EG UK Computer Graphics & Visual Computing (2024)
A. Slingsby and D. Hunter (Editors)

Min-Max Modifiable Nested Octrees (M^3NO): Indexing Point Clouds with Arbitrary Attributes in Real Time

Paul Hermann[1,2], Michel Krämer[1,2], Tobias Dorra[1,2], and Arjan Kuijper[1,2]

[1]Fraunhofer-Institute for Computer Graphics Research IGD, Darmstadt, Germany
[2]Technical University of Darmstadt, Germany

Abstract

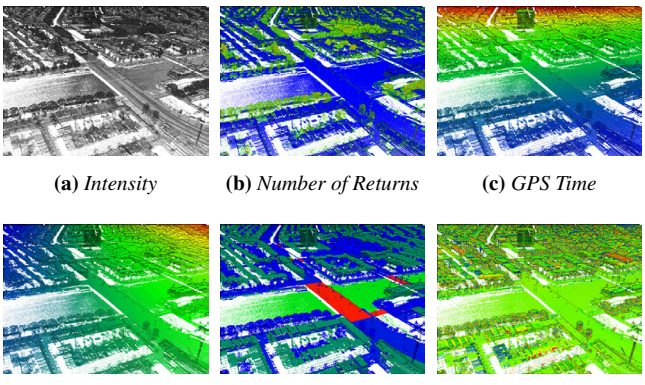
We present a data structure that allows 3D point clouds with arbitrary attributes to be indexed in real time. We focus on large datsets from mobile mapping systems such as airborne and terrestrial laser scanners. Compared to traditional indexing approaches running offline, our data structure can be created incrementally while the points are being recorded. This allows the data to be used (i.e. analyzed or visualized) already during acquisition or immediately after it has finished. The data structure enables queries based on spatial extent and value ranges of arbitrary attributes. This is in contrast to existing works, which focus on either spatial or attribute indexing, typically are not real-time capable, or only support a limited set of attributes. Our approach combines Modifiable Nested Octrees and extended Binned Min-Max Octrees. Using a subset of the well known AHN4 dataset with 138 million points, we evaluate the approach, assess quality and query performance, and compare it with an existing state-of-the-art solution. On commodity hardware, our data structure can process 1.97 million points per second, which is more than most commercially available laser scanners can record. When filtering points by attribute value ranges, it also reduces the number of octree nodes that have to be loaded, and it substantially outperforms naive sequential point filtering.

1. Introduction

Geospatial point clouds are becoming increasingly important in many areas. For example, architects and urban planners use them in building construction projects [VBB22] or for urban infrastructure planning [PSW20]. Other applications include the rollout of fibre optic lines [KBW*24], district heating planning [LDME*22], or flood modeling [LHL*21]. On a larger scale, point clouds are used to generate digital elevation models [PB07] and for the mapping and regular monitoring of forests [AMLS22] or railway lines [TAL22].

Point clouds are captured using terrestrial or airborne LiDAR (Light Detection And Ranging) technology [WYTT21, LLW*21]. LiDAR scanners are becoming more and more precise and record millions of points per second. These points contain not only spatial coordinates but a variety of attributes such as color, GPS time, and the intensity of the reflected laser beam. More attributes such as the 3D normal for each point or its semantic classification can be derived from the recorded information and are often also included in the final result. The LAS specification [Ame19], a widely used point cloud file format, defines a range of possible attributes. Figure 1 shows screenshots of a few selected ones.

LiDAR-recorded geospatial point clouds are usually very large, both in terms of data volume and area. It is not uncommon that datasets reach sizes of many terabytes. Some of them even cover whole countries, e.g. the AHN4 dataset of the Netherlands [AHN20]. Analyzing and visualizing such large datasets requires sophisticated acceleration structures. While spatial indexes enable fast access to specific areas of the point cloud in different levels of detail, attribute indexes allow the points to be quickly filtered based on given criteria or value ranges. A combined index, for example, would be able to return all points that lie within a given spatial extent (or bounding box), have been recorded in a certain period of time, and belong to a specified class (e.g. *Tree* or *Building*).



(a) *Intensity* **(b)** *Number of Returns* **(c)** *GPS Time*

(d) *Scan Angle Rank* **(e)** *Classification* **(f)** *Normal (X-axis)*

Figure 1: *A selection of point attributes in the AHN4 dataset [AHN20], colored by value*

DOI: 10.2312/cgvc.20241235

The process of creating such an index is called *indexing*. The current common practice is to index point clouds offline, i.e. some time after they have been recorded with tools such as *Schwarzwald*, *PotreeConverter* or *Entwine* [BK20, SOW20, Hob24]. This out-of-core process often takes several hours and doing it later unnecessarily extends the time it takes from recording the data until it can actually be utilized.

Many applications, however, could benefit from a more immediate access. For example, if regular processes such as railway monitoring for predictive maintenance had access to the data right after it has been acquired, they could react much quicker to imminent faults. The same applies to emergency cases (e.g. earthquakes or floods) where immediate availability of analysis results is key.

Going one step further, analyzing and visualizing a point cloud already while it is being recorded would enable real-time quality assurance. Errors in the recording or missing spots could immediately be displayed to the operator (e.g. on a tablet computer) who could then directly fix them. This would avoid having to repeat the data acquisition process later, which is time-consuming, cost-intensive and, again, extends the time until the data can be put into use.

Processing LiDAR point cloud data in real time is, however, rather challenging as modern laser scanners record millions of points per second. A real-time indexing algorithm has to be able to process points at least as fast as the laser scanner produces them. In addition, querying the index during acquisition (e.g. for live quality assurance), requires it to be responsive and to return results quickly.

In this paper, we present an approach to point cloud indexing that can be done in real time, i.e. during the data acquisition and as fast as the points are produced. Compared to previous work, our approach not only considers the location of the points in $\mathbb{R}^3$ but also all their attributes. It allows the point cloud to be spatially queried based on a bounding box or a view frustum, and to be filtered by ranges of attribute values. At the same time, the index contains a level-of-detail structure and can therefore be used for interactive visualization. Spatial and attribute filters determine, which parts of the point cloud should be returned at the desired level of detail.

