. The arrays of fields layout is examined in Section 7.5.1, before the SIMD, full

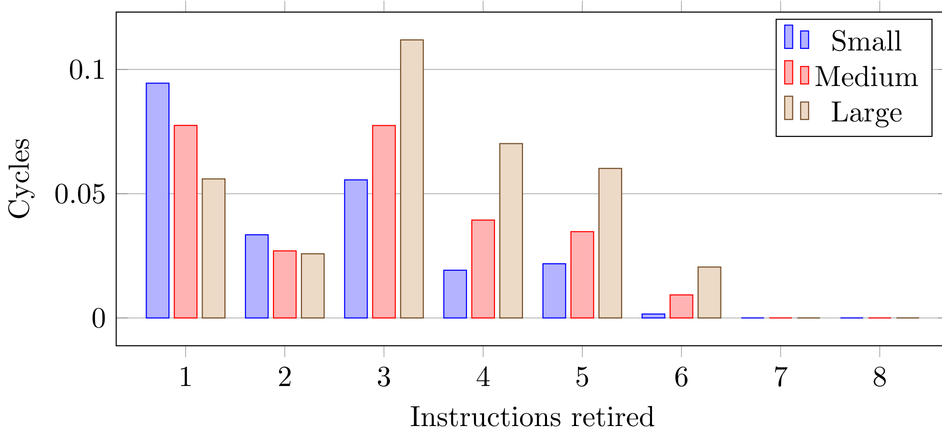


Figure 7.7: Histogram with the number of instructions retiring each cycle for different query set sizes. The cycle count is relative to the total number of cycles.

and pruning node scans are examined in Sections 7.5.2 to 7.5.4

7.5.1 Arrays of Fields

As can be seen from Figure 7.8, the array of fields approach actually made the results worse, especially for the small query set and for the 3 dimensional data sets.

One possible explanation for the poor performance may be that accessing a pointer when a match is found incurs a non-sequential memory access, which is likely to triggers a cache miss. The resulting stalls may be a possible explanation for the performance regression.

This is however not consistent with Figure 7.8 as the small query sets should be accessing fewer pointers due to the reduced result set, which means the penalty of non-sequential memory access should be larges for the largest query set. The figure shows that this is clearly not the result found.

In addition, the number of cache misses for the last level cache has actually been reduced by approximately 9 % for all query set sizes. This can easily be explained by the extra data not accessed, and shows that the arrays of fields optimizations achieves the goal of reducing memory access.

With fewer cache misses and less data loaded from memory, the reason for the poor performance of the arrays of fields layout is likely compute re-

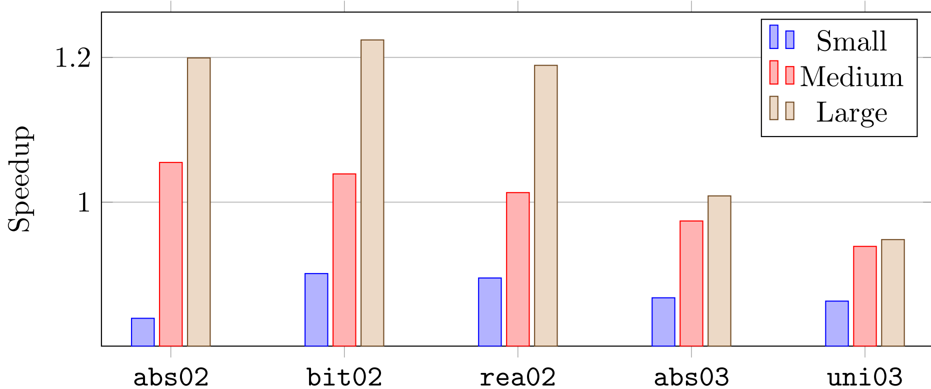


Figure 7.8: Speedup of the revised R*-tree with arrays of fields node layout by query set size.

lated. Indeed, the number of mispredicted branches executed speculatively is around 7 % higher for the small query sets. For the medium query sets, the number is about the same, and for the large query set, the number is about 4 % lower.

The increased number of branch mispredictions is not due to branches that are harder to predict, but rather because the total number of branches executed by the code is larger, especially for the small query sets. This may be due to different compiler optimizations being applied to the two versions of the code, causing more branch instructions to be executed, even though no branches have been introduced in the source code.

Together with the improved memory access pattern, the increased number of branches explains the results found in Figure 7.8, at least for the 2 dimensional data sets. For the small query sets, the increased number of branches with associated mispredictions outweighs the improved memory access. For the medium query sets, the increased number of mispredictions and improved memory access almost cancel each other out. Lastly, the improved performance for the large query sets is likely due to an improvement in both memory access and branch prediction.

For the 3 dimensional data sets, the portion of a node that consists of pointers is smaller since the extra dimension increases the size of an MBR by the size equivalent to 2 pointers. In other words, larger parts of the node has to be loaded anyway, reducing the effectiveness of separating out the pointers. This is confirmed by the numbers; where the 2 dimensional

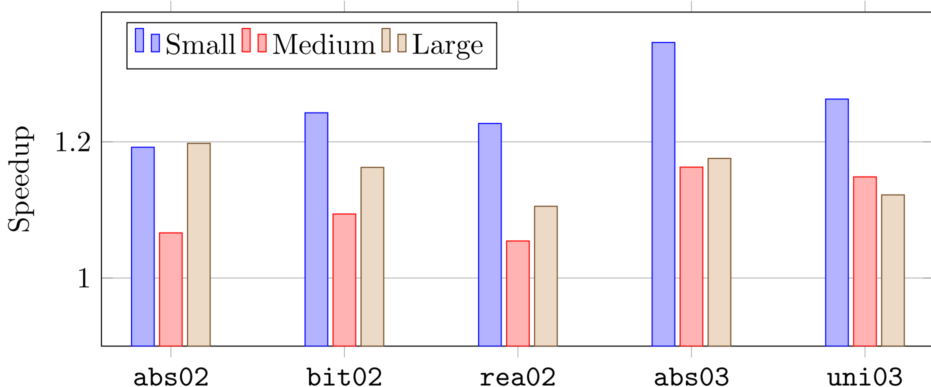


Figure 7.9: Speedup of the revised R*-tree with SIMD scans by query set size.

data sets experience a 17 % reduction of cache misses, the 3 dimensional sets experience a 3 % increase.

