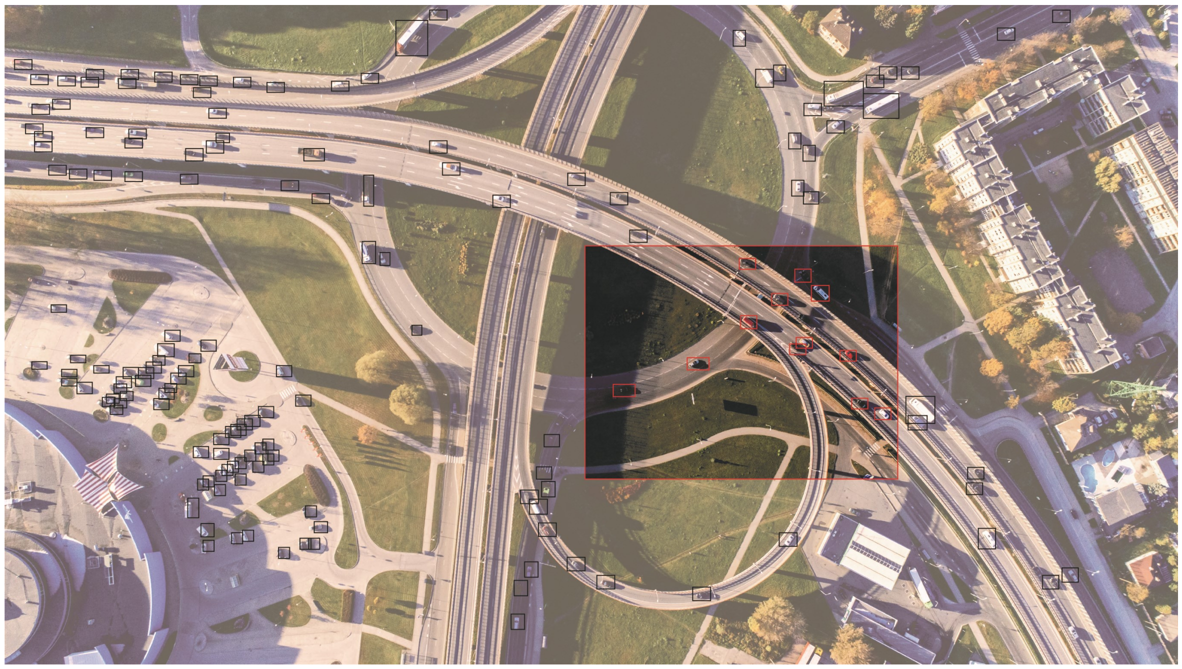
CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG



Spatial Indexing for Moving Geometry in Main Memory

Master's thesis in Computer Science: Algorithms, Languages and Logic

JAKOB AASA
MARCUS LUNDBERG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Spatial Indexing for Moving Geometry in Main Memory

JAKOB AASA
&
MARCUS LUNDBERG

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Spatial Indexing for Moving Geometry in Main Memory
JAKOB AASA
MARCUS LUNDBERG

Supervisor: Erik Sintorn, CSE
Advisor: Patrik Ellrén, Carmenta
Examiner: Ulf Assarsson, CSE

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An example of a window query operation. The background image is in the public domain. Graphics by Karin Nilsson.

Typeset in LaTeX
Gothenburg, Sweden 2019

Spatial Indexing for Moving Geometry in Main Memory
JAKOB AASA & MARCUS LUNDBERG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Spatial indexes are data structures which store objects in the form of points or geometry in two or more dimensions in such a way that subsets can be queried with high performance. However, good query performance is no guarantee for a corresponding update performance. There is currently little research of spatial indexing for non-point geometry which receive frequent updates.
This thesis studies and compares different spatial indexes for this kind of data. The evaluated data structures are the simple quadtree, the loose quadtree, the loose-linear quadtree, and the R*-tree. A dynamic array is also implemented to represent a naïve approach.
Where applicable, we augment the spatial indexes with two update techniques: bottom-up updating and update memo, to assess if these improve performance.
Evaluation is performed by a benchmark suite, where a scenario of objects sampled from different data distributions is used to quantify query and update performance of the spatial indexes. This evaluation is divided into two steps. First, parameters specific to each data structure is chosen, with 10 million objects in the scenario. Then, we compare the data structures, the update techniques, and the memory usage of the selection.
We find that the loose quadtree performs best for all measured scenarios in both updates and queries, while the R*-tree is worst, if not counting the query performance of the dynamic array. Bottom-up updating and update memo yielded unsatisfactory performance given the extra memory that is needed.
The contribution of this thesis is twofold. First, we perform a thorough performance comparison for spatial indexes that support moving non-point geometry. To our knowledge, there exist no such survey at the time of writing. Secondly, we present novel query algorithms for the loose-linear quadtree which perform at least an order of magnitude better than other existing approaches.

Keywords: Computer, science, computer science, engineering, spatial index, quadtree, r-tree, main memory, thesis.

Acknowledgements

We would like to thank our supervisor Erik Sintorn for his invaluable feedback, as well as our examiner Ulf Assarsson. We would also like to thank Carmenta and our advisor at the company, Patrik Ellrén, for providing, among other things, the idea for this thesis, work stations, and copious amounts of coffee.

Jakob Aasa & Marcus Lundberg, Gothenburg, August 2019

Contents

List of Figures

List of Tables

1
Introduction

How to efficiently store and analyze data with spatial properties, such as position and geometrical form, has been an important research area of computer science since the 1970s. This research has resulted in many new, purpose-built data structures. These kinds of data structures are commonly referred to as *spatial indexes*. More recently, the increase in spatial data gathering (positions of cars, phones, etc.) has made spatial indexing a more relevant, and thriving, area of research than ever. While older research on spatial index structures has focused on improving performance for querying subsets of the stored data, how to efficiently modify this data's spatial properties, such as moving objects' positions, has become increasingly important.

The field of spatial indexing mainly focuses two different kinds of data: static geometry and moving point data. An example of static geometry — with a spatial extent and that does not move — is to represent objects such as roads and buildings in traditional geographic information systems. Moving point data, on the other hand, has no spatial extent and is not static; examples being location data of cars and phones. Due to modern tracking technology, spatial indexing of moving point data has seen renewed interest in recent years [43, 50, 11, 46].

The focus on this thesis is on moving geometry, non-static data with spatial extent. Such data is commonly used in n-body physics simulations and computer graphics but can also be used to represent inaccuracy in location tracking of moving point data.

Spatial indexing of moving geometry is a relatively unexplored field [41]; research on the area has only been published in recent years [20, 41, 10]. The purpose of this thesis is to compare different spatial indexes that can store moving geometry data, and evaluate them for different parameters. Focus is on performance given large amounts of objects, especially update performance. Query performance is also considered because the main purpose of spatial indexes is to provide faster querying than simply iterating through all objects. To our knowledge, there exist no published literature that examines and compares spatial indexes for moving geometry.

This thesis also presents novel methods. An update-memo method for the loose quadtree is developed, where garbage cleaning depends more on the dynamic data's distribution compared to existing methods. For the loose-linear quadtree, we describe two algorithms that improve on the query algorithm introduced by Aboulnaga et al [1], with orders of magnitude better query performance in our evaluation.

We implement and test versions of a list (based on the C++ `std::vector`), pointer-based quadtree, loose quadtree, loose-linear quadtree, and R*-tree. Some of these are also equipped with two update techniques: bottom-up updating and update memo.

The results show that the loose quadtree outperforms all other evaluated spatial indexes by far. Bottom-up updating, with which some indexes are augmented, has limited performance effects and large memory overhead. The other update technique, update memo, also proves to performs poorly.

This thesis is done in collaboration with Carmenta.

The following section gives an example of an application where a spatial index can be used. Section 1.2 narrows the scope and describes what the thesis cover in more detail.

1.1 Example Application: Command and control system

Consider an emergency services command-and-control center with a central server. Such a center coordinates and dispatches proper resources to emergency situations. The server tracks substantial amounts of live positional data from varying sources, such as mobile phones, cars, weather phenomena, and so on. Positions for these are updated up to once per second, for each non-static object. Due to inaccuracy from GPS positioning or location estimation based on mobile towers, data represented as points have a certain amount of uncertainty, these are therefore represented as geometric objects with areas in the application.

Control operators' use client workstations to interface with the command-and-control-systems server. Each operator has a window on their screen of a map displaying data from the server corresponding to its view area. It is important that the objects shown on the operator's screen are as current and relevant as possible, and that the operator can pan and zoom around the map and quickly get a response from the server with corresponding data.

The information an operator requests from the server is referred to as a *window-query* operation: fetch all objects that intersect with a given axis-aligned rectangle — the screen window in this case. In addition, for the server to provide updated answers to window queries, it is important to return answers as quickly as possible.

1.2 Scope & Limitations

Spatial indexing is an old and well-established research area. This thesis covers a niche of this area, which is explained below:

- *Two-dimensional data.* Implementations and evaluations are made for two dimensions. It is common to examine spatial indexes for only two or three dimensions [41, 6, 20], even if a spatial index supports any number of dimensions.
- *Axis-aligned rectangles.* Only axis-aligned rectangles are indexed — they can be used to approximate any shape. This is a common approach in the spatial

indexing field [17].

- *Window queries.* The only query operation evaluated is window querying. No other kind of query, such as nearest-neighbor queries, is considered.
- *Main memory.* Databases that use spatial indexing are often stored in secondary memory such as hard drives, where writes and reads to and from disk (I/O operations) are limiting performance factors [38]. Since the problem sizes this thesis researches fits in main memory (RAM), all indexes are implemented for, and evaluated in, main memory.
- *Sequential access.* This thesis only considers sequential operations. Concurrency would complicate the evaluation process substantially: both sequential and concurrent access must be measured by themselves, in order to draw any conclusions of what factors affects the performance of the chosen spatial indexes.

All chosen spatial indexes are implemented in C++ and compiled using the Visual C++ 2019 compiler. Evaluation is performed on a workstation running Windows, with an Intel Core i7-8700 processor and 64 GB of RAM. Results may differ for other configurations.

1.3 Ethical considerations

The technologies and algorithms described in this thesis can be, like most other, used for many different purposes. It is first when applied the usage can be deemed good or evil.

The spatial indexing approaches described and implemented in this thesis might be used to store confidential data. This thesis, and in general, the field itself, does not discuss the security of the data stored. Security is something that has to be considered in specific implementations, if the stake-holders of said implementations deem it necessary. All data used for this thesis was randomly generated, and is therefore not sensitive or confidential.

Real-time applications often rely on relevant data, and this is the area where spatial indexing can make a difference. Some rely on tracking technology, which can be used in both civil and military contexts. One possible usage where ethical concerns would arise is surveillance. A nefarious state might use spatial indexing approaches to track its own or other states' citizens more efficiently than would otherwise be possible. This is, however, only one minor component of all technology required for this purpose. A positioning system is required to gather the data to begin with, for example. On the other hand, rescue services could use the technology (see Section 1.1) to save lives.

Not considering the minor novel additions presented by this thesis, we only compare existing methods that have already been published. The ethical implications of this thesis are therefore limited.

2
Related Work

Spatial indexing is an old and well-researched domain. In this chapter we give an overview of published indexing approaches, grouped in different categories.

Most spatial-indexing approaches are derived from a few variants, quadtrees, kd-trees, and r-tree, published in the 1970s and 1980s. These approaches laid the groundwork for the research area of spatial indexes. These are described together with some of their derivatives in Section 2.1, 2.2, and 2.3. For a more in-depth explanation of the variants evaluated in this thesis, see Chapter 3.

2.1 Quadtrees

One of the earliest spatial indexing structures is the *quadtree*, proposed in 1974 by Finkel et al. [13]. Originally a tree data structure designed for an arbitrary number of dimensions, it is nowadays associated with a two-dimensional version where each node has four children[1], each representing a quarter of the node's space. Finkel et al.'s original version is referred to as a *point quadtree* in other literature [40, 14, 16]. In their version, each node throughout the tree represents a data point.

A drawback of Finkel et al.'s approach is that the areas spanned by a node's children are dependent on how the point, which defines the node, is placed. The *Point-Region quadtree* [35], introduced a few years later, instead place all data in the leaves and split each node into equal-sized rectangular sub-nodes. The partitioning of space into equal-sized sub-nodes at each level proved to be a popular approach. For handling non-point data, the *MX-CIF quadtree* [23] stores objects as deep in the tree as possible, so that a node is minimally enclosing them.

A version of the quadtree in three dimensions is the *octree* [31], where each node has eight children. Octree variants are particularly popular for accelerating intersection testing in the domain of computer graphics [41].

In 2000, Ulrich [47] introduced a version of the octree called *loose octree*, where a node's volume is expanded by a factor; this allows objects to be stored deeper in the tree. A constant-time algorithm for finding the minimally enclosing node for an object in loose quadtrees was developed by Samet et al. [41].

A common representation of a quadtree to use explicit node objects, where every node keeps a pointer to each of its four children. Gargantini [15] proposed an alternative

[1]In a general k-dimensional tree, each node has 2^k children.

representation, the *linear quadtree*, where a unique identifier for each quadtree node is calculated based upon its placement and depth in the tree. This identifier is then used to directly access the node — no traversal in the quadtree is needed if one knows beforehand which node to access. There exist several query algorithms for linear quadtrees, of which Aboulnaga and Samet's [1] is the most recent. In contrast to an earlier algorithm presented by Aref and Samet [4], spatial objects are permitted to overlap each others.

2.2 kd-trees

Another early take on spatial indexing is the k-dimensional tree (kd-tree), introduced by Bentley in 1975 [7]. The kd-tree is a generalization of the binary search tree for k-dimensional data, where every level of the tree orders children on the next level by a given dimension. Extensions of this method include the K-D-B-tree [39], and Bkd-tree [36].

2.3 R-trees

Introduced in 1984 by Guttman [17], the R-tree is a very influential spatial index [30]. Variants that extend and improve the R-tree include R$^+$-tree (1987) [42], R*-tree (1990) [6], Hilbert R-tree (1993) [21], and many more [29].

R-trees is a height-balanced tree where all data is stored in the leaves. Each node keeps a minimum bounding box around its children. This yields good query performance — if the tree is constructed well.

2.4 Storage-specific approaches

One aspect that affects the performance of a spatial index is memory access, whether from cache, main, or secondary memory. Therefore, some research has focused on improving this aspect.

2.4.1 Secondary storage

Due to the limited availability of RAM in early computing, almost all spatial indexes at the time were designed to be stored on a secondary storage such as hard drives. An effect of this is that the bottleneck becomes reading and writing to the storage medium (I/O operations) [38].

Biveinis et al. [8] proposes buffering operations in main memory to reduce disk I/O.

2.4.2 Main memory

Main memory is much faster to access, compared with secondary storage. Since it became workable for some applications to store all objects in main memory, some research has focused on main memory structures. While main memory is much faster than secondary storage, reading and writing to it is still much slower than when the data resides in cache. Cache-conscious approaches include MR-tree [25] and CR-tree [24]; cache-oblivious include Packed-Memory Quadtree [46].

2.4.3 Bulk loading

If the data is known in advance before insertion into the index, it can be bulk-loaded. There are approaches that utilize this to improve performance. Two examples of this is Packed R-tree [22] and Packed Memory Quadtrees [46].

2.5 Update techniques

This section describes approaches used to facilitate a higher updating performance than is usually reached with traditional spatial indexes.

2.5.1 Bottom-up updates

One method to improve update performance for spatial indexes is *bottom-up updating*, which uses a secondary index (often a hash-map) to directly access the current position of an object in the main index instead of traversing the index. It was originally proposed by Lee et al. [26], with application to R-trees. Šidlauskas et al [44] further developed the idea. They also provided a bottom-up version of a uniform grid index.

2.5.2 Update Memos

Another method for updating is the *update memo* technique, introduced by Silva et al. [45], where the newest version of a moved object can co-exist with older versions, which a garbage cleaner regularly removes. Silva et al. used the technique to augment the R-tree, resulting in a new variant called the RUM-tree. Update memo was combined with bottom-up updates in the RUM+-tree [49]. The technique is also applied to loose octrees by Deng et al. [10].

2.5.3 Throwaway index structures

Spatial indexes such as MOVIES [11] and TwinGrid [43] are designed on the assumptions that given enough updates, it is computationally faster to construct a new index rather than updating an old one. Therefore, they quickly construct short-lived indexes which are replaced by newer ones regularly. The defining limitation of this approach is that the constructed index quickly becomes irrelevant, since it is a snapshot of each object's position when it was constructed.

2.6 Distributed spatial indexing

Due to limits in computation performance of a single machine, some research has dealt with spatial indexing for distributed systems [27]. It is worth noting that the main performance limitation on distributed systems is network communication [2].

An example of a distributed spatial index is SIFT [20]. SIFT's basic data structure is a quadtree, but it also separates data according to workload-independent data parameters, so that work is more evenly split between different machines. Computer nodes in the system are also assigned specific parts of the quadtree.

Another distributed spatial index is HQ-Index [19], which is a point-based quad-tree adopted for the Hadoop framework.

Then there is ToSS [2], which uses Voronoi diagrams to partition the index into several nodes.[2] Queries are then sent to a local node, which queries its neighboring nodes in case the query overlaps with those. Updates are performed by constructing new indexes and throwing away the old ones.

2.7 Surveys

There have been several attempts at surveying and summarizing the research field of spatial indexing. Gaede et al. [14] published a survey in 1998, covering the years 1966-1996. Three different versions of a survey, “Spatio-temporal access methods”, consider the years 1980-2003 [32], 2003-2010 [33], and 2010-2017 [27]. Another survey, which also discusses access methods in other domains, was published by Markov et al. [30] in 2008.

[2]A Voronoi diagram separates regions of space by a set of points. A point's associated region is the space that lies closer to it than any other point, according to some distance metric.

3 Theory

This chapter provides an overview of spatial index structures deemed relevant to this thesis as well as techniques for improving update performance.

The term *spatial index* is generally defined as a data structure that organizes a set of multi-dimensional geometry [3, 37]. As the name suggests, spatial indexing differs from other indexing approaches in that it uses the spatial features of the indexed objects to speed up certain kinds of queries. Examples of such queries are fetching all objects within a certain area and finding which objects are closest to a specific object.

Other terms used for the spatial index in the literature include *spatial data structure* [3], *multi-dimensional index* [39], and *Space Access Method* (SAM)[1] [6, 48, 38]. From here on, we will refer to these methods collectively as spatial indexes.

3.1 Domain

This section aims to describe the domain that this thesis covers: how objects are represented, and how a spatial index operates on this representation.

All objects are represented as axis-aligned rectangles. Rectangles provide good approximations of other data: points can be represented as small squares centered on point coordinates and other geometry can be enclosed by minimum bounding rectangles (MBR).

We define an object o to contain two points, $\{x_a, y_a\} \in \mathbb{R}^2$ and $\{x_b, y_b\} \in \mathbb{R}^2$, which span an area, $\{x_a, y_a, x_b, y_b\} \in \mathbb{R}^4$ (see Figure 3.1), and a unique identifier, $\text{id} \in \mathbb{N}$. The object is then $o \in \mathbb{N} \times \mathbb{R}^4 = \mathbb{O}$.

A spatial index containing rectangles can be viewed as a member of the power-set of rectangles, $\mathcal{P}(\mathbb{O})$. Operations on this structure include:

- *intersects*: $\mathcal{P}(\mathbb{O}) \times \mathbb{R}^4 \longrightarrow \mathcal{P}(\mathbb{O})$
- *insert*: $\mathcal{P}(\mathbb{O}) \times \mathbb{O} \longrightarrow \{true, false\}$
- *delete*: $\mathcal{P}(\mathbb{O}) \times \mathbb{O} \longrightarrow \{true, false\}$
- *update*: $\mathcal{P}(\mathbb{O}) \times \mathbb{O} \times \mathbb{O} \longrightarrow \{true, false\}$

Given a query rectangle and a spatial index, the function *intersects* seeks to return

[1]Methods that only support points are often called *Point Access Methods* (PAM) [48, 30].

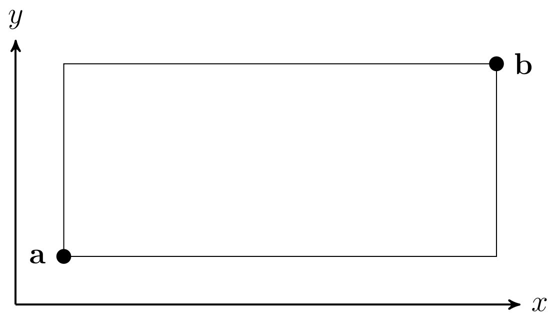


Figure 3.1: A rectangle area spanned by points **a** and **b**.

