Improving the efficiency of point cloud data management

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
Genehmigte Dissertation von Pascal Bormann aus Mannheim
Tag der Einreichung: 21.03.2024, Tag der Prüfung: 08.05.2024

1. Gutachten: Prof. Dr. Dr. eh. Dieter W. Fellner
2. Gutachten: Prof. Dr. Alexander Reiterer
Darmstadt, Technische Universität Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Fraunhofer IGD

Improving the efficiency of point cloud data management

Accepted doctoral thesis by Pascal Bormann

Date of submission: 21.03.2024
Date of thesis defense: 08.05.2024

Darmstadt, Technische Universität Darmstadt

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-275267
URL: https://tuprints.ulb.tu-darmstadt.de/27526
Jahr der Veröffentlichung auf TUprints: 2024

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
https://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Erklärungen laut Promotionsordnung

§ 8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§ 8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§ 9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation – abgesehen von den in ihr ausdrücklich genannten Hilfen – selbstständig verfasst wurde und dass die „Grundsätze zur Sicherung guter wissenschaftlicher Praxis an der Technischen Universität Darmstadt“ und die „Leitlinien zum Umgang mit digitalen Forschungsdaten an der TU Darmstadt“ in den jeweils aktuellen Versionen bei der Verfassung der Dissertation beachtet wurden.

§ 9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 21.03.2024

P. Bormann

Abstract

The collection of point cloud data has increased drastically in recent years, which poses challenges for the data management layer. Multi-billion point datasets are commonplace and users are getting accustomed to real-time data exploration in the Web. To make this possible, existing point cloud data management approaches rely on optimized data formats which are time- and resource-intensive to generate. This introduces long wait times before data can be used and frequent data duplication, since these optimized formats are often domain- or application-specific. As a result, data management is a challenging and expensive aspect when developing applications that use point cloud data.

We observe that the interaction between applications and the point cloud data management layer can be modeled as a series of queries similar to those found in traditional databases. Based on this observation, we evaluate current point cloud data management using three query metrics: *Responsiveness*, *throughput*, and *expressiveness*. We contribute to the current state of the art by improving these metrics for both the handling of raw files without preprocessing, as well as indexed point clouds.

In the domain of unindexed point cloud data, we introduce the concept of *ad-hoc queries*, which are queries executed ad-hoc on raw point cloud files. We demonstrate that ad-hoc queries can improve query responsiveness significantly as they do not require long wait times for indexing or database imports. Using columnar memory layouts, queries on datasets of up to a billion points can be answered in interactive or near-interactive time, with throughputs of more than one hundred million points per second on unindexed data. A demonstration of an adaptive indexing method shows that spending a few seconds per query on index creation can improve responsiveness by up to an order of magnitude. Our experiments also confirm the importance of high-throughput systems when querying point cloud data, as the overhead of data transmission has a significant effect on the overall query performance.

For situations where indexing is mandatory, we demonstrate improvements to the runtime performance of existing point cloud indexing tools. We developed a fast indexer based on task-parallel programming, using Morton indices to efficiently sort and distribute point batches onto worker threads. This system, called *Schwarzwald*, outperformed existing indexers by up to a factor 9 when it was first published, and still has com-

petitive performance to current out-of-core capable indexers. Additionally we adapted our indexing algorithm for distributed processing in a Cloud-environment and demonstrate that its horizontal scalability allows it to outperform all existing indexers by up to a factor of 3.

Lastly we demonstrated point cloud indexing in real-time during Light Detection And Ranging (LiDAR) capturing, based on a similar task-based algorithm but optimized for progressive indexing. Our real-time indexer is able to keep up with current LiDAR sensors in a real-world test, with end-to-end latencies as low as 0.1 seconds.

Together, our improvements significantly reduce wait times for working with point cloud data and increase the overall efficiency of the data access layer.

Zusammenfassung

Die Größe und Menge von Punktwolken-Datensätzen, welche durch verschiedene Verfahren generiert werden, ist im letzten Jahrzehnt stark gewachsen. Datensätze mit Milliarden oder sogar Billionen von Punkten sind keine Seltenheit mehr. Moderne Verfahren ermöglichen den interaktiven Umgang mit verschiedensten Geodaten. Um die Arbeit mit Punktwolken im gleichen Maße zu ermöglichen, bedarf es ausgefeilter Datenhaltungs-Lösungen. Hierbei kommen dabei aktuell zeit- und rechenintensive Verfahren zum Einsatz, welche Punktwolkendaten strukturieren und in optimierte Formate bringen. Dabei kommt es zu langen Wartezeiten in der Aufbereitung und häufig zur Duplikation der Daten, da viele dieser optimierten Formate domänen- oder anwendungsspezifisch sind. In der Entwicklung von Anwendungen auf Basis von Punktwolken ist daher eine effiziente Datenhaltung eine der großen Herausforderungen.

Auch wenn Datenbanken nach wie vor eine Nischenlösung in der Datenhaltung von Punktwolken sind, so lässt sich die Interaktion zwischen Anwendungen und der Datenhaltung analog zu Datenbank-Abfragen darstellen. Basierend auf dieser Beobachtung lassen sich die aktuellen Ansätze zur Punktwolken-Datenhaltung anhand von drei Kriterien bewerten: Die *Antwortzeit*, der *Durchsatz*, und die *Ausdruckskraft* von Punktwolken-Abfragen. Unser Beitrag zum aktuellen Stand der Forschung sind Verbesserungen dieser drei Kriterien für zwei gängige Datenhaltungs-Ansätze: Das Arbeiten mit Rohdaten sowie die Indexierung von Punktwolken.

Im Bereich der Arbeit mit Rohdaten führen wir das Konzept der *Ad-hoc Abfragen* ein und zeigen, dass moderne Hardware effizient genug ist, um viele gängige Abfragen in kurzer Zeit und ohne Vorverarbeitung durchführen zu können. Im Vergleich zu typischen Indexierungsprozessen oder einem Datenbank-Import können *Ad-hoc Abfragen* die Antwortzeit deutlich verbessern. Mit Hilfe spaltenbasierter Datenformate ist dabei ein Durchsatz von über 100 Millionen Punkten pro Sekunde möglich. Weiterhin demonstrieren wir, dass adaptive Indexierung die Antwortzeiten bestimmter Abfragen um bis zu einer Größenordnung verringern kann. Eine umfangreiche Evaluation von *Ad-hoc Abfragen* demonstriert deren Machbarkeit und zeigt den Zusammenhang zwischen Datendurchsatz und Antwortzeit auf.

Im Bereich indexierter Punktwolken verbessern wir die Laufzeit aktueller Indexierungs-

Algorithmen. Basierend auf dem *task-parallel programming* Ansatz und Morton Indizes haben wir *Schwarzwald* entwickelt, ein System zur schnellen Berechnung eines für die Visualisierung optimierten Index. Zum Zeitpunkt der Erstveröffentlichung war *Schwarzwald* bis zu 9 mal schneller beim Erstellen eines gleichwertigen Index verglichen mit bestehenden Lösungen. Die Laufzeit der schnellsten aktuell verfügbaren Out-Of-Core Indexer ist vergleichbar zu der von *Schwarzwald*, wie wir in mehreren Testreihen belegen. Wir demonstrieren außerdem, dass der zugrundeliegende Algorithmus von *Schwarzwald* für die verteilte Verarbeitung in der Cloud adaptiert werden kann, was zu besserer Skalierbarkeit und bis zu dreimal kürzeren Laufzeiten verglichen mit bestehenden Systemen führt.

Zur weiteren Reduktion von Wartezeiten demonstrieren wir außerdem das erste echtzeitfähige Indexierungs-System für Punktwolken, welches eine Indexierung direkt während der Aufnahme am LiDAR Sensor ermöglicht. Die Verwendung unseres Echtzeit-Indexers reduziert dabei die Wartezeit von der Aufnahme bis zur Nutzung der Daten um mehrere Größenordnungen in den Bereich unterhalb einer Sekunde.

Unsere Verbesserungen verringern bestehende Wartezeiten in der Vorverarbeitung signifikant und erhöhen somit die Effizient in der Punktwolken-Datenhaltung.

Acknowledgment

This thesis has been a major part of my life for the past five years. During this time, it occupied not only my working hours but a lot of mental resources as well, resulting in significantly more struggle than I would have wished for. There were several occasions where I was sure I would quit, I was riddled with self-doubt, and to be honest I still am not sure what it is that made me persevere in the end. What I do know is that finishing this thesis would not have been possible if not for the support, both technically and emotionally, of some incredible people. Acknowledging you all is something I care about deeply, and I hope that my words will convey my feelings towards you adequately. Thank you:

Elke, for setting me on this path and for all the opportunities you created for me. Ralf, for showing me that passion and cold, hard science do mix. Eva, for your support and incredible patience while I was anything but. Michel, for being the very model of a scientist and uplifting everyone around you. Kevin, for being my everyday rage-buddy. Tobias, for having my back and all the trips to aniko. Hendrik, for that one hug that I really needed. Also for your Mensa card. Benjamin, for being available when I needed to vent. Daniel, for being a constant throughout so much change, and for more philosophical debates than I can count. Lee, for always creating opportunities for me to get out of the house and party a little. Julia, for all the good we could share despite all the struggles. Melina, for making me believe in myself again. To my family, for not giving up on me even when I gave up on myself.

Lastly, as an avid reader, there were many books which were both a source of inspiration and a means of escape for me. To anyone who might struggle in a similar way while writing their thesis—or any other endeavor for that matter—I want to leave you with some words from one of these books that provided great comfort to me: “The most important step a man can take. It’s not the first one, is it? It’s the next one. Always the next step, Dalinar.”

Contents

Acronyms

AABB Axis-Aligned Bounding Box. 22

ALS Airborn Laser Scanning. 128

API Application Programming Interface. 14, 136, 140

CPU Central Processing Unit. xxi, xxiv, 40, 78, 81, 82, 97, 102, 103, 106, 107, 108, 109, 120, 129, 138

DBMS Database Management System. 7, 8, 17, 18, 19, 28, 29, 32, 39, 44, 45, 69, 74, 76, 91, 132, 136

DEM Digital Elevation Model. 1, 5

MNO Modifiable Nested Octree. xvii, xviii, 115, 128, 130

FTTH Fiber to the Home. 6, 139

GNSS Global Navigation Satellite System. 120, 121, 128

GPU Graphics Processing Unit. 26, 31, 40, 45, 56, 78, 79, 109, 120, 129, 135

IMU Inertial Measurement Unit. 120, 128

I/O Input/Output. 15, 20, 21, 22, 27, 39, 40, 42, 44, 46, 47, 48, 50, 54, 56, 57, 58, 60, 69, 70, 73, 76, 85, 90, 104, 107, 108, 109, 116, 117, 133

LiDAR Light Detection And Ranging. vi, viii, xxi, 5, 6, 10, 11, 13, 14, 15, 16, 27, 36, 37, 51, 73, 91, 111, 112, 113, 114, 116, 117, 120, 127, 128, 129, 130, 133, 136

LOD Level Of Detail. xxi, 8, 19, 25, 26, 27, 29, 30, 31, 32, 34, 35, 36, 38, 40, 46, 60, 61, 62, 63, 68, 69, 75, 76, 78, 79, 82, 92, 93, 94, 113, 114, 118, 119, 125, 133, 135, 136, 137, 140

LRU Least-Recently-Used. 117, 122, 123, 125

MMS Mobile Mapping System. 137, 139

NMEA National Marine Electronics Association. 121

OGC Open Geospatial Consortium. 16, 136, 140

PPS Pulse Per Second. 121

SFC Space-Filling Curve. 19

SLAM Simultaneous Localization And Mapping. 113, 120, 130

TLS Terrestrial Laser Scanning. 128

VM Virtual Machine. 78, 135, 138

Glossary

3D Tiles File format for streaming 3D geospatial datasets in the web [31]. 16, 17, 19, 37, 49, 87, 90, 91, 96, 139

ad-hoc queries Queries issued directly (ad-hoc) on raw point cloud data. xiii, xxiii, 10, 11, 33, 36, 37, 38, 39, 40, 48, 50, 51, 52, 62, 68, 69, 72, 73, 74, 75, 76, 112, 127, 130, 132, 133, 134, 136, 137, 139, 140, 141

Amazon S3 Object storage service provided by Amazon. 112

Apache Cassandra A distributed NoSQL database [41]. 91

CesiumJS JavaScript library for visualizing 3D geospatial data in the web [25]. 15, 37, 136, 139, 140

CSV Comma-separated values, a common textual file format for table data. 28

Entwine Point cloud indexing software [61]. xxi, xxiv, 7, 69, 80, 81, 84, 87, 90, 95, 96, 97, 98, 99, 100, 102, 103, 104, 105, 107, 108, 112, 113, 124, 138

Fibre3D Interactive planning software for fibre network rollout in Germany. xix, xxiv, 6, 7, 112, 131, 136, 137, 138, 139, 140

Hybrid MNO Grid Custom point cloud index structure that combines an MNO with a hash-grid to allow dynamically extending the index. xxi, 92, 93, 113, 114

JSON JavaScript Object Notation, a ubiquituous file format for exchanging data objects. 28

LAS A common file format for LiDAR point clouds [7]. xviii, xix, xx, xxiii, 5, 7, 11, 15, 16, 17, 18, 20, 33, 35, 37, 45, 46, 47, 48, 49, 50, 51, 53, 54, 55, 56, 57, 58, 59, 60, 61, 63, 65, 66, 67, 68, 73, 74, 76, 89, 90, 91, 96, 97, 98, 99, 104, 115, 117, 125, 128

LAZ A compressed file format based on the *LAS* format [70]. xx, 15, 16, 17, 18, 20, 46, 47, 51, 54, 55, 56, 57, 58, 59, 61, 63, 65, 66, 67, 68, 70, 73, 74, 75, 76, 90, 96, 97, 98, 99, 115, 117, 136

MNO Abbreviation for Modifable Nested Octree, a point cloud index structure used for visualization [129]. xv, xxi, 11, 26, 27, 30, 43, 80, 92, 93, 95, 107, 111, 114, 115, 127

MPts/s Million Points per second. 59, 61, 65, 66, 67, 138

out-of-core Refers to an algorithm which can process data that does not fit into working memory. 27, 31, 78, 79, 81, 114, 135

pasture Rust library for point cloud data management developed as part of this thesis [15]. xii, 39, 52, 53, 57, 58, 59, 73, 135

PDAL Point Data Abstraction Layer, a software-library for accessing point cloud data [29]. xxiii, 15, 16, 52, 53, 57, 58, 59, 61, 62, 63, 75, 76, 96, 108, 135

Potree Web-based point cloud viewer [139]. xviii, 15, 28, 37, 43, 44, 87, 90, 95, 98, 136, 139, 140

PotreeConverter Point cloud indexing software developed for the Potree web-viewer [140]. xxi, xxiv, 78, 80, 81, 82, 84, 85, 87, 88, 89, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 107, 108, 109, 110, 113, 115, 118, 129, 133, 138

Schwarzwald Fast point cloud indexing software developed as part of this thesis. v, xxi, xxiv, 77, 78, 80, 81, 87, 89, 91, 93, 94, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 113, 115, 127, 129, 130, 133, 134, 137, 138

List of Figures

List of Tables

1 Introduction

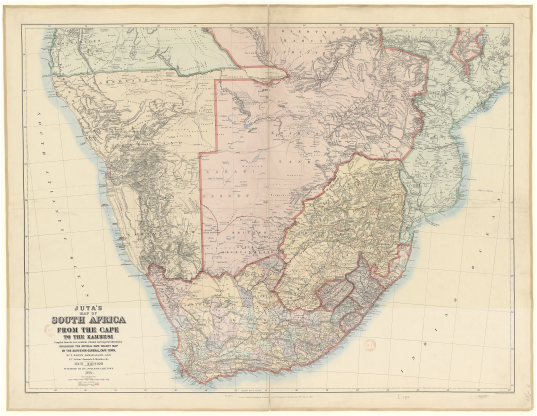
"The world keeps happening, in accordance with its rules; it's up to us to make sense of it and give it value."

Sean M. Carroll

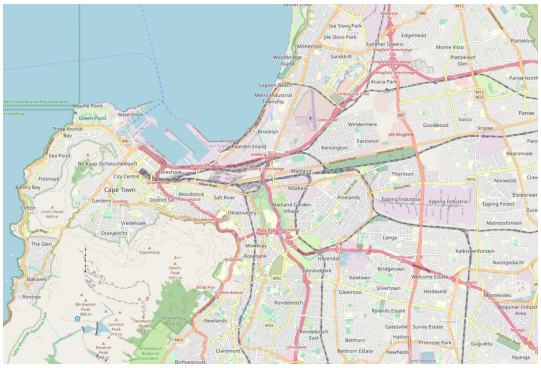
Trying to understand the world around is is a key part of what it means to be human, and a driving factor behind the achievements of the human race. We adopted language to share information about us and the world, developed the written word to persist this information, and came up with the scientific method to make sense of this information. While for the longest time in human civilization, information processing was constrained by the computational power of our brains, the advent of the digital age has given us never-before seen capabilities to process vast quantities of information and make sense of it, all aimed at that singular goal to understand the world around us and our place in it.

When it comes to understanding the physical world, digital representations of reality are necessary to unleash the potential of our computational engines. An important part of these digital representations is played by *geospatial data*: Data that is directly linked to geographical locations [97]. Examples of geospatial data include satellite imagery, weather data, maps, DEMs, city models, and point clouds. Using geospatial data, we can better understand the current state of our world, how it changes, how we humans impact it and drive that change, and it gives us the means to plan how we want to shape the world. As visual creatures, our understanding of the world is often tied to what we can see, as supported by the popular proverb "An image is worth more than a thousand words". Hence, many types of geospatial data are visual in nature: Maps are perhaps the oldest form of visual geospatial data, used by humans for thousands of years to navigate the globe. More recently, we have begun to create elaborate portrayals of the world in digital form. Maps have become digital, augmented with contextual information, and have extended from the two-dimensional plane into three-dimensional space. We have built services that capture the real world and store it in digital form, enabling anyone

to (virtually) visit almost any place on earth through its digital representation [69, 54, 14, 91]. Figure 1.1 shows how the fidelity of these modern representations compares to traditional maps.

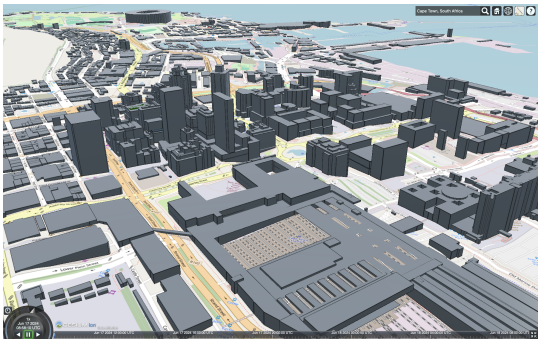


(a) Historical map (South Africa) [72]



(b) Digital map (Cape Town) [107]

(c) Panoramic image (Cape Town) [117]



(d) Interactive 3D city model (Cape Town) displayed with *CesiumJS* [25] using data from *OpenStreetMap* [108]

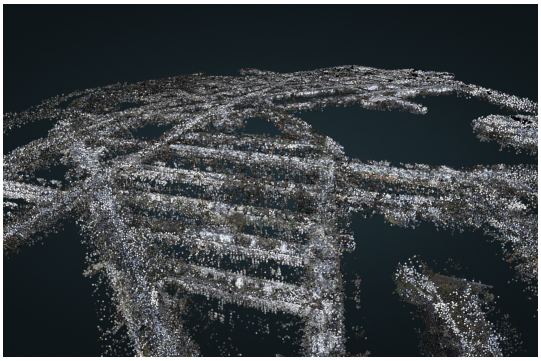
Figure 1.1: Various geospatial data representations of our real world

The world around us is complex. Our digital representations have to be able to capture a wide variety of properties. We want to be able to capture the tree structure of the world's oldest rainforests, the fine structure of cultural heritage sites, the complexity of mega-cities as well as the variety of agricultural land usage. No one data format can capture all these aspects in the required level of detail, but when it comes to capturing and understanding the *shape* of our world, *point clouds* are a powerful and widely used digital

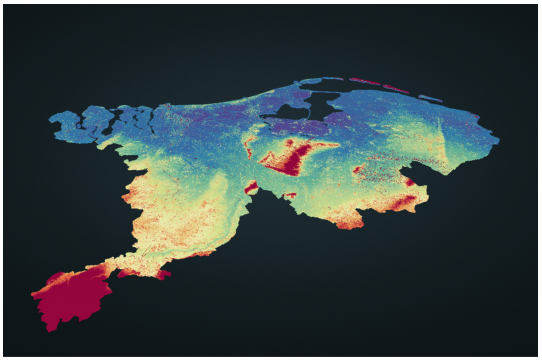
representation. A point cloud is a collection of a large number of points which each carry information encoded in typically only a handful of attributes, such as the point's position in three-dimensional space, its color, or even semantic information such as what type of entity the point represents. Given enough points, it is possible to approximate real-world objects with sufficient detail to gain insight into their structure, either algorithmically or by visualizing the point cloud. Figure 1.2 shows point clouds at several different scales, from single buildings to whole countries.

(a) Indoor point cloud [19]

(b) Street-scale point cloud



(c) City-scale point cloud



(d) Country-scale point cloud [1]

Figure 1.2: With point clouds, indoor scenes can be represented just as well as whole countries. Data for images (b) and (c) courtesy of Deutsche Telekom IT.

1.1 Point clouds as a tool for understanding the world

In this thesis, we explore the domain of point cloud data from a computer science perspective, starting from the bits and structures that make up a point cloud, to the way we can store, organize, and work with point clouds within software. Understanding and in particular optimizing the structure of point clouds requires understanding why point clouds are a useful data representation and how they are used today.

1.1.1 Introduction to point clouds as a data representation

In principle a point cloud has a simple representation, mathematically nothing more than a list of tuples. In a general-purpose programming language, such as C++, a point cloud data structure can be expressed in a few lines of code:

```
struct Point {
    float x, y, z;              // Position in 3D space
    std::byte red, green, blue; // RGB color
    std::byte classification;   // Object class
};
std::vector<Point> point_cloud;
```

The term "point" in "point cloud" does not refer to the mathematical concept of a zero-dimensional object. Instead the points in a point cloud are approximations of the local surface of the geometry that the point cloud represents. When visualizing a point cloud, this surface is reconstructed from the points using a technique called *splatting* [159]. Compared to other geometric representations, such as triangle meshes or parametric surfaces, point clouds require no connectivity information and thus are one of the simplest geometric representations. Conceptually, they sit between triangle meshes as pure surface representations, and voxels as pure volumetric data, which makes them well-suited for representing various natural structures, for example trees [85, 90]. The downside of the lack of connectivity information is that a large number of points are required to accurately represent surfaces, which results in point cloud datasets with millions, billions, or even trillions of points.

These facts illustrate what sets point clouds apart from other types of data in the domain of computer graphics and why working with point clouds has its own set of challenges: Images and voxel data are similar in terms of number of samples, but where their samples fall on a regular grid, point clouds are often irregularly sampled, making data layout and access more challenging. This is one of the key contributors to many

of the performance challenges that this thesis deals with. Triangle meshes on the other hand contain irregular samples as well, but their connectivity information allows data reductions that are not trivially possible for point clouds.

The attributes of each point provide a great deal of flexibility for domain-specific data to be encoded into the point cloud. Physical information, such as reflectance or the number of returned laser pulses, can be encoded just as easily as semantic information about the type of object that a point belongs to. As an example, the popular *LAS* file format defines over 20 different point attributes [7]. There are many different processes through which point clouds can be obtained, from photogrammetric methods that compute a point cloud from one or more images [158], to physical scanners which send out laser pulses and capture their reflections from surfaces and convert them into three-dimensional points [157, 93], a process called LiDAR.

1.1.2 Point cloud applications

The unique properties of point clouds—they are simple, versatile in physical size and expressive capabilities, volumetric, and cheap to obtain—make them well-suited as a data representation for a wide variety of applications: Measuring the precision of small-scale structures in manufacturing [165], preserving cultural heritage sites [116, 162], helping self-driving cars understand their surroundings [58] or classifying the structure of forests [163] are all applications where point clouds have been successfully used.

Depending on the application, point clouds might be used directly (for visual analysis, automated change detection, or entertainment) or as an intermediate format (city model creation, Digital Elevation Model (DEM) creation, object classification). For those applications that use point clouds in their raw form, dealing with the vast number of points becomes a challenge, especially when interactivity is desired. Applications which use point clouds only as an intermediate representation are more concerned with the structure of the point cloud and often employ extensive preprocessing to convert the point cloud into a more usable format, such as a semantic model, DEM or triangle mesh.

Many use cases deal with large-scale point clouds, from buildings to cities to whole countries. Here, the number of points and memory size of the point cloud becomes especially challenging, as datasets become too large to fit into working memory or even onto the harddrives of consumer-grade computers. Interactive applications have to be able to quickly filter these large-scale point clouds for relevant information, while preprocessing applications want to minimize their runtime and resource consumption to save time and money. As the geospatial domain is moving towards Cloud-based processing [77], minimizing resource consumption becomes an important factor as compute and memory usage directly translate to cost. Geospatial data is Big Data [84] and thus faces the same

challenges as other Big Data, in particular due to their large volume. Point clouds are no exception and the challenges due to the size of point clouds have been recognized in the scientific community [150, 119, 132]. The software and algorithms that process, analyze and visualize point clouds have to be able to deal with large quantities of data and should be scalable to deal with the expected increase in data volume due to improvements in point cloud capturing capabilities. Beyond just keeping up with the influx of data, it would be beneficial to decrease the time from capturing the data to it being usable, as this would allow users to quickly decide what data they are interested in, discarding unnecessary data and thus increasing processing speed, reducing resource usage and ultimately cost.

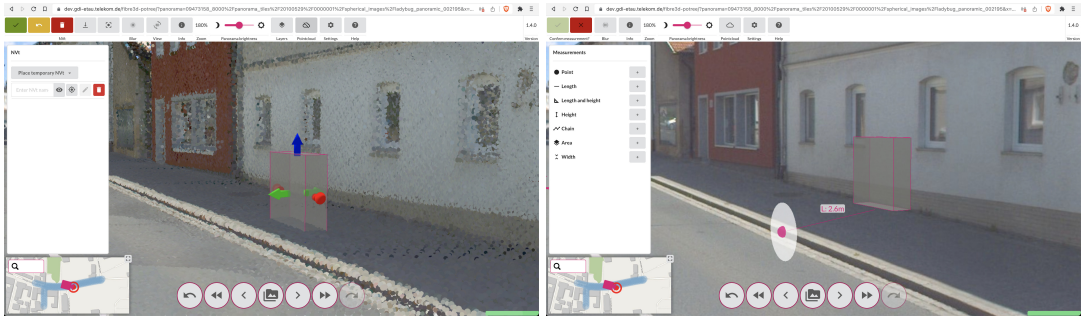


Figure 1.3: Virtually placing a fiber distribution cabinet in the interactive planning software *Fibre3D*, using spatial information provided by a point cloud. Images taken from [78]

A use case that covers many of these challenges is the rollout of high-speed fiber networks in Germany. To plan fiber network connections for as many as 3 million households per year, the telecommunications provider Deutsche Telekom AG built a cloud-based platform for digital infrastructure planning. Through this platform, planners can plan and verify the locations of trenches and distribution cabinets for fiber optics up to individual homes (a technique called Fiber to the Home (FTTH)). Part of this platform is the web-based planning and visualization tool *Fibre3D*, which visualizes 360° panoramic images as well as point clouds and was developed in conjunction between Deutsche Telekom AG and the Fraunhofer Institute for Computer Graphics Research IGD. Through the point clouds, users can perform interactive placement of objects within the panoramic images, as well as perform accurate spatial measurements, as can be seen in Figure 1.3. To enable *Fibre3D*, Deutsche Telekom AG are capturing vast quantities of image and point cloud data through specialized cars equipped with cameras and LiDAR scanners that drive through the cities and villages where fiber rollout is planned. At the point of writing, about half

a Petabyte of data has been captured, with roughly a third of this size for the point cloud alone. The resulting point cloud contains about 6.75 trillion points, almost an order of magnitude more than the current AHN dataset [2], which is frequently used in the scientific community to illustrate challenges related to very large point clouds [150, 96]. For interactive visualization, large-scale preprocessing is required to make the point clouds usable, for example by streaming them to a client application running in a web-browser.

The *Fibre3D* tool and the underlying data processing pipeline are described in a forthcoming paper [78] and are one example for the challenges one encounters when working with large-scale point cloud data. Dealing with these challenges requires investigating and answering the research questions defined in Section 1.3.

1.2 Problem statement

Since point cloud data is generally large and unstructured, point cloud data management is challenging. Continuous data acquisition with sophisticated scanners generates datasets with trillions of points [148] while users are accustomed to consuming digital content through web-browsers and mobile devices. To serve point cloud data to a wide variety of applications, time- and resource-intensive preprocessing is used, which creates optimized data structures [129, 133, 89]. Besides being costly to create, these structures are domain-specific, so different applications require different types of preprocessing. Despite considerable research on using Database Management System (DBMSs) for managing point cloud data in a more general way [150, 32, 86], point cloud databases remain an often ignored solution in favor of file-based data management.

The current state of point cloud data management consists of three competing approaches, all with their own set of shortcomings:

Raw files Point cloud data is stored in one or more files using common formats such as *LAS*. Within a file, points are generally stored unordered with limited structure. This storage model is the simplest one, as it requires only a file system, but the lack of structure limits the types of computations that can be achieved without considerable computational overhead.

Standalone index Point cloud data is stored together with an index, which allows efficient access to subsets of the point cloud. Often, the index is implicit within the structure of the data, for example the *Entwine* indexer uses an octree as the index structure and stores octree nodes as individual files on disk [61]. This is highly efficient especially for web-based streaming, as accessing a single tree node is equivalent to fetching a single file. The downsides of this data management solution in-

clude the computational effort of creating the index structure, sometimes resulting in downright crashes due to excessive memory usage, as well as its domain-specific nature, typically resulting in a copy of the data that is unusable by other applications.

Point cloud Database Management System (DBMS) The point cloud data layout is managed by the DBMS, as well as the creation and structure of any required indices. The query capabilities of the DBMS can support a wide range of applications through a unified data access layer, making it the most general point cloud data management solution. Making a DBMS work well with a large point cloud with billions of points is still nontrivial: Naively storing all points as individual records blows up memory usage and index size, while grouping strategies require time-consuming preprocessing and also decrease the query throughput.

Within this thesis, we improve upon the current state of point cloud data management. For this, we build on an assumption based on observing the way applications access point cloud data in practice, namely that *every point cloud data access is equivalent to a query*. To find a good data management solution for a point cloud application means understanding the types of queries that the application issues on the data:

- Visualizations of point clouds boil down to view-frustum and importance/LOD queries: "The current machine can only render N points per frame at interactive speeds, so for the current view frustum, find the N most important points"
- Segmentation and mesh reconstruction algorithms often boil down to neighborhood queries: "For each point, find its k nearest neighbors in order to approximate the local surface at this point"
- Analysis applications often employ queries natively: "Find all points that intersect the given building footprint and calculate the median height of the points to approximate the building height"

We thus map the problems of the current point cloud data management solutions onto three quantifiable properties of queries: Query *responsiveness*, query *throughput*, and query *expressiveness*. *Responsiveness* dictates how quickly the application will get a response to the query[1]. It not only includes the runtime of the query itself, but for explorative scenarios also includes any preprocessing time, for example an import into a

[1] We use responsiveness in favor of its inverse parameter *latency*. This way all three query properties evaluated in this thesis are positive quantities, which makes comparisons easy: The higher the quantity, the better!

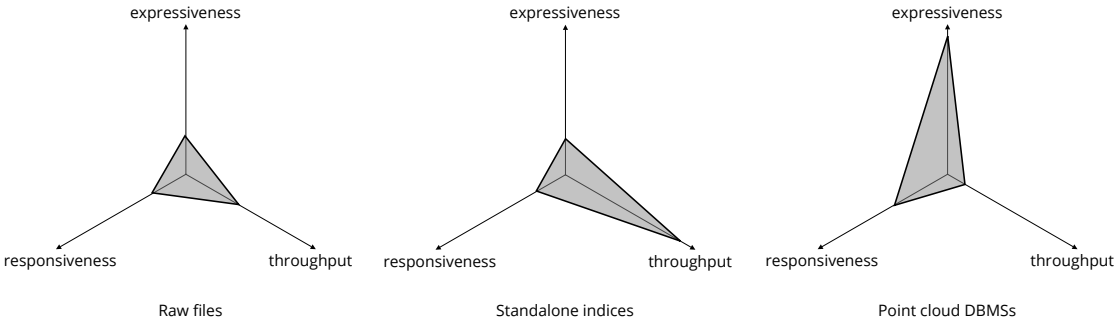


Figure 1.4: Query parameters for the main point cloud data management approaches used today.

database or the creation of an index structure. *Throughput* dictates how much data can be retrieved in a given time period, and is especially important for visualization applications, as they require large quantities of points to achieve good visual quality. *Expressiveness* refers to the complexity and variety of the queries and is often vital to enable analysis applications, which might combine an arbitrary number of point attributes in a single query. Of the three properties, expressiveness is the hardest to quantify, but in Chapter 3 we show that it can be related to the other two properties, in particular to responsiveness.

The current strengths and weaknesses of the three main point cloud data management approaches based on these three query properties are illustrated visually in Figure 1.4. Improving query responsiveness, throughput, and expressiveness will benefit point cloud applications in a variety of ways, such as:

- Performance improvements of preprocessing tools, specifically reduced runtimes, reduced wait times and decreased cost.

- Enabling processes to run interactively and/or in real-time that previously were only possible as offline batch processes.

- Reducing or removing wait-time for data imports, thus enabling faster data exploration.

1.3 Hypothesis and research questions

Based on the problem statement, we state the main *research hypothesis* of this thesis:

> **Hypothesis**
>
> *Using adaptive indexing, parallel programming, as well as columnar memory layouts improves the query throughput, responsiveness, and expressiveness of existing point cloud data management approaches.*

An in-depth overview of the current state of the art in Chapter 2 will explain why we chose the three specific approaches *adaptive indexing*, *parallel programming*, and *columnar memory layouts* as promising candidates for improving the current state of point cloud data management.

To verify this hypothesis, we answer the following research questions:

RQ1 Can *ad-hoc queries* enable applications to work directly with raw point cloud files instead of sophisticated index structures?

RQ2 How can the runtime of current point cloud indexing tools be improved?

RQ3 Can point clouds be indexed in real-time during the capturing process with a LiDAR scanner?

These research questions deal with three related aspects of point cloud data management. Research question 1 covers unindexed point cloud data and hence the data management approach that uses raw files as storage. Improving the capabilities of raw point cloud files is beneficial to all point cloud applications, as ultimately every data management approach has to interact with raw files at some point. Still, indexing is often necessary, so research question 2 will help to understand how point cloud data management based on indexing can be improved. Research question 3 then asks how data management requirements like indexing can be integrated into the capturing process. Capturing data in an optimized format would again benefit all point cloud applications and enable fast data exploration.

1.4 Contributions

Each research question is answered in a dedicated chapter of this thesis. These three content chapters contain several contributions that we made to the current state of the

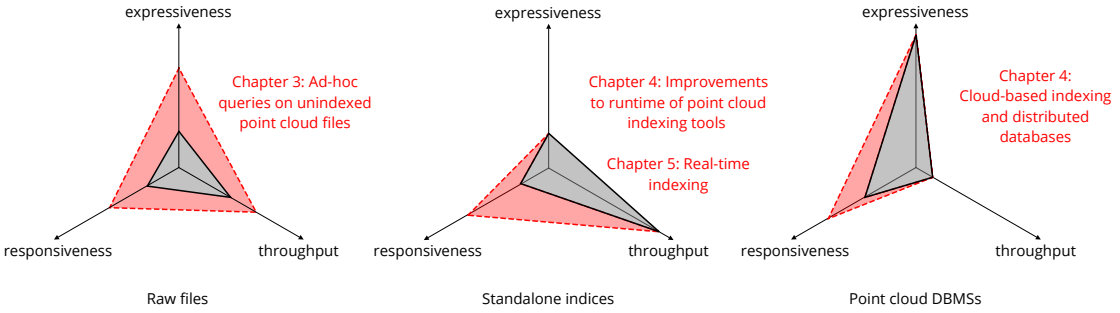


Figure 1.5: Our contributions to the state of the art of point cloud data management

art of point cloud data management. Our contributions are summarized in the following paragraphs, with a visual overview shown in Figure 1.5.

Research question 1 is answered in Chapter 3, where we investigate *ad-hoc queries* as a potential way through which applications can issue queries on raw point cloud data without requiring any indexing. We developed a concept and reference implementation for an *ad-hoc query engine* to achieve this goal, together with a theoretical model for the throughput and latency of *ad-hoc queries*. We also developed a new file format that uses a columnar memory layout to speed up queries on raw data compared to existing formats such as *LAS*. The feasibility of *ad-hoc queries* is evaluated in an extensive case-study.

Research question 2 is answered in Chapter 4, where we introduce our novel point cloud indexing algorithm based on task-parallel programming to index point clouds of arbitrary size faster than the previous state of the art. We then extended our indexing algorithm into a distributed point cloud indexing system built specifically for processing in the Cloud and demonstrate that its horizontal scalability allows it to outperform all existing point cloud indexing systems.

Research question 3 is answered in Chapter 5, where we developed the first system capable of indexing point clouds in real-time during LiDAR capture. To enable this we developed a new stream-based point cloud indexing algorithm that builds a high-quality *Modifiable Nested Octree* index in real-time. We evaluated the stream-based indexer using a real-world sensor system, demonstrating the feasibility of real-time indexing. We were also able to demonstrate an end-to-end latency of about 0.1 seconds from the time a point was captured by the scanner to the time it appeared in the query response, giving real-time indexing a query latency several orders of magnitude lower than any existing point cloud data management solution.

1.5 Publications

This thesis is based upon several peer-reviewed publications, including one upcoming publication which has been reviewed and accepted but is not yet published. We list all relevant publications and the chapters that they belong to.

Chapter 3

- **Bormann, P.**, Krämer, M., & Würz, H. M. (2022). Working Efficiently with Large Geodata Files using Ad-hoc Queries. In Proceedings of the 11th International Conference on Data Science, Technology and Applications (pp. 438-445) - DATA [17]
- **Bormann, P.**, Krämer, M., Würz, H. M & Göhringer, P. (2024). Executing ad-hoc queries on large geospatial data sets without acceleration structures. In Springer Nature Computer Science Journal [18]

Chapter 4

- **Bormann, P.**, & Krämer, M. (2020). A System for Fast and Scalable Point Cloud Indexing Using Task Parallelism. In Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference [16]
- Kocon, K., & **Bormann, P.** (2021). Point cloud indexing using Big Data technologies. In 2021 IEEE International Conference on Big Data (Big Data) (pp. 109-119) [76]

Chapter 5

- **Bormann, P.**, Dorra, T., Stahl, B., & Fellner, D. W. (2022). Real-time Indexing of Point Cloud Data During LiDAR Capture. In Computer Graphics and Visual Computing (CGVC) [19]

Chapter 6

- Krämer, M., & **Bormann, P.** & Würz, H. M. & Kocon, K. & Frechen, T. & Jonas Schmid (2024). A Cloud-Based Data Processing and Visualization Pipeline for the Fibre Roll-Out in Germany. In The Journal of Systems & Software [78]

2 State of the art

> "This is insanity!"
> "No, this is scholarship!"
>
> Brandon Sanderson, Words of Radiance

This chapter covers the current state of the art regarding the three domains *point cloud storage*, *point cloud indexing*, and *point cloud visualization*. Storage and indexing are mandatory topics for any data management solution that deals with larger datasets. We focus on visualization as an application domain due to its importance—many point cloud applications include some form of visualization—as well as the specific challenges that it imposes on the data management layer. These three domains make up the foundation for building an understanding of the current state for point cloud data management. Whenever applicable, we also point out research from beyond the domain of point cloud data if we consider it to be important. This overview also serves as an explanation why this thesis is built around the three specific approaches *adaptive indexing*, *parallel programming*, and *columnar memory layouts* for improving point cloud data management.

2.1 Point cloud storage

Point cloud data poses several challenges to the storage layer of many applications due to the large size of the data. A single point cloud captured by terrestrial or airborne LiDAR sensors is often comprised of billions of points with a size in the high Gigabytes to low Terabytes. Table 2.1 gives an overview over several freely available point cloud datasets with their size characteristics. While hundreds of Gigabytes or even a few Terabytes per dataset ranges at the low end of what is considered Big Data nowadays, it is still large enough that handling point clouds on a single desktop computer can become challenging. Even in a distributed environment such as the Cloud, where compute and storage capabilities often are sufficient, data upload and download operations can become a bottleneck. In addition to that, cheaper and more powerful acquisition systems lead to increases in

Dataset	Size	Number of points	Description
Haiyuan Earthquake Rupture [111]	36.1 GiB	7.5 billion	Survey of the 1920 Haiyuan Earthquake surface rupture in China. Derived using photogrammetry
Southern San Andreas Fault [23]	90 GiB	8.4 billion	Survey of the Southern San Andreas Fault in California, USA. Derived using photogrammetry
Wellington, New Zealand 2013-2014 [3]	247.8 GiB	52.4 billion	Airborne LiDAR scan of the Wellington region, New Zealand
AHN3 [1]	2390 GiB	558 billion	Airborne LiDAR scan of the Netherlands (version 3)
AHN4 [2]	6137 GiB	946 billion	Airborne LiDAR scan of the Netherlands (version 4)
3DEP [148]	$\sim$ 300 TiB	54.6 trillion	Collective airborne LiDAR scan of the United States (ongoing)

Table 2.1: Storage characteristics of several freely available point cloud datasets

both quality and quantity and hence the data volume of current and future point cloud datasets.

The *AHN* dataset [2] is frequently used to illustrate many of the challenges relating to point cloud data size. AHN (short for *Actueel Hoogtebestand Nederland*, roughly translated as *Current Elevation Map of the Netherlands*) is a digital elevation map covering all of the Netherlands based upon airborne LiDAR scans. As part of an ongoing national survey, this dataset has been updated several times since its initial release. The current version is AHN4, which consists of almost a trillion points with a total size of approximately 6 TiB (compressed), while the next version AHN5 is already being created. While the main purpose of the AHN dataset is to have an up-to-date height map, each new version of the dataset also has better data quality than the previous versions. This manifests as an increase in spatial resolution based on higher precision of the LiDAR scanners used. Thus, each new version contains significantly more points than the previous version and is much larger in terms of data volume.

It is interesting to observe that point clouds differ from other geospatial data primarily in the storage domain: Where most geospatial data is mainly stored in databases and accessed through standardized, web-based APIs such as WMS (web map service) and WFS (web feature service), the main storage and exchange format for point clouds are raw files. The following subsections will explore the usage of file-based storage for point cloud data, as well as more recent work into enabling database usage for point clouds.

Name	Type	Description	Programming language(s)	License
CloudCompare [51]	Desktop application	3D point cloud and mesh processing software	C++	GPL
PDAL [29]	Library & command line application	Library and tools for point cloud Input/Output (I/O) and manipulation	C++	BSD
PCL [128]	Library	2D/3D image and point cloud processing	C++	BSD
Potree [139]	Web application	3D point cloud visualization software	JavaScript, C++ (indexer)	BSD
CesiumJS [25]	Web application	3D geospatial data visualization software	JavaScript	Apache 2.0
LAStools [52]	Library & command line applications	Tools for manipulating LiDAR data	C++	GPL 2.1 & commercial license
Point Cloud Utils [160]	Library	Processing and manipulating 3D meshes and point clouds	Python	GPL 2.0
QGIS [121]	Desktop application	Geographic information system	C++	GPL 2

Table 2.2: A list of common tools and software libraries for working with point clouds

2.1.1 File-based storage

To understand how point cloud data is stored and accessed, and in particular why file-based storage is still the method of choice in most applications, we can have a look at the tool landscape. This includes GUI and command line applications, but also software libraries. Table 2.2 gives an overview of the most widely used tools and software libraries for working with point cloud data.

A widely used GUI application is *CloudCompare* [51]. Initially developed to enable change detection between two point clouds [50], it now provides a wide range of algorithms for common tasks such as registration, segmentation, computing geometric features or removing noise. It also enables users to render point clouds in real-time, making use of several of the common visualization techniques discussed in Section 2.3. As one of the standard tools for handling point clouds on desktop computers, it also supports a variety of different file formats for both point cloud and raster data.