When it comes to the distribution of instructions completing each cycle, this is mostly the same as for the naive implementation, for which the histogram is displayed in Figure 7.7.

7.5.2 SIMD Scans

Applying the SIMD scan optimization from Section 5.2 yields the speedups in Figure 7.9. As can be seen, there is a speedup in all cases, but the speedup varies between data and query sets.

One interesting observation is the poor speedup of the medium sized query sets compared to both the small and large ones. The immediate cause of this behavior can be found in Figure 7.10, in which a histogram of the number of instructions retired in one cycle is shown. For 2 to 3 instructions per cycle, the small query sets seems to do slightly better than the larger ones, but for 4 to 6, the pattern from Figure 7.9 can be recognized.

Digging deeper, it turns out the number of last level cache references for the medium sized query sets does not see the same reduction as the for the other query sets. The small and large query results in a reduction of 17 % and 11 % respectively, while the corresponding number for the medium query set is 7 %.

It also seems that the extra cache references done for the medium query

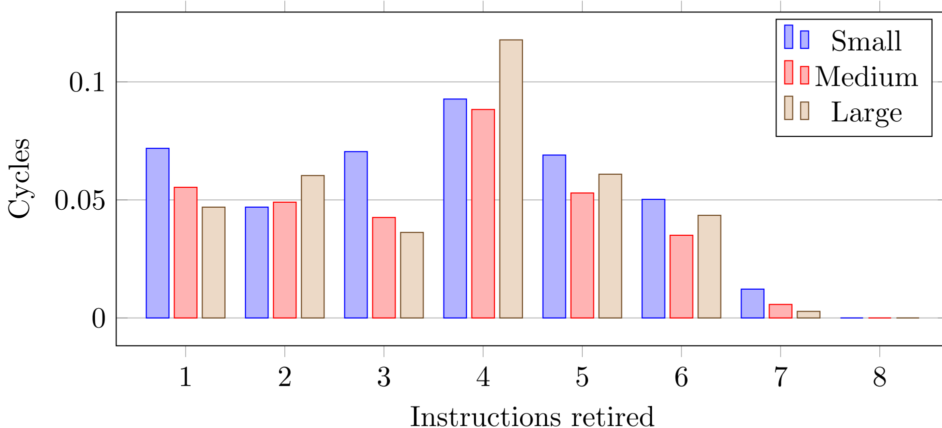


Figure 7.10: Histogram of instructions retired each cycle by query set size for the revised R*-tree with SIMD scans.

set are mostly handled by the last level cache as the reductions in cache misses are approximately the same for all query sets. As illustrated in Section 2.1.1, the delay for accessing last level cache is not insignificant, but far from the delay associated with main memory.

Looking at the number of mispredicted branches executed, the reduction is quite considerable for the small query sets at 83 %. This is accompanied by an increase of 42 % in the number of branches. In other words, the branch prediction is working flawlessly for the small query sets.

For the medium and large query sets, the number of branches is reduced by 10 % and 27 %, and even higher reductions is the number of branch mispredictions. These changes can probably be blamed on the SIMD blocking, which changes the code quite significantly, apparently favoring the large query sets.

As can be seen in Figure 7.10, the instruction level parallelism seems to have increased, especially for the small query sets. This contributes to a significant speedup despite a spectacular increase of 80 % increase in the number of micro operations retired.

7.5.3 Full Node Scans

The speedup using the full node scan optimization can be found in Figure 7.11. Unlike the speedup for SIMD scans, there seems to be a more

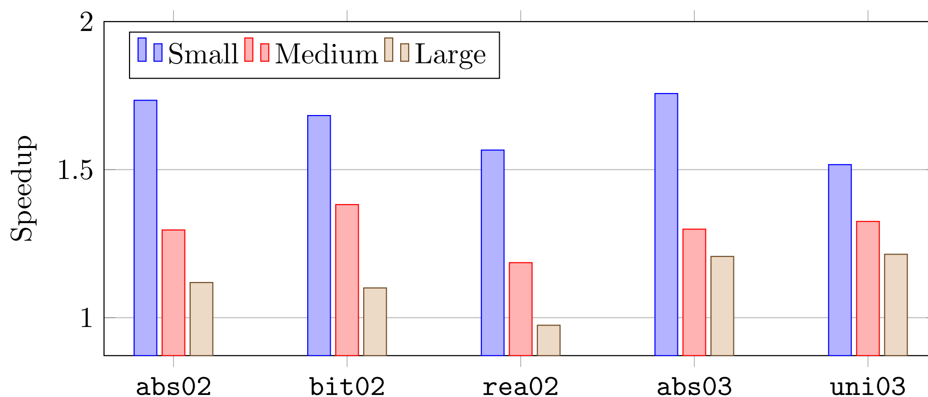


Figure 7.11: Speedup of the revised R*-tree with full node scans by query set size.

clear trend where large query sets are worse off. For the `rea02`, the speedup is even below 1. For small query sets, the results are even better than for the SIMD scan. This may be slightly unexpected since the full node scan was meant to solve the problem of interrupted node scans, a problem which should be more prominent for the large query sets than the smaller ones.

The original thought behind the full scan optimization was to reduce cache misses by optimizing the code for automatic prefetching through more sequential memory reads. This seems to have had the expected consequences; the cache misses have been reduced by 4 % and 12 % for the medium and large query sets, and stays about the same for the small query set, where the number of interrupts is low even for the naive implementation.

This is accompanied by an increase in the number of last level cache references by 25 % to 37 %. This may be because of the lacking temporal locality when iterating through the result set after a node scan. Since a new node scan may have evicted the result set from memory, the result set must be fetched from the large last level cache.