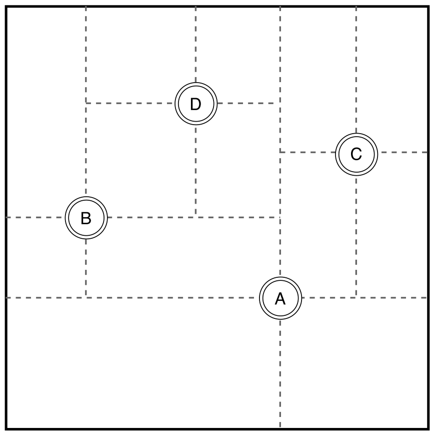
all rectangles in this index that intersect the query rectangle. For example, the query rectangle could be an area explicitly chosen by a user or a window query where an entire screen defines the search rectangle. Functions *insert* and *delete* take a rectangle as input and either adds it to the given spatial structure or removes it. With *update*, the index will receive a new position for a rectangle. Depending on whether the index keeps track of all current positions of objects or not, the function is optionally supplied with the old position of the rectangle. This function can, but does not necessarily have to, be implemented in terms of the *insert* and *delete* operations. The return value $\{true, false\}$ indicates whether executing the function was successful or not.

3.2 Spatial indexes

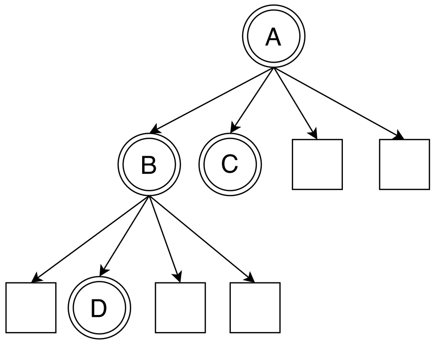
Over the years many methods have been developed in the spatial indexing field. These can be divided into two main areas: space partitioning and data partitioning [44, 38, 34]. Space partitioning means that the available space is partitioned such that no consideration of the data distribution is taken; for example, a tree can be defined where nodes at each level divide the available space equally.

The other approach, data partitioning, explicitly divides according to how the data is distributed. An example of data partitioning is a bounding hierarchy tree, such as the R-tree [17]. In bounding hierarchy trees, internal (non-leaf) nodes define bounding spaces that minimally enclose their child nodes' bounding spaces, and leaf nodes minimally enclose their associated geometric data. Much research in this area has focused on 3D variants, called Bounding Volume Hierarchies (BVHs), where the bounding spaces are boxes, spheres, or some other volume. BVHs are popular in the field of computer graphics [3].

A downside to data partitioning structures is that updates often are computationally expensive, since modifications of single entries can affect large parts of, or the whole hierarchy to the root (for example if minimum bounding spaces need to be recalculated).



(a) A 2D map representation.



(b) Tree representation.

Figure 3.2: Depiction of a point quadtree with point A as root node. Since point B lies north-west of A and was inserted before point D, it is represented by the first node in A. Point C is north-east, so it is represented by the second node.

3.2.1 Quadtrees

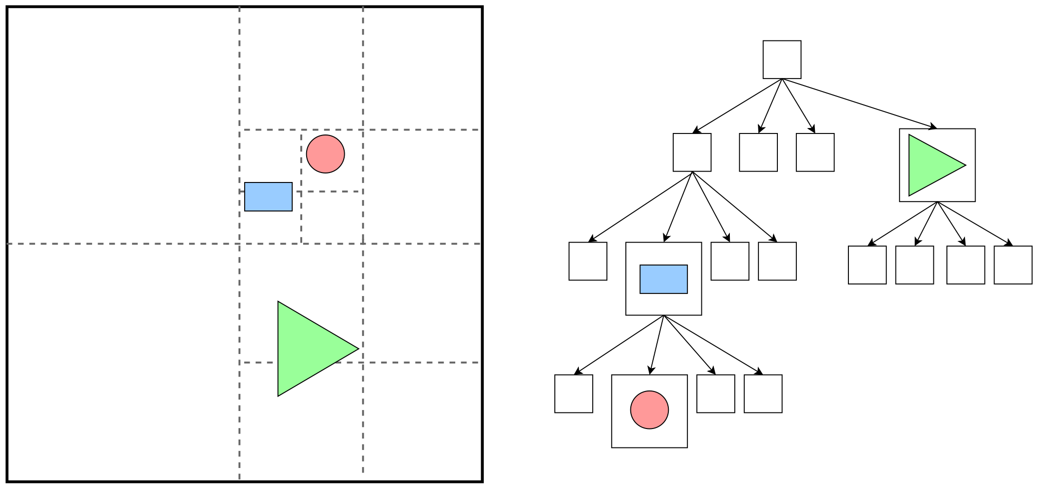
The original quadtree from 1974, called a *point quadtree*, subdivides space into parts based on where a point is placed. For k-dimensional data, the quadtree has 2^k child nodes for each parent node, where a child represents a part of k-dimensional space relative to its parent node and coordinate system axes. While the original definition concerns k-dimensional data, a more common definition of a quadtree is the two-dimensional version, where nodes represent rectangular subdivisions of 2D space. For example, in two dimensions, a node p with point coordinate $\{x_p, y_p\}$ can place another point $q : \{x_q, y_q\}$ in a child based on some predicates:

$$\text{chooseChild}(p, q) = \begin{cases} \text{North-west node,} & \text{if } x_q < x_p \land y_q > x_p, \\ \text{South-west node,} & \text{if } x_q < x_p \land y_q < x_p, \\ \text{North-east node,} & \text{if } x_q > x_p \land y_q > x_p, \\ \text{South-east node,} & \text{if } x_q > x_p \land y_q < x_p \end{cases}$$

Figure 3.2 shows a visual representation of such placement.

Instead of letting the points' placements define the subdivisions' extent and form, quadtrees can also divide the space into four equally sized parts on each level in the tree. In the first case, the layout of the quadtree is dependent on the order of insertion, whereas in the second case this does not matter. Compare Figure 3.2 and Figure 3.3 for visual examples. Whereas nodes in the point quadtree represent points directly, nodes in this other kind just represent an area of space; instead, data is *assigned* to nodes during insertion.

The type of quadtrees which equally divide space usually has a property called the *bucket size*. It decides how much data a leaf node may be assigned before it splits



(a) A 2D map representation.

(b) Tree representation.

Figure 3.3: A depiction of a quadtree that partitions space equally for each level in the tree. Bucket size is one. Objects are placed in nodes which minimally bound them.

and creates children[2]. The bucket size regulates the depth of the tree: the larger the bucket size, the shallower the tree.

Some quadtrees, such as the one in Figure 3.3, allow for geometric data that are not points. Geometric data complicates insertion since data can no longer always be inserted on either side of a line, but instead might overlap one or several nodes. Three main approaches address this: the first is to split the geometry into several parts so that it fits into leaves [3]. The second is to maintain several references to the same object [38]. Finally, the third is to place the object in a node with minimum bounding area [41], meaning that objects can be assigned to internal nodes. The first two approaches need to keep track of each version or reference of an object when it is to be updated or removed, which can become computationally expensive. Therefore, the last approach is most relevant to this thesis; an example of it can be seen in Figure 3.3. Henceforth, the term *quadtree* refers to this variant.

An update operation in the quadtree consists of two parts: deletion and insertion. Both operations traverse the tree to find the node, in which either the old object resides or where the new should be inserted. For a given object, such an algorithm begins by setting the current node to be the root node, and then checks whether any child would enclose the object. If such a node exists, it is set as the current node. This process repeats until no child of the current node encloses the object or the current node is a leaf.

Some literature considers the quadtree to be a space partitioning method [38], because it equally divides the available space into four equal parts at each level, even if there is only data in, for example, two of those parts. However, when a leaf node contains

[2]"The bucket spills over."

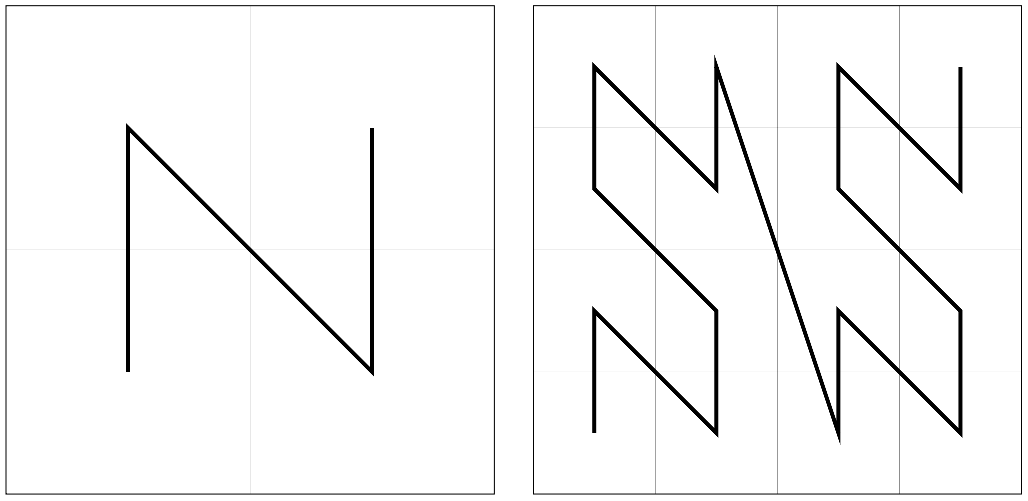

(a) First order **(b)** Second order

Figure 3.4: Two different resolutions of a Morton order curve.

too many entries, it subdivides into smaller parts. The tree will therefore be deeper where the data is highly concentrated. It adapts to the data distribution, which aligns with data partitioning.

3.2.2 Linear quadtrees

A straightforward way to implement a quadtree is to define a node class, construct object instances and keep pointers between these to describe their parent-child relationships. This is referred to as a *pointer-based* or *explicit* representation. Gargantini [15] presents another quadtree representation, which is *pointer-free* or *implicit*: the *linear quadtree*. The idea is to define a unique identifier for each node, and then use this identifier as key in a mapping data structure, referring to a collection of objects associated with the node.

A linear quadtree uses a *Morton order* curve to linearize the search space, that is, to reduce the problem from two or more dimensions to one. A Morton order (also known as Z-order) curve is a path visiting every point in a discrete space exactly once in a certain order, which decides a unique value given to each point, its *Morton value*. Two versions of this curve, with different resolutions, can be seen in Figure 3.4. For a two-dimensional curve, each point's Morton value can be calculated by interleaving the bits of a point's x and y coordinates:

Definition 1 (Morton value)**.** Let x_n be the n-th bit in number x of size k bits, where $0 \leq n < k$. Let a 2D point-coordinate consist of two k-bit numbers $x : x_{k-1}, ..., x_0$ and $y : y_{k-1}, ..., y_0$. The Morton value of this coordinate is defined as the $2k$-bit number: $y_{k-1}, x_{k-1}, y_{k-2}, x_{k-2}, ..., y_0, x_0$.

The Morton value of a node's lower-left corner and its depth can be combined to form its identifier. This identifier is called a *Morton block* [1]. It is a $(2n + l)$-bit long

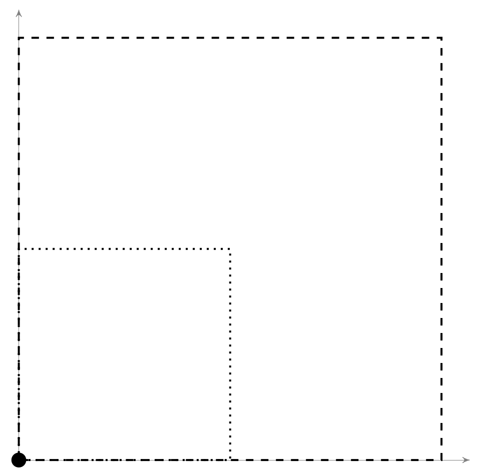


Figure 3.5: An example of a point acting as lower-left corner for two different nodes: the dotted and the dashed square, respectively.

integer, where the corner's Morton value uses the $2n$ upper bits (n bits each for x and y), and the lower l bits are used for the depth. An example of this coordinate-sharing can be seen in Figure 3.5.

Now that each node has a unique identifier, it can be used as a key in a key-value mapping data structure that supports multiple keys, where each entry value stores an object contained in the node. The mapping data structure is traditionally a B-tree since spatial indexes often are used in GIS databases, which commonly use B-trees for indexing general data. The B-tree is a data structure that generalizes the binary search tree in that a node may contain more than two children.

Aboulnaga and Aref [1] proposes a query algorithm which uses the property that keys in a B-tree are sorted. Given a search area and the area of the root node (extent of the world), this algorithm recursively calculates the Morton block for each intersecting node, searches for the nodes in the B-tree, and then performs intersection tests between the search area and the located rectangles.

If a node is entirely enclosed, by transitivity the query area also encloses all nodes inside this node, and these are fetched directly from the B-tree without further recursion. This relies on a specific property of Morton order curves: given a Morton block M_{node}, one can calculate the block M_{max} of the top-right corner of a node. All existing nodes physically enclosed by the node represented by M_{node} will have Morton blocks M, such that $M_{node} \leq M \leq M_{max}$; since all keys are sorted in the B-tree, all nodes enclosed by M_{node} are easily retrieved. See Algorithm 1 for a more detailed description of this query algorithm.

Compared with an explicit representation, the linear quadtree does not require that every node has a parent[3]. This contrasts with a pointer-based representation, where pointers indicate clear relationships. A linear quadtree, on the other hand, could consist of a root node and a small node 30 levels down, with no parent-child

[3]Of course, the root node has no parent in either explicit or implicit representation.

Algorithm 1 A window query algorithm for linear quadtrees. The algorithm takes as input a query area and a node. The first call uses the root node as input. Nodes are represented by their lower-left corner coordinate and width.

```
procedure WindowQuery(area, node)
    $M_{node}$ ← Calculated Morton block of node;
    if area fully encloses node then
        $M_{max}$ ← Morton block of upper-right corner of node;
        Fetch all rectangles from B-tree with key $M$, s.t. $M_{node} \leq M \leq M_{max}$;
        for each fetched rectangle do
            Add to result set;
    else
        Fetch all rectangles from B-tree with Morton block $M = M_{node}$;
        for each fetched rectangle do
            Add to result set if rectangle intersects with area;
        Calculate child nodes;
        for each child node do
            if child node intersects area then
                WindowQuery(area, child node);
```

chain in between. Executing Algorithm 1 may therefore result in many unnecessary calculations and B-tree searches for nodes that do not exist. It must search the B-tree for any intersecting node that may or may not exist; however, in an explicit representation a query would stop where the tree ends — at the leaf nodes.

3.2.3 Loose quadtrees

The loose quadtree, proposed by Ulrich [47], is a modification of the quadtree where nodes' areas are expanded so that they overlap with their neighbors. Given a node with width w, the area is expanded by a constant factor p; the expanded width w_e is then $w_e = (1 + p) \times w$. An object with a width greater than w, which previously would have to be placed higher up in the tree, but smaller than w_e, can now be contained in the expanded node. Just as in a standard quadtree, an object is assigned to the node that minimally encloses it; however, the center point of the object must still be within the original, unexpanded node area.

A query performs intersection tests with all objects contained by any node it visits. For example, if a small object is placed in the root node due to crossing a boundary between the children of the root node, all queries will perform intersection testing against this object, even though this object may not be anywhere near the query area. By loosening the area of each node, objects are stored deeper in the tree and the amount of unnecessary intersection tests is reduced; see Figure 3.6 for an example. This is the main benefit of using loose quadtrees compared with regular quadtrees, according to Ulrich.

While having a deeper tree improves query performance, loosening the quadtree also has downsides. With loosened areas, more nodes will overlap with the search

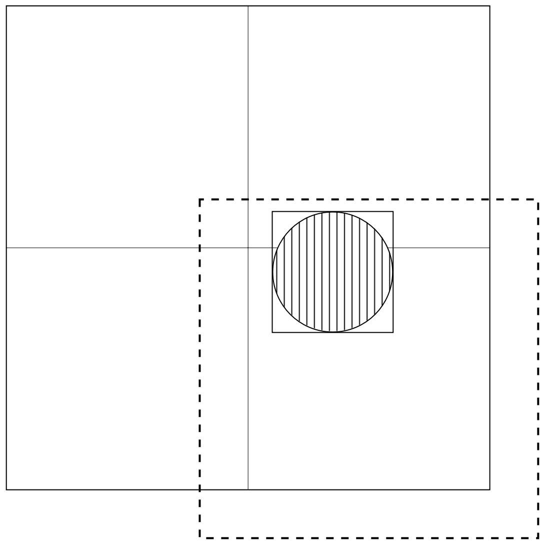


Figure 3.6: A loose quadtree in two dimensions. The lower-right node's expanded area is represented by the thick dotted line. The striped circle is represented by its bounding rectangle, which crosses the boundary of two quadtree nodes. Since the circle's center point lies inside the lower-right node and the rectangle is enclosed by the same node's expanded area, the circle is assigned to this node.

area, meaning that more paths may need to be traversed. It is also the case that unexpanded node areas that previously were fully contained by a query may not be so when loosened, with the consequence that nodes further down may have to be visited.

Loosened node areas also affect update performance. As the expansion factor increases (which yields larger expanded node areas), there will be fewer potential nodes where an object would minimally fit. Ulrich [47] shows that for an expansion factor of 1, there are at most two nodes that minimally enclose an object[4]. Samet et al. [41] generalizes this to any factor greater than or equal to 0.5.

Samet et al. provide an algorithm that calculates all possible minimally enDclosing nodes for a given object based on the object's position, width, and the expansion factor of the loose quadtree[5]. The algorithm is based on the assumption that the root node has a width of 2^g [6], $g \geq 0$. Since the width is halved at each level, all nodes in the tree will have widths that are powers of two: 2^k, $k \leq q$.

The minimally enclosing node for a given object is decided by first calculating all possible minimally enclosing nodes for an object, and then iterating through these nodes, starting with the smallest, until a node fully encloses the object, in which case

[4]Compared to Ulrich's loose quadtree, in a regular quadtree, any node with width larger than or equal to the width of an object can be a minimally enclosing node; the reason for this is that an object may cross any boundary between nodes, and therefore be assigned to the first node enclosing this boundary, which may be much larger than the object.

[5]With an expansion factor greater than or equal to 0.5, there are at most two possible node that enclose an object and the algorithm runs in constant time.

[6]The reasoning behind choosing this width is explained in detail by Samet et al. [41].

the algorithm has found the minimally enclosing node. This algorithm can be used both for inserting a new object and for finding an already existing object that will be deleted. See Algorithm 2 for a step-by-step process, a slightly simplified version of the one described by Samet et al [41].

Algorithm 2 An algorithm for deciding which node a rectangle minimally fits in. The "inside" call determines whether the rectangle is inside the given node or not. See Samet et al. [41] for more detail.

```
procedure FindNode(rectangle)
    $p \leftarrow$ expansion factor
    if rectangle.x > rectangle.y then
        $r \leftarrow$ rectangle.x/2
    else
        $r \leftarrow$ rectangle.y/2
    start $\leftarrow \log_2(\mathcal{M}(\frac{1}{p+1}))$;          $\triangleright\ \mathcal{M}(x) = 2^k$ such that $2^{k-1} < x \leq 2^k$
    end $\leftarrow \log_2(\mathcal{M}(\frac{2}{p})) - 1$
    for $i \leftarrow$ start to end do
        $w \leftarrow 2^{i+1} * \mathcal{M}(r)$;       $\triangleright$ $w$ is the width of the node at the current level
        $x_{lower-left} \leftarrow$ (rectangle.center.x / $w$) $* w$          $\triangleright$ Integer divisions
        $y_{lower-left} \leftarrow$ (rectangle.center.y / $w$) $* w$
        if rectangle.inside($w$, $x_{lower-left}$, $y_{lower-left}$) then
            exit loop
    return Node($w$, $x_{lower-left}$, $y_{lower-left}$);
```

3.2.3.1 Loose-linear quadtrees

Combining linear and loose quadtrees results in what we call a *loose-linear quadtree*. Assume expansion factor $p \geq 0.5$ so any object has at most two possible locations in a quadtree. Linear quadtrees allows fast insertion of objects to a given node using the B-tree, which supports a much higher branching factor compared to a quadtree and therefore a much shorter traversal through the data structure [41]. Combining a linear quadtree with looseness and the constant time lookup algorithm makes it possible to calculate which node an object belongs to, calculate the Morton block value of this node, and then quickly access the node in the B-tree.