While GUI applications are great for interactivity, processing very large data is oftentimes not possible with these tools as their underlying computational model requires all data to be loaded into working memory first. Command line applications are a better alternative in these situations, as they often work with large batches of data in a stream-based manner. A common set of command line applications for dealing with LiDAR data are the *LAStools* [52]. Made up of over 40 standalone command line applications, they provide tools for working with LiDAR data in the standardized *LAS* file format [7] as well as a custom compressed file format based on *LAS* called *LAZ* [70]. Example usage scenarios include file format conversion, for example from binary *LAS* files into ASCII files, clipping and subsampling point clouds, sorting and indexing a point cloud, creating *DEMs* (digital elevation maps) [71], or removing noise from a point cloud.

When it comes to software libraries for working with point cloud data, there are two types of libraries: Those that enable access to a large variety of point cloud file formats,

Name	Storage	Specification available?	Year introduced	Usage
LAS	Binary	Yes [7]	2003	LiDAR data
LAZ	Binary (compressed with custom compression algorithm)	Only reference implementation [52]	2011	LiDAR data
OBJ	ASCII text	No	around 1990	General 3D data exchange format
PCD	ASCII text or binary or compressed binary	Yes [127]	2010	Exchange format of the Point Cloud Library [128]
PLY	ASCII text or binary	Yes [147]	1994	General 3D meshes
E57	XML and binary	Yes [66]	2011	3D imaging data
PTX	ASCII test	Only unofficial [24]	unknown	LiDAR data
BPF	Binary	Yes [101]	2015 (v3)	LiDAR data
EPT	Binary	Yes [62]	2018	Indexed LiDAR data
COPC	Binary (compressed)	Yes [60]	2021	Indexed LiDAR data
3D Tiles	Binary	Yes [31]	2019 (Open Geospatial Consortium (OGC) standardized)	3D geospatial data, including point clouds

Table 2.3: A list of common point cloud file formats

such as *PDAL* [29], and those that provide processing and analysis algorithms, such as *PCL* [128]. *PDAL* is of particular interest when trying to understand the storage landscape for point clouds, as it provides dozens of different readers for a wide variety of common file formats as well as two databases that support point data (*PostgreSQL* and *TileDB*).

Cross-referencing these tools, a reasonable overview of the landscape of point cloud file formats can be gained. The most common file formats and their capabilities are listed in Table 2.3.

While there are several data formats that are human-readable, the large data volume of point clouds makes it more important to have efficient file formats, which is why the most widely-used file formats store points as binary data as opposed to text. Compression is an important factor as well to keep the data size manageable. It is interesting to observe that on the one hand there is a consensus in the research community that data volume is a challenging aspect of point cloud data [150, 86, 132, 32], but at the same time there is no standardized file format with first-class support for point cloud compression. The *LAZ* format is the only one that implements custom compression algorithms designed for common point cloud attributes and thus achieves high compression ratios [70]. The format is widely used in the industry but there exists no official maintained standard, only the initial publication and a reference implementation as part of the LAStools [52]. More recently, the *3D Tiles* format also added support for point cloud compression through the *Draco* library [145], a general-purpose library for 3D graphics compression that also supports point clouds. At the point of writing, the *3D Tiles* format has not been widely adopted for point cloud data.

Adding compression does help for reducing the amount of storage and network traffic when working with large point cloud datasets but comes with a runtime cost that some

applications simply are not able to pay. The *LAZ* format is sometimes avoided for interactive applications due to its slow decompression speed and lack of random access [142]. We explore some of the implications of this in Chapter 3.

Besides compression support, more recently developed formats, in particular *3D Tiles*, *EPT*, and *COPC*, have built-in support for hierarchical data structures and thus are able to store indexed point clouds. The development of these formats coincides with an increase of web-based and Cloud-based point cloud applications, for which efficient access to subsets of the point cloud is required.

Summary: The main challenges of file-based point cloud storage

- *Size*: While many of the large datasets are split up into multiple files, the files themselves still might contain more points than most software can handle efficiently. Expecting the same level of interactivity as working with for example images or video is not yet possible.
- *Compression*: To reduce the data size, compression is used. *LAZ* is the de facto standard format, which achieves good compression ratios but is significantly slower to read than uncompressed data. For this reason, some applications do not support compressed point cloud data.
- *Format hell*: While most processing applications support the standardized *LAS* format, web-based applications tend to require indexed formats and do not work with raw files. These indexed formats in turn are often not supported by processing tools, requiring data duplication.

2.1.2 Databases

While databases are commonplace solutions for data storage and access in many application domains, they are not widely used for point cloud data. Adaptation of point-based data types into existing database management systems (DBMS) has been slow, with ongoing discussions whether these systems are even suitable at all for handling point clouds. While there is a consensus among researchers that the size of many point cloud datasets requires very efficient data management solutions, there is some debate whether or not the capabilities of current DBMSs are sufficient for working with large point clouds [150, 110, 32]. This is in contrast to other forms of geospatial data, such as raster or vector data, which can be handled efficiently by various DBMSs such as *PostgreSQL* (through the *PostGIS* extension [118]) or *Oracle Spatial* [143].

A comprehensive survey of the point cloud storage capabilities of various DBMSs was conducted by VanOosterom et al. in 2015 [150], with additional benchmarking results being published in 2017 [106]. In their survey, they compared the performance and usability of three DBMSs with point cloud support—*PostgreSQL* with the *pgPointclouds* extension [115], *Oracle Spatial* [143], and *MonetDB* [95]—with file-based approaches using *LAStools* [52]. Due to the large variety of different queries as well as the scale of the datasets that they evaluated, their survey has become a reference for the opportunities and challenges of DBMSs in the point cloud community. Their findings can be summarized as follows: For simple queries, or if the query granularity is at point-level, file-based solutions (in this case using *LAStools*) outperform the database solutions in terms of throughput and responsiveness. For more complex queries that go beyond simple query regions (rectangles, circles), DBMSs become the more usable alternative, but care must be taken to use an appropriate storage model. Similar findings were reported by Béjar-Martos et al. [11], though their storage model consisted of storing small *LAZ* files within the database.

Finding an appropriate storage model is one of the major challenges for storing point clouds inside a database. Storage model in this context refers to the memory layout of the point records inside the database. The simplest storage model is the *flat table* model, which refers to storing each point as a single row in a relational database such as *PostgreSQL*. This approach is simple, but has problems in terms of data size and scalability. Not every DBMS supports data types that match the efficient encoding scheme of for example the *LAS* file format, so a single point record inside a database might require significantly more memory than in a file-based solution [153]. This is precisely what VanOosterom et al. measured in their benchmarks, where a point cloud stored with the flat table storage model requires 2x to 4x the memory of the file-based solution (see Table 3 in [150]). The other problem of the flat table model is that the index can become very large since a very large number of rows of comparatively small size have to be indexed. This is also demonstrated in the same benchmark, where for both the *PostgreSQL* and *Oracle Spatial* databases, the size of the index exceeds the size of the actual point cloud files.

To deal with these problems, points are often grouped together into spatially adjacent patches. This is referred to as a *blocking* or *blocked* storage model. Both *PostgreSQL* (through the *pgPointclouds* extension) and *Oracle Spatial* support this storage model and define custom data types that represent patches of points. This solves both the storage problem (as these custom data types can use efficient memory representations similar to the *LAS* file format) and the index problem (as significantly fewer records have to be indexed). The downside of the *blocked* storage model is that single-point granularity during queries is lost, as all queries only operate on groups of points. If single-point

granularity is required, this either means that the DBMS has to unpack each patch, which is computationally expensive, or that the query result might contain points that do not match the query. This is the main reason why DBMSs are outperformed by file-based solutions for simple queries.

Even though DBMSs still do not see the same level of usage for managing point cloud data as they do for other types of geospatial data, there has been considerable progress in the research community over the last decade. Cura et al. proposed the *Point Cloud Server* as a database architecture that covers a wide range of point cloud application use cases and as such serves as a reasonable baseline for what a modern point cloud DBMS should be capable of [32]. On the storage-side, the usage of column-oriented databases has been explored in the context of point clouds, with the observation that a column-oriented memory layout reduces overhead compared to the flat table memory layout of traditional relational DBMSs [95, 150]. To deal with the indexing problem, SFCs have been explored as means to group points by multiple attributes into an efficient memory layout. Typically, SFCs are combined with Index Organized Tables (IOT) to combine data and index into a single data structure, which improves the performance of queries and data extraction [120]. It is worth noting that this approach is also used for some of the modern point cloud data formats, in particular *3D Tiles* and COPC, and is the de facto standard for point cloud data management for visualization applications [132]. SFCs have also proven useful for working with higher-dimensional point cloud data [89, 88, 86].

Summary: The main challenges of point cloud databases

- *Poor throughput*: The blocked storage model keeps the index size manageable, but since the point data is restructured internally, extracting individual points is costly, resulting in poor throughput when outputting point data from the database.
- *Slow data import*: Restructuring the data during the import into the database is significantly slower than working with raw files. This cost might amortize if the data stays in the database and is used frequently, but for explorative scenarios these wait times might be unacceptable.
- *Limited support in the tool landscape*: Few processing tools and virtually no visualization applications have native support for point cloud databases.
- *Lack of first-class LOD support*: Most DBMSs have no native support for LOD, with the recent work by Liu et al. being the notable exception [86].

2.2 Point cloud indexing

While point cloud data is simple in its structure and available in large quantities, in its raw form it is rarely used as a default data type. Problems with structuring the data, as well as applications requiring specific forms of data are two of the main reasons as discussed by Poux [119]. As a result, point clouds are often processed into what they call *application-specific deliverables*. These can be different types of data, such as triangle meshes, shape descriptors, or semantic models, but often they are optimized file formats and data layouts, which are of particular interest to us due to the implications for the data management layer. This section thus explores the current state of the art of file format and data layout optimizations with a special focus on indexing point clouds.

As we explained in Section 2.1, the large volume of point cloud datasets is a major challenge for many applications, from storing trillions of points to streaming large quantities of point cloud data over network connections with limited bandwidth, to processing datasets that do not fit into main memory. Most point cloud applications today are only possible by using highly optimized data layouts which structure the points in a way that improves overall I/O throughput and oftentimes enables efficient access to specific parts of the point cloud by creating an index.

Fixed-width binary file formats such as *LAS* [7] are an easy way to improve I/O throughput: The binary memory layout is more efficient in terms of storage, so more points can be stored using the same amount of memory compared to text-based formats. Additionally, the binary layout will typically match the internal memory layout of point records in an application more closely than text-based data would, making the parsing process more efficient. Additional speedup is possible by using columnar data layouts, as we will demonstrate in Chapter 3. Compression is another strategy that is often employed to keep the size of point cloud datasets manageable, with specialized compression algorithms such as those of the *LAZ* file format [70] achieving lossless compression with ratios of up to 10:1. The downside of compression is a significant increase in complexity of parsing the point cloud data. In terms of point throughput (the amount of points an application can process per second), compressed point cloud formats typically perform worse than uncompressed formats, even though the latter might require up to 10 times more data to be read.

Ultimately, both more efficient memory layouts as well as compression will only ever result in constant-time speedups or reductions in data size. This is where indexing comes into play: With an indexed dataset, areas of interest can be quickly identified, potentially discarding large parts of the dataset and thus drastically reducing the amount of data that has to be read and parsed. Algorithmically, an index enables searching in logarithmic time compared to linear time. With datasets comprised of billions of points, indexing is

often necessary to enable applications such as interactive visualization, which requires millisecond response times, or mesh reconstruction, which requires point neighborhood information.

Indexing is far from a novel technique and has been utilized for decades to enable databases to work with large and complex datasets. The standard textbook *Database Systems: The Complete Book* [49] defines an index *over an attribute* A as a data structure that makes it efficient to find tuples with a fixed value for A. The standard data structure for indices in database systems is the *B-tree*, a variant of a binary search tree optimized for efficient I/O access. For an in-depth explanation of B-trees, the interested reader is referred to the textbook *Introduction to Algorithms* [30]. In the context of this thesis, understanding *multi-dimensional indices* is of bigger importance than delving into the foundations of B-trees and the likes, so we will spend the remainder of this section looking at the specific requirements and techniques for indexing spatial data (which is multi-dimensional) and point clouds in particular.

2.2.1 Spatial index structures

A key limitation of binary tree variants as an index structure is their inability to work with multi-dimensional data. A binary tree requires that its values form a total order, which common data types such as integers, reals (disregarding implementation details such as `NaN`), or strings satisfy. Vectors in k-dimensional space (with $k > 1$) typically do not form a total order that is immediately usable for indexing spatial data. Consider these two vectors:

$$a : \begin{pmatrix} 1.5 \\ 0.5 \end{pmatrix} \quad b : \begin{pmatrix} 0.5 \\ 1.5 \end{pmatrix} \tag{2.1}$$

The question "Is $a < b$ or $b < a$" cannot be answered meaningfully because there is no unique total order for vectors in k-dimensional space. There are special cases, such as the lexicographic order for string comparisons, but for most spatial data, lexicographic ordering makes little sense. A common solution to this problem is to instead treat each dimension of the vector separately, comparing only the elements of this dimension. This leads to a natural extension of the binary tree (a tree in one dimension) to k dimensions, called the $k-d$ *tree* [12]. Where a binary tree splits a one-dimensional interval into two disjunct intervals at each level, a k-d tree splits each of the k dimensions successively using a hyperplane. k-d trees are one of the most widely used data structures for indexing spatial data and are used extensively in many areas of computer graphics.

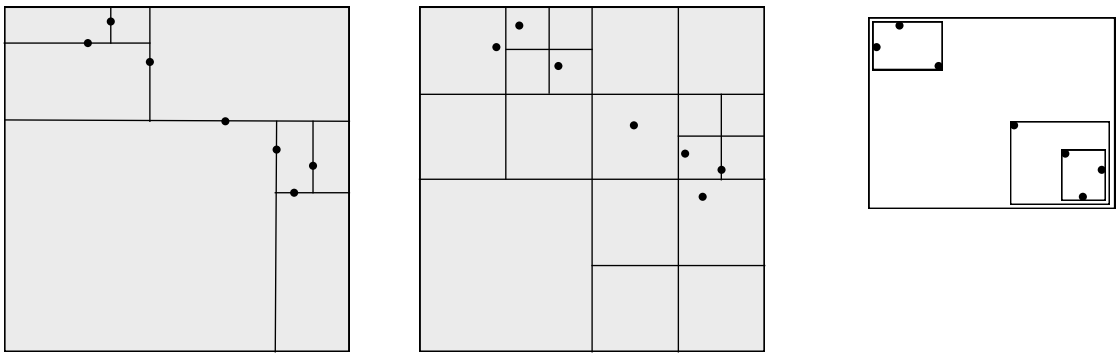


Figure 2.1: A k-d tree, quadtree, and R-tree (left to right)

One of the advantages of k-d trees is the ability to freely choose the position of the splitting hyperplane, allowing it to adjust to unevenly distributed data, keeping the tree depth shallow even for highly irregular data. If we instead go in the opposite direction and always split each dimensions exactly in the middle, we end up with another useful data structure called a *quadtree* (in 2 dimensions) or an *octree* (in 3 dimensions). In a quadtree, each new level of the tree splits its parent node into four equally-sized quadrants (or eight equally-sized octants for the octree). While their regular nature is less favorable for unevenly distributed data, the symmetry of quadtrees and octrees makes traversal algorithms simpler than for k-d trees and allows for a range of memory layout optimizations.

Both the k-d tree and the quad-/octree work by splitting space into smaller cells. The *R-tree* [57] is another spatial data structure that takes a different approach and instead tries to group adjacent elements together using their minimum bounding volume. It is similar to the B-tree in that it is balanced and optimized for I/O performance. As opposed to k-d trees and quad-/octrees, the R-tree natively supports elements that have a spatial extent themselves, for example polygons. The R-tree is a special form of a general class of spatial data structures that group data based on bounding volumes, called a *Bounding Volume Hierarchy* (BVH). Where the R-tree uses an Axis-Aligned Bounding Box (AABB), other bounding shapes such as oriented bounding boxes (OBBs) or bounding spheres can be used as well, usually resulting in a tradeoff between the depth of the resulting tree and the complexity of traversal or intersection tests. Figure 2.1 illustrates some of the spatial index structures introduced in this section.

2.2.2 Space-filling curves and their relation to multi-dimensional trees

Returning to the topic of indexing multi-dimensional data, it is still unclear how we might define a total order on multidimensional vectors. To answer this question, it is important to understand what kinds of queries spatial acceleration structures are meant to accelerate. For one-dimensional data, a query usually boils down to one or more attribute comparisons using the typical binary relations "equals", "less than", "greater than" and so on. The query "Give me all movies directed by Christopher Nolan in the 2010s" boils down to two attribute comparisons: "director EQUALS Christopher Nolan" and "2010 <= year < 2020". Each attribute comparison can be trivially accelerated by sorting this attribute ("director" and "year") using its total order and performing a binary search. A spatial query might not look so different at first glance: "Give me all restaurants which are rated with at least 4 stars and at most 100 meters away from my current location". Again we have two attribute comparisons: "rating >= 4.0" and "location WITHIN (100m, current location)". We can rewrite the second comparison in terms of more familiar binary relations: "distance(location, current location) <= 100m". A good spatial index structure will be one that can speed up queries such as the "distance" query: It enables fast lookup of records within a given area or close to a given position. Asking 'Give me all records at most 100 meters away from location X' is equivalent to asking 'Give me all records that intersect a circle of radius 100 meters centered around point X'. Since the point X is variable, we are looking for a total order that preserves neighborhood relations. Records that were close together in space should be close together within the data structure after applying the total order. This is where *space-filling curves* can help.

A *space-filling curve* is a contiguous function mapping all values of the unit interval $[0, 1]$ onto all values of a higher-dimensional region $[0, 1]^n$ for $n > 1$. Intuitively speaking, a space-filling curve is a curve reaching every point in the region $[0, 1]^n$. Inverting this mapping, we get a function that maps every point in an arbitrary region of n-dimensional space onto the unit interval. Since the unit interval is well-ordered, space-filling curves allow us to define a total order for vectors in n-dimensional space. As an added benefit, space-filling curves typically preserve locality, meaning points with a small distance in n-dimensional space will result in values close to each other in the unit interval. There are many known space-filling curves, such as the *Peano curve*, *Hilbert curve*, or the *Z-order curve*, which are illustrated in Figure 2.3.

The Z-order curve is of particular interest when it comes to creating spatial index structures. A Z-order curve is a space-filling curve that emerges as the result of sorting points based on their Morton index [100]. It can be used to efficiently create octrees, k-d trees and bounding volume hierarchies for arbitrary primitives. There is a particularly intimate relationship between Morton indices and octrees (or any of its lower/higher dimensional

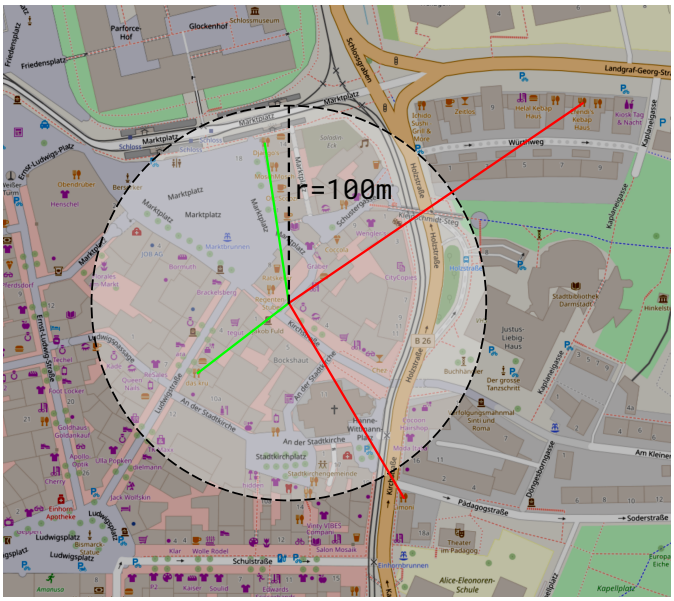


Figure 2.2: A spatial query illustrated. Map data from *OpenStreetMap* [108]

counterparts): A Morton index directly identifies a unique node within an octree. Given an axis-aligned bounding box (AABB) $B \subset \mathbb{R}^3$ and a point $p \in B$, the Morton index $M(p)$ can be calculated in linear time based on the desired number of bits in the Morton index. The number of bits and the dimensionality of the Morton index directly define a constant *depth* for the Morton index, which is equivalent to the depth of the node within an octree that the Morton index represents. Since Morton indices are numerical values, the number of bits is usually chosen in accordance to the word-size of the target processor, which for most modern systems means 64 bits. In $\mathbb{R}^3$ with three bits per level this results in a maximum depth of 21 levels. Depending on the use case, these 21 levels might not be sufficient and larger integer types might be used (such as the builtin `u128` type in Rust) to get more precision. From a practical standpoint, 21 levels is equal to a spatial resolution of $1 : 2097152$, or millimeter precision with a bounding box that is two kilometers wide. Once the number of bits has been chosen, calculating the Morton indices for a set of points and then sorting these points by their Morton index puts all points into the order of a flattened octree in memory. This is a fast way to compute an octree from a set of points, which can easily be parallelized and even extended to the construction of k-d trees and bounding volume hierarchies, as was demonstrated by Karras [74]. Besides computing an octree, sorting points by Morton index can also be used to parallelize gen-

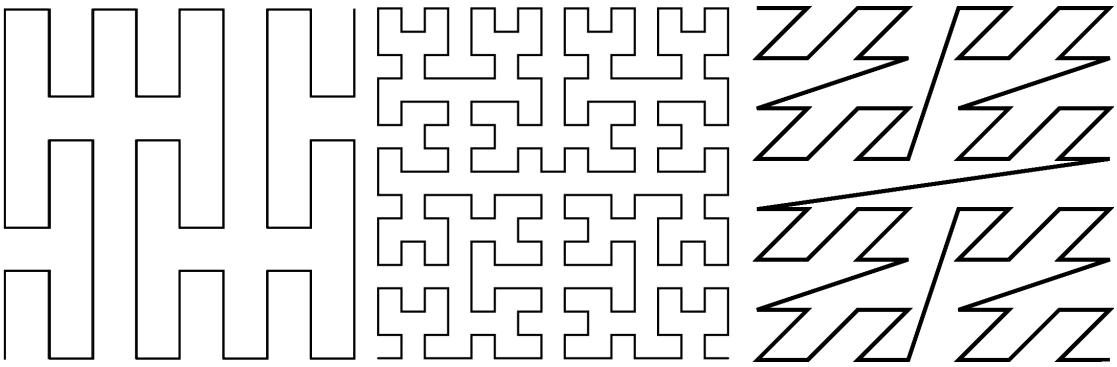


Figure 2.3: The Peano curve, Hilbert curve, and Z-order curve (from left to right)

eral point cloud processing tasks, as demonstrated by Alis et al. for a *k*-nearest-neighbor algorithm [6]. In Chapter 4, we exploit the parallelization potential of Morton indices for speeding up point cloud indexing.

2.2.3 Point cloud index structures

Besides the general-purpose work on spatial index structures, there has been considerable research to develop efficient index structures that can deal with challenges specific to point clouds. Besides data volume, which is a common challenge in many areas of computer graphics or even computer science in general, the two major challenges are good support for LOD, and indexing higher-dimensional point clouds, that is taking into account secondary attributes in addition to the point position.

Most point cloud index structures have been developed with visualization in mind. The most widely used data structures nowadays are variations of either k-d trees or octrees, though historically the early systems for visualizing point clouds used different data structures, such as the BVH variant used by the *QSplat* [126] system, generally considered to be the first system for rendering large point clouds natively. Early research quickly identified the importance of subsampling the point cloud to quickly select appropriate levels of detail, which lead to index structures such as *Sequential Point Trees* [33] or *Layered Point Clouds* [53]. The *Instant Points* system [161] extended the Layered Point Clouds structure and introduced *nested octrees*, which to this day are the foundation for many visualization-optimized point cloud index structures. In a nested octree, points are stored in both internal and leaf cells, where cells closer to the root store coarser subsamples of the point cloud. These subsamples define the different levels of detail, which can be selected adaptively during rendering, typically by considering a maximum allowed error

in screen-space. A small change in the inner structure of the nested octree—replacing inner octrees with a grid—lead to the *Modifiable Nested Octree* structure [129] which has become a de facto standard for many point cloud visualization applications. Since point clouds often have variable densities, especially when combining datasets from different acquisition techniques, nested index structures are preferable as they can adapt to these density variations [104].

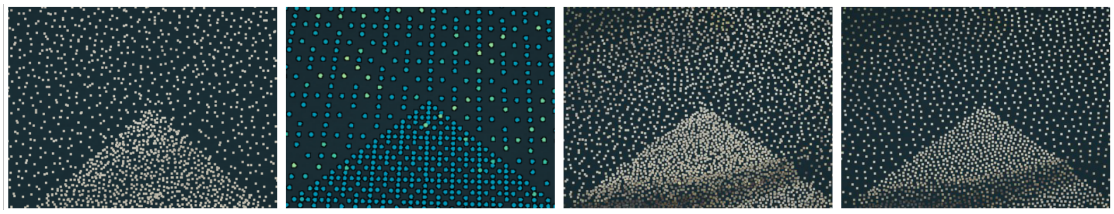


Figure 2.4: The effect of different subsampling methods illustrated visually (from left to right: Poisson-disc, grid-center, random, high-quality Poisson-disc). Images taken from [138]

Subsampling A crucial part of building a good point cloud index suitable for visualization is the subsampling process. Through subsampling, irregular sampling densities that are a natural part of the capturing process can be converted into multiple sets of evenly sampled point clouds, which results in good visual quality when using splatting. Additionally, subsampling enables LOD, which is essential for rendering large point clouds that do not fit into working or Graphics Processing Unit (GPU) memory. The sampling process itself typically works by considering all points that fall into the bounding volume of a node in the index structure (for example a cubic octant in an octree, or a cuboid in a k-d tree) and selecting a subset of these points that match some criterion. The selected points are inserted into the node of the index structure, while leftover points are distributed to the child nodes, where the process repeats. Schütz [133] explored several different subsampling strategies—the criteria by which points are selected—such as random sampling, grid-based sampling or Poisson-disk sampling. In general, these sampling strategies correspond to different types of noise, which explains their visual characteristics: Random sampling corresponds to white noise and has an irregular look, whereas Poisson-disk sampling corresponds to blue noise and gives a smoother, more even look, even with a low number of samples. Blue noise has long since been favored for various sampling-related tasks in computer graphics, such as generating antialiased images [99]. Grid-based sampling methods are more akin to a low-pass filter, as they generally only

allow at most one point per grid cell and hence impose a natural limit on the maximum frequency. Grid-based sampling typically exhibits a more regular look than both white and blue noise sampling, which, while visually less pleasing, is often simpler to compute. Figure 2.4 illustrates the visual characteristics of several common subsampling methods. Besides I/O, the sampling process is the main contributor to the computational complexity of point cloud indexing tools, a point that we explore in Chapter 4 and Chapter 5.

Continuous LOD Index structures such as the *Modifiable Nested Octree* implement a discrete form of LOD: Each node within the tree corresponds to a fixed level-of-detail, realized through a maximum sampling density which shrinks with each deeper level of the tree. During visualization, nodes are selected based on this discrete LOD and the full node is typically loaded and displayed. This leads to characteristic artifacts at the boundaries between two discrete levels-of-detail. Recently, the usage of *continuous* levels-of-detail (cLOD) has been studied by both VanOosterom [149] and Schütz [137]. Extending the discrete levels-of-detail into the real-number domain allows a more fine-grained LOD selection and hence significantly better visual quality. In practice, cLOD is often achieved by first computing an index structure with discrete LOD and then computing the continuous LOD afterwards or even at runtime during rendering.

Other point cloud index structures Besides octrees, there are also other point cloud index structures, both for visualizations and for general-purpose tasks. k-d trees have been used in a variety of point cloud visualization applications [55, 125, 123, 37]. These k-d tree implementations often make use of LOD in a similar manner to the *Modifiable Nested Octree* structure, by storing subsampled point data in interior nodes. Since k-d trees require sorting for determining the split position for a node, constructing them *out-of-core* is more complicated than constructing an octree *out-of-core*. The advantage of k-d trees is that they are generally more balanced than octrees, especially with irregular data distributions as one often finds in terrestrial LiDAR data.

Using space-filling curves, B-trees can also become a viable point cloud index structure. The *HistSFC* structure by Liu et al. [89] is one example of a recently developed index structure that utilizes B+-trees (a variant of the B-tree where data is stored only in the leaf nodes) instead of octrees or k-d trees. It is aimed at indexing higher-dimensional point cloud data, taking into account additional attributes on top of positions in 3D space, and has been demonstrated to work well with up to eight indexing dimensions [87]. It is also one of the few works that consider general-purpose queries on point clouds, beyond visualizations. The *Point Cloud Server* as proposed by Cura et al. [32] is one of the other notable contributions in this field. From a practical point of view, besides the

best effort of many researchers, it seems that there is still a divide between point cloud index structures usable for state of the art visualizations, and index structures usable for general-purpose queries. In particular, there is no demonstration as of today for a system that achieves state of the art visualizations of point clouds similar to for example the *Potree* system [133] but using a general-purpose point cloud index or a point cloud stored in a DBMS.

2.2.4 Adaptive indexing and in-situ queries

An interesting approach to indexing complex or dynamic data is that of *adaptive indexing*. Where in a traditional DBMS, an index is built over a full table with known attributes upfront, adaptive indexing methods build an index dynamically based on the queries. A widely studied approach for adaptive indexing is *database cracking* [68, 67]. The main idea of database cracking is that during a query, when a region of interest in a table is identified, the records in this part of the table are physically reordered so that the records satisfying the query are stored next to each other in memory. For a comprehensive survey of various database cracking approaches we refer to Schuhknecht at al. [131]

The advantages of adaptive indexing techniques over regular indexing are potential improvements in performance (as only the data that is queried is actually indexed, as opposed to all data) and increased flexibility (as the attributes to be indexed can be derived from the queries). As both the volume and the variety of data increase, performance and flexibility become key challenges for modern database systems. Performance is often achieved by adapting the data to optimized formats dictated by the inner workings of the database engine, which comes at the cost of flexibility [73]. Additionally, importing data into a database—thus converting it into the optimized internal format—often results in significant initialization overhead. This is problematic for explorative scenarios where users want to understand large or complex datasets and quickly identify areas of interest, which requires high query responsiveness. Waiting for minutes, hours, or even days for a data import, only to find out that 99% of the data are uninteresting is frustrating and a waste of compute resources. Thus, the field of *in-situ data exploration* has gained interest in recent years [4, 13, 105, 73, 39]. *In-situ* refers to working with the raw data as-is, for example large *CSV* or *JSON* files. Systems that enable in-situ data exploration will work natively with these raw data formats and execute queries on the raw files instead of an optimized copy of the data in some internal format. While adaptive indexing methods such as database cracking solve some of the problems of in-situ data exploration, namely getting rid of the initialization cost, they still require some form of data import into an internal format. The *NoDB* system [4] was one of the first attempts at combining adaptive indexing with in-situ queries on raw files. Since then, both adaptive indexing and in-situ

queries have seen significant developments. Trying to paint a full picture of this rather large area of research is beyond of the scope of this work, and unfortunately to the best of our knowledge, there exists no recent overview study of the current state of the art of both adaptive indexing and in-situ queries. Instead, we will briefly describe what we consider to be the most important developments that our contributions to in-situ processing of point clouds are based on (as explained in Chapter 3).

Progressive indexing as introduced by Holanda et al. [64] deals with the challenge of keeping query response times low. Since in-situ systems work on raw files, they do not have the luxury of highly optimized internal data formats, which often results in worse query response times than regular DBMSs. On top of that comes the additional overhead of creating parts of the index during query execution. Progressive indexing introduces a dynamic budget for queries that they can spend on index creation. By adjusting this budget based on incoming queries, interactive query response times are possible.

Spatial database cracking is an approach for applying adaptive indexing to spatial (i.e. multi-dimensional) data [63, 83] by adaptively generating common spatial index structures such as k-d trees or quadtrees. Dealing with spatial locality requires different algorithms compared to indexing one-dimensional data and techniques such as database cracking do not trivially generalize to higher dimensions.

Visual exploration is an important strategy for exploring and understanding complex data. Its usage has also been explored in the context of in-situ systems, though primarily for two-dimensional exploration through various graphs and plots [13, 94]. What sets visual exploration apart from other exploration techniques are the specific queries that are required, in particular selecting the data corresponding to the current view region. In the 2D case, this corresponds to a two-dimensional range query (such as the *render* query in [13]), but moving into three dimensions or artificially reducing the data with LOD techniques will yield more complex queries not typically found in other explorative data scenarios. Efficiently handling LOD is one of the main challenges when working with point cloud data and none of the existing in-situ data exploration approaches support LOD.

Summary: The main challenges of point cloud indexing

- *Slow processing*: Indexing takes a long time and is computationally expensive due to the subsampling process. The resulting query latency makes these approaches ill-suited for explorative scenarios, where preprocessing time becomes part of the query latency.

- *Specialization*: The most popular index structures are application-specific, causing

either a lock-in to a specific system, or requiring costly reprocessing and data duplication when migrating systems. This is rarely due to fundamental incompatibilities and more often a technical limitation as cross-system compatibility is simply not a focus of application developers.

- *Visual quality and LOD support*: Most visualization systems require high-quality indices to achieve good visual quality. General-purpose indices are often not sufficient because they lack the necessary subsampling to achieve good LOD support.

2.3 Point cloud visualization

Visualizing point clouds has been an active field of study over the past two decades. A real-time visualization of a point cloud can be used for entertainment purposes as well as explorative analysis. “Let’s have a look at the data” is often one of the first things that users say when getting access to a new dataset, point clouds being no exception there.

One of the earliest point cloud rendering systems was *QSplat* [126]. Assuming that the point cloud is evenly sampled, splats are small primitives (for example quads or ellipsoids) that approximate the local surface of a point cloud that give the illusion of a closed surface. To deal with the challenge of data volume, the *Layered point clouds* approach was introduced [53], the first system to support multi-resolution point clouds. Layered point clouds are the predecessor of the *Modifiable Nested Octree* index structure introduced in Section 2.2.3 and their novelty was the support for LOD in point cloud rendering. Based on the required amount of precision, nodes of different resolutions can be rendered at the same time, allowing for significantly larger datasets to be rendered interactively.

Tree-based acceleration structures also allowed the simplification of the splat rendering process. Where in the earlier works such as *QSplat*, splat rendering required per-point normals, rendering screen-aligned quads with a size based on the resolution of the corresponding node in the acceleration structure became a viable alternative [55]. Today, most systems that visualize point clouds do so by rendering screen-aligned quads, typically manipulated in the fragment shader to give more visually pleasing results. An example for this is the paraboloid rendering proposed by Schütz [133], though three-dimensional splats have been proposed as well [130]. Visual quality is further enhanced by the subsampling process during the creation of the acceleration structure, as explained in Section 2.2.3. Postprocessing algorithms in screen-space, such as screen-space ambient occlusion (SSAO) [10] or eye-dome lighting (EDL) [20], are also frequently used to improve the image quality of point cloud renderings.

Virtual reality (VR) systems face additional challenges compared to desktop-based ren-

dering solutions, in particular higher frame-rates and larger point budgets due to an increased field-of-view [37]. On top of that, LOD-based artifacts become more noticeable, such as points suddenly appearing during the transition between different levels-of-detail. Techniques such as continuous LOD can help mitigate these artifacts [137].

Beyond improvements in visual quality, there have been several approaches that utilize computer shaders for the visualization of point clouds, which improve the interactivity of point cloud rendering. Progressive rendering of unindexed datasets [142] is one approach that reprojects the pixels of the previous frame into the current frame and fills holes using random points selected from a shuffled vertex buffer created by a compute shader. Other approaches forego rendering using the traditional GPU hardware pipeline altogether and render point clouds exclusively using compute shaders [135, 136], which can result in higher frame rates.

Summary: The main challenges of point cloud visualization

- *Visible subset selection*: LOD artifacts are still a challenging problem, and while cLOD has been proposed and implemented in some systems, its usage still requires substantial preprocessing or is limited to datasets that fit into GPU memory.
- *Handling* out-of-core *and remote data*: For data that fits into GPU memory, progressive rendering is a viable solution with good interactivity and response time, but no similar solution exists that supports *out-of-core* data or data that is streamed from a remote server.
- *Specialization*: The fastest visualization frameworks require highly specialized data formats and memory layouts to achieve the necessary throughput. Compressed data formats are oftentimes ignored because the overhead of decompressing the data is too large for the desired level of interactivity.

2.4 Implications for point cloud data management as a whole

We saw that there are a wide variety of requirements that applications pose to the point cloud data management layer. From scalability in terms of the amount of data that can be stored, to the runtime of indexing processes, point cloud data volume is far from being a solved problem. Applications want to query point data by an increasing number of attributes, requiring flexible data management that can deal with queries on arbitrary attributes. This in turn has implications for the indexing process, which today is still an

either/or decision: If high fidelity visualizations are required, use one data representation with built-in LOD support, otherwise use a DBMS. Neither approach serves both the visualization and the analysis domain. We saw that in the wider database community, adaptive indexing is a promising approach for dealing with large data quantities and highly variable query patterns, so its application to point cloud data seems promising to us. For structuring the data quickly and efficiently, parallelization also plays an important role and we believe that it is mandatory for a good point cloud data management solution to make efficient use of the available hardware through parallelization. In the end, point cloud data management is largely driven by performance concerns. Here we believe that columnar memory layouts have large untapped potential for performance improvements. There has been limited research on their application for point cloud data management, but they have been successfully applied in the video game industry as a performance optimization technique [98, 122], so we are interested in verifying if this is a technique that transfers well to the point cloud domain.

3 Ad-hoc queries: An approach for efficient point cloud data management without preprocessing

"Most sapients confuse working hard with being miserable."

Becky Chambers, A Closed and Common Orbit

In this chapter, we develop strategies for working with unindexed point cloud data by executing queries on raw files in common formats such as *LAS*. We call such queries *ad-hoc queries* and show that commodity hardware is capable of answering many common point cloud queries in a manner of seconds for datasets of up to a billion points without requiring any preprocessing. The work in this chapter is based on our publication "Working with large, unindexed geospatial data" [17] as well as its extended journal version "Executing ad-hoc queries on large geospatial data sets without acceleration structures" [18] and answers the following research question:

RQ1 Can *ad-hoc queries* enable applications to work directly with raw point cloud files instead of sophisticated index structures?

This chapter starts with the motivation for working with raw point cloud files in Section 3.1. In Section 3.2, we gather a set of representative query types from various point cloud applications. Section 3.3 introduces our concept of *ad-hoc queries* on raw point cloud files, as well as our reference architecture for a system for executing such queries. In this section we also show potential improvements for increasing the response time and throughput of *ad-hoc queries*. These improvements include two new file formats that use a columnar memory layout for increased performance, as well as an adaptive indexing algorithm that can be executed while processing an ad-hoc query. We evaluate our ad-hoc query system and the new file formats in Section 3.4. Section 3.5 contains a critical discussion of the results and answers research question 1.

3.1 Motivation: Benefits of working with unindexed point cloud data

In chapter 2, we saw that raw files are the predominant point cloud data organization medium: Large datasets are often provided as raw files on a remote server for download (such as [109] and [2]), tools such as *LAStools* [52] were developed specifically for working with raw point cloud files, and many point cloud libraries contain parsing code for raw files. They are ubiquitous, but not as expressive as data stored in a database, or as optimized for throughput as the point cloud index structures used by visualization applications. Importing a point cloud into a database or generating a visualization-optimized index is time- and resource-intensive and typically results in a duplicate of the dataset. If neither the data nor the use case is expected to change, this approach works well, as the cost of the initial data restructuring is amortized over the utilization period of the data. Once the data changes more frequently, or the use case (and hence the data access pattern) is not known upfront, neither databases nor custom index structures are flexible enough. Explorative and interactive data analysis are examples that exhibit these characteristics, and novel adaptive indexing techniques are actively researched to deal with these challenges [64, 67, 4, 73].

While adaptive indexing has been successfully applied to regular databases and the data stored therein, point clouds have special characteristics that make the application of adaptive indexing more challenging. Adaptive indexing for spatial data is still an ongoing research topic and no existing technique takes LOD into account, which is a must for large point clouds. At the same time, adaptive indexing would bring substantial benefits for the way we work with point cloud data: File-based query systems such as *NoDB* [4] or *RAW* [73] could increase the query expressiveness when working with raw point cloud files while maintaining decent interactivity. Progressive indexing [64] would reduce the preprocessing time in those cases where an index is required to achieve the desired level of interactivity. Together, these techniques would pave the way for true interactive exploration of point cloud data and would save cost by only processing the data that is actually required.

Nonetheless, we do not believe that adaptive indexing will fully replace traditional index creation. Instead, the two approaches could supplement each other: Quickly identifying interesting or relevant data by using file-based query systems and adaptive indexing, followed by a traditional indexing approach to generate optimal data structures on the selected subset of the data.

The main challenges for working with unindexed point clouds are the size of the data and the complexity of the queries. Even simple queries might require a full scan of poten-

tially billions of points. For interactive visualizations, users might expect query response times of a second or less, for general-purpose queries runtimes of more than a few seconds might be unacceptable. Note that these are numbers based on subjective experience as well as general-purpose research such as the widely-cited response time limits by Nielsen [102]. We discuss the challenges with obtaining more specific numbers in Section 3.3.1. Based on these numbers, a simple back-of-the-envelope calculation illustrates the challenge: A billion points in *LAS* point format 0 take at least 20GB of memory. At the time of writing, a high-class SSD such as the Samsung 990 Pro, which is connected through PCIe 4.0, achieves a peak read speed of about 7.5GB per second. Simply loading the whole file into memory takes about three seconds, and this is before any computations have been applied to actually answer the query. At the same time, if we were to achieve data throughputs of about this magnitude, we might expect a general-purpose query on a one-billion points dataset to have a response time of less than ten seconds. Based on our initial estimates, this would be fast enough for general-purpose applications, but too slow for interactive visualizations, though one might argue that the latter depends on the user and the *perceived* interactivity (the time until something becomes visible, rather than the full query time). Sticking with these estimates, answering the question which queries, if any, have similar or better response times when applied to unindexed point cloud data will help answer research question 1 and be an indicator for how viable raw, unindexed files are as a point cloud data management solution.

3.2 Queries

In order to apply adaptive indexing approaches to point cloud data, we have to understand the types of queries posed by typical point cloud applications. These queries dictate the data access patterns and hence the structure of optimized point cloud data layouts, such as the visualization-optimized indices introduced in Section 2.2.3. While we cannot hope to cover all possible queries that any point cloud application might use, there are many specific types of queries stated explicitly in the literature around point cloud database systems [150, 106, 32, 89, 87, 86, 39]. Table 3.1 gives an overview over the most common types of queries found in the literature.

Notably, most queries are *spatial* queries and operate on the point positions. In practice, spatial queries are often combined with queries on secondary attributes, for example to identify buildings within a given area or vegetation along a road. Computationally more complex are queries on higher-level attributes such as geometric features or importance values (the more general term for LOD), as these values have to be calculated from the lower-level attributes present in the data and typically require neighborhood information.

Description	Example	Required attribute(s)
Bounding volume	Select all points within a bounding box	Position
Shape intersection	Select all points along a road defined through a polygon	Position
Distance	Select all points at most N meters away from a given point	Position
k-nearest neighbors	Select the *k*-nearest neighbors for point X	Position
Object class	Select all points that are classified as buildings	Classification
Return numbers	For all points that have multiple returns, select the first return	Return number and number of returns
LOD	Select all points with LOD level X	Importance
Density	Select the average point density for the given area	Position
View frustum	Select all points intersecting the given view frustum AND with the given LOD level	Position and importance
View frustum (cLOD)	Select all 4D points (X,Y,Z,Importance) intersecting the given cLOD view frustum [86]	Position and importance
Geometric features	Select all points with planarity greater than X	Covariance matrix

Table 3.1: Types of queries that a point cloud application might use

Per-point neighborhood information (typically in the form of the *k*-nearest neighbors of the point) can itself be calculated through a query. As a result, higher-level queries are typically nested queries. Looking at the potential number of point record accesses by these queries, we see that the simpler queries have linear complexity, whereas more sophisticated queries have quadratic complexity. This is the reason why (spatial) indices are typically used, as they reduce the complexity of nested queries from quadratic to log-linear.