One less expected consequence is a large reduction in the number of mispredicted branches executed. This is especially so for the small query set, which experiences a reduction of 93 %, compared to 86 % and 78 % for the medium and large query sets. This can partly be explained by a lower number of branches being executed, but this is only half the story. The rest may be due to branches that are easier to predict, or that the branches that

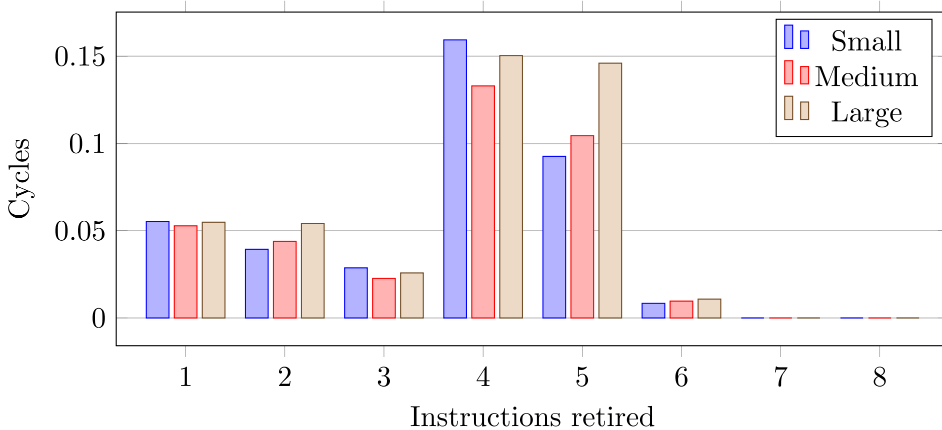


Figure 7.12: Histogram of instruction retired each cycle by query set size for the revised R*-tree with full node scans.

were once hard to predict have been removed.

A possible explanation for the large differences in branches may be the blocking used for the result set. The result set is essentially a long vector of bits where a bit is set when the corresponding MBB overlaps the query box. As for SIMD blocking, this allows skipping an entire block when no overlaps have been found, which is more often the case for the small query sets than the large. As the block size is larger, this effect should also be more prominent.

This difference in how the result sets are handled may also be the cause for the poor performance of the large query sets.

Together, the reductions in cache misses and branch mispredictions allows the processor to march forward at a higher speed than before, as can be seen in Figure 7.12. Comparing the numbers with the ones for the naive implementation shows a considerable increase in the number of cycles in which 4 to 5 instructions are retired at the cost of cycles in which fewer are retired.

7.5.4 Pruning Node Scans

The pruning node scans optimization provides the speedups found in Figure 7.13. As can be seen, by comparing with the speedups for full node scans seen in Figure 7.11, the small query sets have about the same speedup, while

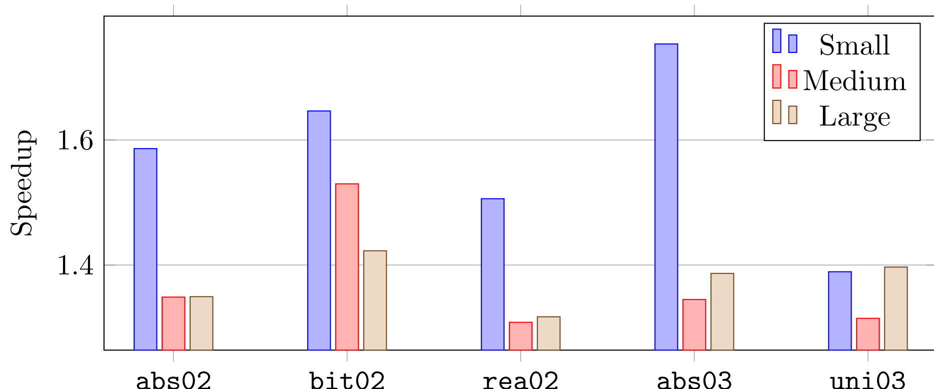


Figure 7.13: Speedup of the revised R*-tree with pruning scans by query set size.

the medium and large query sets improves all over.

This is also the expected behavior because the no overlap and full inclusion scenarios discussed in Section 5.4 are more likely when the query box is small. Thus this optimization mostly adds overhead for the small query set, but can prune away large amounts for the large query set.

Numbers acquired using instrumentation shows that this is indeed the case. Only 0.21 % of the comparisons are pruned away for the small query set, but for the large query sets, the corresponding number is 68 %. Judging by the speedup, this leaves the result extraction and general control logic as the most expensive part of the search.

The hardware counters reveals that cache misses are more frequent for the small query set with an increase of 11 %, while a reduction of 9 % and 39 % is seen for the medium and large query sets. This can also be found in the number of micro operations retired, where a 42 % increase is found for the small query sets, while only the large set experiences a small reduction of 4 %.

One may wonder why the reduction in the number of micro operations is not larger for the medium and large query sets, and the most probable answer is simply that the full scan adds some micro operations, such that the improvements from the pruning is almost neutralized when compared to the naive approach.

Not surprisingly, the distribution of instructions retired each cycle for the pruning scan is very similar to the one found for the full node scan

presented in Figure 7.12.

Chapter 8

Discussion

This chapter discusses the results found in Chapter 7, first from a memory perspective in Section 8.1, before Section 8.2 looks at parallelism. Finally, the research questions from the introduction are revisited in Section 8.3.

8.1 Memory Behavior

As explained in Chapter 2, programs for modern computer architecture must access memory in a pattern that allows the memory system to feed the processor with enough data.

The R-tree, and any tree structure in general, presents a challenge seen from a memory perspective because navigating through the tree consists of following a set of pointers. Since the destination of a pointer cannot be read before the pointer itself has been loaded from memory, the memory access pattern is essentially random.

That is not to say such structures cannot be optimized. Using nodes with several entries, such as the ones in the R-tree, gives a mix of sequential memory access during node scans and random memory access when accessing new nodes. This trades off random memory access for sequential access during node scans, since larger nodes give a lower tree which results in fewer random memory accesses.

Accessing disk pages sequentially also often displays less overhead than accessing pages randomly, so this effect should also exist for disk resident R-trees. In contrast to memory resident R-trees, the large size of disk pages makes the set of reasonable sizes quite small.

The suggested tradeoff between random and sequential memory access is also consistent with what was found in Section 7.3, where the tree height is found to be a major factor for determining the performance. Moreover, an optimal tree height, yielding the best performance, seems to exist.

The existence of an optimal tree height can in that case be seen as the optimal balance between random and sequential memory access. If the tree is too high, each node contains so few entries that the node scan cost drowns in the cost of fetching the nodes before the scan can start. A too low tree means scanning a lot of entries that would otherwise be unnecessary to scan, thus the cost of fetching nodes drowns in the scan cost.

It is also evident from the result that the optimal node size increases when optimizations improving the node scan performance are applied. As the optimizations lower the cost of scanning nodes, the balance is offset to favor a higher ratio of scans, which explains the increasing node size.