3.2.4 R-trees

Another approach that has been the inspiration for many spatial indexing structures is the R-tree [17], which is widely used in databases [28]. Originally an extension of B-trees [5], it was introduced by Guttman in 1984.

The R-tree is a hierarchical tree where each node has a set of *entries*, E, where $m \leq |E| \leq M$. m and M are two fixed values for the minimum and maximum amount of entries in a node, respectively. It is also a strictly height-balanced tree, such that all leaf nodes are on the same level (the same path length from the root). Data entries are contained in leaf nodes and represent geometrical objects and their

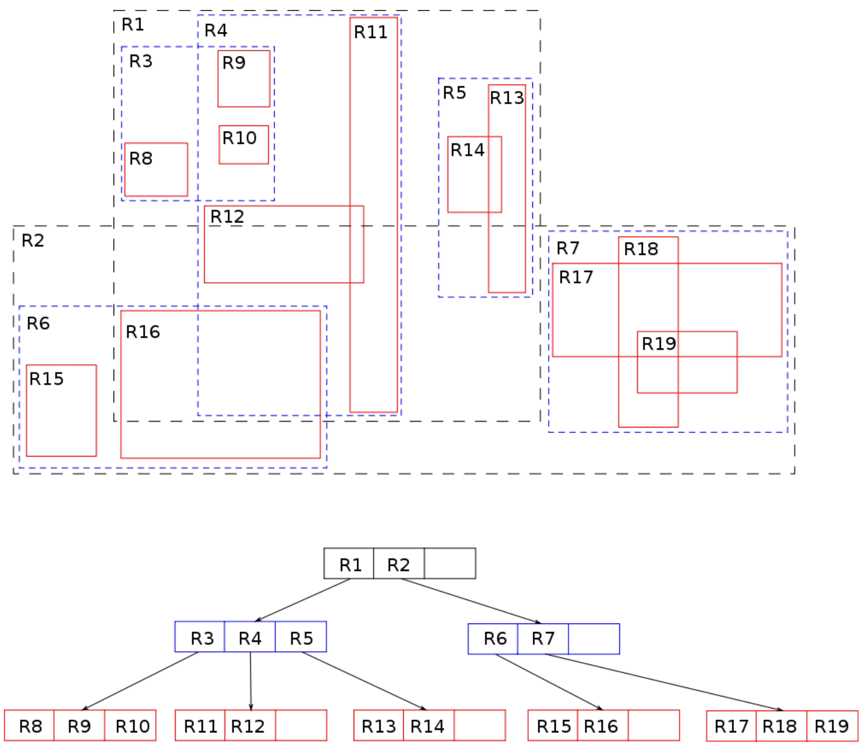


Figure 3.7: A 2D R-tree. Here every node contains at least two, and at most three, entries. The leaf nodes contain entries R8-R19, which represent minimum bounding boxes around geometry not visible in this image. Overlap between entries in the same node is clearly visualized in this image, where R1 and R2 are sharing large areas. The same is true for R3 and R4, which both completely enclose R9 and R10, leading to a traversal down both R3 and R4 when searching for these. The image is in the public domain.

minimum bounding rectangles (MBRs). For internal nodes, an entry contains a child node as well as an MBR enclosing the child node's entries. An example of a 2D version of the R-tree can be seen in Figure 3.7.

Data entries are inserted by starting at the root and recursively inserting into whichever node needs the least expansion of its MBR, until a leaf node is found. If the node contains more than M entries after insertion, it is split into two new nodes according to a chosen split algorithm[7]. This is done recursively until each node contains equal or less than M entries. Afterwards, affected MBRs are recursively recalculated so they contain their corresponding entry.

To remove a data entry, its entry is first located in the tree and then removed from the leaf node. If there are more than m entries left in the node, the MBRs are recalculated up the tree. If there are fewer than m entries left in the node, the remaining entries are put on a separate stack and the node is removed. This is then done recursively until each node has at least m entries. Afterwards, entries in the stack are reinserted from the top of the tree, each inserted into its corresponding level so that the tree is still height-balanced.

Updates in the R-tree consists of a removal operation on the old entry and an insertion operation of the new entry.

Two factors negatively impact R-tree query performance: node overlap and MBR stretch. It is common in R-trees that MBRs of entries that are on the same level in the tree overlap, which means that more branches may have to be traversed by the query algorithm [6]. The MBRs also tend to be more stretched in one dimension in some cases, which means that entries whose MBRs lie far apart still belong to the same node. Both these issues are addressed by the R*-tree, described in Section 3.2.5.

3.2.5 R*-trees

In 1990, Beckman et al. presented a modified version of R-tree which they dubbed R*-tree [6]. They found that in order to improve the R-tree's query performance, several factors need to be considered:

1. Overlap of nodes should be minimized. Overlapping negatively affects query performance since several subtrees needs to be traversed where there is overlap.
2. Entries in a node should be close to each other, so if the query area intersects with a node, there is a high probability that its entries also intersects with the query area. This is the same as minimizing node area.
3. Queries often have a shape that is close to a square, so node MBR's should have a similar shape. If nodes have MBR's that are shaped as thin, elongated stripes, more nodes need to be visited during a query.

An example of the differences between R-tree and R* for a dataset can be seen in Figure 3.8.

The original R-tree was only optimized to reduce node area during insertions and splitting. R*-tree tries to reduce enlargement during insertion when choosing internal nodes (factor 2), but minimizes overlap when choosing leaf node (factor 1)

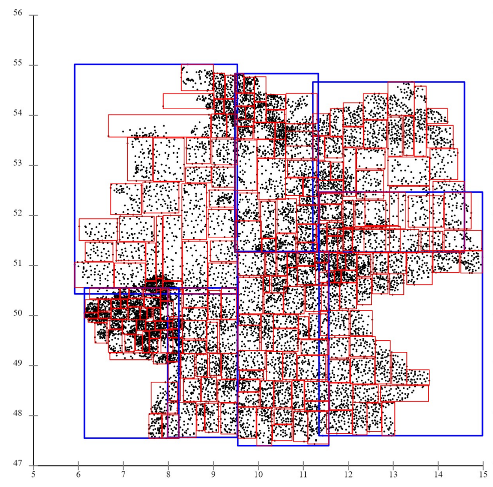
During splits, which is the operation that perhaps affect the tree the most, Guttman's R-tree tries to reduce the total amount of area by calculating which pair of rectangles results in the largest area, creates two new nodes and places one in each. The rest of the entries are placed in these nodes according to one of two proposed algorithms, which differ according to their runtime complexity.

R*-tree splits are performed differently compared to R-trees. When splitting an overfull node, a split axis is chosen first. This is done by calculating sums of margins[9] for different combinations of entries for each axis and choosing the axis which has the lowest total sum. This favors choosing an axis that results in a split into two nodes that are as square-shaped as possible (factor 3). From these different possibilities, it

[7]Guttman [17] describes two algorithms to partition the set of entries into two nodes, a linear cost split and a quadratic cost split, which refer to the running complexity with respect to maximum number of entries for a node (M).

[8]Both images were made by Wikimedia user Chije under Creative Commons Attribution-Share Alike 3.0 Unported license (link).

[9]The margin of a rectangle is the sum of all sides.


(a) R*-tree

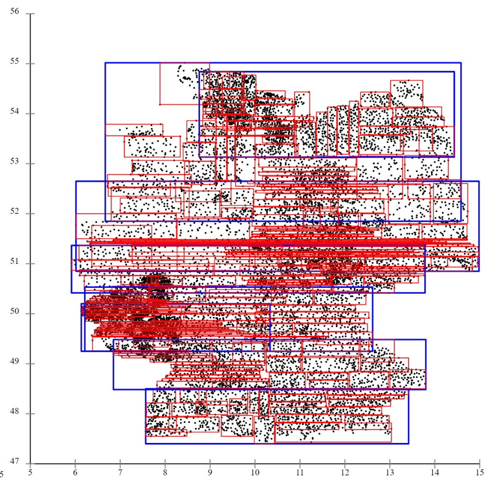

(b) R-tree with Guttman's quadratic split

Figure 3.8: Comparison of two R-tree versions. The data is zip codes in Germany.[8]

chooses the one with least overlap (factor 1). If there are several possibilities with the same overlap, the one with combined minimum area is used (factor 2).

In R*-trees, *forced reinsertion* is used. Beckman et al. found that re-inserting a portion of the objects after the index had been constructed resulted in better query performance. Therefore, if a node becomes overfull during ordinary insertions in R*-tree, it does not split. Instead, a percentage of the entries are re-inserted from the top. Forced re-insertion is not done except for when inserting an entry or updating an old one, since it has a slight performance penalty.

3.3 Update techniques

Some of the spatial indexes discussed in this chapter have variants that focus on improving update performance. Two approaches are relevant to this thesis: *bottom-up updating* and *update memo*.

3.3.1 Bottom-up updating

Bottom-up updating is a category of methods which are based on the idea of maintaining a mapping between a specific object and the node it is associated with in the spatial index. When updating an object in the index, the object's current node is accessed directly by using this mapping. Originally proposed for R-trees by Lee et al. [26], it is called bottom-up updating because updating begins from the leaf — the bottom — where the object resides.

A traditional update first locates the old entry in the tree by search traversal, which begins at the tree's root, deletes the old entry and then inserts the new entry from the top [17, 23]. The efficiency issues with this top-down method are three-fold:

1. It is likely that delete and insert search paths will overlap to some extent [26]. This path will then be traversed twice.
2. The search starts from the root, but objects in structures such as R-trees are always stored at the leaf nodes [26], leading to logarithmic access time for the object that will be deleted.
3. The MBRs of R-tree nodes may overlap, which means that the search to find the old object might traverse several paths even if it only searches for a specific entry [45].

Bottom-up updating uses a secondary index — a key-value mapping where the key is a unique object identifier and the value is a pointer to the current node or cell where the object resides. Both Lee et al. [26] and Šidlauskas et al. [44] implement the secondary index as hash table, but any mapping data structure can be used. The update algorithm consists of two parts, deletion and insertion: first, find the position of the old object using the secondary index and delete the object from the pointed node.

The second part, insertion, varies by implementation. Lee et al.'s [26] algorithm looks for an appropriate sibling node in which to insert the updated object. A different insertion method for R-trees, presented by Šidlauskas et al. [44], ascends the tree until it encounters a node with an MBR that encloses the new position of the object, and then traverses down from that node in top-down fashion until it reaches a leaf.

Some problems with bottom-up updates are discussed by Silva et al. [45]. Specifically, the performance gain compared to a top-down approach diminishes when objects move far, leading to searches for nodes far away in the tree. Also, a secondary index with one record for every object contained in the primary spatial index yields a large memory overhead.

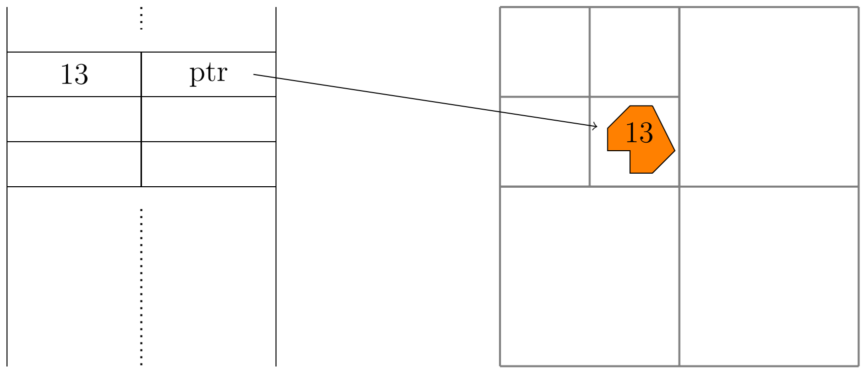


Figure 3.9: An example of bottom-up updating where a key-value mapping is used to directly access objects. Here the object identifier 13 is mapped to a node in a quadtree.

3.3.2 Update memo

In R-tree variants, deletions are considered more expensive than insertions. An insertion must only traverse one path in the tree to choose an appropriate leaf node;

deleting an object, on the other hand, requires searching for it, which means that several paths might be explored since entries can overlap. Deletion also leads to underfull nodes, which are deleted and restructured, further increasing execution time.

In 2009, Silva et al. [45] proposed the RUM-tree, *R-tree with Update Memo*. The goal with update memo is to avoid deletions when updating existing objects by simply inserting the updated objects. An effect of this is that multiple versions of the same object may be present in the spatial index at the same time. By not deleting an object's old entry before inserting the new one, the structure must keep track of which entries are obsolete and which are current. This is accomplished by using a secondary index, which keeps track of the objects that have multiple entries, and which entry is the current one. To separate entries of the same object, a *stamp counter* is maintained: a number that is incremented with each modification (insertion, deletion, update) to the spatial index. When inserting an object, the current stamp counter value is assigned to the new entry in the index. The secondary index is therefore a mapping, with an object's identifier as key:

$$\texttt{obj} \rightarrow \{\text{stamp value of latest entry, number of obsolete entries}\}$$

To update an object, a new entry is either added to the secondary index, if none already exists[10], or has its amount of obsolete entries in the secondary index incremented. The new entry is marked as the current one by setting the current value of the stamp counter in the object in the secondary index. It is then inserted like an ordinary insertion. Since the stamp counter is always increasing, an older entry will always have a lesser stamp value than the one saved in the secondary index.

Queries on the spatial index yields obsolete as well as current objects. This means that the stamp value of each object must be checked: if it is less than the latest stamp value in the secondary index's entry, the object is obsolete and can be discarded.

Obsolete objects are deleted eventually, with a collection of methods called *garbage cleaning*. The common idea of garbage cleaning methods is to access objects in the spatial index, compare their stamp values with their respective entries in the secondary index, and remove those that are obsolete. Silva et al [45] propose two relevant garbage-cleaning methods: *token-cleaning* and *clean-upon-touch*.

The token-cleaning method relies upon using *token* objects. A token is an object assigned a collection of nodes that it will clean; if only one token exists, this collection will be all nodes that can contain objects in the tree (leaf-nodes in R-trees, any node in quadtrees). Every I updates to the tree, one node in each token's collection is cleaned of obsolete objects. This is performed by iterating through all tokens and clean the nodes each has currently selected. When the node has been cleaned, another node in each token's collection is chosen for the next cleaning call; this selection is done in round-robin order. The token-cleaning method requires a parameter: a cleaning interval I, which defines how often this process should execute.

Token-cleaning can be combined with the clean-upon-touch method, where the idea is that each accessed node during insertion or update is cleaned of obsolete entries.

[10]An object has no entry in the secondary index if it has no obsolete entries in the spatial index.

Clean-upon-touch is extra useful when the data resides on secondary memory (disk), as the accessed data has been fetched to main memory, so cleaning it will not incur additional I/O system calls.

Some positive effects of using update memo include:

- Deletion of an object consists of incrementing the stamp counter value[11], setting it as the new stamp value in the update memo entry of the object, and then incrementing the number of obsolete entries by one. No traversal to search and delete the old object is necessary; instead, deletion is handled by delayed garbage cleaning.
- An update operation is in theory almost as quick as an insertion operation since the cost of deletion is postponed.

However, there is an issue with using update memo on dynamic spatial indexes that perform splits and merges on their nodes. When increasing the ratio of data that is obsolete, more of the index depends on the obsolete objects for its structure. This means that unnecessary merging and splitting may occur, which degrades performance. One such example is an R-tree leaf node with maximum number of entries where almost all are obsolete, and a new object is inserted. Even though it would fit in the node in a regular R-tree, in the RUM-tree, a splitting operation would commence.

Other versions of update memo structures include the RUM+-tree [49], which is an improvement upon the RUM-tree; and G-ML-Octree [10], a loose octree with update memo.

It is possible to combine bottom-up updating with update memo, which is the idea behind the RUM+-tree. When used together, bottom-up is used to directly access the node where the object currently resides, to check whether the object's new position belongs to the same node. If it does, the object's data is updated in-place; if it does not, update memo is used to insert the new object.

[11]This value is typically stored as a 64 bit unsigned integer, which would require 2^{64} updates to overflow. This limit will in practice never be reached.

4

Method

The goal of this thesis is to compare different spatial indexes that maintain moving geometry data, with focus on update performance for many dynamic objects. This chapter describes which spatial indexes were chosen and why (Section 4.1), as well as implementation details (Section 4.2). We also describe the test suite (Section 4.3) and evaluation process (Section 4.4).

4.1 Evaluated spatial indexing structures

The following spatial indexes are chosen for evaluation:

- **List**. A simple and easy to implement structure for managing a collection of data, in this case rectangles. A query must iterate through the whole list to find all intersecting objects, an $\Theta(n)$ operation. Updating an object's position would have the same complexity, but we choose to use a *bottom-up* approach for updates; which has $\Theta(1)$ complexity. We choose the list to establish an upper-bound for query performance; if a data structure performs as well as — or worse than — the list, there is no reason for using it over the list.
- **Simple quadtree**. A quadtree where every node maintains explicit pointers to other nodes and there is no regard to cache efficiency. This means that nodes are heap-allocated by themselves and may not reside close in memory. The quadtree is interesting from a comparison point of view, as both loose and loose-linear quadtrees are more complex in their construction; if performance is similar, it is preferable to use a simple implementation.
- **Loose Quadtree**. Compared to the simple quadtree, every node's area is expanded by a factor. This version will also consider cache efficiency by storing nodes together in memory. We choose the loose quadtree because it is an established method for indexing moving spatial geometry. It may yield better update performance than a regular quadtree since there is a larger probability that objects will be enclosed by the expanded node areas.
- **Loose-linear quadtree**. Since it is possible to calculate which node a rectangle belongs to in $\Theta(1)$ time with a loose quadtree, and to quickly index a specified node in a linear quadtree, the combination of these should perform well.
- **R*-tree**. A different approach than the others and a true data-partitioning structure, the R*-tree is a interesting candidate and widely used in GIS settings.

Carmenta supplied an implementation which we extended.

The last three candidates: loose quadtree, loose-linear quadtree, and R*-tree, all have variants implemented with bottom-up updating and update memo, and both methods combined.

4.2 Implementation

All chosen spatial-indexing structures are implemented in C++ 17.

For implementing the list, we use a dynamic array from the C++ standard library: `std::vector`. A mapping data structure, `std::unordered_map`, maintains a mapping directly between rectangle object identifiers and their current index in the array; this map is used when updating the rectangles' positions.

4.2.1 Update techniques

All implementations of bottom-up updating and update memo described in this chapter are approximately the same. Differences to the implementation described in this section are discussed in the respective implementation sections for each spatial index.

To implement bottom-up updates, the spatial index is extended with a key-value mapping (implemented with `std::unordered_map`): the *bottom-up index*. For each tracked rectangle, the bottom-up index has an entry where the rectangle's identifier is used as key, and the value is either a pointer or an index value to the node that contains the rectangle.

All bottom-up implementations use so-called *in-place updating*, which means that when the node containing the old object is accessed, the algorithm checks whether the new object belongs to the same node. If it does, the object is updated in-place, without any more node traversal. Otherwise, the algorithm performs regular insertion top-down.

4.2.2 A simple quadtree

This section describes an implementation of a regular, pointer-based, quadtree. The class definition can be seen in Listing 1. It maintains a pointer to the root node, the maximum number of rectangles in a leaf node (bucket size), and the maximum depth of the tree. Maximum tree depth is needed to catch edge cases of the data distribution; for example, many small overlapping objects may result in an unnecessary deep tree.

In this implementation, any node, leaf and internal, can contain rectangles. A rectangle's size, position, and the tree's bucket size decide which node it is placed in. During insertion, beginning at the root node, the rectangle is checked against each child of the current node and the one that it is fully enclosed by is set as the next current node. This procedure repeats until no child fully enclose the rectangle, or it reaches a leaf node, in which case the object is assigned to the current node. In the

```
class QuadTree {
    QuadNode * root;
    const size_t max_bucket_size;
    const size_t max_recur_depth;

    bool insert(Rectangle r);
    bool remove(Rectangle r);
    bool update(Rectangle r_old, Rectangle r_new);
    std::vector<Rectangle> intersects(Rectangle query_area);
}
```

Listing 1: Class definition for a simple quadtree.

```
class QuadNode {
    const Rectangle area;
    bool leaf;
    const uint8_t level;
    std::vector<Rectangle> rectangles;

    QuadNode * parent;
    QuadNode * topLeft;
    QuadNode * topRight;
    QuadNode * bottomLeft;
    QuadNode * bottomRight;
}
```

Listing 2: The class definition for a node in the simple quadtree.

case where it is a leaf node, the number of rectangles associated with the node is compared with the bucket size; if there are more, the node is split and the rectangles are assigned to the appropriate child (or stays in the node, if no child enclose the rectangle).

