Density-Based Geometry Compression for LiDAR Point Clouds

Xibo Sun[1]
xsunax@connect.ust.hk
[1]Hong Kong University of Science and Technology

Qiong Luo[1,2]
luo@cse.ust.hk
[2]Hong Kong University of Science and Technology (Guangzhou)

ABSTRACT

LiDAR (Light Detection and Ranging) sensors produce 3D point clouds that capture the surroundings, and these data are used in applications such as autonomous driving, traffic monitoring, and remote surveys. LiDAR point clouds are usually compressed for efficient transmission and storage. However, to achieve a high compression ratio, existing work often sacrifices the geometric accuracy of the data, which hurts the effectiveness of downstream applications. Therefore, we propose a system that achieves a high compression ratio while preserving geometric accuracy. In our method, we first perform density-based clustering to distinguish the dense points from the sparse ones, because they are suitable for different compression methods. The clustering algorithm is optimized for our purpose and its parameter values are set to preserve accuracy. We then compress the dense points with an octree, and organize the sparse ones into polylines to reduce the redundancy. We further propose to compress the sparse points on the polylines by their spherical coordinates considering the properties of both the LiDAR sensors and the real-world scenes. Finally, we design suitable schemes to compress the remaining sparse points not on any polyline. Experimental results on DBGC, our prototype system, show that our scheme compressed large-scale real-world datasets by up to 19 times with an error bound under 0.02 meters for scenes of thousands of cubic meters. This result, together with the fast compression speed of DBGC, demonstrates the online compression of LiDAR data with high accuracy. Our source code is publicly available at https://github.com/RapidsAtHKUST/DBGC.

1 INTRODUCTION

A point cloud is a set of data points with spatial coordinates, generated by special cameras such as LiDAR (Light Detection and Ranging) sensors. These sensors are mounted on airplanes, vehicles, and tripods to capture large-scale scenes and structures. Various applications take point clouds as the input data, including survey [59], transportation [8], SLAM (Simultaneous Localization And Mapping) [4], and object classification [31]. Due to the high capture frequency of LiDAR sensors, LiDAR data are typically compressed for transmission and storage in resource-constrained environments. For example, Velodyne HDL-64E [9], a popular LiDAR sensor, generates about 96 Megabits data per second, whereas the bandwidths of 4G mobile networks are under 10 Megabits per second [41]. Furthermore, the compressed data must maintain a high accuracy in comparison with the original data so as to satisfy the downstream processing tasks after decompression. However, current work on LiDAR data compression often sacrifices accuracy for compression ratio. Therefore, in this paper, we study how to compress LiDAR point clouds effectively under a given small error bound.

Point cloud compression is performed on 3D coordinates, so we call it *geometry compression* to distinguish from general data compression. Early approaches on point cloud compression, including space partitioning [3, 36, 48], clustering [17, 57], and transformation [40, 56], focus on 3D object scans, where points are densely and uniformly sampled from the object surface. In contrast, LiDAR data capture much larger physical space and are much more sparse than object point clouds. Furthermore, LiDAR point clouds are sampled uniformly from the sensor to the entire physical space, as opposed to individual objects. As a result, traditional methods on object clouds as well as their adaptations [5, 27, 33, 45] do not work effectively on LiDAR point clouds.

Another feature of LiDAR sensors is that they can output raw point clouds where the spherical coordinates of the points form a regular grid. Taking advantage of this feature, a recent approach [53, 54] maps a raw LiDAR point cloud to a 2D image with respect to the relative position of each point to the sensor, and then compresses the image. However, this approach bears a low compression accuracy in comparison with the calibrated point cloud, due to the noise and errors in the raw data. Such accuracy reduction may be acceptable by tasks such as visualization and SLAM, but not by surveys or measurement applications that require a small error range between the original point cloud and the decompressed one.

Considering the drawbacks of existing LiDAR data compressors, we propose a compression scheme for LiDAR point clouds that preserves the geometry accurately as well as improves the compression ratio. As most applications use the point cloud after calibration, or the *released point cloud*, as the original data, our work has no assumption about the raw point cloud feature and is applicable to both raw and calibrated point clouds. We focus on compressing single-frame point clouds as opposed to a stream of point clouds because (1) some LiDAR sensors can capture static scenes only, (2) some downstream applications select specific frames of LiDAR data to process [2, 20], and (3) single-frame compression can be a building block in compressing point cloud streams.

Specifically, we design an effective point cloud compression scheme, the main idea of which comes from our observation on LiDAR data. As an example, Figure 1 shows the *xoy* plane projection of a real-world city-scene point cloud from the KITTI [22] dataset. In this figure, we can see a pattern similar to a spider web of diverse density: the polylines surrounding the central area are dense, and from the central area outwards the polylines gradually become sparse. However, previous research has not exploited this pattern, but treated all points uniformly in their schemes. As shown in our experiments, the compression ratios of latest algorithms, e.g., the octree compression, decrease significantly as the point cloud becomes sparser. To alleviate this problem, we propose to use the octree to compress only the

© 2023 Copyright held by the owner/author(s). Published in Proceedings of the 26th International Conference on Extending Database Technology (EDBT), 28th March-31st March, 2023, ISBN 978-3-89318-093-6 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

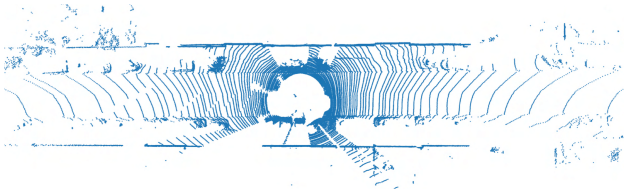


Figure 1: A LiDAR point cloud in 2D Cartesian space.

points in dense regions. To identify these dense points, we adopt density-based clustering on points and set its parameter values to satisfy the user-defined error bound. For efficiency, we first build the octree for all points, and then perform clustering based on the cells of the tree: if a cell contains a dense point, all points in the cell are kept in the cell; otherwise, the cell is removed from the octree. This way, the octree covers all points in dense regions and the compression ratio is improved.

After constructing the octree for dense points, we further organize sparse points into similar polylines in the spherical coordinate system. We define the similarity between polylines by that of the sequence of points on the polylines. This polyline organization is to facilitate the compression of point coordinates. Considering the characteristics of the real-world LiDAR data, we design a radial distance optimized delta encoding scheme and compress the sparse points in their azimuthal and polar angles.

Additional contributions within our approach include (1) coordinate scaling, which takes advantage of the error bound to reduce data entropy; (2) grouping sparse points and then organizing each group to perform polyline compression; and (3) optimized outlier compression, which processes coordinates of outliers (i.e., sparse points not on any polyline) in each dimension separately.

All our compression components satisfy a user-given error bound, e.g., 0.02 meters error bound for each point, and together they achieve a high compression ratio.

We have implemented our compression scheme in a prototype system called **DBGC** and evaluated its effectiveness and efficiency. Our experiments compared our compression scheme with existing work, using large-scale point cloud data from different scenes, with the error bound varied. The results show that our compression scheme outperforms the state of the art considerably on effectiveness. We also evaluated the end-to-end time performance and compression bandwidths of DBGC. We find that the compression bandwidth of DBGC is big enough for the online processing of LiDAR sensor data generated at full speed.

In summary, we make the following contributions.

- We identify the drawback of octree compression on sparse point clouds and design a density-based clustering method to determine the dense and sparse points. Specifically, we set the parameter values of clustering based on the octree structure and the error bound, and improve both the clustering efficiency and the compression effectiveness.
- We propose to organize sparse points into polylines that have similar patterns in the spherical coordinate space. Then, we introduce a compression method for the coordinates of the sparse points on these polylines based on our radial distance optimized delta encoding.
- We propose other optimizations, including coordinate scaling, sparse point grouping for polyline organization, and optimized outlier compression, to further improve the compression ratio of our scheme.
- We design and implement an end-to-end compression system DBGC to evaluate our compression scheme. Then, we conduct experiments on real-world point clouds from different scenes under various settings to examine the effectiveness and efficiency of our approach.

2 PRELIMINARIES AND RELATED WORK

In this section, we present the preliminaries of point cloud compression and then introduce related work.

2.1 Preliminaries

First, we define a point cloud as follows.

Definition 2.1. A point cloud, denoted as $\mathbb{PC}$, is a set of points. Each point $p \in \mathbb{PC}$ is a tuple containing geometry information, represented in coordinates, and optionally, attributes such as intensity, color, and the surface normal at the point.

We denote $|\mathbb{PC}|$ as the number of points in $\mathbb{PC}$. In this paper, we focus on compressing the geometry information in a point cloud, as commonly done in previous work [27, 53, 54]. In the Cartesian coordinate system with the origin at o, a point p is represented by (x_p, y_p, z_p), where x_p, y_p, and z_p are the offsets from o to p on the x, y, and z dimensions. In comparison, in the spherical coordinate system with the origin at o, a point p is represented by (θ_p, ϕ_p, r_p), where θ_p and ϕ_p are the *azimuthal* and *polar angles* of the vector from o to p, respectively. r_p is the *radial distance* between o and p. Given $\mathbb{PC}$, we use c_{max} and c_{min} to denote the maximum and minimum values of the dimension c coordinates of all points in $\mathbb{PC}$.

We represent a polyline l as a sequence points, $l = \langle p_1, p_2, \cdots, p_n \rangle$, and $l.front$, $l.back$, and $l[i]$ are the first, last, and ith points in l, respectively. We also denote $\mathbb{PL}$ as a set of polylines and $\mathcal{L} = \langle v_1, v_2, \cdots, v_n \rangle$ as a sequence of numbers, where v_m is a number.

Given a point cloud $\mathbb{PC}$, a compression procedure C converts $\mathbb{PC}$ into a *bit sequence* $\mathcal{B}$, and C's corresponding decompression procedure DC converts $\mathcal{B}$ into the decompressed point cloud $\mathbb{PC}'$. We denote $|\mathcal{B}|$ as the size of $\mathcal{B}$ in bytes. We define the *compression ratio* as the ratio of the data size of $\mathcal{PC}$ measured in bytes to $|\mathcal{B}|$. In general, we refer to compression performance as the data size reduction capability. The higher the compression ratio, the better the compression performance.

As the decompressed point cloud $\mathbb{PC}'$ and the input data $\mathbb{PC}$ may differ, people define errors to measure the difference. Since our compression scheme ensures one-to-one mapping between points in the input point cloud and those in the decompressed point cloud, we define the error between the coordinates of two corresponding points in $\mathbb{PC}'$ and $\mathbb{PC}$ as follows.

Definition 2.2. Given a coordinate v of point p and v' of p', where $p \in \mathbb{PC}$ and $p' \in \mathbb{PC}'$, the error between v' and v is the absolute value of the difference between v and v'.

We call v the *actual* coordinate value and v' the *approximate* coordinate value, respectively. In our point cloud compression, we further specify the *error bound* q_c, as the maximum error allowed on each dimension c between the original point and the corresponding point in the decompressed point cloud. In the Cartesian coordinate system, the error bound on the x, y, and z dimensions are usually the same, so we denote a single q_{xyz} for all three dimensions for simplicity. In this paper, we study the geometry compression of point clouds with a high accuracy or small error bound.

