PC-LMT: The Point Cloud Log Merge Tree for the Helena Point Cloud Database

Balthasar Teuscher
Professorship Big Geospatial Data Management
Technical University of Munich, Germany
balthasar.teuscher@tum.de

Martin Werner
Professorship Big Geospatial Data Management
Technical University of Munich, Germany
martin.werner@tum.de

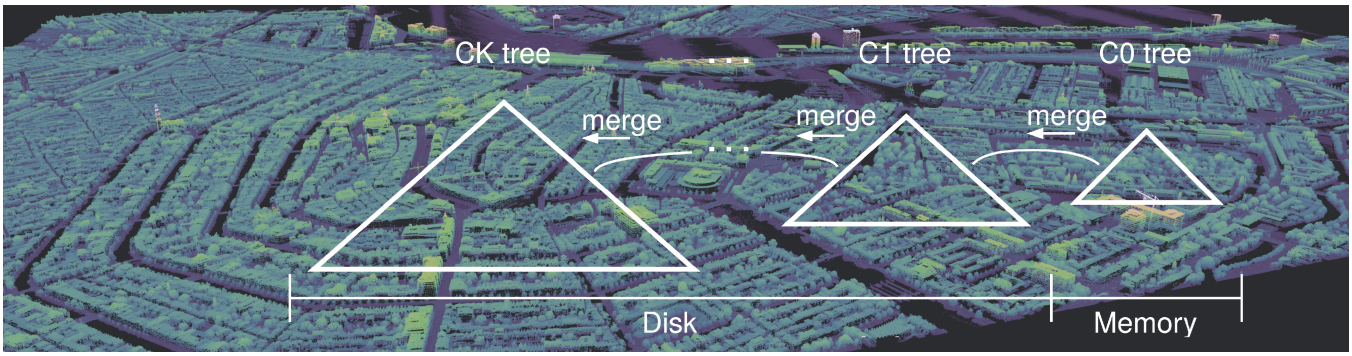

Figure 1: Point cloud visualization of Amsterdam from AHN3 with a log-structured merge-tree (PC-LMT).

Abstract

Point cloud data analysis and visualization workflows traditionally involve the sequential steps of information retrieval and preceding extensive data preparation. For example, visualizing large point clouds often takes days of processing to translate the data into a suitable representation before visual feedback is possible. While this works fine for static datasets and time-insensitive result consumption, it is unsuitable for dynamic contexts requiring real-time analysis, such as autonomous navigation. To address these shortcomings, we propose a point cloud data management approach based on a log-structured merge-tree that facilitates concurrent and continuous data ingestion and retrieval in real-time at scale. In this paper, we illustrate how to adapt this data structure to the peculiarities of point clouds and how various use cases and query modalities can be supported and optimized by specialized merge operations to repartition and refine the data structure and layout. This includes relying on grid-rounded coordinates and integrating importance as a means for effective and efficient storage, processing, and sampling from point clouds. Initial experiments and evaluation results display promising results and affirm the viability of this approach for Helena, a conceptual next-generation point cloud data management platform for interactive visualization and real-time analytics.

BigSpatial'24, October 29-November 1, 2024, Atlanta, GA, USA

ACM ISBN 979-8-4007-1143-5/24/10
https://doi.org/10.1145/3681763.3698476

CCS Concepts

• **Information systems** → **Database design and models**; *Multi-dimensional range search*; *Data structures*; *Data layout*.

Keywords

Point Cloud, Data Management System, LSM-tree

ACM Reference Format:
Balthasar Teuscher and Martin Werner. 2024. PC-LMT: The Point Cloud Log Merge Tree for the Helena Point Cloud Database. In *12th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial'24), October 29-November 1, 2024, Atlanta, GA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3681763.3698476

1 Introduction

Point clouds have gained increasing interest in recent years due to point cloud data generating sensor improvements and dissemination. The variety of sensors deployed on emerging platforms like mobile phones, drones, or cars led to an unprecedented growth of point cloud data volume. Further, the velocity to process and analyze such data reached new heights through autonomous navigation and change monitoring applications.

These developments demand ingesting and analyzing high volumes of data in real time, which presents new challenges for existing point cloud data management solutions. Traditional approaches are commonly constrained to time-insensitive batch processing and time-consuming transformations to enable analytical or visualization workloads [1, 4]. Fortunately, solutions from the broader field of Big Data exist that can help address these domain-specific shortcomings.

This paper explores the potential of a log-structured merge-tree (LSM-tree) data structure for point cloud data management. LSM-trees are widely integrated into database solutions to handle write-heavy workloads such as real-time data processing and stream processing [5]. The main benefits of such an approach are high load throughput while offering real-time query capabilities. We show that this approach provides the following benefits compared to existing solutions for point cloud data management:

- high load/write throughput
- loaded data is instantly queriable
- iterative refinement in the background
- adaptability to various query patterns
- scalability to large datasets
- support for grid-rounded coordinates

While the above characteristics are native to the vanilla LSM-tree data structure, the peculiarities of point cloud data require some adaptations. For one, the internal realization of LSM-trees based on a key-value store with an ordered keyspace requires mapping points onto one dimension. Although the creation time or a lexicographical order is possible, we consider a Z-order curve as a baseline to preserve some spatial locality.