The main contribution of the paper is the *Min-Max Modifiable Nested Octree (M^3NO)* data structure. It is based on the real-time capable Modifiable Nested Octree (MNO) [BDSF22] and combines it with the idea of the Binned Min-Max Quadtree [ZY10], which we extend to work in real time.

We evaluate our data structure with a real-world dataset and based on the following research goals:

G1 – Real-time indexing Our indexing algorithm should be able to insert points into the M^3NO structure at least as fast as they are being recorded by a typical commercially available laser scanner.

G2 – Query Data Reduction The amount of data that needs to be loaded into memory during querying relative to the size of the result set is a quality metric for every index. Our goal is to eliminate as many octree nodes as possible early in the query process to reduce the number of points that need to be sequentially filtered later.

G3 – Query Time Reduction Compared to naive sequential filtering without an index, our data structure should substantially decrease the query execution time. Furthermore, the query times should be comparable or better than those of existing approaches.

For reproducibility, our implementation is available under an open-source license on GitHub [Lid24].

The remainder of the paper is structured as follows. We first summarize related work in the area of point cloud indexing, compare it to our approach, and describe the research gap we bridge (Section 2). We then describe the M^3NO data structure in detail and describe how indexing and querying work (Section 3). After this, we evaluate our approach based on the research goals defined above (Section 4). The paper finishes with a conclusion and an outlook on future work (Section 5).

2. Related Work

Research in the area of point cloud indexing can be divided into two sub-areas: data structures for spatial indexing (Section 2.1) and approaches that allow arbitrary point cloud attributes to be indexed or that combine spatial and attribute indexing (Section 2.2). In this section, we summarize existing approaches and discuss their suitability for real-time indexing.

2.1. Spatial Indexing

A well-known data structure for spatial indexing is the k-d-tree [Ben75]. Each node in such a tree represents a plane that subdivides a k-dimensional space along one axis. For each tree level $l \in \mathbb{N}_{\geq 0}$, a different axis $i \in [0,k)$ is selected, such that $i = l \bmod k$. If the position of the partition plane is chosen based on the distribution of the points in space, the tree becomes well-balanced. This requires all points to be known and sorted along the corresponding axis. Adding points later either results in an unevenly distributed tree or causes expensive rebalancing. This renders the k-d-tree unsuitable for real-time indexing where points might arrive at any time.

The same applies to the R-tree, a data structure similar to a one-dimensional B-tree but supporting multiple dimensions [Gut84]. In an R-tree, each node represents a bounding rectangle that encloses a group of points or other bounding rectangles. Since the number and area of the bounding rectangles depend on the points in the tree and inserting new points requires rebalancing, real-time indexing cannot be implemented efficiently.

An efficient data structure for $\mathbb{R}^3$ is the octree. It divides the space independently of the points to be indexed and centrally along all three axes into eight sub-spaces [Mea82, FB74], which are then further divided recursively. The octree does not require rebalancing, but it still can be unbalanced if the data is unevenly distributed. Building an octree requires the bounding box of all points to be known in advance, which makes it unsuitable for real-time applications.

Another way to index a spatially distributed set of points in linear time is to use a space-filling curve such as a Z-order curve [Mor66] or a Hilbert curve [Hil91]. As the name implies, space-filling curves traverse the entire n-dimensional space and assign an ascending index to each point (or cell) visited. This maps the n-dimensional space to one dimension, which in turn allows points to be inserted and queried through binary search in $\mathcal{O}(\log n)$ time. Most space-filling curves preserve locality to a certain degree, so that points that

are close to each other in space are also close to each other on the one-dimensional curve. They have been used for point cloud indexing [GVOC18], distributed processing [LBA16], as well as to build continuous levels of detail [LVOMV20]. Since curve indexes can typically be calculated in constant time and independently of each other, space-filling curves enable efficient and massively parallel point cloud indexing [KB21].

As space-filling curves and octrees are closely related, they share the same requirement that the bounding box of all points needs to be known in advance, which is impractical in real-time scenarios. Furthermore, as described in Section 2.2, space-filling curves are not able to index an arbitrary number of attributes.

The Modifiable Nested Octree (MNO) is a special type of octree, where each node can contain a set of points stored in a regular three-dimensional grid [Sch16]. This allows the space to be divided both spatially and into levels of detail. The points 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.

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 [KB21] and adapted for MNOs by Bormann et al. [BDSF22]: In a large regular grid, an arbitrary number of MNO trees are created. As soon as points fall into a new cell of the grid, a new MNO tree is created in that cell. The approach presented in this paper is based on this idea. We describe it in detail in Section 3.

2.2. Attribute Indexing

A straightforward way to index a point cloud by attribute is to store it in a database and reuse the existing indexing capabilities. Dobos et al., for example, present a concept for a database model, in which each row represents a single point [DCSG*14]. The coordinates and attributes are all stored in different columns. A primary database index is then created on the coordinates and further indexes can be created on the attributes. The drawback is that each index structure must be evaluated separately for a query, and then the intersection of the results must be formed, which involves considerable overhead.

Storing individual points as records in a database is not efficient given the enormous number of points and the large number of attributes. The PostgreSQL extension pgPointCloud provides a solution by grouping nearby points into patches [RBBL21]. Each patch stores the minimum, maximum, and average value for each coordinate and attribute. As Bormann et al. have shown, the performance of pgPointcloud in terms of indexing and querying is not competitive and in many cases even slower than working directly on the raw data [BKW22], which makes it unsuitable for real-time indexing. However, saving points in chunks instead of saving each point individually is a useful method to keep the size of the index small and to simplify compression [CPP17].

Another way of indexing attributes is to use space-filling curves. Similar to spatial indexing (Section 2.1), they can map the multidimensional attribute space to one dimension. For example, HistSFC presented by Liu et al. [LVOMV20] supports indexing of both spatial coordinates and attributes. However, Liu et al. discovered that space-filling curves do not scale well when applied to an arbitrary number of attributes. In their paper, they show that in tests on the AHN2 dataset, indexing with four dimensions already results in a false positive rate of 164%, which increases rapidly with additional dimensions. This means that only a small number of unwanted points can be discarded in the querying process. To deal with this problem, some approaches automatically rank which attributes in a point cloud are most important to index and should, therefore, be included in the index structure [NDAK20]. Since our aim in this paper is to index arbitrary attributes (and not just a limited number), space-filling curves are not an option for attribute indexing.