The finding of such a tradeoff may seem obvious, but this is not always the case. Some tree structures for spatial search, such as the *kd*-tree [5] is essentially a binary search tree.

Since the cost of memory access and scans is dependent on the hardware on which the search is running, the optimal parameters can be expected from system to system. This means the optimal node size no longer depends on a rather static disk page size, but must be optimized for each individual system.

Another, perhaps more interesting aspect, is that the most effective optimization will vary from system to system, depending on the system performance. Even though the balance between random memory access and compute intensive sequential scans can be adjusted using the parameters, some of the optimizations may reduce the performance in certain cases, such as when a system exhibits bad branch prediction.

This dependency on system specific properties may make optimizations rather unpredictable. As an example, the full node scan optimization sets out to reduce the number of random memory accesses performed, which seems to reduce the number of cache misses. Unfortunately, the bookkeeping information is evicted from the top level cache pretty quickly, causing an increase in the number of last level cache references. Depending on the pressure on the last level cache, this may be either a good or bad tradeoff.

From the results, it seems the speedup gained from an optimization varies significantly depending on the query set size. This is especially interesting for the pruning node scan, where the optimization is essentially

useless for the smallest query size, and at the same time is able to prune away large parts of the scans for the large query sets.

This bias towards a higher speedup for larger queries may make it seem like the pruning node scans has a minor effect on the average query performance, but since the larger queries usually requires much more work to complete than the smaller ones, the average speedup should be quite significant when an even mix of query sizes is expected.

However, in the case of the large queries, the effect of this with real data and query sets should be examined closer. The query sets used are, as explained in Section 6.1, generated based on the data set. This may have introduced scenarios where the pruning scan is especially efficient.

Obviously, a lot of memory related optimizations go untried in this thesis. For example could the latency of memory access during node scans possibly be hidden. One could imagine that each match found during a node scan may instantly trigger the issue of a prefetch instruction for the nodes. Hopefully, the matching node will reside in memory when the scan is done and the node accessed.

It should also be noted that the optimizations applied in this thesis are orthogonal to those applied in the CR-tree [25], meaning that they can all be applied to the CR-tree to further improve performance. For example can the pruning node scans be used to prune the compressed data, in which case the amount of data loaded from memory should be even less.

As presented in Section 7.5, one may be left with a feeling that the optimizations concerning computation are the most important ones. This is however slightly misguided because the experiments are done on a single core where the entire memory bus is as good as dedicated to that core. In a production setting, several cores are likely to share a memory bus exposing the growing gap between memory bandwidth and processor performance [39].

8.2 Parallelism

Filling the memory bus does however still require efficient code to be running on each core. As is evident from the results of the experimentation in Section 7.5, parallelism is the key to performance on modern processors.

The results show that SIMD scans speed up the search, especially for the small query sets. What is less certain is whether this speedup should be attributed to the use of SIMD instructions or to the changes made to

the algorithm. The SIMD instructions explicitly makes parts of the code parallel, but it may be that the processor could handle a set of independent instructions just as well by appropriately pipelining them and running them in parallel over several execution units.

An example of a situation where the implicitly available instruction parallelism seems to increase due to a change in the algorithm can be seen for the full scan node, where a change in the distribution of the number of instructions completing every cycle can be observed after a change in the algorithm similar to the one performed for the SIMD scan.

As expected, one of the real deal breakers for implicit instruction level parallelism, seems to be mispredicted branch instructions. In the case of the arrays of fields optimization, branch mispredictions drowns the benefit of fewer cache misses, but unfortunately, the reason for the large number of branch mispredictions, despite the small change of literally two lines of code, appears hard to track down.

This can possibly be blamed on the compiler applying different optimizations as a result of the small code change. Unfortunately, the compiler is not in the best position to judge what code results in a good or bad branch behavior as the design of modern branch predictors are often kept secret.

Even though a certain number of branches are needed to efficiently complete the task at hand, this does not mean the branches cannot be optimized. All optimizations presented herein, with the exception of the arrays of fields node layout, reduce the number of branch mispredictions, some by as much as 90 %.

In almost all cases are the reductions in the number of branch mispredictions accompanied by a less spectacular reduction of branches retired. Thus the reduction in mispredictions can to some degree be explained by the lower number of branches actually processed, but is also an effect of better branch prediction.

Note that, as for the memory behavior, the branch behavior can also be seen as a tradeoff between a predictable pattern that typically appears during node scans, and the less predictable pattern exhibited when accessing new nodes. The full node scan illustrates this as the number of instructions completing every cycle increases.

As for memory access, the number of mispredicted branches and which approach gives the best results vary based on the query sets used. It does however appear that improving the branch prediction performance of the small query sets is easier than for the others, but this may just as well be

because these query sets trigger a bad behavior for the naive implementation.

When it comes to multi core performance of the R-tree, it should be mentioned that the performance improvements found in this thesis are mostly applicable in a multi core environment as they mainly operate within the core itself. The only exception is that some of the instruction level parallelism may also be delivered using simultaneous multi-threading, in which case improvements to the instruction throughput may not be as useful.

8.3 Research Questions

Using the insight from the previous chapters, the research questions posed in the introduction can be answered more concretely.

> **What aspects of R-tree search are not suited for modern processors?**
>
> The random memory access that mainly occurs when accessing new nodes are not suited for the modern memory hierarchy and forces the processor to wait for data from memory.
>
> Also, a naive implementation of the R-tree does not take full advantage of the available parallelism in modern processors, neither prediction friendly branching or by exposing instruction level parallelism.
>
> **How can range search in R-trees be optimized such that more of the available data and instruction level parallelism can be utilized?**
>
> The memory access pattern can be optimized to ensure the processor can continue without having to wait for data. This largely consists of choosing the right parameters and applying the correct optimizations, in order to achieve the correct balance between random and sequential memory access.
>
> Otherwise, the branch behavior must be optimized for the data distribution such that the number of mispredictions is reduced. This goes hand in hand with making the available parallelism explicitly available to the processor through the use of SIMD instructions,

and perhaps other optimizations used to speed up scans in memory resident relational databases.

What are the main challenges in doing so?

As is evident from the result section, optimizations triggers varying behavior dependent on the query distribution and most likely the system on which the search is running. This complicates the optimization process because there may be no simple solution that fits all problems.