Additionally, by using grid-rounded coordinates and introducing elaborated merging strategies, it is possible to refine the storage layout to various use cases and scenarios for increased query efficiency. For example, one could mimic a bounding volume hierarchy based on point importance sampling, commonly implemented in data structures optimized for visualization.

The remainder of this paper is organized as follows. Section 2 provides the foundational background on point cloud data representations and the original log-structured merge-tree data structure. Section 3 presents our adaptation of a point cloud log merge tree (PC-LMT) and its integration into the Helena Point Cloud Database. Section 4 discusses the algorithmic implications more in-depth. Section 5 evaluates the system on key metrics. Finally, Section 7 concludes the paper and gives an outlook on future work.

2 Background

In this section, we present the foundational concepts of this work. We start by describing point clouds in a formalized manner, including a grid-rounded perspective and its implications. We then continue with the aspect of importance augmentation for sampling points, a necessity for visualizing large point clouds. Finally, the original concept of a log-structured merge-tree (LSM-tree) is introduced to make ground for the following chapters.

2.1 Point Clouds

A point cloud is a set of points in some geometric space with the interesting property that they are invariant under ordering. More formally:

Definition 1. *Let M be a metric space. A point cloud P in M is a set $P = \{p_i \in M\}$ of points. Two point clouds are equal if the sets are equal; hence, the order of the points does not matter. The* ***dimension*** *of the point cloud is the dimension of M.*

Point clouds commonly represent observations from laser scanners, photogrammetry, or SLAM. They can be georeferenced by associating coordinate dimensions with geographic coordinate reference systems (CRS), height systems, or time epochs and scales. Such point clouds are coined **geographic point clouds** if a geographic reference is associated or **spatiotemporal point clouds** if both a geographic and a temporal reference are associated.

For archival purposes and exchange, point clouds are commonly stored in the LAS/LAZ file format [3]. The location encoding of this format adds two important but often overlooked aspects to the theory of point clouds:

Definition 2. *The* ***integral grid*** *in $\mathbb{R}^n$ is the set $\mathbb{Z}^n \subset \mathbb{R}^n$ of points with integer coordinates.*

Definition 3. *Given a metric space M, an* ***affine grid*** *G in M is the image of the integral grid under an affine map $\alpha : \mathbb{R}^n \to \mathbb{R}^n$:*

$$G = \left\{\alpha(z) \in M : z \in \mathbb{Z}^n\right\}$$

This leads to the following essential representation of point clouds:

Definition 4. *A point cloud P is grid-grounded with respect to an affine grid G if it is a subset thereof.*

A point cloud can be easily rounded into a grid-rounded point cloud as follows:

Given a point cloud P and an affine grid G, the grid rounding algorithm replaces each point p in P with its nearest neighbor in G.

The value of grid-rounded point clouds is that they no longer need to represent point locations with floating point coordinates. Instead, we can remember the affine map α and the integral coordinates of the grid.

This representation is the core of the LAS file format, and it has the following important advantages over using floating point numbers:

- better representation accuracy
- smaller memory footprint
- efficient and robust computation

The space efficiency of points in point clouds is paramount, given their significant amount, and problems with visualizing or computing tasks with point clouds are prevalent due to their excessive size.

2.2 Importance in Point Clouds

The problem of visualizing large point clouds is commonly solved by creating a layered quad- or octree[9]. Its nodes represent a hierarchical space partitioning, among which the points are distributed respecting a given point budget per node. The points of a given node are selected such that they are a representative sample of the original point cloud within its bounds. From this, it follows naturally that through the depth of the node in the tree, an implicit discrete importance is given to the points. This is sometimes called Level of Detail (LoD), as these levels are mapped to the camera position so that the points from a few nodes fill the viewport when visualizing.

Having some importance assigned to points in point clouds is necessary for visualizing large point clouds. However, the implicit

discrete importance of a tree is not enough to support generic sampling over the point cloud, which is desirable for analytical approaches. An example is sampling pointsets of different densities for machine learning models[7]. Others are methods that work on small randomized samples to approximate the whole dataset, an approach increasingly valid once point clouds grow larger.

In the following, we present and discuss two methods for defining the importance of points that facilitate arbitrary sampling from point clouds.

2.2.1 Random Importance. Random importance in point clouds can be achieved by assigning each point with a quasi-distinct random importance value from a normal distribution, for example, a random floating point value in the range (0...1). This concept is sometimes called continuous Level of Importance (cLoI), a substitution for LOD. With a random importance attribute, sampling becomes a range query over these importance values, which can be combined with spatial range predicates. Hence, this makes it possible to resolve arbitrary bounding volume with a given density or point budget. While random importance gives a decent sample quality, one can further improve importance generation with dart throwing.

2.2.2 Dart Throwing. As noted in the previous paragraph, sampling equates to an importance range query, and thus, the quality is given by the underlying randomness. A strategy for generating higher-quality samples is Disk- or Blue Noise sampling[2]. Sampled points in these approaches are more evenly distributed regarding the distance to their neighbors as selecting points in the vicinity of already selected points is precluded. At the same time, the distance should be minimal.

2.3 Log Merge Trees in Big Data

In Big Data, the log-structured merge tree (LSM-tree) is a foundational data structure [6]. It efficiently organizes key-value-structured data inserted into the database for key-based (individual) and range-based (key ranges) retrieval under high insert volume and velocity.