Table 1: Notations

Symbol	Description
$\mathbb{PC}$	A point cloud
p	A point
(x_p, y_p, z_p)	The Cartesian coordinates of a point p
(θ_p, ϕ_p, r_p)	The spherical coordinates of a point p
c_{max},c_{min}	The maximum/minimum value of dimension c
l	A sequence of points, i.e., a polyline
$\mathbb{PL}$	A set of polylines
$\mathcal{L}$	A sequence of values
$\mathcal{B}$	A bit sequence
q_c	The error bound of dimension c

We summarize the notations and their descriptions in Table 1.

Finally, we define the problem of point cloud geometry compression with an error bound as follows, which is consistent with previous work [33, 48].

Problem Statement. Given an error bound q_c on each dimension c of a coordinate system, develop a point cloud geometry compression scheme C that satisfies the following conditions: (1) given a point cloud $\mathbb{PC}$, C will generate a bit sequence $\mathbb{B}$, and C's corresponding decompression scheme will convert $\mathbb{B}$ into a decompressed point cloud $\mathbb{PC}'$; (2) there exists a one-to-one mapping M from $\mathbb{PC}$ to $\mathbb{PC}'$; and (3) for each $p \in \mathbb{PC}$, the error between the coordinates of $M(p)$ and p on each dimension c is less than or equal to q_c.

Octree Representation. An octree representation [36] of a 3D point cloud can be constructed by recursive partitioning. Without loss of generality, we let the space represented by each tree node be a cube. First, a bounding cube covering all points is created, serving as the root node of the octree. Then, the cube is partitioned into eight sub-cubes by dividing each dimension into two halves. These eight sub-cubes are the child nodes of the root node, and further partitioning can continue on non-empty cubes until a given tree depth is reached, or each leaf node contains only a single point.

In a point cloud compression scheme using an octree representation, each point in a leaf node is approximated to the cube's center. Therefore, if the side length of the leaf node is less than or equal to $2q_c$ on each dimension c, the error for a point before compression and after decompression on each dimension will be less than or equal to q_c.

Delta Encoding. Delta encoding works on a sequence of values, defined as follows.

Definition 2.3. Given a sequence of values $\mathcal{L} = \langle v_1, v_2, \cdots, v_n \rangle$, delta coding transforms $\mathcal{L}$ into $\Delta\mathcal{L} = \langle v_1, \Delta v_2, \cdots, \Delta v_n \rangle$, where $\Delta v_m = v_m - v_{m-1}$ for $1 < m \leq n$.

Entropy. Given a sequence of n values $\mathcal{L}$, suppose $\mathcal{L}$ contains m distinct values $v_1, v_2, \cdots, v_m$. The entropy of $\mathcal{L}$ is defined as

$$H(\mathcal{L}) = -\sum_{i=1}^{m} P(v_i) \log P(v_i) \tag{1}$$

where $P(v)$ is the frequency of the occurrence of v in $\mathcal{L}$ [50]. Entropy coding is a group of compression methods that compress $\mathcal{L}$ into $\mathcal{B}$ such that each distinct value in $\mathcal{L}$ is represented by a symbol with $H(\mathcal{L})$ bits on average in $\mathcal{B}$.

2.2 Related Work

Object Point Cloud Compression. Traditionally, a point cloud illustrates the surface of an object. Botsch et al. [7] converted a point cloud into an octree and then encoded the octree. Specifically, they represented each non-leaf node of the octree as an *occupancy code*, i.e., a bitmap of length eight, where the nth bit was 0 (resp., 1) if its nth child node was empty (resp. non-empty). After that, the octree was serialized in the breadth-first order into a sequence of occupancy codes. Then, they utilized an arithmetic coder [58] to compress the sequence. Due to the impressive compression performance of the octree, several improvements were proposed to further reduce the redundancy [28, 44, 48]. The latest improvement proposed by Garcia et al. [21] grouped octree nodes by the occupancy code of their parent nodes. Then, they compressed each group separately to improve the overall compression ratio. An alternative representation of a point cloud is a kd-tree [3], partitioning the space into two halves in each step. The point cloud compression program Draco [23] uses the kd-tree.

In comparison, other schemes compress a point cloud without any alternative representation. Gumhold et al. [25] and Merry et al. [37] connected all points into a spanning tree and determined a point's coordinates based on those of its ancestors. Ochotta et al. [40] and Fan et al. [17] partitioned a point cloud into clusters and compressed each cluster by representing it with a fitting function.

As the baseline octree coder [7] is already sufficient for compressing dense point clouds, and further improvements have limited benefit, we implement the baseline octree coder as a building block in our scheme to compress dense points in a LiDAR point cloud.

Scene Point Cloud Compression. 3D scanning sensors represented by LiDAR are widely used for large-scale scenes. Directly adopting compression methods for object point clouds to compress scene point clouds results in a poor performance, because the latter are much sparser than the former. Therefore, some work optimized octree compression methods for scene point clouds. Specifically, G-PCC [33], proposed by the Moving Picture Experts Group (MPEG), represented a point cloud by an octree, and adopted several optimizations such as direct point coding, neighbor-dependent entropy coding, and triangle construction to achieve a better compression performance. Huang et al. [27] and Que et al. [45] designed deep learning models to reduce the space cost of an octree. These methods show limited improvement over the original octree-based methods on LiDAR point clouds.

LiDAR sensors can output raw point clouds where the spherical coordinates of the points form an image. Taking advantage of this feature, Houshinar et al. [26] mapped each pixel in the image to the RGB color space and then applied existing image compression methods. Similarly, Tu et al. [54] treated the data as a greyscale image for compression. Ahn et al. [1] partitioned the image into blocks and applied various compression strategies for the blocks. Zanuttigh et al. [60] and Sun et al. [53] performed segmentation on the image and modeled the pixel value distribution of each segment. However, applications usually use calibrated point clouds rather than raw data, to reduce errors and noise.

In addition to approaches compressing static point clouds, some research focused on reducing the temporal redundancy of a dynamic point cloud stream [5, 55].

In comparison to existing work on point cloud compression, our scheme works for both raw and released point clouds. Furthermore, we distinguish between dense and sparse points, and organize dense points into an octree and sparse points into polylines.

Compression in Databases. There is an abundance of compression schemes in databases, especially column-oriented databases. Fang et al. [18] studied nine lightweight compression methods, including delta encoding, run-length encoding, data scaling, dictionary encoding, and others, with their combinations in databases on GPUs. Additionally, Damme et al. [12] proposed an experimental survey to investigate several compression schemes and their SIMD extensions. Gorilla et al. [43] performed the XOR operation on consecutive values and then compressed the leading and tailing zeros of the XOR results. Sprintz et al. [6] stored the difference between actual values and predicted values and compressed the errors with bit-packing encoding. Liu et al. [32] eliminated the least significant bits of a floating-point number and compressed the integer and decimal components of each number separately.

The coordinates of points can be stored in a relational table in the database, where each tuple corresponds to a point, and the coordinates of points are attributes. Therefore, lightweight database compression schemes can also be applied to domain-specific systems that store points and trajectories. For instance, TrajStore [11] utilizes delta encoding to reduce the space consumption of a trajectory data store. Trajic [39] improves TrajStore by utilizing timestamp information and current positions in predicting future positions. TRACE [30] performs online compression on network-constrained trajectories. In our compression scheme, we optimize delta encoding based on the properties of real-world LiDAR data and adopt several other techniques to achieve compression effectiveness.

General-purpose Compressors. General-purpose compressors take input as a byte sequence. For instance, Huffman coding [29] and arithmetic coding [58] reduce the redundancy based on entropy. LZ77 [61] and LZMA [47] encode frequent subsequences in the input sequence to decrease the space cost. Deflate [13], integrated into the software Gzip [14] and Zlib [34], performs Huffman coding on the output of LZ77. Bzip2 [49] compresses a sequence based on Burrows-Wheeler transform.

The state-of-the-art compressors are Zstd [16] and Snappy [24], focusing on high throughput real-time compression. However, they are not employed in current point cloud compression methods yet, because the data size of a single-frame point cloud is small, and the compression throughput is not a concern. In contrast, many point cloud compression approaches adopt traditional general-purpose compressors as building blocks in their design for a high compression ratio. For instance, after converting a point cloud into an octree and serializing the octree, Botsch et al. [7] utilizes an arithmetic coder to compress the occupancy code sequence. Sun et al. [53] put all pixels in a residual image into a sequence and then compressed the sequence by Deflate and Bzip2. We adopt Deflate and an arithmetic coder in our compression scheme.

3 THE DBGC SYSTEM

3.1 System Overview

We design our point-cloud compression scheme with an error bound, i.e., the points in $\mathbb{PC}$ and those in $\mathbb{PC}'$ have a one-to-one mapping, and the coordinate differences of each pair of points in the mapping meet the error bound. In practice, this error bound is typically the measurement accuracy of LiDAR sensors, for example, 0.02 meters. Our scheme can be utilized as a standalone compression tool in various applications. Additionally, we also integrate the scheme into our prototype system DBGC. DBGC targets at a common class of applications that acquire and transmit LiDAR data in resource-constrained environments and have online processing or storage requirement, for example, remote surveys and online monitoring. In the following, we describe our scheme in DBGC.

Figure 2 presents the architecture of the DBGC system. First, the LiDAR sensor captures the geometry of its surroundings and stores the points as a point cloud $\mathbb{PC}$. Next, the point cloud is transferred from the sensor to the memory of the host computer (*client* in our system) through a wired network. The client performs compression on the point cloud data in memory and generates a bit sequence $\mathcal{B}$ whose data size is smaller than $\mathbb{PC}$. After that, the bit sequence is transferred to the memory of a server (*server* in our system) through a mobile network. The server decompresses the bit sequence into the decompressed point cloud $\mathbb{PC}'$, and processes it or stores it. Alternatively, the server may bypass the decompression procedure and directly store $\mathcal{B}$.

On the client, there are six components: density-based clustering, coordinate conversion, point organization, octree compression, coordinate compression, and outlier compression. The choices of these different compression schemes are based on the characteristics of the subsets of the point cloud, and together they achieve the best overall compression ratio. Specifically, DBGC first performs density-based clustering on the input LiDAR point cloud $\mathbb{PC}$ with the given error bound q_{xyz}, and then compresses the dense points with an octree. Next, DBGC converts the Cartesian coordinates of the sparse points to the spherical coordinates, organizes these points into polylines, and conducts our proposed coordinate compression method. Third, if any sparse points are left out of all polylines, DBGC will convert these outliers back to the Cartesian coordinate system and compress them by an optimized compressor based on the quad-tree. Finally, DBGC puts together the three compressed subsets of the point cloud and generates the final $\mathcal{B}$.