An example of an attribute index structure that is closely related to a spatial index is presented by Ladra et al. [LRLPSC22]. They extend the Binned Min-Max Quadtree data structure [ZY10] and apply it to LiDAR point clouds. The points are sorted into an octree, and for each subtree, they are sorted by attribute. Since the algorithm has access to the whole dataset, the minimum and maximum values of all attribute values within each subtree are known. This allows entire subtrees to be skipped during querying if the value ranges do not match the query condition. However, since all points must be known in advance, the data structure is not real-time capable. Nevertheless, the basic idea of combining a spatial index (i.e. an octree) with an attribute index and storing attribute value ranges for subtrees fits well with our goals and provides a basis for the data structure presented in this paper.

2.3. Research Gap

It can be seen that there are a variety of approaches to spatial indexing, but very few of them support real-time indexing and levels of detail. The approaches for attribute indexing are either not real-time capable or only provide support for a limited number of attributes.

A data structure that works in real time and supports both spatial indexing and indexing of an arbitrary number of attributes of any type does not exist yet. In this paper, we create such a novel data structure by extending and combining Modifiable Nested Octrees for spatial indexing with the ideas behind Binned Min-Max Octrees and pgPointCloud for attribute indexing. The resulting data structure is called *Min-Max Modifiable Nested Octree (M^3NO)*. The following section describes it in detail.

3. Min-Max Modifiable Nested Octree (M^3NO)

The Min-Max Modifiable Nested Octree consists of a spatial indexing structure and an attribute indexing structure. In this section, we describe both components and give details on the insertion process. Finally, we explain the general querying process.

3.1. Spatial Data Structure

The spatial data structure is based on the work of Bormann et al. [BDSF22]. It consists of an arbitrary number of MNO trees in a regular grid with a fixed cell size. In a real-time scenario, the total size of the point cloud is not known in advance, as new points are recorded continuously. Due to this, the indexing algorithm creates new root nodes each time points arrive in grid cells where no MNO

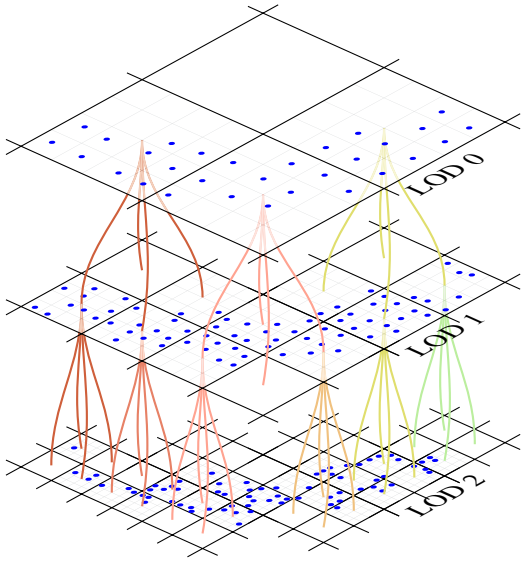


Figure 2: *Modifiable Nested Octree from Bormann et al. [BDSF22]: In the upper region, three root nodes are generated, corresponding to the lowest level of detail (LOD). In the higher LODs, many smaller nodes contain more points in total and thus provide a higher level of detail. For illustration purposes, each node in the figure has a grid of* 4×4 *cells in which the points are stored.*

tree has yet been created. This allows the point cloud to arbitrarily expand during runtime.

Each MNO consists of octree nodes, which recursively divide the space into eight subspaces. This halves the size of the nodes along each axis with every additional level in the tree. All nodes contain a regular grid of $128 \times 128 \times 128$ cells to store the points. A maximum of one point can be stored in each cell of this grid. The size of 128^3 for the regular grid was used by Schütz for the original data structure and also gave us the best results [Sch16]. In most cases, the regular grids are only filled to a very small extent with our test data.

By halving the node size at each level along all axes, the distances between the regular grid cells are also halved. This results in higher densities of points deeper in the tree, and thus a higher level of detail. Figure 2 illustrates this for a two-dimensional point cloud.

The data structure saves all nodes with the corresponding points as individual files, either in the compressed LAZ format [Ise13] or the uncompressed LAS format [Ame19]. Using this out-of-core approach allows point clouds exceeding the size of main memory to be indexed. When nodes need to be read or modified, they are loaded into a cache with a definable maximum capacity. If it is exceeded, the least recently used nodes are written back to disk.

3.2. Indexing Process

The indexer receives incoming points in real time to sort them into the point grid cells of the octree nodes and to update the attribute index structure. For this purpose, each node has an additional in-memory point buffer that collects points for the associated subtree. This is called the inbox of a node. For each incoming point, the indexer looks for an MNO tree in the regular grid of MNO trees. If there is one, it places the point in the inbox of the root node of the tree. If no tree exists at that location, a new one is created.

Multiple worker threads insert the points from the inboxes of the nodes into the corresponding point grid cells in parallel. A priority function selects which inboxes are processed first based on the age of the task and the number of items to be inserted. Each thread then performs the following steps:

1. The thread takes all points from a non-empty inbox of a node.
2. It creates the corresponding node if it does not already exist. Otherwise, unless the node is already in the cache, it loads it from disk.
3. The points from the inbox are inserted into the point grid of the node. To achieve a visually more even point distribution, this is done using a grid center sampling approach, in which the point closest to the center of each cell is selected from all points in the inbox [Sch16, SOW20].
4. If a cell is already occupied, the point is passed to the inbox of the corresponding child node, where the process is repeated. Each point is stored only once.

A more detailed description of the insertion process is given in the work of Bormann et al. [BDSF22].

3.3. Attribute Data Structure

The attribute data structure is based on the concepts of Ladra et. al. [LRLPSC22], which we adapted for real-time capability. The structure stores additional information about the point attributes contained in each subtree. Similar to the approach of pgPointCloud [RBBL21] the attribute value ranges are stored for this purpose (Section 3.3.1). The construction of this data structure is integrated with spatial indexing.