Chapter 9

Conclusion

With the ever exponentially increasing memory size of modern computers, and keeping indexes in memory becomes the default, optimizing search for memory resident databases is increasingly important. This has already been considered for memory resident relational databases, but not so much for spatial search.

A small set of traditional R-tree construction algorithms is first implemented using C/C++ and verified by reproducing the results of Beckmann and Seeger [4], using the data sets, query sets and parameters provided.

Four optimizations are then implemented. The first simply changes the node layout to an arrays of fields layout. The SIMD node scans builds on this, but uses SIMD blocking and SIMD instructions to make the available parallelism more explicit. Full node scans tries increase the amount of sequential memory access by scanning entire nodes and storing the results. Finally, the pruning node scans prune away coordinates from node scans. A suitable node layout means these coordinates are never loaded from memory.

For small query sets, the full node scans commonly yields speedups of 1.6 for the test data. Using hardware event counters, the source of the speedup is traced to an increase in the exploitation of available parallelism inherent in the search algorithm, and to a better memory access pattern.

For larger data sets, pruning node scans allows 68 % of the coordinates to be skipped by using knowledge acquired using the MBB of the node in which they reside. In combination with optimizations increasing the performance for small query sets, this gives a speedup of above 1.3 on all query sets.

The main problem of naive implementations of the R-tree seems to be rather terrible at exploiting the available instruction level parallelism, re-

ducing cache misses and pruning at a lower level than nodes, which is not surprising considering their disk based origin.

To exploit the increasingly parallel hardware, the search must be designed such that branches and cache misses do not hinder the full exploitation of instruction and data level parallelism.

Unfortunately, the results suggest different optimizations are applicable for different query distributions. This is also likely to be the case for varying hardware, and thus complicates the implementations of such optimizations in a production environment. Combining several optimizations yields a speedup, but possibly not as much as applying only a selected few.

9.1 Future Work

This thesis only considers a small set of optimizations, but these could be combined with several other methods such as the quantized and relative MBBs used by the CR-tree [25]. More novel approaches should also be explored, such as using explicit prefetching to mitigate the performance penalties of random memory access when visiting new nodes.

Since the effectiveness of the optimizations relay on the data sets, query sets and system on which the search is running, a solution where the optimizations to applied are determined dynamically at run time may help increase the performance in practice. This can be accomplished by monitoring the search performance while applying different optimizations.

Furthermore, the run time may also be possible to improve by considering different tree construction algorithms. Most popular algorithms today are based on the assumption that most of the cost associated with R-trees is related to disk access, and that an entire node must always be loaded in one go, which is no longer the case for memory resident indexes.

Appendix A

Implementation

This appendix describes the implementation used during benchmarking for constructing R-trees and performing searches in detail.

The code base consists of three parts:

- a framework for measuring index performance,
- spatial search indexes,
- a set of objects used as interface between the indexes and framework.

The framework is first described in Appendix A.1, before Appendix A.2 describes the indexes.

A.1 Framework

The framework for measuring spatial index performance compiles into an executable which loads the index from a dynamically linked library and inserts all data objects from the selected data set. The *reporters* specified on the command line are then run in sequence.

Each reporter performs some operation on the index and stores the result. Examples of such operations include searching, collecting statistics or doing instrumented search, but no operation may modify the index structure itself. When all reporters have completed, the results from each reporter is printed. This architecture allows constructing the index only once and collecting multiple statistics from the resulting structure.

A list of the implemented reporters can be found in Table A.1.

Name	Description
PAPI	Collects statistics captured during search from hardware counters
Search statistics	Collects statistics using an instrumented version of the search algorithm
Structure statistics	Only inspects the structure of the tree and reports the height, number of leafs, etc.

Table A.1: A list of the implemented reporters with a quick description

A.2 Indexes

Each index has been implemented from scratch using the relevant research papers where applicable [2, 4, 14, 24]. Note that, for the R*-tree, some of the improvements suggested by the authors themselves in a later paper are included [4].

The programming language used is C++11, where the configuration parameters (e.g. node size) have been extracted into template parameters. This allows heavier optimization to be performed during compilation since the parameters are available to the compiler.

Each index is compiled together with its configuration parameters into a separate dynamic library, which allows quick recompilation for different configurations.

The R-trees have also contains a routine for validating the structure of the tree after construction. During these self diagnostics, it is verified that for each node in the three

- the node's MBB is contained within its parent's MBR,
- the node's MBB is minimal,
- the number of children does not exceed the maximum, and
- the number of children is not below the minimum.

Violation of some of these conditions would result in a crash or incorrect results being returned from the index, but others would only slow down the search without triggering any error conditions elsewhere. Reporting these errors earlier eases development.

A.2.1 R-tree Composition

For the R-tree, the many variations of construction algorithms and storage layouts makes it necessary to be able to mix and match data layouts with search and construction algorithm. This is accomplished through extensive use of C++ templates, which abstract away the actual storage layouts used in nodes through proxy objects that allow access to the entries in a node.

In addition, the search algorithm relies on the node class to perform the actual node scan. An iterator yielded from the scan operation is then free to scan the node in any way suitable, for example by scanning the entire node at once or by scanning incrementally as more results are requested.

Although this may seem inefficient, the use of static inheritance through curiously recurring template parameters avoids expensive virtual function calls, and makes the code eligible for extensive optimizations by the compiler, such as inline expansion.

Appendix B

Probability Based R-trees

The disk based R-trees all tries to minimize the number of expected disk accesses. Peculiarly enough, no one seems to have gone through the trouble of defining a more formal model for calculating this expectancy.

The general intersection probability between an MBB and a query box is explored in Appendix B.1, before some more concrete results are presented in Appendix B.2. Then the actual expectancies are calculated in Appendix B.3, before a sketch of how this can be utilized when constructing R-trees is given in Appendix B.4.

B.1 Query Intersection Probability

First, we look at the one dimensional case. Let $M = (r_1, r_2)$ be a MBB and $Q = (Q_1, Q_2)$ be the query box, where Q_1 and Q_2 are stochastic variables. Define an indicator variable I_0 such that

$$I_0 = \begin{cases} 1 & \text{if } Q \cap M \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

Finally, let $W = Q_2 - Q_1$ be the width of the query box for convenience.