After the compressed point cloud is sent to the server, DBGC separates $\mathcal{B}$ into three subsequences, and passes each to a corresponding decompressor, i.e., an octree decompressor, a coordinate decompressor, and an outlier decompressor. Since points in polylines are of spherical coordinates, they are converted back to the Cartesian coordinate system. Finally, $\mathbb{PC}'$, the entire decompressed point cloud in Cartesian coordinates, is constructed by putting together the three groups of points.

In the following sections, we present each component of the DBGC system in detail.

3.2 Density-based Clustering

Problem of Octree on LiDAR Point Clouds. Most point cloud compression approaches, including the octree coder, were originally proposed for object point clouds that are generated by 3D object scanning [46], where the spatial coordinates have a small range and points are densely distributed. In contrast, LiDAR point clouds are captured by sensors in large-scale scenes, commonly spanning thousands of cubic meters, which are much sparser than the object point clouds.

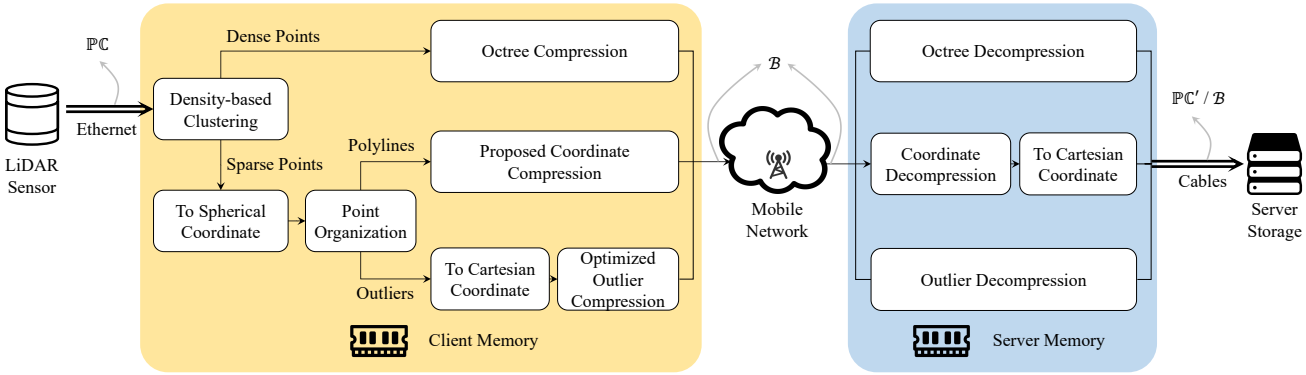
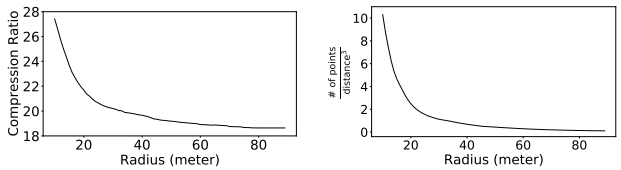


Figure 2: The DBGC system architecture.

To illustrate the impact of the sparsity on the effectiveness of an octree, we select subsets of the real-world LiDAR point cloud we visualized in Figure 1 to compress with an octree. These subsets are concentric spheres centered at the sensor, with radius (the maximum distance from the sensor to a point in the subset) varied. As shown in Figure 3a, the compression performance, measured in compression ratio, decreases as the radius of the point cloud increases. This trend correlates with the decrease of density in the number of points over radius cubed, shown in Figure 3b. In particular, when the radius of the point cloud is over 20 meters, the point density is 2 points per cubic meter, and the compression ratio is 22, both of which are much worse than those of point clouds of smaller spatial ranges. In comparison, object point clouds are much denser and of smaller spatial ranges than LiDAR point clouds. For instance, the *Stanford Bunny* dataset [51] has a density over 10^6 points per cubic meter.

Cell-based Clustering Approach. To make the best use of the octree's compression capability, in DBGC, we only apply it on a subset of the point cloud that is sufficiently dense, and design a more suitable algorithm for the remaining sparse points.

A simple method to identify dense regions is to select all points at distances to the sensor less than a given threshold, as Figure 3 suggests. This method works but is not flexible because (1) The spatial distribution of point clouds may vary significantly in different scenes; and (2) The distance threshold may differ by user requirement on the error bound. The DBSCAN [15] algorithm is widely utilized to cluster points based on the density in their local areas. It first determines the core points with at least $minPts$ neighbors within a radius of ϵ. Then, a core point and its neighbors form a cluster. If any neighbor is also a core point, the cluster expands. This recursive process goes on until no more expansion occurs. However, performing clustering in terms of points is time-consuming, taking $O(n^2)$ time in the worst case [19]. Furthermore, since the distance between two



(a) Compression ratio. **(b) Density.**

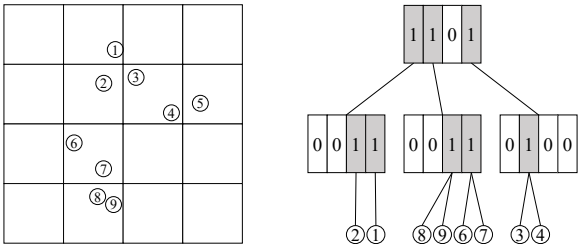
Figure 3: Varying radius.

points is measured in Euclidean distance, the local boundaries of the set of dense points are likely to have a sphere shape, which mismatches the structure of the octree cell represented as a cube. So a single octree cell may contain both the dense and sparse points. Recall that in our previous analysis, we only construct an octree cell if it contains a sufficient number of points. Therefore, as long as the number of dense points inside a cell meets our requirement, the cell can also include all the sparse points within the cube region to further improve the compression ratio of the octree.

As a result, we borrow the general idea of DBSCAN [15] and design a clustering method adapted to the property of octrees. Specifically, we propose to identify dense cells in an octree in addition to dense points in a point cloud. Given the bounding cube of a point cloud, we can determine the boundary of each cell. Initially, all cells are sparse cells. Then, in the clustering, we iterate over each point p in the point cloud and check if p is in a dense cell. If so, we directly set p as a dense point and process the neighbors of p, i.e., all points within a radius of ϵ around p, recursively. Otherwise, we perform the neighbor check operation by counting if p has at least $minPts$ neighbors. If p is determined to be dense, we record the cell it belongs to as a dense cell and process each neighbor of p recursively. In comparison, if p is not a dense point, the algorithm backtracks. Since a point may be processed before its cell is determined to be a dense cell, we need a second iteration to set those sparse points in dense cells to be dense points. This way, our cell-based clustering approach reduces the execution time of clustering compared with DBSCAN, because many costly neighbor check operations are pruned.

Parameters. The distance threshold ϵ determines the region size to compute the local density. If ϵ is small, the accuracy of estimated density may be low. However, a large ϵ leads to a high computational cost. In our approach, we set ϵ based on the user-given error bound. Specifically, $\epsilon = k \cdot q_{xyz}$, where k is a natural number. Since the side length of the octree leaf node is twice the error bound, the k value should be at least 2, so points whose distances are less than $2 \cdot q_{xyz}$ will appear in the cluster results. We have experimented with k values ranging from 2 to 100 and set the final k value to 10, since 10 is sufficiently large for the point cloud data, and the clustering time is short.

$minPts$ specifies the lowest density of a cluster. If $minPts$ is small, many points from the sparse regions are included. In comparison, if $minPts$ is large, only the points with high density are compressed with the octree, limiting the compression performance. As each non-empty leaf node of an octree contains at least



(a) Example points. **(b) Example quadtree.**

Figure 4: Example of density-based clustering.

one point, the minimum number of neighbors of a core point can be set as the number of non-empty leaf nodes in the sphere with the core point as the center and the radius ϵ. This sphere has a volume of $\frac{4}{3}\pi\left(k \cdot q_{xyz}\right)^3$. Since the side length of a leaf node in the octree is $2 \cdot q_{xyz}$, the sphere contains $\frac{4}{3}\pi\left(k \cdot q_{xyz}\right)^3 / \left(2 \cdot q_{xyz}\right)^3 = \frac{k^3}{6}\pi$ leaf nodes. Therefore, we can set $minPts$ to $\frac{k^3}{6}\pi$ so that the sphere around a core point is covered by a sufficient number of non-empty leaf nodes.

After performing density-based clustering with these two parameters on a point cloud, DBGC identifies all points in any cluster as dense points and the remaining points as sparse ones. In our running example of the point cloud demonstrated in Figure 1, about 40% of the point cloud are dense points and 60% are sparse points.

Octree Compression Our clustering method may identify multiple clusters, each of which has a density higher than the given threshold. It is straightforward to build an octree for each cluster. However, these octrees may overlap if data points covered in different trees have overlapping bounding cubes, leading to redundancy. Therefore, DBGC uses a single octree to represent all dense points. Even though this single octree may cover some empty space between clusters, the empty nodes are not further subdivided in the tree. The output of the octree compressor is denoted as $\mathcal{B}_{dense}$.

Example 3.1. Figure 4a shows the example points labeled with point IDs and potential quadtree cells in the 2D space. Suppose we identify two point clusters $\{1, 2, 3\}$ and $\{7, 8, 9\}$. Our cell-based clustering will not only mark these six points but also points 4 and 6 as dense points because point 4 is in the same cell as point 3 and 6 in the same cell as point 7. Figure 4b shows the quadtree built on the clustering result. In the top two levels, each cell is represented as a bit sequence. The bottom level shows the points represented by the quadtree, where all dense points are included.

3.3 Coordinate Conversion

LiDAR point clouds are hard to compress due to their sparsity, so some previous methods [1, 54, 60] exploited the characteristics of LiDAR data and convert each point from the Cartesian coordinates (x, y, z) to the spherical coordinates (θ, ϕ, r), where θ and ϕ are the *azimuthal* and *polar angles* of the vector from the sensor to the point, respectively, and r is the *radial distance* from the sensor to the point. In the raw data, the azimuthal and polar angles form a regular 2D grid, where each cell is attached with an r value. Such a format is essentially an image and related methods thus compress the image instead of the points.

However, most applications take calibrated point clouds rather than raw ones as their input data to reduce the noise and errors from the LiDAR sensor. We plot our running example (calibrated) point cloud in a 2D space with the azimuthal angle on the horizontal dimension and the polar angle on the vertical dimension in Figure 5. The solid boundary of the top rectangle represents the bounding box defined by the sensor. We further zoom into three particular regions enclosed by the dashed rectangles. As shown in the figure, some points are out of the bounding box. For example, some points in the dashed rectangle on the bottom left are out of the top boundary, and some cells are missing points. In short, the points do not exhibit a grid but overall are positioned with regularity.

Consequently, DBGC does not map the points into an image, but only converts the coordinates to take advantage of the regularity as well as ensure the compression to be a one-to-one mapping from the original points. Based on these considerations, we design a new point compression scheme in the spherical coordinate system.

