Real-Time Indexing of Arbitrarily Attributed Point Clouds

Bachelor thesis by Paul Hermann
Date of submission: January 17, 2025

Reviewer: Prof. Dr. Arjan Kuijper
First Supervisor: Dr. Michel Krämer
Second Supervisor: Tobias Dorra
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Fraunhofer IGD

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Paul Hermann, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 17. Januar 2025

P. Hermann

Abstract

The demand for up-to-date airborne or terrestrial Light Detection and Ranging (LiDAR) data sets is increasing. This works by emitting a laser beam from a scanner. Based on the time of flight, the distance between each reflection point and scanner can be calculated, resulting in 3D point clouds. These point clouds are typically stored unindexed during acquisition. After acquisition, processing and indexing algorithms have to be run as time-consuming batch processes. To avoid this post-processing, the necessary processing steps can be implemented in real-time pipelines to further process and store data during acquisition. Not only spatial indexing but also indexing of many possible attributes of LiDAR points play an important role in applications such as data quality assurance or real-time preview.

In this thesis, a real-time approach for attribute indexing of LiDAR point clouds is presented, which extends the spatial index structure of Modifiable Nested Octrees with attribute indexing. The attribute index is implemented by storing attribute ranges contained in each subtree. In addition to spatial restrictions, attribute filters can be defined in the form of attribute-value ranges for queries. Nodes whose ranges do not overlap with the attribute filters can then be sorted using the attribute ranges stored in the octree nodes. An implementation of the approach is presented and measured using a terrestrially acquired dataset of 365 million points. Indexing speeds of over 400 000 points per second can be achieved, and the time for queries with attribute filters can be reduced to 1%-50% of the time required for linear filtering of all indexed points, while on average 33% of all unwanted points can be discarded. These values are achievable on simple consumer hardware, provided a fast hard disk is available.

Zusammenfassung

Der Bedarf an aktuellen luftgestützten oder terrestrischen LiDAR-Datensätzen steigt. Um diese Datensätze zu erfassen, werden Laserstrahlen von einem Scanner ausgesandt. Anhand der Lichtlaufzeit kann die Entfernung zwischen jedem Reflexionspunkt und dem Scanner berechnet werden, woraus 3D-Punktwolken hervorgehen. Die resultierenden Punktwolken werden in der Regel während der Erfassung unindexiert gespeichert. Nach der Erfassung müssen die Verarbeitungs- und Indexierungsalgorithmen als zeitaufwändige Batch-Prozesse ausgeführt werden. Um diese Nachbearbeitung zu vermeiden, können die notwendigen Verarbeitungsschritte in Echtzeit-Pipelines implementiert werden, um die Daten während der Erfassung weiter zu verarbeiten und zu speichern. Nicht nur die räumliche Indexierung, sondern auch die Indexierung der vielen möglichen Attribute von LiDAR-Punkten spielen eine wichtige Rolle in Anwendungen wie der Datenqualitätssicherung oder der Echtzeitvorschau.

In dieser Arbeit wird ein Echtzeit-Ansatz zur Attribut-Indexierung von LiDAR-Punktwolken vorgestellt, der die räumliche Indexstruktur von Modifiable Nested Octrees um die Attribut-Indexierung erweitert. Die Attribut-Indexstruktur wird durch die Speicherung der in jedem Teilbaum enthaltenen Attributbereiche implementiert. Zusätzlich zu den räumlichen Einschränkungen können Attributfilter in Form von Attributwertbereichen für Abfragen definiert werden. Knoten, deren Bereiche sich nicht mit den Attributfiltern überschneiden, können dann anhand der in den Octree-Knoten gespeicherten Attributbereiche sortiert werden. Eine Implementierung des Ansatzes wird vorgestellt und anhand eines terrestrisch erfassten Datensatzes von 365 Millionen Punkten gemessen. Es können Indexierungsgeschwindigkeiten von über 400.000 Punkten pro Sekunde erreicht werden, und die Zeit für Abfragen mit Attributfiltern kann auf 1%-50% der Zeit reduziert werden, die für die lineare Filterung aller indexierten Punkte benötigt wird, während im Durchschnitt 33% aller unerwünschten Punkte verworfen werden können. Diese Werte sind auf einfacher Consumer-Hardware erreichbar, sofern eine schnelle Festplatte vorhanden ist.

Contents

1 Introduction

Point clouds are becoming increasingly important in the dynamic field of urban and environmental planning. Using LiDAR technology, terrestrial [**wu_application_2021**] or airborne [**li_airborne_2021**] LiDAR scans can be used to acquire three-dimensional data sets. For example, architects and urban planners [**pyszny_lidar_2020**] use LiDAR-generated point clouds for BIM processes [**valero_laser_2022**] in building projects or for the planning of infrastructure in cities. LiDAR applications such as extending fiber optic networks, district heating planning [**lumbreras_design_2022**], or flood modeling [**li_application_2021**] are currently becoming more relevant. Beyond city limits, airborne LiDAR scans are often used for creating digital elevation models [**pfeifer_geometrical_2007**] and for mapping and regular monitoring of forest and agricultural stands [**alvites_lidar_2022**] or railway lines [**ton_semantic_2022**]. For all of these applications, recorded data must be stored permanently. To enable accelerated queries at a later time, the data has to be stored in index structures, which is a computationally intensive task.

In most cases, the data needs to be as up-to-date as possible, which is why scans are being performed more frequently. Therefore, it is essential to minimize post-processing times or eliminate them. For this purpose, this work presents an approach to perform the time-intensive indexing in real-time during data acquisition, instead of performing it in a batch-process after scanning. Two significant challenges arise: The first challenge is that the data volumes become quite substantial. A terrestrial dataset from the city of Frankfurt is used for this thesis, which has an uncompressed size of 12.42 GB for a street length of 3km. Datasets of this size, or even larger, usually do not fit into main memory and a linear search over all points would take a significant amount of time. In order to enable fast queries on point clouds, out-of-core index structures are required. Besides its spatial coordinates, each point of a point cloud can have additional attributes that are either generated directly during the acquisition (e.g., intensity) or calculated in a post-processing step (e.g., classification). Not only spatial queries but also filters on the attributes of individual points are important for many applications and need to be supported by these index structures. The second major challenge in real-time processing

is the data rate during acquisition. For terrestrial applications, the widely used Velodyne Puck LiDAR scanner captures up to 600 000 points per second [**_velodyne_2019**]. The airborne RIEGL VQ-1560 scanner can acquire more than 10 million points per second [**_riegl_2022**]. The indexing algorithm has to keep up with these speeds, and therefore it has to be able to handle a higher point rate than the scanner in order to enable real-time processing.

1.1 Baseline

It is important to consider where such an index structure can be used in practice for the acceleration of spatial queries and attribute filters. Therefore, it is necessary to identify the data that may be contained in each point. The data can be categorized into four areas.

Spatial Data
Each point contains a three-dimensional spatial coordinate calculated from the distance between the point and position of the scanner during the LiDAR scan. In addition, the assignment of a Level of Detail (LOD) to each point can be useful. These represent the resolution levels of point clouds. Level zero corresponds to the minimum resolution level. This level contains only a few points, resulting in a low density of points for this LOD. This feature can be useful in rendering areas that are further away from the viewpoint, with a lower resolution level.

Raw Data Recordings
Time-of-flight LiDAR methods use the travel time of the laser beam to determine the distance between a point and the sensor. In addition to the distance, other data such as the intensity of the reflection is recorded. Moreover, sometimes the laser beam hits an edge, resulting in reflecting both the edge and objects behind the edge. In other cases, the laser beam is aimed at translucent materials, such as the leaves of a tree. This is where the first reflection occurs. Objects behind it may cause additional reflections, which can also be detected. To store this information, both the return number for each point and the total number of returns for the corresponding laser beam are stored. Apart from time-of-flight-based LiDAR scanners, alternative technologies based on phase offsets are also used. These methods generate a large volume of data, including waveforms that correspond to each laser beam.

Metadata from the Scanner
Many LiDAR scanners use rotating or tilting mirrors to move a line of laser beams

in different directions. This produces data such as the current rotation angle or the direction in which the mirrors are moving. If a single mirror deflects several laser beams, it is also possible to record which beam was used to detect the point. Additionally, LiDAR scanners are usually equipped with GPS antennas to locate the scanner. The current GPS time stamp is recorded and can be stored separately for each point.

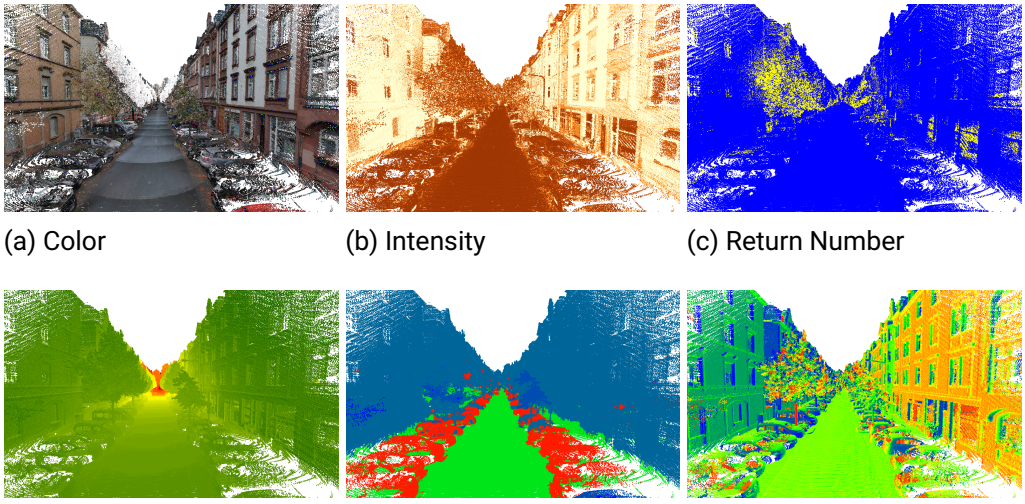
Processed Data
After capturing, additional attributes may be added to the points of a point cloud. This can be achieved either through batch processing after acquisition or by using real-time pipelines. Numerous developments, such as FPGA-based real-time computing units, exist specifically for LiDAR applications in autonomous driving [**bai_pointnet_2020**]. In addition, photographs of the surroundings are frequently taken alongside LiDAR scans to project these onto point clouds and thereby colorize the points. Moreover, the calculation of which objects a particular point belongs to (e.g., ground, cars, or vegetation) is possible. This classification can be stored by assigning a category number to each point. For future applications such as meshing, it can be useful to calculate and store a normal vector for each point using the positions of surrounding points [**purschke_entwicklung_2023**].

Some examples of these point cloud attributes are shown in Figure **??**, where points are colored according to their attributes.

1.2 Motivation

To illustrate the benefits of real-time indexing and the real-world applications of attribute filters, we consider a vehicle-based LiDAR scanning system with a LiDAR scanner mounted on the roof of the vehicle. We assume that a real-time processing pipeline has already classified the data from this scanner and computed its normals. The data is indexed in real-time while the vehicle is moving, and the index structure is stored on a local memory in the vehicle. Data is immediately available and can be used immediately after or during driving. Batch indexing after the journey is not required. The system is shown in Figure **??**.