First, observe that

$$Q \cap M \neq \emptyset \Leftrightarrow Q_1 \leq r_2 \wedge Q_2 \geq r_1$$

which gives

$$\begin{aligned}P(I_0 = 1) &= P\left(Q_1 \le r_2 \wedge Q_2 \ge r_1\right)\\ &= P\left(Q_1 \le r_2 \wedge Q_1 + W \ge r_1\right)\\ &= \int_{-\infty}^{\infty} P\left(Q_1 \le r_2 \wedge Q_1 + W \ge r_1 \mid W = w\right) P\left(W = w\right) dw\end{aligned}$$

Assuming f_W and f_{Q_1} are the probability distribution functions of W and Q_1 respectively, this can be written as

$$P\left(I_0 = 1\right) = \int_0^{\infty} \left[\int_{r_1 - w}^{r_2} f_{Q_1}(r)dr\right] f_W\left(w\right) dw$$

For a d dimensional data space, define the stochastic indicator variable X_d which is 1 if Q intersects M. Let I_i be the probability of overlap in the ith dimension. Further assume the distribution of $I_1, \ldots, I_d$ are independent.

Since the query region overlaps with M if, and only if it does so in all dimensions, we have

$$P\left(X_d = 1\right) = \prod_{i=1}^{d} P\left(I_i = 1\right)$$

B.2 Example Distributions

The probability of intersection calculated in Appendix B.1 is however useless without the probability distributions for W and Q_1. In a real world setting, the distributions of Q_1 and W are likely to vary depending on the application. As will be shown below, existing approaches also implicitly relies on assumptions on the data distribution, but by defining this dependency more explicitly, one can also tailor the evaluation to the application.

For the sake of example and to balance query cost and simplify calculations, assume Q_1 is uniformly distributed in the range $[D_1 - W, D_2]$ where D_1 and D_2 are the upper and lower bounds of the data domain. This is equivalent to requiring that Q always intersects the data domain.

B.2.1 Constant W

The simplest example arise when W is constant. This can be accomplished by defining

$$f_{Q_1}(r) = \begin{cases} 1/(D_2 - D_1 + W) & \text{if } r \in [D_1 - W, D_2] \\ 0 & \text{otherwise} \end{cases}$$

which gives

$$\begin{aligned} P(I_0 = 1) &= \int_0^{\infty} \left[\int_{r_1 - w}^{r_2} f_Q(r) dr \right] f_W(w)\, dw \\ &= \int_0^{\infty} \left[\frac{r}{D_2 - D_1 + w} \right]_{r = r_1 - w}^{r_2} f_W(w)\, dw \\ &= \int_0^{\infty} \left[\frac{r_2 - r_1 + w}{D_2 - D_1 + w} \right] f_W(w)\, dw \end{aligned}$$

Now, assume a constant $W = w$. The probability distribution function f_W is then Dirac's delta function and the integral reduces to

$$P(I_0 = 1) = \frac{r_2 - r_1 + w}{D_2 - D_1 + w}$$

For $w = 0$, this reduces to a stabbing query case. The observant reader will notice that the probability then equals the volume of M normalized by the volume of the data space. This is consistent with what Guttman implicitly assumes by using the volume. As pointed out by Beckmann and Seeger, this also means the $P(I_0 = 1) = 0$ when M lacks volume, in which case Beckmann and Seeger falls back to using the perimeter for evaluation.

As a side note and sanity check, observe that

$$\lim_{w \to \infty} P(I_0 = 1) = 1$$

which is expected as an infinitely large query is bound to intersect the MBB no matter where it is placed.

B.2.2 Uniform Q_1 and W

A more complex example can be found by assuming a uniform W in the domain $[0, w_{\max}]$. This gives

$$f_W(w) = \begin{cases} \frac{1}{w_{\max}} & \text{if } r \in [0, w_{\max}] \\ 0 & \text{otherwise} \end{cases}$$

which slips right into the formula and gives

$$\begin{aligned}
P\left(I_0=1\right) &= \int_0^{w_{\max}} \left[\frac{r_2-r_1+w}{D_2-D_1+w}\right] \frac{1}{w_{\max}} dw \\
&= \frac{1}{w_{\max}} \int_0^{w_{\max}} \frac{r_2-r_1+w}{D_2-D_1+w} dw
\end{aligned}$$

Define $\Delta D = D_2 - D_1$, $\Delta r = r_2 - r_1$, substitute with $s = \Delta D + w \Rightarrow w = s - \Delta D$ and complete the integration

$$\begin{aligned}
P\left(I_0=1\right) &= \frac{1}{w_{\max}} \int_{\Delta D}^{w_{\max}+\Delta D} \frac{\Delta r + s - \Delta D}{\Delta D + s - \Delta D} dw \\
&= \frac{1}{w_{\max}} \int_{\Delta D}^{w_{\max}+\Delta D} \left[\frac{\Delta r - \Delta D}{s} + 1\right] dw \\
&= \frac{1}{w_{\max}} \left[\left(\Delta r - \Delta D\right) \int \frac{1}{s} dw + \int dw\right]_{\Delta D}^{w_{\max}+\Delta D} \\
&= \frac{1}{w_{\max}} \left[\left(\Delta r - \Delta D\right) \ln s + s\right]_{\Delta D}^{w_{\max}+\Delta D} \\
&= \frac{1}{w_{\max}} \left[\left(\Delta r - \Delta D\right) \left(\ln(w_{\max} + \Delta D) - \ln \Delta D\right) + w_{\max}\right] \\
&= \frac{1}{w_{\max}} \left[\left(\Delta r - \Delta D\right) \ln\left(\frac{w_{\max} + \Delta D}{\Delta D}\right) + w_{\max}\right] \\
&= \frac{\Delta r - \Delta D}{w_{\max}} \ln\left(\frac{w_{\max}}{\Delta D} + 1\right) + 1
\end{aligned}$$

Since both $w_{\max}$ and ΔD are constants, the probability is proportional to Δr:

$$\begin{aligned}
P\left(I_0=1\right) &= \left(\Delta r - \Delta D\right)\sigma + 1 \\
&= \sigma \Delta r + \left(1 - \sigma \Delta D\right)
\end{aligned}$$

where

$$\sigma = \frac{1}{w_{\max}} \ln\left(\frac{w_{\max}}{\Delta D} + 1\right)$$

Also, because M is expected to reside within the data domain, it is given that $\Delta r - \Delta D \leq 0$, and thus the probability of intersection decreases with

increasing σ. Since $\frac{d\sigma}{dw} < 0$, the maximum is found when approaching 0.