In the remainder of this paper, we use the following notations to denote variables related to coordinates. Specifically, the metadata of a LiDAR sensor provides the minimum and maximum values of the spherical coordinates, namely $\widehat{\theta_{max}}$, $\widehat{\theta_{min}}$, $\widehat{\phi_{max}}$, $\widehat{\phi_{min}}$, $\widehat{r_{max}}$, and $\widehat{r_{min}}$. The number of samples in the horizontal and vertical directions are also recorded, denoted as H and W, respectively. Therefore, we can get average difference between two adjacent points in the azimuthal angle, $u_\theta = \frac{\widehat{\theta_{max}} - \widehat{\theta_{min}}}{W}$, and that in the polar angle, $u_\phi = \frac{\widehat{\phi_{max}} - \widehat{\phi_{min}}}{H}$.

3.4 Point Organization

In the point organization step, we first identify a sequential order among the sparse points. This total order is necessary because (1) Points are unordered in the point cloud, but our proposed compression scheme works on an ordered sequence of points; and (2) A good point order can boost the compression effectiveness. Our observation in the previous section indicates that the coordinate differences between two points near to each other are similar in the spherical coordinate space. Therefore, we adopt a delta-encoding compression approach, i.e., processing on the coordinate differences of two points rather than the original coordinates of a point, on sparse points in the following step. As a result, in the output of point organization, the coordinate difference between two consecutive points in the sequence is small, so the data entropy is low.

To fulfill our goal, we propose to organize points with similar polar angles into horizontal polylines. Some polylines are marked in Figure 5. To extract a polyline, we first choose a point, and then extend it by adding vertices based on distance and ϕ angle similarity. The details are illustrated in Algorithm 1. The input is a point cloud $\mathbb{PC}$ in the spherical coordinate system. Line 1 initializes an empty polyline set $\mathbb{PL}$. Then, Lines 2-8 iterate over each remaining point in $\mathbb{PC}$ to generate a polyline. As Line 3 selects a point p, we set the minimum and maximum polar

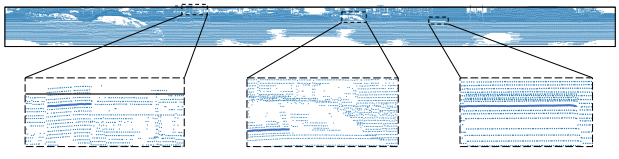


Figure 5: Point cloud in (θ, ϕ) space.

Algorithm 1: OrganizeSparsePoints

Input: $\mathbb{PC}$: a point cloud
Output: $\langle \mathbb{PL} \rangle$: a sorted sequence of polylines, and *Outliers*

1 $\mathbb{PL} \leftarrow \{\}$;
2 **foreach** $p \in \mathbb{PC}$ **do**
3 $l \leftarrow \langle p \rangle$;
/* extend a polyline to the right */
4 **while** $p' \leftarrow$ Extend$(l, \phi_p - u_\phi, \phi_p + u_\phi)$ *exists* **do**
5 move p' from $\mathbb{PC}$ to the back of l;
/* extend a polyline to the left, omitted */
6 $\mathbb{PL} \leftarrow \mathbb{PL} \cup \{l\}$;
7 $\langle \mathbb{PL} \rangle \leftarrow$ sort $\mathbb{PL}$;
8 **return** $\langle \mathbb{PL} \rangle$ and *Outliers*;
9 **Function** Extend$(l, \phi^l_{min}, \phi^l_{max})$:
10 p = tail of l;
11 $C_{p'} = \{p' \in \mathbb{PC} : \phi^l_{min} < \theta_{p'} < \phi^l_{max}$ *and* $0 < \theta_{p'} - \theta_p < 2u_\theta\}$;
12 **if** $C_{p'}$ *is not empty* **then**
13 **return** $p' \in C_{p'}$ with minimum $||p - p'||_2$;
14 **else**
15 **return** *null*;

angles of the points in the polyline to $\phi_p - u_\phi$ and $\phi_p + u_\phi$, where u_ϕ is the average difference between two consecutive points in the polar angle direction. This way, the polyline extends roughly horizontally. After that, Lines 4-5 extend a polyline to the right and Line 6 inserts the line to $\mathbb{PL}$. In the extension routine, Lines 10-11 generate a candidate point set $C_{p'}$ containing all p' whose polar angle is within the range from ϕ^l_{min} to ϕ^l_{max} and the azimuthal angle difference between p' and p is within the range from 0 to $2u_\theta$. Since u_θ is the average difference between two consecutive points in the azimuthal angle direction, this setting allows adjacent sampling points, if present, to be included in the candidate set. Then, if $C_{p'}$ is not empty, Line 13 selects a point from $C_{p'}$ with the minimum Euclidean distance from p, denoted as $||p - p'||_2$. Otherwise, the current extension ends. Then, a polyline is also extended to the left. We omit the details since the extension routines in both directions are similar.

Finally, the resulting polylines are sorted on Line 7. Specifically, we first define the polar angle of a polyline as the polar angle of its first point since all polylines are extended horizontally. Then, we sort all polylines in $\mathbb{PL}$ by the ascending order of their polar angles. If two polylines have the same polar angle, we place the one whose head has a smaller azimuthal angle in front of the other. Line 8 returns the resulting sequence of polylines, denoted as $\langle \mathbb{PL} \rangle$, and the *Outliers*, i.e., the set of points not belonging to any polyline.

3.5 Coordinate Compression of Sparse Points

After organizing the sparse points, we design a coordinate compression approach for them by considering the redundancy within each polyline as well as across polylines. The outline is shown in Figure 6. A rounded rectangle represents a step, where the back circled number at the top-left corner represents the step number. The gray arrows indicate data dependencies. Namely, an arrow from data $\mathcal{D}_1$ to data $\mathcal{D}_2$ means that $\mathcal{D}_2$ is generated based on $\mathcal{D}_1$. Next, we describe each step in detail.

Step 1: Coordinate Scaling. LiDAR sensors record the coordinates of a point with a precision much higher than required by the user. As a result, most applications set an error bound and allow the coordinates to be stored with a difference lower than the error bound from the original values. Also, high precision numbers are likely to have high entropy because they possibly differ on each digit. Therefore, in the first step of our compression, we lower the precision of the coordinate values. Specifically, we first compute the error bound on each dimension in the spherical coordinate system given q_{xyz}. For the azimuthal and polar angles, given a fixed error on a dimension, the farther a point is from the sensor, the larger the error on the position will be. So we set q_θ and q_ϕ to be $\frac{q_{xyz}}{r_{max}}$ to ensure that even the farthest point has a position error not greater than q_{xyz} in the θ and ϕ direction. Also, we set q_r to be q_{xyz} so that the position error on the r direction is within q_{xyz}. Lemma 3.2 states the correctness of our choices of q_θ, q_ϕ, and q_r.

Lemma 3.2. *Given a point $p \in PC$, the maximum position error of p in the spherical coordinate system with $q_\theta = \frac{q_{xyz}}{r_{max}}$, $q_\phi = \frac{q_{xyz}}{r_{max}}$, and $q_r = q_{xyz}$, is not greater than the maximum position error of p in the Cartesian coordinate system with $q_x = q_y = q_z = q_{xyz}$.*

Proof. In both the spherical and Cartesian coordinate systems, a point has the maximum Euclidean error if it has a maximum error on all dimensions simultaneously. Then, given $q_\theta = \frac{q_{xyz}}{r_{max}}$, $q_\phi = \frac{q_{xyz}}{r_{max}}$, and $q_r = q_{xyz}$, the square of the maximum error of a point (θ_p, ϕ_p, r_p) in the spherical coordinate system is

$$\begin{aligned} &[(r_p + q_r)sin(\theta_p + q_\theta)cos(\phi_p + q_\phi) - r_p sin\theta_p cos\phi_p]^2 \\ &+ [(r_p + q_r)sin(\theta_p + q_\theta)sin(\phi_p + q_\phi) - r_p sin\theta_p sin\phi_p]^2 \\ &+ [(r_p + q_r)cos(\theta_p + q_\theta) - r_p cos\theta_p]^2 \\ =\ & q^2_{xyz}[sin\theta_p cos\phi_p + cos(\theta_p + \phi_p)]^2 + q^2_{xyz}[sin\theta_p sin\phi_p \\ &+ sin(\theta_p + \phi_p)]^2 + q^2_{xyz}(cos\theta_p - sin\theta_p)^2 \\ =\ & (2 + sin^2\theta)q^2_{xyz} \end{aligned} \tag{2}$$

In the Cartesian coordinate, given $q_x = q_y = q_z = q_{xyz}$, the square of the maximum Euclidean error of a point (x_p, y_p, z_p) is

$$q_x \cdot q_x + q_y \cdot q_y + q_z \cdot q_z = 3q^2_{xyz} \tag{3}$$

Therefore, given our choices of q_θ, q_ϕ, and q_r, the maximum Euclidean error in the spherical coordinate system is not greater than that in the Cartesian coordinate system. □

Observing that the precision below the given error bound is needless, DBGC performs the data scaling and rounding on the spherical coordinates. Specifically, it divides the coordinate values on each dimension c by the corresponding scaling factor

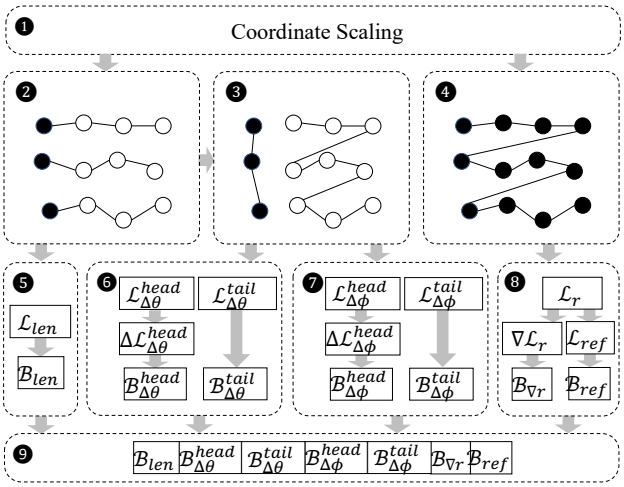


Figure 6: Procedure of coordinate compression.

$2q_c$, and then rounds the division result to an integer. The value of q_c on each dimension is carried with $\mathcal{B}$. In the decompression, all recovered coordinates are multiplied by $2q_c$. The rounding from a floating-point number to an integer introduces an error up to 0.5. Therefore, after the multiplication, the error is within $0.5 \cdot 2q_c = q_c$, namely the error bound.

Step 2: Delta Encoding on θ and ϕ. In this step, for each polyline l generated by Algorithm 1, we perform delta encoding on the θ and ϕ dimensions of the coordinates. After delta encoding, the first point of each polyline is represented as the *original coordinates*, whereas the remaining points are represented as the *delta coordinates*, i.e., the coordinate difference between the point and its preceding point. Compared with the original coordinates, the entropy of the delta coordinates is likely to be small for the following reasons. (1) Since LiDAR points are sampled uniformly from the sensor, delta coordinates on the θ dimension are distributed around u_θ; (2) Delta coordinates on the ϕ dimension are close to 0 because a polyline is extended horizontally.