The node class for the quadtree is specified in Listing 2. Each node contains several variables: the node's spanning area, whether it is a leaf or not, on which level in the tree it exists, a vector containing the rectangles associated with it, and pointers to its child nodes and parent node. The pointer to the parent is kept to ease *merging* of nodes; when a leaf node becomes empty, a procedure is called on its parent, which checks whether $n \leq k/2$, where n is the number of rectangles in the children and k is the number of empty slots in the parent. If that is the case, the child nodes are removed, their rectangles moved to the parent, and the parent becomes a leaf.

The `level` variable in the `QuadNode` class is used to prevent an overfull leaf node from splitting, if its depth is the maximum depth of the tree (defined in Listing 1). The variable also identifies the root node, which has a depth of zero.

4.2.3 Improving the quadtree

One straight-forward approach to improve the simple implementation is make it more cache-friendly. To improve cache efficiency, data used frequently together should be physically close in memory, and, if possible, fit in a cache line. Sibling nodes are often accessed together during operations, for example when choosing traversal directions during insertion, so a more cache friendly approach is to group them together as one object. Then it is also possible to maintain shared data without duplicating it for each node object. This representation, called a *chunk*, can be seen in Listing 3.

```
class QuadNode {
    uint32_t children_index; // The chunk with the node's children
};

class ChunkIndex {
    uint32_t index : 30;
    uint32_t offset : 2; // Specifies child node
};

template <typename T> // float or double
class Chunk {
    uint8_t depth; // Depth where the nodes reside
    T x1, y1, x2, y2; // Area enclosing the nodes
    ChunkIndex parent_index; // Index to their parent's position
    QuadNode[4] nodes; // The four nodes
};
```

Listing 3: Chunk and node representation. The `ChunkIndex` class has *bit fields*, which in this case will limit memory usage to 32-bits for each instance, where the `index` field occupies 30 bits and `offset` two bits. The 32-bit field in the `Node` class references a whole chunk, so only 30 bits are used, but 32 bits must be occupied.

The chunks of a quadtree are kept in a vector addressable by an unsigned 30-bit integer[1], leaving two bits for indexing a specific node in the chunk (four nodes). This means that an unsigned 32-bit integer (`ChunkIndex` datatype; Listing 3) can be used to index any node in the tree: 30 bits for the chunk the node belongs to, and two bits for the specific node. For this thesis, it will be enough to use 32 bits to index the vector, as it can fit about four billion nodes, or one billion chunks.

With this indexing method, the number of bytes a chunk will occupy in memory is

[1]The vector's length must be less than or equal to the maximum value of the 30-bit unsigned integer.

(`field_name` : number of bytes):

$$\begin{aligned} \texttt{depth} &: 1 \\ \texttt{parent\_index} &: 4 \\ \texttt{nodes} &: 4 \times 4 = 16 \\ \texttt{area} &: 4 \times size(\texttt{T}) \\ &\rightarrow \\ 21 + 4 &\times size(\texttt{T}) \text{ bytes} \end{aligned}$$

If `T` is a `float` ($size(\texttt{T}) = 4$), the chunk occupies 37 bytes, and 53 bytes if it is a `double` ($size(\texttt{T}) = 8$). Both these sizes are below 64, the most common cache line size, so if the data is aligned[2] to 64 bytes, each cache line will fit one chunk.

Each node's vector of rectangles is not stored in the `Node` or `Chunk` classes; instead, a vector of vectors is maintained in the main loose-quadtree class that contains these vectors. The position of a node's vector in it is the same as the node's `ChunkIndex` index in the vector of chunks.

The motivation behind using a representation like `Chunk` is that the comparison between quadtrees and R-trees becomes more representative. In R-trees, entries in the same node are frequently used together, and only need to be fetched from memory once, while the simple quadtree's pointer-representation allocate each child of a parent by themselves, even though they almost always are accessed together. Thus, by putting these nodes together, the memory layout becomes more like that of an R-tree, which means that the comparison results between them depend less upon the implementation, and more on how the data structures function.

4.2.4 Loose quadtree

The explicit loose quadtree is based on the improvements described in Section 4.2.3. Just like the simple quadtree (Listing 1), the loose quadtree uses a bucket size and a maximum depth.

An object is inserted by traversing to nodes that enclose its center point and whose expanded areas fully enclose it, starting at the root. Traversing then stops when none of the current node's children enclose the object. This is also the procedure when an object is deleted from the tree, for finding the node where it is currently placed.

The merging process for the loose quadtree is the same as the naïve quadtree: remove four leaf nodes if $n \leq k/2$, where n is the number of rectangles in the children and k is the number of empty slots in the parent.

4.2.4.1 Update techniques

Bottom-up updating is implemented as described in Section 4.2.1. The loose quadtree uses in-place updating for both the bottom-up and top-down variants.

[2]The `Chunk` class is padded at the end using `alignas` so that it occupies 64 bytes.

Update memo was originally proposed for, and applied to, R-trees, where only leaf nodes contain the indexed objects. Our version of the loose quadtree, however, may assign objects to any node in the tree, leaf or otherwise, which complicates the adaptation of update memo techniques.

We adapt token-cleaning to the loose quadtree by letting all nodes, not just the leaf nodes, be assigned to tokens. While the token-cleaning concept allows for an arbitrary number of tokens, we only implement one special case: each chunk (a parent's nodes) is assigned its own token. Implementing more general assignment of nodes makes sense when one can assume how the data distribution changes over time (one can predict which nodes will receive many updates), but as no such assumptions are made, this is not implemented.

Clean-upon-touch is also non-trivial to translate to the loose quadtree, for the same reason as above. The options are to either clean the nodes assigned the old and new entries only, or all nodes accessed during traversal. For R-trees the choice is trivial because internal nodes never contain any objects, and the only leaf nodes accessed are the ones that contain the old and new entries. We implement the former for the loose quadtree: if an object remains in the same node during an update, this node is cleaned; if the object is inserted elsewhere, the new node is cleaned, but no nodes along the way. Cleaning all nodes along the way during traversal would slow down updates significantly; nodes closer to the root are also accessed more frequently, meaning that they will be cleaned unnecessarily often.

We propose a complementary update-memo technique, called *clean-chunk*, where cleaning is dependent on the number of insertions to each chunk. If a tree's bucket size is x, cleaning proceeds when $x/4$ objects have been inserted to any node of a chunk. One of the four children is cleaned, in round-robin order. We also combine clean-chunk with traditional token-cleaning; for this no additional logic is needed, as they are separate methods that work on their own. The result should be that clean-chunk keep often used nodes from overfilling with old entries, while token cleaning regularly removes old entries from less used nodes.

4.2.5 Loose-linear quadtree

The implementation of the loose-linear quadtree is based on the find-node algorithm (Algorithm 2, Section 3.2.3) by Samet et al. [41] and the query algorithm (Algorithm 1, Section 3.2.2) by Aboulnaga and Aref. [1]. Updating is implemented as two calls to Algorithm 2, one for finding the node for removal, and one for insertion. For the B-tree we use a third-party library: "cpp-btree"[3].

In the bottom-up variant, the direct mapping replaces Algorithm 2 for the deletion part of updating.

For the loose-linear quadtree, it is not possible in practice to implement token-cleaning or clean-upon-touch as they are described by Silva et al. [45]. For a quadtree with explicit representation, there exist a multitude of intuitive approaches for deciding how the nodes should be partitioned to tokens. The loose-linear quadtree, however,

[3]https://code.google.com/archive/p/cpp-btree/

has no parent-child relationships, which makes it difficult to identify an appropriate partitioning scheme. Therefore, the cleaning procedure in the loose-linear quadtree is implemented as a simple iteration through all nodes in the B-tree, where nodes with obsolete entries are deleted.

Implementing clean-upon-touch is problematic, since an existing node is almost never accessed during insertion or updating (except if deleting the old rectangle's node), which is the whole point of clean-upon-touch.

4.2.5.1 Query algorithms

We implement several different query algorithms, all based upon Algorithm 1. These variants exist to address several problems with the original algorithm:

1. The algorithm is recursive, but the only stopping condition for the recursion is when a node is fully enclosed by the query area; this may not happen for many recursions if the borders of the node and query are very close.
2. Recursion will continue from an intersecting node even if no sub-nodes exist in the B-tree.
3. Recursion will continue from an intersecting node even if the smallest rectangle in the tree cannot possibly be contained by any sub-nodes (i.e., the recursion reaches the currently smallest node size in the tree).
4. If the largest rectangle is enclosed by a node on tree depth x, the algorithm will still search the B-tree for all nodes from root to depth x even though no such nodes exist in the B-tree.
5. In a case where there exist very small and very large rectangles, but no rectangles of sizes in between, there will be several levels in the tree for which no nodes exist in the B-tree. The algorithm will still search for these nodes.

To mitigate problems 2-5, two extended algorithms are proposed: WindowQueryRoot and WindowQueryOffset. Both address problems 2, 3, and 5 the same way but differ in their approach to problem 4. Unfortunately the first problem is inherent to the algorithm since it is recursive and can therefore not be fixed, but its effects can be limited by solving problems 2, 3, and 5: solving problems 2 and 3 will limit the number of recursions and while solving problem 5 does not limit recursion, it potentially speeds it up by not searching the B-tree on certain levels.

The second problem can be fixed by making use of the property that all possible sub-nodes enclosed by node M_{node} will have Morton block values between M_{node} and M_{max}, which can be calculated from the top-right corner of M_{node}. The B-tree maintains its keys in sorted order, which means that it is possible to check what the next Morton block after M_{node} is in the B-tree; if it is larger than M_{max}, no sub-nodes fully enclosed by M_{node} exist in the B-tree. This means that it is not necessary to traverse further down this path.

Problem 3 is solved by maintaining the currently smallest node's size in the tree. The query algorithm then stops recursion at this level. A similar solution is applied to problem 5, where an array of numbers is used to maintain the number of nodes

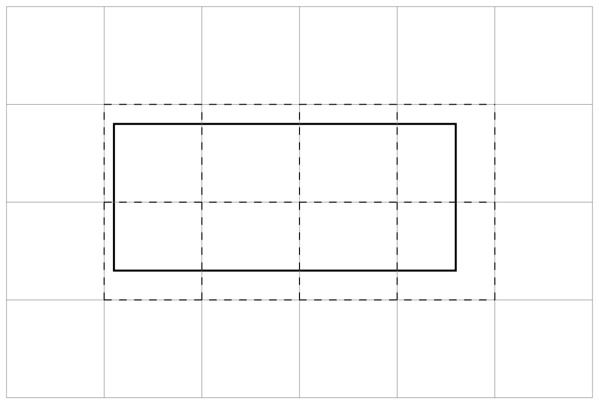


Figure 4.1: Depiction of the partitioning step of the WindowQueryOffset algorithm. A query area (solid line) is partitioned into node areas (dashed lines) of the same size that the largest object in the tree is assigned to. Each partitioned area is then used as initial input to the recursive query algorithm.

that exist on a certain level. The level is used as index in the array. If the number for a certain level is zero, meaning that there are no nodes at this level, no search in the B-tree is performed.

The problem where our extended algorithms differ concerns the fact that the original algorithm searches the B-tree for all intersecting nodes from the root, even if the largest object is contained by a node many levels below. Both solutions maintain the depth of the currently largest node. From this information, the simplest approach is to continue the recursion as usual, but not search the B-tree until reaching the depth of the largest node. This is the approach of WindowQueryRoot. In contrast, WindowQueryOffset begins recursion at this depth. This is done by splitting the query area into node areas of the size that contains this object, and then executing the regular query algorithm (with the other extensions) with each area as initial input. An example of this can be seen in Figure 4.1. An advantage of using this method instead of WindowQueryRoot is that recursion will skip initial levels that contain no data. However, WindowQueryRoot can take full advantage of another effect that WindowQueryOffset can not: larger query areas may fully enclose nodes that are larger than the currently largest in the B-tree, which means that the other branch (enclose-branch) of Algorithm 1 is executed. This results in the algorithm fetching all sub-nodes without any further recursion or intersection testing.

In addition to the two proposed query algorithms, we implement the original algorithm from Aboulnaga and Aref [1], here called WindowQueryPaper. Only one small alteration is done on this algorithm compared to the original: the recursion stops when the maximum depth of the tree is reached. We also implement another algorithm, WindowQuerynaïve, which simply fetches all nodes stored in the B-tree and performs intersection testing on all associated objects.

4.2.6 R*-tree

The R*-tree implementation is based on an incomplete library supplied by Carmenta which only has support for insertions and queries. We extend the library with deletion as described in the original R-tree paper [17]. Updates are also implemented as described[4], with the addition of *in-place updating*, that is, if the new entry fits in the same node as the old entry, it is updated without further recursion. Insertion is also as described, but with the minor difference that node splits are done directly when a node becomes overfull; not when inserting an element to an already full node. We expect this to have little, if any, impact on performance.

In the implementation, three main classes are defined:

- The Tree class (see Listing 4), which contains a pointer to the root node as well as the *re-insertion handler*. This class also represents the public interface.
- The Node class (see Listing 5), which contains a bounded amount of entries, at least m and less than M. The node class contains most of the program logic.
- The Entry class (see Listing 6), which contains either a pointer to a node class, or data (in our case an object identifier). It also contains the MBR for the contained node or data.

```
template<class CT, class D, int NC>
class RStarTree
{
    RStarNode * rootNode_;
    RSReInsertHandler riHandler_;
}
```

Listing 4: Definition of an R*-tree class.

The R*-tree uses three C++ template parameters. `CT` is the datatype used for the values representing an MBR; intuitive types for `CT` are `float` and `double`. `D` represents the type of data payload, which in this thesis is an object identifier (`unsigned int`). `NC` is the branching factor of each node in the tree, that is, how many entries each node may contain at most (M).

The `RSReInsertHandler` helper class consists of a vector of Booleans corresponding to each depth level in the tree, and a stack of entries. The vector is used by the nodes to track whether a re-insertion or a split should occur when a node becomes overfull. Entries that need to be re-inserted are pushed on the stack and popped when re-inserted.

[4] R* does not modify these operations.

```
template<class CT, class D, int NC>
class RStarNode
{
    RStarTree<CT, D, NC>* tree_;
    const unsigned int level_;
    std::array<RStarEntry, NC> entries_;
    unsigned int numEntries_;
}
```

Listing 5: Definition of an R*-tree node class.

For a node to be able to push entries on to the re-insertion stack, it maintains a pointer to the main tree. In order to know which nodes are leaf nodes, a depth level value is needed. Since the array of entries is pre-allocated, an unsigned integer with the amount of actual entries is kept.

Each node contains a set of `RStarEntry` objects. These contain a pointer to another node (or `nullptr`), and the data payload of type `D`. Only leaf nodes contain data payload.

```
template<class CT, class D, int NC>
class RStarEntry
{
    RStarNode * node;
    D data;
    RSRect<CT> mbr;
}
```

Listing 6: Definition of an R*-tree entry class.

Traversal through the tree is recursive. Each node in the tree manipulates itself or its entries.

Insertion of a new rectangle starts with the `RStarTree` class invoking the function insert

```
RStarNode * insert(const D& data, const RSRect& mbr)
```

on the root node. It, in turn, uses the algorithm *chooseSubtree* [6] to calculate which of the child nodes the rectangle should be placed in. This is repeated until a leaf node is found. If the resulting node overflows (i.e., reaches M entries), 30% of the data entries in the node are placed in the re-insertion handler according to the *close re-insert* [6] and re-insertion for the leaf level is disabled.

The re-insertion handler reinserts each of its entries in the tree after the insertion algorithm has finished by calling `insert` as before. For the first node at a given level that becomes overfull[5], its entries are re-inserted and re-insertion for its level of the tree is disabled. If a node becomes overfull and re-insertion is disabled for its

[5]Re-insertion implies that a node at the leaf level has already become overfull, and therefore re-insertion at the leaf level is already disabled.

level of the tree, it is split using the split algorithm [6]. A pointer to the new node is returned to the previous call of `insert` which places it as an entry. In the case that the root node is split, a new root node is created, containing the previous root node and its sibling as entries.

Deletion uses a recursive strategy similar to `insert`. The `remove` function

```
remove_code remove(const D& data, const RSRect& mbr)
```

in the `RSNode` class is used. A recursive depth-first search for the correct entry is started at the root node. If an entry's MBR encloses the MBR of the object, `remove` is called on the contained node. In a leaf node, each entry is checked to see if it contains the data to be removed. The search stops when the correct object is found or if it is not found.

When removing an object in an R*-tree, there are three possible results a delete call to a node might return. In our implementation, these three cases are implemented using an enumeration type, called (`remove_code`), which contains a value for each of these cases:

- Not found: The object to remove was not found in this sub-tree.
- Condense: The object was removed in this sub-tree and the entry's MBR should be recalculated.
- Eliminate: The object was removed, resulting in the node becoming underfull. The node's remaining entries are placed in the re-insert handler and the node can be removed by its parent.

If a condense value is returned from a child node, the MBR of the child node's entry is recalculated. The current node then returns a condense value, propogating upwards. If an eliminate value is returned from a child node, the child node is removed. If the current node becomes underfull, remaining entries are placed in the re-insertion handler and an elimination value is returned to the parent, otherwise a condense value is returned. After a deletion, the re-insertion of entries is done as in the insertion procedure.

A window query recursively traverses down all intersecting entries, starting at the root node. For each entry, a containment test is first performed; if the query fully encloses the entry's MBR, all entries in its sub-tree can be added without any further checking. Otherwise, the algorithm checks whether the entry's MBR intersects with the query area or not. If it does, the algorithm traverses to the entry's child node. The function takes a pointer to a vector of results as an argument, and intersecting entries in the leaf nodes are appended to this vector.

Updating is done by first removing the object's old entry from the index and then inserting the new entry.

4.2.6.1 Bottom-up updates

For bottom-up updates to work, each `RSNode` is extended with a pointer to its parent. When the object has been found using the bottom-up index, the bounds of the node are checked to determine if the updated entry still fits within — if so, the entry is simply updated, leaving the node and parents untouched. If the updated entry does not fit, the original entry is removed from the bottom up using pointers to the parent nodes.

4.2.6.2 Update memo

For token-cleaning, each node one level above the leaf nodes receives their own token, which maintains an index value: the next entry in the node to clean. All tokens are stored in a vector. Each token consists of a pointer to its corresponding node and the index value. Every I updates, this vector is traversed; for each token, the index value is used to access and clean the corresponding leaf node. The index value is then incremented.

When cleaning the leaf nodes, it is possible for them to become underfull which necessitates removal from the parent node, which, in turn, may also become underfull; therefore, each non-leaf node, except the root node, must also maintain a pointer to their parent. The cleaning procedure can then recursively travel up towards the root node to either condense or further eliminate nodes.

Particular care needs to be taken to not add new tokens when cleaning up in a token, since appending tokens to the vector might invalidate the iterator used to traverse it. This is solved by temporarily storing the new tokens in a separate vector until the cleaning procedure has finished and then append the temporary vector to the token vector. If a token's node is removed when cleaning, it can be removed from the vector without invalidating the iterator.

To add *clean-upon-touch*, the `insert` function is modified so it also returns a Boolean value. When accessing the leaf node where the object is inserted, a cleaning procedure begins. The Boolean return value signals the parent node during the recursion upwards whether it should remove its child node or not.

4.3 Test environment

To evaluate the spatial indexes, a test environment is needed. This environment consists of several important parts: which physical machines are used, which data scenario to test, and how the benchmarks are executed.

The tests are run on the following hardware and software:

- Intel Core i7-8700 @ 3.2GHz.
- 64 GB DDR4 RAM in dual channel, 2666 MT/s.
- Windows 10, build 17763.
- Compiled using Visual C++ 2019 compiler with precise floating point and O2 optimization flags.

The performance of the chosen spatial indexes are affected by how the data is distributed; we have therefore specified a *scenario*: a collection of n object generator specifications $\{\mathcal{D}_0, ..., \mathcal{D}_{n-1}\}$, which decide how objects are generated. Each specification $\mathcal{D}$ consists of the following data:

- Weight specifying the percentage of all objects that will be sampled from $\mathcal{D}$.
- Two distributions for deciding objects' center positions in x and y axis. If these are uniform distributions, minimum and maximum values are specified; if they are normal distributions, means and variances are used instead.
- Two normal distributions for deciding objects' widths and heights in x and y axis.
- A fixed speed for each rectangle in the set.

The scenario consists of several of these distributions to simulate objects of varied sizes, with different velocities, and placement in the world. An example of it can be seen in Figure 4.2. Visible in the figure are several hotspots: many objects that lie close together. It also has several sparse uniform distributions with different velocities.