The indexed data enables real-time visualization of the data for the driver or co-driver. A tablet can be installed in the cab to display the recorded data and ensure the quality of the recorded data. This way, errors or areas of poor coverage can be detected while the vehicle is still on the road. Such tablets have very limited graphics memory and bandwidth



(a) Color (b) Intensity (c) Return Number

(d) Time (e) Classification (f) Normal (X-Axis)

Figure 1.1: Attributes in Frankfurt Testdata

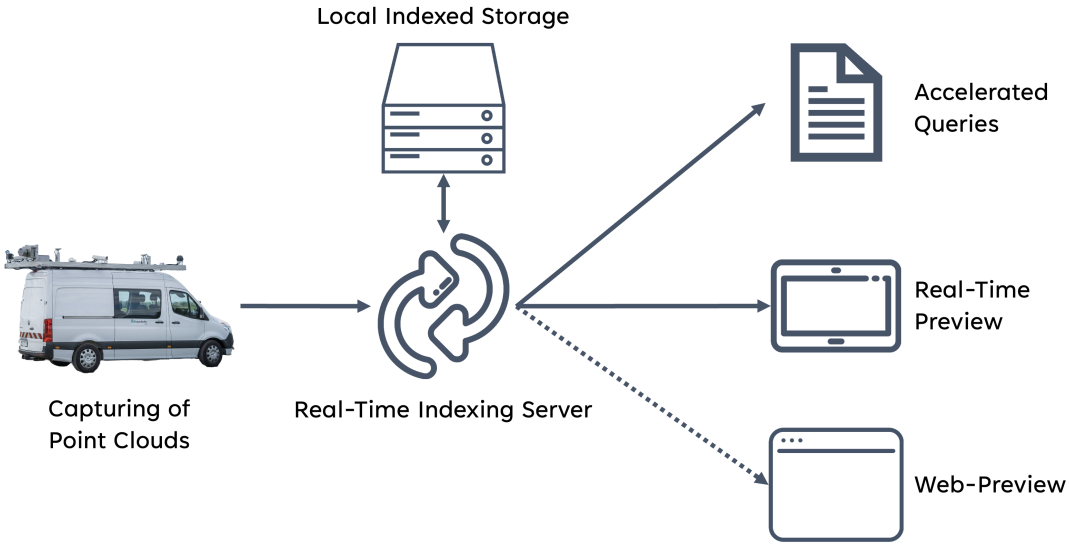


Figure 1.2: Example System

to the index server. It is therefore important to ensure that only the visible points are transmitted to the tablet, with distant points displayed at low LOD. To further reduce the number of points to be transmitted, indexing and server-side filtering of attributes is interesting at this point. This way there will be no sending of points that are of no interest to the tablet. In the case of overlapping travel areas, time filtering can be used to display only the points captured in the last few minutes. Leaves of trees or reflections from raindrops can be filtered out using the Return Number and Number of Returns or the classification values. When only the road surface is of interest, all points whose normal do not point vertically upwards or which have not been classified as a road can be filtered out. Even after the drive has been made, points of a specific period of time can be checked, e.g., if it has rained during a particular period and the correct recording needs to be ensured. There is also the option of sending points to the internet in real-time via 5G technology, in addition to displaying them on a tablet. This could enable the generation of a topographic preview map or the usage of larger servers for more complex real-time post-processing. Again, due to high bandwidth limitations, it is beneficial if points can be pre-filtered onboard the vehicle based on their attributes before being sent to the Internet. Finally, the index structure offers the benefit that on-board processing pipelines can use the index structure directly. For example, because traffic signs are highly reflective, traffic sign recognition algorithms could use the intensity attribute to detect them. The index structure can quickly provide the points searched for.

1.3 Goals

The goal of this work is to extend an existing real-time spatial indexing approach with attribute indexing capability. To achieve this objective, various spatial and attribute index structures are compared, placing particular emphasis on real-time capability. The implementation of the selected approach is outlined and measurement results for indexing and queries are presented. Possibilities for further optimization are presented as future work. The main requirements for the developed index structure are as follows.

1. **Real-Time Indexing**
   The indexing speed measured in points per second must be higher than the speed of conventional LiDAR scanners to enable indexing in real-time. In the terrestrial test dataset used, the maximum point throughput is around 400 000 points per second. This number is defined as a lower limit for real-time indexing in this work.

2. **Query Time Acceleration**
   The query time for attribute filtering when using the attribute index structure should be faster than linear filtering without the index structure.

3. **Query False Positive Point Reduction**
   A certain proportion of points are not searched for but still have to be loaded from memory. These are false positives. The attribute index structure should be able to sort out at least half of all non-searched points before linear filtering. If the subsequent linear filtering is done on the client side rather than the server side, this false positive point reduction can save additional bandwidth or graphics memory on the clients.

4. **Low Computational Load**
   The algorithms should be able to achieve the required indexing and query speeds on simple and energy-efficient hardware so that they can be used on the move without requiring large power capacities. The existing hardware must be used with maximum efficiency.

1.4 Outline

In Chapter **??**, different spatial data structures as well as attribute indexing data structures are compared and examined. The approach of the data structure *Min Max Modifiable Nested Octree (M^3NO)* developed in this thesis for real-time spatial and attribute indexing is explained in Chapter **??**, divided into spatial index and attribute index. An additional acceleration stage using histograms is presented in Section **??**. The implementation of the data structure based on the Lidarserv software is described in Chapter **??**, where the software architecture as well as the indexing process and the query process are explained. In Chapter **??**, the performed measurements are explained and discussed. For this purpose, the methodology and the used methods are explained in Sections **??** and **??**, the measurement results are presented in Section **??** and discussed in Section **??**. Finally, a summary is presented in Section **??**, as well as future work and other open research topics in this area.

2 Related Work

Real-time indexing of point clouds should not be confused with real-time processing of point clouds, which is highly relevant in the fields of autonomous driving and robotics. It has been the subject of many scientific papers. However, real-time processing is only of limited relevance and use in this work, as the point clouds are not stored and are only processed in main memory. Nevertheless, sub-areas of this research are of interest for future work. Classification, normalization, or coloring methods in real-time can be useful pipeline stages before or after real-time indexing [**bai_pointnet_2020**, **chen_parallelnn_2023**].

Research in the area of real-time indexing of arbitrarily attributed point clouds can be divided into three sub-areas: Spatial Index Structures, Attribute Index Structures, and Real-Time Capability of these Index Structures (Figure **??**). The intersection of all three is the subject of this thesis.

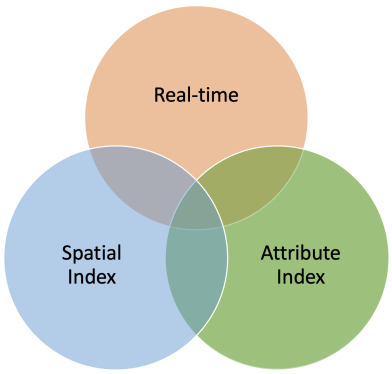


Figure 2.1: Related Work Areas

In the following Section, the different approaches of the subdomains are examined concerning the four goals set: **real-time indexing**, **query time acceleration**, **query false positive point reduction**, and **low computational load**.

2.1 Spatial Index Structures and Real-Time Capability

Spatial indexing structures are data structures that divide three-dimensional space into different sub-spaces to organize the points into these sub-spaces. Spatial queries can then be accelerated using these data structures. Common types of spatial queries include

Axis-Aligned Bounding Box (AABB) Queries
All points within an axis-aligned bounding box are queried. Optionally, a maximum level of detail to be queried can be specified. This type of query is used to retrieve specific relevant subareas from large datasets.

View Frustum Query
For the visualization of the data, a camera field of view is defined, consisting of the viewing axis and the opening angle of the camera. Nearer areas are queried with a higher level of detail, while more distant areas are queried with a lower level of detail.

K-Nearest Neighbors Query
For many algorithms (e.g., normal computation), the k nearest points to a defined 3D position are required and queried.

The suitability of the data structure for real-time indexing is also discussed in the following comparison of spatial data structures. There are three important requirements that the index structure must meet for real-time indexing. Because new points are constantly arriving, the points cannot be pre-sorted and the bounding box of the point cloud is never known, because it can grow with new points being added. Since real-time capability requires indexing of at least as many points per second as the scanner can acquire per second, not too many expensive rebalancing operations must be performed during indexing. The rebalancing of the index tree structures can increase query speeds but, conversely, also causes indexing speeds to slow down.

2.1.1 Space-Filling Curves

Space-filling curves are complete traversals of a multidimensional space. Each point in space can be described one-dimensionally by a curve index. Most space-filling curves are designed in such a way that index values that are close to each other have a certain locality in multidimensional space. They can be used for point cloud indexing by mapping the multidimensional coordinates of points to a linear, one-dimensional space

[**guan_parallel_2018**]. This mapping allows for efficient range queries and nearest neighbor searches by exploiting the inherent locality preservation of the curve. Since more than three dimensions can easily be mapped, the level of detail can also be used as an additional dimension. This allows the implementation of continuous levels of detail (with many very small steps) instead of having larger discrete differences between levels of detail as in Modifiable Nested Octrees [**liu_optimized_2020**]. Commonly used types of space-filling curves include Morton (Z-order) and Hilbert codes. These make it possible to calculate codes and index points in real-time in a massively parallel manner without any problems, enabling efficient point cloud indexing [**kocon_point_2021**].

2.1.2 k-d Trees

In k-d trees, each node represents a plane, that subdivides the space in one of the dimensions, splitting the space into two halves [**bentley_multidimensional_1975**]. If the partition plane is chosen based on the distribution of points in the space, the result is a well-balanced tree. However, since this requires that all points are already sorted based on the corresponding dimension, this data-dependent partitioning is not an option for real-time indexing. Instead, real-time indexing either results in more unevenly distributed trees or requires many expensive rebalancing options. This makes k-d trees unsuitable for this application.

2.1.3 R-Trees

R-trees are similar to one-dimensional B-trees but in three dimensions [**guttman_rtrees_1984**]. Each node represents a bounding rectangle that encloses a group of points or other bounding rectangles. However, since the bounding rectangles are created in a data-dependent manner, the problem arises, as with k-d trees, that real-time indexing cannot be efficiently implemented.

2.1.4 Octrees

Octrees divide the space, independent of the points to be indexed, centrally in all three axes into eight subspaces, which can then be further divided recursively [**meagher_octree_1982**, **finkel_quad_1974**]. This approach does not require any rebalancing operations. However, the tree is not well balanced if the data is unevenly

distributed. There are several variants of octrees for the indexing of point clouds, which differ mainly in the position within the tree where the points are stored, and in the way in which the points are stored.

2.1.5 Modifiable Nested Octrees (MNOs)

One type of octree for point cloud indexing is the Modifiable Nested Octree (MNO) [**scheiblauer_interactions_2014**]. Here, each node of the tree can contain a set of points, for example, stored in a regular three-dimensional grid. This not only results in a spatial subdivision of the points but also in a level of detail subdivision. The points contained in the root node correspond to the lowest level of detail (LOD 0), and each lower level in the tree corresponds to a higher level of detail. However, MNOs are not capable of real-time indexing because, as with all octrees, the bounding box of the data is required in advance to define the fixed planes for subdivision.