$$\begin{aligned}\lim_{w_{\max}\to 0} \sigma &= \lim_{w_{\max}\to 0} \frac{1}{w_{\max}} \ln\left(\frac{w_{\max}}{\Delta D} + 1\right) \\ &= \lim_{w_{\max}\to 0} \frac{1}{\frac{w_{\max}}{\Delta D} + 1} \frac{1}{\Delta D} \\ &= \frac{1}{\Delta D}\end{aligned}$$

This gives the maximum possible probability of intersection, assuming $w_{\max} \to 0$,

$$\begin{aligned}P_{\max}(I_0 = 1) &= (\Delta r - \Delta D) \frac{1}{\Delta D} + 1 \\ &= \frac{\Delta r}{\Delta D}\end{aligned}$$

which is 0 only when $\Delta r = 0$, or in other words when the box lacks volume. This makes sense as an infinitely thin query box has absolutely no probability of intersecting an infinitely thin MBB.

Note however that $P(I_0 = 1) > 0$ assuming either $\Delta r > 0$ or $w_{\max} > 0$, which is a very reasonable assumption as stabbing queries in point data makes no sense. This renders falling back to perimeter, as done by Beckmann and Seeger for the revised R*-tree, unnecessary.

B.3 Expectancies

As a slight simplification, assume each node accessed incurs a disk access, and let the number of node visits when searching through the sub tree defined by a node N be given by the stochastic variable V_N.

Using a recursive approach, it is obvious that visiting a node incurs 1 visit in addition to the number of visits required in each of the sub trees intersecting with the query rectangle. Since expectancies are linear, this gives

$$E(V_N) = 1 + \sum_{N' \in N} E(V_{N'}) P(\mathrm{MBB}(N') \cap Q \neq \emptyset)$$

For a memory resident R-tree, it may however make more sense to minimize the expected run time. Assuming the time required for scanning a node is linear in the number of entries, and that accessing a node incurs

a time penalty of a main memory access, the expected time for searching through a node N is given by

$$E(T_N) = t_a + |N|\, t_s + \sum_{N' \in N} E(V_{N'}) P(\mathrm{MBB}(N') \cap Q \neq \emptyset)$$

where t_a is the memory access latency and t_s is the time required for scanning a single entry.

Note that these assumptions are consistent with what was found in Section 7.3.

More complex expressions can be developed to include other optimizations, such as coordinate pruning as described in Section 5.4.

B.4 Implementation

Calculating the expectancies is not useful in itself and must be combined with a strategy for constructing the tree. As mentioned earlier, inserting nodes is already treated as an optimization problem, but few of the existing solutions takes all the possible solutions into account.

Taking all possible solutions into account would not be reasonable to assume either, since the problem is NP-hard in the general case, as will be shown in Appendix B.4.1, after which Appendix B.4.2 then gives a sketch of how a branch-and-bound approach may help solve the problem.

B.4.1 NP-hardness

The problem of finding an optimal split can be formulated as follows. Given

- a set of entries S,
- a weighting function $w(N)$ giving the expected cost of visiting the node associated with the entries in N, and
- a probability function $f(N)$ giving the probability of visiting a node consisting of the entries in the set N,

find a set $\mathcal{N}$ of disjoint sets covering S such that

$$Z = 1 + \sum_{N \in \mathcal{N}} w(N) f(N)$$

is minimized and $|\mathcal{N}| > 1$ (otherwise no split has occurred).

Next, a reduction from the vertex cover problem can be as follows. Given a graph from the set cover problem $G = (V, E)$, let $w(N) = 1$ for all N and

$$S = \begin{cases} V & \text{if } |V| \text{ is odd} \\ V \cup \{v\} & \text{otherwise} \end{cases}$$

where v is a dummy vertex without any incident edges. Note that the result is that $|S|$ is always odd. Finally, let

$$f(N) = \begin{cases} \frac{|N \setminus \{v\}|}{|S|} & \text{if } N \setminus \{v\} \text{ is a vertex cover of } G \text{ and } |N| \text{ is odd} \\ 1 & \text{otherwise} \end{cases}$$

Clearly, since $f(N) \leq 1$, a solution with $|\mathcal{N}| > 2$ will never be optimal. Consequently, assuming no vertex cover exits in a solution $\mathcal{N}$, it is known that $Z = 3$.

Next observe that only one of the sets $N \in \mathcal{N}$ is such that $f(N) < 1$, as only one of the sets will be odd. Furthermore, $f(N)$ will be smaller for smaller $|N|$, thus the minimal value of $f(N)$ will be achieved only for the minimal set cover, in which case

$$Z = 2 + \frac{|N \setminus \{v\}|}{|S|} < 3$$

since $|N| < |S|$.

The minimal set cover can thus be found by finding by solving the problem for the set $\mathcal{N}$ that minimizes Z, and selecting the set $N \in \mathcal{N}$ that is a set cover for G. The minimal set cover is then $N \setminus v$.

Note that this proof handles $f(N)$ as an unknown, thus there may be special cases where finding the optimal solution is not as hard, depending on $f(N)$. The most trivial example is perhaps that where $f(N) = 0$, in which case any solution is optimal.

B.4.2 Using Branch-and-bound

The most straight forward application of the above may be during splits, in which case two major choices are to be made:

1. the number of new nodes must be decided, and

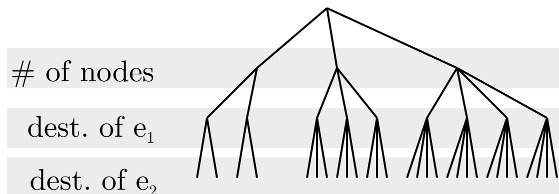


Figure B.1: Decision tree for distributing 2 entries over 2 to 4 nodes.

2. a distribution of entries over the new nodes.

Note that Item 1 may already be decided, and commonly is so for the existing algorithms. For example, the R*-tree always creates only one new node.

When it comes to distributing the entries between the nodes, this can be seen as a series of choices. For each entry, a destination node must be selected. Based on this, a decision tree can be constructed as illustrated in Figure B.1.

Furthermore, the best path through this tree can be found using branch and bound given a goal function. For this to be efficient, the goal function must have good bounds even when only a few choices have actually been made.