Definition 5. *A **key-value set** S is a set of pairs $(k, v) \in K \times V$ where K is a domain of keys and V is a data domain.*

In common cases, both keys and values are modeled as strings (or more generally as variable length byte arrays) to bring maximum flexibility into the system. However, it is essential for LSM-trees that the keys K form an *ordered set*.

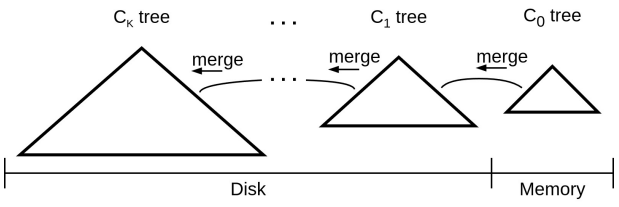

Figure 2: An LSM-tree of K + 1 components [6]

In a log-structured merge tree, data incoming to the system is kept in the main memory and written to a write-ahead log for persistence (Figure 2). Whenever this log in-memory and on-disk exceeds a predefined size, a **log rotation operation** is triggered, and a new log file and memory buffer pair is prepared for incoming data. The data of the previous log is written into a new file on Level 1 (L_1) of the LSM-tree. When this is successful, the write-ahead log on Level 0 (L_0) can be safely discarded, and the associated main memory can be reused.

This way, the main memory needed to accommodate incoming data is limited, and the individual file's sizes do not exceed a predefined threshold. If the system crashes, the log file can be replayed to reconstruct the L_0 data so that such systems support persistence and recovery from faults. However, this system can now generate more and more files on L_1, which are impossible to handle for queries (and the file system and operating system) over time.

Therefore, a log-structured merge tree often foresees additional levels obtained by a **compaction operation**. When the number of files on a certain level exceeds a threshold, multiple files with overlapping key ranges from this layer are selected to generate one or more files on a higher level. Commonly, the compaction operation will merge the sorted files into new files of disjoint key ranges with a designed maximal size for each. Consequently, throughout levels, data with similar keys will end up in the same file after one or more compactions, even if its insertion was scattered in time and thus scattered over many different log files. Note that this compaction operation can be elegantly implemented in a merge-sort without much memory requirement and no random disk I/O.

Now, when keys are arriving uniformly at random at L_0, all files on this level and on L_1 (which are just sorted versions of L_0 log slices) can contain information on a specific key and must, thus, be scanned during queries. Hence, the complexity of querying for a particular key is linear in the number of objects in the database. To optimize this linearity at least a bit, the compaction operation aims to reduce the number of data files that need to be examined for a given query by reducing the key range of each file. This can lead to efficient pruning of irrelevant files when querying for a specific key or key range.

LSM-trees are ubiquitous nowadays; for example, the embeddable implementation LevelDB originating in Google is nowadays in use as part of Google Chrome, as a backend to Riak, in Bitcoin and Ethereum, Minecraft, and Autodesk AutoCAD. In addition to this library, stand-alone databases employing LSM-trees include Apache Cassandra, a distributed, high-performance key-value store used by some major tech companies.

In the following, we generalize log merge trees to spatial data in various ways. Therefore, it might be helpful to remind us of the following facts about classical key-value log merge trees:

- The key range of every L_0/L_1 file is expected to exhaust the key space as the keys can arrive in uniformly random order.
- The compaction operation should not only merge files but also split them. Otherwise, large gaps in the key space do not pay off in queries.
- The system's performance is primarily defined by the number of files needed to be examined for each query. However, the number of files in the system can put pressure on real-world implementations and must be kept small enough with care.

- The data structure is optimized for range browsing (finding the minimum key followed by scanning) and lookup of exact keys

We will discuss these facts during the generalization as point clouds differ significantly from the simple one-dimensional case of key-value stores.

3 The Point Cloud Log Merge Tree (PC-LMT) and the Helena Point Cloud Database

This section introduces the Helena Point Cloud Database and its operations. In addition, we cover the query semantics envisioned for efficiently working with point clouds and a variant of a Log Merge Tree infrastructure for managing point cloud data.

3.1 The Helena Point Cloud Database Life Cycle

In this section, we briefly introduce the Helena Point Cloud Database operations along the data life cycle, which consists of one or more applications from the following four larger families of operations.

- **init:** initialize and configure a data space
- **load:** append data to the database
- **query:** retrieve data to the database
- **maintain:** plan and run maintenance steps, including merge, split, index, and cleanup operations

The Init Operation. Firstly, the **init** function initializes a new database. This encompasses creating a folder and associating selected metadata with the database and an empty log. The most crucial aspect to fix is the metadata of the point cloud representation, for example, information on geospatial projection and affine placement of data. Finally, the data type for holding points is fixed, and we prefer an integer representation for optimal compressibility and memory efficiency.

The Load Operation. After initializing the database, the point cloud database provides an API call **load** that appends points to the log. Note that we explicitly do not expect the data to respect the grid setting of the database. However, each load query can contain only one grid. The grid specification is recorded in the log, followed by the points.