Since the scenario includes all kinds of different data, it represents, in some fashion, general usage of a spatial index. Different specific data distributions will affect the performances of the spatial indexes; it is therefore difficult to draw general conclusions of which generally performs best. This scenario is a good candidate, however, since it contains all kinds of data. Measurements on this scenario is therefore viewed as an appropriate basis to draw conclusions on overall performance.

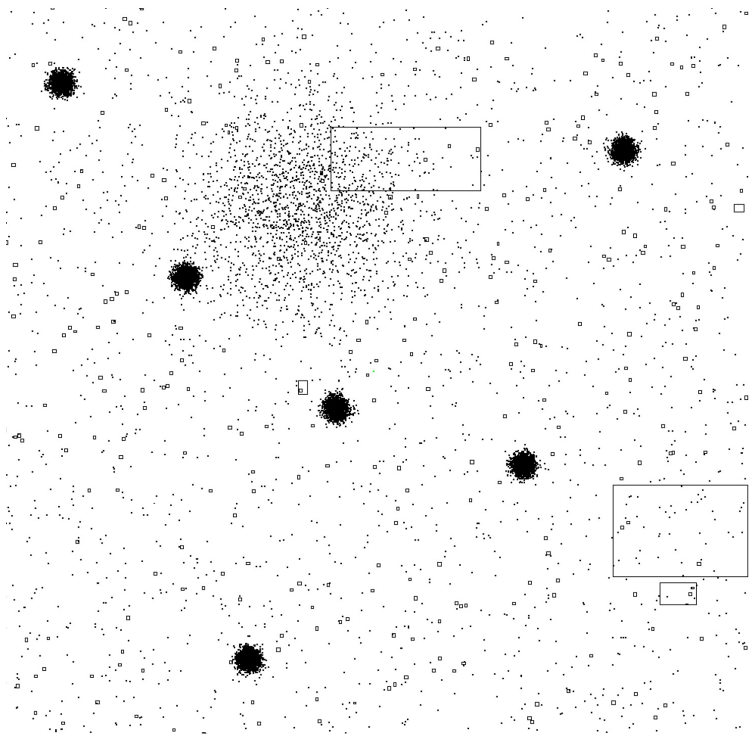


Figure 4.2: The chosen scenario with 15'000 objects. It is made up of densely packed normal distributions of smaller objects; a few larger objects; some middle-sized. Velocities of the objects vary.

We fix the world size to a square of size $10^9 \times 10^9$ for the specified scenario. If an object tries to move outside this world, it is bounced against the edge.

Query and update performances are measured using a benchmarking suite. When updating objects, a random object identifier is selected, and the corresponding object's location is updated by randomly generating a direction and calculating the new position based on the objects velocity — this results in something similar to Brownian motion. This update is then sent to the spatial index. We measure the time it takes for the index to perform this update.

For measuring queries, we generate a collection of query rectangles[6] which are uniformly distributed over the data. Query performance is measured as the time it takes for the index to return results from all queries.

4.4 Evaluation

The evaluation of the spatial indexes is performed in two steps:

1. Each spatial index is evaluated by itself in order to choose the best index-specific parameters values.
2. The spatial indexes, equipped with the chosen parameters, are compared to each other's across varying problem sizes.
3. Update variants are evaluated.

To choose the best index-specific parameters, query and update performances are measured.

For the specific case of selecting a clean interval for token-cleaning in update memo, three measurements are performed. The overall objective is to choose an interval which maximizes update performance while still retaining good update performance. First, for a collection of different intervals, a *garbage ratio* is measured, which is defined as the number of obsolete entries in a spatial index divided by the current, non-obsolete ones. This yields a value quantifying how much of the index is made up of obsolete entries. Then we measure how queries perform on static spatial indexes with different garbage ratios. Lastly, the intervals are used for update performance estimation, which we cross-reference with the query performance and garbage ratio results to choose a fitting cleaning interval.

[6]Visual inspection showed that 200 query rectangles resulted in good coverage of each scenario.

5 Results

In this chapter, we present and analyze the results of our evaluation process. The first step is to choose index-specific parameter values, which is done in Section 5.1. Update memo parameter selection and analysis is presented separately in Section 5.2. The next step is to compare the spatial indexes with the chosen parameter values. The spatial indexes' memory usage is also analyzed. This is done in Section 5.3.1.

5.1 Selecting index-dependent parameters

Each spatial index has unique parameters that need to be chosen in order to perform the final comparison evaluation:

- **Simple quadtree**: Bucket size.
- **Loose quadtree**: Bucket size, expansion factor.
- **Loose-linear quadtree**: Expansion factor.
- **R*-tree**: Maximum number of entries per node.

5.1.1 Quadtree variants

In this section we choose parameters for the simple, loose, and loose-linear quadtrees.

For evaluating the bucket size, the following values are used: 1, 16, 32, 64, 128, 256, 512, 1024. The expansion factor is evaluated in steps of 0.1: 0.1, 0.2, ... 1.0, with one exception: 0.999. This value performed best for Samet et al. [41], for the loose-linear quadtree. It is therefore included in the evaluation.

The B-tree, which is used by the loose-linear quadtree, has a node size of 512. While 256 is the default value in the library, 512 performed better in a quick evaluation.

First, we evaluate the loose quadtree with values for the parameters *expansion factor* and *bucket size* on the dataset described in Section 4.4. Figure 5.1 shows the query performance results and Figure 5.2 shows the update results. An expansion factor of zero results in poor query performance, while a bucket size of one results in both poor query and update performance. With a bucket size of one, only one object may be assigned to a leaf node before it must split; this means that the tree grows deeper than if a larger bucket size is used. A deeper tree means that both query and update algorithms must traverse more nodes, where many of the nodes contains either very few or no entries. When the expansion factor is zero, query performance is affected

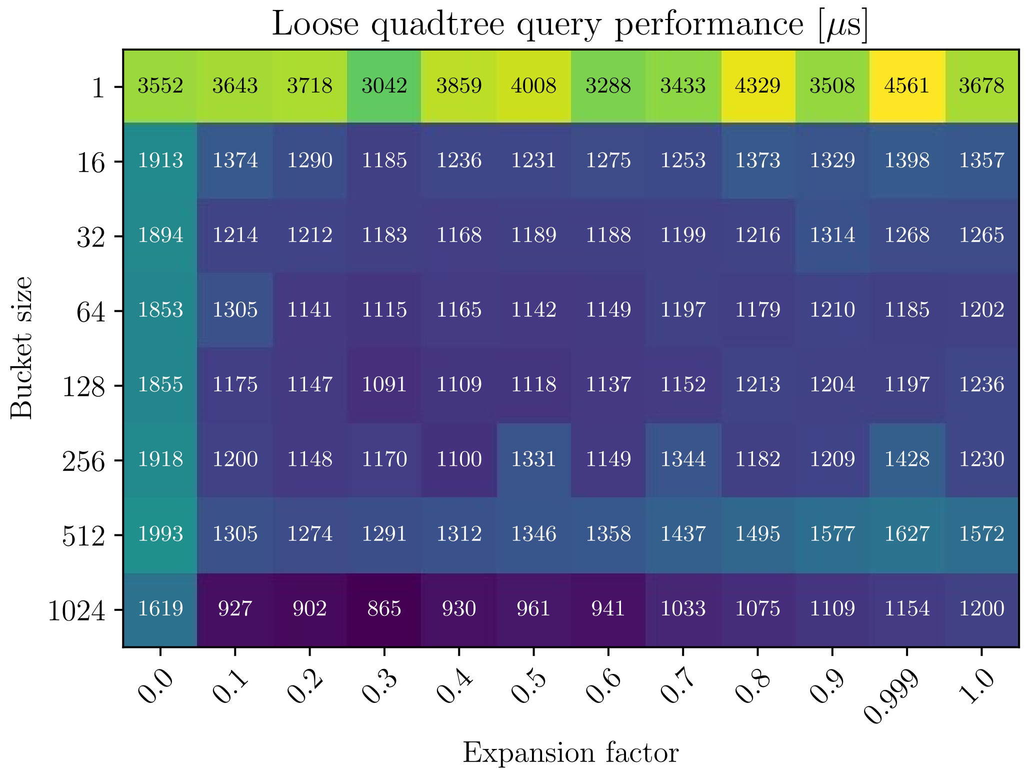
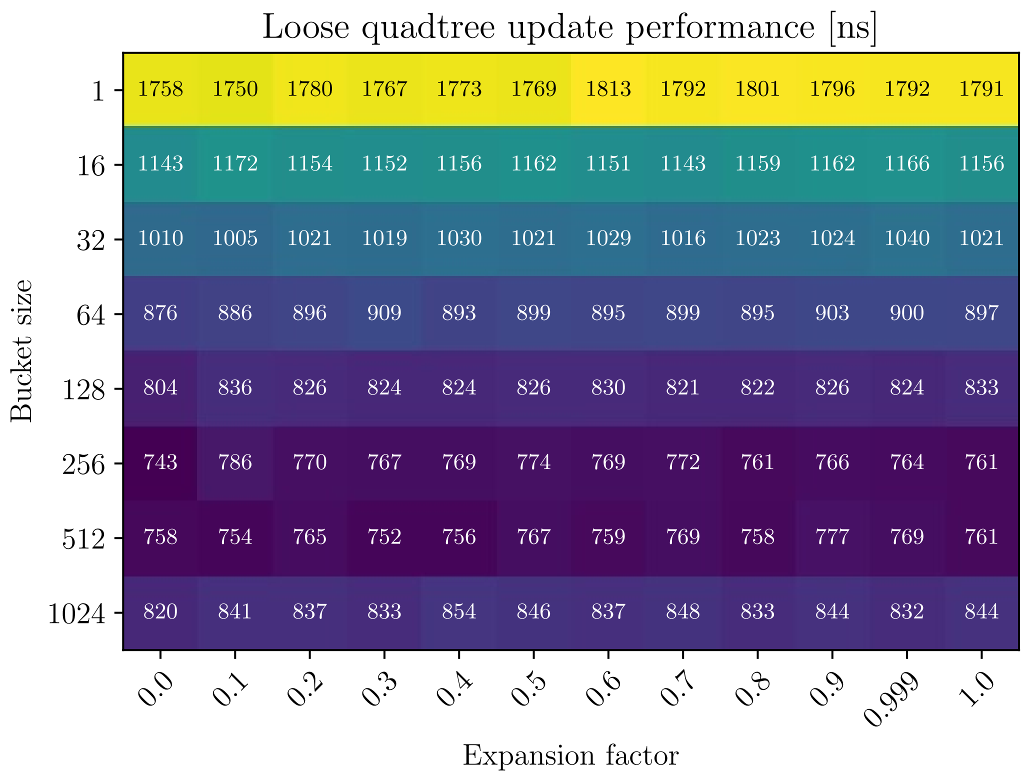


Figure 5.1: This graph shows the average query performance for the loose quadtree, given different bucket sizes and expansion factors. The evaluation was performed with 10 million objects.

because many entries crossing boundaries between nodes high up in the tree will be stored there. With only a small expansion factor, many of those objects will instead be stored deep in the tree, so the query algorithm does not need to test those.

Apart from a bucket size of one and an expansion factor of zero, the rest of the tested parameter values performs similarly. For the rest of the evaluations, we select an expansion factor of 0.4 and bucket size of 256 for the loose quadtree.

It is interesting to note that the parameters are important in different cases. On large objects, choosing an expansion factor has significant impact on final performance, while the bucket size does not matter very much. For smaller objects, the inverse is true. These results are in Figure A.1 and Figure A.2. When objects are very small, almost point-like, their center points tend to move outside the unexpanded area as they move. In this case the expansion factor does not matter, as objects must have their center points inside the unexpanded area, even if the expanded area encloses the full object. It is also likely that the depth of the tree is limited by the bucket size which results in the unexpanded area being very large compared to the objects. Updates will then be much more likely to still be contained by the unexpanded area,



Bucket size	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.999	1.0
1	1758	1750	1780	1767	1773	1769	1813	1792	1801	1796	1792	1791
16	1143	1172	1154	1152	1156	1162	1151	1143	1159	1162	1166	1156
32	1010	1005	1021	1019	1030	1021	1029	1016	1023	1024	1040	1021
64	876	886	896	909	893	899	895	899	895	903	900	897
128	804	836	826	824	824	826	830	821	822	826	824	833
256	743	786	770	767	769	774	769	772	761	766	764	761
512	758	754	765	752	756	767	759	769	758	777	769	761
1024	820	841	837	833	854	846	837	848	833	844	832	844

Figure 5.2: This graph shows the average update performance for the loose quadtree, given different bucket sizes and expansion factors. The evaluation was performed with 10 million objects.

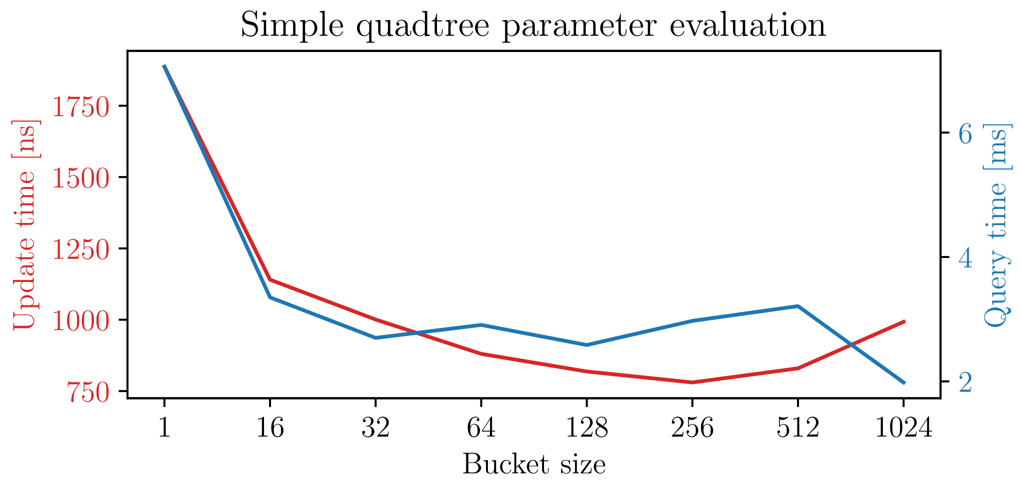


Figure 5.3: How the bucket size property affects update and query performance for the simple quadtree. The evaluation was performed with 10 million objects.

compared to moving into the unexpanded area. Larger objects, on the other hand, have a higher probability to be enclosed by a node's expanded area and still have their center points be inside the unexpanded area. In this case the expansion factor has a larger impact and the bucket size a smaller impact on performance.

The simple quadtree's performance for different bucket sizes is seen in Figure 5.3. For this variant, just like the loose quadtree, a bucket size of one degrades performance substantially. For querying, the best-found bucket size is 1024, while update performs best with 256. Given that the focus of this thesis is more on update performance, 256 is picked as parameter value for continued evaluation.

As seen in Figure 5.3, the update performance at first improves as the bucket size grows, until a certain threshold: 256, the performance then starts to degrade. There is a trade-off between the performance overhead of traversing through the tree and iterating through a node's vector of objects to find the correct one. As the bucket size becomes larger, searching through a node's vector of objects takes longer time, but fewer nodes need to be traversed as the tree's depth is limited.

Next, we choose an expansion factor for the loose-linear quadtree. See Figure 5.4 for the evaluation results. Selection is more difficult here than for the other quadtrees, because querying and updates yield very different results. Update performance generally improves when the expansion factor is larger; however, query performance becomes worse and worse. Update result only varies in a few hundred nanoseconds, but queries perform five times worse with 0.9 in expansion factor, compared to 0.1. This means that larger consideration must be put into query performance effects when choosing an expansion factor, compared to other spatial indexes. From Figure 5.4 we deduce two values, where the subsequent value substantially worsens query performance: 0.3 and 0.7. Both are good picks, but 0.7 yields better update performance, so it is selected as the expansion factor for the rest of the evaluation.

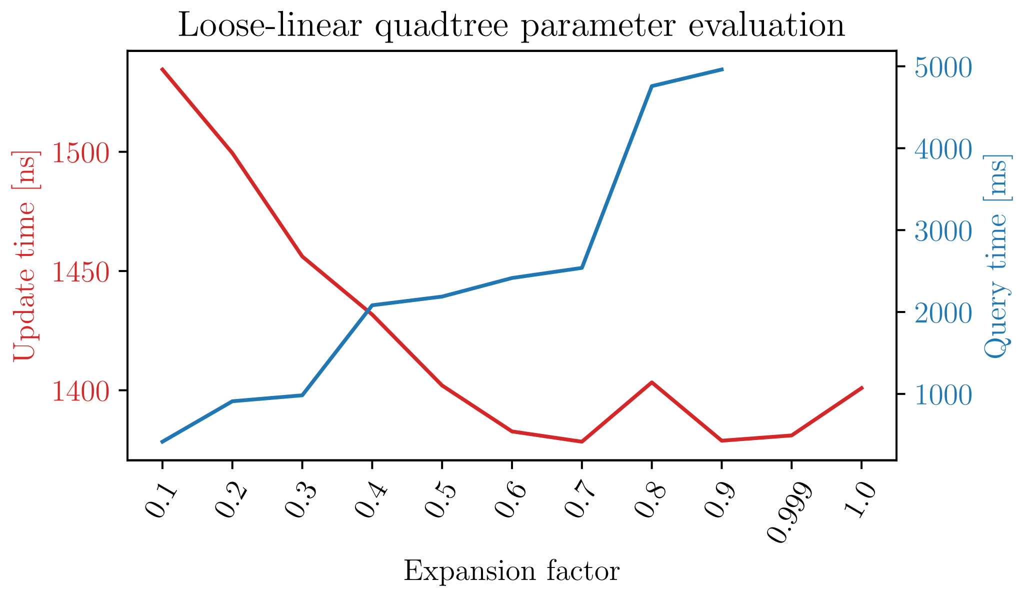


Figure 5.4: A depiction of the loose-linear quadtree's performance with different expansion factors. The evaluation was performed with 10 million objects.

5.1.2 Query algorithm for loose-linear quadtree

Several different algorithms (see Section 4.2.5) have been implemented for the loose-linear quadtree; these are evaluated in this section. Figure 5.5 shows the results.

The algorithm with worst performance is WindowQueryPaper, the algorithm proposed by Aboulnaga and Aref. An improved algorithm, WindowQueryRoot, which starts the search from the root node, has two orders of magnitude, or 97.8%, better performance than WindowQueryPaper and performs the best. This result is expected, with the limitations discussed in Section 4.2.5 in mind. The authors that introduced the algorithm designed it for accessing data in databases; the limiting factor then is the number of I/O operations required for the operation. Under such a scenario the algorithm might perform better, but for main memory usage it is useless. Even WindowQueryNaive, which simply iterates through all key-value pairs in the B-tree, performs better for all problem sizes measured in Figure 5.5.

The performance of WindowQueryNaive degrades linearly as more objects are stored. This makes sense since, as already mentioned, the algorithm iterates through all values.

Both WindowQueryRoot and WindowQueryOffset show similar performance in this evaluation. In WindowQueryOffset, the search space is partitioned into nodes of the same size as the largest current node in the index[1], and then executes a recursive query for each. When the largest node in the tree is small compared to the search area, the partitioning will result in many starting nodes.

[1]The loose-linear quadtree does not contain any empty nodes.

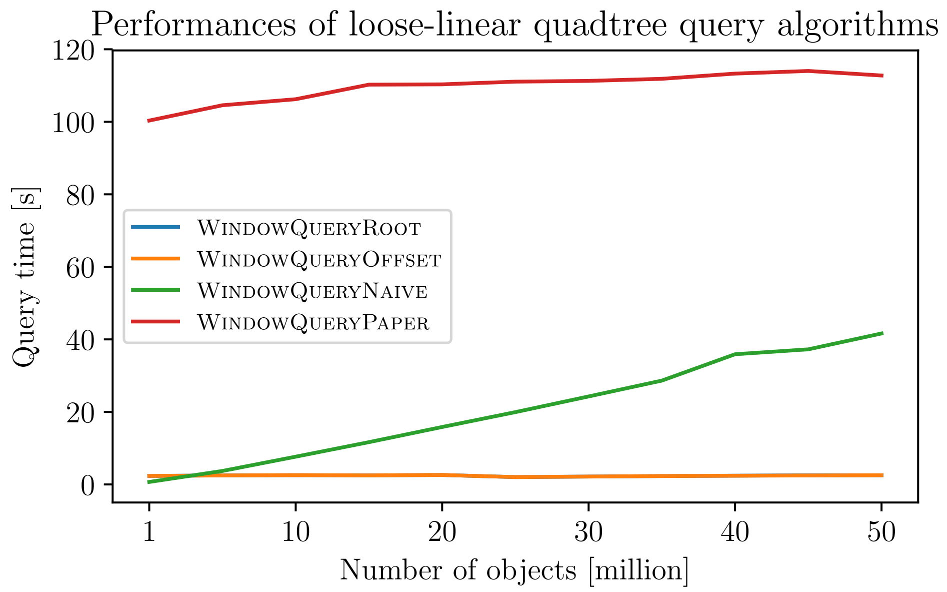


Figure 5.5: Performance comparison of different loose-linear quadtree query algorithms plotted for varying problem sizes. The algorithms WindowQueryRoot and WindowQueryOffset has similar performance so the line of WindowQueryRoot is barely visible.