A solution to this problem was presented by **kocon_progressives_2021** [**kocon_progressives_2021**] and adapted for MNOs by **dorra_indexing_2022** [**dorra_indexing_2022**]: In a large regular grid, an arbitrary number of MNOs is created. As soon as points fall into a new cell of the grid, a new MNO is created for that grid cell. Figure **??** illustrates this for a two-dimensional point cloud. In the upper region, three root nodes are generated, corresponding to the lowest LOD. In the higher LODs, many smaller nodes contain more points in total and thus provide a higher level of detail. Each node in the Figure contains a regular grid of size 4x4, where the points of the node are stored.

2.2 Attribute Index Structures and Real-Time Capability

Current research in the field of real-time indexing is largely limited to spatial indexing. Real-time attribute indexing has not been investigated. Therefore, the methods and data structures listed below are examined for both the ability to index many attributes and for possible real-time capability extensions.

The methods for attribute indexing fall into three categories. First, there are methods that create a completely independent index structure for attribute indexing. Second, there are data structures where a single index structure can be used for both spatial and attribute

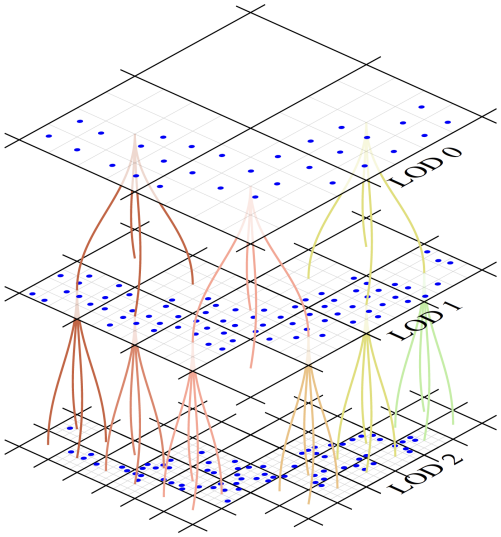


Figure 2.2: Modifiable Nested Octree from **bormann_realtime_2022** [**bormann_realtime_2022**]

indexing. Third, there are methods where the attribute index is closely related to the spatial index structure.

2.2.1 Independent Attribute Index Structures

One possible approach to the creation of separate index structures for spatial coordinates and multiple attributes is presented in a point cloud database system [**dobos_point_2014**]. A traditional relational column-based database model is presented in which the three-dimensional coordinates and all attributes are stored in different columns, each row represents a single point. A primary database index is then created on the coordinates and further indexes can be created on the attributes. However, this has the drawback that each index structure must be evaluated separately for a query and then the intersection of the results must be formed, which involves a relatively large overhead and is therefore not chosen for this work.

2.2.2 Combined Index Structures

Combined spatial and attribute index structures are essentially n-dimensional index structures. Many approaches implement such n-dimensional (nD) index structures using a Space Filling Curve (SFC), which can map high-dimensional data to one dimension. For point clouds, temporal indexing is most often used in addition to spatial indexing, sometimes to map moving points [**kim_utilizing_2021**]. However, it turns out that the SFC approach does not scale well to an arbitrary number of attributes, since from about 6 dimensions on it is hardly possible to sort out points during queries, and the false positive rate increases to almost 100% [**liu_optimized_2020**]. To deal with this problem, some approaches automatically rank which attributes in a point cloud are most important to index, based on test queries, and should therefore be included in the index structure [**nathan_learning_2020**]. However, space-filling curves limited to indexing only a small number of attributes are unsuitable for this work.

2.2.3 Attribute Index Structures Related to Spatial Index Structures

An example of an attribute index structure that is closely related to the spatial index structure is the Binned Min Max Quadree [**ladra_compact_2022**]. Here, points are spatially sorted into an octree data structure based on their coordinates. The value ranges of the underlying subtree are then stored in each node for all attributes. This allows to sort out entire subtrees for queries on attributes that do not contain the desired value range. Binned Min Max Octrees are not yet real-time capable. This is because the octree structure is not real-time capable due to the unknown bounding box size and because the attributes are first sorted and histogrammed, and then the histogram bin ranges are stored for the subtrees. However, the basic idea of binned min-max octrees serves as a basis for this work.

2.3 Research Gap

It can be seen that there are a variety of approaches to spatial indexing, very few of which support real-time indexing. For attribute indexing, there are some approaches in the three categories listed, but they are either not real-time capable or they only provide support for a very small number of attributes. In this thesis, I present a novel indexing structure that supports both spatial indexing and indexing of an arbitrary number of attributes

in real-time. It is designed to combine the most promising approaches to spatial and attribute indexing: Modifiable Nested Octrees for spatial indexing and Binned Min Max Octrees for attribute indexing. The resulting index structure is called *Min Max Modifiable Nested Octree (M^3NO)*. The following Chapter describes the approach and structure of this index structure.

3 M^{3}NO Indexing Data Structure

This chapter presents the developed data structure for spatial indexing of point clouds and attribute indexing of any number of arbitrary point attributes in real-time. For the development of the data structure, different spatial index structures and attribute index structures were examined in Chapter **??** concerning the goals set in Section **??**. The lack of approaches that combine spatial indexing of arbitrary attributes, attribute indexing, and real-time capability was identified as a research gap in Section **??**. The proposed index structure closes this gap and is composed of two parts. The real-time capable spatial index structure is taken largely unchanged from **dorra_indexing_2022** [**dorra_indexing_2022**]. It is then extended using the index structure for attribute indexing by **ladra_compact_2022** [**ladra_compact_2022**] with a slight modification for real-time capability. The index structure is described in detail in the following Sections.

3.1 Spatial Index

The spatial data structure consists of the following elements. There is an arbitrary number of **MNO trees** in a regular grid of a fixed configurable size. Each MNO consists of **octree nodes**, which recursively divide the space into eight subspaces. Every node contains a **regular grid** of the same size for storing points. The number of grid cells should be $128 \times 128 \times 128$ according to the recommendation of **schutz_potree_2016** [**schutz_potree_2016**], which also gave the best results in this implementation. The fixed size of the grid results in a low LOD in the root node. In deeper levels of the tree, the distance between the grid cells decreases, and the LOD, and thus the resolution of the points, increases. Each octree node has an **inbox** where new points are placed. A configurable number of **worker threads** then take these points from the inboxes and insert them into the regular grids of the corresponding octree nodes. If multiple points fall into the same grid cell, the point closest to the center of the cell is inserted into

the cell (**grid-center-sampling**). The other points that would fall into this cell are each added to the inbox of the corresponding one of the eight child nodes of the octree node. A **file-based** approach is used to store the nodes. Each octree node is stored as a separate file in LAS or LAZ format on disk out of core, since the total size of the point cloud may exceed the size of the main memory. To avoid slow writes and reads to the disk, a **least recently used cache (LRU)** implementation is provided to keep frequently used nodes in memory. Only nodes that have not been used for a long time are flushed to disk. The maximum number of nodes in the cache is configurable. As an additional level of optimization, the concept of **bogus points** was implemented. Here, after the grid center sampling step, a fixed number of points is stored in the same node up to a defined number. Only when this number is exceeded, the bogus points are inserted into the inboxes of the child octree nodes. This can significantly reduce I/O operations, but it will cause irregularities in the LOD levels. The selected attribute indexing may also be negatively affected.

3.2 Attribute Index

This spatial index structure is extended for attribute indexing by storing, for each subtree of the octree, which attribute ranges (minimum and maximum) of each attribute are contained in it. Thus, each node stores a set of attribute ranges that are contained in the points of the corresponding subtree. Unlike proposed in the approach of **ladra_compact_2022** where the ranges of used histogram bins were stored for each node, the values are neither presorted nor inserted into histograms. Instead, the value ranges are stored directly, ensuring real-time capability. The construction of this data structure is integrated with spatial indexing. When there are points in an inbox of an octree node, the worker thread scheduled for that node first computes the value ranges of all attributes based on the points in the inbox. These computed attribute ranges are then used to expand and update the attribute ranges of the octree node. All nodes that match both spatial queries (AABB or View Frustum) and attribute filters for queries are loaded from the disk. Attribute filters, like octree nodes, consist of value ranges for different attributes. Only if all attribute filter ranges overlap with the attribute ranges of an octree node, the node will be loaded. A node to which an attribute filter is applied does not necessarily contain only the desired points. It usually contains a certain number of points to which the filter does not apply (false positive points). Depending on the application, these points can be additionally linearly filtered after loading the node. There is also the possibility that the false positives do not have to be filtered out, since they are not necessarily a problem in some applications.

A very simplified example of this attribute index structure can be seen in Figure **??**, where only the GPS time and intensity attributes are indexed. This example also uses a two-dimensional quadtree instead of a three-dimensional octree for convenience. In each node, the value ranges for each of the two attributes are noted for the points in the underlying subtree (including the node itself). We now define an AABB query on this point cloud. To do this, we need to specify the following components:

1. **Bounding Box:** In this case, we choose a bounding box that is larger than the point cloud so that the entire point cloud can be returned.
2. **Maximum LOD:** In this example, we choose an arbitrary value greater than two so that all nodes can be considered.
3. **Attribute Filters:** Here we define two filters in the form of attribute-value ranges, to filter for intensities in the range $[230; 255]$ and GPS time in the range $[300; 400]$.

The spatial constraints apply to all nodes in the tree, so we will focus only on the attributes. Notice that the value ranges of the root node overlap with the attribute filters. Therefore, the file associated with the root node must be loaded from disk and all value ranges of the child nodes must be checked. The first two child nodes in LOD 1 do not contain any value ranges that overlap with the attribute filter. Therefore, no searched point can be contained here and we can skip all subtrees marked in red. This is the desired speedup case of the attribute index structure. The value ranges of the remaining two octree nodes in LOD 1 overlap with the attribute filters and therefore have to be loaded from the disk. After that, all loaded nodes can optionally be deserialized and the contained points can be filtered.

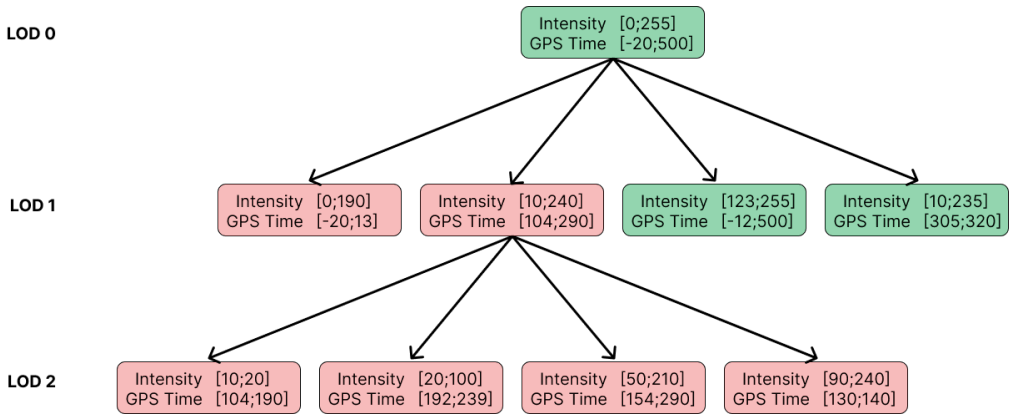
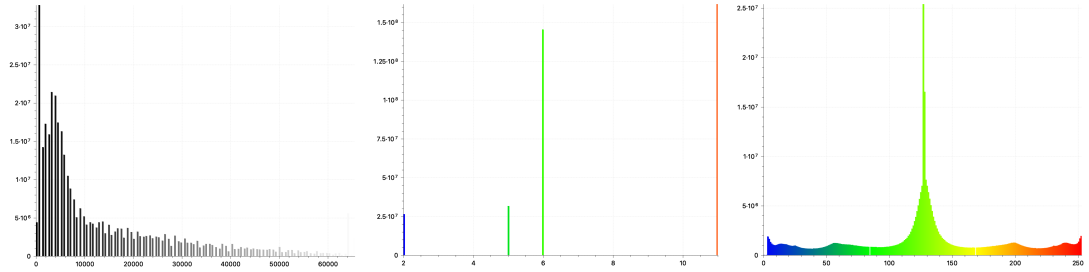


Figure 3.1: Example of Attribute Index Structure

3.3 Histogram Acceleration

The presented approach to attribute indexing is well suited for scalar attributes like intensities (Figure **??**), where value ranges can provide a meaningful boundary. However, there are also non-scalar attributes, such as classification, where arbitrarily distributed identifiers are assigned to different object types. For this use case, indexing by minimum and maximum does not make sense. For example, if the classification values 2 and 11 occur very often, the range of values in octree nodes is mostly $[2; 11]$. If the value 6 is searched for, most of the octree nodes cannot be sorted out, because the searched value overlaps with the value range, although it does not occur. A histogram of the classification values from the Frankfurt dataset in Figure **??** shows these unevenly distributed values. Similar problems occur with scalar values such as normal vectors. Individual axes of normal vectors often have similar orientations (e.g., top, front, and back) due to angular buildings and flat streets in urban point clouds. In the Frankfurt dataset, these common orientations can be seen as peaks in the histogram at values 0, 127, and 255 (Figure **??**). This results in the value range $[0; 255]$ for many nodes, making it difficult to filter for the values around 127.