Therefore, delta encoding provides opportunities to reduce the data size with entropy coding. In Figure 6, circles filled with white and black represent the original and delta coordinates, respectively. We denote the first point of a polyline by the *head*, and the sequence of remaining points by the *tail*.

Step 3: Data Reorganization. In the third step, the head and tail of each polyline are concatenated separately. DBGC performs this operation because the head of a polyline is stored as original coordinates, which have different data distributions from the delta coordinates of the remaining points.

Step 4: Polyline Concatenation. DBGC also directly concatenates all polylines and gets a single concatenated polyline.

Step 5: Compressing Lengths. Since in Steps 3 and 4, points in polylines are reorganized, the length of each polyline is recorded so that the decompressor can correctly recover each polyline from the bit sequence. Therefore, in this step, DBGC stores the length of each polyline into $\mathcal{L}_{len}$ and then compresses it into $\mathcal{B}_{len}$ with arithmetic coding.

Step 6: Compressing θs. Polylines with similar polar angles are likely to contain points captured from the same set of objects in the scene. Therefore, these polylines may have similar patterns in the azimuthal angle. As such, DBGC performs compression on the azimuthal angles of all polylines.

Specifically, DBGC concatenates the coordinates of the heads of all lines on the θ dimension into $\mathcal{L}_{\Delta\theta}^{head}$ and those of the tails into $\mathcal{L}_{\Delta\theta}^{tail}$. After that, DBGC transforms $\mathcal{L}_{\Delta\theta}^{head}$ into $\Delta\mathcal{L}_{\Delta\theta}^{head}$ by delta encoding. Finally, both $\Delta\mathcal{L}_{\Delta\theta}^{head}$ and $\mathcal{L}_{\Delta\theta}^{head}$ are compressed by Deflate [13] into $\mathcal{B}_{\Delta\theta}^{head}$ and $\mathcal{B}_{\Delta\theta}^{tail}$. Deflate performs both LZ77 compression [52] and entropy coding. We adopt Deflate as opposed to simple entropy encoder because it can compress the two sequences more effectively as there are many repeated patterns.

Step 7: Compressing ϕs. Similar to compressing azimuthal angles across all lines, DBGC generates $\mathcal{L}_{\Delta\phi}^{head}$ and $\mathcal{L}_{\Delta\phi}^{tail}$, converts $\mathcal{L}_{\Delta\phi}^{head}$ into $\Delta\mathcal{L}_{\Delta\phi}^{head}$ by delta encoding, and compresses $\Delta\mathcal{L}_{\Delta\phi}^{head}$ and $\mathcal{L}_{\Delta\phi}^{tail}$ by arithmetic coding. We use arithmetic coder instead of Deflate because the redundancy in polar angles is not as much as in azimuthal angles, and therefore, the former is sufficient.

Step 8: Compressing r. Different from θ and ϕ, the delta coordinates on r of some consecutive points on a polyline may be relatively large due to the complexity of real-world scenes. Therefore, rather than performing delta encoding on the r dimension,

Algorithm 2: GeneratingConsensusPolyline

```
Input: 𝕃*_l: a sorted sequence of reference polylines
Output: l*: a consensus reference polyline
1 foreach l' ∈ 𝕃*_l do
2     if l* is empty or θ_{l*.back} < θ_{l'.front} then
3         insert l' to the end of l*;
4     else
5         idx_left = max{i : θ_{l*[i]} > θ_{l'.front}};
6         idx_right = min{i : φ_{l*[i]} < φ_{l'.back}};
7         erase l* from idx_left to idx_right;
8         insert l' to l* at idx_left;
9 return l_con;
```

we propose our *radial distance optimized delta encoding* method, described in Definition 3.3, to further reduce the redundancy among all values in $\mathcal{L}_r$.

Definition 3.3 (Radial Distance Optimized Delta Encoding). Given a radial distance sequence $\mathcal{L}_r = \langle r_{p_1}, r_{p_2}, \cdots, r_{p_n} \rangle$, radial distance optimized delta encoding transforms $\mathcal{L}_r$ into $\nabla\mathcal{L}_r = \langle r_{p_1}, \nabla r_{p_2}, \cdots, \nabla r_{p_n} \rangle$, where $\nabla r_{p_m} = r_{p_m} - r_{Ref(p_m)}$ for $1 < m \leq n$, and $Ref(p)$ returns the *reference point* of p.

Delta encoding is a special case of Definition 3.3, where $Ref(p_m) = p_{m-1}$, i.e., p_m's preceding point in the same polyline. However, in Definition 3.3, $Ref(p)$ can be chosen from all points in the polyline and also points from other polylines. Next, we define the reference polyline set.

Definition 3.4 (Reference Polyline Set). Given $\langle \mathbb{PL} \rangle$, $l \in \mathbb{PL}$, and a polar angle threshold TH_ϕ, the reference polyline set $\mathbb{PL}_l^*$ of l contains all polylines l', such that (1) l' precedes l in $\langle \mathbb{PL} \rangle$, and (2) the difference between the polar angle of l' and that of l is less than or equal to TH_ϕ.

TH_ϕ controls the range for finding reference polylines. We set TH_ϕ to $2u_\phi$ so that most polylines have a non-empty reference polyline set. We denote the sequence of reference polylines as $\langle \mathbb{PL}_l^* \rangle$. Polylines in $\langle \mathbb{PL}_l^* \rangle$ have the same order as in $\langle \mathbb{PL} \rangle$.

Reference polylines of l are vertically close to l. Therefore, due to the spatial locality, the rs of points in l are similar to those in $l' \in \mathbb{PL}_l^*$. To mitigate the complexity of handling multiple polylines in the encoding, we summarize all the $l' \in \mathbb{PL}_l^*$ and generate a *consensus reference polyline*, denoted as l^*, for each l. The generation procedure is illustrated in Algorithm 2. Lines 1-8 loop over each polyline $l' \in \mathbb{PL}_l^*$. Specifically, Line 2 checks if the current l^* is empty or the azimuthal angle of the tail of l^* is less than that of the head of l'. If so, the current polyline l' is directly inserted into the end of l^*. Otherwise, Line 5 finds the index of the leftmost point of l^* whose azimuthal angle is greater than that of the head of l', idx_{left}. In contrast, Line 6 finds the index of the rightmost point of l^* whose azimuthal angle is less than that of the tail of l', idx_{right}. After that, Lines 7-8 replace the points between idx_{left} and idx_{right} in l^* with l'.

With the assistance of radial distance optimized delta encoding and l^*s, DBGC performs compression on $\mathcal{L}_r$. Elements in $\mathcal{L}_r$ are processed one after another. Given r_p, suppose p resides on l. DBGC first checks if p is the head of l.

(1) If so, an l^* is created for l, and $Ref(p)$ is the rightmost point $p' \in l^*$ whose azimuthal angle is less than θ_p. If l^* does not exist or such a p' does not exist in l^*, $Ref(p)$ is set to be the head of the preceding polyline of l in $\langle \mathbb{PL}_l^* \rangle$.
(2) Otherwise, we denote the preceding point of p in l as p_{bl}, i.e., the bottom-left point. Also, in l^*, we denote the rightmost

point whose azimuthal angle is less than $\theta_{p_{bl}}$ as p_{ul}, i.e., the upper-left point, and the leftmost point whose azimuthal angle is greater than θ_p as p_{ur}, i.e., the upper-right point. Additionally, if there is a point next to p_{ur} on the left, which is not p_{ul}, it is denoted as p_{um}, i.e., the upper-middle point. If l^* does not exist or p_{ul} or p_{ur} does not exist in l^*, $Ref(p)$ is set to p_{bl}. Otherwise, there are two situations:

(a) The difference between each pair of points among $r_{p_{ul}}$, $r_{p_{ur}}$, and $r_{p_{bl}}$ is less than or equal to a radial distance threshold TH_r, then the local scene near p is likely to be flat and DBGC directly sets $Ref(p)$ to p_{bl}. We will describe the choice of TH_r later in this section.

(b) Otherwise, $Ref(p)$ is set to be one of the point from p_{ul}, p_{um} (if any), p_{ur}, and p_{bl}, whose r is the nearest to r_p.

In situation (1) and (2)(a), the choice of $Ref(p)$ does not need recording, because as long as the θ and ϕ of all points are decompressed, the choice of $Ref(p)$ can be reproduced by the decompressor based on the same procedure in the compression. However, in situation (2)(b), the choice of $Ref(p)$ should be recorded since the decompressor can not determine which point from p_{ul}, $r_{p_{um}}$ (if any), p_{ur}, and p_{bl} has the r nearest to r_p. As a result, DBGC maintains another sequence $\mathcal{L}_{ref}$. p_{ul}, p_{um} (if any), p_{ur}, and p_{bl} are represented by symbols 3, 2, 1, and 0, respectively. As long as the condition (2)(b) is applied in the compression phase, DBGC pushes the symbol of its choice to $\mathcal{L}_{ref}$. In this way, to decompress the raidal distance of p, DBGC checks if the difference between any pair of points among $r_{p_{ul}}$, $r_{p_{ur}}$, and $r_{p_{bl}}$ is greater than TH_r. If so, DBGC reads an element from $\mathcal{L}_{ref}$ to get the reference point of p and then computes r_p.

In the compression scheme, a greater TH_r leads to a higher entropy of $\nabla\mathcal{L}_r$ but a shorter $\mathcal{L}_{ref}$, since a high threshold is unlikely to reach and Situation (2)(b) is encountered less often than other conditions. By contrast, a smaller TH_r leads to a lower entropy of $\nabla\mathcal{L}_r$ but a longer $\mathcal{L}_{ref}$. We set it to be $2m$ in our scheme because consecutive points with a radial distance difference of more than $2m$ are likely to appear at the boundary of an object.

Example 3.5 shows a concrete example of generating $\nabla\mathcal{L}_r$ and $\mathcal{L}_{ref}$ based on $\mathcal{L}_r$. After that, DBGC performs arithmetic coding on both $\nabla\mathcal{L}_r$ and $\mathcal{L}_{ref}$ to get $\mathcal{B}_{\nabla r}$ and $\mathcal{B}_{ref}$.

Step 9: Organizing Output. Finally, DBGC combines all the compressed sequences generated by our scheme together, such as $\mathcal{B}_{len}$, $\mathcal{B}^{head}_{\Delta\theta}$, $\mathcal{B}^{tail}_{\Delta\theta}$, $\mathcal{B}^{head}_{\Delta\phi}$, $\mathcal{B}^{tail}_{\Delta\phi}$, $\mathcal{B}_{\nabla r}$, and $\mathcal{B}_{ref}$, to produce the final bit sequence for the coordinate compression phase, denoted as $\mathcal{B}_{sparse}$.