For *ad-hoc queries*, anything beyond linear complexity regarding the number of point record accesses will probably be out of reach for larger datasets, which is why we exclude such queries from our analysis. This is a limiting factor which we discuss in Section 3.5.

In terms of data access patterns, we see that many queries benefit from spatial locality in the data. As an example, if a point matches a bounding volume query, a spatially adjacent point is likely to also match. While we want to work with data *in-situ* for our ad-hoc query system, meaning using data on the local machine with as little preprocessing as possible, there are properties of certain types of point clouds that result in locality that an ad-hoc query might exploit. As Cura et al. point out, the nature of LiDAR scanners

result in implicit locality between consecutive point records [32].

Lastly, we want to point out that even the more complex queries rarely use more than two or three attributes at the same time. This fact has been noted by El-Mahgary et al., who propose splitting up larger point cloud files into smaller files containing only points with a specific attribute value [39]. In principle, this approach already constitutes a rough index and requires data restructuring, which we want to prevent within our ad-hoc query system. Instead, we will investigate how column-oriented memory layouts could achieve similar effects. Grouping point attributes into contiguous memory regions is similar to grouping points into files by attribute. While the popularity of the *LAS* file format for LiDAR data is undisputed, we believe it is important to investigate how applications might benefit from file formats with alternative memory layouts and whether there are incentives to support such formats. There is currently one point cloud format (*3D Tiles*) which natively stores data in a columnar memory layout, however it has poor support for a larger number of point attributes, as they are commonly found in *LAS*. For this reason, we do not think it is a suitable format for storing LiDAR data.

3.3 Methodology

We want to evaluate the query throughput and responsiveness for the queries introduced in Section 3.2 when executed *ad-hoc* on raw, unindexed files. These *ad-hoc queries* in principle work with any file format and could be integrated into any point cloud application as a data management solution. Therefore it is important to identify those types of queries for which throughput and responsiveness are sufficient for their respective applications.

From the back-of-the-envelope calculations in Section 3.1, it is clear that *ad-hoc queries* will require substantial computational resources. For this reason, we focus on single-user scenarios, as can be found in the scientific community where individuals often perform interactive exploration of data.

We assume a data processing model where an *ad-hoc query engine* acts as a data provider and filter between raw point cloud files and a higher-level application (see Figure 3.1). This processing model has the advantage of being non-intrusive, requiring little to no changes to the applications while being able to work with the raw files as is. From an engineering perspective, we believe that this is a more versatile approach than the typical standalone, black-box systems that are often developed both in the industry but also within the scientific community. As an example, the two web-visualization frameworks *Potree* [133] and *CesiumJS* [25] both require metadata files describing the structure of the indexed point cloud, thus implicitly assuming that the data has been indexed before.

This is also what makes the adoption of point cloud databases for visualization applications difficult, as the internal index structure is typically not propagated to clients of the database. The cLOD query described by Liu et al. [88] is an example that shows that point cloud visualizations using index-agnostic queries are at least theoretically possible. An open question that is not investigated by them is how to deal with redundancies between queries for adjacent camera positions. Visualization applications using tree-based accelerators solve this problem by querying for visible tree nodes on the client using the aforementioned metadata. To emulate such queries, we can use bounding-volume queries as well as on-the-fly LOD calculation, both of which are supported by our ad-hoc query engine.

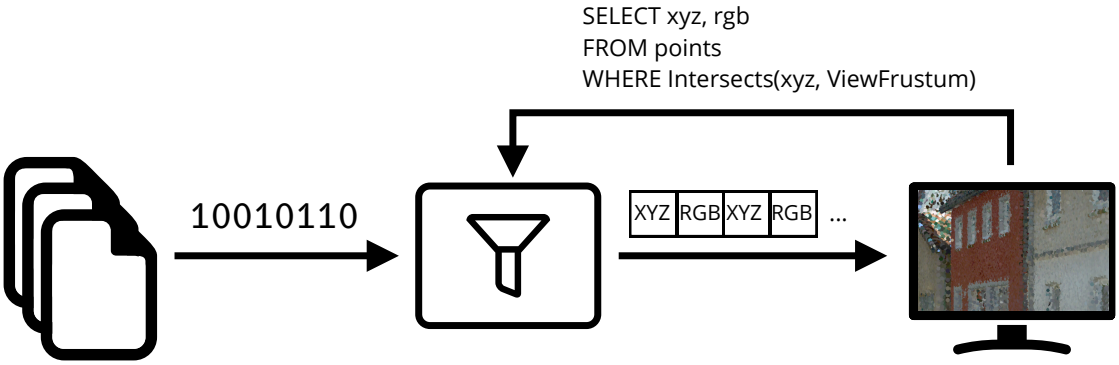


Figure 3.1: Point cloud data management using an *ad-hoc query engine* (middle) that connects applications (right) to raw point cloud files (left). Raw data is read and processed by the query engine and sent to the application in a structured form.

Beyond simply executing *ad-hoc queries* and measuring their throughput and responsiveness, we would like to gain a deeper understanding about which parameters of the point cloud data have the biggest influence on query throughput and responsiveness. Research question 1 deals with the possibility for using *ad-hoc queries* to extend the usefulness of raw point cloud files as a data management solution, which requires understanding these influences. To this end, we will answer more in-depth questions, namely:

- Which factors influence the query throughput and responsiveness of *ad-hoc queries*?
- Are columnar memory layouts preferable for *ad-hoc queries*?
- Does adaptive indexing improve the responsiveness of *ad-hoc queries*?

To answer these questions, we perform a case study where we execute a broad range of queries on common point cloud datasets varying in size and number of attributes. For each dataset, we compare the ad-hoc query approach on different point cloud file formats, both uncompressed and compressed, with queries executed using a DBMS.

We first develop a theoretical model for ad-hoc query throughput and responsiveness in Section 3.3.1, which can be used to estimate limits for throughput and response times on commodity hardware. These limits then guide the design of an ad-hoc query engine, explained in Section 3.3.2. We develop several optimizations when facing I/O-bound queries (Section 3.3.3) and compute-bound queries (Section 3.3.4), such as two new file formats using columnar memory layout. Section 3.3.5 introduces the software library *pasture* that we developed specifically for the implementation of high-performance point cloud applications and which was used for implementing the ad-hoc query engine using the aforementioned optimizations. Lastly, we discuss applications of adaptive indexing strategies which can be used to improve *ad-hoc queries* on compressed point cloud data (Section 3.3.6).

3.3.1 Measuring the performance of ad-hoc queries

To perform adequate measurements, we must first define the terms *query throughput* and *query responsiveness* more rigorously. The definition is especially important for point cloud data since the size of a query response will typically be quite large, easily going into the millions. As a result, the time to *transfer* the data from the query engine to the target application becomes relevant. Additionally we will define the two terms I/O-bound and compute-bound, as we use them a lot to refer to certain kinds of queries. Lastly we develop a theoretical model for the limitations of *ad-hoc queries*.

Query throughput Throughput can either be the number of points *processed* during a given time interval, or the number of points that are *outputted* during a given time interval. For *ad-hoc queries*, we define throughput as *the number of points processed during a given time interval*, as the alternative interpretation is not independent of the number of matching points, causing queries with few matches to have lower throughput, which is counterintuitive. Nonetheless, the time it takes for outputting the data from the ad-hoc query engine to the actual application is important, and we include this fact into the design of the ad-hoc query engine described in Section 3.3.2. When we talk about *processing a point* in the ad-hoc query engine, this refers to the time from reading the point from the raw file until the point is outputted to the target application. Effectively, this includes I/O time into the throughput, which is important for point clouds, where a single datum is small and millions of matches per query are to be expected.

Query responsiveness Responsiveness is typically defined as the time from issuing the query by the application to receiving the query response in the application. A more fine-grained approach to responsiveness is to instead measure the time from issuing the query until the *Nth percentile* of the query response has been received. For stream-based and interactive applications, it is often more important how quickly *any* data from the response is received than how long it takes for all data to be received. Visualizations are a prime example, where there are typically diminishing returns in visual quality when rendering more primitives beyond a certain threshold, which is the reason why LOD is often used.

Another important consideration is whether or not preprocessing time should be included in the responsiveness of a query. This depends on the use case and the expected rate of data acquisition. Since we see *ad-hoc queries* as a tool for explorative data analysis, we follow the definition of *data-to-query time* by Alagiannis et al. [4] and include preprocessing time as part of the responsiveness.

The terms *I/O-bound* and *compute-bound* We use the term *I/O-bound* to describe a situation in which the reading or writing of the data from an external storage medium (harddrive or network) is a bottleneck, mainly because the system could process more data than it can read from or write to the external storage. The opposite situation is called *compute-bound*, which is an umbrella term for situations where the available computational resources (CPU or GPU cycles) are the bottleneck. There is a third term, *memory-bound*, which refers to situations where the speed or sometimes capacity of working memory is the bottleneck. When we talk about *compute-bound* queries, this includes the term *memory-bound*. We discuss implications of this in Section 3.5.

Theoretical limits To understand the theoretical limits of ad-hoc query performance on point clouds, we use a simple mathematical model that describes query throughput and responsiveness in terms of known parameters, such as the number of points in a dataset, disk I/O throughput, and the complexity of the query. In the following formulas, we use lower-case letters to denote time quantities and upper-case letters for throughput quantities.

Assuming a basic model of an ad-hoc query system that consists of the steps *reading point data from disk*, *matching point data against query*, and *writing matching points to target application*, an estimate for the runtime $t(Q)$ for a query Q in this system can be given as follows:

$$t(Q) = t_{read}(Q) + t_{match}(Q) + t_{write}(Q) \tag{3.1}$$

From there, we can calculate the throughput $T(Q)$, which is the number of points

evaluated by the query ($\#points$) divided by the query runtime:

$$T(Q) = \frac{\#points}{t(Q)} = \frac{\#points}{t_{read}(Q) + t_{match}(Q) + t_{write}(Q)} \tag{3.2}$$

The throughputs for reading, matching, and writing can be expressed in terms of their individual runtimes:

$$\begin{aligned} T_{read}(Q) &= \frac{\#points}{t_{read}(Q)} \\ T_{match}(Q) &= \frac{\#points}{t_{match}(Q)} \\ T_{write}(Q) &= \frac{\#points}{\text{MatchRatio} * t_{write}(Q)} \end{aligned} \tag{3.3}$$

While the read and match stages will process the full range of points, the write stage only writes matching points, so we have to introduce a factor MatchRatio to get the correct throughput value. Combining Equation (3.2) and Equation (3.3), we get an equation that relates the overall query throughput to the individual throughputs:

$$T(Q) = (T_{read}(Q)^{-1} + T_{match}(Q)^{-1} + (\text{MatchRatio} * T_{write}(Q))^{-1})^{-1} \tag{3.4}$$

In practice the three steps reading, matching, and writing can be parallelized using pipelined processing, which gives a simpler formula for the overall query throughput:

$$T(Q) = \min\{T_{read}(Q), T_{match}(Q), \text{MatchRatio} * T_{write}(Q)\} \tag{3.5}$$

Since $MatchRatio$ will always be between zero and one, a more conservative upper bound for the throughput can be stated as:

$$T(Q) = \min\{T_{read}(Q), T_{match}(Q), T_{write}(Q)\} \tag{3.6}$$

Note that this formula is independent of the actual query result and thus matches our initial definition for query throughput.

The input throughput $T_{read}(Q)$ can be expressed as follows:

$$\begin{aligned} T_{read}(Q) &= (T_{in}(Q)^{-1} + T_{decode}(Q)^{-1})^{-1} \\ T_{in}(Q) &= \frac{\text{DiskReadSpeed}}{\text{BytesPerPoint}(Q)} \end{aligned} \tag{3.7}$$

$T_{decode}(Q)$ refers to the number of points that can be decoded from the input data into an in-memory format that can be used for answering the query. It depends on the type of query as the query might not require all attributes of the point record, so only a subset of the data might have to be read and decoded. In contrast to Equation (3.6), Equation (3.7) assumes that disk I/O and decoding happen sequentially instead of in parallel, which reflects the way that point cloud parsing code is usually written.

The output throughput $T_{write}(Q)$ can be stated in a similar way:

$$\begin{aligned} T_{write}(Q) &= (T_{out}(Q)^{-1} + T_{encode}(Q)^{-1})^{-1} \\ T_{out}(Q) &= \frac{\text{WriteSpeed}}{\text{BytesPerPoint}(Q)} \end{aligned} \tag{3.8}$$

$T_{encode}(Q)$ refers to the number of points per second that can be encoded into the format that the target application expects. We use the more general $WriteSpeed$ as opposed to $DiskWriteSpeed$, since point data might be written to either a file on disk, piped into another application, or written to a socket, depending on the target application.

From these formulas, we see that there are several contributing factors for the query throughput:

- The I/O throughput of the machine, which not only includes how fast point data can be read from disk, but also how fast it can be written to the target application. Consequently, using high-throughput I/O devices and system calls (such as `mmap`) should improve query throughput.
- The (average) size of a single point record within the input file format. The smaller the point records, the more points can be read during a given time period. This favors compressed file formats, as they are expected to require fewer bytes per point as uncompressed formats.
- The similarity of the binary layout of the point records within the input file format to the binary layout of point records in working memory. This favors fixed-width, uncompressed formats, as their decoding overhead is smaller than that of compressed files. Intuitively, if the point record within the file has the same binary layout as the composite type used in the code (for example a C `struct`), decoding the point data is equal to a static type cast of the memory, which is typically a no-op.
- The similarity of the binary layout of point records in working memory to the requested layout of the target application. The more closely these layouts match, the less encoding work is required for writing the matching points to the output. This

has interesting implications for the internal memory layout for points in the ad-hoc query engine, as the target application ideally dictates in which layout the points are read from the input files. This explains why $T_{encode}(Q)$ is dependent on the query Q.

Using these formulas, the experimental verification will not only result in empirical values of $T(Q)$ for various queries, but is expected to give insights into internal parameters such as $T_{encode}(Q)$ and $T_{decode}(Q)$. Using these parameters, we are able to quantify the effect that compressed data formats have on query throughput.

For query responsiveness, finding a reasonable formula is harder as the responsiveness depends heavily on the distribution of the points with regard to the queried attributes. To illustrate this, assume that a query Q matches 1% of the points of a given dataset. If all of these points happen to be located at the start of the point cloud file(s), a sequential scan will yield the full query response after scanning only the first one percent of the data. Conversely, in the worst case the matching points will be stored at the end of the file(s) and the query will yield a response only after scanning the full dataset. Even more interesting is the case where one part of the matching points is located at the start of the files, and the other part at the end of the file. This is why the *Nth-percentile responsiveness* $r_N(Q)$ can give more interesting insights. The *50th-percentile responsiveness* $r_{50}(Q)$ is equivalent to the average case responsiveness if we assume an even distribution of the matching points over the whole dataset. Under this assumption, query responsiveness becomes a function of the query throughput:

$$r_{50}(Q) = \frac{\#(points)}{2T(Q)} + \frac{t_{pre}}{\#(queries)} \tag{3.9}$$

$\#(points)$ is equal to the number of points in the dataset, so we assume that 50 percent of the matching points of the query have been found after half of the dataset has been scanned. As we explained in the definition of query responsiveness, we also include any preprocessing time t_{pre}, weighted by the number of queries $\#(queries)$. This way, preprocessing time amortizes the more queries are issued on the data.

These formulas can also be applied to understand queries on indexed data, as the following two examples illustrate:

Example 1 is the *Potree* application and how it queries the point data for the current view frustum, using the file format described in [138]: Based on the structure of the *Modifiable Nested Octree*, which is loaded as a list of existing octree nodes, the viewer performs view-frustum culling on the client, which results in a number of visible nodes. For each node, a "query" is issued to a file server, which simply corresponds to loading the range of memory corresponding to that node. Looking at the formula for the query

throughput, we see that $T_{match}(Q)$ is infinite, as no matching has to take place. As a consequence, no data decoding has to take place, so $T_{decode}(Q)$ also becomes infinite. Lastly, as *Potree* performs file format decoding on the client, the file server has to perform no encoding, so $T_{encode}(Q)$ also is infinite. Infinite terms can be ignored in a min statement, so the resulting throughput is:

$$T(Q) = \min\{\frac{DiskReadSpeed}{BytesPerPoint}, \frac{WriteSpeed}{BytesPerPoint}\}$$

As *Potree* is a web-based point cloud viewer, $WriteSpeed$ will be the network throughput from the file server to the client, so the query throughput while rendering points with *Potree* is bound by either disk or network I/O.

For example 2, consider a query for all points within a target polygon using a point cloud DBMS, similar to the queries used in the benchmark by VanOosterom et al. [150]. These queries run in two stages: First, a rough query using the index identifies potentially matching ranges of points, then a fine query matches each point against the query. Since the rough query reduces $\#(points)$—the number of points that the fine query has to match—the query response time improves compared to a sequential scan, even if the query throughput is similar to a sequential scan.

Throughput and responsiveness numbers in the literature If we look at numbers for query throughput and responsiveness in literature, specifically work on point cloud databases and visualization, we find a wide variety of numbers that are difficult to relate to one another. This has multiple reasons: Experiments are conducted using different hardware, different datasets and different queries. There is no agreed-upon set of queries and datasets for a standardized benchmark. The work by VanOosterom et al. [150] was a step in that direction, though they—as many other researchers—were more interested in the scalability of their solution than establishing a standardized benchmark. So while we cannot hope to achieve a direct comparison, the numbers should still give an estimate on what is considered to be acceptable in terms of throughput and responsiveness.

VanOosterom et al. report 100th-percentile query response times of about one second for small- and medium-sized bounding volume queries, a few tens of seconds for larger polygons, and anywhere from one minute to one hour for municipality-sized shapes for a dataset containing over 600 billion points using a high-performance server rack. They do not state throughput numbers, and without knowing the number of points from the rough query stage, we cannot reconstruct these numbers. Cura et al. report only throughputs for various data export scenarios, which range between approximately 0.1 to 1 million points per second [32], and they state their dissatisfaction with these numbers. Liu et

al. report response times for various queries, in particular perspective view queries with a response time between 0.8 to 1.4 seconds for selecting about 150.000 points from a dataset of two billion points[87], but no throughput numbers are given.

These numbers give little insight into more user-centric questions, such as “What are acceptable response times for point cloud queries?” and more specifically “How does point throughput affect the query runtime?”. Regarding the seminal works on point cloud rendering, there is seldom any mention of the performance of the view frustum queries at all [53, 133, 55, 137]. Instead, common metrics are frame times and total number of points rendered per frame, as well as qualitative demonstrations of the visualizations using image comparisons. A notable exception is the work on progressive point cloud rendering by Schütz et al. [142], which states how many frames it takes their system to converge to the highest-quality representation of the point cloud (less than a second in most cases for datasets up to a billion points, plus up to half a minute for loading the data from disk). Their work also illustrates the impact that point throughput has on the overall system performance, with up to an order of magnitude difference between data stored in *LAS* format versus data stored in a GPU-optimized columnar data format.

In conclusion, we are left with a somewhat unsatisfying landscape of performance measurements that are hard to relate to each other. Based on the available data, the state of the art in point cloud querying seem to be response times in the low seconds for up to a few million matching points, with relative independence of the actual size of the dataset. For the largest datasets, being able to issue any queries at all seems to be a success in and of itself. In addition, achieving high point throughput is a desirable but challenging goal for visualization applications.

3.3.2 Design of an ad-hoc query engine

In this section, we explain the design of a prototypical *ad-hoc query engine*, a software component that sits between applications and raw point cloud files. It acts similar to a typical DBMS, but operates on raw files instead of an optimized internal memory layout. The goal of designing and implementing such a system is the evaluation of the queries stated in Section 3.2 that are at least theoretically feasible based on raw, unindexed data. As stated previously, this excludes queries with anything higher than linear complexity in terms of the queried points.

The design of this ad-hoc query engine is driven by several insights and constraints:

1. Minimize data-to-query time by operating directly on raw point cloud files on the local file system, while allowing progressive index creation when necessary

2. Support common point cloud file formats and take their binary layout into consideration when executing the queries

3. Output point data in a flexible but realistic manner that includes the overhead of point data I/O

4. Support a wide range of possible queries based on the common query types introduced in Section 3.2

Based on these factors, our ad-hoc query engine is comprised of three main layers: An *Input Layer* for loading point data from raw files, a *Query Layer* responsible for executing the queries, and an *Output Layer* for sending matching point data to the target application.

The purpose of the input layer is to access the raw files in an efficient manner and load only relevant data into main memory. Since one of the assumptions that we made is that the point cloud data resides on local storage, we can use optimizations such as memory-mapped I/O. The input layer is also responsible for informing the query layer about the memory layout and structure of the raw files. The most important information is whether or not random access reads are supported and if data is stored in interleaved or columnar memory layout. In its current implementation, the input layer supports the file formats *LAS* and *LAZ*, as well as two custom file formats similar to *LAS /LAZ*, which are explained in Section 3.3.3 and Section 3.3.4.

The query layer is responsible for executing the actual queries based on the data provided by the input layer. Our prototypical implementation includes a rudimentary query language which supports the following set of query types and combinators:

- Basic binary operators (`==`, `!=`, `<`, `<=`, `>`, `>=`)

- Optimized range-queries, such as bounding-box queries for positions

- Intersections with 2D polygons

- Discrete LOD-queries based on grid-center sampling

For most queries, query execution boils down to applying a predicate function to each point and filtering the input range of points for all those that match the predicate. Some of the optimizations introduced in Section 3.3.3 require the query layer to know the exact memory layout of the input data, so there is a tight coupling between these two layers. While unfavorable from a software architecture perspective, it does improve the performance as the results in Section 3.4 show.

Lastly, the output layer is responsible for outputting the matching points to the application that is using the ad-hoc query engine. To correctly include data output overhead into our measurements, the output layer by default writes all matching points to the standard output stream. We support the following output modes for matching points:

- *Raw data*: Outputs the raw binary data of the point as it was read by the input layer. For a *LAS* file for example, this returns the raw *LAS* point record
- *Custom layout*: Outputs the point data in a custom memory layout defined by the application. This way, applications can filter only for the point attributes they care about, reducing the amount of data traffic

We implemented the ad-hoc query engine as a standalone command line application using the Rust programming language. The source code is available under an open-source license on GitHub [43].

3.3.3 Optimizations for I/O-bound queries

All queries whose runtime complexity is linear are expected to benefit from better I/O performance, as each point has to be read and inspected during the query. There are two ways to improve I/O performance: By *reading data faster*, or by *reading less data*. The speed by which the ad-hoc query engine can read point data is bound by the two factors *disk read speed* and *decoding speed* as seen in Equation (3.7). If the query is bound by I/O performance, decoding speed is not the limiting factor by definition. Hence, once the limit of the disk read speed is reached, further improvements in I/O performance require reading less data instead. This can be achieved by either compressing the data, so that the same number of points require fewer bytes, or by ignoring parts of the point data that are irrelevant for the current query.

Compressed file formats such as *LAZ* effectively turn I/O bound applications into compute-bound applications, as they are known for their poor decompression performance [106, 81]. Additionally, they do not support true random access to the point data, which prevents optimizations such as skipping unneeded point attributes. For that reason, we evaluate potential alternatives to the *LAZ* file format in Section 3.3.4.

Fixed-width binary formats on the other hand, such as LAS, are well-suited for optimizations that skip unimportant data. As an example, a pure bounding-box query only needs the position attribute of the point, which takes 12 bytes in uncompressed LAS. All other attributes could be skipped during reading in the input layer. Since *LAS* uses an *interleaved* memory layout—storing all attributes for a given point contiguously in memory—the number of bytes that will be skipped between two consecutive points will

be small, typically less than a cache line. On top of that, when using memory-mapped I/O, data is loaded into main memory page-wise. Even if a large portion of the data of a single point is irrelevant, the interleaved memory layout still causes this data to be loaded into main memory.

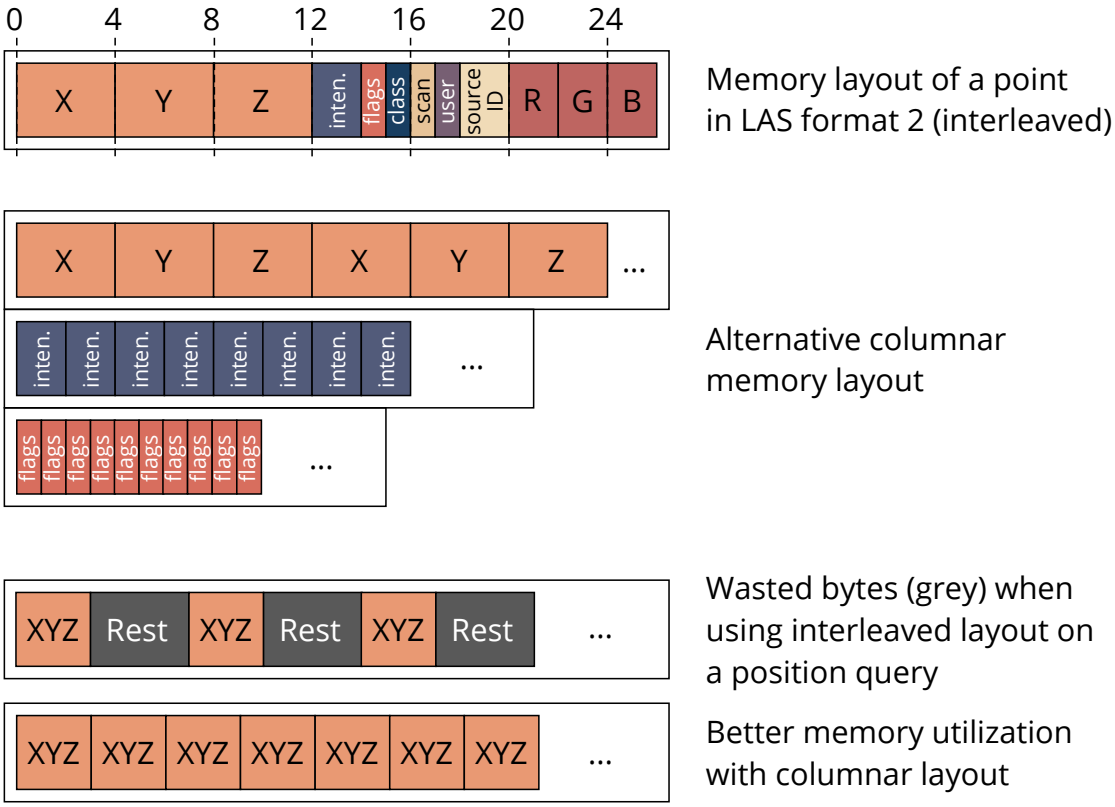


Figure 3.2: Interleaved and columnar memory layouts explained using the *LAS* file format as an example (top). When loading the point data during a query that does not require all attributes, the interleaved memory layout wastes bytes (bottom).

A better memory layout for skipping unnecessary data is the *columnar* layout, which stores the data for each attribute together in memory. Running *ad-hoc queries* on data in a columnar layout allows for effective skipping of large portions of the file for many queries. On top of that, the data comes in a more cache-friendly way, as there are no waste-bytes between consecutive points, as shown in Figure 3.2. The downside is that

accessing multiple attributes for the same point requires larger jumps in memory. We investigate the effect of this for more complex queries in Section 3.4.

The *3D Tiles* format uses this memory layout, but it is not widely used as a point cloud storage format. We thus propose a variation of the *LAS* file format with columnar instead of interleaved memory layout. We call this artificial file format *LAST*, an abbreviation of *LAS Transposed*, indicating the fact that the memory of a *LAST* file is simply the transposed memory of the point records of an *LAS* file. Figure 3.3 illustrates the memory layout of the *LAST* file format.

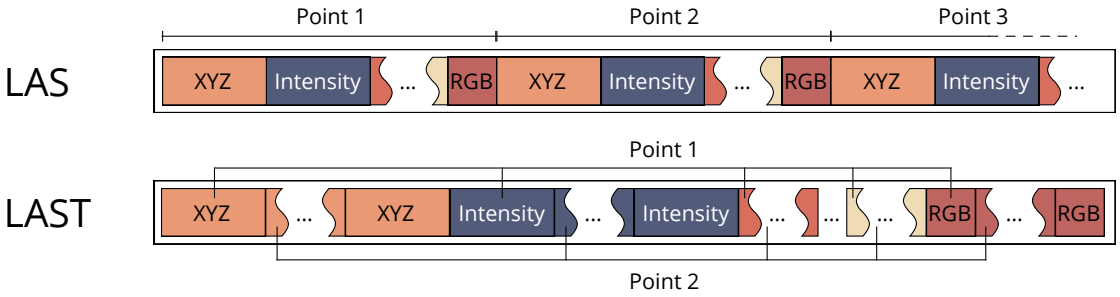


Figure 3.3: The memory layout of the *LAST* format compared to the *LAS* format. With *LAST*, all attributes of the same type (such as positions, represented as XYZ) are stored contiguously in memory.

Another optimization that we use for skipping data is based on metadata available for certain point cloud file formats. The *LAS* file format and all its derivatives contains the bounding box of the point cloud. Since point cloud datasets are often stored in multiple files—often in the form of disjoint tiles—this gives a rough positional index virtually for free. The ad-hoc query engine uses the bounding box information to discard irrelevant files during spatial queries which drastically speeds up positional queries on multi-file datasets.

3.3.4 Optimizations for compute-bound queries

For compute-bound queries, we are looking at optimizations that reduce the number of operations to perform for matching a single point against the query. We propose some optimizations for uncompressed data first, then explain our reasoning for dealing with compressed data.

Uncompressed point cloud formats

For uncompressed data, running the query on points in the exact memory layout of their file format prevents additional copy operations and data transformations. Together with memory-mapped I/O, this allows the query layer to operate on the raw memory buffer of the file. Instead of working with the memory layout of the source file(s), most applications will first read the point data into an internal memory layout. Depending on the difference between these two memory layouts, this operation can introduce significant overhead, which is undesirable for *ad-hoc queries*. This is the main reason why compressed point cloud data is typically one to two orders of magnitude slower to read than uncompressed point cloud data, as the compressed memory layout is very different from the uncompressed memory layout.

For uncompressed *LAS* data, it is easy for the query layer to take the native memory layout of the *LAS* point records into account. For most point attributes, this only requires the size and offset of the attribute and the size of the point record (often referred to as the *stride*). As an example, suppose the query layer executes a query for points matching a specific *object class*. The object class is represented by the *classification* attribute in the *LAS* file format using a single unsigned byte. If the input file uses *LAS* point record format 2, a single point record has a size of 26 bytes, with the classification value being located at byte offset 15 within a point record. The query layer thus reads single bytes with a stride of 26 bytes, starting at offset 15 from the beginning of the point records in the memory mapped file, as illustrated in Figure 3.4.

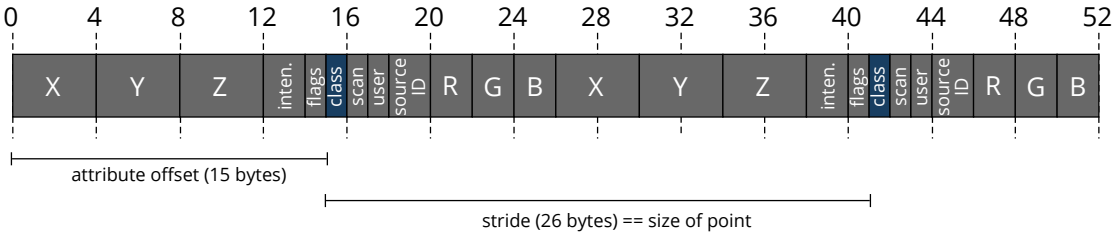


Figure 3.4: A query on the *classification* attribute can operate on the raw memory of a *LAS* file by reading every Nth byte, starting at an initial offset of K bytes (N=26 and K=15 for *LAS* point format 2)

There are certain types of attributes which require additional steps, since *LAS* supports so-called *scaled* attributes. The position attribute (made up of three individual attributes *X*, *Y*, and *Z*) is one of these scaled attributes. In LAS, positions are stored as 32-bit signed integers in local space, with 64-bit offset and scale values in the file header, which have

to be applied to the local positions to get the actual world-space positions.

Certain queries can be optimized so that these local-to-world-space calculations are not necessary. Shape intersections are an example for this optimization: Instead of transforming each point from local space to world space and intersecting it with a bounding box, we can instead transform the bounding box into local space one time and then compare it with the unscaled positions in local space. This is possible since this transformation uses only translation and scaling and thus is shape-preserving. We implement this optimization for both bounding box and polygon intersections in the ad-hoc query engine.

Compressed point cloud formats

The optimizations for skipping data during reading are not possible when using compressed point cloud formats such as *LAZ*, which do not provide true random access to individual points. *LAZ* is a chunk-based format and provides constant-time seeking to individual chunks, but within each chunk, points have to be decompressed sequentially. The underlying encoder is based on entropy and difference coding, which gives good compression ratios but is computationally expensive. As a point cloud storage format, *LAZ* has become a de facto standard for LiDAR data, but its poor decompression speed and lack of true random access are reasons why *LAZ* might be fully unsuitable to highly interactive applications. As an example, the progressive rendering system by Schütz et al. was only evaluated with uncompressed *LAS* files since *LAZ* was deemed too slow [142]. Since many datasets are stored using *LAZ* this might limit the usability of *ad-hoc queries* in practice.

To evaluate the usability of *ad-hoc queries* with compressed point cloud files, we developed an alternative file format based on the *LAST* file format, using the LZ4 compression algorithm which is optimized for decompression speed [92]. We call this format *LAZER* (short for *LAZ for Efficient Reading*) and compare it against *LAZ*. *LAZER* stores points in a series of blocks, similar to *LAZ*, where each block stores point attributes in columnar memory layout identical to the *LAST* format, but compressed using LZ4 using one compression context per attribute. At the beginning of each block, a block header stores the offsets to the start of the compressed attributes within that block, which is necessary for decoding the data. We deliberately designed *LAZER* to be a simple format similar in structure to *LAZ*, as this enables us to determine the effect that decompression speed has on the responsiveness and throughput of *ad-hoc queries*. Figure 3.5 illustrates the *LAZER* format visually.

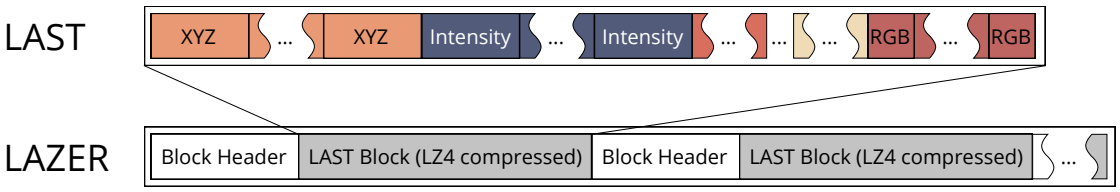


Figure 3.5: The memory layout of the *LAZER* format compared to the previously introduced *LAST* format.

3.3.5 *pasture* - A software library for working with point cloud data

Two of the main aspects of ad-hoc query performance are the effect of the memory layout of the point data (interleaved or columnar) and the number and type of point attributes that are loaded, parsed, and processed during a query. To aid in the development of the ad-hoc query engine and supporting tools, such as readers and writers for the *LAST* and *LAZER* formats, we developed a software library called *pasture* using the Rust programming language. In this section, we briefly describe the main design decisions for *pasture* and how it helped us to write the necessary code for the experiments described in Section 3.4. The source code of *pasture* is available under the Apache 2.0 license on GitHub [15].

We chose the Rust programming language not only for the development of *pasture* but for all experiments described in this chapter due to three reasons: First, Rust is a systems programming language with performance comparable to C and C++, and we expected that *ad-hoc queries* on point cloud data would require this level of performance and control. Second, the tooling landscape in Rust enables fast prototyping (for a systems programming language) and makes performance evaluations simple. There are several powerful benchmarking, profiling, and tracing tools such as *criterion* [48], *Flame Graphs* [56], and *tracy* [144] that either have native support within the Rust ecosystem or are easy to integrate. Lastly, we expected the memory safety and concurrency features of the language to simplify the development process compared to our own personal experience with C++ development in the past. While these matters are subjective—discussions about "Which programming language is the best for XYZ?" are often fought with near-religious zeal—the Rust programming language did become quite popular in recent years and has been recognized for its potential to develop scientific software [114].

The main inspiration for the design of *pasture* was *PDAL* [29], a widely used C++ library for working with point cloud data. Compared to *PDAL*, which supports a wide range of point attributes and readers/writers, *pasture* has a more flexible memory model for the storage of the point data. Where data in *PDAL* is always stored in interleaved

format, *pasture* supports several types of buffers, depending on the ownership of the memory and the layout of the memory. This makes it possible to write generic code with precise requirements for memory layouts (using the `InterleavedBuffer` and `ColumnarBuffer` traits), or code that works with buffers in any memory layout (using the `BorrowedBuffer` trait). A special buffer type is the `ExternalMemoryBuffer<T>`, which is an interleaved buffer that does not own its memory. Using this buffer type in conjunction with a memory-mapped file enables viewing the contents of a *LAS* file with virtually zero parsing overhead.

Internally, point data is stored as raw bytes with a piece of metadata called `PointLayout`, which describes which attributes the point data has and which primitive datatype represents the values of this attribute. This is similar to how *PDAL* handles point data, with added type safety through the usage of procedural macros, which allow safe transitions from static types to dynamically typed point buffers by generating the appropriate `PointLayout` for a Rust `struct` at compile-time. This is something that is not possible with *PDAL*.

The following listing shows a simple implementation of an ad-hoc query on a raw point cloud file using *pasture*:

Listing 3.1: Loading and querying an *LAS* file using *pasture*

```
// Imports redacted for brevity

let mut las_reader = LASReader::from_path("pointcloud.las")?;
let points = las_reader.read::<VectorBuffer>(
    las_reader.remaining_points())?;

const CLASS_BUILDING: u8 = 6;
for (index, _) in points
    .view_attribute::<u8>(&CLASSIFICATION)
    .into_iter()
    .enumerate()
    .filter(|(_, c)| *c == CLASS_BUILDING)
{
    std::io::stdout().write_all(points.get_point_ref(index))?;
}
```

3.3.6 Adaptive indexing strategies for compressed point cloud data

The main challenges when querying compressed point cloud file formats such as *LAZ* are the overhead of decompressing the data and the lack of constant-time random access. While faster compression algorithms might solve the first problem (as proposed in Section 3.3.4), without constant-time random access any query will have to decode all point data. One possible solution to this problem is the creation of a rough index structure that indexes the compressed blocks, allowing a query to filter for matching blocks within for example an *LAZ* file. This index should be lightweight, fast to create, and should preserve the original files to keep I/O operations to a minimum.

The *NoDB* system by Alagiannis et al. [4] fits these requirements and can be adopted to point cloud data in a straightforward way. In the original work, the authors used what they call an *Adaptive Positional Map* which refers to the position of attributes in a raw CSV file. CSV files are similar to point clouds, as both essentially are lists of tuples, but with some key differences that make it necessary to alter the way the adaptive positional map is applied for point cloud data. CSV files by themselves do not have constant-time random access to individual attributes within a row (or even to individual rows) since they are not fixed-width formats. If the position to a specific attribute is known however, constant-time random access is possible, as the attribute value can be read and parsed from that position. Compressed point cloud formats such as *LAZ* are different, since the decompression algorithm stores a context which is continuously updated, so the value of the Nth point depends on all previous points. As a consequence, even knowing the position of the Nth point within the compressed block, all previous points still would have to be decompressed in order to access the Nth point. As a result, the adaptive positional map in its initial form would not speed up the querying process. Instead, we propose to use an adaptive positional map that points to the start of compressed *LAZ* blocks and contains metadata information about the whole block. This metadata will be attribute-dependent and forms a simple index through which the query engine can decide if a given *LAZ* block contains matching points. Just as in the *NoDB* system, this positional map can be constructed on the fly during queries for all of the attributes required by the queries, thus making it adaptive as well. We call this structure an *Adaptive Block Index* in our system to distinguish it from the adaptive positional map in the *NoDB* system.

There are some similarities between the Adaptive Block Index and the file-pruning strategy using bounding box information in the *LAS* headers introduced in Section 3.3.3. In fact, the *LAS* headers form a block index over the position attribute where the blocks are whole files. This index is not adaptive since the *LAS* headers are already present in the files and do not change during query execution. We use the bounding boxes from the *LAS* headers of each file to initialize the Adaptive Block Index for the position attribute

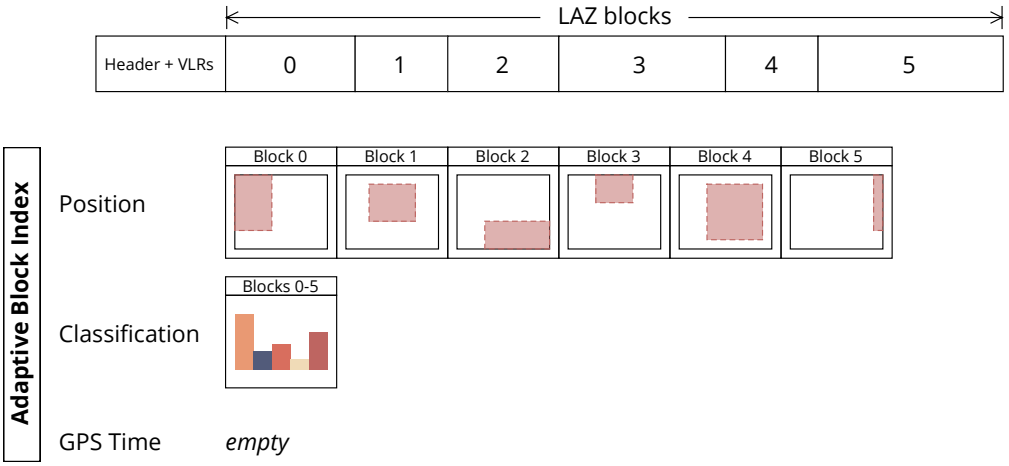


Figure 3.6: The Adaptive Block Index for a single *LAZ* file and three attributes. For the positions, a fully refined index exists that stores the AABB of each compressed *LAZ* block. For the classifications, the index is not yet fully refined and stores a single histogram for all blocks. For the GPS time attribute, no index has yet been created.

upon first loading of a dataset.

We implemented the Adaptive Block Index for several attributes defined by the *LAS* standard using three different data structures depending on the type of attribute. For positions, we use axis-aligned bounding boxes, for discrete values (for example the classification values or return number) we use a histogram. Using a histogram, we can prioritize matching blocks by the expected number of matches. While this does not affect the 100th-percentile responsiveness of the query, it does improve the Nth-percentile responsiveness for all $N < 100$ and consequently decreases the time until first results arrive. Figure 3.6 illustrates the Adaptive Block Index for a single *LAZ* file.

Refining the adaptive index happens during runtime based on the blocks that were queried. Each block that is to be refined is split into a number of smaller blocks by splitting the range of points into disjunct ranges and creating a block for each of these ranges. This process terminates when a block is as small as a single compressed block in a *LAZ* or *LAZER* file, which defaults to 50000 points. When splitting a block, we ensure that the boundaries of the point ranges of the blocks match the boundaries of the compressed blocks. This prevents blocks that would require costly seeking within a compressed block. We evaluate the Adaptive Block Index using multiple consecutive queries in Section 3.4.3.

3.4 Experiments

To evaluate the ad-hoc query engine, we conducted a series of experiments based on freely available point cloud data:

E1 *I/O performance*: Evaluates the read throughput of several common point cloud file formats and the custom formats described in Section 3.3.

E2 *Ad-hoc queries (no index)*: Evaluates the ad-hoc query engine on a wide range of queries and measures query throughput and responsiveness. Uses raw files only and no index.

E3 *Ad-hoc queries (adaptive index)*: Evaluates the adaptive indexing approach described in Section 3.3.6 on compressed point cloud data.

The specific datasets that we used are shown in Table 3.2, with detailed information about the number of points, number of files, and size in each of the file formats *LAS* and *LAZ*, as well as the custom formats *LAST* and *LAZER*. For the *CA13-S* dataset we used a subset of the full *CA13* dataset as our test machines did not have enough disk space to store the full dataset in its uncompressed form. The implications of this are discussed in Section 3.5.