(a) Intensity Histogram (b) Classification Histogram (c) Normal Vector Histogram

Figure 3.2: Histograms of Frankfurt Dataset Attributes

To solve these problems, I implemented an optimization stage using histograms. Since fixed ranges must be known for histograms, but the ranges are not known in advance for floats, as with GPS time, histogram acceleration can only be implemented for data types with fixed bounds. However, it is possible to use it for the optimization of classification or normal vectors. In addition to the attribute value ranges for all subtrees, histograms for selected attributes are also stored. The attributes to be accelerated as well as the number of histogram bins is individually configurable. For queries, the following checks shown in Algorithm **??** are performed for each node.

Algorithm 1: Querying with Histogram Acceleration

if *Attribute Filter Ranges* $\cap$ *Octree Node Attribute Ranges* $\neq \emptyset$ **then**
  if *Attribute Filter Ranges* $\in$ *Octree Node Histograms* **then**
    Load Node
  else
    Skip Node
  end
else
  Skip Node
end

Additionally, the histograms can provide helpful statistics about the point cloud. For example, the histograms of all root nodes could be summed up to get histograms for the entire point cloud and to see how many points have which classification. Furthermore, histograms offer the possibility to implement point cloud manipulation in the future. If points are deleted, for instance, the point attributes can be subtracted from the corresponding histogram bins to update the attribute index structure.

4 Implementation

The implementation of this thesis is being done open source and available on Github under the name Lidarserv [**lidarserv_2023**]. The repository was created by **dorra_indexing_2022** as part of his thesis [**dorra_indexing_2022**] and now is extended in the *attribute-indexing* branch for this thesis. The software was developed in the programming language *Rust*, which allows for high execution speed and low probability of runtime errors. In the following Chapter, I describe the architecture and the implementation.

4.1 Architecture

Lidarserv has a client-server architecture. The central task of indexing and query processing is performed by the Lidarserv server. Various input and output clients can be connected to the server via network (Figure **??**).

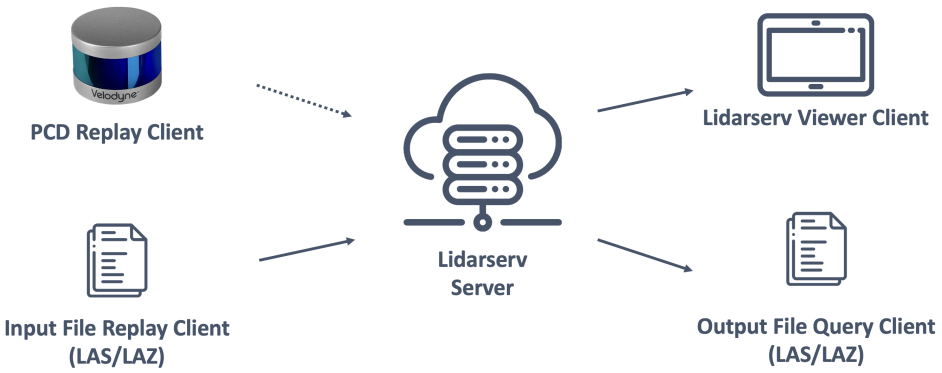


Figure 4.1: Lidarserv Software Architecture

4.1.1 PCD Replay Client

The PCD Replay Client is an input client developed by **dorra_indexing_2022** that provides points in real-time to the server for insertion. It acts as an interface between a real LiDAR scanner and Lidarserv by reading already spatially registered points from the point cloud data (PCD) format [**pcd_format_2023**], converting them, and sending them to the server. Thus, this client provides the real use case of Lidarserv, which unfortunately could not be tested in this thesis due to the lack of a LiDAR scanner. Since the PCD format also supports arbitrary attributes, this client can also be used to efficiently integrate pre-processing pipelines for coloring, normal calculation, or classification.

4.1.2 Input File Replay Client

The Input File Replay Client is a command line tool that allows the emulation of a real LiDAR scanner by replaying point cloud files to the server. For this thesis, I extended it with support for LAS and LAZ files. In a pre-conversion step, the individual points are sorted using the GPS time attribute to represent the real sequence of the scanning process. The sorted points are then divided into frames for network transmission based on their time stamp. The number of frames per second can be specified via the command line interface. The individual frames are then stored as individual LAZ chunks in an LAZ file, from which they can be sent to the server with another command at the defined number of frames per second.

4.1.3 Lidarserv Viewer Client

The Lidarserv Viewer Client is a proof of concept program from the thesis of **dorra_indexing_2022** for displaying point clouds in real-time. It sends real-time view frustum queries to the Lidarserv server and simultaneously receives points to display them. This display of points is especially possible even if the indexing process in the Lidarserv server is still running. This allows applications to provide a real-time preview of the indexing for quality assurance purposes as mentioned in Section **??**. Attribute filtering can be easily added to this application.

4.1.4 Output File Query Client

The Output File Query Tool is a command line tool I developed to output point clouds with AABB queries and attribute filters. The command line can be used to configure the AABB, the maximum LOD, and all types of attribute filters. In addition, it is possible to configure which form of attribute index structure should be used for acceleration. Either no index structure, attribute bounds, or additional histograms can be used. In addition, it is possible to configure whether pointwise linear filtering should be performed on the server side after the nodes are loaded, in addition to the node filtering with the attribute index structures. Query and filter are then sent over the network to the Lidarserv server, which continuously returns updates. As soon as the server has no more new points in the inboxes, a ResultComplete network message is sent and the points are written to a LAS or LAZ file by the output query client. The application is intended to be used mainly for debugging purposes, but it could also be used for applications such as the integration into a geographic information system.

4.2 LAS/LAZ Format

The LAS format (version 1.4) was chosen for the storage of the index files, as well as for the input to the Input File Replay Client and the output to the Output File Query Client [**_specification_2019**]. This is a widely used file format specification for storing LiDAR and point cloud data, where attributes of individual points can also be stored. For each LAS file, a point record format can be selected, which defines which additional attributes are stored per point. The implementation of this thesis supports the point record formats 0-3, which support the attributes shown in Table **??**. In addition, extra bytes can be stored for each point record, allowing any additional attributes to be stored and indexed. However, this is not yet supported by the implementation of this thesis but could be added in the future.

Using the lossless LASZip compression proposed by **isenburg_laszip_2013**, LAS files can be compressed to a size between 7% and 20% of the original size [**isenburg_laszip_2013**]. Since this compression requires a lot of CPU power, it can be enabled or disabled in the Lidarserv implementation.

Item	Type	Size (bits)	Required	Point Record Format
X	signed	32	yes	0-3
Y	signed	32	yes	0-3
Z	signed	32	yes	0-3
Intensity	unsigned	16	no	0-3
Return Number	unsigned	3	yes	0-3
Number of Returns	unsigned	3	yes	0-3
Scan Direction Flag	bool	1	yes	0-3
Edge of Flight Line	bool	1	yes	0-3
Classification	unsigned	5	yes	0-3
Scan Angle Rank	signed	8	yes	0-3
User Data	unsigned	8	no	0-3
Point Source ID	unsigned	8	yes	0-3
GPS Time	float	64	yes	1, 3
Red	unsigned	16	yes	2, 3
Green	unsigned	16	yes	2, 3
Blue	unsigned	16	yes	2, 3

Table 4.1: LAS Point Record Formats 0-3 Overview

4.3 Indexing

This Section describes the implementation of the indexing process. The focus will be on the attribute indexing that was added to the spatial index structure. The general procedure for spatial indexing has been roughly described in Section **??**.

Before the indexing process can start, Lidarserv has to be initialized and configured, for which several settings can be set inside of a configuration file. The settings made here cannot be changed during the runtime of the server and after the initial creation of the index. General settings such as the number of threads, the size of the LRU cache, the size of the octree root nodes, and the number of bogus points can be set here. Specific to LAS data storage are the settings to enable LAZ compression and the selection of the used point record format. Attribute indexing and additional histogram acceleration for indexing can also be enabled or disabled here. In addition, attribute-specific bin counts can be configured for histogram acceleration.

When the server is started for the first time, the index structure is created. The attribute index structure consists of the following nested data structures, listed from the outside in (Figure **??**).

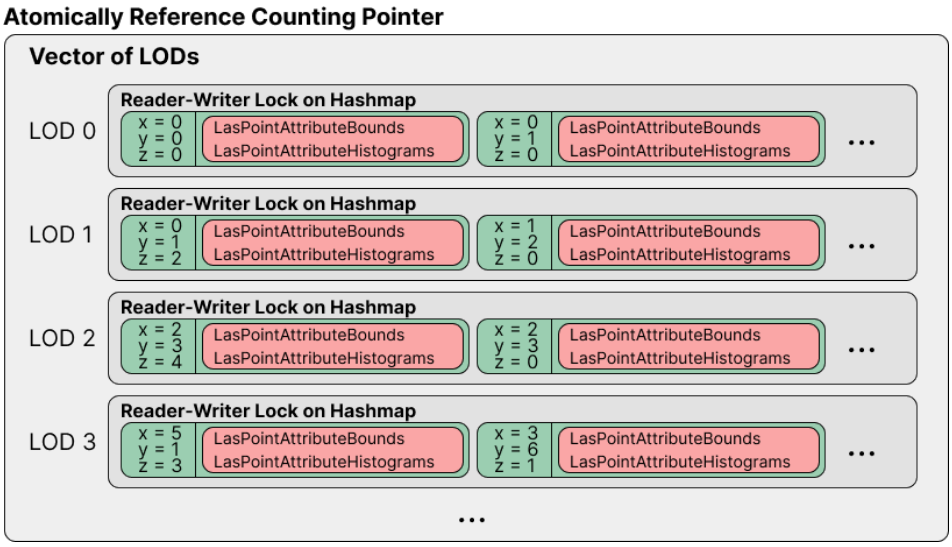


Figure 4.2: Index Data Structure

- **Atomic Reference Counting Pointer:** Allows access to the contained data structure to be shared by multiple threads.
- **Vector of LODs:** Each vector element corresponds to a level of detail.
- **Reader-Writer Lock:** Prevents race conditions between threads by allowing all nodes of the LOD to be temporarily locked for reading or writing.
- **Hashmap with octree node coordinates as keys:** Allows quick access to individual nodes.
- **LasPointAttributeBounds and Option<LasPointAttributeHistograms>:** Stored as values in the hashmap, the struct *LasPointAttributeBounds* contains tuples of minimum and maximum values for each attribute, the struct *LasPointAttributeHistograms* contains histograms of predefined attributes with corresponding primitive data types.