When the log file contains a certain number of points allowed, a log rotation is performed, and the in-memory points are reorganized and written into a L_1 file. This reorganization applies the *PCL-DB Grid Consolidation Algorithm* to consolidate multiple grid systems as described in Section 4.1. We decided for the L_1 representation to be HDF5 as it provides (1) a highly optimized storage system, (2) support for attributes and slicing from disk, and (3) supports parallel file systems in HPC environments, and (4) has built-in compression support. Using any other storage container (e.g., flat file, binary, etc.) is possible.

The Query Operations. The database allows the user to perform queries of four categories:

- A **point query** to check whether a certain point is set in the database and retrieve the storage location triple.
- A **range query** retrieving *all* points in a specified range.
- A **importance sample query** retrieving *a specified fraction* or *a specified number* of points in a given range giving (probabilistic) preference to "important" points.
- A **random sample query** retrieving *a specified fraction* or *a specified number* of points in a given range, giving uniform probability to all points.

Note that the first two operations are well-aligned with classical use cases of LSM-trees. We can hope to quickly locate any item based on its key, and we can expect data locality to improve along the layer structure so that queries will be efficient.

The two latter queries, however, are entirely different from what classical databases provide efficiently and counter-advocate data locality as the only design principle. When point clouds grow large, most of the points are not needed as points get their meaning from the statistics of their surroundings, and thus, not all points are required. For example, in visualization, there is commonly a point budget of a few million points the user's GPU can render in real time. Also, in point cloud data mining and deep learning, the results do not rely too much on data completeness.

The Maintenance Operations. The maintenance of the log merge tree is performed as a background task called **maintain** that is run periodically, event-based (e.g., on log rotation), or on explicit administrator request. Four different maintenance tasks can be distinguished:

- Reproject point cloud data from multiple grid systems towards the single grid system prescribed in the init operation
- Merge, restructure, and split multiple files from a given level of the LSM-tree into a higher level
- Replay the log after crashes
- Drop files that are not needed anymore

3.2 Probabilistic Query Semantics through an Additional Attribute

As described before, data locality somewhat contradicts queries that are supposed to return a sample of the query region. In previous work, we have fixed a mechanism for embedding various such probabilistic query semantics by integrating an additional dimension to the data called importance (even if it is random) [10]. As such, the importance is represented by values from a known range (e.g., $0 \ldots 1$ for floating point coordinates, the full representable number range for integer coordinates). For the remainder of the paper, we will discuss the importance attribute in terms of the unit interval, though it will often be technically represented as an integer interval.

To support the various semantics of this variable, subranges of the value range of this importance variable are to be associated with a specific semantic. Therefore, we decompose the importance attribute into disjoint intervals and associate a particular meaning and order to the data represented in those.

The default approach decomposes the data into an importance sample followed by multiple random samples.

$$I = [0, 1] = [0, \alpha_{\text{Importance}}] \amalg S_0 \amalg \ldots \amalg S_m$$

In this representation, a parameter $\alpha_{\text{importance}}$ models how much data shall be organized according to visual point importance and how many independent random samples S_i are kept. Note that

within each of those importance regions, the data on disk is organized in lexicographic order to maximize compression efficiency. In contrast, we can optionally organize the importance sample ordered by an importance value such that slices of the importance region represent more or less important points.

The query algorithm needs to ensure that the query semantics pertain to this representation; for example, we would expect that random samples contain important points. As a convenience function for the user, we propose a stratified reservoir sampling algorithm for querying the database with a point budget in Section 4.5.

3.3 Physical Storage Layout

To comprehend the algorithms presented in the subsequent section, knowing how the point cloud data is represented on disk is essential. The physical storage layout for classical key-value-based LSM-trees is comparably simple: data is represented as a sorted sequence of key-value pairs called "run". Each run can be individually indexed (e.g., using a B-tree) and compressed, and a Bloom filter data structure can be leveraged to represent the set of keys in every run. Finally, the key range (e.g., the minimal and maximal keys in each file) is stored.

From this information, query processing first finds all files overlapping the query, checks the Bloom filter and index structures, and then materializes the data from the disk.

The physical storage layout for our approach is a bit more complex as we do not have a key-value structure to sort after. It would now be easy to sort the data on disk by the importance dimension, a crucial aspect for querying the data. However, this is inefficient in terms of storage, especially for integer-rounded point clouds, and it requires that we either store for each point its projection information individually as the affine map placing the integer grid in world coordinates will have to vary in large (e.g., continental scale) datasets.

Therefore, we employ a hierarchical bucket sort storage order in which we first sort by grid system, then by importance interval, and inside each importance interval in a configurable manner, choosing from:

- random storage order providing the fastest access to random samples by slicing out pieces of the file,
- lexicographic storage order providing best compression,
- importance storage order providing slicing for the interval $I_{\text{Importance}}$ as well.

This storage layout has a scalability problem in terms of the number of affine maps, and therefore, an affine compaction operation is introduced, reducing the number of affine maps in the database described in Section 4.4

4 Algorithms for Handling Point Clouds in Log-Structured Merge-Trees

This section details the core algorithms designed to manage point cloud data efficiently within an LSM-tree data structure. This includes algorithms for consolidation, merging, sorting, and tiling strategies, as well as compaction and sampling approaches. Combined, they seek to optimize the data structure and layout through repartitioning for different use cases, analogous to creating a layered octree, commonly done for visualization purposes.