Dataset (shorthand)	Size				Points	Files	Notes
	LAS	LAST	*LAZ*	LAZER			
District of Columbia 2018 (*DoC*) [156]	25.23GiB	25.23GiB	3.4GiB	11.04GiB	876M	328	
CA13 subset (*CA13-S*) [112]	45.27GiB	45.27GiB	6.82GiB	24.66GiB	1.43B	234	Subset consisting of the first 10% of all files in lexicographic order
AHN4 subset (*AHN4-S*) [2]	38.01GiB	38.01GiB	8.86GiB	22.54GiB	1.2B	1	Single tile `C_25GN1` (city center of Amsterdam)

Table 3.2: Datasets used in the experiments for the ad-hoc query engine

The experiments were run on the following two systems: A 2019 MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 processor, 32 GB 2667 MHz DDR4 RAM, a AMD Radeon Pro 5300M 4 GB dedicated GPU and 1 TB SSD running macOS 12.5.1 (labeled *MacBook*), and a desktop computer with a 3.2 GHz 6-Core Intel Core i7-8700 processor, 32 GB 2667 MHz DDR4 RAM, an nVidia GeForce GTX 1080 Ti and 500 GB SSD (Model WDC WDS 500G2B0B) running Ubuntu 22.04 (labeled *Desktop*). We benchmarked the SSDs on the

two test systems to achieve average read throughput values of 530MiB/s for the *Desktop* system—measured with the `hdparm` tool—and 2700MiB/s for the *MacBook* system—measured with the "Blackmagic Disk Speed Test" tool.

3.4.1 Experiment 1 - I/O performance

We tested the performance of reading point data into memory for each of the four formats *LAS*, *LAZ*, *LAST*, *LAZER*. The *pasture* library implements readers for *LAS* and *LAZ* and we wrote two custom readers for *LAST* and *LAZER*. For the common formats *LAS* and *LAZ* we also compared the performance to *PDAL* and *las-rs* [8], the Rust port of *libLAS*. We used the Rust port instead of *libLAS* so that the surrounding benchmarking code could be written in an identical way.

This experiment provides multiple insights: First and foremost, it establishes a baseline for the I/O performance of the target systems and each point cloud file format that we can compare to theoretical limits calculated from Equation (3.6). Secondly, it acts as a benchmark of the *LAZER* format, which we estimated to be significantly faster to read and parse than the *LAZ* file format. Lastly, it evaluates the performance of the *pasture* library.

The experiment takes a single point cloud file in each of the aforementioned file formats and reads all points within the file into an in-memory buffer. For *las-rs* and *pasture*, we wrote a small Rust executable that opens the file and reads all points into either a `Vec<Point>` using the default point type of *las-rs*, or one of `VectorBuffer` and `HashMapBuffer` for *pasture*, depending on the memory layout of the file format. For *PDAL*, we used the command line tool with the `pipeline` mode and measured its execution time using an output stage of type `writers.null`, which discards all read points. We also tested both *pasture* and *las-rs* in two configurations, once with a Rust `BufReader<File>`—which uses the default `read` system call—and once with memory-mapped I/O. To analyze the effect that the disk page cache has on I/O performance, we ran each experiment twice, one time with explicitly purging the cache and one time without.

We ran this experiment with a single point cloud file from the *DoC* dataset (`1321.las`) containing 3.9 million points. The resulting runtimes are shown in Table 3.3. For the *LAST* and *LAZER* formats, we only record results for *pasture* as the other tools do not support these custom formats. Looking at the data, a few things are noteworthy:

- Reading *LAZ* is between 3 to 4 times slower for both *pasture* and *las-rs* than reading uncompressed LAS. For *PDAL*, the difference is less significant, with *LAZ* being 1.5 to 2 times slower to read than LAS. Since the memory throughput is more than an

order of magnitude less for *LAZ*, this confirms that reading *LAZ* is compute-bound.

- Contrary to our initial assumption, using memory-mapped I/O does not give a noticeable performance benefit. In particular, there are large differences between the performance of `mmap` on Linux and macOS. `mmap` is as fast or slightly faster than using `read` on Linux, but up to 2 times slower on macOS for uncached files.
- *PDAL* benefits the most from disk page caching, with *LAS* parsing being about twice as fast if the file is cached. This effect also occurs when reading *LAZ* with *PDAL* which is surprising, as a primarily compute-bound problem should not benefit this much from an increase in I/O speed. Since the tests with *PDAL* run it as a standalone executable, whereas the other tests run multiple benchmarks within one executable, these values might be distorted by the overhead of launching the *PDAL* process.
- The *LAZER* file format is up to 5 times faster to read than *LAZ*. It is even slightly faster than reading uncompressed LAS, which is explained by the way that *pasture* implements point layout conversions: For all file formats, data is first read into an internal buffer in a point layout that exactly matches the binary layout of the point records in the file. This buffer is then converted into the desired memory layout, which for LAS-like formats means converting local-space positions into world space for example. This conversion process is more cache-efficient for data in columnar memory layout. This is also the reason why *LAST* is almost twice as fast to read as LAS.
- On the *Desktop* system, which has a slower SSD, the uncompressed file formats achieve anywhere from a third to close to half of the theoretical maximum disk read throughput. This does not scale to the faster SSD on the *MacBook* system, where only about 10 percent of the maximum disk read throughput are achieved. Together with the estimated values for T_{decode}, which show that the *Desktop* system has more compute power than the *MacBook* system, this is evidence that reading even uncompressed file formats is compute-bound.

Format	Tool	Throughput [MiB/s]		T_{read} [*MPts/s*]		T_{decode} [*MPts/s*]	Disk load factor
		No cache	Cached	No cache	Cached		
MacBook							
LAS	*pasture* (file)	238.092 ± 15.430	243.814 ± 16.480	8.322 ± 0.539	8.522 ± 0.576	9.127	8.8%
	pasture (mmap)	148.834 ± 1.977	224.919 ± 3.192	5.202 ± 0.069	7.861 ± 0.112	5.506	5.5%
	las-rs (file)	137.390 ± 2.595	145.595 ± 2.279	4.802 ± 0.091	5.089 ± 0.080	5.059	5.1%
	las-rs (mmap)	97.618 ± 4.778	166.142 ± 1.354	3.412 ± 0.167	5.807 ± 0.047	3.540	3.6%
	PDAL	49.625 ± 1.076	107.298 ± 11.65	1.735 ± 0.038	3.750 ± 0.407	1.767	1.8%
LAZ	*pasture* (file)	8.697 ± 0.178	8.879 ± 0.119	1.908 ± 0.039	1.948 ± 0.026	1.947	0.3%
	pasture (mmap)	8.192 ± 0.082	8.906 ± 0.111	1.797 ± 0.018	1.954 ± 0.024	1.832	0.3%
	las-rs (file)	7.597 ± 0.085	7.709 ± 0.059	1.666 ± 0.019	1.691 ± 0.013	1.696	0.3%
	las-fs (mmap)	7.133 ± 0.081	7.689 ± 0.085	1.565 ± 0.018	1.687 ± 0.019	1.591	0.3%
	PDAL	4.698 ± 0.279	7.650 ± 0.159	1.030 ± 0.061	1.678 ± 0.035	1.042	0.2%
LAST	*pasture* (file)	369.010 ± 31.430	413.166 ± 44.41	12.898 ± 1.099	14.441 ± 1.552	14.93	13.7%
	pasture (mmap)	187.353 ± 5.541	350.058 ± 7.580	6.548 ± 0.194	12.235 ± 0.265	7.037	6.9%
LAZER	*pasture* (file)	118.163 ± 6.700	133.497 ± 16.50	9.325 ± 0.529	10.535 ± 1.302	10.34	4.4%
	pasture (mmap)	84.561 ± 5.263	125.959 ± 13.68	6.673 ± 0.415	9.940 ± 1.080	7.181	3.1%
Desktop							
LAS	*pasture* (file)	172.263 ± 3.088	239.098 ± 26.690	6.021 ± 0.108	8.357 ± 0.933	8.920	32.5%
	pasture (mmap)	175.942 ± 0.322	252.436 ± 7.444	6.150 ± 0.011	8.823 ± 0.260	9.205	33.2%
	las-rs (file)	189.650 ± 1.453	199.499 ± 0.673	6.629 ± 0.051	6.973 ± 0.024	10.32	35.8%
	las-rs (mmap)	227.821 ± 1.097	242.706 ± 0.705	7.963 ± 0.038	8.483 ± 0.025	13.96	43.0%
	PDAL	103.011 ± 2.455	130.271 ± 7.312	3.600 ± 0.086	4.553 ± 0.256	4.469	19.4%
LAZ	*pasture* (file)	9.150 ± 0.084	9.290 ± 0.115	2.007 ± 0.018	2.038 ± 0.025	2.251	1.7%
	pasture (mmap)	9.126 ± 0.155	9.333 ± 0.098	2.002 ± 0.034	2.047 ± 0.021	2.244	1.7%
	las-rs (file)	8.778 ± 0.045	8.935 ± 0.029	1.925 ± 0.010	1.960 ± 0.006	2.149	1.7%
	las-fs (mmap)	8.806 ± 0.023	8.937 ± 0.036	1.932 ± 0.005	1.960 ± 0.008	2.157	1.7%
	PDAL	7.129 ± 0.185	7.943 ± 0.122	1.564 ± 0.041	1.742 ± 0.027	1.708	1.3%
LAST	*pasture* (file)	221.876 ± 13.410	371.439 ± 68.850	7.755 ± 0.469	12.983 ± 2.407	13.33	41.9%
	pasture (mmap)	241.148 ± 3.725	446.821 ± 3.644	8.429 ± 0.130	15.617 ± 0.127	15.46	45.5%
LAZER	*pasture* (file)	108.338 ± 7.384	129.893 ± 0.941	8.549 ± 0.583	10.250 ± 0.074	15.876	20.4%
	pasture (mmap)	106.765 ± 0.766	129.810 ± 0.211	8.425 ± 0.060	10.244 ± 0.017	15.453	20.1%

Table 3.3: Throughput values when reading a single point cloud file with 3.9 million points using various tools and file formats. $T_{decoding}$ values are estimated from Equation (3.7) using the peak disk read speed (2700MiB/s for the *MacBook* system, 530MiB/s for the *Desktop* system).

One of the assumptions that we made in Section 3.3.1 was that the binary layout of the point cloud file format, compared to the binary layout of the in-memory format, plays a significant role in how quickly point data can be read. The observations from the first

experiment support this assumption, which is why we conducted a second experiment to illustrate the overhead of binary layout conversions. In this experiment, we read data in the *LAS* and *LAST* file formats into multiple different memory layouts. For most tools, when reading *LAS* files, the default memory layout will contain positions in world-space using double-precision floating-point numbers, and might unpack the packed bit flag attributes, such as return number and number of returns. We compare this to a memory layout that reads all attributes exactly as they are stored in the *LAS* point records (positions as 32-bit integer coordinates in local space and packed bit flags), as well as two memory layouts that read only the positions or only the classifications. Table 3.4 shows the results of this experiment. The following observations can be made:

- The differences between the *Desktop* and *MacBook* systems are much larger in this experiment. This indicates that a closer match between the binary layouts of the file and in-memory format decreases the compute overhead and makes the parsing process mostly I/O bound. Both systems achieve over 80% disk read utilization for the native point layout and the *LAS* file format, but since the SSD of the *MacBook* system has a much higher read throughput than the one of the *Desktop* system, the point read throughput for the *MacBook* system is ten times as high as reading in the default memory layout, whereas for the *Desktop* system it is only twice as high.
- Using the default memory layout, reading *LAST* is slower than reading *LAS* because each attribute in the *LAST* file is read into a separate memory buffer, whereas a whole *LAS* file can be read using a single `read` call into one consecutive buffer.
- Reading individual attributes is much faster using the *LAST* format compared to the *LAS* format, which confirms the assumption that columnar memory layouts are better suited to accessing specific point attributes. This allows reading over half a billion classification values per second on the *MacBook* system even with an uncached file, and multiple billions of values on both systems if the file is cached.

3.4.2 Experiment 2 - Ad-hoc queries (no index)

To evaluate query performance using the ad-hoc query engine, we executed a series of queries on the three datasets *DoC*, *CA13-S* and *AHN4-S*. The queries are described in Table 3.5. The queries are inspired by those found in the literature (as discussed in Section 3.2) and consist of spatial queries with bounding boxes and polygons, queries on secondary attributes, as well as discrete LOD queries, all listed in Table 3.5. We compared the performance of our ad-hoc query engine to that of an existing database solu-

Format	Attributes	Throughput [MiB/s]		T_{read} [MPts/s]		T_{decode} [MPts/s]	Disk load factor
		No cache	Cached	No cache	Cached		
MacBook							
LAS	All (default)	250.810 ± 13.10	251.617 ± 13.22	8.766 ± 0.46	8.794 ± 0.46	9.664	9.3%
	All (native)	2316.742 ± 52.22	3750.762 ± 145.81	80.975 ± 1.83	131.097 ± 5.10	570.401	85.8%
	Local positions	1417.458 ± 50.16	1717.828 ± 37.62	49.543 ± 1.75	60.041 ± 1.32	62.712	52.5%
	Classifications	1425.647 ± 48.99	1782.741 ± 38.73	49.829 ± 1.71	62.310 ± 1.35	50.722	52.8%
LAST	All (default)	371.156 ± 28.81	403.484 ± 39.24	12.973 ± 1.01	14.103 ± 1.37	15.040	13.7%
	All (native)	1418.031 ± 82.24	1905.393 ± 95.38	49.564 ± 2.88	66.598 ± 3.33	104.388	52.5%
	Local positions	1807.343 ± 213.50	3666.920 ± 219.67	157.928 ± 18.65	320.420 ± 19.19	477.681	66.9%
	Classifications	492.147 ± 171.30	4153.255 ± 919.11	516.053 ± 179.60	4355.003 ± 963.7	631.085	18.2%
Desktop							
LAS	All (default)	178.081 ± 4.39	254.331 ± 30.50	6.224 ± 0.15	8.889 ± 1.07	9.374	33.6%
	All (native)	432.821 ± 4.32	1889.978 ± 6.31	15.128 ± 0.15	66.058 ± 0.22	82.500	81.7%
	Local positions	393.298 ± 0.31	1301.060 ± 10.70	13.747 ± 0.01	45.475 ± 0.37	19.549	74.2%
	Classifications	388.234 ± 0.99	1258.451 ± 2.05	13.570 ± 0.04	43.985 ± 0.07	13.909	73.3%
LAST	All (default)	228.982 ± 14.34	403.992 ± 80.68	8.003 ± 0.50	14.120 ± 2.82	14.091	43.2%
	All (native)	435.700 ± 4.67	2303.033 ± 45.31	15.229 ± 0.16	80.497 ± 1.58	85.591	82.2%
	Local positions	484.567 ± 2.87	6233.315 ± 42.59	42.342 ± 0.25	544.675 ± 3.72	493.944	91.4%
	Classifications	248.654 ± 13.71	12093.154 ± 2915.27	260.733 ± 14.37	12680.591 ± 3056.89	491.170	46.9%

Table 3.4: Memory and point throughput when reading a single point cloud file with 3.9 million points using the given set of attributes in the *LAS* and *LAST* file formats. All reading done using the `read` systems call instead of `mmap`. The *All (native)* memory layout corresponds to *LAS* point record format 6.

tion (*PostGIS* with the *pgPointclouds* extension), which supports all query types except discrete LOD queries.

Data preparation

For data preparation, we converted each of the datasets from its original file format (*LAS* for *DoC*, *LAZ* for *CA13-S* and *AHN4-S*) into all four file formats *LAS*, *LAZ*, *LAST*, and *LAZER*. We did not include the conversion time into the final query runtime as we do not consider this conversion to be part of the typical query process. Since we want to evaluate the effect of different point cloud file formats on query performance, having data available in all four formats is mandatory, in practice point cloud data will be stored in one specific format during data capturing and postprocessing.

For the *PostGIS* database, we employed *PDAL* [29] to insert the point cloud data into the database and recorded the time for this import in Table 3.6. For data exploration, this data import time is an important part of the overall data-to-query time [4] and has

Query name	Number of matches			Description
	DoC	*CA13-S*	*AHN4-S*	
AABB small	4.21M	9.01M	6.25M	Points within a small bounding box
AABB large	225.32M	329.32M	347.72M	Points within a large bounding box
AABB full	876.18M	1.43B	1.2B	Points within a bounding box as large as the dataset
AABB none	0	0	0	Points within a bounding box that does not intersect the dataset
Rect small	4.21M	4.20M	10.55M	Points within a small 2D rectangle
Polygon small	5.27M	15.31M	21.78M	Points within a small 2D polygon
Rect large	230.03M	433.18M	218.49M	Points within a large 2D rectangle
Polygon large	224.55M	269.85M	186.62M	Points within a large 2D polygon
Polygon with holes	3.06M	52.66M	11.34M	Points within a 2D polygon that has holes
Buildings	121.14M	930.49k	295.31M	Points classified as buildings
Buildings in small polygon	1.79M	15.58k	7.66M	Points classified as buildings within a small 2D polygon
Vegetation	175.98M	361.85M	0	Points classified as vegetation
First returns	741.77M	1.25B	989.35M	Points that are first returns
Canopies estimate	120.72M	160.13M	163.25M	Estimate canopy points through first returns from points with multiple returns
LOD0	3.44k	9.63k	19.67k	Discrete LOD 0
LOD3	170.87k	246.38k	1.1M	Discrete LOD 3

Table 3.5: All queries used in the *ad-hoc queries* experiment

Dataset	Upload time [hh:mm:ss]	
	Desktop	MacBook
DoC	5:55:07	2:45:23
CA13-S	N/A	3:54:46
AHN4-S	N/A	4:29:34

Table 3.6: Time for uploading the point cloud datasets into the *PostGIS* database. *N/A* values indicate a crash due to insufficient working memory during upload with *PDAL*.

to be taken into account when evaluating the query responsiveness.

During uploading the data, we encountered problems with the amount of working memory that *PDAL* requires, which exceeded the available resources on the *Desktop* system. The *MacBook* system had more swap space so we performed the data import there and manually loaded a database dump of the relevant tables onto the *Desktop* system. We believe that imports of larger datasets would not have worked on the *MacBook* system as well due to poor scalability of *PDAL*.

For the query polygons listed in Table 3.5, we created a single vector layer per dataset using the *QGIS* software and stored it as a shapefile. Our ad-hoc query engine loads these shapefiles and converts them into `INTERSECTS` queries. We also upload the shapes to *PostGIS* into a separate table which is referenced by the queries. The shapes are depicted in Figure 3.7.

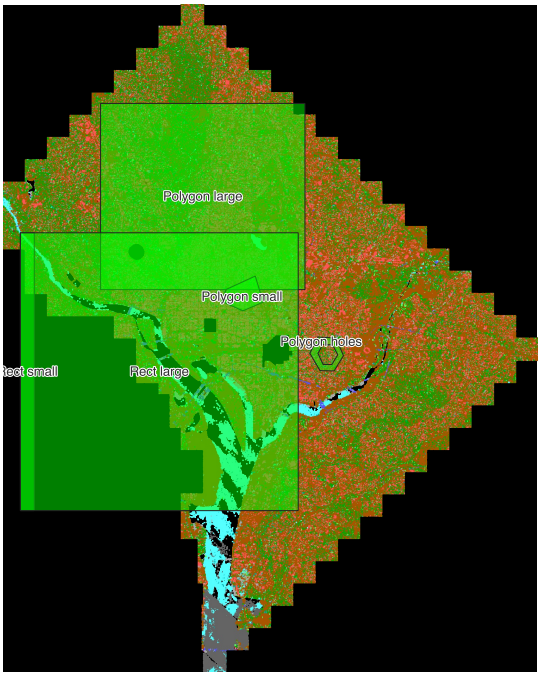
Query results

The resulting runtimes for the queries are shown in Table 3.7 (*DoC*), Table 3.8 (*AHN4-S*) and Table 3.9 (*CA13-S*). There are several important observations that can be made from the data regarding the following points:

- The effect of compression on query responsiveness and throughput
- The effect of columnar memory layouts on query responsiveness and throughput
- Differences between single-file and multi-file datasets
- The overhead of discrete LOD calculation during a query
- The performance of the database solution in comparison to the ad-hoc query engine
- Differences between the *Desktop* and *MacBook* systems

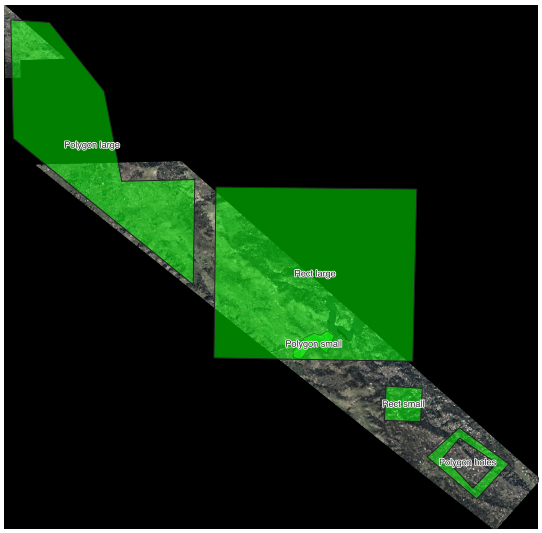
We discuss each of these observations in turn in the following paragraphs.

Queries on compressed vs. uncompressed data Queries on compressed data are generally significantly slower than on uncompressed data. Comparing the existing formats *LAS* and *LAZ*, *LAZ* is on average 4 to 6 times worse than *LAS*, both for responsiveness and throughput. With query throughputs of 5 to 10 million points per second, queries that have to inspect a large number of points (such as for buildings or first returns) take multiple minutes to complete. Our custom *LAZER* format has better performance, both



(a) Query shapes for the *DoC* dataset

(b) Query shapes for the *AHN4-S* dataset



(c) Query shapes for the *CA13-S* dataset

Figure 3.7: The query shapes for Experiment 2

Query	Runtime [s]					Throughput [*MPts/s*]				
	LAS	*LAST*	*LAZ*	*LAZER*	*PostGIS*	*LAS*	*LAST*	*LAZ*	*LAZER*	*PostGIS*
Desktop										
AABB (full)	31.341	70.612	79.663	73.764	1870.249	27.956	12.408	10.999	11.878	0.468
AABB (large)	9.603	17.656	37.020	19.180	504.126	28.219	15.348	7.320	14.128	N/A
AABB (none)	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	N/A
AABB (small)	0.470	0.422	2.206	0.506	9.054	27.893	31.014	5.936	25.867	N/A
Buildings	26.789	9.672	120.263	16.549	243.709	32.707	90.586	7.286	52.944	3.595
Buildings in small polygon	0.565	0.297	3.965	0.509	9.232	29.670	56.376	4.226	32.898	N/A
Canopies estimate	27.935	9.834	157.924	20.901	859.552	31.365	89.097	5.548	41.920	1.019
First returns	31.970	52.448	119.312	58.405	1315.611	27.407	16.706	7.344	15.002	0.666
LOD0	29.061	14.330	112.918	20.454	N/A	30.150	61.143	7.759	42.837	N/A
LOD3	28.896	14.382	111.932	20.443	N/A	30.322	60.922	7.828	42.859	N/A
Polygon holes	0.646	0.807	3.008	0.761	8.108	31.047	24.859	6.672	26.373	N/A
Polygon large	9.248	18.640	28.929	18.672	486.584	28.263	14.022	9.035	13.998	N/A
Polygon small	0.609	0.813	2.654	0.814	12.433	27.518	20.617	6.313	20.593	N/A
Rect large	9.528	18.433	29.307	19.383	500.381	28.441	14.701	9.246	13.980	N/A
Rect small	0.474	0.651	2.170	0.548	8.885	27.613	20.106	6.036	23.881	N/A
Vegetation	27.349	10.335	120.648	17.105	433.574	32.037	84.777	7.262	51.225	2.021
MacBook										
AABB (full)	43.387	96.095	95.205	98.177	3787.025	20.194	9.118	9.203	8.925	0.231
AABB (large)	13.433	23.613	41.427	25.853	933.980	20.172	11.476	6.541	10.481	N/A
AABB (none)	0.000	0.000	0.000	0.005	0.003	0.000	0.000	0.000	0.000	N/A
AABB (small)	0.653	0.491	2.502	0.817	17.066	20.050	26.665	5.234	16.033	N/A
Buildings	36.624	14.697	139.150	22.006	559.242	23.924	59.618	6.297	39.816	1.567
Buildings in small polygon	0.883	0.477	4.832	0.831	14.532	18.985	35.107	3.468	20.160	N/A
Canopies estimate	42.310	14.598	186.793	25.549	1306.469	20.709	60.020	4.691	34.294	0.671
First returns	46.809	72.200	157.799	79.522	3416.167	18.718	12.136	5.553	11.018	0.256
LOD0	39.285	21.658	136.787	24.380	N/A	22.303	40.456	6.405	35.939	N/A
LOD3	39.761	21.523	139.900	26.389	N/A	22.037	40.709	6.263	33.203	N/A
Polygon holes	0.923	0.766	3.675	1.014	14.438	21.743	26.187	5.461	19.801	N/A
Polygon large	13.594	24.511	154.746	26.038	947.055	19.226	10.663	1.689	10.038	N/A
Polygon small	0.857	1.091	2.983	1.080	24.028	19.554	15.364	5.617	15.519	N/A
Rect large	12.973	24.349	34.018	25.579	1022.159	20.887	11.129	7.966	10.594	N/A
Rect small	0.669	0.713	3.827	0.902	21.240	19.570	18.358	3.422	14.521	N/A
Vegetation	38.111	14.891	139.938	20.794	949.123	22.990	58.840	6.261	42.136	0.923

Table 3.7: Results of the queries on the *DoC* dataset for both our ad-hoc query engine (using 4 file formats) and PostGIS

Query	Runtime [s]					Throughput [*MPts/s*]				
	LAS	*LAST*	*LAZ*	*LAZER*	*PostGIS*	*LAS*	*LAST*	*LAZ*	*LAZER*	*PostGIS*
Desktop										
AABB (full)	76.171	103.066	113.197	101.808	2427.955	15.759	11.646	10.604	11.790	0.494
AABB (large)	64.643	44.609	181.923	41.270	940.807	18.569	26.908	6.598	29.085	N/A
AABB (none)	0.001	0.000	0.000	0.000	0.001	0.000	0.000	0.000	0.000	N/A
AABB (small)	60.888	40.580	177.088	27.328	46.823	19.714	29.580	6.778	43.924	N/A
Buildings	63.188	42.743	181.272	27.828	479.874	18.996	28.083	6.622	43.135	2.501
Buildings in small polygon	66.575	43.583	260.247	36.711	30.550	18.030	27.541	4.612	32.697	N/A
Canopies estimate	67.081	45.567	260.498	35.698	1022.489	17.894	26.343	4.608	33.625	1.174
First returns	80.624	76.056	190.959	77.062	1551.304	14.888	15.782	6.286	15.576	0.774
LOD0	64.269	40.438	186.891	30.385	N/A	18.677	29.683	6.423	39.505	N/A
LOD3	68.484	42.732	186.115	30.767	N/A	17.527	28.090	6.449	39.014	N/A
Polygon holes	59.649	41.009	179.627	27.964	31.212	20.124	29.270	6.682	42.925	N/A
Polygon large	66.274	52.599	193.274	35.281	373.298	18.112	22.821	6.211	34.023	N/A
Polygon small	61.152	44.364	179.720	27.617	42.317	19.629	27.057	6.679	43.464	N/A
Rect large	62.820	47.285	192.003	36.944	433.243	19.108	25.385	6.252	32.491	N/A
Rect small	60.828	39.480	180.158	27.571	20.347	19.734	30.404	6.663	43.536	N/A
Vegetation	64.193	43.668	177.649	26.376	6.709	18.699	27.488	6.757	45.509	178.908
MacBook										
AABB (full)	64.208	76.635	156.081	84.749	4978.637	18.695	15.663	7.691	14.164	0.241
AABB (large)	40.085	27.287	226.540	39.683	1893.588	29.945	43.989	5.299	30.249	N/A
AABB (none)	0.000	0.000	0.000	0.000	0.008	0.000	0.000	0.000	0.000	N/A
AABB (small)	14.293	4.324	215.965	14.105	90.947	83.982	277.615	5.558	85.100	N/A
Buildings	71.943	31.175	223.808	41.271	1323.129	16.685	38.504	5.363	29.085	0.907
Buildings in small polygon	23.939	6.439	316.118	19.844	54.545	50.142	186.422	3.797	60.491	N/A
Canopies estimate	77.519	26.932	319.734	48.094	1535.545	15.485	44.569	3.754	24.958	0.782
First returns	84.782	66.403	247.755	76.571	4389.366	14.158	18.077	4.845	15.676	0.273
LOD0	75.273	26.796	226.157	46.759	N/A	15.947	44.796	5.308	25.671	N/A
LOD3	76.847	27.426	226.503	48.035	N/A	15.620	43.767	5.299	24.989	N/A
Polygon holes	14.791	3.941	219.579	14.598	51.966	81.152	304.542	5.467	82.227	N/A
Polygon large	24.455	16.000	223.982	24.466	802.454	49.083	75.024	5.359	49.062	N/A
Polygon small	15.465	5.357	217.051	14.854	95.486	77.616	224.091	5.530	80.807	N/A
Rect large	27.311	17.382	223.969	26.679	910.757	43.951	69.056	5.359	44.992	N/A
Rect small	14.662	4.564	220.108	12.194	45.278	81.868	263.017	5.453	98.438	N/A
Vegetation	10.657	0.408	214.573	9.637	30.152	112.636	2944.845	5.594	124.558	39.810

Table 3.8: Results of the queries on the *AHN4-S* dataset for both our ad-hoc query engine (using 4 file formats) and PostGIS

Query	Runtime [s]					Throughput [*MPts/s*]				
	LAS	*LAST*	*LAZ*	*LAZER*	*PostGIS*	*LAS*	*LAST*	*LAZ*	*LAZER*	*PostGIS*
Desktop										
AABB (full)	73.200	81.463	134.123	77.880	2869.008	19.529	17.548	10.658	18.355	0.498
AABB (large)	15.845	18.334	125.654	18.308	675.901	22.517	19.460	10.974	19.487	N/A
AABB (none)	0.000	0.000	0.005	0.000	0.001	0.000	0.000	0.000	0.000	N/A
AABB (small)	0.970	0.637	0.746	0.778	19.503	26.143	39.821	12.226	32.613	N/A
Buildings	66.530	6.361	116.013	13.778	11.111	21.487	224.726	12.322	103.753	128.652
Buildings in small polygon	1.294	0.783	10.896	0.738	0.886	53.264	87.957	6.323	93.324	N/A
Canopies estimate	81.241	63.618	303.999	31.336	1604.268	17.596	22.470	4.702	45.619	0.891
First returns	82.335	76.969	222.834	72.626	2344.680	17.362	18.572	6.415	19.683	0.610
LOD0	82.551	72.264	213.770	30.635	N/A	17.317	19.782	6.687	46.662	N/A
LOD3	78.254	73.403	212.691	31.644	N/A	18.267	19.475	6.721	45.174	N/A
Polygon holes	7.083	5.412	25.678	4.833	124.046	26.025	34.064	7.179	38.141	N/A
Polygon large	21.217	21.478	40.005	16.039	540.183	14.984	14.803	7.947	19.822	N/A
Polygon small	2.491	1.420	9.294	1.582	30.348	27.655	48.513	7.413	43.545	N/A
Rect large	20.767	24.070	51.990	24.452	880.286	22.947	19.798	9.166	19.489	N/A
Rect small	4.041	2.785	14.769	3.241	89.348	25.990	37.710	7.111	32.403	N/A
Vegetation	74.273	64.538	211.142	36.566	698.968	19.246	22.150	6.770	39.093	2.045
MacBook										
AABB (full)	62.952	80.796	124.807	82.048	6176.889	22.708	17.693	11.454	17.423	0.194
AABB (large)	15.619	19.096	36.793	19.648	1436.911	22.842	18.683	9.697	18.159	N/A
AABB (none)	0.000	0.000	0.000	0.000	0.003	0.000	0.000	0.000	0.000	N/A
AABB (small)	1.319	0.691	4.656	0.868	43.488	19.228	36.705	5.447	29.218	N/A
Buildings	20.348	3.601	132.394	11.954	64.874	70.251	397.007	10.797	119.579	18.503
Buildings in small polygon	1.739	0.513	12.208	1.050	1.799	39.618	134.374	5.644	65.601	N/A
Canopies estimate	80.801	26.772	348.048	35.916	2040.786	17.692	53.395	4.107	39.801	0.588
First returns	87.057	73.871	280.368	78.250	5421.134	16.420	19.351	5.099	18.268	0.221
LOD0	76.064	30.051	244.066	40.033	N/A	18.793	47.569	5.857	35.708	N/A
LOD3	76.164	30.360	245.345	40.462	N/A	18.769	47.086	5.826	35.330	N/A
Polygon holes	8.326	4.536	30.305	5.771	271.420	22.142	40.645	6.083	31.945	N/A
Polygon large	16.304	16.207	44.411	17.174	1218.651	19.500	19.617	7.159	18.512	N/A
Polygon small	2.845	1.479	10.201	2.000	72.654	24.218	46.570	6.754	34.454	N/A
Rect large	22.562	25.348	53.347	25.830	1897.197	21.122	18.800	8.933	18.449	N/A
Rect small	4.638	3.067	16.406	3.684	189.937	22.643	34.243	6.401	28.505	N/A
Vegetation	77.221	34.753	246.419	40.143	1619.259	18.512	41.134	5.801	35.610	0.741

Table 3.9: Results of the queries on the *CA13-S* dataset for both our ad-hoc query engine (using 4 file formats) and PostGIS

for queries with fewer matches as well as queries on secondary attributes such as classifications and return numbers. The advantage of *LAZER* over *LAZ* also comes from the columnar memory layout of the former file format, as discussed in the next paragraph. Overall using a faster and simpler compression scheme does increase the performance of *ad-hoc queries* significantly, at the cost of worse compression ratios.

Queries on columnar vs. interleaved data Columnar memory layouts achieve better results for queries that either have fewer matches or operate on few or small point attributes. The combination of these two factors, as seen for example with the "Buildings" queries on the *DoC* dataset, causes response times and throughputs to be up to three times faster and higher using the columnar *LAST* format compared to regular *LAS*. In these cases, even compressed *LAZER* outperforms uncompressed *LAS*, as only a fraction of the data has to be loaded during query execution when using a columnar memory layout. When outputting a large amount of points, columnar formats become slower than interleaved formats in our implementation. This is not a principle limitation but comes from the implementation of our point output routine, which always writes data to the standard output stream in interleaved memory layout. The performance overhead of memory transpose operations is a limitation of columnar formats, as many applications require point data in interleaved memory layout.

Single-file datasets vs. multi-file datasets There are large differences between the ad-hoc query runtimes for single-file and multi-file datasets. As discussed in Section 3.3.3, the *LAS* file format includes bounding-box information in the file header. Data delivery for large point clouds typically includes a form of spatial tiling, so that all files in the dataset have little to no overlap. This enables sub-second response times for spatial queries if their bounding region is sufficiently small, as shown by the results for the *DoC* and *CA13-S* datasets. Without this information, there is little difference in runtime between querying a small bounding region and querying all points within the dataset. For the single-file dataset *AHN4-S*, all queries have similar runtimes as they all amount to a full scan. Ad-hoc queries thus benefit immensely from datasets that are already pre-tiled. It is important to point out the difference between this tiling process and the computationally more expensive index creation discussed in Chapter 4. It is sufficient to perform rough tiling to get speedups of one to two orders of magnitude with *ad-hoc queries*.

Discrete LOD calculation Selecting representative subsets of a point cloud using LOD is especially important for visualization applications, hence we evaluated on-the-fly discrete LOD generation in the context of *ad-hoc queries*. The resulting runtimes are similar

to those of the "AABB full" query and thus are far from interactive speeds for all tested datasets. Nonetheless it is interesting to observe that the computational overhead compared to non-LOD queries over all points is small, on average less than 25%. Our ad-hoc query engine uses a grid-center sampling approach for discrete LOD calculation, which is used in more sophisticated point cloud indexing tools such as *Entwine* [61] as well. To decide whether or not *ad-hoc queries* are suitable for querying data with LOD, the point throughput numbers serve as a guideline. For the *DoC* dataset, between 40 and 60 million points per second can be queried using LOD.

Ad-hoc queries vs. a database solution The results for the query runtimes using the *PostGIS* DBMS with the *pgPointclouds* extension illustrate a problem that has been noted by Cura et al. [32]: Point cloud databases have poor data export performance. The results in Table 3.7 and the related tables show that *PostGIS* is one to two orders of magnitude slower than using *ad-hoc queries*. These numbers exclude the initial data upload time (see Table 3.6). Together, the data-to-query time when using *PostGIS* is several orders of magnitude worse than when working with our ad-hoc query engine.

Differences between the *Desktop* and *MacBook* systems Lastly we want to point out interesting differences between the *Desktop* and *MacBook* systems. As we observed in experiment 1, the usage of memory-mapped I/O is not a silver bullet and there are differences in the implementation on the operating-system level that cause memory-mapped I/O to be slower on macOS than on Linux. The results of this experiment show that disk I/O is less important than memory speed and compute resources, as the *Desktop* system often outperforms the *MacBook* system, although the latter has a faster SSD. Implementation-wise there are also caveats, as memory-mapped I/O makes parallelization easier as it requires no explicit synchronization. This becomes especially important for the single-file dataset *AHN4-S*: Our implementation, which parallelizes over blocks of 1 million points, performs many concurrent read and seek operations within the same file. The typical types for file-based I/O within the Rust standard library are not synchronized, and adding a synchronization primitive prevents all parallelization. For this reason, we are using memory-mapped I/O also on the *MacBook* system.

3.4.3 Experiment 3 - Ad-hoc queries (adaptive index)

In this experiment we evaluate the usage of adaptive indexing together with *ad-hoc queries*. The previous two experiments showed that *ad-hoc queries* on compressed files, as well as on large, single-file datasets exhibit poor performance. To evaluate whether

adaptive indexing can help in these situations, we ran a series of queries on the *AHN4-S* dataset using the adaptive indexing strategy described in Section 3.3.6. This approach builds a simple block-based index that stores one acceleration structure per consecutive range of points and value type, and adaptively refines the index by splitting blocks with larger point ranges into smaller ones. We tested two different strategies for adaptive indexing:

1. *Refine all*: All blocks visited by the query that can be refined are refined into up to four smaller blocks.

2. *Time budget*: Refine blocks for a given maximum time. The blocks to refine are selected using a heuristic based on the peak I/O throughput values from experiment 1. This is similar to the progressive indexing budget introduced by Holanda et al. [64].

To see the effect of adaptive indexing, all queries were executed eight times and the total runtime for querying and refinement, as well as the Nth-percentile query responsiveness was recorded. We tested three queries from experiment 2 (*AABB small*, *AABB large*, and *Buildings*), as well as a query that starts with the bounding box of the *AABB large* query and interpolates to the bounding box of the *AABB small* query.

The runtimes for this experiment are shown in Figure 3.8. These results show that small spatial queries benefit most from adaptive indexing, with the cost of adaptive indexing amortized after at most three queries for all file formats. Due to their slower parsing speed, compressed file formats require more time for index refinement, which is especially prominent for the *LAZ* format and the time-budget refinement strategies. Where other formats have a fully refined index after at most four iterations, the index is still not fully refined after the full eight iterations using *LAZ*. Nonetheless there is a noticeable improvement in query responsiveness as can be seen for example in the results for the *AABB small* query. Here the index refinement takes less than 10 percent of the total query runtime and consistently decreases the runtime of the next query by about the same percentage.

Our block-based index operates without sorting the points and thus exploits locality inherent to the points within the file(s). There is often decent spatial locality in point cloud datasets due to the way points are recorded and postprocessed, but this locality does not extend to other point attributes. For the *Buildings* query, only a fraction of the blocks in the refined index do not contain any points classified as buildings, hence the improvement in query runtime is marginal. Even in those cases though there is an improvement of the Nth-percentile query responsiveness, especially for low values of N, as can be seen in Figure 3.9. For discrete values like point classifications, our implementation prioritizes

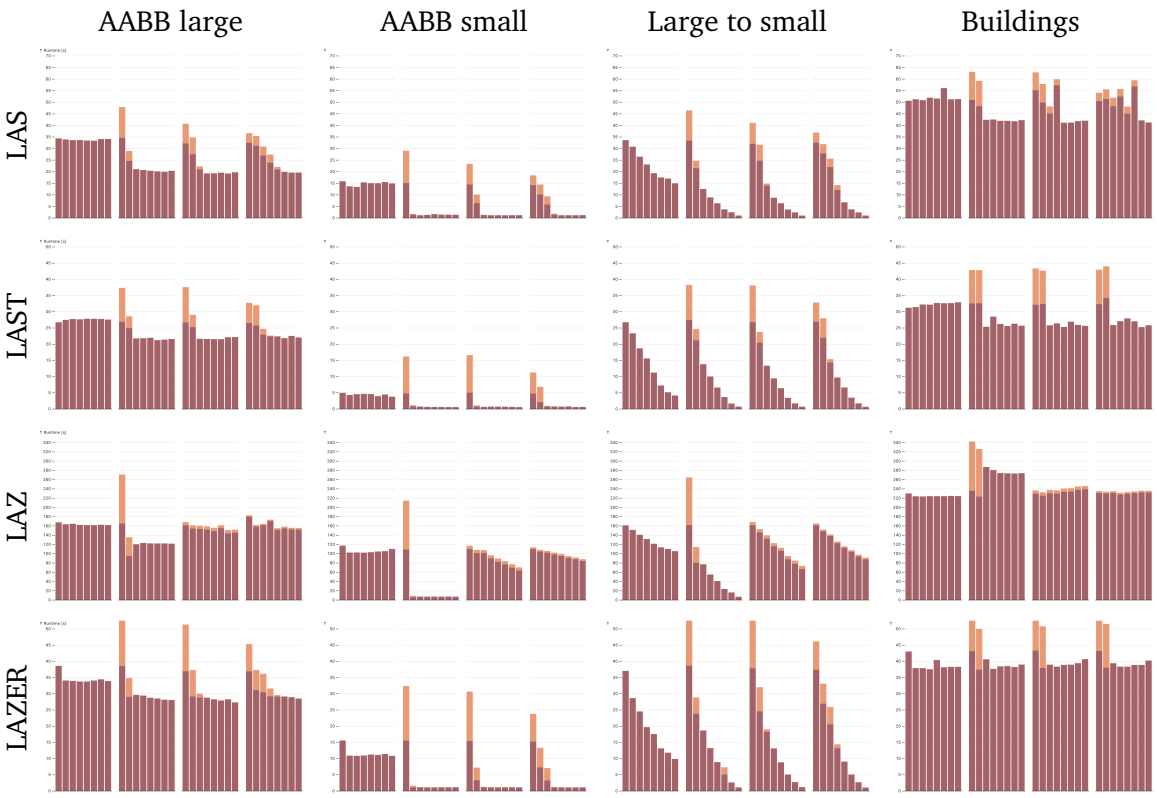


Figure 3.8: Runtimes of 8 consecutive queries using four adaptive indexing strategies (left to right within each chart: No indexing, *Refine all*, *Time budget 10s*, *Time budget 5s*) and four file formats. Red bars show the query runtime, orange bars stacked on top show the time spent for indexing.

blocks with a high number of matches, which improves the time until first query results are obtained, even though the total query runtime remains nearly constant.

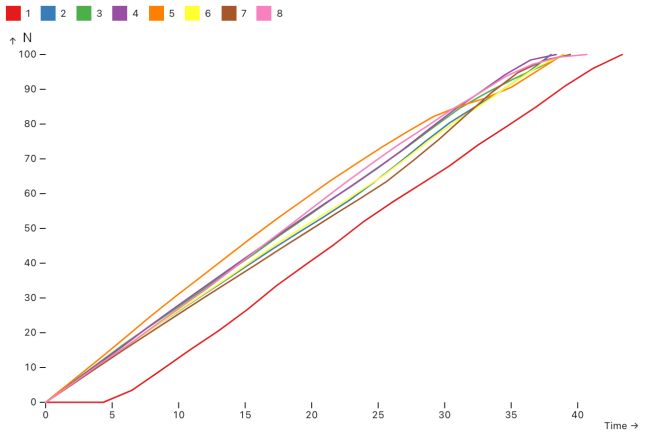


Figure 3.9: Nth-percentile query responsiveness $r_N(Q)$ for the *Buildings* query on the *AHN4-S* dataset in the *LAZER* format with an adaptive indexing time budget of 10 seconds. Colored lines show the different iterations of the repeated query.