Example 3.5. Figure 7 shows the example points and their radial distances. p_5 is the head of polyline l, and a part of l^* is shown as the upper polyline. Since p_5 is the first point of l, Situation (1) is applied. $Ref(p_5)$ is set to be p_1, so DBGC pushes $r_{p_5} - r_{p_1} = -1.4m$ to $\nabla\mathcal{L}_r$. Next, r_{p_6} is encoded, where p_{ul}, p_{ur}, and p_{bl} are p_1, p_2, and p_5, respectively. Since the difference between p_{ur} and p_{bl} is greater than $2m$, Situation (2)(b) is applied. DBGC finds a point from p_{ul}, p_{ur}, and p_{bl} with the r nearest to r_{p_6}, which is $p_{ul} = p_1$. As a result, DBGC pushes $r_{p_6} - r_{p_1} = 0.2m$ to $\nabla\mathcal{L}_r$ and 3 to $\mathcal{L}_{ref}$. Then, for p_7, p_{ul}, p_{um}, p_{ur}, and p_{bl} are p_1, p_2, p_3, and p_6, respectively. Among these four points, no two points have the difference on r greater than $2m$, so Situation (2)(a) is applied, and $r_{p_7} - r_{p_6} = -1.2m$ is pushed into $\nabla\mathcal{L}_r$. For p_8, according to Situation (2)(b), p_4 is selected as the reference point. Therefore, DBGC pushes $r_{p_8} - r_{p_4} = 0.8m$ to $\nabla\mathcal{L}_r$ and 1 to $\mathcal{L}_{ref}$.

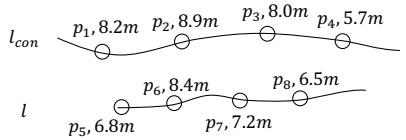


Figure 7: Example points with radial distances.

Point Grouping. The data scaling and rounding techniques still keep precision higher than the accuracy. Specifically, q_θ and q_ϕ guard the position error of the farthest point, but the points near the sensor could have an error greater than them on the θ and ϕ directions. Such a small error bound restricts the effectiveness of the compressor. Therefore, after the density-based clustering, DBGC splits all the sparse points into several groups evenly by the radial distance and then processes each group separately to improve the performance. After that, the points not belonging to any polylines in any group are regarded as outliers. We find that a small number of groups can already achieve a high compression performance, so we set the number of groups to 3 in our experiments.

3.6 Optimized Outlier Compression

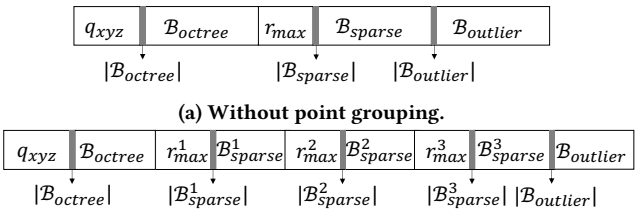
In the output of Algorithm 1, the set of outliers is small, and its impact on the overall compression performance is negligible. Therefore, we choose to compress the outliers based on existing methods on the Cartesian coordinates. Specifically, we build a 2D quad-tree on the x and y coordinates and keep z as an attribute of the point. This choice is because the z coordinate range is relatively small due to the inherent characteristics of LiDAR data, i.e., LiDAR sensors have a much smaller vertical scanning range than the horizontal one, and outliers are typically far points on the xoy plane. If we use an octree on (x, y, z), a lot of the space on the z dimension of the bounding cube will be wasted.

After encoding the x and y coordinates of the point cloud with the quad-tree compressor, DBGC records the z coordinate of all points into $\mathcal{L}_z$. Then it performs delta encoding on $\mathcal{L}_z$ to get $\Delta\mathcal{L}_z$ and utilizes entropy coding to compress $\Delta\mathcal{L}_z$ into $\mathcal{B}_{\Delta z}$. After that, DBGC moves $\mathcal{B}_{\Delta z}$ to the back of the result of quad-tree compression. The final bit sequence of outlier compression is denoted as $\mathcal{B}_{outlier}$.

3.7 Output Layout and Decompression

The layout of the bit sequence $\mathcal{B}$ without point grouping is shown in Figure 8a. DBGC records the error bound q_{xyz} and the maximum value on the r dimension among all sparse points participating in the coordinate compression. The two values determine the data scaling factor so that DBGC can recover the coordinates of the points in polylines correctly. $\mathcal{B}$ also contains the three bit sequences $\mathcal{B}_{dense}$, $\mathcal{B}_{sparse}$, and $\mathcal{B}_{outlier}$, output by the octree compressor, the coordinate compressor, and the outlier compressor, respectively. The grey block before each sequence records the length. The decompressor splits the three components based on their lengths and passes them to different blocks in Figure 2. Next, the octree decompressor gets the points from the octree representation. The coordinate decompressor splits each bit sequence, listed in Step 9 in Figure 5, and recovers the spherical coordinates of all the points in polylines. After that, these points are scaled by the scaling factor and converted to the Cartesian coordinates. Finally, the Cartesian coordinates of the outliers are decompressed through outlier decompression.

Additionally, Figure 8b shows the layout of DBGC with point grouping of 3 groups. In particular, r^n_{max} and $\mathcal{B}^n_{sparse}$ represent the maximum value of r and the bit sequence of coordinate compression in the nth group, respectively. The decompression procedure is similar to that without point grouping, except that multiple groups of polylines are decompressed separately.



(a) Without point grouping.

(b) With point grouping.

Figure 8: Final layout of $\mathcal{B}$ produced by DBGC.

4 EXPERIMENTS

This section evaluates the performance of our compression scheme and the DBGC system on real-world LiDAR point cloud datasets with the state of the art.

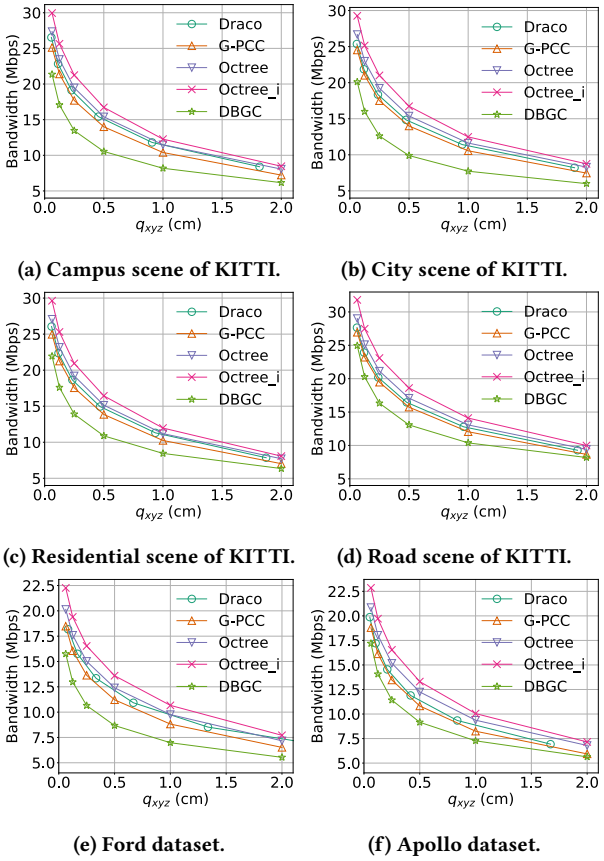
4.1 Experimental Setup

Implementation. Both the DBGC client and server are implemented in C++. DBGC has built-in Velodyne LiDAR sensor support, but users can easily apply DBGC on other types of sensors by importing the metadata of the sensor. DBGC client automatically pulls point clouds from the sensor and compresses the point clouds. We utilize the Linux socket model to transfer data from the client to the server. For the server storage, DBGC supports storing data into files or relational databases through ODBC. We also package compression and decompression functions of DBGC into libDBGC, so that existing point cloud systems can adopt DBGC's compression scheme as a building block with minor efforts.

We ran our experiments on a Linux machine with two Intel Xeon Gold 5218 CPUs and 512G RAM.

Datasets. We use three large-scale datasets in our evaluation captured from a variety of scenes spanning thousands of cubic meters. The first dataset KITTI [22] is widely used for evaluation in previous studies [27, 45, 53]. It contains four types of scenes – campus, city, residential areas, and road scenes. We select 1000 frames from each scene to conduct the experiments. Each frame contains approximately 100K points, stored in a binary file. Each point is represented by the x, y, and z coordinates and the intensity. The second dataset we use is Apollo [35], with the same file format as KITTI capturing urban scenes. We randomly select 1000 frames from this dataset, and each frame contains about 100K points. The third dataset we adopt is a Ford LiDAR dataset [42] capturing campus scenes. We also randomly select 1000 frames, each of which stores the coordinates of about 80K points in a *mat* file.

Methods under comparison. We compare the compression scheme of DBGC with the state-of-the-art point cloud compression schemes. Our main competitor is the Octree [36] as well as an improved version [21], denoted as Octree_i. The latter groups octree nodes by the occupancy code of their parent nodes and compresses each group separately. We re-implemented both methods according to the original papers. We also compare with Draco,



(a) Campus scene of KITTI. **(b) City scene of KITTI.**

(c) Residential scene of KITTI. **(d) Road scene of KITTI.**

(e) Ford dataset. **(f) Apollo dataset.**

Figure 9: Bandwidth vs q_{xyz} of five schemes.

a kd-tree-based compressor, whose code is publicly available [23]. Finally, we evaluate the TMC13 tool version 14 [38], which conforms to the MPEG G-PCC [33] standard for point cloud compression. We use the tool with the `mergeDuplicatedPoints` flag disabled so that the compressed point cloud has no missing points.

Metrics. We utilize compression ratio to indicate the effectiveness of a compression approach, which is the ratio of the data size of $\mathbb{PC}$ measured in bytes to $|\mathcal{B}|$. However, to evaluate our DBGC system in the data management context, we also report the average bandwidth requirement, measured in Mbps (Megabits per second), for transferring the compressed bit sequence in each compression scheme. Given $\mathbb{PC}$ and $\mathcal{B}$, if the number of frames generated per second in the dataset is f, then the bandwidth requirement is $8f|\mathcal{B}_i|$ bit per second. In particular, the factor 8 converts the length of $\mathcal{B}_i$ measured in bytes into that measured in bits.

Additionally, we use throughput and latency as performance metrics to evaluate the efficiency of the system. The throughput represents the number of point clouds compressed per second, while the latency measures the time passed between a point cloud is produced by the sensor and stored into the server storage.

4.2 Evaluation of Compression Capability

In this experiment, we evaluate the effectiveness of competing compression schemes on various scenes from different datasets. For each scene, we vary q_{xyz} from 0.06 to 2.0 centimeters to study the compression performance. Specifically, since all points

in an octree leaf node are approximated to the center of the cube represented by the node, we let the leaf side length be $2q_{xyz}$ in octree-based methods such as Octree, Octree_i, and G-PCC, to fulfill the compression requirement. However, in Draco, a user can only set the number of bits for quantizing the coordinates, qb, not q_{xyz}. Therefore, we vary qb to get the specified q_{xyz}. Specifically, given a point cloud with the maximum spatial range of all three dimensions in the Cartesian coordinate system Ω, $q_{xyz} = \frac{\Omega}{2^{qb+1}}$.