4.1 PCL-DB Grid Consolidation Algorithm

As described before, point clouds are commonly represented in a grid-rounded fashion, where only integer coordinates are stored referencing floating-point coordinates in arbitrary CRS through an affine map consisting of translation, scale, and rotation, essentially "placing" the integer grid in the real world.

For a database as described in this paper, this means that over time, multiple grids will occur in the database, for example, a different grid for each load operation. In the log, this can efficiently and effectively be managed by putting the grid specification (e.g., the parameters of the affine map) into the log data stream, implementing a stateful engine.

However, when consolidating the database with merge operations, spatial overlap might imply different grid systems in the input data of a merge operation. At the same time, we expect the output to be homogeneously represented in only one grid system per output file.

This leaves us with two interesting algorithmic problems formulated as follows:

Problem 1. *Given a set of k input point clouds, each with an associated affine map A_k, estimate a single grid affine map $\bar{A}$ minimizing the distortion of data in the A_k.*

This problem can be further complicated when we allow the number of output affine maps to be non-zero.

Problem 2. *Given a set of k input point clouds C_i, each with an associated affine map A_i, decompose the input points into $l \leq k$ point clouds C_j, each associated with its affine map A_j such that the spatial overlap of the C_i is small.*

This problem is more involved and cannot be generally solved without background information. Therefore, we will only provide a solution in the context of a prescribed global grid of affine maps resembling a grid system for the underlying CRS.

This often requires a configured preference of grid placements where one can easily imagine aligning the grid systems successively with grid systems for the world cover, including the UTM coordinate grid, the Sentinel-2 grid, or the military grid system (MGS). This way, data across otherwise isolated systems will find a homogeneous representation.

4.2 The Point Cloud Merge Operation

During the system's lifetime, loading data into the database generates a sequence of log files. Though some workloads (e.g., loading a tiled set of files) have certain structures, we generally do not expect any prior knowledge about the relation of the files. The merge operation aims to restructure a certain number of files from some level l into a more structured, commonly smaller set of files on level $l + 1$.

More concretely, the merge operation needs to solve the following problem:

Problem 3. *Given a log-structured merge-tree, the merge operation **selects** several files on a level l and **emits** one or more files on level l and $l + 1$.*

Multiple ways of selecting files are possible without prior knowledge of the relations between files, leading to different advantages and drawbacks.

In this paper, we select files based on the amount of overlap of their bounding boxes. Then, a new file on a higher layer is computed after selecting several files on one layer. For this, the following basic strategies have been identified:

- **union:** all points are concatenated
- **shuffle:** all points are taken, but recorded in shuffled order
- **lexicographic:** all points are sorted in lexicographical order over the configured dimensions
- **dimension_monotonous:** a selected dimension is used to sort the points
- **dartthrowing:** all points are taken and ordered by importance such that at the beginning of the result file of the merge, highly important points are recorded
- **tiling**: points are sorted into tiles as configured for the dataspace

Note that all of these operations generate a single data object. Still, typically, the merge operation is chained with a tiling operation described in the following subsection (4.3).

Subsequently, the strategies, as mentioned above, are described in more detail, including discussing their properties, advantages, and disadvantages.

4.2.1 The Union Strategy. The following algorithm precisely defines the union strategy:

Given m files on layer k, the merge operation with union strategy emits one file on layer $k + 1$ by concatenating the points from the m files in lexicographic order of filenames (and, thus, in temporal order).

The **union** strategy is a common baseline and excels in write speed and implementation efficiency. When merging multiple files, we read all input files and write to a single output file. Consequently, we can efficiently implement this strategy with only two file handles and a freely chosen amount of main memory for sequential reading and writing a block.

However, depending on the order of the input files, this will create artifacts and inefficiencies in higher layers as the data for a specific type of query is scattered in the files depending on how the data has been ordered at input time. For example, when the input has a tiled structure commonly found in published point clouds, the random selection of slices for the merge operation generates files whose bounding boxes are largely overlapping. As a consequence, range queries would have to read data in a random I/O fashion from a high number of files.

Summary: The union strategy is the fastest regarding I/O. It is reasonable only for database bulk loading where the input order has been chosen wisely and shall be retained.

4.2.2 The Shuffle Strategy. The following algorithm precisely defines the shuffle strategy:

The shuffle strategy is the opposite of the union strategy: it removes any structures and artifacts introduced to the database from point ordering or load operation ordering.

Given m files on layer k, the merge operation with the shuffle strategy emits one file on layer $k + 1$ by concatenating the points from the m files in shuffled order.

In terms of implementation, however, shuffling large datasets is similarly complex to sorting data and requires a certain amount of computational resources (memory, CPU, disk I/O). Furthermore, the strategy is the worst regarding data compression, as discussed in the evaluation. At the same time, random samples of point clouds have a high user value in visualization and deep learning, and shuffling part of the data can imply efficient queries as any block of the shuffled data is a random sample of the inputs.

Summary: The shuffle strategy reduces artifacts introduced due to ordering and random choice of slices. Furthermore, it makes queries for random samples efficient. However, it requires quite some computation and reduces the data compression ratio.

4.2.3 The Lexicographic Strategy. The following algorithm precisely defines the lexicographic strategy:

Given m files on layer k, the merge operation with the lexicographic strategy emits one file on layer $k + 1$ by sorting the input in lexicographic order along its data dimensions.