3.5 Discussion

In this section we discuss the results from the experiments with regard to the initial research question 1. Beyond that, we point out remaining challenges and limitations of our proposed *ad-hoc queries* approach.

3.5.1 Implications for research question 1

We set out to answer the following research question regarding file-based point cloud data management:

RQ1 Can *ad-hoc queries* enable applications to work directly with raw point cloud files instead of sophisticated index structures?

We now summarize the results of this chapter, including the experimental data shown in Section 3.4, through which we are able to answer this question.

We evaluated two different point cloud memory layouts (*interleaved* and *columnar*) and demonstrated their performance characteristics both in isolated I/O benchmarks (Section 3.4.1) as well for various queries (Section 3.4.2). We found significant differences in I/O and query performance between the interleaved and columnar memory layouts, both for compressed and uncompressed data. For raw read performance, the interleaved memory layout is preferable, as it requires the fewest number of read and seek operations within a file. In situations where only one or few specific point cloud attributes are of interest to answer a query, the columnar memory layout becomes more efficient as it allows selective reading of specific point cloud attributes. For LiDAR data, there are several interesting point attributes that are stored in a single byte, such as the point classification, number of returns, or the return number. Queries that operate on these attributes benefit the most from columnar memory layouts, as the amount of data that has to be read and parsed is typically more than an order of magnitude less than with point cloud data in the interleaved memory layout. Using our new *LAST* file format, this enables query throughputs of 100 million points per second and more for many queries on consumer-grade hardware.

Besides read throughput, the overhead for decoding point cloud data from the native binary layout of a file format into an in-memory representation also plays a significant role in the overall query throughput and responsiveness. Here, columnar memory layouts are more cache-efficient when it comes to attribute conversions, such as the local-to-world-space conversion typically applied when reading data in the *LAS* format. If possible, it pays off to work with point cloud data in a binary layout that closely matches the file layout, as the results of experiment 1 showed. Our ad-hoc query engine is developed with memory layout awareness in its various layers, but we ultimately output data in interleaved memory layout as this is what most applications work with. This columnar-to-interleaved conversion in the output layer introduces a non-trivial runtime overhead. If the target application that uses *ad-hoc queries* can work with columnar point cloud data, there is potential for large performance improvements. Our conclusion is that writing software that is agnostic to the point cloud memory layout results in unnecessary loss of performance. The *pasture* library that we developed helps with writing code that is aware of the point cloud memory layout, but even with this library writing such memory-layout-aware code adds significant complexity to the codebase as basically all important code paths have to deal with both interleaved and columnar memory layouts.

The effects of compression are interesting as well. In almost all experiments the *LAZ* format is close to an order of magnitude slower than the uncompressed formats. Faster compression schemes can help, as we demonstrated with our custom *LAZER* format, at the expense of the compression ratio. Perhaps the biggest advantage of *LAZER* over *LAZ* is selective decompression of attributes, as it allows the same kind of efficient data loading

that *LAST* has over *LAS*. It is worth noting that the latest version of *laszip* also supports selective attribute (de)compression, however only for the newer point record formats 6 and higher introduced with version 1.4 of the *LAS* standard, which most of our test datasets do not use.

Given all these data, the question remains whether *ad-hoc queries* are a viable data management solution for point clouds. Comparing our results to the existing point cloud DBMS *pgPointcloud* showed that the latter has very poor query responsiveness and throughput. This is due to a deficit in the implementation of *pgPointcloud* which outputs all point attributes as double-precision floating-point numbers, thus significantly increasing the amount of memory that has to be written compared to our ad-hoc query engine. Notwithstanding these implementation deficits, there also is significant overhead for preparing and inserting the data into the database. For our experiments, these preprocessing steps took several hours per dataset, several orders of magnitude higher than the query responsiveness of all *ad-hoc queries* that we tested, even for slowest compressed data format *LAZ*. In terms of pure data-to-query time, *ad-hoc queries* do have a clear advantage over point clouds DBMSs in their current form.

We also demonstrated the potential that adaptive indexing methods have for improving query performance and bridging the gap between *ad-hoc queries* on unindexed data and fully indexed data. Our indexing method was optimized for fast indexing, at the expense of the quality of the index. Depending on the time that users are willing to spend on indexing, more sophisticated methods can be used that restructure the data, either in-place or by creating copies of it. Hence, we believe that future point cloud DBMSs should focus on supporting existing point cloud file formats (or variations thereof like *LAST* and *LAZER*) as a first-class storage medium. Our experiments demonstrate that it is possible to answer many types of common point cloud queries in reasonable time on these raw files, while still allowing the usage of indexing wherever necessary.

We conclude that *ad-hoc queries* can allow applications to work with raw point cloud files, making them a viable point cloud data management solution, so long as their limitations are well understood and care is taken in the implementation of tools that work with the raw files. Without a more efficient alternative to the *LAZ* format, uncompressed file formats are most certainly required to achieve decent query responsiveness and throughput. With further research in the area of adaptive indexing, the gap between file-based solutions and DBMSs can probably be bridged, which would make the two approaches complimentary instead of mutually exclusive, as it is today.

3.5.2 Limitations, challenges, and future work

There are obvious limitations with *ad-hoc queries* in their current state. First and foremost, data size is still a limiting factor, as our proposal for an ad-hoc query engine assumes that data resides on fast, local storage (preferably an SSD). There are many existing point cloud datasets whose size exceeds the current SSD capacity of commodity hardware, especially when using uncompressed or lightly compressed formats. We were unable to evaluate both the *CA13* and *AHN4* datasets in their full size due to this limitation. Even if storage size would not be a limiting factor, many *ad-hoc queries* would not be feasible due to the expected runtime with these large datasets. While many of the queries that we tested on the three datasets finished in a minute or less, this already stretches the definition of "interactive response time" and is certainly still too slow for highly interactive applications such as visualizations, with the notable exception of small, spatial queries. Using the formulas from Section 3.3.1 and the resulting query runtimes from experiment 2, we can estimate that a query for discrete LOD level 0 on the full *AHN4* dataset in its default *LAZ* format would take about 2 days. Even in the faster *LAZER* format, where we achieve throughputs of close to 40 million points per second for LOD0 queries on the *Desktop* system, such a query would still take more than 6 hours. LOD queries belong to a category of queries that do not benefit from locality in the data while requiring a full scan of the data. As pointed out in the beginning of this chapter, constant-time speedups can only go so far and the largest point cloud datasets are still out of reach for *ad-hoc queries* on commodity hardware.

Beyond these limitations, we see some challenges that we expect can be overcome with more research and development effort. First, in the field of compressed point cloud formats, we believe there is room for significant improvements over the current de facto standard format *LAZ*. We do not see our custom format *LAZER* as a real replacement, as its compression ratio is not competitive to that of *LAZ*, but fast compression algorithms such as *LZ4* might still play a role for future compressed point cloud formats. Second, we believe that columnar data formats will become more commonplace for point cloud applications. Already the research community acknowledges the benefits of these formats, both in the database world [150] as well as for highly interactive applications such as real-time rendering [142, 138]. Lastly, there is need for improvements of the tools that currently exist for dealing with file-based point cloud data. While developing our ad-hoc query engine and running the experiments, we frequently ran into problems with existing tools that simply were not able to handle large point cloud datasets on our test machines because they ran out of memory (*PDAL* while uploading data to *pgPointcloud*) or took hours for file format conversions (*LAStools* when converting from/to *LAZ*). We ended up using our own ad-hoc query engine for aiding in its own development, for example to in-

spect datasets using LOD or extracting portions into *LAS* files. We acknowledge that our own tool does not have the same amount of features as for example *PDAL* or *LAStools*, and we merely want to point out the potential for performance and efficiency improvements in existing tools.

3.6 Conclusion

In this chapter, we introduced the concept of *ad-hoc queries* on point cloud data as a potential means for using raw files as a point cloud data management solution. First, an analysis of point cloud queries found in the literature was conducted through which the most common query types could be identified. We then described the design and implementation of a prototypical *ad-hoc query engine*, which is able to answer a wide variety of common queries using raw, unindexed point cloud files. Two custom file formats called *LAST* and *LAZER* were introduced, which are closely related to the common formats *LAS* and *LAZ*. These new formats use a columnar memory layout for data storage and a fast compression scheme (*LZ4*) in the case of *LAZER*. All four formats were evaluated in terms of their I/O performance through which key performance characteristics of interleaved and columnar memory layouts were identified. We then evaluated the same four formats using our ad-hoc query engine on several different queries, both spatial and on secondary attributes, and compared the responsiveness and throughput of these queries to an existing DBMS (the *pgPointcloud* extension for *PostGIS*). Lastly, we evaluated an adaptive indexing algorithm that builds a simple block-based index during querying and showed the impact that adaptive indexing has on the responsiveness of repeated queries.

Our results indicated that columnar memory layouts are a good alternative to the more common interleaved memory layouts, as they can speed up both I/O itself as well as many types of queries. Taking into account the required preprocessing time for using a point cloud DBMS, *ad-hoc queries* exhibit significantly lower *query-to-insight time*. Depending on the file format and type of query, our ad-hoc query engine can query more than 100 million unindexed points per second. We also demonstrated shortcomings of compressed file formats and how columnar memory layouts and faster compression algorithms can help circumvent them. Lastly, we pointed out limitations of *ad-hoc queries*, such as a maximum data size that can reasonably be handled, as well as query response times that are still too slow for highly interactive applications.

4 Improving the performance of batch-based point cloud indexing

"It is easy to make things hard, but hard to make them easy."

Rutger Bregman, Humankind: A Hopeful History

In this chapter, we improve upon the current state of the art for batch-based point cloud indexing. Current indexing tools work by batch-processing the whole point cloud in a single long-running process, and we demonstrate how such a process can be optimized to run faster even when scaling up to terabyte-sized datasets. The work in this chapter is based on our publications "A System for Fast and Scalable Point Cloud Indexing Using Task Parallelism" [16] and "Point cloud indexing using Big Data technologies" [76] and answers the following research question:

RQ2 How can the runtime of current point cloud indexing tools be improved?

This chapter starts with a discussion regarding the motivation for batch-based point cloud indexing in Section 4.1, including an overview of the current state of the art and limitations, as well as the theoretical background for scalable point cloud indexing (Section 4.2). We then introduce our *Schwarzwald* system, a batch-based point cloud indexing system based on task-parallel programming, in Section 4.3. We then extend the core algorithmic concept of *Schwarzwald* to a distributed, Cloud-based point cloud indexing system in Section 4.4, discussing potential benefits and challenges compared to indexing on a single machine. We evaluate both approaches in a series of experiments in Section 4.5 which compare their performance, scalability, and quality to that of state of the art tools. The results and their implications for research question 2 are discussed in Section 4.6.

4.1 Motivation: Handling very large point clouds interactively

Where in the previous chapter we explored situations in which point cloud data can be used in its raw, unindexed form, we now look at the state of the art for handling very large point cloud datasets. When we talk about a point cloud being "very large", what is typically meant is a dataset that does not fit into memory on a client device, for example a desktop computer, laptop, or smartphone. One billion points currently are a reasonable threshold for what is possible to handle *in-core*, i.e. within working memory or GPU memory, as both our own work [17] as well as that of other researchers [142] point out. Even then, the limitations of systems that deal with point clouds that sit right at this threshold are considerable, as we have shown in the previous chapter.

As soon as the size of the data, the number of concurrent users, or the query complexity increases beyond a certain point, working with point clouds requires indexing. Significant advancements in the domain of point cloud visualization have led to the emergence of specialized index structures such as *Nested Octrees* [129] which are used in interactive, Web-based visualization applications [133] that make point clouds accessible to a wide audience. Creating these index structures is a resource-intensive process, taking many hours or even days [96], for reasons which we will explore in-depth in this chapter. The amount of data itself is challenging, since a lot of data copying has to take place to reorder the point cloud into a Nested Octree. On top of that, most of the time the data does not fit into working memory, requiring *out-of-core* algorithms that cache intermediate results to disk. Lastly, the sampling process for LOD creation is often computationally expensive.

Point clouds are a form of Big Data and improvements to capturing devices and methods have caused an exponential increase in the amount of collected data [113]. To keep point cloud indexing feasible while data volume grows, better scalability is required, both vertically onto multiple cores of a single machine, as well as horizontally onto multiple machines. This requires improvements to the input and output layers of point cloud indexers, as well as truly scalable algorithms for the actual indexing process, with point sampling for LOD creation at its core.

We introduce two systems that improve upon the state of the art of point cloud indexing by allowing faster indexing as well as indexing of larger datasets. We build on the concept of Morton indices to distribute the workload of point cloud indexing evenly onto multiple compute-units (either CPUs or VMs in the Cloud). First, we demonstrate a single-process system called *Schwarzwald*, which is able to index point clouds with tens of billions of points. At the time of the initial publication, it outperformed the existing systems by up to 9 times while maintaining similar visual quality. Since then, further improvements have been made to existing systems, in particular version 2.0 of *PotreeConverter* [138], with significant improvements in performance. We show that *Schwarzwald* is often able

to hold up with the current state of the art of *out-of-core* point cloud indexing in terms of runtime, while showing better scalability due to it using less working memory. We then extend the idea of indexing using Morton indices to a distributed algorithm using the *Map-Reduce* paradigm [34]. Combined with a distributed database, this algorithm achieves even better scalability, outperforming all existing systems by up to a factor of 3 while running natively in Cloud environments.

4.2 Morton indices and scalable point cloud processing

Creating a spatial index for a point cloud requires grouping spatially adjacent points together into cells, typically nodes in a tree-based data structure. Disregarding the necessary sampling process to create LOD layers, at the core this is a sorting problem that tries to preserve locality. As we explained in Section 2.2.2, space-filling curves can be used to efficiently generate various tree-based acceleration structures, in particular by using Morton indices. Various massively-parallel and GPU-based algorithms have been devised for the construction of bounding volume hierarchies, k-d trees, and octrees [82, 74, 40] or for scheduling point cloud processing tasks onto distributed systems [6].

While Morton indices are widely used in the computer science community, there are some caveats when computing them, specifically due to the handling on floating-point values. We briefly state a formal definition for a Morton index in $\mathbb{R}^3$ and illustrate how we calculate it in our implementations.

Given a bounding box $B \subset \mathbb{R}^3$, subdivide it into a regular grid of k^3 cells. Let $c_i = (x_i, y_i, z_i)^T$ be the index of cell i, with $x, y, z \in [0; k-1]$. The Morton index m_i for cell c_i is an integer number obtained by interleaving the bits of x_i, y_i and z_i. In $\mathbb{R}^3$, there are six possible orders for interleaving the bits from most significant to least significant bit (XYZ, XZY, YXZ, YZX, ZXY, and ZYX), in our implementation we use the order XYZ. The bit-interleaving process is illustrated visually in Figure 4.1.

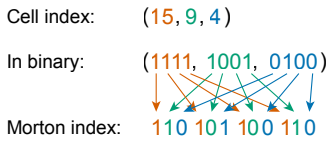


Figure 4.1: A 3D Morton index is calculated from X (red), Y (green) and Z (blue) coordinates through interleaving of the bit representations of the coordinates. Image source: [16]

In order to compute a Morton index for a point $p \in \mathbb{R}^3$ in a point cloud, we have to subdivide the bounding box of the point cloud into a regular grid and then find the grid cell that contains p. While the definition of the Morton index allows for non-cubic grid cells, we are interested in calculating octrees which are cubic, hence we first calculate the cubic bounding box of the point cloud and use it as a reference for Morton index calculation. Working with a cubic bounding box makes it easier to achieve uniform sampling of points over all three spatial dimensions. We then have to choose an appropriate value for the subdivision factor k, which is a tradeoff between precision and the number of bits required to store the Morton index. The value $k = 21$ is reasonable for our implementations, as it allows a Morton index to be stored efficiently within a single 64-bit integer value with only one unused bit. From the subdivision factor, one can derive certain thresholds, such as the maximum depth of the octree (21 levels) and the minimum side length of any octree node, the latter being equal to $2^{-k} = \frac{1}{2097152}$ times the side length of the cubic bounding box. This defines the maximum ratio between point density and size of the dataset that can be represented. As an example, a dataset with a cubic bounding box of 20 km side length can be processed if the smallest nodes are no smaller than about 1cm, a limit that was sufficient for all datasets that we encountered during our tests. In principle, it would be possible to use 128-bit Morton indices, yielding $k = 42$ and hence about millimeter precision for earth-sized point clouds. In practice, commodity hardware typically does not support 128-bit arithmetic natively yet, resulting in decreased performance since it has to be emulated using multiple 64-bit operations, as well as increased memory usage.

To find the cell c_p for a point p, we first translate p into the local reference frame of the cubic bounding box B, whose origin is the minimum vertex of B. This yields a point $p_B \in [0; l_B]^3$, where l_B equals the side length of B. Multiplying p_B by $\frac{2^k}{l_B}$ yields a point $p_{norm} \in [0; 2^k]^3$. The coordinates for the indices of c_p are then computed as $x_p = min(\lfloor p_{norm_x} \rfloor, 2^k - 1)$ and correspondingly for y_p and z_p.

4.3 Schwarzwald - A fast point cloud indexing system based on task parallelism

At the time of the original publication in which we introduced the *Schwarzwald* system [16], the two fastest available systems for creating a visualization-optimized point cloud index (the *Modifiable Nested Octree* index) were the 1.7 version of *PotreeConverter*, as well as *Entwine*. Our goal was to implement a system with better scalability and runtime performance than the existing tools, while maintaining similar visual quality. We

will cover the main design decisions for our indexing algorithm and the *Schwarzwald* reference implementation in this section and then compare it to existing systems in Section 4.5.

4.3.1 Design of the Schwarzwald system

The main design goals for the *Schwarzwald* system were:

- Design goal 1 - Scalability: Index creation duration should scale inversely proportional to the number of CPU cores, scalability should be linear
- Design goal 2 - Efficiency: It should make better usage of existing hardware resources than the fastest indexing tool (at the time) *Entwine*, which also scales with the number of available CPU cores
- Design goal 3 - No size limit: There should be no limit on the number of points in the point cloud that can be indexed. In particular, the system has to be fully *out-of-core*, being able to process point clouds that are larger than the available amount of working memory
- Design goal 4 - Sampling: Support both grid-based and blue noise sampling

These design goals are achieved through a series of observations and implementation decisions:

- The indexing process can be modeled as a recursive task graph where each task processes one node. This task graph can then be scheduled onto an arbitrary number of CPU cores and thus achieves design goal 1 (scalability).
- Processing does not have to be exclusively top-down or bottom-up. This aids in achieving design goal 1 by preventing poor parallelization when processing the first few levels of the tree, where most points will fall into the same node.
- Sorting points by Morton indices enables fast identification of independent points. This makes the indexing algorithm very efficient and also allows for better scalability, serving both design goals 1 and 2 (scalability and efficiency).
- We process point clouds in batches and store the results of each batch immediately to disk. This is the same approach that both the legacy-version of *PotreeConverter* and *Entwine* use, and achieves design goal 3 (no size limit).

- Sorting points by Morton index effectively puts the points onto a grid structure, hence making grid-based sampling possible with little additional effort. Blue noise sampling is possible without disturbing the sorted point order by using a stable partitioning algorithm with the hash-grid-based Poisson-disk implementation of *PotreeConverter*. This achieves design goal 4 (sampling).

The next sections cover these observations and their implications in more detail.

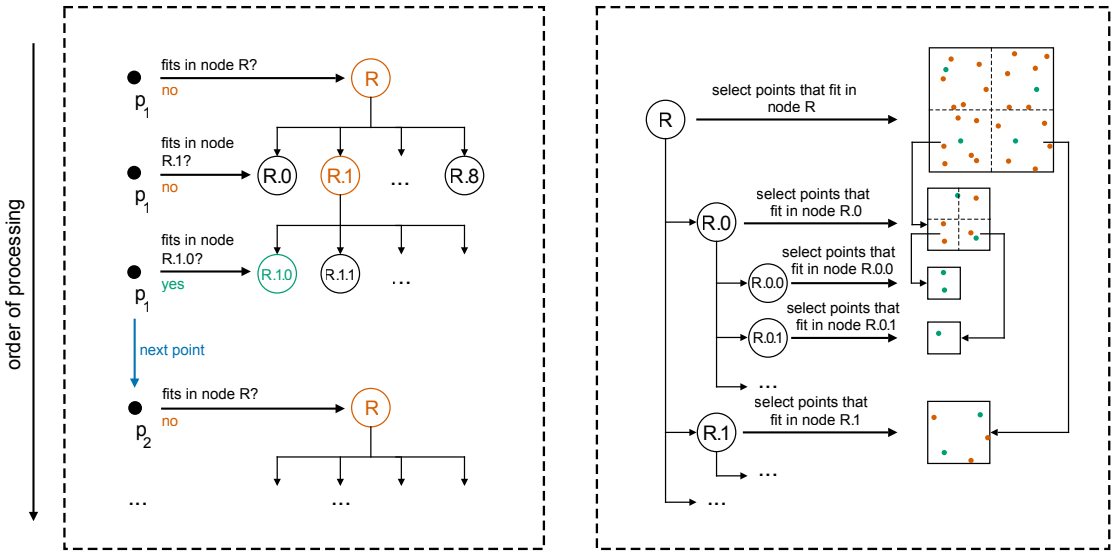
4.3.2 Modeling the indexing process as a recursive task graph

Task-parallel programming is a common method to speed up processing by using concurrency. The idea is to split an algorithm into separate execution blocks called *tasks* and model their dependencies as a graph, which can then be scheduled onto a parallel or distributed system. There are many systems and tools for realizing task-parallel programming, for a general overview Thoman et al. provide a good taxonomy [146]. For our implementation we chose the *cpp-taskflow* library [65] since it is lightweight and has good performance characteristics.

By modeling the point cloud indexing process as a task graph, we can utilize the task-parallel programming pattern to distribute the indexing work onto all available CPU cores. The actual amount of achievable parallelism depends on the structure of the task graph and on the maximum number of independent tasks at any point during processing. As is often the case with parallel programming, reducing data dependencies between tasks is crucial to achieve a high degree of parallelism. The way we process the points therefore plays an important role in meeting this goal.

The two main sets of data in our system are the points as well as the nodes of the acceleration structure. Therefore, there are two ways of indexing the point cloud: either iterate over all points, checking each point against all nodes until a matching node is found (Figure 4.2a), or iterate over all nodes, selecting all matching points for the current node from the set of all points (Figure 4.2b). Since each node can contain multiple points, but each point only belongs to a single node, parallelizing over the points would require extensive synchronization as two points in different tasks might fall into the same octree node. Instead, we chose to create one task per octree node and check all possible points for their affiliation to this node.

Since we create a nested octree index, the resulting data structure will contain points in both leaf nodes and interior nodes. We have to decide where the points in the interior nodes will be sourced from. If we start with a regular octree without LOD support, all points from the input point cloud will be distributed to leaf nodes. An interior node then has to store a subsampled version of the points in its up to eight child nodes (up



(a) Find matching node for each point (b) Find matching points for each node

Figure 4.2: Indexing a point cloud can be performed by either finding a matching node for each individual point (a) or by finding all matching points for each individual node (b). Image source: [16]

to eight since some nodes might be empty). The points in the interior node can either be selected by copying points from the child nodes into the interior node, or by moving them from the child nodes to the interior node. Both approaches have their respective advantages and disadvantages: Copying is simpler because it does not affect the memory layout of the child nodes, but it does duplicate points and hence increases the overall size of the indexed point cloud. During rendering, when both interior and leaf nodes are rendered at the same time, these duplicate points do not contribute to the rendered image. Moving some points from child nodes to their parent node during sampling fixes this problem, as each additional node that is rendered only contains unique points. It does however introduce a data dependency between the parent node and its child nodes. To keep the computations efficient, moving points requires that we first sample the points for the parent node and then distribute the leftover points to the child nodes to repeat the sampling process there. As a consequence, sibling nodes in the octree can be processed independently, but nodes in a parent-child relationship have to be processed with the parent node first and then the child nodes. In addition, the full structure of the task

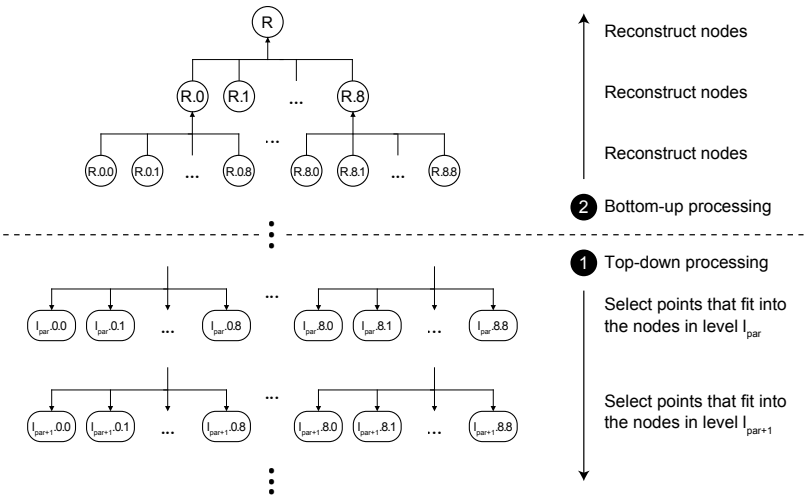


Figure 4.3: The hybrid top-down, bottom-up approach first skips all levels from the root to level l_{par}. It then inserts points into l_{par}, l_{par+1}, etc. At the end, it reconstructs the levels above l_{par}.

graph for a single batch cannot be known in advance. Each node has to first be processed before it is clear how many—if any—points remain to be sorted into the children of this node. This is a direct consequence of the online nature of the sampling algorithm, as well as its dependence on the order in which points are processed.

Putting this structure into effect yields a recursive task graph, where processing starts at the root node of the tree, followed by up to eight independent tasks for each of the children of the root node. Each of these children are in turn followed by up to eight independent tasks. This process continues until each point is assigned to a node. This way, the deeper the processing progresses into the octree, the more independent tasks are present in the task graph and the higher the degree of achievable parallelism is (see also the lower half of Figure 4.3).

4.3.3 Hybrid top-down, bottom-up processing

The task graph introduced in the previous section models a full top-down processing scheme. Top-down processing always starts at the root node and progresses down into the tree. It is used in both *Entwine* and *PotreeConverter* v1.7, where each point is first checked against the root node, then passed on to the appropriate child node if it does not fit into the root node, and so on. In contrast, bottom-up processing starts at the leaf

nodes of the tree and moves up towards the root node and reconstructs the upper nodes from the lower nodes. Examples for bottom-up processing are *PotreeConverter* v2.0, as well as the multi-way k-d tree by Goswami et al. [55].

Both approaches have benefits and drawbacks: Top-down processing touches each node only once, whereas bottom-up processing may access a single node multiple times, which results in more I/O load. However, with top-down processing, every point has to be checked against the root node. Since there is a very high probability that the point will be rejected, this results in a high number of checks that cannot easily be parallelized. Bottom-up processing, on the other hand, immediately starts at the leaf nodes of the tree and enables trivial parallelization over all leaf nodes.

Our system uses a hybrid approach that starts at some level l_{par} in the octree where the number of non-empty nodes (i.e. nodes whose bounding boxes contain some of the points of the current batch) is at least as big as the desired level of parallelism. This guarantees that all threads are immediately saturated with indexing tasks. Since every node has, on average, more than one non-empty child node, the number of available tasks for scheduling remains constantly higher than the number of available threads. As the workload of each of these tasks may vary drastically, having a large number of tasks to schedule helps counteracting these variations. This hybrid approach skips all octree levels above l_{par}. The nodes are reconstructed from those at level l_{par} in a final postprocessing step after all batches have been processed. The whole process is depicted in Figure 4.3.

Determining the right value for l_{par} is done by analyzing the first batch of points, counting the number of nodes that contain more than k points at each level. l_{par} is defined to be the first level where there are at least as many nodes containing k or more points as there are logical cores on the current machine. k is chosen to be large enough that there is substantial work involved in sampling from k points, but not so large that the probability of finding enough nodes with at least k points in a single batch becomes too small. In our implementation, we chose $k = 100'000$ and also defined level 6 to be the maximum allowed value for l_{par} to prevent skipping too many nodes in certain edge cases.

4.3.4 Quickly identifying independent points using Morton indices

The remaining problems are how to determine the number of non-empty nodes at a given level in the octree and how to quickly gather all points that are contained within the bounding box of a specific node. Both can be achieved by sorting all points in a batch by their three-dimensional Morton index. This way, all points that belong to any specific node in the octree are stored sequentially in memory. Since no point belongs to more than one node, this results in a series of disjunct memory regions that can be processed independently without having to shuffle memory whenever a new point processing task

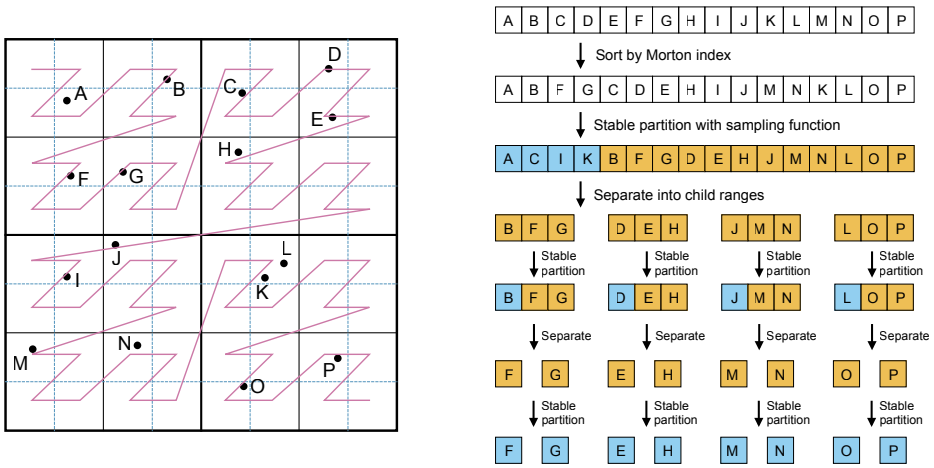


Figure 4.4: Overview of the process of sampling points for each node (illustrated in 2D and with full top-down processing for brevity). On the left, example points and the resulting Z-order curve are shown. On the right, a step-by-step overview illustrates how points are sampled at each node, with selected points in blue and remaining points in orange. Image source: [16]

is scheduled. The split positions that indicate where the range for one node ends and the range for the next node begins can be identified in logarithmic time using binary search. Using Morton indices to group points has the additional advantage that no octree data structure has to be kept in memory, the octree structure is implicit within the order of the points.

Figure 4.4 depicts the indexing process. The diagram uses full top-down processing for brevity. The resulting task graph is illustrated in Figure 4.5 and contains the following steps:

1. Morton index calculation and sorting, trivially parallelized using the fork-join pattern
2. Merging the sorted ranges of step 1 into one range for each node at level l_{par}
3. Processing each non-empty node at level l_{par}
   3.1 Processing a single node, which is comprised of the following steps:

(a) Loading points from disk that were selected for this node in previous batches

(b) Merging new points in this batch with previous points

(c) Applying the sampling method to each point to select all points that belong to the current node

(d) Storing selected points on disk

(e) Splitting remaining points into up to eight disjunct ranges containing all points that fall into the bounding boxes of each of the child nodes

(f) Processing each of the child nodes as a new task, starting again from step 3.1

Step (d) writes the points for each node into a separate file. The structure of such a file defines how applications can use the resulting index structure. Our system is able to write files in both the expected file format of *Potree* as well as the standardized *3D Tiles* file format. While it seems trivial to support various output file formats, there are some caveats which are explained in Section 4.3.6 due to the fact that our system uses the same file format for caching intermediate results of each batch as well as for the final point cloud index.

4.3.5 Sampling methods

	White noise	Blue noise	Grid-based
PotreeConverter v1.7	No	Yes (*Poisson-disk*)	No
PotreeConverter v2.0	Yes	Yes (*Poisson-disk*)	No
Entwine	No	No	Yes
Schwarzwald	No	Yes (*Poisson-disk* and *jittered*)	Yes (*grid-first* and *grid-center*)

Table 4.1: Supported sampling methods for several point cloud indexing tools

Our system supports a variety of sampling methods for selecting representative point subsets for each node, illustrated in Table 4.1 alongside a comparison with the available sampling methods in other point cloud indexing tools. We implemented two grid-based sampling methods and two blue noise sampling methods in our tool. The first grid-based sampling method is named *grid-first sampling* and selects the first point that falls into a grid cell, while the second method is named *grid-center sampling* and inspects all potential points within a grid cell to select the one closest to the center of the grid cell. Since the

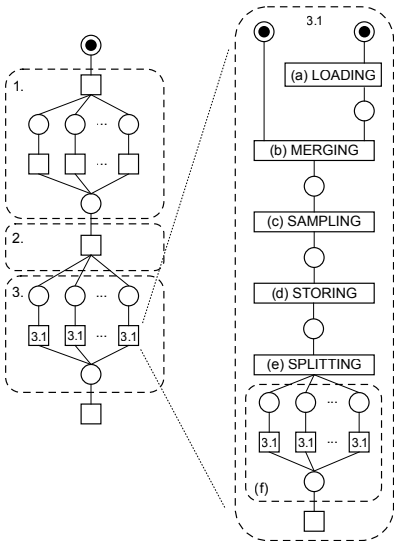


Figure 4.5: The task graph for the indexing process in our system. The recursive nature of processing can be seen with task 3.1, which processes a single node and calls itself recursively. Image source: [16]

points are sorted by Morton index in our system, no additional data structure is required to locate all points within a cell.

For blue noise sampling, the first method we implemented is the *Poisson-disk sampling* method of *PotreeConverter* v1.7, which iterates over all potential points for a node and selects a point only if its distance to all previously selected points is not less than the allowed minimum distance. We use the same hash-grid implementation as *PotreeConverter* v1.7 to speed up distance checks. It is worth noting that we chose the hash-grid based Poisson-disk variant over the improved Poisson-disk sampling used by *PotreeConverter* v2.0, even though the latter gives better visual results. The reasoning behind this decision was that the improved Poisson-disk sampling method requires sorting points by their distance to the center of the current node, whereas the hash-grid variant works with points in any order. Implementing this sorting in our system would defeat the purpose of our algorithm, which is based on sorting points by Morton index. Points would have to be sorted first by Morton index, then by distance to center prior to sampling, then again by Morton index immediately after sampling. Preliminary tests showed that this excessive sorting roughly doubles the runtime of our tool. We discuss the implications of

this finding in Section 4.5.

Our second blue noise sampling method, called *jittered sampling*, is based on the Correlated Multi-Jittered Sampling approach by Kensler [75]. For this approach, we subdivide the node into a regular grid and define one target point for each grid cell. In each grid cell, we select the closest point to a specific target point within the cell. In contrast to *grid-center sampling*, where the target point is the center point of each grid cell, *jittered sampling* uses a unique target point for each grid cell based on permutations of the canonical grid as shown by Kensler. To maintain the blue noise property, we restrict the shuffling to balanced permutations as suggested by Roberts [124]. We provide a small number of different permutations that are selected based on the level of the current node to generate the cell target positions. Since the permutations are cyclic, point spacing between points in adjacent nodes is implicitly maintained, which improves the visual quality without significantly increasing the computational cost compared to *grid-center sampling*.

4.3.6 Implementation details

The design goal for efficiency and scalability is met not only by algorithmic decisions, but also by a series of code optimizations. While most of them are not necessary for the algorithm itself to function correctly (supporting lossy file formats being the exception), we still believe these engineering feats to be of interest to other implementers of point cloud processing software, so we explain the most important implementation details in this section.

Internal point representation

The data layout of point data in memory during processing can have an impact on the performance, as we already established in Chapter 3. Of the many attributes that formats such as *LAS* define, during the indexing process the computationally expensive steps, such as Morton index calculation or sampling, only require the positions of points and no other attributes. We thus chose a columnar memory layout for the internal point buffer that *Schwarzwald* uses. Storing all positions in a contiguous memory region exhibits good cache locality, thus increasing performance. The only other tool that we found that uses a similar memory layout is the most recent version 2.0 of *PotreeConverter*, while all other tools use an interleaved memory layout.

Parallel file reading

A sometimes overlooked factor for the performance of point cloud indexing systems is the read and write performance or various point cloud data formats. The task graph shown in Figure 4.5 is able to saturate potentially dozens of threads with indexing tasks, but only if enough points can be read from the input files during the same time. The file format evaluation of Section 3.4.1 showed that reading compressed *LAZ* can be up to an order of magnitude slower than reading uncompressed *LAS* due to the computational overhead of decompressing the data. To prevent the indexing tasks from starving for points, we thus parallelize the point cloud loading from disk using a second task graph consisting of multiple read tasks that concurrently decode the read point data and fill the internal point buffer. Both the indexing task graph and the reading task graph are executed concurrently. We choose appropriate fork factors for both task graphs so that the sum total of the fork factors does not exceed the total number of available threads. While it is common to have more threads than logical cores for I/O heavy work, the main bottleneck in our case is not I/O per se, but instead the computational overhead of compressing/decompressing the data. Choosing the actual fork factors for reading and indexing is either done through a fixed allocation of available threads at program startup, or using an adaptive strategy that calculates reading and indexing throughput factors for each batch and adjusts the fork factors accordingly. Similar to *Entwine*, parallel file reading is implemented with file-level granularity. The maximum fork factor for the reading task graph is thus limited by the number of files. This can have detrimental effects on the runtime performance, which is discussed in Section 4.5.

Supporting lossy file formats

During indexing, we write the selected points at each node to disk in the desired output file format (*Potree*-format or *3D Tiles*). These files serve as a way for caching data during processing, while at the same time being the final output data once processing is finished. Using the same file format for caching and output data is efficient as no postprocessing step is required to convert intermediate results into the final data format.

However, depending on the output file format, some additional work may be necessary. If during a batch a node is being processed for which data exists on disk, our system loads these points from disk and calculates the Morton indices for these points again. When using lossless file formats such as *3D Tiles* the Morton indices are recomputed exactly. Since we use a stable partitioning function during sampling, all selected points for each node are still sorted by Morton index when being stored to disk, and thus remain sorted after retrieval. For lossy file formats such as *LAS*, which quantizes floating point values

as 32-bit integers, there might be rounding errors, resulting in slight differences between the recalculated and original Morton indices. This can break the sorting of the points. The only way to fix this is by sorting the points again after reading from disk. As a consequence, the indexing process is slightly slower when outputting points in the *LAS* file format as compared to the *3D Tiles* format.

4.4 Scalable point cloud indexing in a Cloud environment

In this section we discuss how to bring point cloud indexing into a Cloud-environment to achieve horizontal scalability. Building on the core ideas of *Schwarzwald*, we develop a distributed point cloud indexing system that supports scalable processing and storage within the Cloud.

4.4.1 Adapting Schwarzwald for the Cloud

The task-based processing model of *Schwarzwald* maps well onto compute-resources of a single machine, but it assumes uniform memory access to the point data for all tasks. This does not translate well onto distributed systems, so we had to adjust our algorithm in order to scale beyond a single machine. We still use Morton indices as a core structure for distributing point workloads, but the increased overhead of distributing data onto workers in the Cloud requires a different approach. We used the distributed computing framework *Apache Spark* [164] to achieve horizontal scalability and mapped the indexing process to the *Map-Reduce* paradigm [34]. The adapted algorithm is explained in Section 4.4.3.

We also wanted to achieve good scalability for the storage layer, while allowing simple access to the indexed data, so we decided against storing indexed nodes as files in the local file system. We considered options such as *Hadoop Distributed File System (HDFS)* [42] but ultimately decided on using a distributed database (*Apache Cassandra* [41]), which has multiple advantages. First, both regular and distributed file systems do not scale well when storing a very large number of small files [35]. Second, we get increased flexibility over how we want to query the data. Lastly, *Apache Cassandra* works decentralized, eliminating a single point of failure. Evaluations of different point cloud DBMSs also indicate that *Apache Cassandra* has the best performance for storing LiDAR data in a database [11].

Going beyond pure indexing performance, a Cloud-based point cloud indexing system should also be able to deal with changes in the data, in particular expansions of the data. Point cloud acquisition is often an ongoing process which continuously produces results, either from the same location at different times [154] or a large area scanned step by step,

such as the *AHN* dataset from the Netherlands [2]. Expansion is possible with regular octrees [155], but the LOD requirement of the *Modifiable Nested Octree* data structure prevents this, so we extended the *Modifiable Nested Octree* structure to be more suitable to data updates.

4.4.2 Adjustments to the Modifiable Nested Octree data structure

We extended the *Modifiable Nested Octree* structure into the *Hybrid MNO Grid* by combining it with a hash-based grid, which is depicted in Figure 4.6. It combines the advantages of a hash-based grid with that of the *Modifiable Nested Octree*: The grid allows adding new areas with little overhead, while each grid cell contains a full *Modifiable Nested Octree* that supports high-quality LOD. The size of the grid cells can be used to control the depth of the octrees as well as a granularity for progressive index construction. Adding new data only requires recreating an existing *Modifiable Nested Octree* if the data falls into an existing cell.

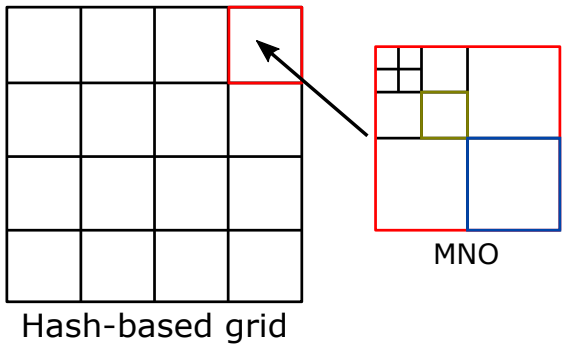


Figure 4.6: The *Hybrid MNO Grid*, a combination of the *Modifiable Nested Octree* data structure and a hash-based grid. Image source: [76]

Storage-wise, we store all points for a single *Modifiable Nested Octree* node as a *Binary Large Object (BLOB)* in the *Apache Cassandra* database. This is the *blocked* storage model discussed by VanOosterom et al. [150], which drastically reduces the number of database entries. To identify nodes within the database, we use a combination of the *Grid ID* and *Node ID* of a node. The *Grid ID* represents the location of the grid cell of the *Modifiable Nested Octree* of the current node within the hash-grid. The *Node ID* represents the position of the node within the *Modifiable Nested Octree*, including its LOD (see Figure 4.7). We use 64-bit Morton indices to represent both IDs, which are combined into a single

128-bit ID that acts as the primary key for database entries.

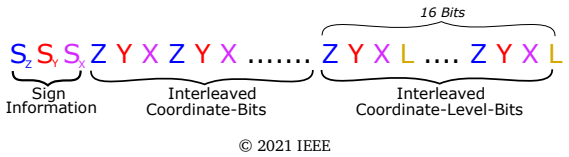


Figure 4.7: The *Node ID* of the *Hybrid MNO Grid*, a 64-bit Morton index that includes LOD information in the 16 least significant bits. Image source: [76]

4.4.3 A Map-Reduce algorithm for point cloud indexing

We designed a Cloud-first point cloud indexing algorithm based on the Map-Reduce paradigm. To achieve good scalability and performance, the algorithm has to meet the following two conditions:

- A high fan factor for the Map-phase, ideally independent of the actual data distribution
- A low number of shuffle operations, as redistributing data across the nodes within a cluster is expensive

We chose a top-down processing scheme, as bottom-up processing would require too many shuffle operations. Instead of the hybrid top-down/bottom-up processing, we achieve good parallelism by first dividing the point cloud into small, disjoint sub-areas called *spacing cells*. The size of these areas is defined by the *spacing* factor, which defines the minimum distance between points in the root node of an *Modifiable Nested Octree*, identical to the *Schwarzwald* system. In the Map-phase, points within a spacing cell are sampled and assigned to their respective octree levels. In the Reduce-phase, individual spacing cells are merged into the final *Modifiable Nested Octree* nodes. The two phases are illustrated in Figure 4.8 and Figure 4.9.