3.3.1. Value Range Index

Unlike proposed in the work of Ladra et al., where all points had to be known in advance, in a real-time scenario the minimum and maximum values of each attribute cannot be pre-computed. Instead, in our data structure, we incrementally update the value ranges for each subtree whenever a new point is added to it, which enables real-time capability.

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. This information is then used to expand and update the attribute value ranges of the subtree. Note that storing the attribute value ranges only for subtrees requires less memory than indexing them point-wise. This allows the attribute data structure to be kept permanently in memory due to its small size, resulting in higher insertion rates and query execution times due to faster access times.

A simplified example of this attribute index structure can be seen in Figure 3, where only the *GPS time* and *intensity* attributes are indexed. For legibility, this example uses a two-dimensional quadtree

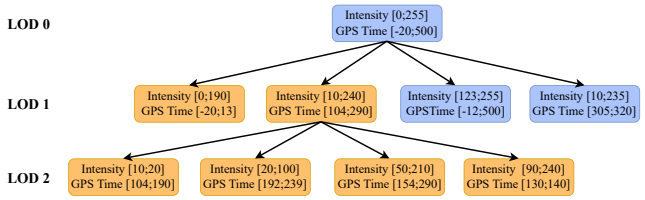


Figure 3: *Example of attribute index structure while filtering for intensities in the range [230; 255] and a GPS time in the range [300; 400]. Orange nodes are discarded by the attribute index structure, blue nodes are loaded.*

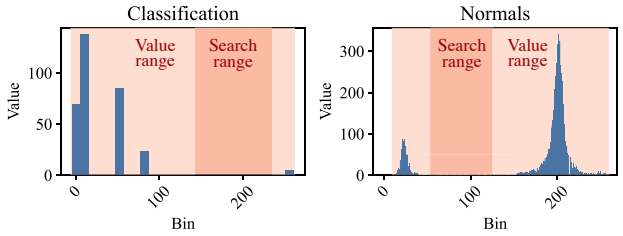


Figure 4: *Histograms of subtree attributes: Although the search range overlaps with the value range of the subtree, no attribute value is contained in the search range.*

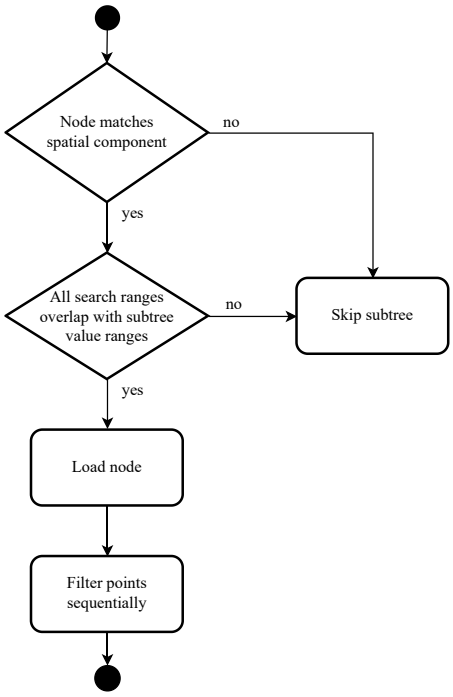


Figure 5: *Filtering stages when querying a node*

instead of a three-dimensional octree. 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). Nodes with intensities in the range [230; 255] and a GPS time in the range [300; 400] are searched. The value ranges of the root node overlap with the attribute filters. Therefore, the file associated with the root node has to be loaded from disk and all value ranges of the child nodes have to be checked. The first two child nodes in LOD 1 (orange) do not contain any value ranges that overlap with the attribute filter. Therefore, no searched point can be contained here, and we can skip both subtrees entirely. The value ranges of the remaining two octree nodes in LOD 1 (blue) overlap with the attribute filters and therefore have to be loaded from disk. After that, all loaded nodes are deserialized and the contained points are filtered sequentially.

It is still possible that a node does not contain any points that match the attribute filter, even if the value ranges overlap. This is illustrated in Figure 4. This case is called a false positive node and has a negative impact on querying performance.

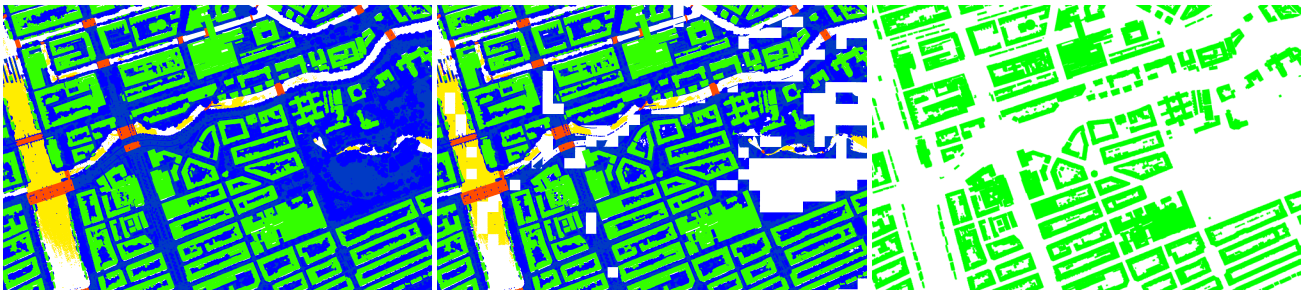
3.4. Querying

Queries are defined by a spatial component and an attribute filter. The spatial component is defined by the spatial extent (e.g. using a bounding box) and the maximum level of detail desired. The attribute filter consists of value ranges for attributes. Only the points which fulfill the spatial component and whose attribute values lie in all search ranges should be returned. To execute the query, the algorithm traverses all MNO trees. It then runs through a series of filtering stages as shown in Figure 5. It checks for each node beginning at the root whether it satisfies the spatial component, i.e. whether it is located in the searched bounding box and the level of the node is less than or equal to the maximum searched level. If the spatial component does not apply to the node, the entire subtree can be skipped. If the spatial component is evaluated as positive, the algorithm checks whether all attribute value ranges of the subtree overlap with the corresponding search ranges. Otherwise, the subtree can be discarded. If an overlap is detected, the file corresponding to the node gets loaded and filtered sequentially by spatial component and attribute. After this, the algorithm continues with the children.