This data structure is stored entirely in memory. The fully indexed Frankfurt dataset requires a relatively small memory footprint of 134MB. When the server is shut down,

the data structure is written to disk in compressed form and loaded back into RAM when the server is restarted. For larger point clouds, depending on hardware limitations, the attribute index structure may take up too much RAM, so it may be necessary to move parts of it to disk. In the future, a grid-based approach could be implemented that reads needed attribute index tiles from the disk and writes unneeded tiles back to the disk. The actual attribute indexing process starts as soon as points are added to the inbox of a node and a worker thread begins processing these points. Before further spatial processing, the attribute index structure gets updated as shown in Figure **??**. If attribute indexing and histogram acceleration are enabled, new temporary *LasPointAttributeBounds* and *LasPointAttributeHistograms* structs are created (**Step 1**). The points from the inbox are iterated over and the two structs are updated with their corresponding update function (**Step 2**). All attributes of a point are passed to the update function of *LasPointAttribute-Bounds*, which then expands all attribute bounds if the attributes are outside the previous bounds. The attributes of a point are also passed to *LasPointAttributeHistograms* and the histograms at the bins corresponding to the attributes are incremented by one value each. In **Step 3**, the bounds and histograms of the node get loaded from the attribute index structure using a read lock. Then, the loaded node bounds can be extended with the temporary bounds, and the temporary histograms can be added to the loaded node histograms (**Step 4**). If histograms or bounds have changed, the bounds and histograms of the node are overwritten with the previously updated ones using a write lock (**Step 5**).

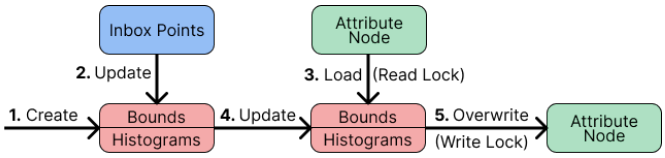
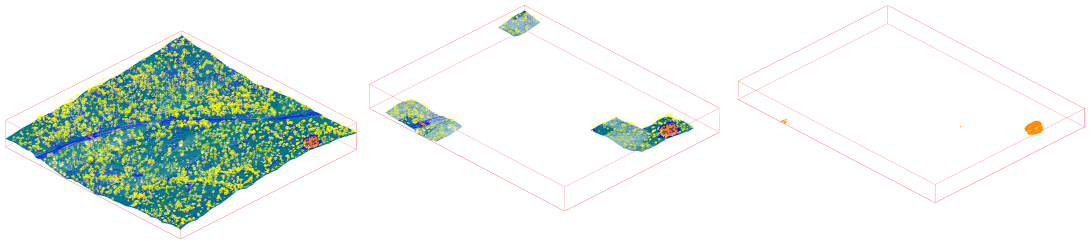


Figure 4.3: Indexing Process

4.4 Querying

The queries are processed by the *OctreeReader*. In this struct, spatial queries and attribute filters can be set independently. It is also possible to define whether the filtering should be done on the point level or the node level. The reader then continuously returns new or changed nodes and corresponding points to which the queries apply, or the information that the queries no longer apply to the sent nodes via the implemented network protocol. To do this, the *OctreeReader* must keep track of the nodes that have already been checked

for the query and filter, and the nodes that have already been sent to the client. A frontier data structure stores all nodes that are currently being validated. All nodes in the tree above the frontier nodes have already been successfully validated and sent to the client; all nodes in the tree below the frontier nodes have not yet been validated. At the start of a new query, the frontier contains all root nodes. If a root node matches the query and filter, the root node is removed from the frontier and all children are added to the frontier and validated. This involves calling a function for each node below the frontier to check if the query applies to the node. This function first checks the spatial query and sorts out nodes to which it does not apply. Next, a function of the attribute index structure is called to check if the node matches the attribute filter. In this case, matching means that all attribute ranges of the filter overlap with the attribute ranges of the node. This means that searched points may be contained. The function first sets a read lock on the corresponding LOD of the attribute index structure. Then the *LasPointAttributeBounds* of the node are checked for overlap with a corresponding function. If there is no overlap, the node can be omitted from the query. If there is an overlap, the *LasPointAttributeHistograms* of the node are also checked. The histograms are used to check whether attribute ranges are not contained in the node, even though they overlap with the boundaries. This can only be the case if an attribute range of the filter is a true subset of the attribute range of the node. If a node does not pass the above checks, depending on the point filtering setting, it is iterated over all points and the attribute filtering is performed again individually at the point level. Finally, the points of the node are sent to the client via the network protocol. A possible optimization for the future is to send the LAZ binaries instead of the raw points to save bandwidth. Theoretically, the client could then do the point filtering. For an overview of the different filter levels, see Figure **??**.



(a) Unfiltered Point Cloud (b) Node Filtered Point Cloud (c) Point Filtered Point Cloud

Figure 4.4: Different Attribute Filtering Methods on an Airborne Dataset From NEON [**subsets_neon_2017**] While Filtering for the Orange Classification Value

5 Evaluation

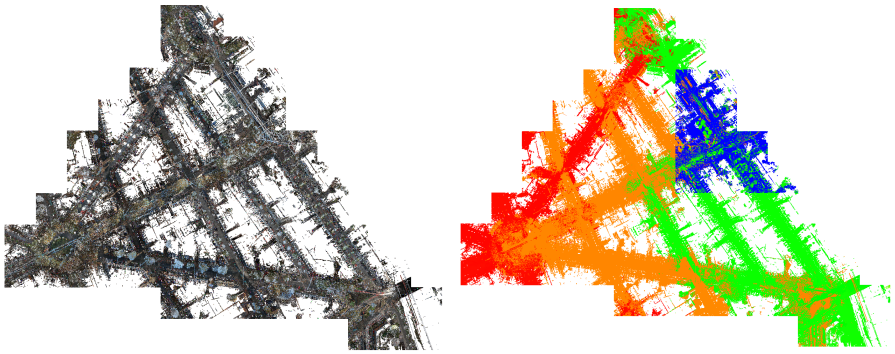
This Chapter presents the used test data, test environment, and test results to examine the implementation against the four goals from Section **??**.

1. **Real-Time Indexing**
2. **Query Time Acceleration**
3. **Query False Positive Point Reduction**
4. **Low Computational Load**

For goal 1 (**Real-Time Indexing**), the focus is on measuring point insertion rates under the influence of various parameters. For goals 2 and 3 (**Query Time Acceleration and False Positive Point Reduction**), sample queries are measured under various settings for duration and point reduction. Goal 4 (**Low Computational Load**) is evaluated by indexing rates and query results with respect to the hardware used. Sections **??** and **??** describe the used test data and environment and the resulting influences on the test results.

5.1 Test data

A terrestrially captured urban LiDAR dataset was required. This dataset needed to include GPS time to simulate real-time rendering in the correct order of capture. In addition, the data set had to contain as many of the attributes listed in Section **??** as possible, to be able to measure them. The Stadtvermessungsamt Frankfurt kindly provided a dataset from Schweizer Strasse in the city of Frankfurt (Figure **??**). The point cloud contains 365 347 416 points, has a size of 12.42GB as a LAS file, and covers about 3km of street. Additional cameras were used to take photos of the roadway, which were projected onto the point cloud for colorization. The attributes intensity, return number,



(a) Frankfurt Dataset (Color) (b) Frankfurt Dataset (GPS Time)

Figure 5.1: Frankfurt Dataset Overview

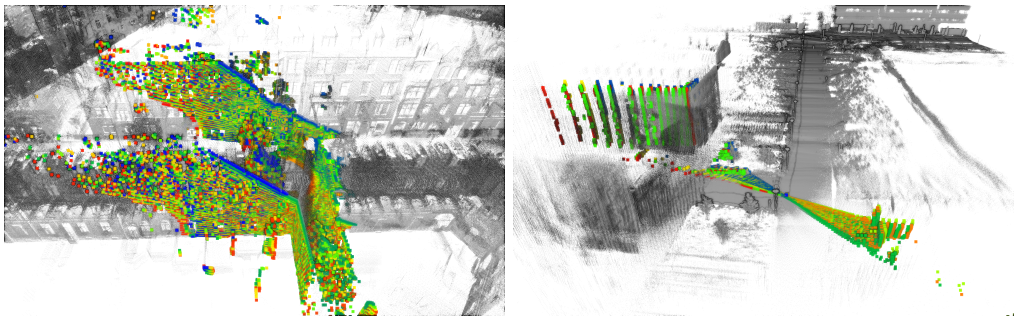
number of returns, and GPS time were also included. Using the Agisoft Metashape software [**metashape_2023**], I also computed classification values for the categories of high vegetation, buildings, road surface, and vehicles, and computed normal vectors of the points. For testing purposes, I stored the x-axis of the normal vectors in the LAS user data attribute. Certain difficulties were encountered when using the dataset, which need to be considered when the results are evaluated:

No Continuous Time: The point cloud contains a spatial Section that was recorded with many drives at different times. Therefore, there is no continuous drive, but there are multiple time jumps, as can be seen in Figure **??**, where the dataset has been colored according to the time of acquisition. This results in a lower indexing speed than could be achieved with continuous recording. However, the query results are barely affected.

Intersections: A total of 13 road intersections are part of the point cloud, which causes the octree nodes of the intersections to be loaded from the disk into the cache each time they are passed from different directions, instead of being able to create the nodes from scratch. This results in reduced indexing speed. However, this effect is always to be expected when recording in dense urban areas.

Scanner Orientation: The data indicates that the scanner is a rotating LiDAR scanner, tilted by about 45° in the direction of travel. This results in a relatively large spatial area within one rotation of the scanner in which points are collected since the scanner captures both the road in front of the car and the house walls and trees far behind the car. Thus, a relatively large number of octree nodes must always be cached to

cover the active scan area (300-400 nodes per second). The Freiburg dataset used by **dorra_indexing_2022** utilized a scanner position where the scanner covered only a very small area to the left and right of the car per rotation, which means that significantly fewer octree nodes need to be cached at the same time (less than 100 nodes per second). Figures **??** and **??**, where 100 000 consecutive points are colored, show this difference. In Freiburg, the colored area is much more compact, while in Frankfurt the area with low point density is very widely spread. This results in lower indexing speed and higher cache requirements for the Frankfurt dataset. Unfortunately, the Freiburg dataset could not be used because no attributes were recorded.