In total, this algorithm requires two shuffle operations: First to group the points into spacing cells, which happens during parsing of the input data based upon the Morton index of each point, and second to group sampled spacing cells into the *Modifiable Nested Octree* nodes. For parsing points, we use the *Spark IQmulus Library* [21] to read points in parallel on each worker node. Morton index calculation works the same way as with the *Schwarzwald* system. For point sampling, we implemented the following sampling strategies:

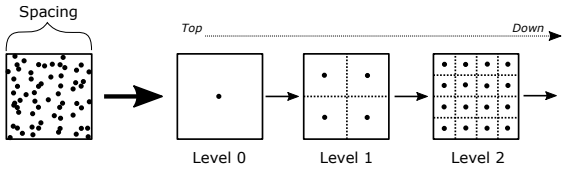


Figure 4.8: The Map-phase of the distributed point cloud indexing algorithm. Points are grouped into spacing cells, which are sampled into the individual LOD levels in the octree. Image source: [76]

- *Random*: Selects a random number of points per node
- *Grid-first*: Divides the spacing cell into virtual cells using a grid and selects the first point to fall into each grid cell
- *Grid-center*: Divides the spacing cell into virtual cells using a grid and selects the point closest to the center of each grid cell

We did not implement a blue-noise sampling method because sampling happens within each spacing-cell and not within each octree node, as in *Schwarzwald* or *PotreeConverter*. Combining multiple spacing-cells into a single node thus would not preserve the blue-noise characteristic at the edges between the spacing-cells.

The resulting processing pipeline is illustrated in Figure 4.10.

4.5 Evaluation

Within this section we show the results of a series of experiments that evaluate the runtime performance and quality of our *Schwarzwald* system as well as its Cloud-optimized variation, compared to the current state of the art in point cloud indexing. We first discuss the landscape of existing point cloud indexing tools in Section 4.5.1. Then, we evaluate the *Schwarzwald* system in a series of experiments on a single machine in Section 4.5.2, as well its Cloud-optimized variant in a Cloud environment to demonstrate horizontal scalability. To distinguish our implementations, we refer to the single-machine implementation introduced in Section 4.3 as *Schwarzwald* (Desktop), and the Cloud-optimized implementation introduced in Section 4.4 as *Schwarzwald* (Cloud).

The reference implementations for both the single-process variant of *Schwarzwald* and the Cloud-optimized variant are both available under an open-source license on GitHub [46, 45].

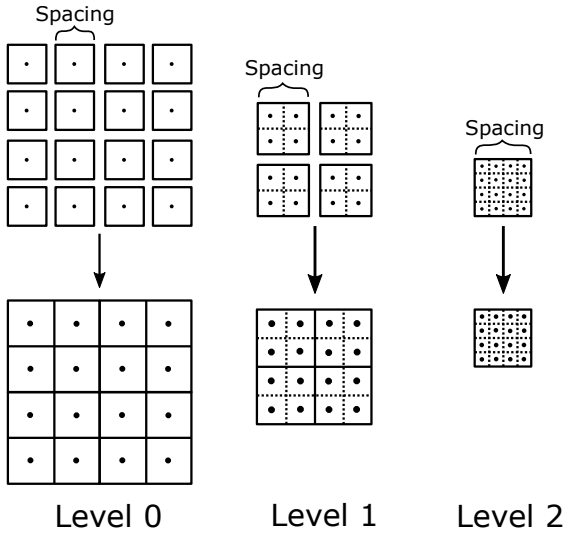


Figure 4.9: The Reduce-phase of the distributed point cloud indexing algorithm. Multiple spacing cells are combined into the final *Modifiable Nested Octree* nodes. Virtual grid within a spacing cell shown using dotted lines. Image source: [76]

4.5.1 Existing point cloud indexing systems

As the *Modifiable Nested Octree* structure has become the de facto standard point cloud indexing structure for point cloud visualization applications, there are several tools that generate an *Modifiable Nested Octree* from a raw point cloud. *PotreeConverter* is one popular tool, developed in conjunction with the *Potree* [139] web framework for point cloud visualization. There are multiple versions of *PotreeConverter*: The legacy-version (1.7), which uses a form of Poisson-disk sampling but almost no parallelism, and the more recent version 2.0, which uses a multi-stage algorithm designed for good parallelization, as well as a novel implementation of Poisson-disk sampling based on point sorting [138]. *Entwine* [61] is another system for point cloud indexing that uses grid-based sampling instead of Poisson-disk sampling. In addition to *Entwine* and *PotreeConverter*, which are available as open-source systems, there are also commercial products for point cloud indexing, most notably *Cesium Ion* [26] as well as *Arena4D* [151]. Since both *Arena4D* and *Cesium Ion* are paid-only software, we excluded them from the experiments and focused on the open-source tools *PotreeConverter* and *Entwine*. Table 4.2 gives an overview over

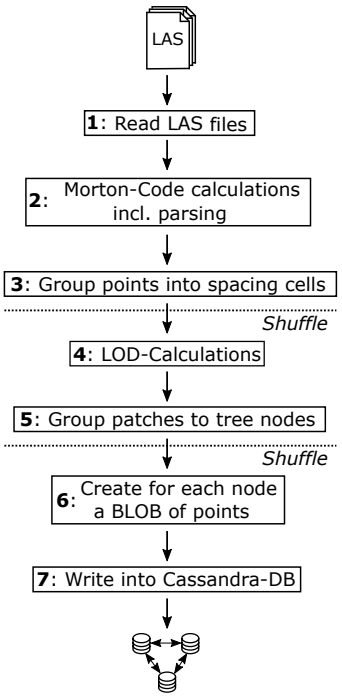


Figure 4.10: The processing pipeline for the distributed point cloud indexing algorithm. Image source: [76]

the existing tools and their capabilities, together with the systems introduced by us.

Tool	Formats (input)	Formats (output)	Platforms	License
PotreeConverter v1.7	*LAS*, *LAZ*, PTX, PLY	*LAZ*, custom binary format	Windows, Linux, MacOS, Docker	BSD-2
PotreeConverter v2.0	*LAS*, *LAZ*	*LAS* with optional *Brotli* [5] compression	Windows, Linux	BSD-2
Entwine	All formats supported by *PDAL* [29]	*LAZ*, *3D Tiles* (postprocessing), binary format with optional *Zstandard* [27] compression	Windows, Linux, macOS, Docker	LGPL2
Arena4D	*LAS*, *LAZ*, E57, PTS, PTX, ASCII	Proprietary format *VPC*	Windows	Paid
Cesium Ion	*LAS*, *LAZ*	*3D Tiles*	Web service	Paid
Schwarzwald (Desktop)	*LAS*, *LAZ*	*LAS*, *LAZ*, *3D Tiles*, binary format with optional *zlib* [36] compression	Linux, Docker	Apache License 2.0
Schwarzwald (Cloud)	*LAS*	Apache Cassandra Database	Linux, Docker	Apache License 2.0

Table 4.2: Feature comparison of common point cloud indexing tools

4.5.2 Single-process indexing

To evaluate the performance and scalability of single-process point cloud indexing tools, we conducted two experiments using the datasets described in Table 4.3 and the tools *PotreeConverter* v1.7, *PotreeConverter* v2.0, *Entwine* (version 2.1) as well as our own *Schwarzwald* (Desktop) tool. In *Experiment 1.1* we analyze the runtime performance, quality of the resulting octree, and the visual quality of the indices generated by each tool. In *Experiment 1.2* we analyze the scalability of *Schwarzwald* compared to the other tools. We excluded *PotreeConverter* v1.7 from this experiment as it uses a fixed number of two threads. For *Experiment 1.1*, we used a virtual machine in an OpenStack cloud with 8 virtual CPUs, 16 GB RAM, and a 3 TB volume residing on an HDD. For *Experiment 1.2*, we used a virtual machine in Amazon AWS, using the `m5a.16xlarge` flavor, which has 64 virtual CPUs, 256 GB RAM and block storage of type General Purpose SSD. All tools were run using Docker.

Dataset (*shorthand*)	Size	Points	Files
District of Columbia (*DoC*) [156]	24GiB (*LAS*)	854M	320
High Resolution Topography of House Range Fault, Utah (*Utah*) [22]	15GiB (*LAZ*)	2B	16
PG&E Diablo Canyon Power Plant (*CA13*) [112]	85GiB (*LAZ*)	17.7B	2337
Wellington, New Zealand 2013 (*Wellington*) [3]	248GiB (*LAZ*)	52.5B	9405

Table 4.3: Datasets used for the single-process indexing experiments

Experiment 1.1 - Indexing performance

In this experiment, we evaluate the runtime and quality of the existing point cloud indexing tools and compare then to our *Schwarzwald* tool. To investigate the effect that different sampling strategies have on the runtime and visual quality, we ran *Schwarzwald* three times to generate indices using its *Poisson-disk*, *grid-center*, and *jittered* sampling strategies. We ran *Schwarzwald* and *PotreeConverter* v1.7 with `-d 111` and *Entwine* with `--span 64` as the spacing parameters. The parameter `-d 111` for *Schwarzwald* and *PotreeConverter* v1.7 determines the maximum number of points on the diagonal of the root bounding box and is a close approximation to `--span 64`, the latter dictating the maximum number of points on a single axis. A similar spacing parameter is not available anymore with the *PotreeConverter* v2.0, so all runs with this version were conducted with the default parameters. Inspecting the code shows that *PotreeConverter* v2.0 uses a spacing parameter of 128 points per axis, which is equivalent to calling *Schwarzwald* with `-d`

222. As a result, *PotreeConverter* v2.0 generates nodes that are significantly more dense than those of all other tools. All tools generated their data as *LAZ* files in a structure readable by the *Potree* renderer, with the exception of *PotreeConverter* v2.0, which does not support writing *LAZ* files and instead writes uncompressed *LAS* files.

Dataset	*PotreeConverter* v1.7	*PotreeConverter* v2.0	*Entwine*	*Schwarzwald* (Desktop)		
				Poisson-disk	*Grid-center*	*Jittered*
Wellington	70h 39m	(10h 38m)	N/A	8h 26m	**7h 27m**	7h 28m
CA13	21h 16m	**3h 15m**	9h 29m	4h 21m	4h 8m	3h 57m
Utah	1h 47m	**16m 56s**	52m 17s	17m 26s	17m 23s	18m 12s
DoC	45m 43s	7m 36s	13m 35s	6m 25s	**5m 26s**	5m 39s

Table 4.4: Runtime comparison for indexing the test datasets of the single-process experiment with all tested tools. (Results for Wellington dataset with *PotreeConverter* v2.0 are from a run that crashed at 100% progress)

The measurements for the runtimes of all tools are displayed in Table 4.4. The results show that *Schwarzwald* consistently outperforms both *Entwine* and *PotreeConverter* v1.7, while yielding similar performance to *PotreeConverter* v2.0. The performance is heavily dependent on the dataset, with *Schwarzwald* running about 1.2 times faster on the *DoC* dataset and about 1.25 times faster on the *Wellington* dataset, but running about 1.33 times slower on the *CA13* dataset. For the Utah dataset, the runtimes of *Schwarzwald* compared to *PotreeConverter* v2.0 are about equal, with *PotreeConverter* v2.0 being slightly faster (factor 1.03). On the largest dataset with 52.5 billion points, both *Entwine* and *PotreeConverter* v2.0 terminated early due to insufficient memory. *PotreeConverter* v2.0 reported 100% completing before the crash, so we still included the runtime in the analysis. The resulting dataset however was incomplete for both *Entwine* and *PotreeConverter* v2.0. Compared to these two systems, *Schwarzwald* does not keep any state in memory between multiple batches, so as long as each batch fits into memory, *Schwarzwald* will be able to process point clouds of arbitrary size. It is also worth noting the difference in performance between the *Poisson-disk* sampling strategy and the other sampling strategies for *Schwarzwald*. The impact of the sampling strategy depends on the dataset, ranging from almost no difference in runtime for the *Utah* dataset, to about 15% performance overhead for the *Wellington* dataset.

In addition to the runtimes, we also analyzed the size and node counts of the resulting octrees, which can be seen in Table 4.6 and Table 4.5. The lowest number of nodes is produced by the *Entwine* system by a large margin. This is because *Entwine* combines multiple related nodes if they all contain only a few points. *Schwarzwald* and *PotreeCon-*

Dataset	*PotreeConverter* v1.7	*PotreeConverter* v2.0	*Entwine*	*Schwarzwald* (Desktop)		
				Poisson-disk	*Grid-center*	*Jittered*
Wellington	**10.85M**	N/A	N/A	13.28M	13.66M	15.48M
CA13	3.1M	8.3M	**1.78M**	7.65M	7.58M	8.8M
Utah	630k	408k	**257k**	485k	560k	695k
DoC	144k	235k	**87k**	186k	200k	239k

Table 4.5: Resulting number of octree nodes after indexing with all tested tools in the single-process experiment

Dataset	*PotreeConverter* v1.7	*PotreeConverter* v2.0	*Entwine*	*Schwarzwald* (Desktop)		
				Poisson-disk	*Grid-center*	*Jittered*
Wellington	**314 GiB**	(1715 GiB)	N/A	317 GiB	322 GiB	320 GiB
CA13	**100 GiB**	578 GiB	123 GiB	226 GiB	227 GiB	223 GiB
Utah	15 GiB	50 GiB	16 GiB	**11 GiB**	**11 GiB**	11.4 GiB
DoC	**4.1 GiB**	23 GiB	4.5 GiB	7.6 GiB	7.8 GiB	5.6 GiB

Table 4.6: Data sizes of the resulting datasets after indexing with all tested tools in the single-process experiment. (Results for Wellington dataset with *PotreeConverter* v2.0 are from a run that crashed at 100% progress)

verter v2.0 produce similar node counts, with slightly less nodes for our system for all but the Utah dataset. In terms of data size, *PotreeConverter* v1.7 creates the smallest datasets, which is expected because this version of *PotreeConverter* did not yet support all available attributes of the *LAS* file format. *Schwarzwald*, *PotreeConverter* v2.0 and *Entwine* do support all common *LAS* attributes. Here, *Entwine* creates the smallest datasets because it produces fewer nodes in general and avoids nodes that contain only a few points. For the compressed *LAZ* format, larger point counts in a single file enable higher gains from compression, which explains the results from *Entwine*. As mentioned in Section 4.5, *PotreeConverter* v2.0 does not support *LAZ* compression, resulting in significantly larger datasets.

Lastly, we performed a visual comparison of the resulting indexed point clouds, which can be seen in Table 4.7. All tested systems generate the same kind of acceleration structure, but the visual results are different. Using the *grid-center sampling* method, the results of *Schwarzwald* are indistinguishable from those of *Entwine*. Both suffer from certain visual artifacts due to the regular nature of the grid sampling pattern. Our new *jittered sampling* method gets rid of some of these artifacts and looks similar to the results of *Poisson-disk sampling* in many regions. *Jittered sampling* is still a grid-based sampling

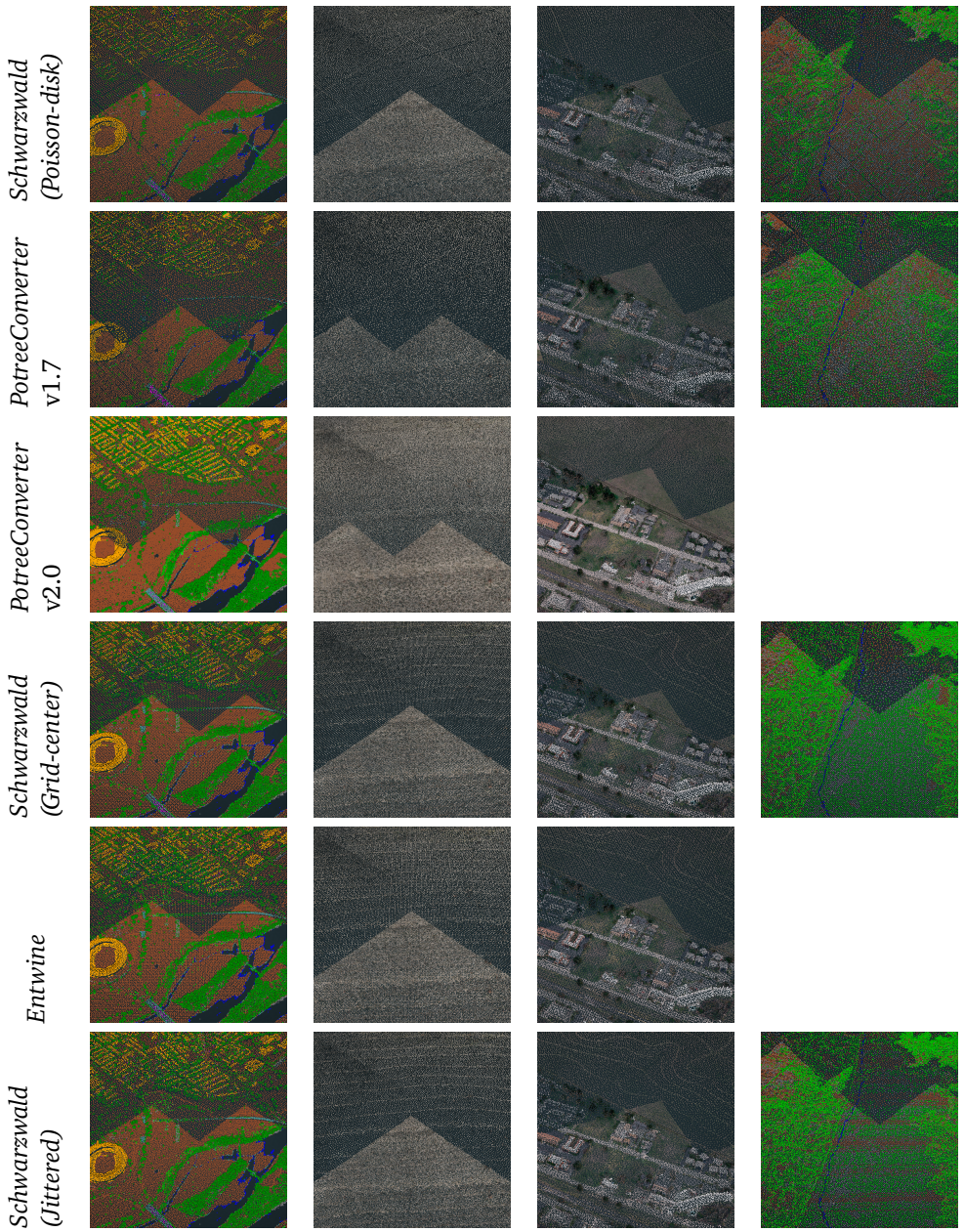
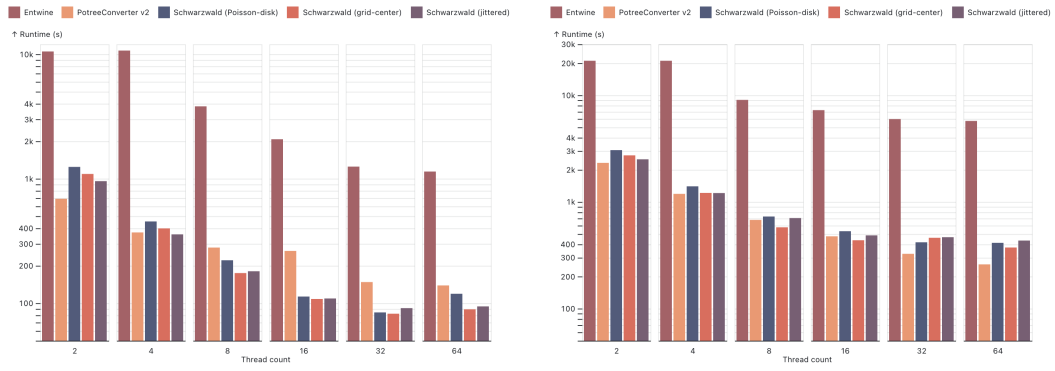


Table 4.7: Visual comparison of the resulting point cloud octrees for the *DoC*, *Utah*, *CA13* and *Wellington* datasets (from left to right). The *Wellington* dataset conversion did not finish successfully with *PotreeConverter* v2.0 and *Entwine* and is thus not included in this figure.

method and as such exhibits some artifacts that a true blue-noise sampling method does not share. With the *Poisson-disk sampling* method, both *Schwarzwald* as well as *PotreeConverter* v1.7 have certain visible artifacts, in particular an increase of samples close to the edges of the nodes, which leads to visible lines where two nodes touch. *Schwarzwald* uses the same implementation of *Poisson-disk sampling* as *PotreeConverter* v1.7, so the resulting artifacts are similar, although slightly more prominent with *Schwarzwald* due to the spatial sorting of points, which affects the way samples are chosen. *PotreeConverter* v2.0 addresses this problem by using a different implementation for *Poisson-disk sampling* and displays no line artifacts.

Experiment 1.2 - Scalability

To compare the scalability of the existing tools with our *Schwarzwald* tool, we ran the tools on the same dataset using different numbers of threads and recorded their runtimes. The resulting data is displayed in Figure 4.11.



(a) *DoC* (b) *Utah*

Figure 4.11: Runtime in relation to number of threads for *Entwine*, *PotreeConverter* v2.0 and *Schwarzwald*. Tested on the *DoC* dataset (left) and the *Utah* dataset (right).

Since *PotreeConverter* v2.0 has no explicit parameter to set the maximum number of threads, we limited its CPU usage through the `docker run` parameter `--cpuset-cpus`. *Entwine* also never uses less than 4 threads, so the runtimes for 2 threads and 4 threads are equal. We chose the two datasets *DoC* and *Utah* because they are sufficiently different to illustrate the effects of file format and number of files on the runtime performance. For processing performance and scalability, the *DoC* dataset is more favorable, as it contains a large number of small, uncompressed files. Uncompressed files are significantly faster to read, and the large number of files enables reading many files in parallel. In contrast, the *Utah* dataset consists of a small number (16) of compressed files, with extensive variation in the file size (from 57 MiB to 2.8 GiB).

All tested systems in this experiment benefit from using more threads, but to different degrees and depending on the dataset. In almost all cases, the scalability up to 8 threads including is about linear, where doubling the number of threads cuts the runtime in half. For *Schwarzwald*, the jump from two to four threads sometimes results in more than a 2x speedup, which is a result of the adaptive scheduling that *Schwarzwald* uses. With two threads, one thread is scheduled for reading points, leaving only one thread for indexing,

even if reading is substantially faster than indexing. Increasing the thread count to four, three threads can be used for indexing, which explains why the speedup is greater than 2x.

Going beyond eight threads, the scaling becomes less than linear for all tested tools. Both *PotreeConverter* v2.0 and *Entwine* benefit from going up to 64 threads, although the difference in runtime between 32 threads and 64 threads is marginal in most cases (a speedup of 1.06x-1.25x for *PotreeConverter* v2.0 and 1.04x-1.1x for *Entwine*). Our tool scales well up to 32 threads, with only a marginal speedup for 64 threads and the Utah dataset. With the *DoC* dataset, the runtime of *Schwarzwald* is about equal for 32 and 64 threads. A reason for this is the synchronization between the different threads: Since *Schwarzwald* uses a task-based parallelism approach, the synchronization overhead depends on the number of concurrent tasks and the difference in runtime of each task. For very large thread counts, a single batch of points is often not big enough to yield a large enough quantity of independent tasks, so that some threads will remain idle while others still work on tasks. Increasing the number of points in a single batch can help here, at the expense of increased memory usage.

Lastly, it can be seen that *PotreeConverter* v2.0 does benefit substantially from running on an SSD in certain cases. While *Schwarzwald* runs between 1.15x to 1.75x faster than *PotreeConverter* v2.0 for the *DoC* dataset (32 and 64 threads respectively), *PotreeConverter* v2.0 outperforms *Schwarzwald* by 1.28x to 1.6x for the Utah dataset.

4.5.3 Cloud-based indexing

To evaluate the Cloud-optimized indexing algorithm that we described in Section 4.4, we conducted another set of experiments in an OpenStack Cloud-environment. We used one *MasterNode* and multiple *WorkerNodes* for the *Apache Spark* cluster. Each node has 8 CPU cores, 32GB RAM, HDD storage and runs Ubuntu 18.04. The nodes for the *Apache Cassandra* database are located on the worker nodes of the *Apache Spark* cluster. We used three different cluster configurations with different numbers of worker nodes: A *small* configuration (4 worker nodes, 32 cores, 128 GB RAM), a *medium* configuration (8 worker nodes, 64 cores, 256 GB RAM), and a *large* configuration (16 worker nodes, 128 cores, 512 GB RAM). We used slightly different datasets than for the single-process experiments, in particular subsets with well-known sizes so that we can evaluate scalability in terms of data size. The datasets we used are described in Table 4.8.

Dataset	Size (*LAS* format)	Points	Files
CA13-S	0.1GiB	3M	1
CA13-M	1GiB	30M	1
CA13-L	5GiB	163M	1
AHN3-S	10GiB	370M	1
AHN3-M	20GiB	757.5M	1
AHN3-L	40GiB	1.48B	2
AHN3-XL	342GiB	13.13B	25
AHN3-XXL	2187GiB	81.9B	168

Table 4.8: Datasets used for the Cloud-based indexing experiments

Experiment 2.1 - Indexing performance

We evaluated the indexing performance on the *small* cluster for all datasets up to 40GB in size using the three different sampling strategies *Random*, *Grid-first* and *Grid-center*. To confirm our assumption that bottom-up processing would not scale well, we also included an implementation of bottom-up indexing using the Map-Reduce pattern, which is strongly based on the processing model of *PotreeConverter* v2.0. The resulting runtimes are shown in Figure 4.12 and confirm the assumption regarding bottom-up processing, which takes about twice as long as all other sampling methods that use top-down processing. The difference between the random and grid-based sampling methods is small, with random sampling being slightly faster. Compared to the results reported by Schütz et al. [138] for *PotreeConverter* v2.0, the difference in runtime between the different sampling strategies is significantly smaller in our tool, which indicates that the sampling process itself is not the main performance bottleneck in our tool.

We also compared our Cloud-optimized implementation to *PotreeConverter* v2.0, *Entwine*, and the single-process variant of *Schwarzwald* by indexing all test datasets on the *large* cluster. The resulting runtimes are shown in Table 4.9. Our Cloud-optimized implementation outperforms all other tools by about a factor of 3 on the largest datasets and a factor of 2 for medium-sized datasets. In addition, even the single-process implementation of *Schwarzwald* consistently outperforms *PotreeConverter* v2.0 in this experiment. This is evidence for the larger number of I/O operations that *PotreeConverter* v2.0 performs in contrast to *Schwarzwald*, causing the former tool to become I/O bound when

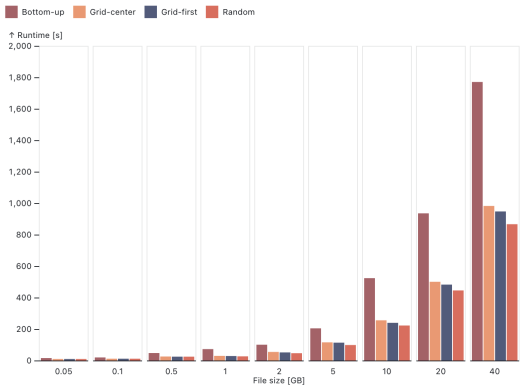


Figure 4.12: Comparison of the sampling strategies of *Schwarzwald* (Cloud) with a bottom-up implementation

using HDDs.

We also found a mistake in our initial publication from which these numbers came [76]. The runtime for *Schwarzwald* (Desktop) for the largest dataset *AHN3-XXL* is twice as large as with the other tools, even though for all other datasets *Schwarzwald* (Desktop) had similar or lower runtimes. Upon investigating we found that this is due to the large number of files written which can cause the filesystem (`ext4` in our case) to run out of inodes. Our tool does not crash but produces invalid results and a large number of error messages. We fixed this by enabling the `large_dir` feature of the `ext4` filesystem, resulting in a much lower runtime as shown in parentheses in Table 4.9.

Dataset	*Schwarzwald* (Cloud)	*Schwarzwald* (Desktop)	*Entwine*	*PotreeConverter* v2.0
CA13-S	14s	**3s**	13s	5s
CA13-M	21s	**15s**	1m 13s	19s
CA13-L	**45s**	1m 14s	5m 59s	1m 45s
AHN3-S	**1m 18s**	2m 21s	12m 11s	3m 23s
AHN3-M	**2m 34s**	4m 50s	25m 18s	7m 44s
AHN3-L	**5m 8s**	8m 47s	33m 27s	17m 32s
AHN3-XL	**1h 19m 7s**	2h 48m 13s	3h 53m 55s	3h 25m 45s
AHN3-XXL	**9h 53m 4s**	53h 10m 59s (22h 54m 17s)	28h 3m 21s	29h 51m 36s

Table 4.9: Runtime comparison of *Schwarzwald* (Cloud) with other indexing tools

Experiment 2.2 - Scalability

At the onset of this chapter, we postulated that scalability is an important property for point cloud indexing tools, which was the main motivation behind the Cloud-optimized indexer implementation. To demonstrate scalability in terms of the number of available CPU cores as well as the data size, we indexed the datasets up to 40GB on all three cluster configurations. Table 4.10 shows the runtimes of this experiment and Figure 4.13 plots the runtime in terms of data size and number of cores. It can be seen that our system exhibits linear scalability with the number of CPU cores, but only above a certain size threshold, with no measurable scalability for the smallest (0.1GB) dataset and a scalability factor of about 1.5 for the 5GB dataset. For the largest datasets, we get a scalability factor of about 1.72. The difference to the theoretical optimum of 2 comes from the overhead of data distribution to the worker nodes as well as general fluctuations in performance that are expected when working in a Cloud environment.

Dataset	32 cores	64 cores	128 cores
CA13-S	14s	14s	14s
CA13-M	29s	24s	21s
CA13-L	1m 41s	60s	45s
AHN3-S	3m 45s	2m 7s	1m 18s
AHN3-M	7m 28s	4m 25s	2m 34s
AHN3-L	14m 29s	8m 29s	5m 8s

Table 4.10: Runtime of *Schwarzwald* (Cloud) for different datasets with different numbers of CPU cores

Experiment 2.3 - Successive indexing

Lastly, we evaluate the performance of successive indexing with *Schwarzwald* (Cloud). For this, we took a subset of files from the *AHN3* dataset and first index them in a single step, and then in multiple consecutive steps using different numbers of files that either are adjacent to each other or disjoint. A comparison of the runtimes of combined vs. successive indexing is shown in Figure 4.14 and shows that there is only a marginal overhead when indexing data successively compared to indexing all data at once. For

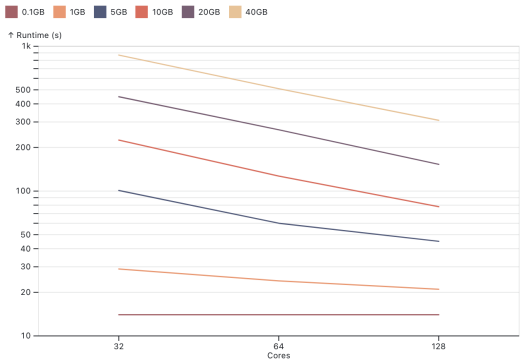


Figure 4.13: Scalability of *Schwarzwald* (Cloud) in relation to the number of CPU cores and the size of the dataset

non-adjacent regions, there is no significant runtime difference, and for adjacent regions the overhead is about 5%.

4.6 Discussion

In this section we discuss the results from the experiments with regard to research question 2. We also point out limitations and remaining challenges.

4.6.1 Implications for research question 2

We set out to answer the following research question regarding batch-based point cloud indexing:

RQ2 How can the runtime of current point cloud indexing tools be improved?

Using the experimental data shown in Section 4.5 we can now answer this question.

We developed a new algorithm for creating the *Modifiable Nested Octree* index structure using task-based parallel programming. The evaluation of *Schwarzwald*, our reference implementation for this algorithm, shows a significant decrease in runtime compared to several existing point cloud indexing tools. While our data does not give a detailed breakdown into I/O and compute effort, we can make some educated guesses where the bottlenecks of the existing tools (*Entwine* and *PotreeConverter* v1.7) lie, and whether parallelization does have a positive impact. The comparison with *PotreeConverter* v1.7 is

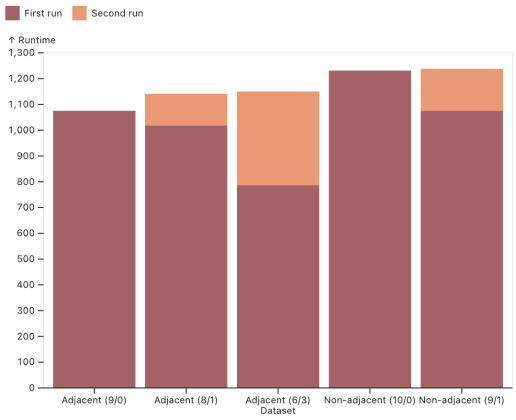


Figure 4.14: Comparison of runtime performance of *Schwarzwald* (Cloud) for combined indexing of multiple files vs. successive indexing. Numbers in parentheses indicate file counts for first and second run.

especially interesting, as *Schwarzwald* uses the same library (*libLAS*) for point I/O. The significant performance improvement of *Schwarzwald* over *PotreeConverter* v1.7 indicates that the majority of the indexing runtime is due to compute-intensive work and not due to I/O, as *PotreeConverter* v1.7 only uses two concurrent threads regardless of the number of available CPU cores. For *Entwine*, which does scale to all available CPU cores, the performance difference is harder to trace. A potential reason could be that *Entwine* uses *PDAL* for point I/O, which—as we measured in Section 3.4.1—is slower than *libLAS*. More likely though this is due to differences in the implementation of the indexing algorithm since the performance differences between *Entwine* and *Schwarzwald* are independent of the actual data format, whereas the throughput difference between *PDAL* and other tools as measured in Section 3.4.1 was dependent on the file format.

Regarding the computational complexity of indexing, we see clearer results in Section 4.5.2 with the evaluation of different sampling strategies in *Schwarzwald*. *Poisson-disk* sampling is measurably slower for the *DoC* and *Wellington* datasets, with marginal differences for the *CA13* dataset and no difference for the *Utah* dataset. While there is some dependence on the distribution of points in the dataset, overall the choice of sampling strategy has a measurable but small effect on the runtime of point cloud indexing tools. Specifically, a blue-noise sampling method such as *Poisson-disk* sampling is computationally more expensive due to adjacency checks between points compared to grid-based sampling methods, which have a constant number of distance checks. Similar

findings can be seen in the initial publication for *PotreeConverter* v2.0 [138], where a random sampling strategy is significantly faster than blue-noise sampling, though their blue-noise sampling implementation is using sorting which increases the computational complexity and hence the measurable effects.

Lastly we want to point out that the distinction between I/O and compute performance does not capture all the nuances of performance in systems programming, the domain in which applications such as the point cloud indexing tools discussed in this chapter fall. We use the term "compute" somewhat loosely to refer to anything going on inside the CPU, compared to fetching data from or writing data to I/O devices such as the harddrive. It has long since been known that memory access is a major bottleneck in most data-intensive applications—the so-called "Von Neumann Bottleneck" described as early as the 1970s [9]—and it is challenging to write code that saturates the *ALU* (arithmetic logic unit) and/or *FPU* (floating-point unit) of a processor. Hijma et al. list dozens of publications related to optimizing techniques for memory access on the GPU, showing the importance of memory access patterns in compute-intensive applications [59]. Both our own *Schwarzwald* tool as well as *PotreeConverter* v2.0 use columnar memory layouts for storing point data in working memory for precisely this reason.

Based on these insights, research question 2 can be answered directly. With *Schwarzwald* we demonstrated a heavily parallelized point cloud indexing system which outperformed many existing systems and has similar performance to the more recent *PotreeConverter* v2.0, which also makes heavy use of parallelization. Additionally, we demonstrated that horizontal scalability can speed up point cloud indexing even further beyond anything that is possible on a single machine.

4.6.2 Limitations, challenges, and future work

While we demonstrated competitive performance results for our tools, there are several limitations which have to be addressed. First, both *Schwarzwald* as well as our Cloud-optimized algorithm do not achieve the same level of sampling quality as the improved blue-noise sampling of *PotreeConverter* v2.0. The Cloud-optimized algorithm is limited to grid-based sampling methods due to the way data is processed in parallel. *Schwarzwald* does process whole nodes and hence can implement *Poisson-disk* sampling, but is based heavily on keeping points sorted by their Morton index. The improved blue-noise sampling method used in *PotreeConverter* v2.0 sorts points by their distance to the center of the node, which is impossible with the algorithm used in *Schwarzwald* without severely degrading performance.

Another challenge is the output format for the generated point cloud index. *Schwarzwald* works in batches and hence might process the same node multiple times. For this reason,

each node is written into an individual file. This results in a very large number of files, which is challenging to handle for the file system. It also makes copying data, especially over network, a lot slower. *PotreeConverter* v2.0 uses preprocessing that allows it to write each node only once and hence write all node data into a single file. Ultimately this is a tradeoff, as the processing scheme of *PotreeConverter* v2.0 requires more working memory than the batch-based processing scheme of *Schwarzwald*. We address this limitation by using a distributed database for the Cloud-optimized implementation.

Lastly, point cloud indexing is still a time-consuming process. Even with the optimizations that we introduced in this chapter, indexing multi-billion point datasets still takes several hours. Especially in a Cloud-environment this directly translates to cost, so further reductions in runtime due to more efficient processing are still sought after.

4.7 Conclusion

In this chapter, we introduced two systems for point cloud indexing which are optimized for parallel and distributed processing. We set out to investigate why point cloud indexing is a time-consuming process and whether parallel processing can be employed to speed this process up. To this end, we developed the *Schwarzwald* system, which uses Morton indices for distributing points onto tasks executed in a parallel processing environment. Since point cloud indexing typically creates tree-based structures, we introduced a hybrid top-down/bottom-up processing scheme that prevents the lack of parallel processing within the first few levels of the tree. *Schwarzwald* supports the most widely-used sampling methods found in other tools, in particular grid-based and blue-noise sampling. We evaluated *Schwarzwald* by comparing it to other existing point cloud indexing tools. At the time of its release, *Schwarzwald* was faster by up to a factor of 9, and it still performs competitively against the current state of the art, with similar performance, lower memory usage, but slightly worse visual quality.

Based on the core ideas of *Schwarzwald*, we then developed a massively parallel point cloud indexing algorithm meant for running in a distributed Cloud-environment. We implemented this algorithm in a reference system, using a distributed database for data storage. We demonstrated that this distributed system shows excellent scalability and outperforms all other indexing systems, including *Schwarzwald*, by up to a factor of 3 within the tested Cloud-environment. It also works with multi-terabyte datasets that would be challenging or even impossible to process on a single machine.

Based on the data we gathered we were able to answer the research question stated at the beginning of the chapter, as well as point out shortcomings and potential improvements that are expected to further improve the area of point cloud indexing.

5 Real-time point cloud indexing

"Progress isn't made by early risers. It's made by lazy men trying to find easier ways to do something."

Robert Heinlein

In this chapter, we investigate how point clouds can be indexed in real-time during the capturing process with a LiDAR scanner. We build on our findings of the previous chapter and show how task-based parallel programming can not only speed up batch-based indexing but enable real-time indexing as well. We analyze the necessary requirements for a real-time point cloud indexing system and introduce our reference implementation which generates the same *Modifiable Nested Octree* structure as the batch-based indexing tools. We demonstrate this reference implementation on a real-world sensor system using a Velodyne VLP-16 LiDAR sensor and show that all data produced by the scanner is indexed in real-time.

The work in this chapter is based on our publication "Real-time indexing of point cloud data during LiDAR capture" [19] and answers the following research question:

RQ3 Can point clouds be indexed in real-time during the capturing process with a LiDAR scanner?

We first discuss the motivation and challenges for real-time indexing in Section 5.1. Then we introduce a stream-based point cloud indexing algorithm in Section 5.2, which is based on task-parallel programming and provides various task priority functions to prioritize either high point throughput or low latency until a point is inserted into the index. We evaluate a reference implementation of this indexing algorithm in Section 5.3 and critically discuss the results and limitations in Section 5.4.

5.1 Motivation: Accessing point cloud data in real-time during capturing

In Chapter 3 we discussed reasons for improving the data-to-insight time when working with point cloud data, for example a need for explorative data analysis as can be seen in both practical (e.g. emergency response) and theoretical/research scenarios. On top of that, there are secondary concerns such as removing data duplication and decreasing cost due to expensive preprocessing. *Ad-hoc queries* are one step in this direction, but we also saw their limitations, which is why in Chapter 4 we explored how to speed up the point cloud indexing process. In both situations (*ad-hoc queries* and batch-based indexing) we worked with existing point cloud datasets, which already underwent a series of preprocessing steps by the data providers, including registration, georeferencing, and potentially spatial tiling. The last step in particular already constitutes a form of rough indexing—which we exploited to our advantage for ad-hoc query processing—so it is natural to ask: Can we perform point cloud indexing earlier in the data acquisition process, maybe even directly during LiDAR capturing?

If such an early indexing would be possible, this would enable a series of improvements over the traditional pipeline and even allow novel use cases to improve quality control and reduce cost. First and foremost, indexing point clouds during capturing would result in a change to the way point cloud data is delivered: Instead of delivering large unindexed or roughly spatially grouped files, a high-quality index structure would become the main delivery medium, which would allow more flexible data delivery while also getting rid of the typical data duplication that point cloud indexing introduces. The *3DEP* [148] dataset is a prime example for this situation: The original LiDAR data is stored in a publicly available (requester-pays) *Amazon S3* bucket, while the preferred data delivery medium is a second bucket containing indexed point clouds generated with *Entwine*. The raw data bucket contains over 300TB of data, with at least 300TB more for the indexed data[1]. Given the current pricing scheme by Amazon[2], storage costs for the duplicated data due to indexing can be estimated at about $75,000 per year.

Early indexing would also have benefits for quality control, as problems with the data could be identified sooner. Faulty data deliveries can significantly increase cost depending on how long it takes to identify the faults. For the *Fibre3D* application introduced in Section 1.1.2 we encountered this exact situation, where problems with a specific point cloud delivery where encountered by end users, which led to a long search for the root cause involving several parties. The earlier such mistakes become apparent, the easier

[1] Precise numbers not publicly available
[2] See `https://aws.amazon.com/s3/pricing/`

they will be to fix. If we could index the point clouds directly during capturing, we could provide real-time visualizations of the data as it is being captured, regardless of the number of points, which would allow operators to perform visual quality control in real-time.

While the list of potential advantages that real-time point cloud indexing would bring is long, so is the list of challenges. The main challenge is that indexing requires a localized point cloud, and not all systems support real-time localization. For our evaluation, we worked with a mobile LiDAR system that supports real-time localization through Simultaneous Localization And Mapping (SLAM), but higher-precision terrestrial and airborne LiDAR systems often are limited to localization as a post-process. We discuss potential implications of this in Section 5.3. Assuming real-time localized points are available, there are three additional challenges that currently prevent real-time point cloud indexing:

- Dealing with unknown bounds of the final point cloud
- Continuously updating an existing index with new points while supporting LOD
- Keeping up with the point output rate of the LiDAR sensor

The first problem can be solved with hybrid data structures, such as the *Hybrid MNO Grid* that we developed in Section 4.4. The second problem would be solvable with some of the algorithms discussed in Chapter 4, in particular all those which process the point cloud in incremental batches, which includes *Schwarzwald* and the legacy version of *PotreeConverter*, but these do not support an extensible index structure. *Entwine* does support indexing in multiple steps, which can be used to emulate real-time indexing by repeatedly adding to the existing index, but we show in Section 5.3 that this is not sufficient for true real-time indexing. Lastly, the throughput rates of *Schwarzwald* and *PotreeConverter* 2.0 would be enough to keep up with current LiDAR scanners, which range anywhere from a few hundred thousand [152] to close to three million [38] measurements per second. However neither *Schwarzwald* nor *PotreeConverter* support the necessary extensible index structure. Version 2 of *PotreeConverter* also requires the full point cloud for its mandatory preprocessing, so its underlying algorithm is unsuited for stream-based indexing. We thus develop a novel stream-based point cloud indexing algorithm, which we describe in the following section.

5.2 Stream-based indexing

In this section, we introduce our novel stream-based point cloud indexing algorithm which is able to perform point cloud indexing in real-time during LiDAR capturing, given that the capturing system is capable of real-time localization. Batch-based point cloud indexers already have substantial requirements, in particular support for LOD and being capable of *out-of-core* processing. To enable real-time indexing, the additional requirements listed in Section 5.1 apply, of which especially the last one (keeping up with the point output rate of the LiDAR sensor) is important. Without fulfilling this requirement, incoming points will accumulate over time and eventually cause the system to run out of memory. Additionally, we want support for real-time query updates, which allows our stream-based indexing algorithm to provide real-time updates of the data to clients as it is being captured, for example in a live-viewer. All requirements and the techniques we use to fulfill them are listed in Table 5.1. In the following subsections, we give an in-depth overview over the design and reference implementation for the stream-based indexing algorithm.