Since there exist objects in our scenario that are large, the starting areas defined by WindowQueryOffset will also be large, and therefore there will be fewer starting nodes. Therefore, the algorithm will start at nodes close to the root, which WindowQueryRoot does. This explains why their performances are so similar. WindowQueryRoot, however, performs better when there only are small objects in the tree, since WindowQueryOffset will start recursion at many smaller nodes. The first is therefore a better choice when the data size distribution cannot be assumed. It will be used for the remainder of the evaluation.

5.1.3 R* variants

The R*-tree has three parameters: minimum node-count factor, re-insertion percentage, and maximum number of entries per node. Two of these, re-insertion percentage and minimum node count factor, are set to values recommended by Beckmann et al., 30% and 0.4 respectively [6]. In this section, an appropriate value for the maximum number of entries is chosen.

As seen in Figure 5.6, increasing the maximum number of entries per node does not affect update performance significantly, except for values 10 and below. A low maximum number of entries will result in nodes underflowing or overflowing very easily, which results in very expensive operations of re-inserting many entries to maintain the tree.

The query performance, on the other hand, slightly degrades in performance as the number of entries increase. These results are quite volatile, however. More entries per node mean more intersection tests at each node; on the other hand, fewer nodes need

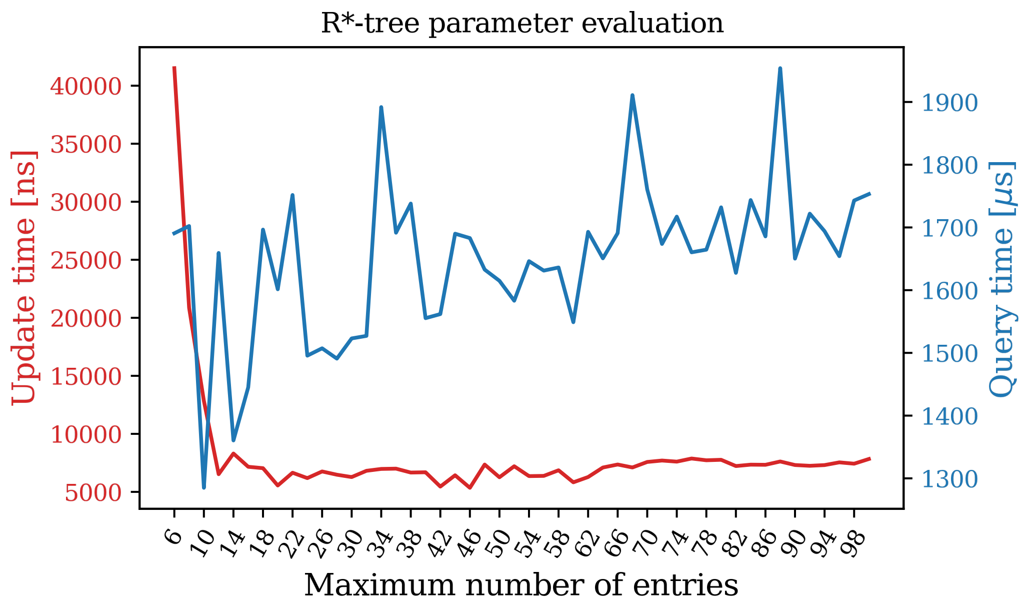


Figure 5.6: A depiction of update and query performance for the R*-tree with different maximum number of entries per node. The evaluation was performed with 10 million objects.

to be traversed, but it seems that this overhead is negligible compared to intersection testing entries.

A reason for the volatility is that slight changes in the maximum number of entries per node could lead to widely different R*-trees. Nodes throughout the tree will split at different times, and this may result in query areas traversing either more or fewer subtrees which affects performance.

With both update and query performance in mind, a maximum amount of 42 entries is chosen for the rest of the evaluation. It is one of the best performing parameter values for updates, while query performance is still good.

5.2 Update memo parameters

Update memo has been implemented for the following spatial indexes: loose quadtree, loose-linear quadtree, and R*-tree. We optimize the *cleaning interval* parameter for the variants of update memo that use it. This is performed by cross-referencing the impact that garbage ratio has on query performance (Section 5.2.1), the relation between cleaning interval and garbage ratio (Section 5.2.2), and finally the impact that cleaning interval has on update performance (Section 5.2.3). It is necessary to choose a good cleaning interval for both query and update performance. Another choice that must be made is which update-memo technique to use for the loose quadtree: token-cleaning with clean-chunk or just token-cleaning (Section 5.2.4).

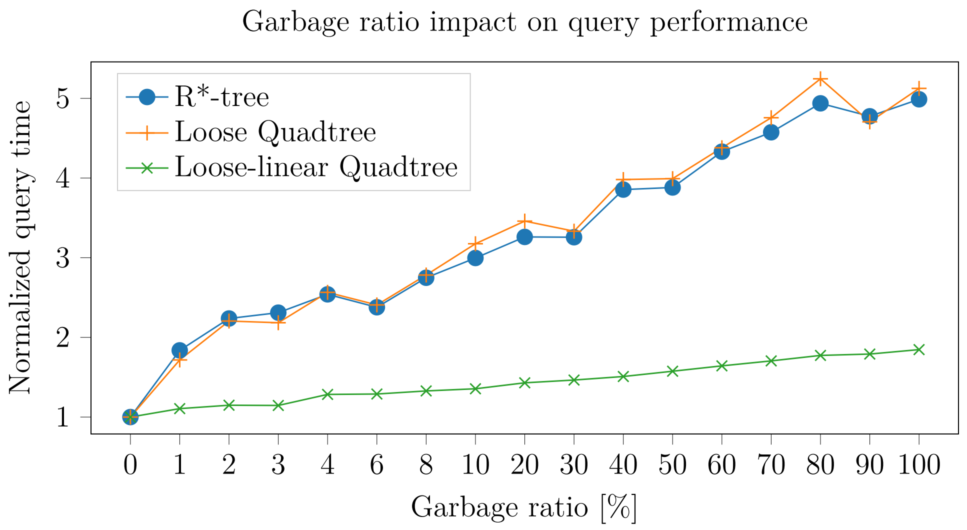


Figure 5.7: How different garbage ratios affects query performance. These benchmarks were run with 10 million objects. Normalized query performance, with no garbage as the baseline compared to the garbage ratio of three spatial indexes. Lower value is better.

5.2.1 Garbage ratio

The garbage ratio is defined as the number of obsolete objects in the spatial index divided by the current objects. It is a measure of how many of the objects stored in the index are obsolete.

To measure query performance when using *update memo* techniques, we fix the garbage ratio. This shows how the number of obsolete objects affects query performance. To fix a certain garbage ratio, we insert objects into each spatial index, disable all cleaning and perform a set number of updates. Since cleaning is disabled, the number of updates is equal to the number of obsolete entries.

The results for different garbage ratios are seen in Figure 5.7. The huge performance impact when increasing garbage ratio from 0% to just 1% can be explained by the secondary update-memo index. At 0%, there are no stale entries in the index, and therefore the secondary index will be empty. Our assumption is that the implementation of `std::unordered_map` does not compute any hash when the map is empty, which results in very little performance impact. The minor increase in performance at 6% is probably due to the mapping structure rehashing for the increase in entries, thus decreasing average bucket sizes.

The loose-linear quadtree is not nearly as affected, compared to the other indexes, because it is much slower overall (see Figure A.4), and the performance impact of lookups to the secondary update-memo index is therefore much smaller.

Given the results, it seems reasonable to aim for a garbage ratio between 0–20%. Of course, this depends on how update performance is affected when the cleaning interval is set to achieve that garbage ratio.

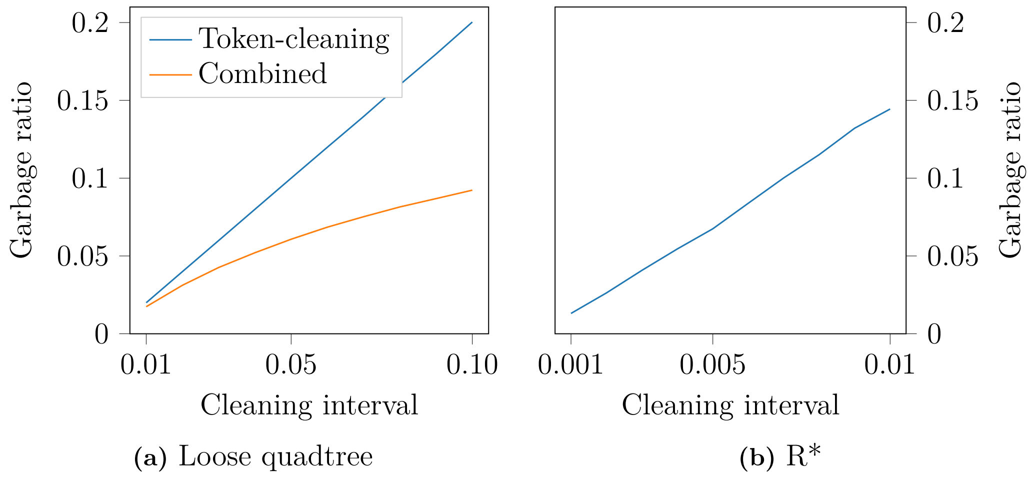


Figure 5.8: The effect of different cleaning intervals on the garbage ratio for the loose quadtree and R*-tree. In the loose quadtree evaluation, "Combined" refers to the case when we use token-cleaning and clean-chunk together. Clean-upon-touch is disabled.

5.2.2 Cleaning interval garbage ratio

The next step is to see how the different cleaning intervals affect the garbage ratio. We define the cleaning interval as number of updates divided by the total number of objects in the index. For example, a cleaning interval of 0.5 means that a cleaning operation is performed when the number of updates equals a multiple of half the number of objects in the index. Results of the evaluation are seen in Figure 5.8. Note that due to differences in implementation, to achieve similar garbage ratios, the cleaning interval value is an order of magnitude lower for the R*-tree than the loose quadtree.

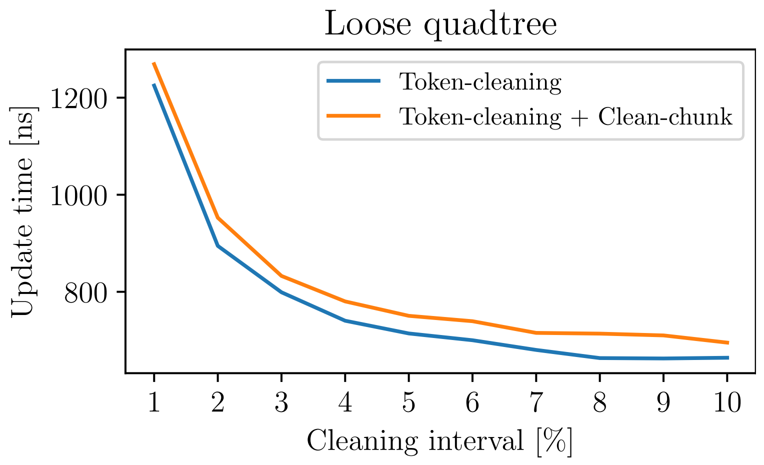
Clean-chunk combined with token cleaning keeps the garbage ratio much lower than when token-cleaning is used by itself, since clean-chunk will clean extra nodes with no regard to which cleaning interval is used. When only the cleaning interval is used, for loose quadtree and R* respectively, there is a clear linear relationship between the cleaning interval and garbage ratio — when the tree is cleaned less frequently, there exists more obsolete objects.

For all cleaning intervals evaluated in Figure 5.8 with clean-upon-touch enabled, the garbage ratio is 0.13% for the loose quadtree and 0.13 - 0.39% for the R*-tree. Choosing a cleaning interval for this case is therefore not necessary.

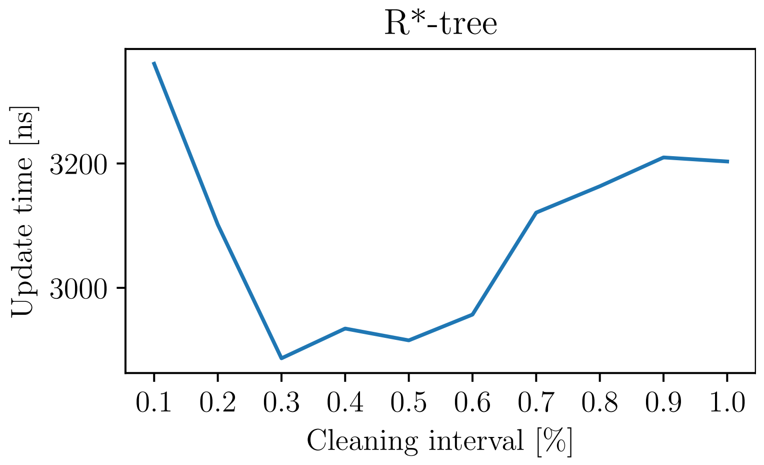
5.2.3 Cleaning interval update performance

To measure update performance, we ran several benchmarks for cleaning intervals that kept the garbage ratio contained somewhere below 20%. The results are seen in Figure 5.9.

In Figure 5.9a, token-cleaning by itself and token-cleaning + clean-chunk are measured



(a) Update performance for two cleaning methods for the loose quadtree.



(b) Update performance for the R*-tree.

Figure 5.9: Different cleaning intervals affect update performances for the loose quadtree and R*-tree. Since there are fewer tokens in the R*-tree, it must be cleaned more frequently to be on par with the loose quadtree. Therefore, lower cleaning intervals are evaluated for the R*-tree.

for update performance. Both methods perform roughly the same; but just using token-cleaning is slightly better.

For the R*-tree, the best interval is 0.003. Lower intervals clean the tree too often, severely degrading performance. Cleaning less frequently also negatively affects update performance. One reason is that a large amount of obsolete entries means nodes are more likely to become full and split during updating. Another reason is that when cleaning finally is performed, many nodes will become underfull instead due to containing to many obsolete entries which will result in many re-insertions.

5.2.4 Selecting cleaning intervals

For the loose quadtree, the combination of token-cleaning and clean-chunk has the benefit of a much lower garbage ratio, while still yielding good update performance. Since performance mostly stops improving with a cleaning interval over 0.05, which correlates to a garbage ratio of about 6%, the combination of both methods, with a cleaning interval of 0.05, is chosen for the rest of the evaluation.

The R*-tree had best update performance with a cleaning interval of 0.003. From Figure 5.8, we can deduce that this leads to a garbage ratio of 5%. Figure 5.7 shows that this leads to a limited degradation in query performance, and therefore seems like a good pick for cleaning interval.

5.3 Index comparison

In this section, each spatial index is equipped with the parameter values chosen in the previous section, for an index to index comparison.

5.3.1 Memory requirement

This section analyzes how much memory each spatial index requires, for different numbers of objects. These are measured on a 64-bit machine[2].

Figure 5.10 shows how much memory each selected spatial index uses, given a fixed number of objects to store. The R*-tree has the most memory overhead, this is because each node allocates memory for the maximum amount of entries, even if they on average contain less.

The loose-linear quadtree has similar memory usage as the list; both are worse than the simple and loose quadtrees. One might think that the list would have the lowest memory overhead, but this implementation uses a secondary map index to directly access objects that are to be updated, which essentially doubles memory usage.

In Figure 5.11, the additional memory performance of the different update techniques is shown.

Bottom-up updating adds a sizeable increase for all indexes, since a secondary index containing references to all objects must be stored. Note the lower increase in memory

[2]32-bit versions will show different memory usage due to pointer size, but still show same trend and relative performance for the data structures.

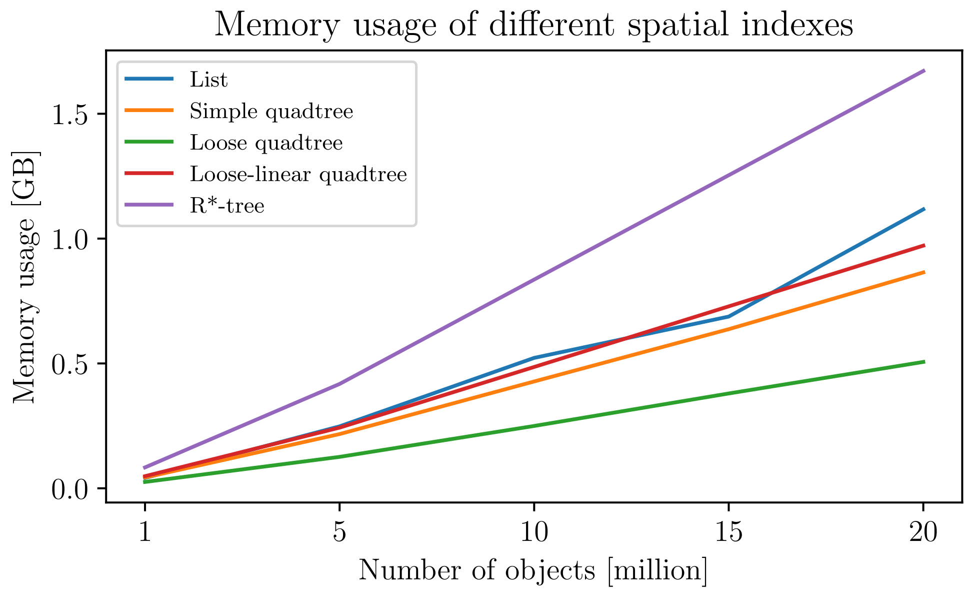


Figure 5.10: This graph shows the memory usage of the chosen spatial indexes for different amounts of stored objects. Bottom-up updating or update memo is not used.

usage for the loose quadtree when adding bottom up, and the higher amount for the loose-linear, compared to R*-tree. The data structure `std::unordered_map` keeps *key-value pairs* as the internal data storage. In R*-tree, while the object identifier is still four bytes, the pointer to the corresponding node is eight bytes. Due to padding of the resulting pair, a combined 16 bytes must be allocated, instead of what might be the expected 12 bytes. For the loose quadtree, the object identifier is four bytes as well as the chunk identifier, which results in a combined eight bytes — half the amount of the implementation in R*. In the case of the Loose-linear bottom-up, the Morton encoding is used as the value, which itself takes 16 bytes due to padding, the resulting pair is then 24 bytes in size.

Update memo adds a different amount of memory usage for each index, due to the differences in implementation. In general, this is lower than the additional memory that bottom-up allocates.

5.3.2 Effect of quadtree implementation

Even though the simple quadtree and loose quadtree with expansion factor zero function the same in their update and query algorithms, their query performances differ due to the different node management. Recall that the simple quadtree allocates each node by itself and keeps pointers between nodes to indicate parent-child relationships, and that the loose quadtree uses the *chunk* representation described in Section 4.2.3.

The query performance gap between the two variants is seen in Figure 5.12. Also included is the loose quadtree with the expansion factor that was selected in Section 5.1.1. As seen in the figure, the node management is responsible for approximately

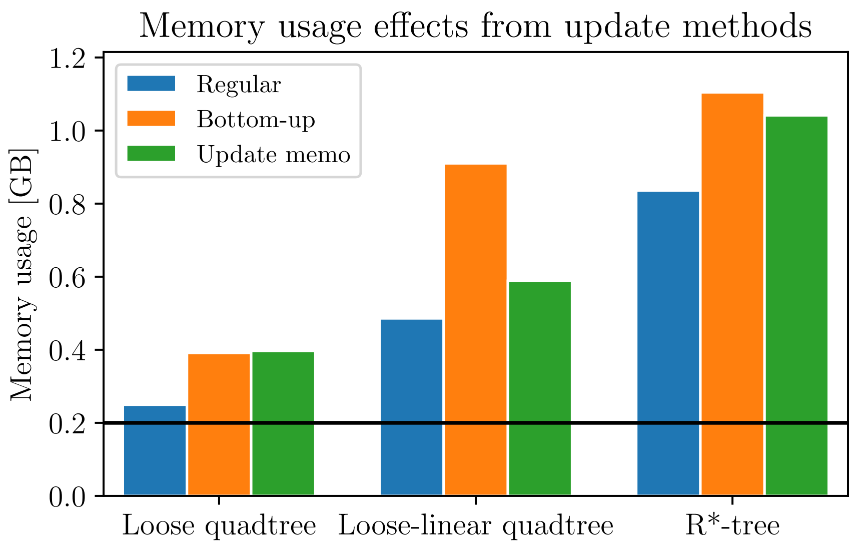


Figure 5.11: The additional memory overhead of different update techniques. Loose quadtree uses Token cleaning and clean chunk for Update memo (see Section 4.2.1). 10 million objects. The black line indicates the memory usage of the stored objects, i.e., the minimum amount of memory that needs to be allocated.

half the performance improvement; the other half is due the expanded node areas.