(a) Slice of Frankfurt Dataset (b) Slice of Freiburg Dataset

Figure 5.2: Comparison of Frankfurt and Freiburg Dataset

5.2 Test Setup

The implementation has been tested on a Macbook Pro 14” 2021 with an Apple M1 Max processor and 64GB of RAM as well as an internal SSD. During testing, all other applications, as well as some potentially disruptive operating system features (Apple Spotlight background indexing, Time Machine, software updates, cloud services), and internet access were disabled to avoid interfering with the measurements. However, the macOS operating system has a very poor I/O performance in some cases, which has a strong negative impact on the measurement results. In the future, Linux should be preferred for further testing. A custom application is used for the evaluation, which behaves differently from the Lidarserv applications in some respects. It does not use a client-server structure but contains the point insertion as well as the indexing and querying in one application.

Therefore, it should be noted that network bandwidth is not considered in the tests. The test point cloud to be indexed resides entirely in memory in the evaluation application to avoid memory bottlenecks when reading the points, which would not be the case in the real application due to the real-time input from the scanner. To test the maximum possible indexing speed, the points are not inserted in real-time, but as fast as possible. Different parameters can be set for each test run. The following set of parameters is used for all test runs unless otherwise specified. All measurements were performed with the Frankfurt dataset.

- **Number of threads:** This parameter determines how many worker threads process points from inboxes. According to the hardware-supported threads of the test Macbook, 10 threads are used.
- **Cache size:** This parameter specifies the maximum number of octree nodes stored in the last-recently used cache. Larger cache sizes can reduce disk accesses, but increase memory requirements. For testing, the cache size is chosen to be large enough to completely cache the number of nodes used in a short period of time. This cached area moves with the scanner, nodes entering the area in the direction of travel must be created or loaded from the disk, and nodes leaving the area behind the vehicle must be written to the disk. The number of octree nodes in the cache is therefore set to 10 000, which required about 1.2 GB of memory with the test data used.
- **Bogus points:** The maximum number of bogus points that can be stored in nodes can be defined here. Bogus points increase the indexing speed enormously, however, this also means that not all points are indexed completely and correctly. The effect of bogus points on indexing speed and query performance is examined in more detail in Section **??**, but by default they are disabled.
- **Histogram Bins:** The number of histogram bins is chosen as follows, to select a good minimum number of bins for scalar attributes and to have one bin for each single value for non-scalar attributes.
  - Intensity, Scan Angle Rank, User Data, Color: 25 bins
  - Return Number and Number of Returns: 8 bins
  - Classification and Point Source ID: 256 bins
- **Node Size:** This parameter sets the size of the root node of the octree. Here a size of approximately $32m \times 32m \times 32m$ was chosen, this parameter will be examined in more detail in Section **??**.

- **Compression:** This parameter configures whether the octree nodes are saved to disk as LAS or as a compressed LAZ file. Compression reduces disk space requirements of the point cloud at the expense of higher CPU utilization. Since disk accesses are the bottleneck and CPU utilization is relatively low, compression offers a performance gain for the indexing process, so it is enabled here by default.

5.3 Results

This Section is divided into measurements of indexing speed (Section **??**) and measurements of queries and filters (Section **??**). In addition, measurements of the effects of node size and bogus points on indexing and querying are presented in Section **??**. The measurement results are then discussed and analyzed in Section **??**.

5.3.1 Indexing Measurements

Indexing speed is essential for achieving goal 1 (**Real-Time Indexing**). To get an overview and to make sure that attribute indexing does not significantly reduce speed, comparative measurements were taken without attribute indexing, with simple attribute indexing, and with additional histogram acceleration. In addition, these three measurements were taken with compression enabled and disabled. The results are shown in Figure **??**. Indexing with attribute indexing and compression enabled took 7311 seconds (∼122 minutes). To determine what portion of the execution time was taken up by attribute indexing, the indexing was analyzed using the Tracy performance tracing tool [**tracy_2023**]. This showed that the execution has two phases: In the first phase, at the start of indexing, only new octree nodes are created which can be stored directly in the cache until it is full. In this phase, attribute indexing took about 5.1% of the indexing time. The first phase lasts only a short time until the cache is full and nodes have to be flushed to disk. Then the second memory-bound phase begins, where 97.8% of the measurement time was spent writing the already compressed LAZ binaries back to disk. In this second phase, attribute indexing took only 0.4% of the indexing time.

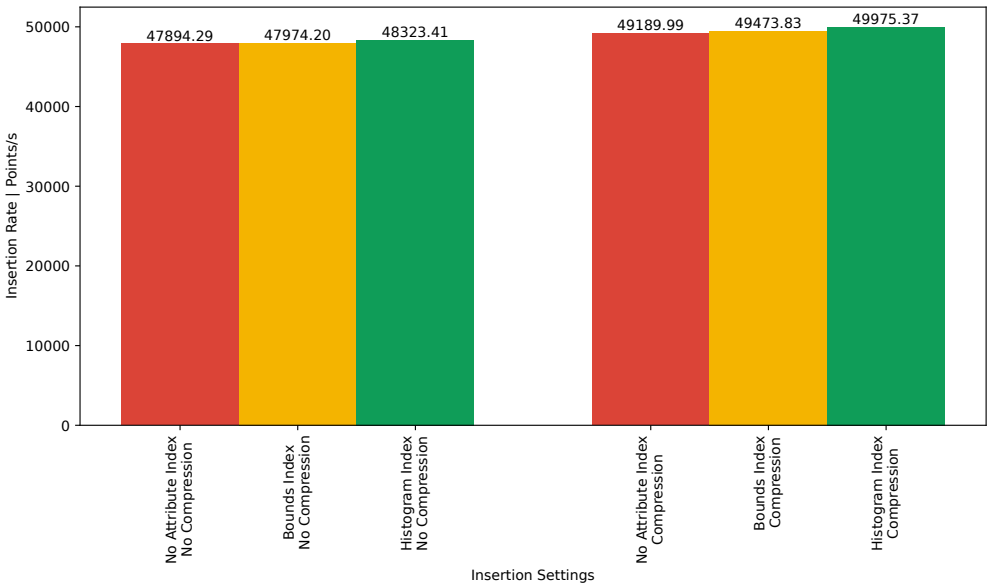
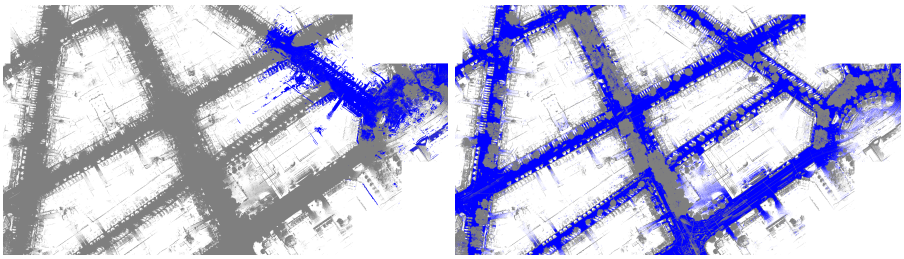


Figure 5.3: Insertion Speed Comparison

5.3.2 Query and Filter Measurements

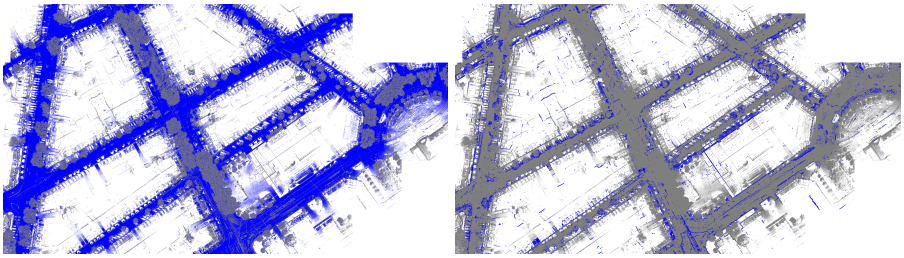
To measure query and filter performance, sample spatial queries and attribute filters are required. Since the focus is on attributes, an AABB query is defined as a spatial query that includes the entire point cloud and queries the highest possible LOD. Thus, no node is excluded by the spatial query, and the performance of the attribute filters can be tested on all nodes.

For the attribute filters, six individual filters were defined, each of which filters only a single attribute according to an attribute range. In addition, one attribute filter has been defined, which filters according to ranges of multiple attributes. The attributes Time, Classification, Normal Vectors, Intensity, Color, and Number of Returns were tested. All filters are visualized in Figure **??**, the resulting points are colored blue.



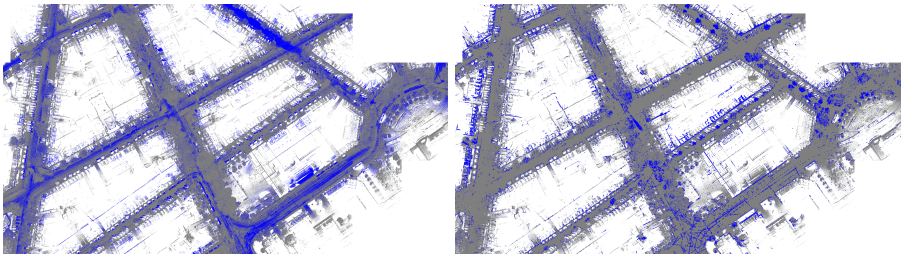
(a) Time - Small Range

(b) Classification - Ground



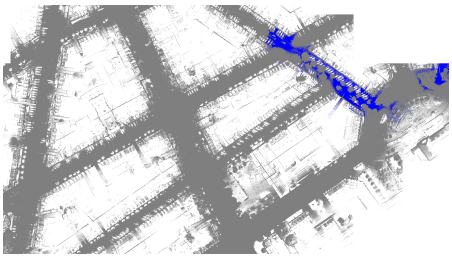
(c) Normal - Vertical

(d) Intensity - High Value



(e) Intensity - Low Value

(f) Color - High Red Value



(g) Mixed - Ground and Time Range

Figure 5.4: Defined Filters

The results of the measurements are broken down into four graphs: one for the number of points, one for the number of nodes, one for false positive rates, and one for the execution times. The first three graphs are relevant for goal 3 (**Query False Positive Point Reduction**), the first graph also provides a good overview of the selectivity of each filter.

Figure **??** lists the total number of points, the number of points after attribute acceleration with attribute bounds, the number of points after additional histogram acceleration, and the number of points after pointwise filtering for each filter. The number of filtered points differs from the number of filtered nodes because not every node contains the same number of points. However, since each node takes a certain minimum time to load, the number of nodes to load has a significant effect on the execution time of the filters.