Requirement	Mandatory?	Solved by
Unknown point cloud bounds	Yes	Hybrid, extensible index structure
Incremental processing	Yes	Stream processing using task-parallel programming
Minimum point throughput	Yes	Parallelization and task-priority functions
Interactive query updates	No	Custom point cloud server implementation

Table 5.1: Requirements for our stream-based indexing algorithm, showing which ones are mandatory for real-time indexing, and which technique(s) we use to fulfill these requirements

5.2.1 Index structure

Our stream-based indexer uses the *Modifiable Nested Octree* structure to store points with various levels-of-detail. To deal with an unknown extent of the point cloud, we use a grid of fixed-sized root nodes similar to the *Hybrid MNO Grid* introduced in Chapter 4, though we are not using Morton indices for identifying points. Even though we showed in Chapter 4 that Morton indices are useful to speed up point cloud indexing, in our stream-based indexer each node keeps track of its own inbox of points, and nodes are potentially processed out of order. As a result we require less in-memory partitioning as for example

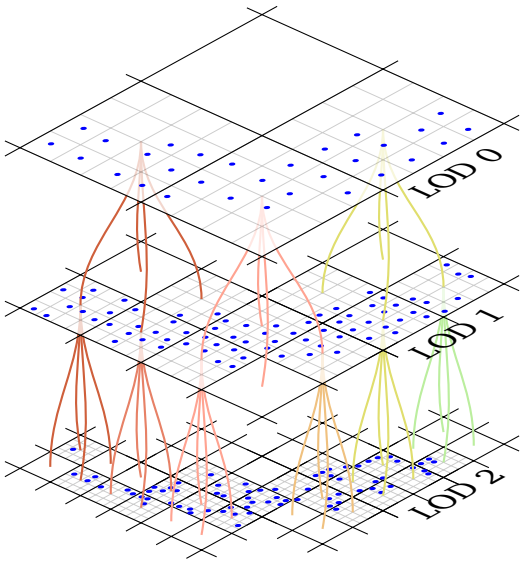


Figure 5.1: Multi-root *Modifiable Nested Octree* structure. Image source: [19]

Schwarzwald, which makes Morton indices less useful. The multi-root Modifiable nested octree (MNO) structure is illustrated in Figure 5.1. For data storage, we store each node on disk in its own file in either *LAS* or *LAZ* format. Just as with *Schwarzwald*, we cannot store all nodes within a single file as *PotreeConverter* version 2 does, since we do not know in advance how much space to reserve for each node. For processing, we use grid-center sampling because it is faster than blue-noise sampling and store the points of each node using a hash map that maps grid cells to the contained point.

5.2.2 Indexing process

To enable stream-based indexing, our indexer uses an in-memory point buffer for each node that collects incoming points for this node. We call this buffer the *inbox* for the node. This way we can insert multiple points in one step, having to load and store the updated node only once, rather than for each incoming point separately. Indexing then works using top-down processing, with new points starting at the root node of their respective octree and descending down into the tree until they fit into a node based on the sampling strategy. Similar to *Schwarzwald* indexing happens task-based where each task processes one node. The whole indexing process repeatedly performs the following steps:

- Choose a node with a non-empty inbox and take the points, leaving the inbox empty
- If the node already exists, load the current points for this node from disk
- Try inserting points into the node by using grid-center sampling
- Store the modified node to disk
- Split the rejected points into up to eight groups depending on the child-node they fall into
- Add each group to the inbox of the corresponding node

The tasks are trivially parallelized by dispatching them onto multiple worker threads, which reduces the synchronization overhead since only node inboxes have to be synchronized.

For selecting the next node to process in step 1, we propose several task priority functions which assign a priority value to each node. We always choose the node with the highest priority for processing. Our proposed priority functions are:

NrPoints Always select the node with the highest number of points in its inbox. This prioritizes processing many points together and thus reduces the number of I/O operations for swapping nodes between memory and disk. A node will only get processed once enough incoming points have been collected in its inbox.

TaskAge Always select the node whose inbox has been non-empty for the longest time. While the *NrPoints* priority function makes no guarantee for how long it takes until a node is processed, the *TaskAge* priority function guarantees that all nodes are processed in a timely manner. It ensures a relatively steady flow of points down the tree for all parts of the data structure.

NrPointsTaskAge A combination of both previous priority functions, which tries to find a compromise between the number of points that are processed together and the time until a node is processed. The following Equation (5.1) shows how priority values are calculated by combining the previous two priority functions:

$$Priority = NrPoints \times 2^{TaskAge} \tag{5.1}$$

The additional factor $2^{TaskAge}$ prevents nodes from taking arbitrary long until they are processed, which is especially helpful for small nodes or nodes where the LiDAR sensor has moved away from and thus will never receive new points. Through this factor, nodes that have not been processed for a long time are continuously increasing in their priority.

TreeLevel Always select the node with the highest level within the tree, i.e. the node that is deepest within the tree. This causes all points of a root node being fully inserted into the index before the next root node can be selected again.

The whole stream-based indexing process is depicted in Figure 5.2.

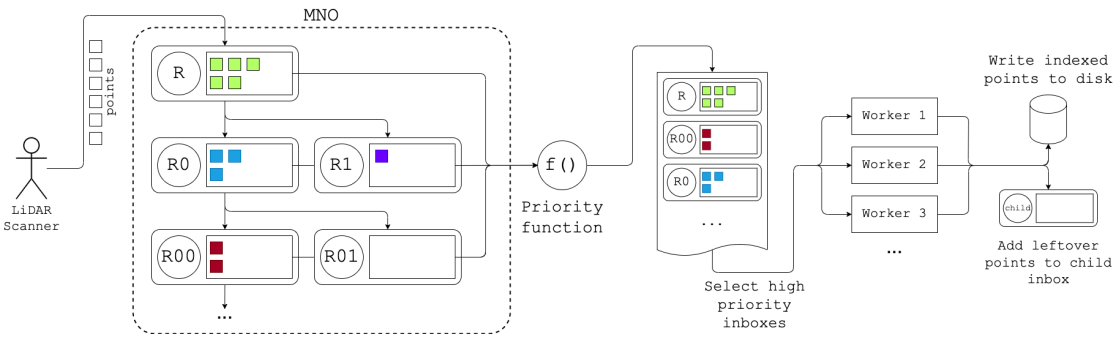


Figure 5.2: Overview of the stream-based indexing algorithm, starting at the LiDAR scanner and ending at the indexed points on disk. Image source: [19]

5.2.3 Optimizations

To reduce the number of I/O operations, we added an additional caching layer by accessing all nodes via a global Least-Recently-Used (LRU) cache.

During indexing, loading and storing of frequently accessed nodes is less expensive, because they can be accessed from the cache rather than disk. As long as the sensor does not move, points will mostly fall into the same set of nodes. If the cache is large enough to fully cover this set, the disk only needs to be accessed for nodes entering or leaving this set as the sensor moves. The cache is shared between the indexing process and query execution. This allows the point data of updated nodes to be made available to the query without requiring an extra round trip to disk. For each node, the cache can store the point data either as *LAS* encoded binary data, or in its decoded form, or both. The point data is lazily (de)serialized into the representation that is required when accessing it. This avoids unnecessary (de)serialization operations which is especially important if compression is enabled, which makes *LAZ* encoding and decoding expensive.

Small insertion operations with only a handful of points do not contribute much to the overall indexing progress, but the average cost for loading and storing the point data is the same as for any other insertion operation. We implemented an optional optimization that tries to avoid such insertion operations:

In addition to the sampled points, each node can store up to n points that have been rejected in the grid-center sampling step. We call these bogus points. When enabled, we add the rejected points to the node's bogus points after the grid-center sampling step. If this list contains less than n points, we store the node as is. Only if the number of bogus points exceeds the threshold n, we empty the bogus points list and let the points further descend down the tree.

In leaf nodes, bogus points help to avoid excessive tree heights. Here, they are equivalent to how the first version of *PotreeConverter* only expands a leaf node into eight child nodes if a sufficient number of points is exceeded. Another advantage of bogus points is that, when used together with the *NrPoints* priority function, they help counteract its problems with slowly growing inboxes in the higher tree levels.

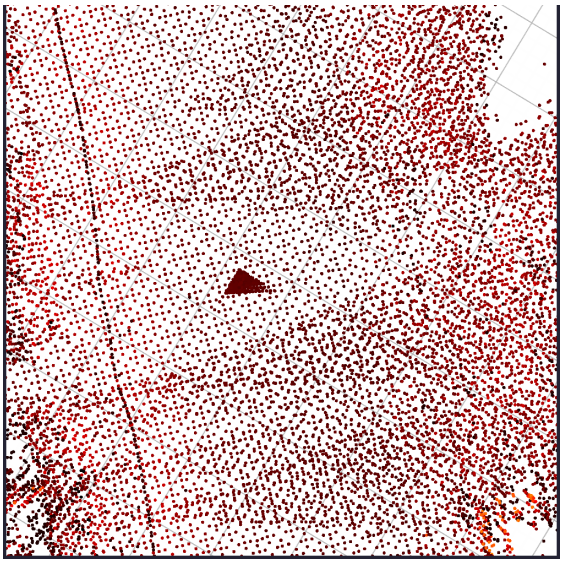


Figure 5.3: Visual artifact due to bogus points which are part of a node without being correctly sampled. Image source: [19]

The disadvantage of bogus points is that they lead to visual artifacts, as can be seen in Figure 5.3. Since the bogus points lead to a local increase of the point density, these artifacts are only visible when rendering with a relatively small point size compared to the average point distance in the selected LOD. Therefore, we deem this not to be an issue in most practical visualization applications. Alternatively, excluding bogus points from the query result hides these irregularities. However, from a correctness perspective this means that some of the captured points will never be visible in query results, as if

they were not part of the point cloud. Considering this, the threshold n has to be picked carefully, acting as a trade-off between render quality and indexing performance.

5.2.4 Implementation

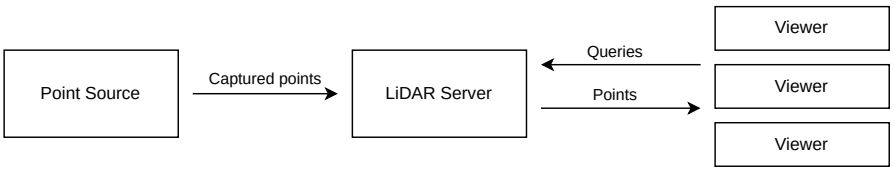


Figure 5.4: Software components in the reference system. Image source: [19]

To demonstrate the feasibility of our stream-based indexing algorithm, we implemented a reference system consisting of three network-connected components, as shown in Figure 5.4.

The core of the system is the *LiDAR Server*, a central server that is responsible for indexing and storing the point data it receives from the point source. Multiple clients can connect to the LiDAR Server and query for points. The LiDAR Server executes and serves these queries and also keeps clients updated whenever new points arrive that match the query. This is what enables live-viewing during real-time indexing.

The *Point Source* is a small application that connects the physical capturing device to the LiDAR Server component. It receives point and trajectory data from the capturing device and sends it to the LiDAR server in its expected format. Separating the LiDAR Server from the Point Source makes the whole setup more extensible, as new capturing devices or a component that plays back known data for testing purposes can be easily added to the whole system.

The *Viewer* visualizes a live view of the point cloud data as it is being captured and supports basic viewer functionality including LOD queries. An unlimited number of querying clients and at most one point source are allowed to connect to the server in the current implementation.

The source code for all components is available under an open-source license on GitHub [44].

5.3 Evaluation

In this section we perform an evaluation of our stream-based indexing algorithm using two different test-scenarios: A *synthetic test* that analyzes the limits for point throughput

and scalability (Section 5.3.1), and a *real-world test* on an actual sensor system where our software was run live during data acquisition (Section 5.3.2).

Dataset	Size	Points	Capture duration
Indoor 1	216MB	6.39M	1m 31s
Indoor 2	404MB	11.94M	2m 32s
Outdoor 1	2.7GB	89.8M	5m 23s

Table 5.2: Datasets used for the evaluation of the stream-based indexing algorithm

Name	Storage	CPU cores	Memory
Virtual Server	HDD	16	64GiB
Laptop	SSD (NVMe)	8	16GiB
Nvidia Jetson	SDD (USB-C)	8	32GiB

Table 5.3: Test systems used for the evaluation of the real-time indexing system

Table 5.2 gives an overview of the datasets that were used for the evaluation. The *Indoor 1* and *Indoor 2* datasets were generated during the real-world test, the *Outdoor 1* dataset was generated upfront with the same sensor system and used for the synthetic test. We used different test systems for the evaluation, which are listed in Table 5.3. For the synthetic test, we used both the *Virtual Server* and *Laptop* systems, for the real-world test our indexer ran on the *Nvidia Jetson* system that was integrated into the capturing setup.

The capturing setup itself is shown in Figure 5.5. It consists of a Velodyne VLP-16 hi-res LiDAR system combined with a 360-degree RGB-camera ring. Additionally, a further Intel RealSense stereo camera is integrated. The positioning system contains a Global Navigation Satellite System (GNSS) receiver combined with an Inertial Measurement Unit (IMU). Furthermore, the prediction of the position is supported by a visual odometry algorithm. The localization of the point cloud happens in real-time using a SLAM algorithm based on *RTAB-Map* [80]. The communication of the individual sensors as well as the referencing of the data runs via the Robot Operating System (ROS) on an Nvidia Jetson Xavier as embedded GPU to provide a fast data processing pipeline. The individual sensor data streams are each provided with a time stamp, which is made available

Figure 5.5: The capturing setup used for the evaluation. Image source: [19]

by the GNSS module. The scanner is synchronized via the National Marine Electronics Association (NMEA) protocol and a Pulse Per Second (PPS) signal. For the indexing of the data, only the data of the Velodyne VLP16 hi-res without color information was used. The scanner produces up to 300,000 points per second with a vertical field of view of ± 10° and has a range of 100 m.

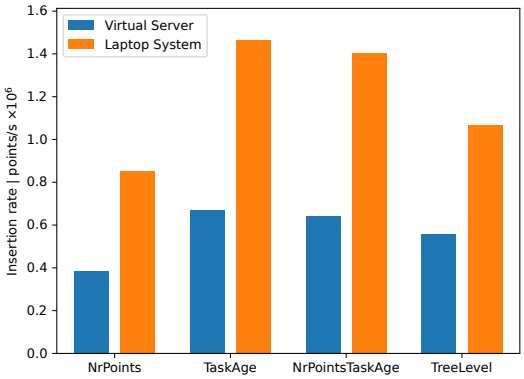
5.3.1 Synthetic test

For the synthetic test, we measured the performance of the stream-based indexer in isolation, without the surrounding LiDAR Server and without potential overhead for transmitting point data over a network connection. For testing, we used the *Outdoor 1* dataset, which was repeatedly replayed to the indexer to get reproducible results.

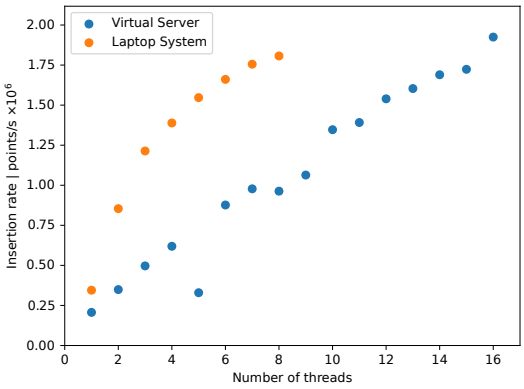
First we measured the insertion rate by monitoring the overall number of points in all inboxes. In regular intervals, we added enough new points from the dataset to fill the inboxes to a fixed number of points. New points are added as fast as previously added points are indexed. From the time for indexing the whole dataset, the point insertion rate is calculated.

Figure 5.6a shows the insertion rate for the different priority functions. The *TaskAge* function achieves the highest insert rate on both test systems, but the *NrPointsTaskAge* function also achieves similar results. The slowest function is the *NrPoints* priority function due to the issues we already described in Section 5.2.2.

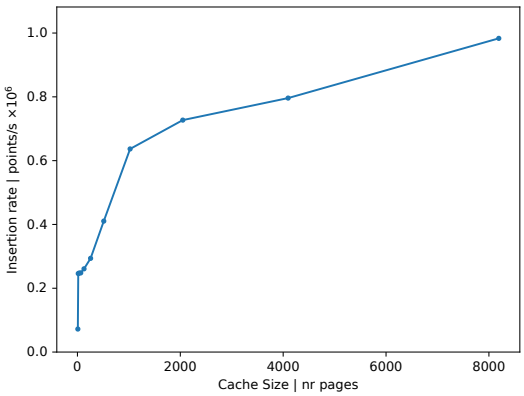
Next we measured the insertion rate in relation to the number of worker threads. Figure 5.6b shows that both tested systems scale well but to different degrees. The Laptop system scales linearly up until 4 threads are used, with more threads scalability becomes



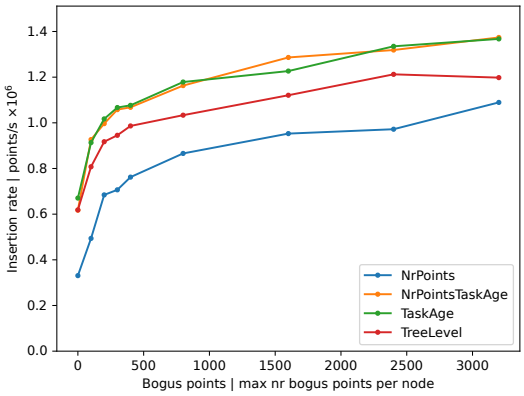
(a) Point insertion rate measured for different priority functions. Image source: [19]



(b) Point insertion rate in relation to the number of threads used. Image source: [19]



(c) Point insertion rate in relation to the size of the LRU cache. Image source: [19]



(d) Point insertion rate in relation to the number of allowed bogus points. Image source: [19]

Figure 5.6: Measurements for the synthetic test

logarithmic instead. The Virtual Server system however exhibits linear scalability for up to 16 threads.

We also analyzed the effect that the LRU cache and the number of allowed bogus points have on the point insertion rate, with the results shown in Figure 5.6c and Figure 5.6d. For the cache size, allowing as few as 32 nodes to be cached (lowest measured value) already more than doubles the point insertion rate, with linear scalability for up to 1024 cached nodes. For the number of bogus points, the situation is similar: Small numbers of allowed bogus points (200) lead to a significant improvement in performance, with diminishing returns for higher numbers of bogus points.

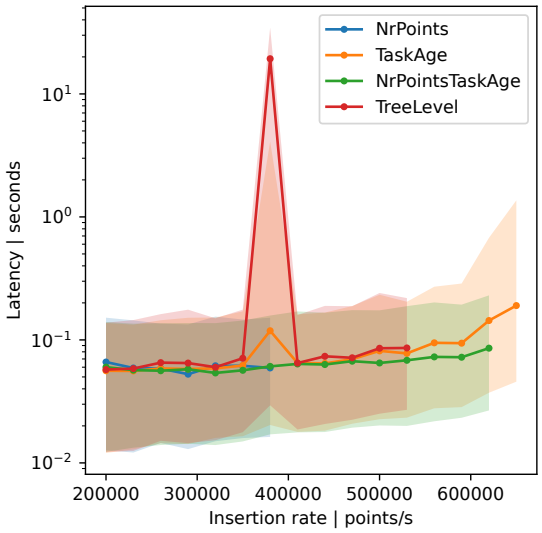


Figure 5.7: Point latency measured for different insertion rates and priority functions. Image source: [19]

For the second part of the synthetic test, we measured the latency for point insertion to determine how long it takes for a point to be indexed from the time it was passed to the indexer. To measure the latency value, the test data is replayed and indexed at a fixed point rate and a query is executed concurrently. For each point, we record the timestamp when it was passed to the indexer, and the timestamp when it was first seen in the query result. The difference between the two timestamps is the point latency, which we aggregated to the median as well as 10th and 90th percentiles. The results of this measurement for different priority functions are shown in Figure 5.7. Latencies are stable with values around 0.1s, but contrary to our initial assumption, the different

task priority functions have little effect on the point latency. The single spike for the *TreeLevel* priority function is most likely due to fluctuating performance on the Virtual Server system that the test ran on, which was hosted in a Cloud environment. In general, the point latency values are fast enough to give users a real-time impression in the viewer.

Lastly, we compared our real-time indexer to *Entwine*. As stated in Section 5.1, *Entwine* does not truly support real-time indexing. Instead, we use a feature that allows adding new points to an existing index by running the *Entwine* batch-process with new input data on a directory containing an existing index. To emulate real-time indexing, we split the *Outdoor 1* dataset into multiple files, where each file contains all points recorded during a given time interval from the start of the recording. We used 1 second, 2 seconds, 4 seconds, and 8 seconds as the target intervals, and then executed *Entwine* once for each file of the split dataset and recorded the runtime. Figure 5.8 shows the cumulative runtimes for each of the split datasets, together with the threshold at which indexing happens in real-time. *Entwine* is not able to index points with a high enough throughput to stay within this threshold, however it can be seen that larger time increments and hence larger batches of points increase the indexing throughput.

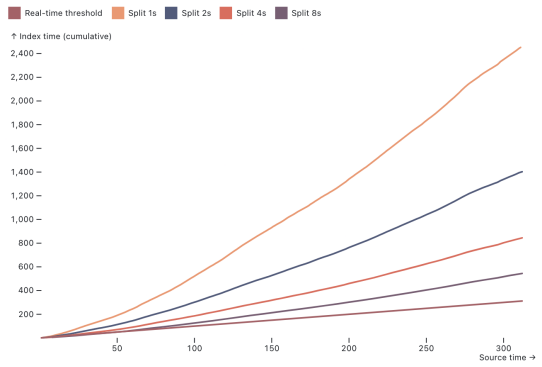


Figure 5.8: Cumulative runtimes of emulated real-time indexing using *Entwine*, together with a threshold value for real-time indexing.

5.3.2 Real-world test on the sensor system

To demonstrate our system in a real-world scenario, we ran our LiDAR Server on the Nvidia Jetson system which is part of the sensor system, while driving the sensor system through two different indoor scenes. Figure 5.9 shows an overview of one of the datasets generated during this test scenario.

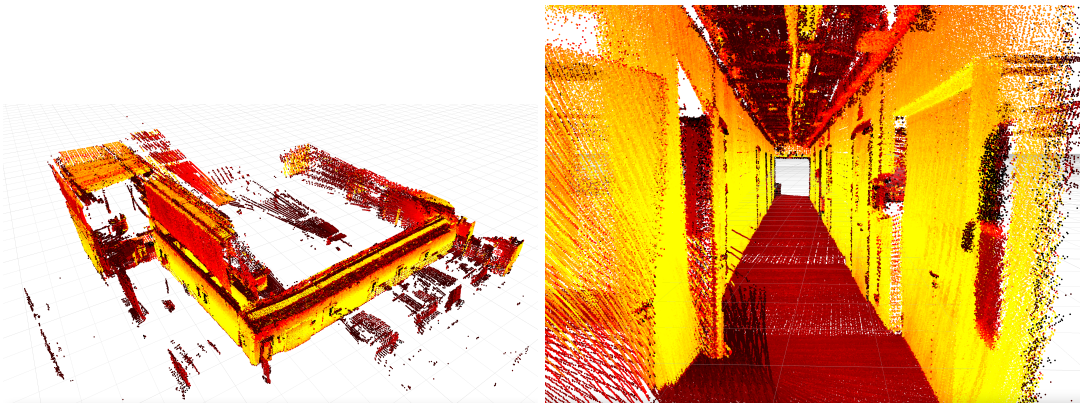
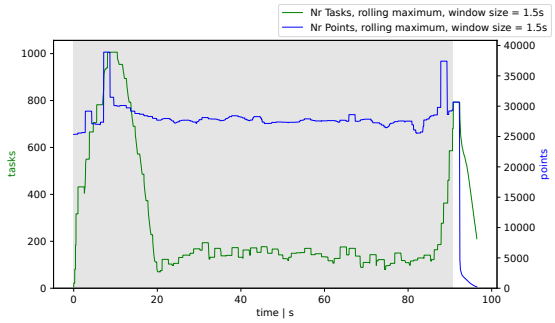


Figure 5.9: An overview of the *Indoor 2* dataset generated during the real-world test. Image source: [19]

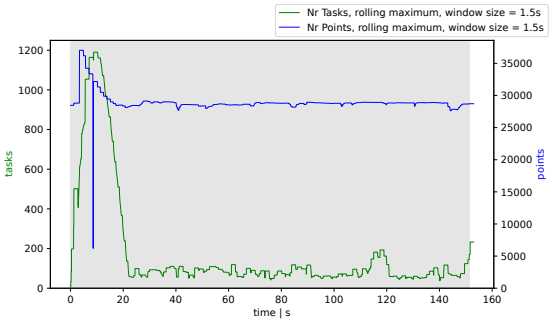
Both test runs used the following parameters:

1. Worker threads: 6
2. Priority function: *NrPoints*
3. LRU cache size: 500 nodes
4. Maximum LOD: 10
5. Output format: *LAS*
6. Bogus points: 0

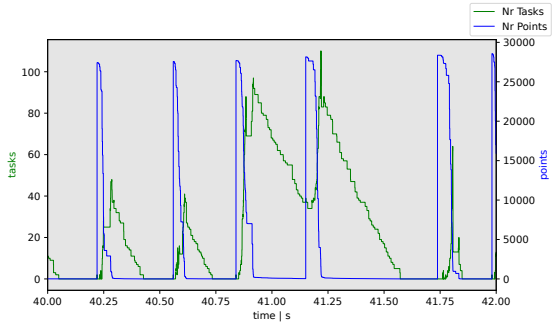
During the test runs, we recorded the number of non-empty inboxes, as well as the number of points in the inboxes. Figure 5.10 shows the results of the two runs, which indicate that the indexer is able to keep up with the incoming points from the sensor. This is the case since the number of non-empty inboxes ("Nr Tasks") as well as the point counts stay mostly constant over the runtime of the experiments. Only at the beginning and the end of the test runs do we see a spike, since the scanner system was not moving here, which caused large numbers of points to be captured at similar coordinates. In both cases, the indexer was able to quickly catch up once the system started moving. A detailed view into one of the test runs, as shown in Figure 5.10c, illustrates that the indexer completely processes all points of the current batch before the next batch arrives from the sensor.



(a) Indoor 1



(b) Indoor 2



(c) Detail view into *Indoor 2*

Figure 5.10: Capturing performance during the two real-world test cases. Gray areas indicate the time for which the LiDAR sensor was active. Image source: [19]

5.4 Discussion

In this section we discuss the results from the experiments with regard to the initial research question 3. Beyond that, we point out remaining challenges and limitations of real-time indexing.

5.4.1 Implications for research question 3

In this chapter we set out to answer the following research question:

RQ3 Can point clouds be indexed in real-time during the capturing process with a LiDAR scanner?

We now summarize the results of this chapter and then answer research question 3. We introduced a stream-based point cloud indexing algorithm that progressively builds a *Modifiable Nested Octree* index in real-time during LiDAR capture. Similar to the *Schwarzwald* system introduced in the previous chapter, we build on task-based parallel programming, with the difference that for real-time indexing, nodes are processed based on a priority queue instead of a fixed scheduling algorithm based on Morton indices. We introduced several task priority functions and evaluated their effect on overall indexing throughput as well as latency. We evaluated our implementation in both a synthetic test scenario as well as a real-world test based on a sensor setup using the popular Velodyne VLP-16 LiDAR sensor. Based on the evaluation we demonstrate that our stream-based indexer can index up to 1.8 million points per second in real-time on the test hardware.

Beyond high throughputs, we also demonstrated substantial improvements to query responsiveness: The latency measurements shown in Figure 5.7 show a responsiveness of about 0.1 seconds on average, which is fast enough for real-time visualizations. Considering the throughput limits of *ad-hoc queries* or the preprocessing time for batch-based indexing, a data-to-insight time of 0.1 seconds is several orders of magnitude faster than any of the other approaches presented in this thesis, both our own as well as in the literature.

With our stream-based indexer we have therefore demonstrated that it is possible to index LiDAR data in real-time and we established thresholds for the maximum capture rate of a scanner that still supports real-time processing. Research question 3 can therefore be answered affirmatively.

5.4.2 Limitations, challenges, and future work

While these numbers are a big improvement over the current state of the art when it comes to point cloud query responsiveness, there are several factors that limit the usability of real-time point cloud indexing as of today. These factors are:

Real-time localization The biggest hurdle for establishing real-time indexing during LiDAR capture is the dependence on a sensor system that outputs localized points in real-time. Autonomous systems often support this, as localization is mandatory for the autonomous agent. Examples for this are self-driving cars [79] or robotics [47], which are typically not the systems that capture large-scale point clouds which require sophisticated indexing. For mobile mapping systems such as Terrestrial Laser Scanning (TLS) and Airborn Laser Scanning (ALS) systems, localization happens through sensor fusion of GNSS and IMU data, potentially with correction factors that are applied in a postprocess [103]. Sticking with real-time capable localization would result in lower quality point clouds, as the positions are less accurate than those obtained from high-quality trajectories during postprocessing. In principle a multi-stage process might be conceivable, in which real-time indexing is performed based on the rough point positions and a postprocess then adjusts the index with the high-quality positions. With regular octrees, this seems reasonable, as the expected positional shift is probably small for most points, causing few points to move from one octree node to another. The nested octree structures that are commonplace for point cloud indexing seem unsuitable for changes in the point cloud due to their reliance on sampling the point cloud at various resolutions. Sampling thus distributes points vertically within the nested octree, so changes in the point cloud can cause points to move not only between different nodes on one level of the nested octree, but also between different levels of the octree. This in turn can cause a ripple-effect, where one point moving out of a node at one level can cause another point to replace it, which in turn leaves a new spot for yet another point to replace it, and so on. While the MNO structure does support adding and deleting points, which is sufficient to move points around, the overhead of locating and deleting the moved points might be larger than simply rebuilding the whole index. On top of that, moving points requires unique point IDs for all points, which is not something that is typically stored for points obtained from LiDAR. It is doable, for example by using the Extra Bytes feature of the *LAS* format, but increases the size of the point cloud.

Dependence on point distribution Like all point cloud indexing algorithms that build nested octree structures, there is a certain dependence on the point distribution.

If many points fall into the same small volume of space, they will end up deep within the tree and either exceed the maximum tree depth, cause the tree to degenerate, or simply prevent any node-based parallelization. This is challenging for a mobile scanner system whenever the scanner is running but the system is not moving, as points at similar locations accumulate over time. There are workarounds for this, such as disabling the scanner after a certain time of standing still, or limiting the precision of point positions so that multiple points that are very close together are combined into a single point. If all else fails, incoming points could always be cached to disk and processed once the indexer becomes idle to prevent memory overflows.

Quality of the resulting index For performance, our indexer uses grid-based sampling, which has worse visual quality than for example blue-noise sampling as it is supported by *Schwarzwald* and *PotreeConverter*. Additionally, our stream-based indexer cannot write all nodes into a single file, as *PotreeConverter* version 2 does, since it is not known how much space to reserve for each node upfront. In contrast to *Schwarzwald*, for which this was a limitation due to the chosen algorithm, for real-time indexing this is fundamentally impossible as any node might receive new points at any time. The only way to guarantee that nodes are truly finished is to include sensor system trajectories, if they are known upfront. An example would be terrestrial LiDAR scanning of a city with a known route. Combined with the maximum scanner range, it would be possible to calculate in real-time which octree nodes will never receive new points.

Maximum throughput Our test results indicate maximum point insertion rates of about 1.8 million points per second, which is enough for the sensor system that we tested with, which produces at most 300.000 points per second. More powerful sensor systems do exist which can exceed the maximum throughput. Based on the scalability values shown in Figure 5.6b, it is debatable whether adding more CPU cores would bring the necessary performance improvement to manage the most powerful LiDAR scanners. Tests with faster hardware would be required to confirm this. There exist GPU-based point cloud indexing algorithms [141] which have substantially higher throughput, but they are currently limited to in-core processing and further research is needed to integrate GPU-acceleration into stream-based point cloud indexing.

While the real-time localization requirement currently seems to be a fundamental blocker for the application of real-time indexing, many of the other limitations are engineering problems which should be solvable with further research and development.

Similar to *ad-hoc queries*, we believe that real-time indexing is a technique that can compliment other point cloud data management solutions, instead of completely replacing them. We discuss the implications of this for point cloud data management as a whole in Chapter 6.

5.5 Conclusion

In this chapter, we introduced a stream-based point cloud indexing system, which is the first system that is able to index point clouds in real-time during capturing with a LiDAR scanner. The stream-based indexer is based on task-parallel programming, similar to the *Schwarzwald* system introduced in the previous chapter, but was developed specifically to work in real-time without knowing the full point cloud upfront. We proposed several different task priority functions through which the indexer selects which points are to be processed next. The resulting index is based on a multi-root MNO structure which can deal with arbitrarily large point cloud bounds. We evaluated a reference implementation of the stream-based indexer, both in terms of theoretical capabilities such as point insertion rates and point latency, as well as in a real-world test on a physical sensor setup. Based on the test hardware, we achieve point insertion rates of up to 1.8 million points per second, which is sufficient for many but not all LiDAR scanners currently available on the market. Additionally we demonstrated query latencies as low as 0.1 seconds from the time a point is captured by the scanner to the time it appears in a query response. This makes real-time indexing by far the most responsive point cloud data management approach available today.

We also pointed out several limitations of our real-time indexing system, such as its dependence on real-time point cloud localization, which we solve by using SLAM. Since many terrestrial and airborne LiDAR systems perform localization as a postprocess, we discussed potential solutions and further research opportunities for making real-time point cloud indexing more usable.

All things considered, we were able to answer the research question stated at the beginning of the chapter and demonstrate a significant improvement in point cloud query responsiveness through real-time indexing.

6 Conclusion

> “Expectations were like fine pottery. The harder you held them, the more likely they were to crack.”
>
> Brandon Sanderson, The Way of Kings

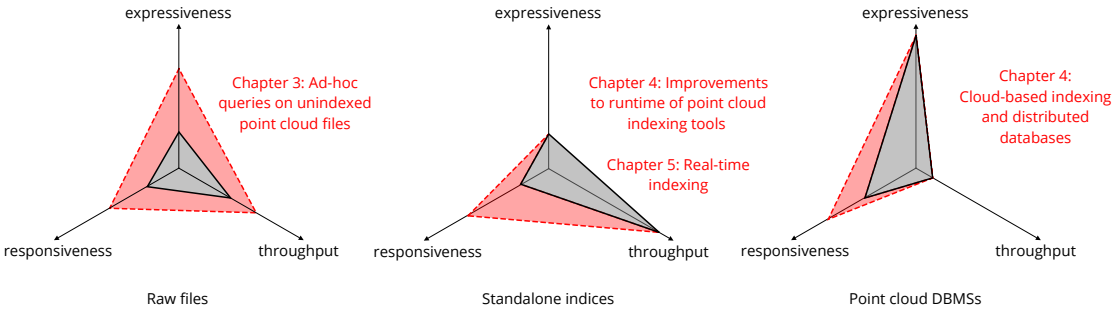


Figure 6.1: Our contributions to the state of the art of point cloud data management

In this chapter, we summarize the work presented in this thesis and critically discuss the results from Chapters 3 to 5. First, we discuss the implications for the research questions and the main research hypothesis in Section 6.1. We then give an outlook on research perspectives in Section 6.2. Lastly in Section 6.3 we discuss the potential impacts of our work on the state of the art of point cloud data management using the *Fibre3D* project as a showcase for practical applications.

6.1 Implications for the research hypothesis

With this thesis, we set out to confirm or reject the following research hypothesis:

> **Hypothesis**
>
> *Using adaptive indexing, parallel programming, as well as columnar memory layouts improves the query throughput, responsiveness, and expressiveness of existing point cloud data management approaches.*

To this end, we presented improvements to various fields related to point cloud storage and indexing, shown schematically in Figure 6.1. With these results we are able to answer the research questions stated in Section 1.3. We now explain our answers to each research question in detail, followed by an explanation what this means for the research hypothesis.

RQ1: Can ad-hoc queries enable applications to work directly with raw point cloud files instead of sophisticated index structures? The results shown in Chapter 3 show that raw point cloud files do have more potential as a data management solution than it was previously assumed. Our proposed *ad-hoc queries* improves data-to-insight time compared to traditional batch-based indexing, while also outperforming existing point cloud DBMSs in terms of data throughput and oftentimes even response times. Moving from interleaved to columnar memory layouts significantly improves query responsiveness for single-attribute queries, as demonstrated by our proposed *LAST* file format. Preliminary results with adaptive indexing indicate that this is a promising technique to mitigate some of the shortcomings of *ad-hoc queries* for larger datasets, as adaptive indexing increases query responsiveness. The major downsides of *ad-hoc queries* are that they are computationally expensive and have hard limits regarding data size, query complexity, and the number of concurrent users.

Summary: Our work introduces *ad-hoc queries*, which extend the power of raw files as a point cloud data management solution beyond what was previously considered feasible, while also quantifying their limits. When working within these limits, we show that *ad-hoc queries* are a viable technique for interactive exploration of point cloud data, in the same way that systems like *NoDB* [4] are for textual data.

RQ2: How can the runtime of current point cloud indexing tools be improved? In Chapter 4 we introduced a new algorithm for batch-based point cloud indexing based

on partitioning points using Morton indices. Point cloud indexing is a resource-intensive process due to the calculation of LOD, but it was unclear what the main reason for this is and to which extent I/O and compute performance play a role. We demonstrated that the choice of sampling function does have an impact on the indexing performance, but this impact depends on the choice of the indexing algorithm itself. Our novel *Schwarzwald* system performs well on SSDs as well as on slower HDDs, with relatively consistent performance regardless of the choice of sampling function. In contrast, the existing *PotreeConverter* 2.0 system is more dependent on I/O performance, with significantly slower runtimes on HDDs than on SSDs. Both *Schwarzwald* and *PotreeConverter* 2.0 achieve high degrees of parallelism through intelligent data partitioning, which shows that parallel programming done right does speed up the point cloud indexing processing significantly. Beyond that, we also demonstrated the benefit of horizontal scalability for point cloud indexing, through our Cloud-optimized indexer, which outperforms all other indexing tools by up to a factor of 3 in our tests. It uses a similar partitioning scheme than the Desktop version of *Schwarzwald* and hence confirms the effectiveness of this approach for improving the runtime of point cloud indexers.

Summary: Both systems that we developed improved the runtime of point cloud indexing compared to the previous state of the art. This indicates that parallel processing using partitioning based on Morton indices is an effective way to speed up point cloud indexing.

RQ3: Can point clouds be indexed in real-time during the capturing process with a LiDAR scanner? When we talk about query responsiveness, we always include preprocessing times and hence use responsiveness as a synonym to data-to-insight time. In Chapter 5 we demonstrated an approach that is able to index point cloud data in real-time during capturing with a LiDAR scanner, thus getting rid of all preprocessing. Our reference system, which uses stream-based processing, allowed simultaneous indexing and querying with an average query responsiveness of 0.1 seconds. We confirmed these numbers in a series of synthetic tests as well as tests on a real-world sensor system. We thus achieve a query responsiveness that is about two orders of magnitude faster than for *ad-hoc queries*, and three to five orders of magnitude faster than when using batch-based indexing. From a theoretical point of view, this is close to the limit for query responsiveness of zero seconds, disregarding network latency which in practice can be significant. Achieving these responsiveness values is only possible for systems that have real-time localization capabilities, which is a limiting factor. The quality of the resulting point cloud is also lower when using real-time localization compared to localization through postpro-

cessing based on high-precision trajectories collected from various sensors.

Summary: As the first system of its kind, we demonstrated both the feasibility of real-time point cloud indexing as well as the substantial improvement for data-to-insight time that this indexing approach enables.

Confirming the research hypothesis The research hypothesis of this thesis stated that each of the three approaches *adaptive indexing*, *parallel programming*, and *columnar memory layouts* can be used to improve the throughput, responsiveness, and expressiveness of queries on point cloud data. We now show why this hypothesis holds.

Regarding adaptive indexing, we showed that it can improve the responsiveness of *ad-hoc queries* with some restrictions concerning the query type. Overall we also demonstrated that *ad-hoc queries* allow a wide range of queries to be answered on raw files, thus increasing the expressiveness of raw files as a point cloud data management solution. A different approach that can be labeled as adaptive indexing was shown in Chapter 5 with our stream-based progressive indexer that is capable of indexing point clouds in real-time. Here we demonstrated significant improvements to the query responsiveness compared to traditional batch-based indexing.

Concerning the usage of parallel programming, all software systems that we developed as part of this thesis use some form of parallelization. We specifically demonstrated the positive impact of parallel programming on query responsiveness in Chapters 4 and 5. Task-based parallel programming as well as the distributed Map-Reduce paradigm allowed us to develop point cloud indexing systems with significantly reduced runtimes compared to the state of the art. For *ad-hoc queries*, parallel processing also plays a significant role in increasing throughput and responsiveness.

Lastly we evaluated the usage of columnar memory layouts for point cloud storage. Here we showed significant effects for improving both query responsiveness and throughput when working with raw data. Additionally the in-memory storage format for our *Schwarzwald* indexer also uses a columnar memory layout which contributes to the decrease in runtime.

Since we demonstrated positive effects on query throughput, responsiveness, and expressiveness for each of the three approaches we thus conclude that the research hypothesis holds.

6.2 Research perspectives

While we demonstrated several improvements to various point cloud data management approaches in this thesis, new questions arose that we believe are worth pursuing in the future.

Combining horizontal and vertical scalability of point cloud indexing Our work on faster batch-based indexing was focused mainly on *out-of-core* processing. GPU-based point cloud indexing has recently become an active area of research [141, 134], demonstrating significant performance improvements due to the large parallelization of GPUs. These approaches are currently limited to in-core processing but have excellent vertical scalability. Adding *out-of-core* capabilities would make these approaches usable for large-scale point clouds, which take days to index with the fastest current indexers. Another area of research is the unification of horizontal and vertical scalability through the development of an indexer that can adapt to the available resources, using GPU-acceleration if available while scaling onto multiple VMs in the Cloud.

True progressive indexing We demonstrated a proof-of-concept for adaptive indexing based on incoming queries in Section 3.3.6. Further research is needed to make true progressive indexing possible, which adapts not only to the incoming queries but also uses available resources intelligently and generates a high-quality index structure with LOD support. Progressive indexing during rendering using the GPU has recently been demonstrated [134], but as all GPU-based indexing methods it is currently restricted to in-core processing and not capable of indexing other attributes besides positions. While GPU-based indexers are fast, all current approaches lack the adaptive component that makes it possible to index only those subsections of the data that are actively queried.

Improve real-time indexing In Chapter 5 we introduced the novel concept of real-time point cloud indexing. Our proof-of-concept system could be improved in various ways, for example by supporting adjustments to the generated index with more accurately located points computed in a postprocess after capturing. As discussed, this would currently required a full rebuild of the index. GPU-support for the indexer would also be beneficial to keep up with scanners with higher measurement rates. Lastly some engineering effort is required to integrate our solution with more sophisticated mobile-mapping systems.

A unified data access layer A central problem of current point cloud data management solutions is the way they provide access to the data. The *PDAL* library and our own *pas-*

ture library are attempts to provide building blocks for unified point cloud data management for applications, but both are specific tools instead of general solutions. Point cloud databases are a potential solution, either through existing systems like those studied in the literature [150, 32] or concepts like the *ad-hoc queries* we introduced, but neither are currently suited for high-quality point cloud visualization. Web-based applications such as *Potree* and *CesiumJS* depend on specific metadata for client-side frustum culling which a point cloud DBMS typically does not provide. Highly interactive renderers like those studied by Schütz et al. require optimized data formats accessed directly through the file system and thus lack generality [142, 135]. We believe that the flexibility that accessing point clouds through a structured query language offers is the most promising approach for unified point cloud data access, but it will require research, engineering and standardization effort. In terms of research, better support for LOD and higher query throughput are challenges for the existing point cloud DBMSs. Adding support for query-based data access to web-based renderers requires engineering effort, which can be justified if for example the OGC proposes a standardized point cloud access Application Programming Interface (API), similar to those for raster data and vector features (Web Map Service (WMS) and Web Feature Service (WFS)).