The resulting point clouds of the filtering stages are shown in Figure 6. Here, Figure 6a shows a portion of the AHN4 dataset [AHN20] colored by classification value. Filtering is based on the building classification value (shown in green). In Figure 6b, only the range filter has been applied. Some subtrees have already been sorted out, which can be seen as rectangular white spots. However, there are some red and yellow areas, especially on the left side, which do not contain any building classification and were returned anyway. These are false positive results of the range filter. The points in the returned nodes can then be filtered sequentially according to the building classification, leaving only the green building points. This sequential filtering could have been done without the range filter. However, it would have taken much longer, because all existing nodes would have been loaded.

4. Evaluation

For evaluation purposes, we implemented the data structure in our open-source software called Lidarserv [Lid24] and measured various properties using exemplary test data. The same measurements were also carried out with pgPointCloud for comparison. The test environment and dataset are described in Section 4.1. The results are presented in Section 4.2.



(a) *Raw, unfiltered point cloud* **(b)** *Point cloud after range filter* **(c)** *Final result after sequential point filtering*

Figure 6: *The AHN4 pointcloud is colored by classification. The figure shows the results of the individual attribute filtering stages while querying for buildings (green). Note that the range filter already removes a range of nodes, which then no longer have to be filtered sequentially.*

4.1. Test Setup

Lidarserv provides a server that can be used to index and query points in real time with the data structure explained in Section 3. The software package includes an evaluation program that indexes a point cloud as fast as possible to determine the maximum indexing speed, which was used for the following measurements. All measurements were executed on a commodity Linux system with an Intel® Core™ i7-8700 processor, 32 GB of RAM, and SSD storage with a write speed of 365 MB/s and a read speed of 530 MB/s.

A subset of the AHN4 dataset containing 138 million points was used as a test dataset. AHN4 covers the entire Netherlands and was recorded with state-of-the-art airborne laserscanning technology [AHN20]. The selected section is a consecutive flight path with continuous time to simulate real-time acquisition. It was recorded with an average of 1.92 million points per second. Prior to the evaluation, we configured the node and cache size of lidarserv to achieve the best performance for the AHN4 dataset. We set the root node size to 32.77 m, limited the least recently used cache size to 10 000 octree nodes, set the number of threads to 32, and disabled compression for the indexed files. For the pgPointCloud measurements, we chose a PDAL pipeline and a PDAL chipper with a capacity of 400 to divide the points into patches for the insertion process [PDA24].

4.2. Results

We took measurements for each of the three goals of the paper. The goal of real-time capability (G1) was checked by measuring the maximum possible indexing speed (see Section 4.2.1). The data reduction goal (G2) was measured by running several sample queries and recording the number of nodes and points returned (Section 4.2.2). The time reduction goal (G3) was evaluated by measuring the execution time of sample queries (Section 4.2.3).

4.2.1. Real-Time Indexing

In order to reach the goal of real-time indexing, our data structure had to be capable of processing more points per second than received from the LiDAR sensor. The speed of indexing was evaluated with both compression enabled and disabled. The measured indexing speeds are listed in Figure 7.

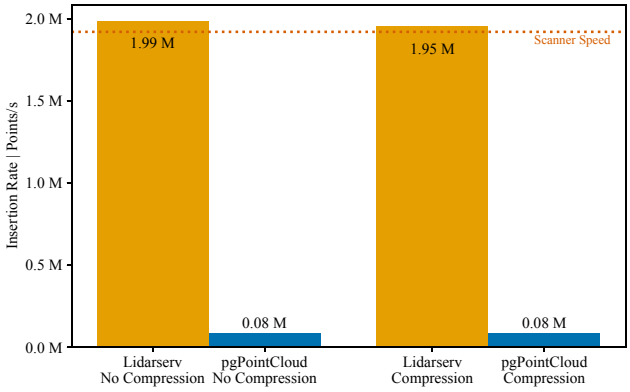
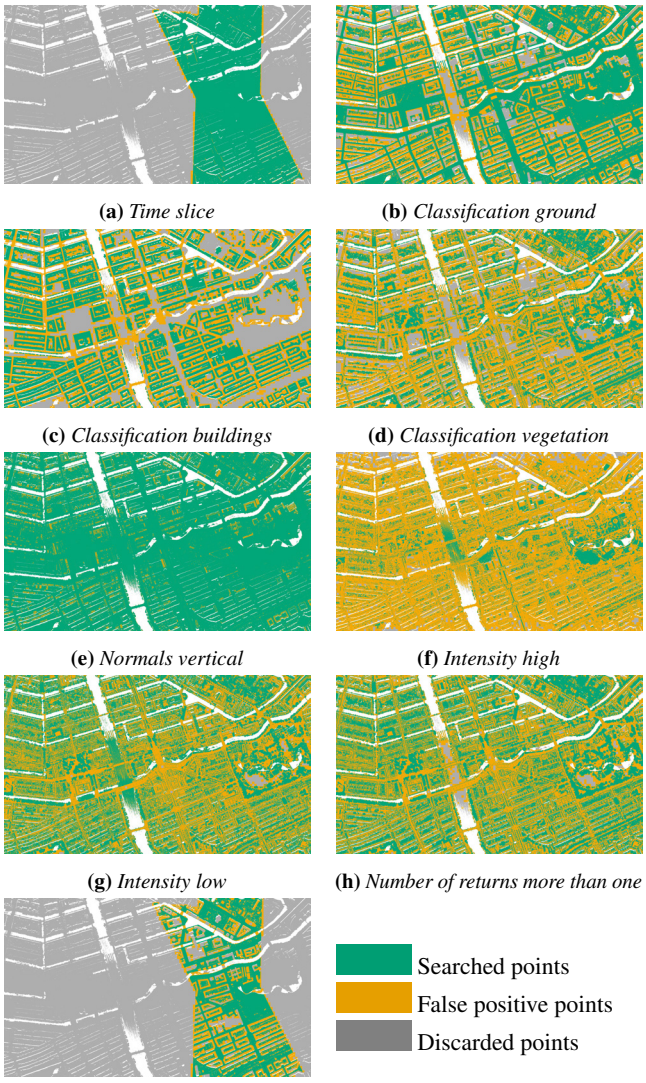