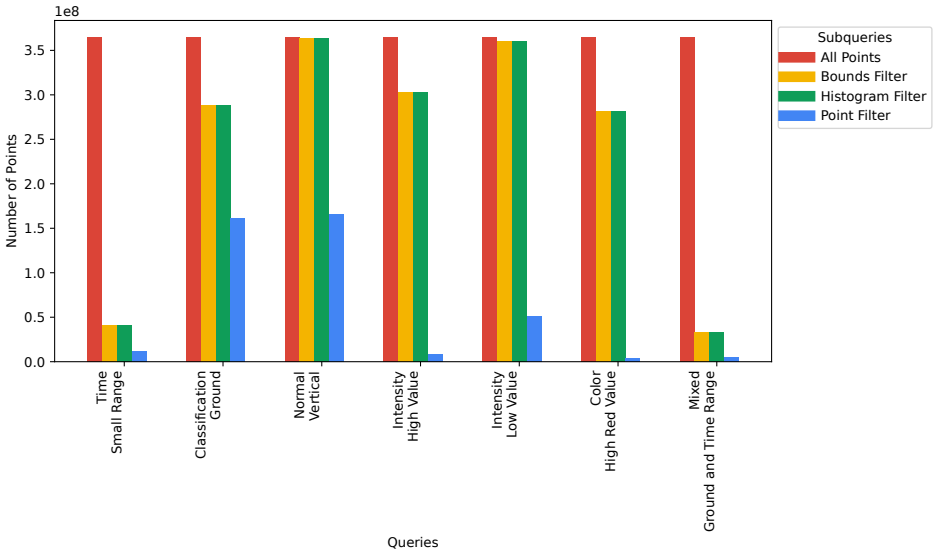


Figure 5.5: Query Comparison by Number of Points

Figure **??** lists the filters by number of nodes. For each filter, the total number of nodes, the number of nodes after the attribute bounds filter, and the number of nodes after additional histogram acceleration are listed. In both graphs, the point reduction is the difference between the bar for all points and the bars for attribute bounds filtering and histogram acceleration, respectively. Therefore, a larger difference corresponds to the desired point reduction. The difference between bounds filtering and histogram acceleration corresponds to the additional optimization provided by histogram acceleration.

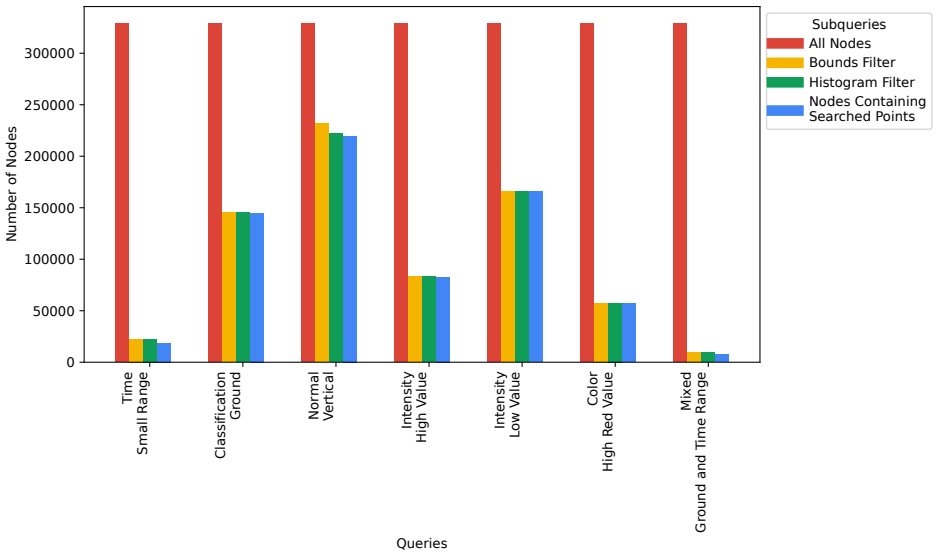


Figure 5.6: Query Comparison by Number of Nodes

Figure **??** lists the percentage of false positive nodes and points for each filter. False positives are defined as all points not matched by the filter and all nodes containing only points not matched by the filter.

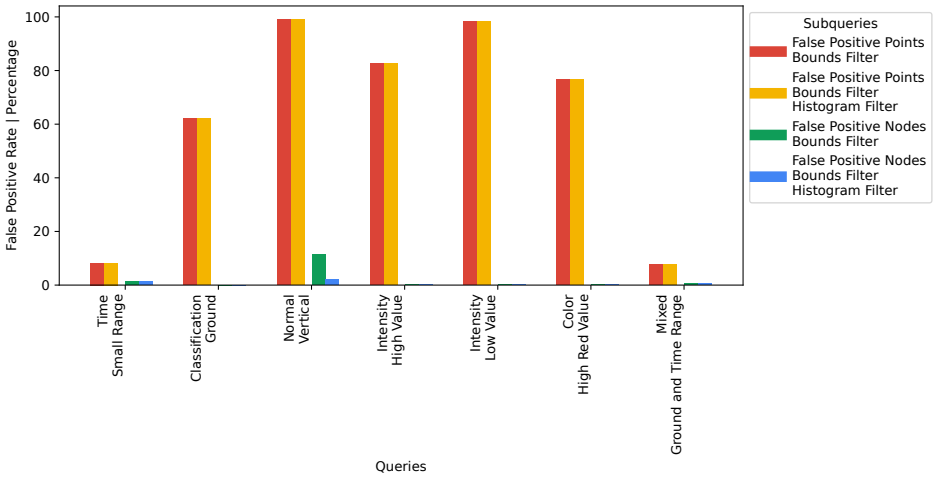


Figure 5.7: False Positive Rates

Figure **??** lists the execution times of each filter, which are important for goal 2 (**Query Time Acceleration**). For each query, the total time is given for the spatial-only query and for point filtering without any acceleration structure. Additionally, the times for point filtering with attribute bounds filter and with additional histogram acceleration are listed. The loading times for the nodes without point filtering are not listed, because due to the current implementation all LAS or LAZ binaries are always unpacked and all points are loaded. Therefore, these times do not differ significantly from the times with additional point filtering. It should also be noted that the times for the spatial query alone are slightly different despite identical queries, which is due to interfering processes and operating system scheduling.

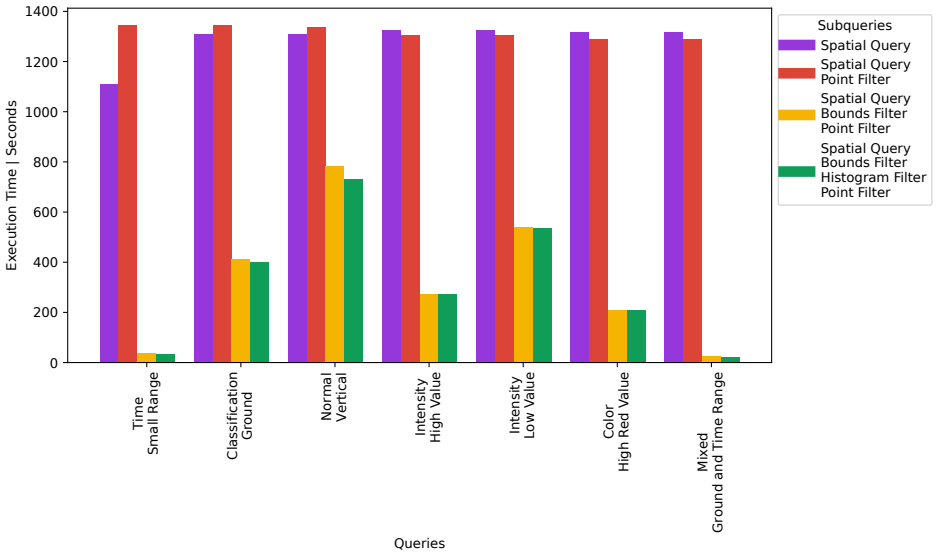


Figure 5.8: Query Comparison by Time

5.3.3 Parameter Optimization

Some of the program parameters have a significant impact on the results of indexing or querying. Especially the parameters *Node Size* and *Bogus Points* are to be emphasized. Both parameters affect the relevant metrics indexing speed, average query time, and average point reduction. The average point reduction indicates the percentage of non-searched points that could be sorted out by the attribute indexing. For both measurements, only a small subset of the Frankfurt dataset was used, which contains 12.4 million points. The values measured here should differ only slightly from the measurements with the entire point cloud. Measurements were taken with histogram acceleration enabled and a cache size of 5000 octree nodes. It is important to note that the point reduction and query times are averaged over the seven specific filters and are not generalizable.

For the node size, measurements were taken with eight different sizes. The results can be found in Figure **??**.

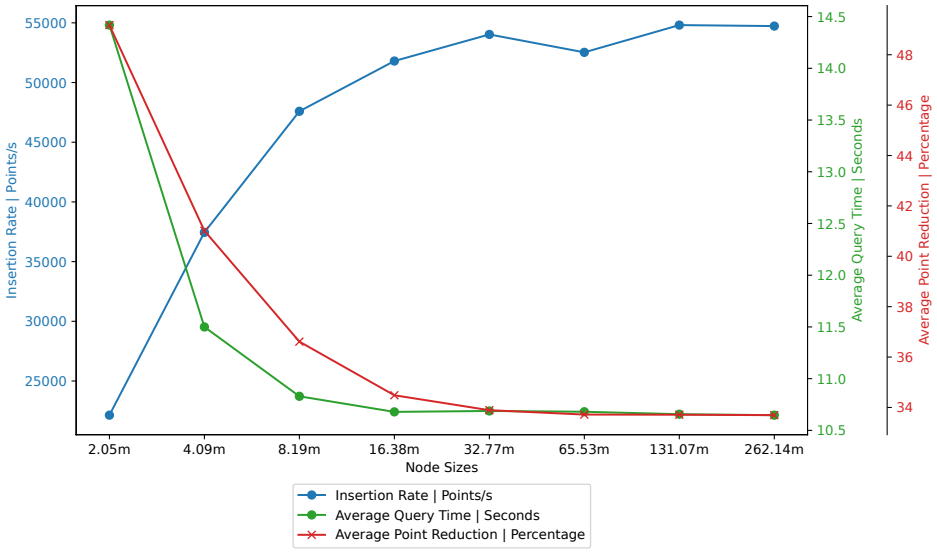


Figure 5.9: Performance Comparison for Node Sizes (Default Size: 32.77m)

The results of the different maximum numbers of bogus points can be found in Figure **??**. Here, an additional line at 400 000 points per second is shown for the indexing speed, which represents the intended minimum goal of this thesis.

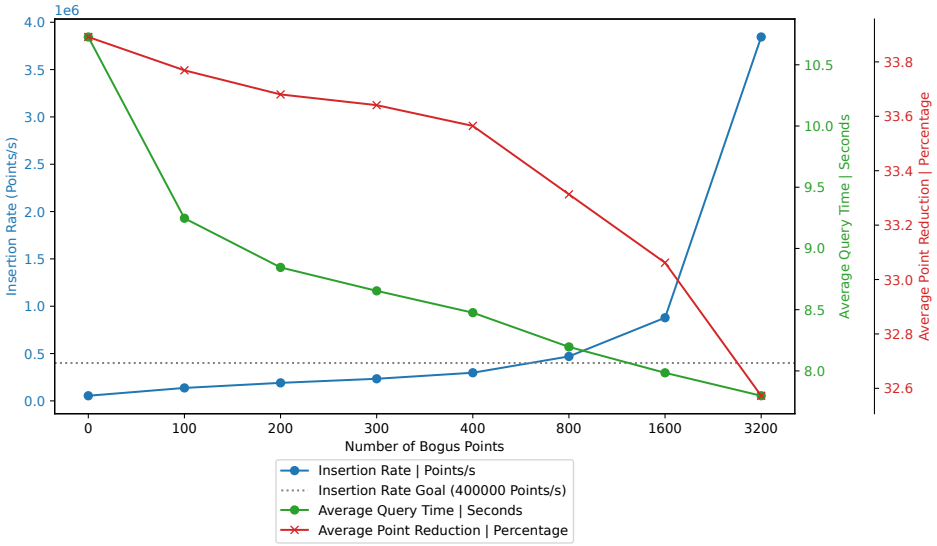


Figure 5.10: Performance Comparison for Bogus Points (Default Value: 0 Bogus Points)

5.4 Discussion

This Section discusses the test results and examines them in relation to the objectives set.