As shown in Figure 9, the compression scheme of DBGC outperforms all previous approaches on all datasets. Specifically, on the campus scene of the KITTI dataset, shown in Figure 9a, our scheme achieves an 18% bitrate reduction over G-PCC and a 25%-31% improvement over Octree, Octree_i, and Draco. Similar results are observed on the other scenes in KITTI, as well as on the other two datasets, as illustrated in Figure 9b, 9c, 9d, 9e, and 9f, respectively. This result clearly demonstrates the effectiveness of our density-based compression method in handling points with different local densities. In particular, the majority of the points are in sparse areas, and our sparse point compression scheme on the spherical coordinates yields a good compression ratio by taking advantage of the characteristics of both the LiDAR sensors and outdoor scenes. Given the typical error bound of LiDAR sensors, 2 cm, DBGC can provide an up to 19 compression ratio.

Among the four previous methods, G-PCC outperforms Octree, Octree_i, and Draco because its optimizations, such as direct point coding and neighbor-dependent entropy coding, make the method, to a certain extent, handle LiDAR point clouds well. Interestingly, Octree_i underperforms Octree in most instances, which is inconsistent with the results reported in the original paper [21]. One reason might be that the experiments in the paper were solely conducted on object point clouds, which are quite different from scene ones.

In summary, our method outperforms four representative point cloud compression methods on various datasets and requires the lowest bandwidth at the same accuracy requirement. A reduction of 18%-30% on the required bandwidth is significant for high accuracy requirements under 2cm on scene LiDAR point clouds.

4.3 Evaluation of Individual Techniques

In this section, we study the impact of each individual technique in the DBGC's compression method.

Density-based Clustering. In the result of the example point cloud, 39.4% of points are dense, and 60.6% are sparse. Only 1.2% of all points are outliers. To evaluate the impact of the clustering methods on the compression effectiveness, we manually vary the percentage of points nearest to the sensor from 0% to 100% to be compressed by the octree. Specifically, 0% means the algorithm performs our coordinate compression on the entire point cloud, whereas at 100% the point cloud is compressed by an octree solely. Figure 10 shows the compression performance with an error bound of 2cm. Our density-based clustering method shows the highest compression ratio among the entire spectrum of dense versus sparse categorization, with octree compression or coordinate compression on the entire point cloud on each end.

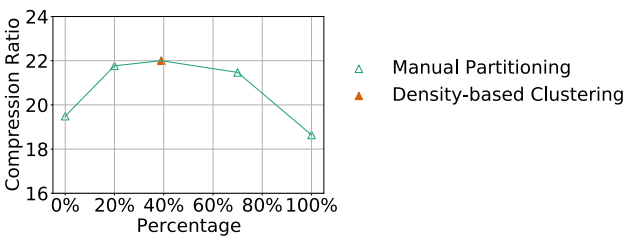


Figure 10: Percentage of points encoded in the octree varied.

Approximate Density-based Clustering. In Section 3.2, we have proposed a cell-based method to accelerate the clustering. However, the method has a complexity of $O(n^2)$. In this experiment, we adopt an approximate clustering approach [19], with a time complexity of $O(n)$, to further improve the efficiency. We first get all cells as in Section 3.2 and define the *surrounding cells* of a cell $\mathcal{N}$ as the set of all cells $\mathcal{N}'$ (including $\mathcal{N}$) where the coordinate difference between the central points of $\mathcal{N}$ and $\mathcal{N}'$ is not greater than ϵ on at least one dimension. We count the number of points in each cell and then loop over each non-empty cell. If the total number of points in all surrounding cells of $\mathcal{N}$ is not less than *minPts*, we mark $\mathcal{N}$ as dense. After that, we check all sparse cells - if a sparse cell has at least one dense cell as a surrounding cell, we mark that sparse cell a dense cell. Finally, all points in dense cells are marked as dense points and all other points are sparse ones.

The difference between the approximate method and the original one is the size and shape of the region to find neighbor points. Through our experiments, we find the sets of resulting dense points generated by the two algorithms are nearly the same. Furthermore, the approximate method is more time efficient, resulting in a 2.1X speedup over the original cell-based clustering method. After integrating the approximate clustering method into our compression scheme, we achieve a 1.2X speedup on the overall compression time compared with the original version.

Coordinate Compression. To evaluate the impact of spherical coordinate conversion, we present an alternative approach which extracts and compresses polylines in the Cartesian coordinate system, denoted as -Conversion. Additionally, we run two modified schemes by disabling the radial distance optimized delta encoding and point grouping techniques of our scheme, respectively, denoted as -Radial and -Group. We run these methods with the original DBGC on the campus scene of KITTI and present the compression ratios in Figure 11 with various error bounds. As we can see, -Radial, -Group, and -Conversion can only reach 88%, 85%, and 29% of the compression performance of DBGC on average.

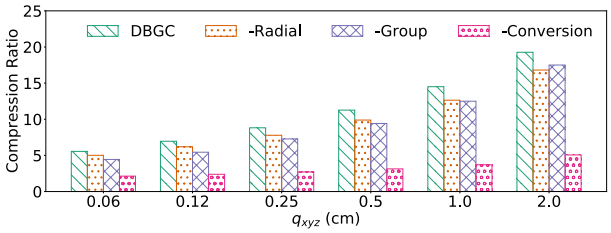


Figure 11: Improvement of optimizations in DBGC.

Optimized Outlier Compression. For the point clouds in the campus scene of KITTI, we develop two alternative approaches, one compresses the outliers by an octree and the other does not compress outliers, denoted as Octree and None, respectively. Then, we compare them with DBGC which performs outlier

Table 2: Compression ratios.

Scene	Campus	City	Residential	Road
Outlier	**19.32**	**18.80**	**18.78**	**15.29**
Octree	19.18	18.58	18.61	15.20
None	12.09	12.39	12.96	9.84

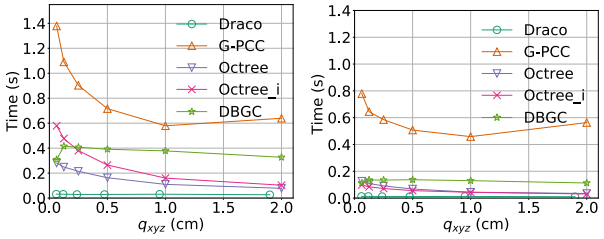
compression with a quadtree, denoted as Outlier. Given an error bound of 2 cm, the results in Table 2 show that, the outlier compression method of DBGC achieves slightly higher average compression ratios than the octree and both methods outperform None where outliers are not compressed.

4.4 End-to-end Evaluation

This section presents the end-to-end performance and evaluates the efficiency of the entire system. We take the KITTI dataset as an example since the total size of point clouds generated per second is the largest.

Throughput. KITTI dataset is captured by a Velodyne HDL-64E sensor [9]. In the default setting, the sensor generates 10 frames of point cloud per second, each with around 100,000 points. If each Cartesian coordinate is represented by a floating-point number, a point needs 32 bits × 3 = 96 bits to store. Then, a frame occupies approximately 9.6 Megabits, and the total size of point clouds generated per second is 96 Megabits. The most popular Ethernet 100BASE-TX [10], as well as the recently developed Gigabit Ethernet, have the bandwidth big enough to transfer the LiDAR data. Additionally, HDDs, commonly used in data centers, have a data write speed of more than 500 Megabits per second. Therefore, our system can nicely transfer $\mathbb{PC}'$, with the same size of $\mathbb{PC}$, from the memory to the storage of the server online. The bottleneck of the system is the connection from the client to the server. Specifically, the current 4G mobile network has an average upload speed of 8.2 Megabits per second [41], much lower than the data generation speed of the LiDAR sensor, and we present our compression scheme to fill in this gap. As shown in Figure 9b, at the error bound of 2cm, $\mathcal{B}$ needs a bandwidth of approximately 6.0 Mbps to transfer, lower than the typical 4G mobile network speed. Therefore, the bit sequences $\mathcal{B}$ can be transferred online from the client to the server. In conclusion, our DBGC system can achieve online compression and storage of point cloud data, with the throughput higher than the data generation rate of the LiDAR sensor.

Latency. Next, given a point cloud $\mathbb{PC}$, we evaluate the time spent on each step in our system. We first compare the compression and decompression time of all competing schemes given various error bounds and present the results on the city scene of KITTI in Figure 12a and 12b. DBGC, taking about 0.4 and 0.1 seconds on compression and decompression, respectively, is slower than Octree, Octree_i, and Draco, but faster than G-PCC. Generally, DBGC's compression time and decompression time slightly decreases as the error bound increases. However, when the error bound is very small ($q_{xyz} = 0.06cm$), DBGC's compression time is the shortest. This is because, when q_{xyz} is very small, there is hardly any repeated pattern in polylines, and Deflate can pass through the sequence quickly. Additionally, a frame of $\mathbb{PC}$ in the KITTI dataset is about 9.6 Megabits, taking approximately 0.1 seconds to transfer through 100BASE-TX Ethernet. Additionally, if q_{xyz} is 2 cm, $\mathcal{B}$ has a size of 0.6 Megabits, which can be transferred in 0.1 seconds. The time to write $\mathbb{PC}'$ takes negligible time.



(a) Compression time. (b) Decompression time.

Figure 12: Compression and decompression time.

To sum up, a point cloud spends 0.7 seconds totally before being generated from the sensor and stored into the server storage.

Detailed Evaluation. We further present the compression and decompression time breakdown of DBGC given $q_{xyz} = 2$ cm in Figure 13. The compression scheme contains six building blocks: (1) density-based clustering (DEN), (2) octree compression and decompression (OCT), (3) coordinate conversion (COR), (4) point organization (ORG), (5) coordinate compression and decompression of sparse points (SPA), and (6) outlier compression and decompression (OUT). In the compression, steps (2), (3), and (6) take negligible time, while steps (1), (4), and (5) take 31%, 22%, and 44% of the total compression time on average, respectively. By contrast, SPA dominates the total time of decompression.

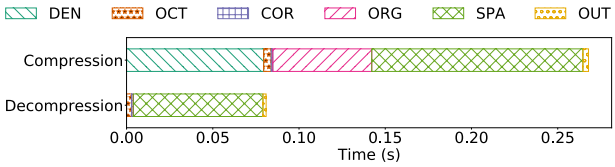


Figure 13: Time Breakdown.

We also measure the memory usage by reading the peak resident set size (VmHWM) of the process from the `proc` filesystem. The compression and decompression of DBGC consume around 45-Megabyte and 12-Megabyte space, respectively, negligible considering the configuration of current computers.