It is a bit counter-intuitive why we would want a lexicographic order of point clouds: basic range queries, even with a small result set, will lead to random input-output (I/O) as the last dimension is spread across the whole file.

However, combining the fact that many point clouds are grid-rounded, as implied by the LAS/LAZ file format, the lexicographic ordering maximizes the nearby co-occurrence of sequences of coordinates: in the first N rows of a lexicographically ordered point cloud from a grid, the first few coordinate axes are likely equal. This leads to extraordinary compressibility of the file.

Summary: The lexicographic strategy introduces redundancy in data blocks for best compressibility at the expense of random I/O for most queries. It provides a good ordering for network transport.

4.2.4 The Dimension-Monotonous Strategy. The following algorithm precisely defines the dimension-monotonous strategy:

Given m files on layer k and a selected dimension (e.g., X, Y, Z, or importance), the merge operation with dimension-monotonous strategy emits one file on layer $k + 1$ by stably sorting the points from the m files by the selected dimension.

This strategy can be used for geometric slicing: When data is ordered in an X-monotonous fashion, bounding boxes and range queries in X dimension will be fast. Over multiple levels, an alternating pattern of X and Y monotonous ordering can be used. However, the most likely approach is to use this strategy with the importance dimension such that queries for small importance intervals near zero can be answered from the beginning of the files.

4.2.5 The DartThrowing Strategy. The following algorithm precisely defines the DartThrowing strategy:

Given m files on level k and a dart radius, the DartThrowing strategy runs a dart-throwing algorithm with the given parameter and stores points in a way that, first, the points surviving the dart throwing are stored, followed by the points that have been omitted due to the dart throwing algorithm. An additional table represents the ranges of surviving and non-surviving points.

This approach ensures that a good sample of the set of files given is used and quickly accessible in the first file, while no data is lost due to storing the "long tail" of the set of files in another file. It would even be possible to drop those files if visualization is the system's only aim or to distribute the files to different nodes in a distributed system.

4.2.6 Tiling Strategy. The following algorithm precisely defines the tiling strategy:

Given m files on layer k and a tiling grid (2D or 3D), the tiling strategy generates a new file on layer $k + 1$ ordered by tiles. In addition to the points, a table of associations between point ranges and tiles is stored.

This strategy can be considered a batched version of the dimension-monotonous and lexicographic strategies where the values on the dimensions are binned regarding the tiling grid.

4.3 Tiling Strategies

The merge operation turns k input files into one logical output. Still, this output will often be too large to be written in a single file or a data distribution, where cutting the data into multiple files leads to smaller file footprints. Therefore, the tiling operation follows the merge operation, of which we list and discuss a few basic strategies in the following.

Tiling can be based on

- **fixed size**: the intermediate data is cut into chunks of equal size, each written to a different file
- **external tileset**: the data is written into tiles as described in the dataspace
- **internal tiling**: the data is tiled in a promising dimension

4.3.1 The Fixed Size Tiling Strategy. This strategy splits files surpassing a configured upper bound for the file size into multiple files of size with the bound. Through this, constraints from the underlying storage system or available memory capacity can be incorporated.

4.3.2 The External Tileset Tiling Strategy. This strategy follows a predefined partition scheme analogous to space partitioning data structures as present, for example, in spatial grid, quad- or octrees. The benefit is a known tiling scheme, which can be optimized further to specific query workloads when the data distribution is known beforehand.

4.3.3 The Internal Tiling Strategy. This strategy follows a data-driven approach similar to data partitioning in trees. As such, the tiles emerge from splitting the data based on some derived statistics. For example, split the bounding volume in the middle along the longest edge. The benefit of this strategy is that it will result in tiles of data with compact bounds adapted to the overall data distribution.

4.4 Affine Compaction Algorithm

Over time, loading a point cloud database from large collections of geospatial point clouds introduces many different affine grid systems into the database, limiting the organizational flexibility concerning data storage and putting a constraint on the efficiency of querying.

Therefore, this section introduces an algorithm called **Affine Compaction**, which collects multiple files in one layer with overlapping data from numerous affine projections and reorganizes them into multiple files with fewer projections per file. This operation does not elevate levels.

In addition, the maximal representation error of merging two affine maps is computed based on the bounding boxes, and an overall threshold is used to decide if a single one can replace the two representations. Immediately, two variants do exist, which behave differently in different workloads: in a majority strategy, the idea of merging two affine maps would merge the one with fewer points into the one with more points; in a mean merge, a new affine map is needed which minimizes the grid rounding distortion for both grids. As a third aspect, log merge trees shall converge to highly structured data through levels, such that we can also choose the best affine map in terms of distortion to a preconfigured affine map for the whole database. The latter has a clear advantage: a sequence of affine merge operations cannot accumulate small errors into a large one.

4.5 Stratified Reservoir Sampling for Count Limits on Query Results

In point cloud data management, an essential operation is generating random samples or importance samples of all data matching a query predicate. In this setting, it is most intuitive for the user to specify a point budget and to expect a good sample of the data fulfilling the query predicate being smaller than or equal to the given budget.

Given that the importance dimension is cut into slices of different semantics, we need to design a sampling algorithm that ensures that all slices of the importance dimension contribute fairly to the query result.