5.4.1 Indexing Measurements

According to the measurement results, the indexing performance is mainly memory-bound. Therefore, it is also recommended to enable compression, if the available computing power allows it, in order to counteract the memory bottleneck. Unexpectedly, the indexing graph **??** shows that the indexing speed increases with the attribute indexing enabled and even reaches the highest speed with histogram acceleration. Using performance tracing, it is visible that some, but relatively small, time spans are required for attribute indexing, and additional computation time is required for histogram acceleration. However, since the worker thread must actively wait for the operating system while writing the binaries, it is reasonable to assume that the additional time for attribute indexing has a positive effect on the scheduling. In general, it can be said that attribute indexing does not affect indexing speed negatively in this test setup. Nevertheless, the achieved indexing speed of 49 975 points per second is relatively low. At this speed, real-time indexing of the Frankfurt dataset with Lidarserv is impossible: During the replay of the dataset with the Input File Replay Client **??** an increasing number of unindexed points accumulates on the Lidarserv server, and after sending the last points considerable amount of time is needed to clean up the accumulated points. However, since the dataset has the problems mentioned in Section **??**, such as time jumps, many intersections, and a poor scanner orientation, a higher indexing speed can be expected in reality, since here no time jumps occur and a more suitable scanner position can be chosen.

5.4.2 Query and Filter Measurements

The query diagrams show how many unwanted points and nodes can be sorted out by the attribute index so that they do not have to be loaded from memory. The number of unsearched points/nodes that can be removed is referred to as the point or node reduction. For many filters, the node reduction is proportionally higher than the point reduction. This is because these filters are most likely to remove small nodes that only contain a few points. Most of these small nodes are leaf nodes with high LODs that are not completely filled. Nodes with few points have a higher probability of being sorted out because the

probability of containing a searched point is usually lower with fewer points. In addition, the spatial distribution of the attribute affects whether many or few points can be sorted out. Attributes with high spatial locality, such as GPS time, usually contain only similar values in individual nodes, since points that are close to each other were usually recorded at similar times. Therefore, the false positive rate of points in this case is very low, since there are few unsearched points in the selected nodes. On the other hand, there are attributes such as intensity, color, or normals, which contain very different values in a small space. Again, almost all unneeded nodes can be removed, partly with histogram acceleration. However, there are still a lot of unwanted points in the remaining nodes in comparison to the GPS time. Classification can remove about 40% of the unsearched points in this example. Since the classification ID of the road is the highest classification value in the data, all unneeded nodes can already be sorted out without using histogram acceleration.

In the false positive rates diagram it can be seen, that after node and histogram filtering, a maximum of only one percent of nodes that do not contain a search point will still be loaded for all queries. This means that almost the maximum possible node filtering proportion has already been achieved. However, the loaded nodes still contain a high proportion of non-searched points for most filters, which results in rather high false positive rates for the points. To further speed up attribute filtering at the point level within nodes, additional attribute indexing within individual nodes could be considered in the future. For example, multidimensional SFCs could be used to index both the position and selected attributes within nodes.

In the timing diagram, it can be seen that despite the partially high false positive rates of the points, a significant time improvement can generally be achieved by sorting out the nodes. This is due to the fact that each time a node is loaded, in addition to the variable number of points, a certain amount of time is spent reading a LAS header of constant size.

5.4.3 Parameter Optimization

In Section **??** on indexing, it was noted that the indexing speed is too low for real-time indexing of the Frankfurt dataset. In the querying Section **??**, it was recognized that there is a rather high false positive rate of the points after node-wise filtering. Depending on the application, either faster indexing or higher point reduction may be more important. The parameters of node size (Figure **??**) and bogus points (Figure **??**) provide possibilities for regulation. For both parameters, there is a trade-off between indexing speed and point reduction.

Node Size The node size increases indexing speed at larger root sizes because there are fewer small nodes to write to in the first LODs, each of which causes a large time overhead for indexing and querying due to the increased disk accesses. Above a certain node size (in this case 32 meters of side length) the indexing speed does not increase anymore, because then the first LODs contain only very few nodes and points and the actual distribution takes place on higher LODs. However, smaller nodes also ensure that the average point reduction is significantly higher since smaller nodes are more likely to be sorted out by the attribute index, as described in Section **??**. In addition, the influence of the node size on the LOD structure and the appearance of the real-time visualization with view frustum queries must not be ignored. If the node size is too small, even the lowest level of detail will have a very high resolution, which can have a negative impact on the real-time visualization due to longer loading times.

Bogus Points The bogus points work similarly to the node size. A higher number of bogus points ensures that new nodes are not created until a certain minimum number of points is reached. This avoids creating nodes with fewer points, which improves both indexing speed and query time. However, since the bogus points are not distributed in small nodes, they are also less likely to be sorted out by node filtering. As the number of bogus points increases, the average point reduction also decreases. However, it only decreases by 1.2% between 0 and 3 200 bogus points, while at the same time the indexing speed increases from about 54 229 points per second to 3 844 395 points per second, an improvement by a factor of 70. Therefore, bogus points are an excellent acceleration method for attribute indexing. However, visual effects must also be considered, as visual irregularities in the LOD grid structure are caused by keeping these points in nodes. Therefore, the number of bogus points should always be as high as necessary for real-time indexing, but as low as possible to avoid visual artifacts. In this example, a number of 800 bogus points would be appropriate, as it just exceeds the required minimum indexing speed.

6 Conclusion

In this thesis, an approach for attribute indexing of point clouds was developed using an octree-based method. For this purpose, the spatial index structure by **dorra_indexing_2022**, consisting of multiple MNOs, was extended to store the attribute ranges of all contained points for each subtree. For queries, attribute filters were implemented in addition to spatial queries. With the help of the attribute index structure, it was then possible to sort out entire subtrees that did not contain the desired attribute value ranges of the attribute filters. In order to better identify which subtrees can be sorted out, especially for non-scalar attributes, an additional histogram acceleration was implemented. This was done by storing histograms for each subtree for selected attributes, which can be used to check even more precisely which attributes are contained. The entire attribute index structure is small enough to reside in memory and can therefore accelerate queries very quickly. In addition, the Input File Replay Client has been extended to support LAS and LAZ input files, allowing points to be sent to the index server based on their timestamps to simulate a real scanner. For querying indexed data, the Output File Query Client has been implemented, which can be used to send spatial queries and attribute filters to the index server via the command line, the results are then saved as LAS or LAZ files.

Four goals were defined for the data structure, which can now be assessed here in terms of their completion, following the discussion of the evaluation results in Section **??**.

The goal of **real-time indexing** was defined with a lower limit of 400 000 points per second indexing speed. With the default settings described in Section **??**, the goal could not be met; only just under 50 000 points per second could be achieved. However, with the parameter optimizations presented in Section **??**, the indexing speed can be increased to well over 400 000 points per second, while the point reduction and optical quality are only slightly worse. Thus, the goal of real-time indexing is achieved.

The goal of **query time acceleration** requires that the attribute index achieves shorter query times than pure linear filtering of all points. In Section **??**, the time results were

discussed, the query times with attribute indexing were only between 1% and 50% of the time for linear filtering of all points, depending on the filter. This goal was clearly achieved.

The goal of **query false positive point reduction** was to ensure that a significant portion of the points that are not needed can be sorted out or do not need to be loaded. It turned out that almost all unnecessary nodes could be sorted out, but only 33% of the points could be sorted out on average. In Section **??**, methods were presented to further increase the point reduction at the expense of indexing speed. For further point reduction, the approach of n-dimensional SFC indexing of points within nodes is proposed for future work. Thus, the goal can be considered partially achieved.

The **low computational load** goal had as a prerequisite that the required indexing and querying goals could be achieved with simple and energy-efficient hardware. The testbed described in Section **??** clearly fulfills this requirement, and the CPU utilization rarely exceeded 50% during testing. This is also because the performance of both indexing and querying is largely limited by disk access operations. In general, however, the goals can be achieved with a fast hard disk and consumer hardware.

The evaluation of the data structure shows that the four defined goals were all well achieved. In addition, the bottlenecks of the data structure as well as possible optimization approaches in terms of indexing speed and point reduction were identified. These are presented in the following section on future work.

With the results of this thesis, the attributes of a point cloud can be indexed in real-time during point cloud acquisition, in addition to spatial indexing. The attribute filters, which are accelerated with the index structure, can be used for an attribute-filtered real-time preview of the data, but the implemented interfaces also enable the usage of further processing algorithms and the possibility for users to use the index structure for queries. Extensive node filtering ensures that nodes that do not match the attribute filter are not transmitted to the querying client, saving bandwidth.

6.1 Future Work

The data structure itself provides several directions for future work. During the indexing phase, CPU usage is quite low. This is because most of the time is spent waiting for the individual octree nodes to be written. In the future, asynchronous memory operations or separate memory management that avoids a multi-file approach could be used to achieve

significantly higher indexing speeds. In particular, because the overhead of LAS headers can be avoided, separate memory management could bring significant performance benefits. In addition, the influence of the operating system and the associated I/O drivers should also be taken into account in the future, as significantly better results can be expected with Linux.

For queries, this thesis only presented an approach to filter out whole octree nodes based on the attributes they contain. However, this can easily be extended with additional indexing approaches that also index the points within the nodes. For example, n-dimensional space-filling curves (as described in Section **??**) could be used to index and query the coordinates and attributes of points within nodes more efficiently. However, this is likely to have a negative impact on indexing speed.

A promising approach for real-time visualization is to use the histogram data from histogram acceleration to determine which octree nodes contain the most points searched by attribute filters and send them to the viewer client first. This helps to reduce the latency of getting most of the points to be displayed. Nodes with a very small number of searched points can then be loaded in a time-consuming manner once most of the searched points have been transferred from the large nodes.

Beyond the presented data structure, the comparison of the measured data in Section **??** showed that the orientation of the LiDAR scanner has an impact on the indexing speed. Investigating the influence of scanner position and which scanner position is best suited for real-time indexing would be interesting in future work.

Finally, another direction for future work would be the identification or implementation of more real-time pipeline components for the completion of a full LiDAR processing pipeline. Particularly important here are processing elements to compute the processed data attributes described in Section **??**, such as classification or normals, which must already be available for indexing and therefore must also be computed in real-time.

6.2 Acknowledgments

Frankfurt Pointcloud Dataset provided by © Stadtvermessungsamt Frankfurt am Main, Stand 07.11.2022 und 03.12.2021.

Freiburg Pointcloud Dataset provided by Fraunhofer Institute for Physical Measurement Techniques IPM, department Object and Shape Detection.

Glossary

This document is incomplete. The external file associated with the glossary 'main' (which should be called `main.gls`) hasn't been created.

Check the contents of the file `main.glo`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

If you don't want this glossary, add `nomain` to your package option list when you load `glossaries-extra.sty`. For example:

```
\usepackage[nomain]{glossaries-extra}
```

Try one of the following:

- Add `automake` to your package option list when you load `glossaries-extra.sty`. For example:

  ```
  \usepackage[automake]{glossaries-extra}
  ```

- Run the external (Lua) application:

  ```
  makeglossaries-lite.lua "main"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "main"
  ```

Then rerun LaTeX on this document.

This message will be removed once the problem has been fixed.