Figure 7: *Insertion speed comparison: Lidarserv indexes the points considerably faster than pgPointCloud. Furthermore, the system is capable of exceeding the threshold required for real-time indexing of the AHN4 dataset.*

The measurements show, that with just our commodity hardware, we can already surpass the point rates of many commercially available airborne LiDAR scanners, such as the RIEGL VQ-1560i-DW with 1.33 million measurements per second on the ground [RIE22]. Our test dataset was recorded with an average rate of 1.92 million points per second, which we just about exceeded with our fastest insertion rate measurement of 1.99 million points per second. It is noteworthy, that the overhead induced by the attribute indexing on top of the spatial index is fairly small. In comparison to pgPointCloud, our indexing speed is 25 times faster, which is a significant improvement.

4.2.2. Query Data Reduction

The second goal addresses the quality of the index in its ability to reduce the amount of data that has to be loaded. This is measured in the number of nodes and points that cannot be eliminated by the index despite not matching the query. For this, we defined a set of sample queries (see Figure 8). To better show the differences between the attribute filters, the spatial component of the queries was chosen to cover the entire point cloud with all LODs.



(a) *Time slice* **(b)** *Classification ground* **(c)** *Classification buildings* **(d)** *Classification vegetation* **(e)** *Normals vertical* **(f)** *Intensity high* **(g)** *Intensity low* **(h)** *Number of returns more than one* **(i)** *Time slice & ground classification*

Figure 8: *Sample queries: the searched points are displayed in green. The false positive points returned from lidarserv are marked in yellow. Discarded points are shown in grey.*

Figure 9 provides an overview of how many points are searched for each query and how many were returned by Lidarserv and pgPointCloud. This allows the efficiency of the attribute index structures to be assessed. Fewer points returned by both range filters indicate a higher efficiency and a faster query. The various queries have very different selectivity, which is why the results also differ between the queries.

Both approaches allow for removing a high number of points before sequential filtering. A greater range of points can be filtered out when filtering for attributes, where points spatially close to each other have similar values (e.g. time or classification). pgPointCloud achieves better results due to a smaller average patch size.

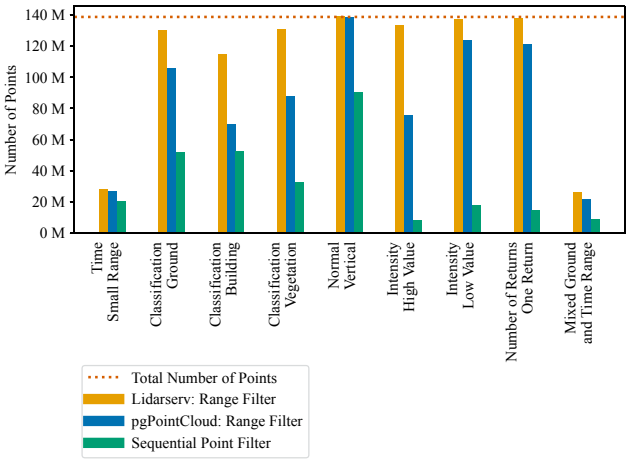


Figure 9: *Query comparison by number of points between lidarserv and pgPointCloud*

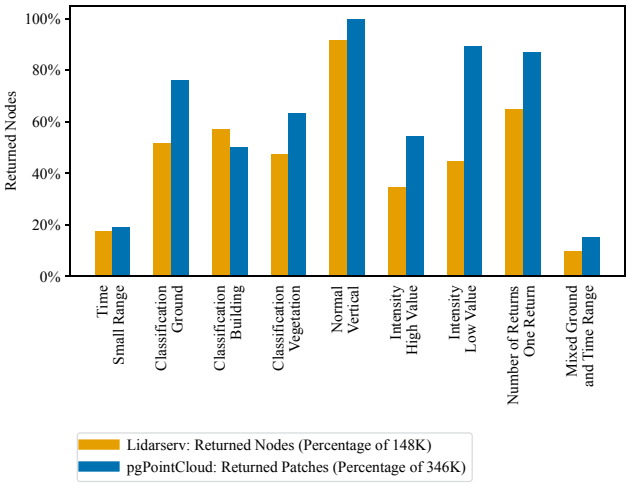


Figure 10: *Query comparison by number of nodes between lidarserv and pgPointCloud*

Figure 10 compares the percentage of nodes/patches returned from the attribute index. Here you can see that Lidarserv can filter out more nodes on average than pgPointCloud. This is due to the fact that Lidarserv has a tree structure with nodes of different sizes and therefore contains many small nodes with few points, which have a higher probability to be filtered out.

Note that even if the point reduction is not optimal, the good node reduction helps lidarserv decrease the query time due to the relatively high overhead for loading each node. This is especially true for cases where the index allows many nodes with only few points to be eliminated.

4.2.3. Query Time Reduction

The index structure should decrease query execution times for end users or connected applications, enabling interactivity and short

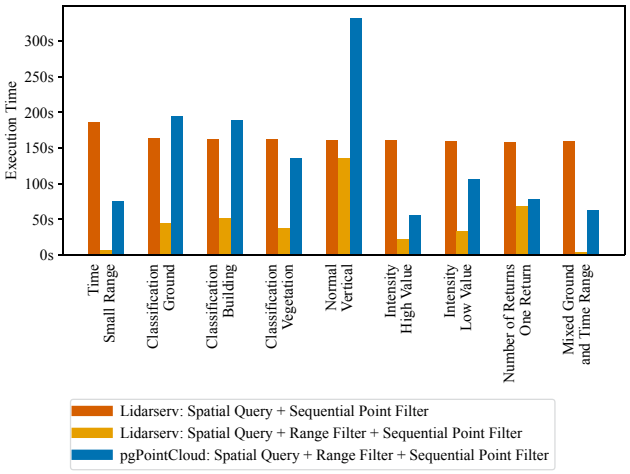


Figure 11: *Query time comparison between lidarserv and pgPointCloud*

waiting times. In Figure 11, we compared the query times of Lidarserv and pgPointCloud. The results demonstrate that, in all test queries, the range filter significantly reduces query time in comparison to naive sequential point filtering. Lidarserv even demonstrated superior querying speeds, with an average of 1.22 Mpts/s, compared to pgPointCloud, which achieved only 0.22 Mpts/s.