Problem 4. *The **Stratified Reservoir Sampling Problem** is given a point budget N, a set S_i of k point clouds (e.g., our disjoint slices in the importance dimension) and returns a point cloud C_i such that $|C_i| \leq N$ and the probability of a point being from a slice S_i shall be near $\frac{1}{k}$.*

This problem is comparably involved as the amount of points in the slices can differ. A first idea for an algorithm would be based on the concept of first computing (on the server side) the entire query and then constructing the random sample by sampling $N \cdot \frac{1}{k}$ points from every slice. However, slices can be of different sizes, and it might be that some slices are even smaller than $N \cdot \frac{1}{k}$.

In these cases, the result set can be returned, or an optional round-robin fill algorithm can be used.

More precisely

Algorithm 1 Stratified Reservoir Sampling with Budget Completion

procedure sample_reservoir(N,$S_1 \dots S_k$,P)
  $R = \emptyset$
  for $i = 1 \dots k$ **do**
    // compute shuffled subqueries
    $C_i \leftarrow$ Shuffled (Query(S_i, P))
    H_i, T_i
    $R = R \cup$ (firstN, C_i)
  end for
  // plan the amount of points per slice
  missingPts
end procedure

5 Evaluation

This section illustrates selected experiments from a proof of concept implementation. As such, they focused on exemplifying and validating the feasibility of concepts and approaches presented in this paper. Comparison with state-of-the-art systems, including quantitative performance metrics, are planned for the future once a proper implementation is available.

The datasets used in the following are subsets of the Actueel Hoogtebestand Nederland (AHN). All experiments are conducted on a mobile computer with 8 cores (16 threads), 32GB memory, and 1TB SSD storage.

5.1 Load Performance

The LSM-tree data structure is known for offering high write performance. While the ingested data can be kept in memory, it has to be spilled to persistent storage at some point. That gives us two upper limits: for one, write speed, including serialization and compression, and second, the read speed, including deserialization and decompression.

Our test case of reading LAZ files from the SSD into memory and writing to LAS or LAZ from memory to SSD resulted in a throughput of about 20 million points per second. This gives us an upper limit for data ingestion, which is exhausted and sustained in the load operation of our proof of concept, even when applying grid rounding on the fly.

5.2 Point Representation

In Figure 3, we plot the relative size of different point representations and formats. As a baseline, we use NumPy, which gives us the uncompressed on-disk and in-memory sizes of the native data types. This is compared against an HDF5 container with auto chunking and gzip compression, once with random ordered points (middle) and once with lexicographically sorted points (right). As payloads, we generated 5 million points, once with three normally distributed double precision coordinates in the range 0 to 1 (blue), once with normally distributed integers over the total value range (orange), and once with the upper range set to 2 million (green).

The results show that sorting is indeed beneficial for the compression ratio. Interestingly, this effect gets heightened when limiting the integer range to 2 million, resulting in about a 20% space decrease in the random case and up to 40% decrease in the sorted case compared to the uncompressed size. Although the data in this experiment is synthetically generated, the latter represents a grid-rounded approach that can cover a cube with a 2000-meter edge length with millimeter precision. Due to the internal representation of floating point numbers, their value range does not substantially influence the compression ratio.

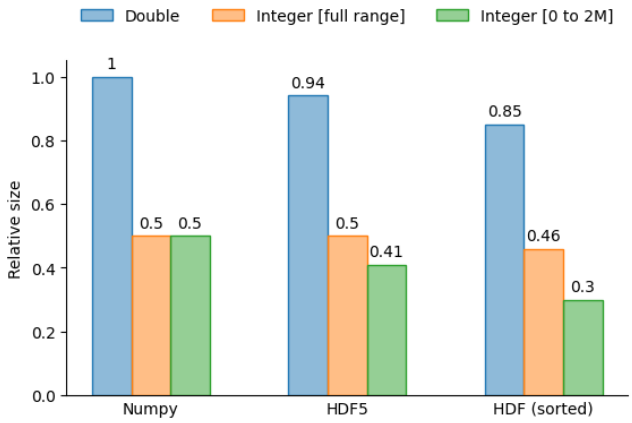

Figure 3: Storage footprint and compression efficiency for different coordinate representations.

5.3 Data Layout

Figure 4 depicts the spatial footprints of the files on different levels. We used an LAZ file from the AHN3 aerial laser scan as input. Upon the loading operation, we see that the scan lines are cut into pieces by the log rotation on Level 0 (left). After a merging operation with the union strategy, we end up with fewer though larger files on Level 1 (middle). Finally, through an internal tiling strategy, we end up with nonoverlapping files on Level 2 (right).

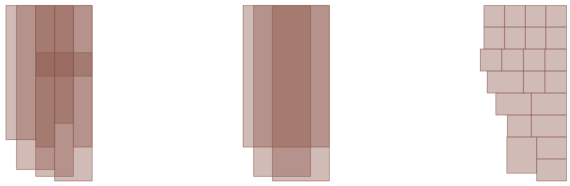

Figure 4: Spatial footprint of (log)-files on different levels. Level 0 files after the load operation (left), after a merge operation on Level 1 (middle), and on Level 2 (right) after tiling.