5 CONCLUSION

This paper proposed a density-based geometry compression system, DBGC, for LiDAR point clouds. In this system, we propose to classify points based on the cell-based clustering method with optimized parameter values and improved routines. Then, we pass the dense points to an octree coder and propose a coordinate compression method for the sparse points. Specifically, we present a point organization approach in the spherical coordinate space and then reduce the redundancy based on our radial distance optimized delta encoding method. Furthermore, we include several optimizations into the system to increase the compression performance. The experimental results show that at the same error bound, our method achieves a 20% space-reduction over the state of the art, which is a significant improvement in the topic of LiDAR point cloud compression according to previous research [5, 27]. Furthermore, we study the impact of each individual technique and the end-to-end performance of DBGC. The compression time of DBGC is comparable to existing methods, and the system can compress point clouds generated by the LiDAR sensor online with high accuracy.

REFERENCES

[1] Jae-Kyun Ahn, Kyu-Yul Lee, Jae-Young Sim, and Chang-Su Kim. 2014. Large-scale 3D point cloud compression using adaptive radial distance prediction in hybrid coordinate domains. *IEEE Journal of Selected Topics in Signal Processing* 9, 3 (2014), 422–434.

[2] Jens Behley, Martin Garbade, Andres Milioto, Jan Quenzel, Sven Behnke, Cyrill Stachniss, and Jurgen Gall. 2019. Semantickitti: A dataset for semantic scene understanding of lidar sequences. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 9297–9307.

[3] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[4] Vittorio Bichucher, Jeffrey M Walls, Paul Ozog, Katherine A Skinner, and Ryan M Eustice. 2015. Bathymetric factor graph SLAM with sparse point cloud alignment. In *OCEANS 2015-MTS/IEEE Washington*. IEEE, 1–7.

[5] Sourav Biswas, Jerry Liu, Kelvin Wong, Shenlong Wang, and Raquel Urtasun. 2020. MuSCLE: Multi Sweep Compression of LiDAR using Deep Entropy Models. *Advances in Neural Information Processing Systems* 33 (2020).

[6] Davis W. Blalock, Samuel Madden, and John V. Guttag. 2018. Sprintz: Time Series Compression for the Internet of Things. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2, 3 (2018), 93:1–93:23.

[7] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. 2002. Efficient high quality rendering of point sampled geometry. *Rendering Techniques* 2002 (2002), 13th.

[8] Hubo Cai and William Rasdorf. 2008. Modeling road centerlines and predicting lengths in 3-D using LIDAR point cloud and planimetric road centerline data. *Computer-Aided Civil and Infrastructure Engineering* 23, 3 (2008), 157–173.

[9] Usermanual contributors. 2022. *HDL 64E Manual_Rev D Manual.* https://usermanual.wiki/Pdf/HDL64E20Manual.196042455/html

[10] Wikipedia contributors. 2022. *Fast Ethernet.* https://en.wikipedia.org/wiki/Fast_Ethernet

[11] Philippe Cudré-Mauroux, Eugene Wu, and Samuel Madden. 2010. TrajStore: An adaptive storage system for very large trajectory data sets. In *Proceedings of the 26th International Conference on Data Engineering ICDE*. 109–120.

[12] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017*. 72–83.

[13] L. Peter Deutsch. 1996. *DEFLATE Compressed Data Format Specification.* https://datatracker.ietf.org/doc/html/rfc1951

[14] Peter Deutsch. 1996. Gzip file format specification version 4.3. *Technical Report* RFC 1952 (1996).

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *kdd*, Vol. 96. 226–231.

[16] Facebook. 2022. *zstd.* https://github.com/facebook/zstd

[17] Yuxue Fan, Yan Huang, and Jingliang Peng. 2013. Point cloud compression based on hierarchical point clustering. In *2013 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*. IEEE, 1–7.

[18] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database Compression on Graphics Processors. *Proc. VLDB Endow.* 3, 1 (2010), 670–680.

[19] Junhao Gan and Yufei Tao. 2015. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 519–530.

[20] Hongbo Gao, Bo Cheng, Jianqiang Wang, Keqiang Li, Jianhui Zhao, and Deyi Li. 2018. Object classification using CNN-based fusion of vision and LIDAR in autonomous vehicle environment. *IEEE Transactions on Industrial Informatics* 14, 9 (2018), 4224–4231.

[21] Diogo C Garcia and Ricardo L de Queiroz. 2018. Intra-frame context-based octree coding for point-cloud geometry. In *2018 25th IEEE International Conference on Image Processing (ICIP)*. IEEE, 1807–1811.

[22] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. 2013. Vision meets Robotics: The KITTI Dataset. *International Journal of Robotics Research (IJRR)* (2013).

[23] Google. 2021. *Draco.* https://github.com/google/draco

[24] Google. 2022. *snappy.* https://github.com/google/snappy

[25] Stefan Gumhold, Zachi Kami, Martin Isenburg, and Hans-Peter Seidel. 2005. Predictive point-cloud compression. In *ACM SIGGRAPH 2005 Sketches*. 137–es.

[26] Hamidreza Houshiar and Andreas Nüchter. 2015. 3D point cloud compression using conventional image compression for efficient data transmission. In *2015 XXV International Conference on Information, Communication and Automation Technologies (ICAT)*. IEEE, 1–8.

[27] Lila Huang, Shenlong Wang, Kelvin Wong, Jerry Liu, and Raquel Urtasun. 2020. Octsqueeze: Octree-structured entropy model for lidar compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1313–1323.

[28] Yan Huang, Jingliang Peng, C-C Jay Kuo, and M Gopi. 2006. Octree-Based Progressive Geometry Coding of Point Clouds.. In *PBG@ SIGGRAPH*. 103–110.

[29] David A Huffman. 1952. A method for the construction of minimum redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

[30] Tianyi Li, Lu Chen, Christian S. Jensen, and Torben Bach Pedersen. 2021. TRACE: Real-time Compression of Streaming Trajectories in Road Networks. *Proc. VLDB Endow.* 14, 7 (2021), 1175–1187.

[31] Yuan Li, Bo Wu, and Xuming Ge. 2019. Structural segmentation and classification of mobile laser scanning point clouds with large variations in point density. *ISPRS Journal of Photogrammetry and Remote Sensing* 153 (2019), 151–165.

[32] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J. Elmore. 2021. Decomposed Bounded Floats for Fast Compression and Queries. *Proc. VLDB Endow.* 14, 11 (2021), 2586–2598.

[33] Hao Liu, Hui Yuan, Qi Liu, Junhui Hou, and Ju Liu. 2019. A comprehensive study and comparison of core technologies for MPEG 3-D point cloud compression. *IEEE Transactions on Broadcasting* 66, 3 (2019), 701–717.

[34] Jean loup Gailly and Mark Adler. 2004. Zlib compression library. (2004).

[35] Yuexin Ma, Xinge Zhu, Sibo Zhang, Ruigang Yang, Wenping Wang, and Dinesh Manocha. 2019. Trafficpredict: Trajectory prediction for heterogeneous traffic-agents. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 6120–6127.

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

[37] Bruce Merry, Patrick Marais, and James Gain. 2006. Compression of dense and regular point clouds. In *Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*. 15–20.

[38] MPEGGroup. 2021. *mpeg-pcc-tmc13.* https://github.com/MPEGGroup/mpeg-pcc-tmc13

[39] Aiden Nibali and Zhen He. 2015. Trajic: An Effective Compression System for Trajectory Data. *IEEE Trans. Knowl. Data Eng.* 27, 11 (2015), 3138–3151.

[40] Tilo Ochotta and Dietmar Saupe. 2004. *Compression of point-based 3D models by shape-adaptive wavelet coding of multi-height fields.* 103–112 pages.

[41] Opensignal. 2019. *USA Mobile Network Experience Report.* https://www.opensignal.com/reports/2019/07/usa/mobile-network-experience

[42] Gaurav Pandey, James R McBride, and Ryan M Eustice. 2011. Ford campus vision and lidar data set. *The International Journal of Robotics Research* 30, 13 (2011), 1543–1552.

[43] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (2015), 1816–1827.

[44] Jingliang Peng and C-C Jay Kuo. 2005. Geometry-guided progressive lossless 3D mesh coding with octree (OT) decomposition. In *ACM SIGGRAPH 2005 Papers*. 609–616.

[45] Zizheng Que, Guo Lu, and Dong Xu. 2021. VoxelContext-Net: An Octree based Framework for Point Cloud Compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 6042–6051.

[46] CMPPC Rocchini, Paulo Cignoni, Claudio Montani, Paolo Pingi, and Roberto Scopigno. 2001. A low cost 3D scanner based on structured light. In *Computer Graphics Forum*, Vol. 20. Wiley Online Library, 299–308.

[47] David Salomon and Giovanni Motta. 2010. *Handbook of Data Compression (5. ed.).* Springer.

[48] Ruwen Schnabel and Reinhard Klein. 2006. Octree-based Point-Cloud Compression.. In *PBG@ SIGGRAPH*. 111–120.

[49] Julian Seward. 2009. A program and library for data compression.

[50] Claude E. Shannon. 1948. A mathematical theory of communication. *Bell Syst. Tech. J.* 27, 3 (1948), 379–423.

[51] Stanford. 2014. *The Stanford 3D Scanning Repository.* http://graphics.stanford.edu/data/3Dscanrep/

[52] James A Storer and Thomas G Szymanski. 1982. Data compression via textual substitution. *Journal of the ACM (JACM)* 29, 4 (1982), 928–951.

[53] Xuebin Sun, Han Ma, Yuxiang Sun, and Ming Liu. 2019. A novel point cloud compression algorithm based on clustering. *IEEE Robotics and Automation Letters* 4, 2 (2019), 2132–2139.

[54] Chenxi Tu, Eijiro Takeuchi, Chiyomi Miyajima, and Kazuya Takeda. 2016. Compressing continuous point cloud data using image compression methods. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 1712–1719.

[55] Chenxi Tu, Eijiro Takeuchi, Chiyomi Miyajima, and Kazuya Takeda. 2017. Continuous point cloud data compression using SLAM based prediction. In *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1744–1751.

[56] Lujia Wang, Luyu Wang, Yinting Luo, and Ming Liu. 2017. Point-cloud compression using data independent method—A 3D discrete cosine transform approach. In *2017 IEEE International Conference on Information and Automation (ICIA)*. IEEE, 1–6.

[57] Michael Waschbüsch, Markus H Gross, Felix Eberhard, Edouard Lamboray, and Stephan Würmlin. 2004. Progressive Compression of Point-Sampled Models.. In *PBG*. 95–102.

[58] Ian H Witten, Radford M Neal, and John G Cleary. 1987. Arithmetic coding for data compression. *Commun. ACM* 30, 6 (1987), 520–540.

[59] Linfu Xie, Qing Zhu, Han Hu, Bo Wu, Yuan Li, Yeting Zhang, and Ruofei Zhong. 2018. Hierarchical regularization of building boundaries in noisy aerial laser scanning and photogrammetric point clouds. *Remote Sensing* 10, 12 (2018), 1996.

[60] Pietro Zanuttigh and Guido M Cortelazzo. 2009. Compression of depth information for 3D rendering. In *2009 3DTV Conference: The True Vision-Capture, Transmission and Display of 3D Video*. IEEE, 1–4.

[61] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* 23, 3 (1977), 337–343.