Figure A.3 displays the update effects of the same variants that are evaluated in Figure 5.12. It shows that update performance effects are negligible in this case, probably because fewer nodes must be traversed compared to the query algorithm.

5.3.3 Queries

The results from evaluating the query-performance for the chosen spatial indexes are seen in Figure 5.13. They are measured with respect to how many objects are indexed on the interval 2–100 million. It should be noted that the query rectangles have the same size, and so the amount of entries returned from each query has a linear relationship with the amount of objects.

As can be seen, the loose quadtree widely outperforms the others. In the case when 100 million objects are indexed, it performs 69% better than the simple quadtree, 81% better than the R*-tree, and 85% better than the list implementation. The simple quadtree, in turn, outperforms the R*-tree by 28%.

The results for the loose-linear quadtree are not included in Figure 5.13 since they are much worse than the others. A version of the figure with it included is shown in Figure A.4. For 50 million objects, the loose quadtree performs 99.9% better than the loose-linear quadtree. This result is based on the improved algorithm developed in this thesis; the original algorithm proposed by Aboulnaga et al. [1] would do even worse. This large gap is due to the problem with the query algorithm discussed in Section 4.2.5. It is also the case that objects generally are placed deeper

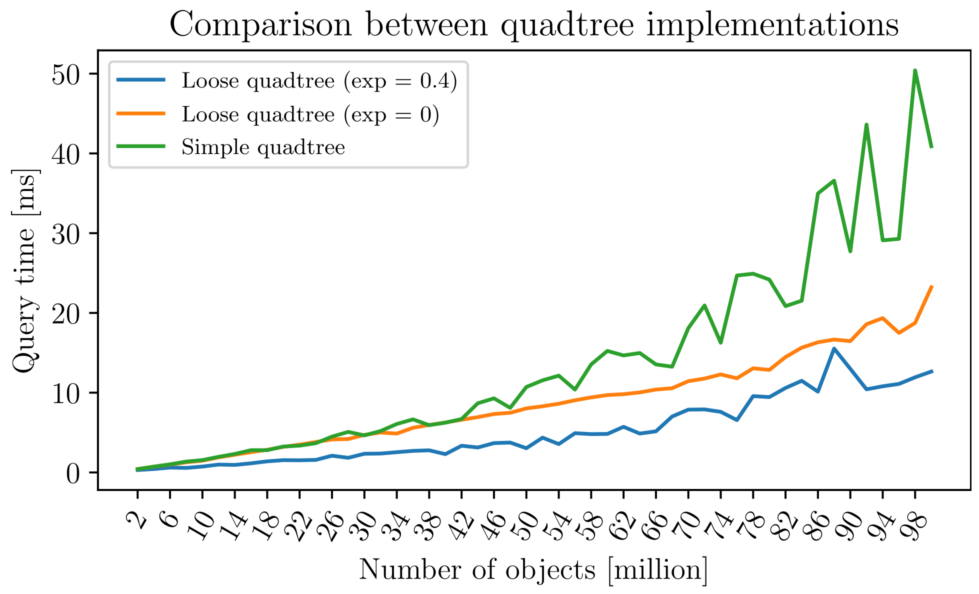


Figure 5.12: This graph shows the query performance effect of node management in the quadtree implementations, and the effect that looseness has on the quadtree's performance.

in the loose-linear quadtree than in the loose quadtree. The number of paths that must be explored by the query algorithm increase substantially at each level, so the loose-linear quadtree's algorithm will traverse many more nodes than the loose quadtree's.

We believe there are several reasons why the loose quadtree outperforms its peers. Two effects of looseness described in Section 3.2.3: that objects can be stored deeper in the tree, and that node overlap makes it necessary to traverse more paths, gives a hint.

Objects stored deeper in the tree means that query performance will improve, because less intersections tests need to be performed with objects outside the query area. In the simple quadtree, objects are stored closer to the root, and will therefore be tested by queries, even though they are not close to the query area.

The negative effect from node overlap is also probably not that significant, as the bucket size limits the depth of the tree. In short, the positive factor still affects to some degree, for larger objects, and the negative factor is minimized since the bucket size limits tree depth.

Another reason why the loose quadtree performs better could be cache efficiency. The simple quadtree does not consider cache performance at all.

We expected the R*-tree to perform well in the query tests, as it is designed to optimize querying. However, the R*-tree was originally designed to minimize I/O operations to secondary memory, which is not applicable here. The reason behind lack in performance for these datasets could be that overlaps between objects are permitted, and happen often, huge rectangles can cover several thousand smaller

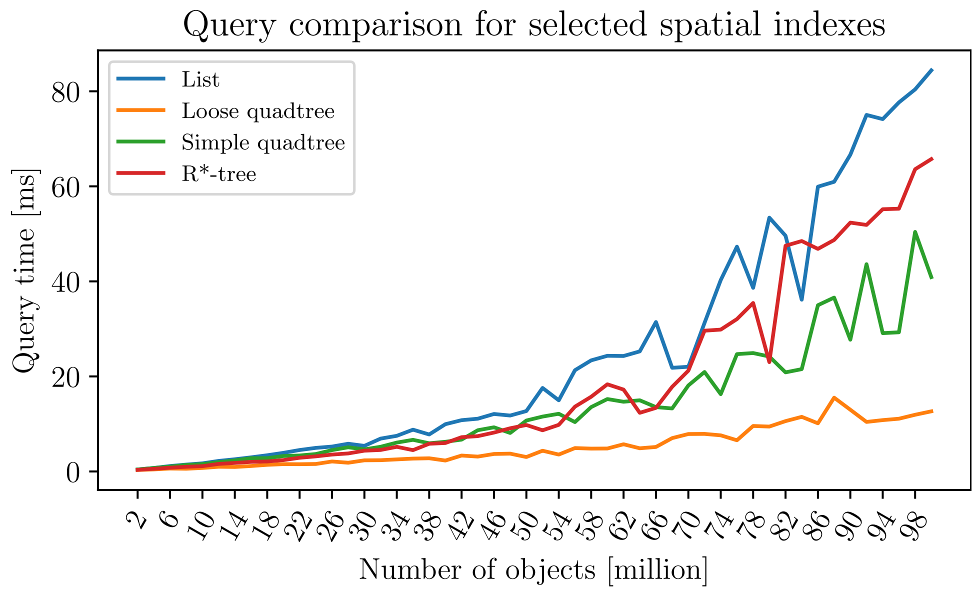


Figure 5.13: Query results for the selected spatial indexes with different number of objects indexed.

ones. It might also be that the maximum amount of entries should be lower or higher for different number of objects.

5.3.4 Updates

This section is divided into three parts. First, we evaluate how bottom-up updating affects the spatial indexes. Secondly, update-memo techniques are evaluated. Finally, a comparison between the chosen spatial indexes without these augmentations is performed.

5.3.4.1 Bottom-up updating

The effect of bottom-up updating is seen in Figure 5.14. For both the loose quadtree and loose-linear quadtree, the technique has almost no effect on update performance. The former has a limited depth because of the bucket size, so relatively few nodes need to be traversed. The latter already has a quick update algorithm which requires no tree traversal.

However, bottom-up updating makes a difference for the R*-tree. One reason for this is probably that all objects reside in the leaf nodes, so the number of nodes that have to be traversed is always the height of the tree when accessing the object to be moved; the loose quadtree, on the other hand, only has to traverse to leaf nodes in the worst case. Another reason is when there are several nodes overlapping the location of the old entry, the removal algorithm will need to search through all of these to find the entry.

These results mean that bottom-up updating is not worthwhile for the loose and loose-linear quadtrees, especially when considering the extra memory overhead. For

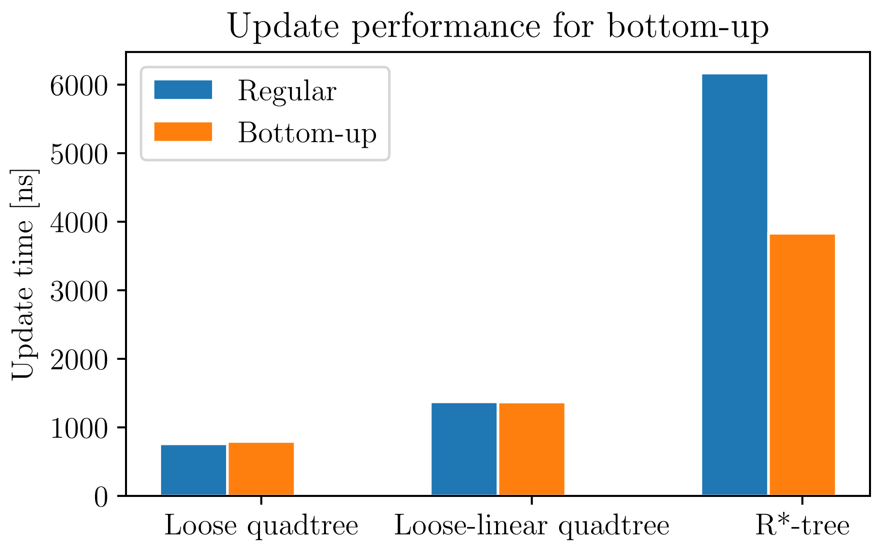


Figure 5.14: A graph showing how bottom-up updating affects the update performance. The evaluation was performed with 10 million objects.

the R*-tree, the technique can be worthwhile, if memory usage is of lesser concern. It is also a viable alternative to update memo, which will be discussed shortly.

5.3.4.2 Update memo

In Figure 5.15, the effects of update memo are shown. For both the loose quadtree and the R*-tree with bottom-up enabled, update memo performs worse; the overhead is too large for update memo to be a viable alternative. For the regular R*-tree, however, update-memo performs better, but not as well as just bottom-up without update-memo. This means that update-memo does not triumph for any spatial index in this evaluation.

The figure also shows the update performance with clean-upon-touch enabled. It performs worse with all three variants, probably because at least one node is cleaned every update operation, and all the entries in the node needs to be checked against the update-memo index. This adds a constant overhead to update performance. Clean-upon-touch manages to keep the spatial indexes very clean, however (see Section 5.2.2). The technique could therefore be feasible if query performance is more important than update performance, but then one might as well not use update memo at all since it was meant to improve update performance.

5.3.4.3 General update performance

The general update performances of the chosen spatial indexes are shown in Figure 5.16. Of all evaluated structures, the R*-tree has the worst performance, which is to be expected. The reasons for this are that updating is an expensive operation for these kinds of data structures: MBRs of entries must be re-calculated from the leaf

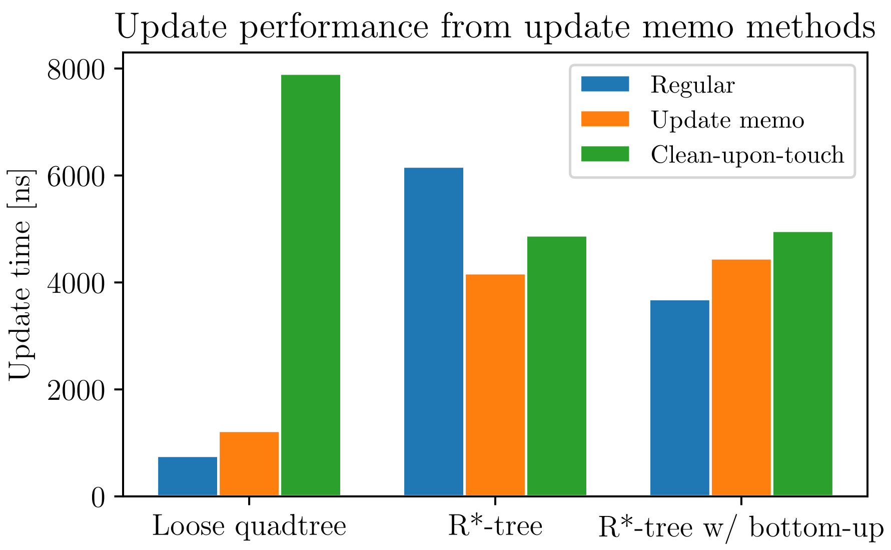


Figure 5.15: This graph shows how update memo-techniques affect update performances for the loose quadtree and R*-tree. “Regular” has no update-memo augmentation. “Update memo” uses the best update-memo technique without clean-upon-touch: token-cleaning + clean-chunk for the loose quadtree, and token-cleaning for the R*-tree. Both use the optimal parameter values derived in Section 5.2. Lastly, “Clean-upon-touch” are the same variants as “Update memo”, but with clean-upon-touch enabled. The evaluation was performed with 10 million objects.

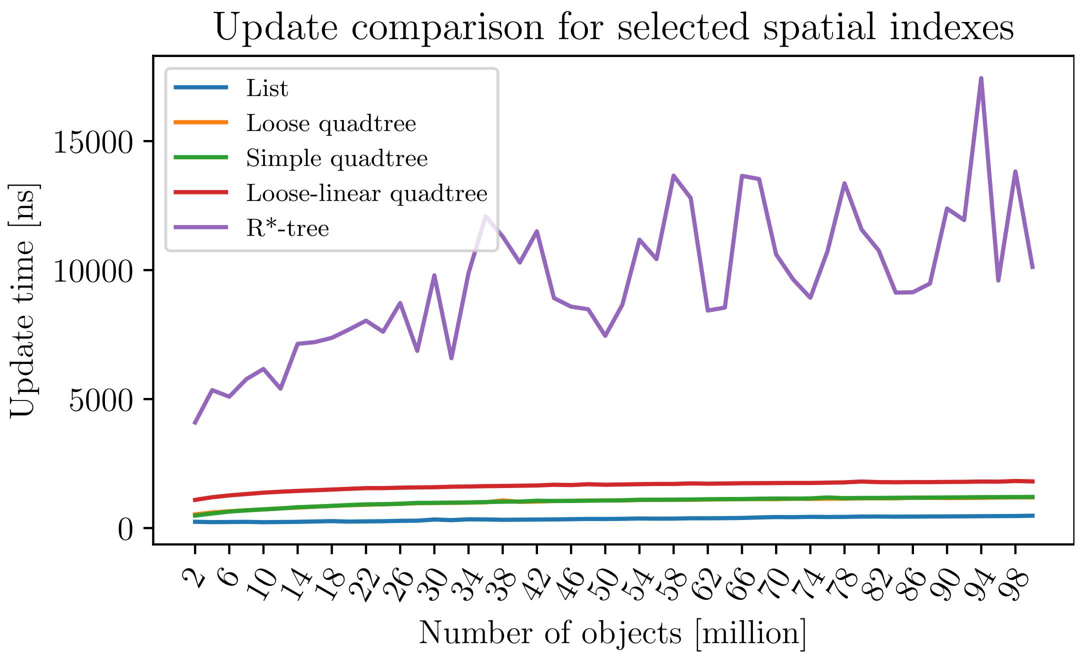


Figure 5.16: Update performance comparison. The loose quadtree lies directly behind the simple quadtree and can therefore not be seen. All approaches except R* scales well with an increase in the amount of objects.

node that was modified and upwards. Splits and re-insertion when nodes become overfull or underfull are also expensive.

The rest of the spatial indexes perform much better, with loose-linear being slower than the loose and simple quadtrees, which in turn are slower than the list. This is expected as the list directly access the objects that will be updated.

One might think that the update algorithm of the loose-linear quadtree would perform better than the loose quadtree's, but it does not. An explanation for this could be because of the effect that the bucket size has. The loose and simple quadtrees' heights are limited by the bucket size, which limits the number of nodes that have to be traversed to find the object's old entry. The overhead of calculating which node an object resides in — which the loose-linear quadtree's update algorithm does — is likely higher than traversing a few nodes.

6 Conclusion

6.1 Discussion

The aim of this thesis was to compare some spatial indexes with each other under a specified scenario, and with certain update techniques. For query and update performances the loose quadtree clearly outperforms the other indexes, as can be seen in Figures 5.13 and 5.16.

For each of the indexes there were several parameters that affected performance. When choosing each, the other parameters in some cases had to remain static, and therefore be assumed. Update memo, for example, was tested using parameters chosen for the traditional update approach. Testing each combination of parameters would not be feasible.

Results shown depends on the data used in this thesis, other scenarios may yield other results. Since performance is so data dependent, indexes must be compared for their intended use case. We chose what we believe is an interesting scenario, described in Section 4.3. It should give a good indication as to what the general performance of each index is. We believe that most of the common pitfalls for spatial index approaches can be found in this scenario, thus high performance is only achieved in it when the index performs well in most situations.

The bottom-up update method has minor effects for all indexes, except the R*-tree. Using bottom-up also leads to significant memory overhead since for each object there is an entry in the bottom-up index. However, the technique is easy to implement. With minor benefits and large memory usage, the bottom-up method seems unnecessary in general.

Update memo improved update performance for the R*-tree but worsened for all quadtree variants. The biggest shortcoming, however, was query performance, which degraded significantly. It was also complicated to implement and added several more parameters and choices to account for.

Update memo makes more sense when the index is stored on secondary memory, where I/O performance is the biggest bottleneck. Designed for R-trees, both papers that developed the method evaluated the number of I/O accesses [45, 49]. Postponing deletion to be performed in batches reduces I/O accesses for a node, as it is accessed once and all obsolete objects are removed, compared to accessing it every time an object should be deleted. Clean-upon-touch also makes sense from this perspective: when a node has already been accessed, cleaning it of old entries is cheap. On main-memory systems, however, where the latency for fetching data is lower, the extra operations and accesses to the secondary index reduces performance.

6.2 Conclusion

Much of previous research has focused on data that are either static geometry or moving-point data. The purpose of this thesis was to examine the field of spatial indexing for moving geometry.

Several spatial indexes were chosen and evaluated with respect to memory usage, query and update performance. These indexes are: list, simple quadtree, loose quadtree, loose-linear quadtree, and R*-tree. We also implemented two update techniques, bottom-up and update memo for applicable indexes. All implementations were made in C++ and a scenario consisting of several data distributions was specified and used for evaluation.

The results show that the loose quadtree outperforms all other indexes in both query and update tests. We find that update memo and bottom-up works well for R*-tree but perform either poorer or makes no difference for the other indexes.

6.2.1 Future Work

For the loose quadtree there are many aspects that can be examined in future works. Optimizing the merging of quadtree nodes, which was not a focus of this thesis, is one aspect that could improve performance further. There is a performance gap between the loose-linear quadtree and the loose quadtree. Updates were not as good as we hoped, and query performance was abysmal for the loose-linear version. Further research on improving the algorithms used in the loose-linear quadtree is warranted. It is also the case that update memo has not been used for linear quadtrees previously, so there are no update memo methods adapted for it.

When performing background research, many methods were deemed interesting but incomplete for our purpose. Some of these focus on query performance but lack any discussion about update methods and performance. These methods could be interesting to investigate further, and there is potential to produce original ideas to solve the problems.

Some encountered structures are aimed at indexing with the aid of a GPU. While this thesis focus is on CPU implementations, a comparison with GPU alternatives could be interesting, and to our knowledge no such survey exists at the time of writing.

One of few grid approaches that handles non-point spatial data by default is BLOCK [34], where a hierarchical grid is maintained, and queries are split into parts that query the different levels. While query performance was good, there was no discussion about how updating the structure would work. If one could introduce an update algorithm that does not compromise query results significantly, the resulting spatial index could become an interesting alternative to R-trees [17] and the loose quadtree

[47]. Extending other newer grid methods like PGrid [50] and QGrid [9] to support non-point data could also yield interesting results.

Hema and Easwarakumar [18] proposes a unique approach, where a rectangle is divided into line segments in two dimensions which are then kept in two segment trees, one for each dimension. The segment tree is called a BITS-tree [12]. The papers, however, lack any discussion about update methods. No performance analysis at all is provided, meaning that the impact of this approach is unknown. A rigorous analysis of the BITS-tree [18], evaluation of query performance, and proposing new update techniques could make an entire thesis by itself.