From the spatial footprints, it becomes evident that one can expect range queries to become more efficient as individual files are more compact, which allows for pruning outside the query extent. With this, we show that it is possible to end up with a data organization that resembles space-partitioning similar to the leaves in an R-tree or quadtree.

6 Discussion

This section discusses our approach in the context of traditional point cloud data management and processing and current research and development trends.

6.1 Arbitrary Space Partitioning

With our proposed approach, space partitioning methods like grids, space-filling curves, and quad- and octrees can be represented. This can be facilitated for static datasets by merely describing an external tileset definition accordingly and applying a tiling operation if the extent is known upon initialization. For dynamic datasets or streams, where the full extent is unknown, one approach is to describe an external tileset in terms of levels and deltas. In this scenario, levels are the layers of the tree, respectively, the fixed value ranges of the associated dimension. Deltas are the cell size of the remaining dimensions per level. From an even more generic view, we can support arbitrary space partitioning by defining a function that maps points to partitions $f : Points \mapsto Partitions$.

6.2 Workflow Conversion

Analogous to the example of arbitrary space partitioning, almost any existing processing workflow to process point cloud data can be facilitated in our proposed setup. This is true if we can formalize a workflow as a transformation function that takes a point cloud as input and returns a point cloud. Then, we can write the input upon reading to level 0, define the function as a maintenance operation, apply it, and write the output to the next level. Of course, this somewhat invalidates the usefulness of an LSM-tree in the extreme case of waiting for the whole function to be processed until we can transition to the next level. Nevertheless, most workflows targeting large datasets operate in a streaming, multi-stage pipeline, or batch-processing fashion. Such workflows offer great potential to be handled within an LSM-tree as single processing steps can be extracted as multiple maintenance operations and results can be made accessible on intermediate steps and iteratively change the level.

6.3 Related Work

While LSM-trees and similar concepts are well-researched, they have never been explicitly applied to point cloud or geospatial data to our knowledge. One explanation is that the inner workings of LSM-trees are generally not exposed to the user directly but hidden in database management systems. One exposure in this direction can be found in Apache Iceberg, where users can control, to some extent, the table's partitioning scheme. Another explanation is that workflows for data processing, often described as pipeline operations, run in a streaming or batch-processing fashion. Even though this is somewhat similar to the operations applied in this paper, the approach presented makes data accessible for the final use case right after ingestion and continuously while processing, moving to higher levels.

The intrinsic characteristics of LSM-trees, mainly high read throughput with subsequent data layout refinement while offering query capabilities upon ingestion, seem to gain emerging interest. Recently, SimLOD was presented as an approach to simultaneously build an octree for levels of details and render point clouds while loading from disk[8].

7 Conclusion and Outlook

This paper investigates the log-structured merge-tree (LSM-tree) data structure for point cloud data management. As such, we present the Point Cloud Log Merge Tree (PC-LMT), comprising the definition for grid-rounded point representation and customized algorithms for merge, tiling, and compaction operations. These adaptations of key aspects of an LSM-tree reveal the feasibility of point cloud data management. Its flexibility exhibits great potential to adapt to various use cases, as numerous dedicated point cloud data structures can be reproduced in a unified framework. The concept's soundness is successfully validated in a proof of concept, and we are excited to further explore and develop this promising approach.

Focus areas for the near future are to extend the implementation in terms of features and performance and target a distributed version thereof. Another area is integrating existing systems and components, for example, making them mappable to existing key-value stores.

References

[1] Howard Butler, Bradley Chambers, Preston Hartzell, and Craig Glennie. 2021. PDAL: An open source library for the processing and analysis of point clouds. *Computers & Geosciences* 148 (March 2021), 104680. https://doi.org/10.1016/j.cageo.2020.104680

[2] Robert L. Cook. 1986. Stochastic sampling in computer graphics. *ACM Transactions on Graphics* 5, 1 (Jan. 1986), 51–72. https://doi.org/10.1145/7529.8927

[3] Martin Isenburg. 2013. LASzip: lossless compression of LiDAR data. *Photogrammetric Engineering and Remote Sensing* (2013).

[4] Chamin Nalinda Lokugam Hewage, Debra F. Laefer, Anh-Vu Vo, Nhien-An Le-Khac, and Michela Bertolotto. 2022. Scalability and Performance of LiDAR Point Cloud Data Management Systems: A State-of-the-Art Review. *Remote Sensing* 14, 20 (Oct. 2022), 5277. https://doi.org/10.3390/rs14205277

[5] Chen Luo and Michael J. Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (Jan. 2020), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[6] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[7] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. 2017. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. (2017).

[8] Markus Schütz, Lukas Herzberger, and Michael Wimmer. 2024. SimLOD: Simultaneous LOD Generation and Rendering for Point Clouds. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 7, 1 (May 2024), 1–20. https://doi.org/10.1145/3651287

[9] Markus Schütz, Stefan Ohrhallinger, and Michael Wimmer. 2020. Fast Out-of-Core Octree Generation for Massive Point Clouds. *Computer Graphics Forum* 39, 7 (Oct. 2020), 155–167. https://doi.org/10.1111/cgf.14134

[10] Balthasar Teuscher and Martin Werner. 2024. Random Data Distribution for Efficient Parallel Point Cloud Processing. *AGILE: GIScience Series* 5 (May 2024), 1–10. https://doi.org/10.5194/agile-giss-5-15-2024