Goal Function The expected search time or node accesses should be minimized for memory or disk resident R-trees respectively. The previously calculated expectancy for a node can be used directly to find the expectancy $E(V_S)$ for a split candidate S.

Since the expression is recursive, saving the expected number of scanned entries in each entry is probably necessary to avoid duplicated calculations when the expression is evaluated repeatedly, which would be the case using branch and bound.

Lower Bound A lower bound can be found by simply calculating $E(V_s)$ assuming the entries not yet distributed do not exist. This is computationally cheap, but may not yield a tight bound.

Another option is to assume the remaining entries all fit within the node with the lowest probability of intersection, $P(e.M \cap Q \neq \emptyset)$, and calculate $E(V_s)$ accordingly.

Upper Bound For the upper bound, one solution is to assume the remaining entries are all scanned, or in other words that they will end up in a node intersecting the query rectangle in all cases. This is cheap, but unlikely to be tight.

Alternatively, all the remaining entries can be assumed to be added to the node giving the lowest bound.

Optimizations The search space for a branch and bound algorithm is extremely large, $(n-1)\sum_{i=2}^{n} i^{|E|}$ where m is the maximum number of resulting nodes to consider and $|E|$ is the number of entries in total, thus it is likely that the algorithm needs some optimizations to have a reasonable performance.

A possible way to speed up the algorithm is to sort the entries in a suitable order to maximize the cases where the upper and lower bounds are able to prune the search space. For example, sorting the entries by the distance from the center of all entries may increase the chance of the split candidate's MBBs growing during the first choices.

A given order for the entries also allows precalculating the sum of the expected entries scanned and MBB of the MBBs, thus speeding up the bound calculations.

Another way may be to restrict the search space as done by existing R-tree variants.

B.4.3 Leaf Selection (`ChooseSubtree`)

Selecting a leaf can be done using the same approach by, at each level, using the same bound with only a single entry left. In this case the search space is considerably smaller as it is limited to the size of the tree.

Bibliography

[1] J. Alber and R. Niedermeier. On multidimensional curves with Hilbert property. *Theory of Computing Systems*, 33(4):295–312, 2000.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 322–331, 1990.

[3] N. Beckmann and B. Seeger. A benchmark for multidimensional index structures. http://www.mathematik.uni-marburg.de/˜rstar/benchmark/.

[4] N. Beckmann and B. Seeger. A revised R*-tree in comparison with related index structures. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 799–812, 2009.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[6] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: an index structure for high-dimensional data. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 28–39, 1996.

[7] H. Berliner. The B* tree search algorithm: a best-first proof procedure. *Artificial Intelligence*, 12(1):23–40, 1979.

[8] M. Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[9] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[10] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 426–435, 1997.

[11] R. W. Eglese. Simulated annealing: a tool for operational research. *European Journal of Operational Research*, 46(3):271–281, 1990.

[12] Z. Feng, E. Lo, B. Kao, and W. Xu. ByteSlice: pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 31–46, 2015.

[13] D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 606–615, 1989.

[14] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 47–57, 1984.

[15] H. Haverkort. An inventory of three-dimensional Hilbert space-filling curves. 2011. arXiv: 1109.2323.

[16] J. L. Hennessy, D. A. Patterson, and K. Asanović. *Computer architecture: a quantitative approach*. 5th ed edition, 2012.

[17] D. Hilbert. Ueber die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, (38):459–460, 1891.

[18] J. Huh, D. Burger, and S. W. Keckler. Exploring the design space of future CMPs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 199–210, 2001.

[19] S. Hwang, K. Kwon, S. K. Cha, and B. S. Lee. Performance evaluation of main-memory R-tree variants. In *Advances in Spatial and Temporal Databases*, pages 10–27. 2003.

[20] Intel Corporation. Intel® Xeon® processor E5-2683 v3 (35m Cache, 2.00 GHz) specifications.

[21] Intel Corporation. Intel® 64 and IA-32 architectures optimization reference manual, 2016.

[22] Intel Corporation. Intel® 64 and IA-32 architectures software developer's manual volume 1: Basic architecture, 2017.

[23] I. Kamel and C. Faloutsos. Parallel R-trees. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 195–204, 1992.

[24] I. Kamel and C. Faloutsos. Hilbert R-tree: an improved R-tree using fractals. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1993.

[25] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 139–150, 2001.

[26] J. K. Lawder. Calculation of mappings between one and n-dimensional values using the hilbert space-filling curve. Technical Report JL1/00, 2000.

[27] Y. Li and J. M. Patel. BitWeaving: fast scans for main memory data processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 289–300, 2013.

[28] K. I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: an index structure for high-dimensional data. *The VLDB Journal*, 3(4):517–542, 1994.

[29] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 261–270, 2009.

[30] D. Molka, D. Hackenberg, and R. Schöne. Main memory and cache performance of Intel Sandy Bridge and AMD Bulldozer. In *Proceedings of the ACM Workshop on Memory Systems Performance and Correctness (SIGPLAN)*, pages 4:1–4:10, 2014.

[31] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel. Cache coherence protocol and memory performance of the Intel Haswell-EP architecture. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 739–748, 2015.

[32] N. Mukherjee, S. Chavan, M. Colgan, D. Das, M. Gleeson, S. Hase, A. Holloway, H. Jin, J. Kamp, K. Kulkarni, T. Lahiri, J. Loaiza, N. Macnaughton, V. Marwah, A. Mullick, A. Witkowski, J. Yan, and M. Zait. Distributed architecture of Oracle database in-memory. *Proceedings of the VLDB Endowment*, 8(12):1630–1641, 2015.

[33] M. A. Roth and S. J. Van Horn. Database compression. *SIGMOD Record*, 22(3):31–39, 1993.

[34] J. Von Neumann and M. D. Godfrey. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[35] B. Wang, H. Horinokuchi, K. Kaneko, and A. Makinouchi. Parallel R-tree search algorithm on DSVM. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 237–244, 1999.

[36] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, volume 98, pages 194–205, 1998.

[37] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 516–523, 1996.

[38] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.

[39] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[40] S. Zeuch, F. Huber, and J.-C. Freytag. Adapting tree structures for processing with SIMD instructions. In *Proceedings of the International Conference of Extending Database Technology (EDBT)*, pages 97–108, 2014.