5. Conclusion

In this paper, we presented the M^{3}NO data structure, which provides means to index and query geospatial 3D point cloud data acquired by LiDAR devices in real time, i.e. while they are being recorded. The data structure consists of both a spatial index and an attribute index, which enables point cloud filtering based on spatial component and attribute value range. Compared to existing approaches, M^{3}NO works in real time and with an arbitrary number of attributes of any type. It outperforms the state-of-the-art solution pgPointCloud in terms of indexing and querying speed and even has additional LOD support. We have evaluated the data structure with regard to the achievable indexing performance (G1), the number of octree nodes that need to be loaded into memory during querying (G2), and the query time (G3). In summary, we managed to meet all three of our research goals.

In comparison to naive sequential point filtering, we can skip entire octree nodes that do not contain any matching points, while naive filtering always requires all points to be loaded. M^{3}NO especially benefits from selective queries and from queries over attributes that offer a high spatial locality.

Our point cloud index combines attribute and spatial queries with the support for multiple levels of detail. While the first is useful for many analysis tasks, the latter is a requirement for efficient point cloud visualization. To the best of our knowledge, HistSFC by Liu et al. is the only other approach that does that [LVOMV20]. However, their approach is based on space-filling curves that do not scale well with the number of attributes, which Liu et al. already recognize. Our data structure is not limited in this regard.

In addition, M^{3}NO is capable of real-time indexing. We have evaluated this using a subset of the real-world AHN4 dataset with 138 million points. AHN4 has been recorded with an average of 1.92 million points per second. We were able to achieve a higher indexing speed of 1.99 million points per second already on commodity hardware. Using a more modern CPU and a faster SSD drive would lead to even better results.

We were also able to show that our data structure is fast enough to support queries in real time. Bormann et al. have already demonstrated this for the spatial index [BDSF22], but M^{3}NO can now also index arbitrary attributes. Combined with the level-of-detail support, this enables applications such as live quality assurance, e.g. through visualization on a tablet computer while recording. This allows errors and missing spots to be fixed already during the acquisition process and avoids time-consuming and costly repeated acquisitions.

In any case, indexing points with M^{3}NO while recording allows the point cloud to be used right after the data acquisition process has finished. Traditional downstream batch indexing, which is usually started some time later and which may take many hours, can be entirely skipped. This is useful in any situation where timely access to data is of importance, for example in emergency cases.

Due to the strong data dependency, it would be interesting to test the approach with other airborne and terrestrial datasets in the future. All known indexing approaches usually require parameters that are adapted to the dataset and the application. Here, either the parameter selection could be automated, e.g. based on point density, or a data-independent approach could be developed.

References

[AHN20] Ahn4 landsdekkende dataset 2020. https://www.ahn.nl/ahn-4, 2020. Accessed: 2024-01-24. 1, 5, 6

[Ame19] LAS Specification 1.4—R14. https://www.asprs.org/wp-content/uploads/2019/03/LAS_1_4_r14.pdf, 2019. Accessed: 2023-03-29. 1, 4

[AMLS22] ALVITES C., MARCHETTI M., LASSERRE B., SANTOPUOLI G.: Lidar as a tool for assessing timber assortments: A systematic literature review. *Remote Sensing 14*, 18 (2022). doi:10.3390/rs14184466. 1

[BDSF22] BORMANN P., DORRA T., STAHL B., FELLNER D. W.: Real-time Indexing of Point Cloud Data During LiDAR Capture. In *40th Computer Graphics & Visual Computing Conference: CGVC 2022* (2022), Eurographics-European Association for Computer Graphics, pp. 65–73. doi:10.24406/publica-343. 2, 3, 4, 8

[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517. doi:10.1145/361002.361007. 2

[BK20] BORMANN P., KRÄMER M.: A System for Fast and Scalable Point Cloud Indexing Using Task Parallelism. In *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference* (2020), The Eurographics Association. doi:10.2312/stag.20201250. 2

[BKW22] BORMANN P., KRÄMER M., WÜRZ H.: Working Efficiently with Large Geodata Files using Ad-hoc Queries:. In *Proceedings of the 11th International Conference on Data Science, Technology and Applications* (2022), SCITEPRESS - Science and Technology Publications, pp. 438–445. doi:10.5220/0011291200003269. 3

[CPP17] CURA R., PERRET J., PAPARODITIS N.: A scalable and multi-purpose point cloud server (pcs) for easier and faster point

cloud data management and processing. *ISPRS Journal of Photogrammetry and Remote Sensing 127* (2017), 39–56. Geospatial Week 2015. URL: https://www.sciencedirect.com/science/article/pii/S092427161630123X, doi:https://doi.org/10.1016/j.isprsjprs.2016.06.012. 3

[DCSG*14] DOBOS L., CSABAI I., SZALAI-GINDL J. M., BUDAVÁRI T., SZALAY A. S.: Point cloud databases. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management* (2014), ACM, pp. 1–4. doi:10.1145/2618243.2618275. 3

[FB74] FINKEL R. A., BENTLEY J. L.: Quad trees a data structure for retrieval on composite keys. *Acta informatica 4*, 1 (1974), 1–9. doi:10.1007/BF00288933. 2

[Gut84] GUTTMAN A.: R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data - SIGMOD '84* (1984), ACM Press, p. 47. doi:10.1145/602259.602266. 2

[GVOC18] GUAN X., VAN OOSTEROM P., CHENG B.: A Parallel N-Dimensional Space-Filling Curve Library and Its Application in Massive Point Cloud Management. *ISPRS International Journal of Geo-Information 7*, 8 (2018), 327. doi:10.3390/ijgi7080327. 3

[Hil91] HILBERT D.: Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen 38* (1891), 459. doi:10.1007/978-3-662-38452-7_1. 2