Bibliography

[1] Ashraf Aboulnaga and Walid G Aref. Window query processing in linear quadtrees. *Distributed and Parallel Databases*, 10(2):111–126, 2001.

[2] Afsin Akdogan. Partitioning, indexing and querying spatial data on cloud, 2016.

[3] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-time rendering*. AK Peters/CRC Press, 2018.

[4] Walid G Aref and Hanan Samet. Efficient window block retrieval in quadtree-based spatial databases. *GeoInformatica*, 1(1):59–91, 1997.

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

[6] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, volume 19, pages 322–331. Acm, 1990.

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

[8] Laurynas Biveinis, Simonas Šaltenis, and Christian S. Jensen. Main-memory operation buffering for efficient r-tree update. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 591–602. VLDB Endowment, 2007.

[9] Kun-Lun Chen, Yan-Ru Liu, and Qing-Xu Deng. Indexing Moving Objects Using Query Oriented Parallel Grids. In *Proceedings of the 2nd International Conference on Computer Science and Application Engineering*, CSAE '18, page 1–5. Association for Computing Machinery, 2018.

[10] Ze Deng, Lizhe Wang, Wei Han, Rajiv Ranjan, and Albert Zomaya. G-ML-Octree: An Update-Efficient Index Structure for Simulating 3D Moving Objects Across GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):1075–1088, 2018.

[11] Jens Dittrich, Lukas Blunschi, and Marcos Antonio Vaz Salles. Indexing Moving Objects Using Short-Lived Throwaway Indexes. In *Advances in Spatial and Temporal Databases*, pages 189–207, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[12] KS Easwarakumar and T Hema. Bits-tree-an efficient data structure for segment storage and query processing. *arXiv preprint arXiv:1501.03435*, 2015.

[13] Raphael A. Finkel and Jon Louis Bentley. Quad trees - a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[14] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, June 1998.

[15] Irene Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, 1982.

[16] T. M. Ghanem, , M. F. Mokbel, W. G. Aref, and J. S. Vitter. Bulk operations for space-partitioning trees. In *Proceedings. 20th International Conference on Data Engineering*, pages 29–40, April 2004.

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

[18] T. Hema and K. S. Easwarakumar. On higher dimensional window query: Revisited using bits-tree. In *Proceedings of the International Conference on Informatics and Analytics*, ICIA-16, pages 91:1–91:6, New York, NY, USA, 2016. ACM.

[19] Y. Hu, L. Luo, and L. Yin. Xq-index: A distributed spatial index for cloud storage platforms. In *2018 7th International Conference on Agro-geoinformatics (Agro-geoinformatics)*, pages 1–6, Aug 2018.

[20] Anand Padmanabha Iyer and Ion Stoica. A scalable distributed spatial index for the internet-of-things. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 548–560. ACM, 2017.

[21] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An improved R-tree using fractals. Technical report, 1993.

[22] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In *Proceedings of the second international conference on Information and knowledge management*, CIKM93, page 490–499. Association for Computing Machinery, 1993.

[23] G. Kedem. The Quad-CIF Tree: A Data Structure for Hierarchical On-Line Algorithms. In *19th Design Automation Conference*, pages 352–357, June 1982.

[24] Kihong Kim, Sang K. Cha, and Keunjoo Kwon. Optimizing Multidimensional Index Trees for Main Memory Access. *SIGMOD Rec.*, 30(2):139–150, May 2001.

[25] Kyung-Chang Kim and Suk-Woo Yun. Mr-tree: A cache-conscious main memory spatial index structure for mobile gis. In *Web and Wireless Geographical Information Systems*, pages 167–180, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[26] Mong Li Lee, Wynne Hsu, Christian S Jensen, Bin Cui, and Keng Lik Teo. Supporting frequent updates in R-trees: A bottom-up approach. In *Proceedings 2003 VLDB Conference*, pages 608–619. Elsevier, 2003.

[27] Ahmed R. Mahmood, Sri Punni, and Walid G. Aref. Spatio-temporal access methods: a survey (2010 - 2017). *GeoInformatica*, Oct 2018.

[28] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N Papadopoulos, and Yannis Theodoridis. R-trees have grown everywhere. Technical report, Technical Report available at http://www. rtreeportal. org, 2003.

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

[30] Krassimir Markov, Krassimira Ivanova, Ilia Mitov, and Stefan Karastanev. Advance of the access methods. 2008.

[31] Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

[32] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-temporal access methods. *IEEE Data Eng. Bull.*, 26(2):40–49, 2003.

[33] Long-Van Nguyen-Dinh, Walid G Aref, and Mohamed Mokbel. Spatio-temporal access methods: Part 2 (2003-2010). 2010.

[34] Matthaios Olma, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. BLOCK: Efficient Execution of Spatial Range Queries in Main-Memory. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, page 1–12. Association for Computing Machinery, 2017.

[35] Jack A Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, 1982.

[36] Octavian Procopiuc, Pankaj K. Agarwal, Lars Arge, and Jeffrey Scott Vitter. Bkd-Tree: A Dynamic Scalable kd-Tree. In *Advances in Spatial and Temporal Databases*, pages 46–65, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[37] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw Hill, 2000.

[38] P. Rigaux, M. Scholl, and A. Voisard. *Spatial Databases with Application to GIS*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[39] John T. Robinson. The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, pages 10–18, New York, NY, USA, 1981. ACM.

[40] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.*, 16(2):187–260, June 1984.

[41] Hanan Samet, Jagan Sankaranarayanan, and Michael Auerbach. Indexing methods for moving object databases: Games and other applications. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 169–180, New York, NY, USA, 2013. ACM.

[42] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree: A Dynamic Index for Multi-Dimensional Objects. Technical report, 1987.

[43] Darius Šidlauskas, Kenneth A. Ross, Christian S. Jensen, and Simonas Šaltenis. Thread-Level Parallel Indexing of Update Intensive Moving-Object Workloads. In *Advances in Spatial and Temporal Databases*, pages 186–204, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[44] Darius Šidlauskas, Simonas Šaltenis, Christian W Christiansen, Jan M Johansen, and Donatas Šaulys. Trees or grids?: indexing moving objects in main memory. In *Proceedings of the 17th ACM SIGSPATIAL international conference on Advances in Geographic Information Systems*, pages 236–245. ACM, 2009.

[45] Yasin N Silva, Xiaopeng Xiong, and Walid G Aref. The RUM-tree: supporting frequent updates in r-trees using memos. *The VLDB Journal—The International Journal on Very Large Data Bases*, 18(3):719–738, 2009.

[46] Julio Toss, Cícero AL Pahins, Bruno Raffin, and João LD Comba. Packed-memory quadtree: a cache-oblivious data structure for visual exploration of streaming spatiotemporal big data. *Computers & Graphics*, 76:117–128, 2018.

[47] Thatcher Ulrich. Loose octrees. *Game Programming Gems*, 1:434–442, 2000.

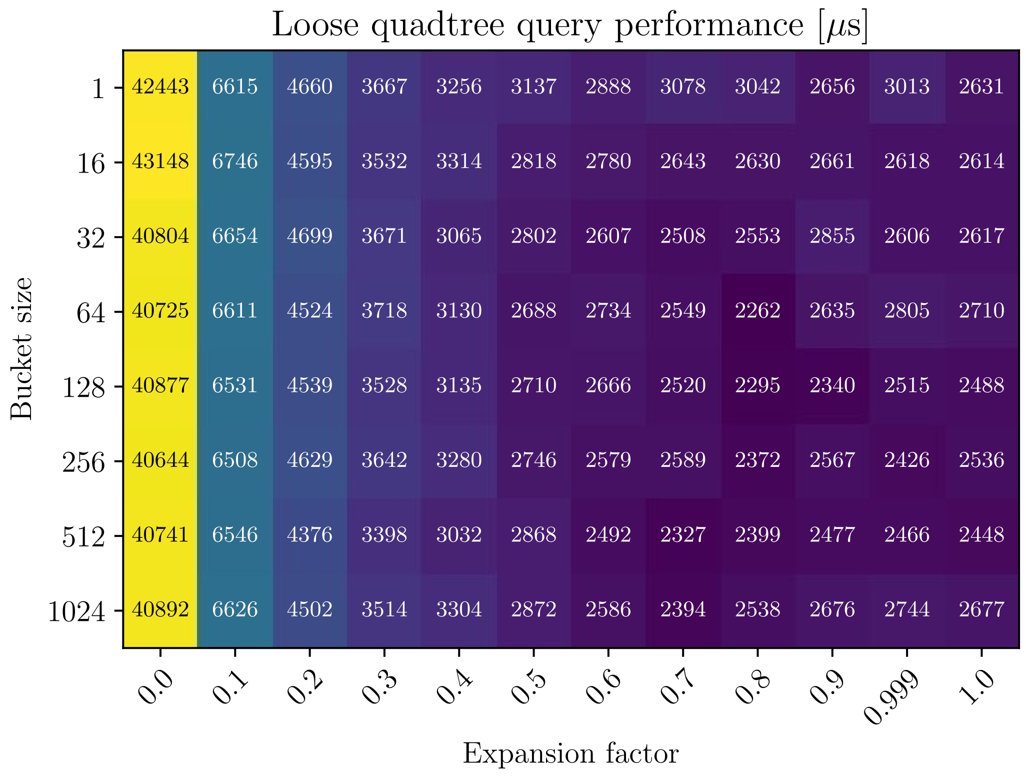
[48] Tilmann Zäschke, Christoph Zimmerli, and Moira C. Norrie. The PH-tree: A Space-efficient Storage Structure and Multi-dimensional Index. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 397–408, New York, NY, USA, 2014. ACM.

[49] Yuean Zhu, Shan Wang, Xuan Zhou, and Yansong Zhang. Rum+-tree: A new multidimensional index supporting frequent updates. In *Web-Age Information Management*, pages 235–240, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[50] Darius Šidlauskas, Simonas Šaltenis, and Christian S. Jensen. Parallel main-memory indexing for moving-object query and update workloads. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD/PODS '12, page 37–48. Association for Computing Machinery, 2012.

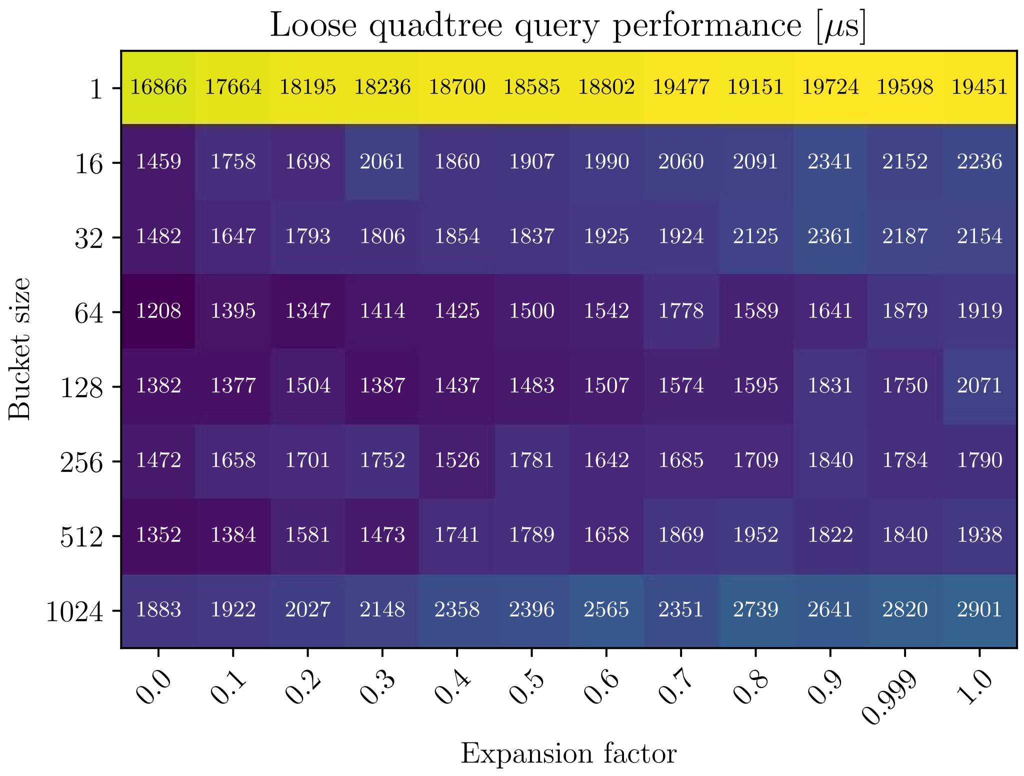
A

Appendix 1



Bucket size	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.999	1.0
1	42443	6615	4660	3667	3256	3137	2888	3078	3042	2656	3013	2631
16	43148	6746	4595	3532	3314	2818	2780	2643	2630	2661	2618	2614
32	40804	6654	4699	3671	3065	2802	2607	2508	2553	2855	2606	2617
64	40725	6611	4524	3718	3130	2688	2734	2549	2262	2635	2805	2710
128	40877	6531	4539	3528	3135	2710	2666	2520	2295	2340	2515	2488
256	40644	6508	4629	3642	3280	2746	2579	2589	2372	2567	2426	2536
512	40741	6546	4376	3398	3032	2868	2492	2327	2399	2477	2466	2448
1024	40892	6626	4502	3514	3304	2872	2586	2394	2538	2676	2744	2677

Figure A.1: Query performance of the loose quadtree for different expanson factors and bucket sizes, on a scenario with uniformly distributed, **large** objects. The evaluation was performed with 10 million objects.



Bucket size	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.999	1.0
1	16866	17664	18195	18236	18700	18585	18802	19477	19151	19724	19598	19451
16	1459	1758	1698	2061	1860	1907	1990	2060	2091	2341	2152	2236
32	1482	1647	1793	1806	1854	1837	1925	1924	2125	2361	2187	2154
64	1208	1395	1347	1414	1425	1500	1542	1778	1589	1641	1879	1919
128	1382	1377	1504	1387	1437	1483	1507	1574	1595	1831	1750	2071
256	1472	1658	1701	1752	1526	1781	1642	1685	1709	1840	1784	1790
512	1352	1384	1581	1473	1741	1789	1658	1869	1952	1822	1840	1938
1024	1883	1922	2027	2148	2358	2396	2565	2351	2739	2641	2820	2901

Figure A.2: Query performance of the loose quadtree for different expanson factors and bucket sizes, on a scenario with uniformly distributed, **small** objects. The evaluation was performed with 10 million objects.

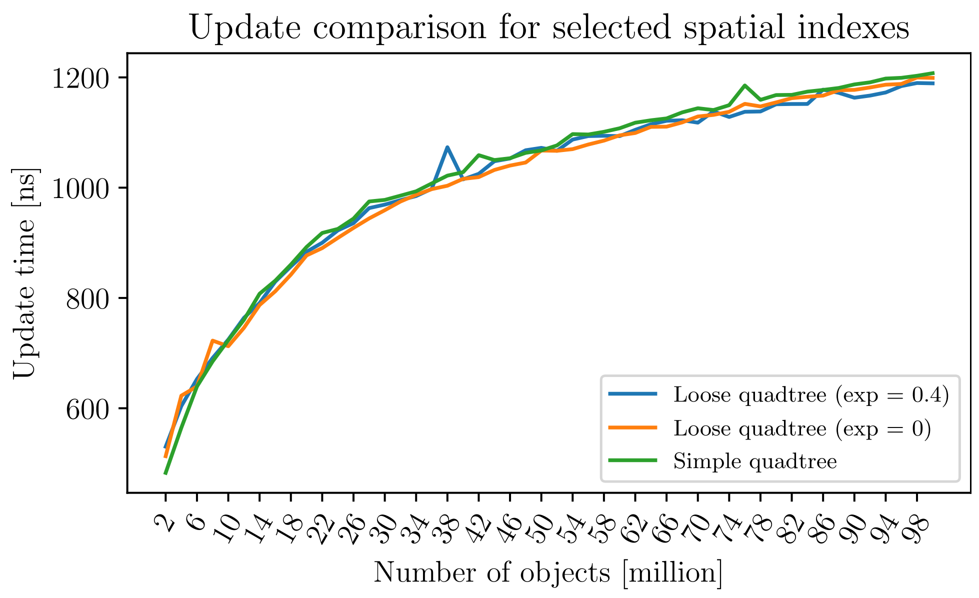


Figure A.3: This graph shows how update performance is affected by using either the simple quadtree or the loose quadtree, with another node management method.

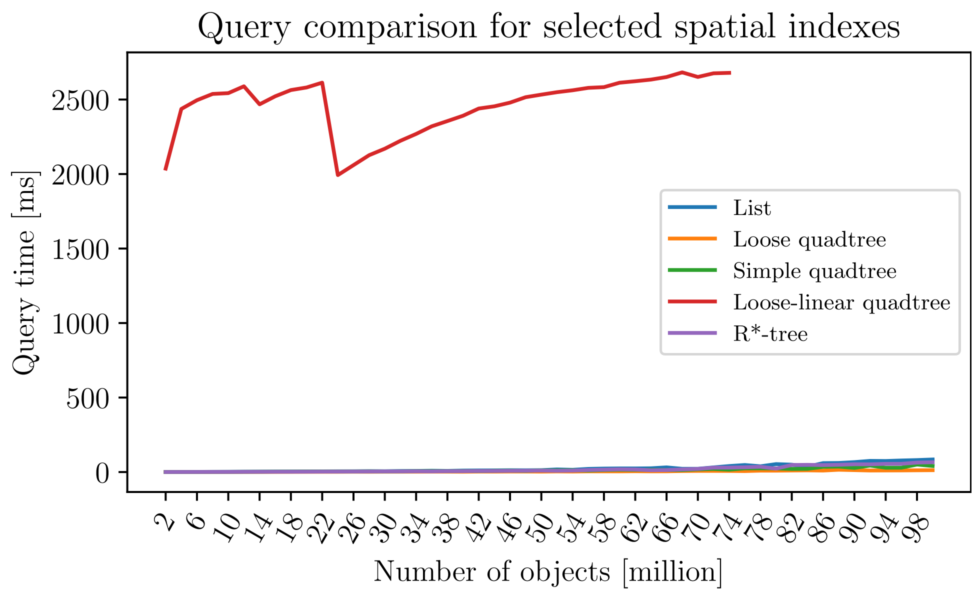


Figure A.4: Query comparison between selected spatial indexes, with the loose-linear quadtree included.

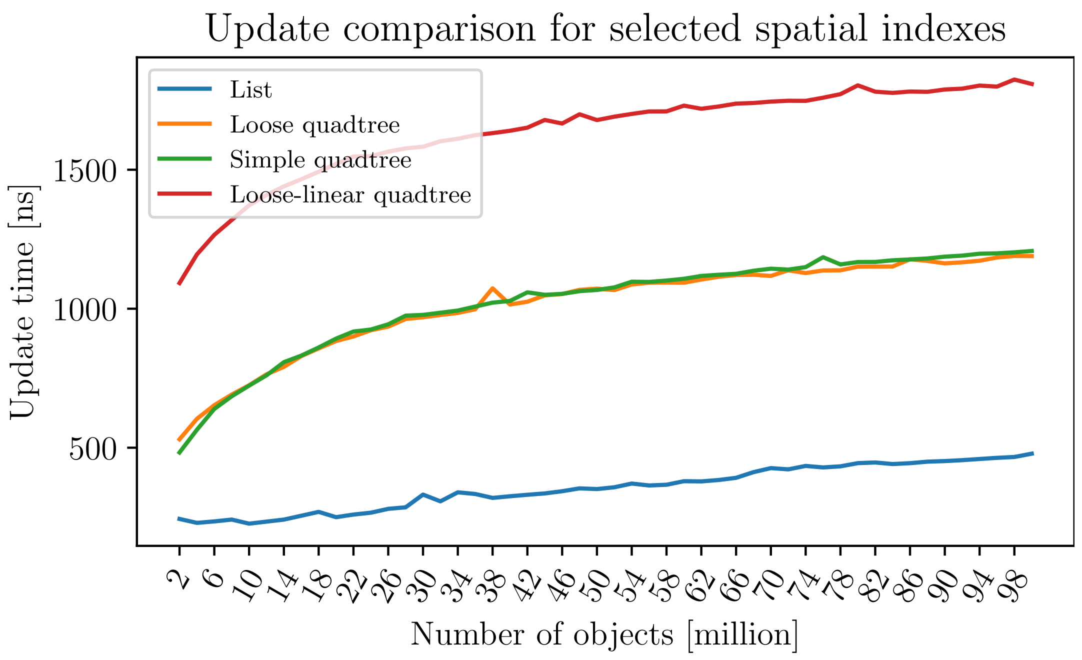


Figure A.5: This graph is a closer look at update performances without the R*-tree.