[Hob24] HOBU .: Entwine. https://entwine.io/, 2024. Accessed: 2024-02-27. 2

[Ise13] ISENBURG M.: LASzip: Lossless compression of LiDAR data. *Photogrammetric Engineering & Remote Sensing 79* (2013). doi:10.14358/PERS.79.2.209. 4

[KB21] KOCON K., BORMANN P.: Point cloud indexing using Big Data technologies. In *2021 IEEE International Conference on Big Data (Big Data)* (2021), IEEE, pp. 109–119. doi:10.1109/BigData52589.2021.9671659. 3

[KBW*24] KRÄMER M., BORMANN P., WÜRZ H. M., KOCON K., FRECHEN T., SCHMID J.: A cloud-based data processing and visualization pipeline for the fibre roll-out in germany. *Journal of Systems and Software* (2024). doi:https://doi.org/10.1016/j.jss.2024.112008. 1

[LBA16] LIU K., BOEHM J., ALIS C.: Change detection of mobile lidar data using cloud computing. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences 41* (2016), 309–313. doi:10.5194/isprsarchives-XLI-B3-309-2016. 3

[LDME*22] LUMBRERAS M., DIARCE G., MARTIN-ESCUDERO K., CAMPOS-CELADOR A., LARRINAGA P.: Design of district heating networks in built environments using GIS: A case study in Vitoria-Gasteiz, Spain. *Journal of Cleaner Production 349* (2022), 131491. doi:10.1016/j.jclepro.2022.131491. 1

[LHL*21] LI B., HOU J., LI D., YANG D., HAN H., BI X., WANG X., HINKELMANN R., XIA J.: Application of LiDAR UAV for High-Resolution Flood Modelling. *Water Resources Management 35*, 5 (2021), 1433–1447. doi:10.1007/s11269-021-02783-w. 1

[Lid24] Lidarserv github repository. https://github.com/igd-geo/lidarserv/tree/v.1.1.0, 2024. Accessed: 2024-02-01. 2, 5

[LLW*21] LI X., LIU C., WANG Z., XIE X., LI D., XU L.: Airborne LiDAR: State-of-the-art of system design, technology and application. *Measurement Science and Technology 32*, 3 (2021), 032002. doi:10.1088/1361-6501/abc867. 1

[LRLPSC22] LADRA S., R. LUACES M., PARAMÁ J. R., SILVA-COIRA F.: Compact and indexed representation for LiDAR point clouds. *Geo-spatial Information Science 0*, 0 (2022), 1–36. doi:10.1080/10095020.2022.2121664. 3, 4

[LVOMV20] LIU H., VAN OOSTEROM P., MEIJERS M., VERBREE E.: An optimized SFC approach for nD window querying on point clouds. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences VI-4/W1-2020* (2020), 119–128. doi:10.5194/isprs-annals-VI-4-W1-2020-119-2020. 3, 8

[Mea82] MEAGHER D.: Geometric modeling using octree encoding. *Computer Graphics and Image Processing 19*, 2 (1982), 129–147. doi:https://doi.org/10.1016/0146-664X(82)90104-6. 2

[Mor66] MORTON G. M.: *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966. Technical Report. International Business Machines Company New York. 2

[NDAK20] NATHAN V., DING J., ALIZADEH M., KRASKA T.: Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), ACM, pp. 985–1000. doi:10.1145/3318464.3380579. 3

[PB07] PFEIFER N., BRIESE C.: Geometrical aspects of airborne laser scanning and terrestrial laser scanning. In *IAPRS* (2007), vol. 36, Citeseer, pp. 311–319. URL: http://hdl.handle.net/20.500.12708/42102. 1

[PDA24] PDAL - Point Data Abstraction Library. https://pdal.io/, 2024. Accessed: 2024-06-05. 6

[PSW20] PYSZNY K., SOJKA M., WRÓŻYŃSKI R.: LiDAR based urban vegetation mapping as a basis of green infrastructure planning. *E3S Web of Conferences 171* (2020), 02008. doi:10.1051/e3sconf/202017102008. 1

[RBBL21] RAMSEY P., BLOTTIERE P., BRÉDIF M., LEMOINE E.: pgPointcloud - A PostgreSQL extension for storing point cloud (LIDAR) data. https://pgpointcloud.github.io/pointcloud/index.html, 2021. Accessed: 2021-07-19. 3, 4

[RIE22] RIEGL VQ-1560i-DW Data Sheet. http://www.riegl.com/uploads/tx_pxpriegldownloads/RIEGL_VQ-1560i-DW_Datasheet_2022-02-22.pdf, 2022. Accessed: 2024-02-29. 6

[Sch16] SCHÜTZ M.: Potree: Rendering large point clouds in web browsers, 2016. 3, 4

[SOW20] SCHÜTZ M., OHRHALLINGER S., WIMMER M.: Fast Out-of-Core Octree Generation for Massive Point Clouds. *Computer Graphics Forum 39*, 7 (2020), 155–167. doi:10.1111/cgf.14134. 2, 4

[TAL22] TON B., AHMED F., LINSSEN J.: Semantic Segmentation of Terrestrial Laser Scans of Railway Catenary Arches: A Use Case Perspective. *Sensors 23*, 1 (2022), 222. doi:10.3390/s23010222. 1

[VBB22] VALERO E., BOSCHÉ F., BUENO M.: Laser scanning for BIM. *Journal of Information Technology in Construction 27* (2022), 486–495. doi:10.36680/j.itcon.2022.023. 1

[WYTT21] WU C., YUAN Y., TANG Y., TIAN B.: Application of Terrestrial Laser Scanning (TLS) in the Architecture, Engineering and Construction (AEC) Industry. *Sensors 22*, 1 (2021), 265. doi:10.3390/s22010265. 1

[ZY10] ZHANG J., YOU S.: Supporting web-based visual exploration of large-scale raster geospatial data using binned min-max quadtree. In *Scientific and Statistical Database Management* (Berlin, Heidelberg, 2010), Gertz M., Ludäscher B., (Eds.), Springer Berlin Heidelberg, pp. 379–396. 2, 3