Improvements to existing file formats We are currently seeing the development of new point cloud file formats aimed at web- and Cloud-based processing and access [31, 60]. We believe that taking into account our findings regarding columnar memory layouts could help speed up data access. A remaining challenge is the overhead of compression for point cloud file formats. *LAZ* is widely used as a storage format due to its excellent compression ratios, but the poor decoding performance makes it unsuitable for many interactive applications. A faster compression scheme with similar compression ratios to *LAZ* could help bridge this gap.

6.3 Demonstrating the impact of our work with the *Fibre3D* project

In Chapter 1 we introduced *Fibre3D*, a system developed by us and used by Deutsche Telekom AG to facilitate planning of fiber network connections. *Fibre3D* uses point cloud data to enable interactive placement of objects with centimeter precision within a panoramic image, and has been used for the planning of fiber networks in over 1000 areas throughout Germany [78]. To enable this, data is continuously captured using cars equipped with terrestrial LiDAR scanners and spherical imaging systems. As of late 2023,

about 166 TiB of point cloud data have been captured, with a total of about 6.75 trillion points. The size of the point clouds used by *Fibre3D* makes this project an ideal candidate to demonstrate the potential impact of the techniques introduced in this thesis. Some techniques, such as our task-parallel indexing algorithm implemented in the *Schwarzwald* system, were developed specifically for this project and have seen usage in a production environment. For the other techniques, we will discuss potential improvements as well as identify research gaps that currently prevent their application.

We identified the following challenges during the development and operation of *Fibre3D*:

Cost Minimizing the cost for data preprocessing and storage

Migrations Migrating between different platforms, for example different rendering engines

Debugging Identifying and fixing problems with data deliveries as well as rendering algorithms

Challenge	Technique	Potential benefit	Limitations
Cost	Task-based parallel programming (Section 4.3)	Decreases runtime of point cloud indexing and hence cost for compute infrastructure	Large number of files is challenging for storage and copying
	Real-time indexing (Chapter 5)	Prevent most preprocessing in the Cloud	Integration into existing Mobile Mapping System (MMS)s required
	Adaptive indexing (Section 3.3.6)	Index only data that is requested by users	Our tested approach does not support LOD
			Other approaches [142, 134] currently are limited to in-core datasets
Migration	(distributed) point cloud database (Section 4.4)	Provides unified data access through a common query language	Missing protocol for unified point cloud data access
			Web-based renderers do not support point cloud databases
Debugging	*Ad-hoc queries* (Chapter 3)	Interactive exploration of data simplifies debugging	Only feasible with datasets up to about one billion points

Table 6.1: Potential impact that the various techniques introduced in this thesis could have for three main challenges encountered during the development and operation of the *Fibre3D* project.

Table 6.1 briefly summarizes the potential impacts of the techniques introduced in this

thesis for these three challenges. We discuss these impacts in detail in the following sections.

6.3.1 Reducing cost through more efficient indexing

From an operational perspective, cost is one of the primary factors that businesses aim to minimize. For data-intensive applications such as *Fibre3D*, the computational resources required for preprocessing and providing the data to end users can be substantial. Specific factors that influence the cost for compute infrastructure are the runtime of processing tools, which in the Cloud translate to Virtual Machine (VM) uptimes, the required capabilities of these VMs, and the amount of required storage. The *Schwarzwald* system introduced in Section 4.3 was developed specifically to reduce the runtime of the point cloud indexing process in the context of *Fibre3D*. Based on the runtime improvements over the existing indexers as shown in Section 4.5.2, *Schwarzwald* has about 2.4 times higher throughput than *Entwine* and 6.5 times higher throughput than version 1.7 of *PotreeConverter*. Since all tools run on a single machine, total VM runtimes for point cloud indexing are between 58% (*Entwine*) to 85% (*PotreeConverter* v1.7) lower. Table 6.2 shows an estimate for the runtime for indexing all 6.5T points in the *Fibre3D* dataset.

Tool	Average point throughput [*MPts/s*] Based on values from Section 4.5.2	Estimated runtime
PotreeConverter v1.7	0.27	289 days
Entwine	0.74	106 days
Schwarzwald	1.75	45 days
PotreeConverter v2	1.79	44 days

Table 6.2: Estimated runtime for indexing all 6.75 trillion points in the *Fibre3D* dataset

Since Cloud providers typically bill VM uptime instead of CPU time, making efficient usage of the hardware resources of a single VM is an effective way to reduce the cost of the compute infrastructure. It is worth noting that the same does not apply to algorithms that employ horizontal scalability, such as the Cloud-based indexer introduced in Section 4.4. While it did achieve a speedup of about 3x over the single-process variant of *Schwarzwald* in our experiments, it did so by running on four VMs instead of a single one, resulting in an overall increase in cost of 33%.

To reduce cost for preprocessing in the Cloud even more, real-time indexing as introduced in Chapter 5 could be used. If all data is indexed using on-premise hardware integrated into the MMS, it completely removes the need for processing in the Cloud, while also resulting in a dataset that is structured in a way that makes it immediately usable by client applications. The only cost would be due to uploading the data from the on-premise hardware into the Cloud-environment, which is negligible compared to the cost for indexing itself. Due to the limitations of real-time indexing, additional engineering effort is needed to integrate this indexing scheme within existing capturing systems.

Another problem of preprocessing approaches is that they process data that might never be used at all. For *Fibre3D* it is not unreasonable to assume that the vast majority of the point cloud is never requested by any user due to the usage of the application. It is mainly used for verifying locations for fiber distribution cabinets. For FTTH, there will be one distribution cabinet every few hundred meters and planners will position these cabinets within a radius of a few tens of meters around predefined positions. So there is a small sphere around these positions for which users require high-precision data, but large regions that have a low chance of ever being requested by users. Adaptive indexing methods could help reduce cost further by only indexing the areas of the point cloud that are actually requested by users.

6.3.2 Dealing with technology and platform migrations through a unified point cloud access layer

A frequent reality of the lifecycle of any software are migrations, be it versions of libraries, changes in used technologies, or platform migrations. We encountered a similar situation with *Fibre3D*, where the initial version which was built on top of *CesiumJS*. Once the software was used daily by many users, performance problems became apparent, both due to the point cloud rendering support of *CesiumJS* as well as due to the large number of individual files that the indexed point cloud in the *3D Tiles* format was made up of. We hence chose to migrate to *Potree*, which does not support the *3D Tiles* file format. This resulted in a costly migration process in which all point cloud data had to be converted from the *3D Tiles* format into a format supported by *Potree*. This could have been prevented with a unified data access layer through which the client applications such as *CesiumJS* or *Potree* access the data. The *ad-hoc queries* approach introduced in Chapter 3 is one possible approach for unified point cloud data access, though in the context of *Fibre3D* a point cloud database as studied by ourselves [76] and others [150, 32] would be the more scalable approach. We were unable to use such a system, as neither *CesiumJS* nor *Potree* support query-based point cloud access. We believe this to be an interesting

perspective for future research and engineering efforts, as a unified point cloud data access layer would prevent the dependence of point cloud applications on specialized data formats that are not interoperable. The search for a standardized, universal point cloud data access API is also an ongoing effort of the OGC [28].

6.3.3 Faster debugging by using *ad-hoc queries* for interactive data exploration

The development of *Fibre3D* required substantial development resources. A challenging aspect is the development and testing of the rendering code. Test datasets have to be generated, as the actual data is too large to be usable for tests on developer machines. Currently, the data resides as large archives on an object storage within the Cloud, which have to be manually downloaded and decompressed before relevant subsections of the data can be extracted. A unified data access API as provided by *ad-hoc queries* would help developers to quickly select relevant subsets of the data and perform tests without prior indexing. Alternatively, storing the data inside a distributed database would achieve the same result, while giving end users concurrent access to the data.

We also noticed that being able to quickly query datasets during development can help speed up debugging, as there are many situations where quickly looking at the data is more efficient than trying to write an automated test. An example for this is the alignment between the point cloud and the panoramic images in the *Fibre3D* client. Rendering a subset of the point cloud on top of the panoramic image quickly shows whether the two datasets are correctly aligned, but writing an automated test for this is challenging. Since the common web-based point cloud renderers *Potree* and *CesiumJS* both require specialized tiled data formats, preprocessing the raw point cloud data is a must even if only for a quick manual test. If these tools could instead use *ad-hoc queries*, no preprocessing would be required and these manual tests could be done using just the raw data. During testing the performance limitations of *ad-hoc queries* are also less relevant: Waiting ten seconds until a view frustum query with LOD has finished is still orders of magnitude faster than indexing the whole point cloud.

Quality assurance is another aspect that is challenging. While this is often a contractual obligation of data providers, it is still desirable to detect problems with the data as early as possible. We encountered situations where faulty data was processed and uploaded to the production environment, where it was noticed by the end users. Ideally, an automated process would detect such problems as early as possible, but real-time indexing and monitoring during capturing could help to detect problems manually. Adaptive indexing would also help to mitigate costs in such a situation as it prevents running a costly indexing process for a large but faulty dataset in the first place.

6.4 Closing remarks

In this thesis we explored the domain of point cloud data management. We developed algorithms and tools that improve the way that existing data management solutions deliver data to applications. Query responsiveness is the main factor that our work focused on as it is crucial that applications get fast access to point cloud datasets, even if their size goes into the billions of points. New parallelized indexing algorithms developed by us, both for batch-processing but also for real-time processing, improve query responsiveness by reducing the amount of preprocessing. Data layout optimizations, in particular columnar memory layouts, play an important role in increasing the query throughput, which ultimately also affects responsiveness. Adaptive indexing is a novel and promising approach for creating the necessary index structures for managing large data, complex queries and multiple concurrent users. In situations where we do not deal with these extremes, we showed that *ad-hoc queries* can be a viable alternative for working with raw files, which still are one of the primary point cloud data management solutions.

While we still have some way to go until a unified point cloud data management solution becomes feasible, our work brings the main approaches raw files, standalone indices, and databases closer together so that future research will certainly be able to bridge that gap.

Bibliography

[1] *Actueel Hoogtebestand Nederland (v3)*. Dataset. Accessed: 2023-12-14. URL: https://www.ahn.nl/.

[2] *Actueel Hoogtebestand Nederland (v4)*. Dataset. Accessed: 2023-12-14. URL: https://www.ahn.nl/.

[3] Aerial Surveys. *Wellington, New Zealand 2013*. Dataset. Accessed: 2023-12-14. 2013–2014. DOI: https://doi.org/10.5069/G9CV4FPT. URL: https://portal.opentopography.org/datasetMetadata.jsp?otCollectionID=OT.042017.2193.2.

[4] Ioannis Alagiannis et al. "NoDB: efficient query execution on raw data files". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, pp. 241–252. DOI: 10.1145/2830508.

[5] Jyrki Alakuijala and Zoltan Szabadka. *Brotli compressed data format*. Tech. rep. IETF, 2016. DOI: 10.17487/RFC7932.

[6] C. Alis, J. Boehm, and K. Liu. "Parallel processing of big point clouds using Z-Order-based partitioning". In: *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences-ISPRS Archives*. Vol. 41. International Society of Photogrammetry and Remote Sensing (ISPRS). 2016, pp. 71–77. DOI: 10.5194/isprs-archives-XLI-B2-71-2016.

[7] American Society for Photogrammetry and Remote Sensing (ASPRS). *LAS SPECIFICATION, VERSION 1.4 - R13*. Accessed: 2022-09-22. 2013. URL: https://www.asprs.org/wp-content/uploads/2010/12/LAS_1_4_r13.pdf.

[8] Jorge Aparicio and Brook Heisler. *gadomski/las-rs: Read and write ASPRS las files, Rust edition*. Accessed: 2023-12-14. 2015. URL: https://github.com/gadomski/las-rs.

[9] John Backus. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". In: *ACM Turing Award Lectures*. New York, NY, USA: Association for Computing Machinery, 2007, p. 1977. ISBN: 9781450310499. DOI: 10.1145/1283920.1283933.

[10] Louis Bavoil and Miguel Sainz. *Screen Space Ambient Occlusion*. Tech. rep. NVIDIA, 2008. URL: https://developer.download.nvidia.com/SDK/10.5/direct3d/Source/ScreenSpaceAO/doc/ScreenSpaceAO.pdf.

[11] Juan A. Béjar-Martos et al. "Strategies for the Storage of Large LiDAR Datasets—A Performance Comparison". In: *Remote Sensing* 14.11 (2022). ISSN: 20724292. DOI: 10.3390/rs14112623.

[12] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". In: *Commun. ACM* 18.9 (1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007.

[13] Nikos Bikakis et al. "RawVis: Visual Exploration over Raw Data". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11019 LNCS (2018), pp. 50–65. ISSN: 16113349. DOI: 10.1007/978-3-319-98398-1_4.

[14] Filip Biljecki et al. "Applications of 3D City Models: State of the Art Review". In: *ISPRS International Journal of Geo-Information* 4.4 (2015), pp. 2842–2889. DOI: 10.3390/ijgi4042842.

[15] Pascal Bormann and Contributors. *Mortano/pasture: Rust library for point cloud processing*. Accessed: 2023-12-14. 2021. URL: https://github.com/Mortano/pasture.

[16] Pascal Bormann and Michel Krämer. "A System for Fast and Scalable Point Cloud Indexing Using Task Parallelism". In: *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. Ed. by Silvia Biasotti, Ruggero Pintus, and Stefano Berretti. The Eurographics Association, 2020. ISBN: 978-3-03868-124-3. DOI: 10.2312/stag.20201250.

[17] Pascal Bormann., Michel Krämer., and Hendrik Würz. "Working Efficiently with Large Geodata Files using Ad-hoc Queries". In: *Proceedings of the 11th International Conference on Data Science, Technology and Applications - DATA,* INSTICC. SciTePress, 2022, pp. 438–445. DOI: 10.5220/0011291200003269.

[18] Pascal Bormann et al. "Executing ad-hoc queries on large geospatial data sets without acceleration structures". In: *Springer Nature Computer Science* 5.647 (2024). DOI: 10.1007/s42979-024-02986-z.

[19] Pascal Bormann et al. “Real-time Indexing of Point Cloud Data During LiDAR Capture”. In: *Computer Graphics and Visual Computing (CGVC)*. Ed. by Peter Vangorp and Martin J. Turner. The Eurographics Association, 2022. ISBN: 978-3-03868-188-5. DOI: 10.2312/cgvc.20221173.

[20] Christian Boucheny. “Visualisation scientifique de grands volumes de données : Pour une approche perceptive.” PhD thesis. Université Joseph Fourier, 2009. URL: https://hal.science/tel-00438464.

[21] Mathieu Brédif, Kun Liu, and Pengfei Xuan. *Spark SQL IQmulus Library*. Accessed: 2023-12-14. 2023. URL: https://github.com/IGNF/spark-iqmulus.

[22] M.P. Bunds et al. *High Resolution Topography of House Range Fault, Utah*. Dataset. Accessed: 2023-12-14. 2019. DOI: https://doi.org/10.5069/G9348HH6. URL: https://portal.opentopography.org/datasetMetadata?otCollectionID=OT.102019.6341.1.

[23] M.P. Bunds et al. *High Resolution Topography of the Southern San Andreas Fault from Painted Canyon to Bombay Beach, USA*. Dataset. Accessed: 2023-12-14. DOI: https://doi.org/10.5069/G94M92RG. URL: https://portal.opentopography.org/datasetMetadata?otCollectionID=OT.092021.32611.1.

[24] Paul Burke. *ptx plain text data format*. 2019. URL: http://paulbourke.net/dataformats/ptx/.

[25] Inc. Cesium GS. *CesiumJS - Geospatial 3D Mapping and Virtual Globe Platform*. Accessed: 2023-12-14. 2023. URL: https://cesium.com/platform/cesiumjs/.

[26] Cesium GS, Inc. *Cesium ion - Cesium*. Accessed: 2023-12-14. 2023. URL: https://cesium.com/platform/cesium-ion/.

[27] Y. Collet and M. Kucherawy. *Zstandard Compression and the application/zstd Media Type*. RFC 8478. RFC Editor, 2018.

[28] Open Geospatial Consortium. *OGC Testbed-14: Point Cloud Data Handling Engineering Report*. Tech. rep. Open Geospatial Consortium, 2019. URL: https://docs.ogc.org/per/18-048r1.html.

[29] PDAL Contributors. *PDAL Point Data Abstraction Library*. Aug. 2022. DOI: 10.5281/zenodo.2616780. URL: https://doi.org/10.5281/zenodo.2616780.

[30] Thomas H. Cormen et al. *Introduction to Algorithms*. Cambridge, MA, USA: The MIT Press, 2009. ISBN: 9780262033848.

[31] Patrick Cozzi, Sean Lilley, and Gabby Getz. *3D Tiles Specification 1.0*. Tech. rep. Cesium, Open Geospatial Consortium, 2019.

[32] Rémi Cura, Julien Perret, and Nicolas Paparoditis. "A scalable and multi-purpose point cloud server (PCS) for easier and faster point cloud data management and processing". In: *ISPRS Journal of Photogrammetry and Remote Sensing* 127 (2017), pp. 39–56. DOI: 10.1016/j.isprsjprs.2016.06.012.

[33] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. "Sequential point trees". In: *ACM Transactions on Graphics (TOG)* 22.3 (2003), pp. 657–662. DOI: 10.1145/882262.882321.

[34] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113. DOI: 10.1145/1327452.1327492.

[35] David Deibe, Margarita Amor, and Ramon Doallo. "Big data storage technologies: a case study for web-based LiDAR visualization". In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 3831–3840. DOI: 10.1109/BigData.2018.8622589.

[36] L. Peter Deutsch and Jean-Loup Gailly. *ZLIB Compressed Data Format Specification version 3.3*. RFC 1950. RFC Editor, 1996. URL: http://www.rfc-editor.org/rfc/rfc1950.txt.

[37] Sören Discher et al. "A point-based and image-based multi-pass rendering technique for visualizing massive 3D point clouds in VR environments". In: *Journal of WSCG* 26.2 (2018), pp. 76–84. ISSN: 12136964. DOI: 10.24132/JWSCG.2018.26.2.2.

[38] *Dual Channel Waveform Processing Airborne LiDAR Scanning System for High Point Density Mapping and Ultra-Wide Area Mapping*. VQ-1560 II-S. Riegl. 2023. URL: http://www.riegl.com/uploads/tx_pxpriegldownloads/RIEGL_VQ-1560II-S_Datasheet_2023-09-20.pdf.

[39] Sami El-Mahgary, Juho Pekka Virtanen, and Hannu Hyyppä. "A simple semantic-based data storage layout for querying point clouds". In: *ISPRS International Journal of Geo-Information* 9.2 (2020). ISSN: 22209964. DOI: 10.3390/ijgi9020072.

[40] Jan Elseberg, Dorit Borrmann, and Andreas Nüchter. "One billion points in the cloud - An octree for efficient processing of 3D laser scans". In: *ISPRS Journal of Photogrammetry and Remote Sensing* 76 (2013), pp. 76–88. ISSN: 09242716. DOI: 10.1016/j.isprsjprs.2012.10.004.

[41] The Apache Software Foundation. *Apache Cassandra*. Accessed: 2023-11-08. URL: https://cassandra.apache.org/.

[42] The Apache Software Foundation. *Apache Hadoop*. Accessed: 2023-11-08. URL: https://hadoop.apache.org/.

[43] Fraunhofer IGD. *Ad-hoc queries on point clouds - Benchmark implementation*. Accessed: 2023-12-14. 2021. URL: https://github.com/igd-geo/adhoc-queries-pointclouds.

[44] Fraunhofer IGD. *lidarserv - Implementation of LiDAR Server, Point Source, and Viewer*. Accessed: 2022-05-25. 2022. URL: https://github.com/igd-geo/lidarserv.

[45] Fraunhofer IGD. *Progressive Point Cloud Indexing with Big Data Technologies*. Accessed: 2023-12-14. 2021. URL: https://github.com/igd-geo/cloud-based-pointcloud-indexing.

[46] Fraunhofer IGD. *Schwarzwald - A fast point cloud tiling tool*. Accessed: 2023-12-14. 2021. URL: https://github.com/igd-geo/schwarzwald.

[47] Jorge Fuentes-Pacheco, José Ruiz-Ascencio, and Juan Manuel Rendón-Mancha. "Visual simultaneous localization and mapping: a survey". In: *Artificial intelligence review* 43 (2015), pp. 55–81. DOI: 10.1007/s10462-012-9365-8.

[48] Pete Gadomski and Contributors. *Criterion.rs - Statistics-driven Microbenchmarking in Rust*. Accessed: 2023-12-14. 2023. URL: https://github.com/bheisler/criterion.rs.

[49] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. 2nd. Upper Saddle River, NJ: Pearson, 2008. ISBN: 0-13-187325-3.

[50] D. Girardeau-Montaut et al. "Change detection on points cloud data acquired with a ground laser scanner". In: *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives* 36 (2005). ISSN: 16821750.

[51] Daniel Girardeau-Montaut. *CloudCompare*. 2016. URL: https://www.cloudcompare.org/.

[52] rapidlasso GmbH. *LAStools: converting, filtering, viewing, processing, and compressing LiDAR data in LAS and LAZ format*. Accessed: 2023-12-14. 2007-2022. URL: https://lastools.github.io/.

[53] Enrico Gobbetti and Fabio Marton. “Layered Point Clouds ”. In: *SPBG’04 Symposium on Point - Based Graphics 2004*. Ed. by Markus Gross et al. The Eurographics Association, 2004. ISBN: 3-905673-09-6. DOI: /10.2312/SPBG/SPBG04/113-120.

[54] Google. *Google Maps*. Accessed: 2023-12-14. URL: https://www.google.com/maps.

[55] Prashant Goswami et al. “High Quality Interactive Rendering of Massive Point Models Using Multi-way kd-Trees”. In: *18th Pacific Conference on Computer Graphics and Applications*. IEEE. 2010, pp. 93–100. DOI: 10.1109/PacificGraphics.2010.20.

[56] Brendan Gregg. *Flame Graphs*. Accessed: 2023-12-14. 2011. URL: https://www.brendangregg.com/flamegraphs.html.

[57] Antonin Guttman. “R-trees: A dynamic index structure for spatial searching”. In: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 1984, pp. 47–57. DOI: 10.1145/971697.602266.

[58] Jeff Hecht. “Lidar for self-driving cars”. In: *Optics and Photonics News* 29.1 (2018), pp. 26–33. DOI: 10.1364/OPN.29.1.000026.

[59] Pieter Hijma et al. “Optimization techniques for GPU programming”. In: *ACM Computing Surveys* 55.11 (2023), pp. 1–81. DOI: 10.1145/3570638.

[60] Hobu, Inc. *Cloud Optimized Point Cloud Specification - 1.0*. Tech. rep. Hobu, Inc., 2021. URL: https://copc.io/.

[61] Hobu, Inc. *Entwine*. Accessed: 2023-12-14. 2023. URL: https://entwine.io/.

[62] Hobu, Inc. *Entwine Point Tile*. Accessed: 2023-12-14. 2023. URL: https://entwine.io/en/latest/entwine-point-tile.html.

[63] Pedro Holanda et al. “Cracking KD-Tree: The First Multidimensional Adaptive Indexing (Position Paper)”. In: *DATA 2018 - Proceedings of the 7th International Conference on Data Science, Technology and Applications* (2018), pp. 393–399. DOI: 10.5220/0006944203930399.

[64] Pedro Holanda et al. “Progressive indexes: Indexing for interactive data analysis”. In: *Proceedings of the VLDB Endowment* 12.13 (2020), pp. 2366–2378. ISSN: 21508097. DOI: 10.14778/3358701.3358705. URL: \url{https://doi.org/10.14778/3358701.3358705}.

[65] Tsung-Wei Huang et al. “Cpp-taskflow: Fast task-based parallel programming using modern C++”. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, pp. 974–983. DOI: 10.1109/IPDPS.2019.00105.

[66] Daniel Huber. “The ASTM E57 File Format for 3D Imaging Data Exchange”. In: *Proceedings of SPIE Electronics Imaging Science and Technology Conference (IS&T), 3D Imaging Metrology*. Vol. 7864. 2011. DOI: 10.1117/12.876555.

[67] Stratos Idreos. “Database Cracking: Towards Auto-tuning Database Kernels”. PhD Thesis. Harvard University, 2010. DOI: 11245/1.324880.

[68] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. “Database Cracking”. In: *Proceedings of the 3rd International Conference on Innovative Data Systems Research (CIDR)*. Asilomar, California, 2007, pp. 68–78. URL: https://stratos.seas.harvard.edu/sites/scholar.harvard.edu/files/IKM_CIDR07.pdf.

[69] Apple Inc. *Maps - Apple*. Accessed: 2023-12-14. URL: https://www.apple.com/maps/.

[70] Martin Isenburg. “LASzip: Lossless compression of lidar data”. In: *Photogrammetric Engineering and Remote Sensing* 79.2 (2013), pp. 209–217. ISSN: 00991112. DOI: 10.14358/PERS.79.2.209.

[71] Martin Isenburg et al. “Generating raster DEM from mass points via TIN streaming”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4197 LNCS (2006), pp. 186–198. ISSN: 16113349. DOI: 10.1007/11863939_13.

[72] J. C. (Cape town). *Juta’s Map of south Africa, from the Cape to the Zambie*. 1891. URL: https://commons.wikimedia.org/wiki/File:Juta%27s_Map_of_south_Africa,_from_the_Cape_to_the_Zambie.jpg.

[73] Manos Karpathiotakis et al. “Adaptive query processing on RAW data”. In: *Proceedings of the VLDB Endowment* 7.12 (2014), pp. 1119–1130. ISSN: 21508097. DOI: 10.14778/2732977.2732986.

[74] Tero Karras. “Maximizing parallelism in the construction of bvhs, octrees, and k-d trees”. In: *High-Performance Graphics 2012, HPG 2012 - ACM SIGGRAPH / Eurographics Symposium Proceedings* (2012), pp. 33–37. DOI: 10.2312/EGGH/HPG12/033-037.

[75] Andrew Kensler. *Correlated Multi-Jittered Sampling*. Tech. rep. Pixar Animation Studios, 2013. URL: https://graphics.pixar.com/library/MultiJitteredSampling/paper.pdf.

[76] Kevin Kocon and Pascal Bormann. "Point cloud indexing using Big Data technologies". In: *2021 IEEE International Conference on Big Data (Big Data)*. IEEE. 2021, pp. 109–119. DOI: 10.1109/BigData52589.2021.9671659.

[77] Michel Krämer. "A Microservice Architecture for the Processing of Large Geospatial Data in the Cloud". PhD thesis. TU Darmstadt, 2018. DOI: 10.2312/2632114.

[78] Michel Krämer et al. "A cloud-based data processing and visualization pipeline for the fibre roll-out in Germany". In: *Journal of Systems and Software* 211 (2024), p. 112008. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2024.112008. URL: https://www.sciencedirect.com/science/article/pii/S0164121224000517.

[79] Sampo Kuutti et al. "A survey of the state-of-the-art localization techniques and their potentials for autonomous vehicle applications". In: *IEEE Internet of Things Journal* 5.2 (2018), pp. 829–846. DOI: 10.1109/JIOT.2018.2812300.

[80] Mathieu Labbé and François Michaud. "RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation". In: *Journal of Field Robotics* 36.2 (2019), pp. 416–446. ISSN: 15564959. DOI: 10.1002/rob.21831.

[81] Susana Ladra et al. "Compact and indexed representation for LiDAR point clouds". In: *Geo-Spatial Information Science* 00.00 (2022), pp. 1–36. ISSN: 10095020. DOI: 10.1080/10095020.2022.2121664. URL: https://doi.org/10.1080/10095020.2022.2121664.

[82] C. Lauterbach et al. "Fast BVH construction on GPUs". In: *Computer Graphics Forum* 28.2 (2009), pp. 375–384. ISSN: 14678659. DOI: 10.1111/j.1467-8659.2009.01377.x.

[83] Eleazar Leal and Le Gruenwald. "Spatial Database Cracking with Quadtrees". 2020. URL: https://www.cs.ou.edu/~database/HIGEST-DB/publications/SIGSPATIAL20-Spatial%20Cracking%20-%20Submitted.pdf.

[84] Jae-Gil Lee and Minseo Kang. "Geospatial big data: challenges and opportunities". In: *Big Data Research* 2.2 (2015), pp. 74–81. DOI: 10.1016/j.bdr.2015.01.003.

[85] Wenkai Li et al. “A new method for segmenting individual trees from the lidar point cloud”. In: *Photogrammetric Engineering & Remote Sensing* 78.1 (2012), pp. 75–84. DOI: 10.14358/PERS.78.1.75.

[86] Haicheng Liu. “nD-PointCloud Data Management”. PhD thesis. Delft University of Technology, 2022. ISBN: 9789463665728. DOI: 10.7480/abe.2022.12.

[87] Haicheng Liu et al. “An efficient nD-point data structure for querying flood risk”. In: *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 43 (2021), pp. 367–374. DOI: 10.5194/isprs-archives-XLIII-B4-2021-367-2021.

[88] Haicheng Liu et al. “Executing convex polytope queries on nD point clouds”. In: *International Journal of Applied Earth Observation and Geoinformation* 105.October (2021), p. 102625. ISSN: 1872826X. DOI: 10.1016/j.jag.2021.102625. URL: https://doi.org/10.1016/j.jag.2021.102625.

[89] Haicheng Liu et al. “HistSFC: Optimization for nD massive spatial points querying”. In: *International Journal of Database Management Systems (IJDMS)* 12.3 (2020), pp. 7–28. DOI: 10.5121/ijdms.2020.12302.

[90] Xingcheng Lu et al. “A bottom-up approach to segment individual deciduous trees using leaf-off lidar point cloud data”. In: *ISPRS Journal of Photogrammetry and Remote sensing* 94 (2014), pp. 1–12. DOI: 10.1016/j.isprsjprs.2014.03.014.

[91] Mona Lütjens et al. “Virtual Reality in Cartography: Immersive 3D Visualization of the Arctic Clyde Inlet (Canada) Using Digital Elevation Models and Bathymetric Data”. In: *Multimodal Technologies and Interaction* 3.1 (2019). ISSN: 2414-4088. DOI: 10.3390/mti3010009. URL: https://www.mdpi.com/2414-4088/3/1/9.

[92] LZ4 Team. *LZ4 - Extremely fast compression*. Accessed: 2023-12-14. 2021. URL: https://github.com/lz4/lz4.

[93] Clément Mallet and Frédéric Bretar. “Full-waveform topographic lidar: State-of-the-art”. In: *ISPRS Journal of photogrammetry and remote sensing* 64.1 (2009), pp. 1–16. DOI: 10.1016/j.isprsjprs.2008.09.007.

[94] Stavros Maroulis et al. “Resource-aware adaptive indexing for in situ visual exploration and analytics”. In: *VLDB Journal* 32.1 (2023), pp. 199–227. ISSN: 0949877X. DOI: 10.1007/s00778-022-00739-z.

[95] Oscar Martinez Rubi et al. “A column-store meets the point clouds”. In: *FOSS4G-Europe Academic Track* (July 2014). DOI: 10.5281/zenodo.1045077.

[96] Oscar Martinez Rubi et al. “Taming the beast: Free and open-source massive point cloud web visualization”. In: *Capturing reality: The 3rd, laser scanning and LIDAR technologies forum*. Nov. 2015, pp. 23–25. DOI: 10.13140/RG.2.1.1731.4326/1.

[97] Merriam-Webster. *Geospatial Definition & Meaning*. https://www.merriam-webster.com/dictionary/geospatial. Accessed: 2023-12-14.

[98] Mike Acton. “Data-oriented design and C++”. Talk at cppcon 2014. 2014. URL: https://neil3d.gitee.io/assets/img/ecs/DOD-Cpp.pdf.

[99] Don P. Mitchell. “Generating Antialiased Images at Low Sampling Densities”. In: *SIGGRAPH Comput. Graph.* 21.4 (1987), pp. 65–72. ISSN: 0097-8930. DOI: 10.1145/37402.37410. URL: https://doi.org/10.1145/37402.37410.

[100] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. Tech. rep. International Business Machines Company New York, 1966.

[101] National Geospatial-Intelligence Agency (NGA). *Binary Point File 3 (BPF3), BPF Public License File Format Definition, Implementation Guide, 2015-August-5*. Standard. National Center for Geospatial Intelligence Standards (NCGIS), National Geospatial-Intelligence Agency (NGA), 2015.

[102] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1993. ISBN: 978-0-12-518406-9. DOI: https://doi.org/10.1016/C2009-0-21512-1.

[103] National Oceanic and Atmospheric Administration (NOAA) Coastal Services Center. *Lidar 101: An Introduction to Lidar Technology, Data, and Applications*. Tech. rep. National Oceanic and Atmospheric Administration (NOAA) Coastal Services Center, 2012. URL: https://coast.noaa.gov/data/digitalcoast/pdf/lidar-101.pdf.

[104] Carlos J. Ogayar-Anguita et al. “Nested spatial data structures for optimal indexing of LiDAR data”. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 195.February 2022 (2023), pp. 287–297. ISSN: 09242716. DOI: 10.1016/j.isprsjprs.2022.11.018. URL: https://doi.org/10.1016/j.isprsjprs.2022.11.018.

[105] Matthaios Olma et al. “Slalom: Coasting through raw data via adaptive partitioning and indexing”. In: *Proceedings of the VLDB Endowment* 10.10 (2017), pp. 1106–1117. ISSN: 21508097. DOI: 10.14778/3115404.3115415.

[106] Peter Oosterom et al. “Realistic Benchmarks for Point Cloud Data Management Systems”. In: Springer, Oct. 2017, pp. 1–30. ISBN: 978-3-319-25689-4. DOI: 10.1007/978-3-319-25691-7_1.

[107] OpenStreetMap contributors. *Cape Town, OpenStreetMap*. 2022. URL: https://www.openstreetmap.org/#map=12/-33.9350/18.4546.

[108] OpenStreetMap contributors. *OpenStreetMap*. Available as open data under the Open Data Commons Open Database License (ODbL). 2024. URL: http://openstreetmap.org/.

[109] OpenTopography Facility. *OpenTopography*. Accessed: 2022-03-21. 2022. URL: https://opentopography.org/.

[110] Johannes Otepka et al. “Georeferenced point clouds: A survey of features and point cloud management”. In: *ISPRS International Journal of Geo-Information* 2.4 (2013), pp. 1038–1065. DOI: 10.3390/ijgi2041038.

[111] Q. Ou et al. *Survey of 1920 Haiyuan Earthquake Rupture, China 2013-2017*. Dataset. Accessed: 2023-12-14. DOI: https://doi.org/10.5069/G9833Q6Z. URL: https://portal.opentopography.org/datasetMetadata?otCollectionID=OT.052022.32648.1.

[112] Pacific Gas and Electric Company. *PG&E Diablo Canyon Power Plant (DCPP): San Simeon, CA Central Coast*. Dataset. Accessed: 2023-12-14. 2013. DOI: https://doi.org/10.5069/G9CN71V5. URL: https://portal.opentopography.org/datasetMetadata?otCollectionID=OT.032013.26910.2.

[113] Vladimir Pajić, Miro Govedarica, and Mladen Amović. “Model of point cloud data management system in big data paradigm”. In: *ISPRS International Journal of Geo-Information* 7.7 (2018), p. 265. DOI: 10.3390/ijgi7070265.

[114] Jeffrey M. Perkel. “Why scientists are turning to Rust”. In: *Nature* 588 (2020), p. 185. DOI: 10.1038/d41586-020-03382-2.

[115] *pgPointcloud - A PostgreSQL extension for storing point cloud (LIDAR) data*. Accessed: 2023-12-14. 2022. URL: https://pgpointcloud.github.io/pointcloud/index.html.

[116] Massimiliano Pieraccini, Gabriele Guidi, and Carlo Atzeni. “3D digitizing of cultural heritage”. In: *Journal of Cultural Heritage* 2.1 (2001), pp. 63–70. DOI: 10.1016/S1296-2074(01)01108-6.

[117] Romain Pontida. *View from Table Mountain, Cape Town, South Africa*. 2018. URL: https://commons.wikimedia.org/wiki/File:View_from_Table_Mountain,_Cape_Town,_South_Africa.jpg.

[118] *PostGIS*. Accessed: 2023-12-14. 2023. URL: https://postgis.net/.

[119] Florent Poux. "The Smart Point Cloud: Structuring 3D intelligent point data". PhD thesis. ULiège - Université de Liège, 2019, pp. 1–268. ISBN: 9789463754224. DOI: 10.13140/RG.2.2.20457.75367.

[120] Stella Psomadaki et al. "Using a space filling curve approach for the management of dynamic point clouds". In: *ISPRS Ann. Photogramm. Remote Sens. Spat. Inf. Sci* 4 (2016), pp. 107–118. DOI: 10.5194/isprs-annals-IV-2-W1-107-2016.

[121] QGIS Development Team. *QGIS Geographic Information System*. Accessed: 2023-12-14. Open Source Geospatial Foundation, 2009. URL: http://qgis.org.

[122] Richard Fabian. *Data-Oriented Design*. Accessed: 2024-02-11. 2018. URL: https://www.dataorienteddesign.com/dodbook/.

[123] Rico Richter, Sören Discher, and Jürgen Döllner. "Out-of-Core Visualization of Classified 3D Point Clouds". In: *3D Geoinformation Science*. Springer, 2015, pp. 227–242. DOI: 10.1007/978-3-319-12181-9_14.

[124] Martin Roberts. *A new method to construct isotropic blue-noise sample point sets with uniform projections*. Accessed: 2023-12-14. 2020. URL: http://extremelearning.com.au/isotropic-blue-noise-point-sets/.

[125] Marcos Balsa Rodriguez et al. "Interactive Exploration of Gigantic Point Clouds on Mobile Devices". In: *VAST: International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*. Ed. by David Arnold et al. The Eurographics Association, 2012. ISBN: 978-3-905674-39-2. DOI: 10.2312/VAST/VAST12/057-064.

[126] Szymon Rusinkiewicz and Marc Levoy. "QSplat: A multiresolution point rendering system for large meshes". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, pp. 343–352. DOI: 10.1145/344779.344940.

[127] Radu B. Rusu. *The PCD (Point Cloud Data) file format*. Accessed: 2023-12-14. 2022. URL: https://pcl.readthedocs.io/projects/tutorials/en/latest/pcd_file_format.html.

[128] Radu Bogdan Rusu and Steve Cousins. "3D is here: Point Cloud Library (PCL)". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, 2011. DOI: 10.1109/ICRA.2011.5980567.

[129] Claus Scheiblauer. “Interactions with Gigantic Point Clouds”. PhD thesis. Favoritenstrasse 9-11/186, A-1040 Vienna, Austria: Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2014. URL: \url{https://www.cg.tuwien.ac.at/research/publications/2014/scheiblauer-thesis/}.

[130] Patric Schmitz et al. “High-Fidelity Point-Based Rendering of Large-Scale 3-D Scan Datasets”. In: *IEEE Computer Graphics and Applications* 40.3 (2020), pp. 19–31. ISSN: 15581756. DOI: 10.1109/MCG.2020.2974064.

[131] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. “The uncracked pieces in database cracking”. In: *Proceedings of the VLDB Endowment* 7.2 (2013), pp. 97–108. DOI: 10.14778/2732228.2732229.

[132] Markus Schütz. “Interactive Exploration of Point Clouds”. PhD thesis. Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria: Research Unit of Computer Graphics, Institute of Visual Computing and Human-Centered Technology, Faculty of Informatics, TU Wien, Apr. 2021, p. 107. DOI: 10.34726/hss.2021.91668.

[133] Markus Schütz. “Potree: Rendering large point clouds in web browsers”. In: *Technische Universität Wien, Wiedeń* (2016).

[134] Markus Schütz, Lukas Herzberger, and Michael Wimmer. “SimLOD: Simultaneous LOD Generation and Rendering for Point Clouds”. In: *Proc. ACM Comput. Graph. Interact. Tech.* 7.1 (2024). DOI: 10.1145/3651287. URL: https://doi.org/10.1145/3651287.

[135] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. “Rendering Point Clouds with Compute Shaders and Vertex Order Optimization”. In: *Computer Graphics Forum* 40.4 (2021), pp. 115–126. ISSN: 14678659. DOI: 10.1111/cgf.14345. arXiv: 2104.07526.

[136] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. “Software Rasterization of 2 Billion Points in Real Time”. In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 5.3 (2022), pp. 1–16. ISSN: 25776193. DOI: 10.1145/3543863.

[137] Markus Schütz, Katharina Krösl, and Michael Wimmer. “Real-time continuous level of detail rendering of point clouds”. In: *26th IEEE Conference on Virtual Reality and 3D User Interfaces, VR 2019 - Proceedings* (2019), pp. 103–110. DOI: 10.1109/VR.2019.8798284.

[138] Markus Schütz, Stefan Ohrhallinger, and Michael Wimmer. “Fast Out-of-Core Octree Generation for Massive Point Clouds”. In: *Computer Graphics Forum* 39.7 (2020), pp. 1–2. DOI: 10.1111/cgf.14134.

[139] Markus Schütz and Potree Contributors. *Potree*. Accessed: 2023-12-14. 2021. URL: https://github.com/potree/potree/.

[140] Markus Schütz and PotreeConverter Contributors. *potree/PotreeConverter at 1.7*. Accessed: 2023-12-14. 2021. URL: https://github.com/potree/PotreeConverter/tree/1.7.

[141] Markus Schütz et al. “GPU-Accelerated LOD Generation for Point Clouds”. In: *Computer Graphics Forum* 42 Number 8 (June 2023), pp. 1–12. ISSN: 1467-8659. DOI: 10.1111/cgf.14877. URL: https://www.cg.tuwien.ac.at/research/publications/2023/SCHUETZ-2023-LOD/.

[142] Markus Schütz et al. “Progressive Real-Time Rendering of One Billion Points Without Hierarchical Acceleration Structures”. In: *Computer Graphics Forum* 39.2 (2020), pp. 51–64. ISSN: 0167-7055. DOI: 10.1111/cgf.13911. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13911.

[143] *Spatial Database | Oracle*. 2023. URL: https://www.oracle.com/database/spatial/.

[144] Bartosz Taudul. *wolfpld/tracy: Frame profiler*. Accessed: 2023-12-14. 2017. URL: https://github.com/wolfpld/tracy.

[145] The Draco authors. *Draco 3D Graphics Compression*. Accessed: 2023-12-14. 2017. URL: https://google.github.io/draco/.

[146] Peter Thoman et al. “A taxonomy of task-based parallel programming technologies for high-performance computing”. In: *The Journal of Supercomputing* 74.4 (2018), pp. 1422–1434. DOI: 10.1007/s11227-018-2238-4.

[147] Greg Turk. *The PLY Polygon File Format*. Accessed: 2023-12-14. 1994. URL: https://web.archive.org/web/20161204152348/http://www.dcs.ed.ac.uk/teaching/cs4/www/graphics/Web/ply.html.

[148] *USGS 3DEP LiDAR Point Clouds*. Dataset. Accessed: 2023-12-14. URL: https://registry.opendata.aws/usgs-lidar.

[149] Peter Van Oosterom. “From discrete to continuous levels of detail for managing nD-PointClouds”. 2019. URL: http://www.nd-pc.org/documents/vario-nD-PC-v7.pdf.

[150] Peter Van Oosterom et al. “Massive point cloud data management: Design, implementation and execution of a point cloud benchmark”. In: *Computers and Graphics (Pergamon)* 49 (2015), pp. 92–125. ISSN: 00978493. DOI: 10.1016/j.cag.2015.01.007.

[151] Veesus. *Arena4D - Veesus*. Accessed: 2023-12-14. 2023. URL: https://www.veesus.com/arena4d/.

[152] *VERSATILE REAL-TIME LIDAR SENSOR*. Puck. Velodyne. 2019. URL: https://velodynelidar.com/wp-content/uploads/2019/12/63-9229_Rev-K_Puck-_Datasheet_Web.pdf.

[153] Anh-Vu Vo, Debra F Laefer, and Michela Bertolotto. “Airborne laser scanning data storage and indexing: state-of-the-art review”. In: *International journal of remote sensing* 37.24 (2016), pp. 6187–6204. DOI: 10.1080/01431161.2016.1256511.

[154] Sander Vos et al. *A six month high resolution 4D geospatial stationiary laser scan dataset of the Kijkduin beach dune system, The Netherlands*. Dataset. 2021. DOI: 10.1594/PANGAEA.934058. URL: https://doi.org/10.1594/PANGAEA.934058.

[155] Michael Wand et al. “Interactive Editing of Large Point Clouds”. In: *CSymposium on Point-Based Graphics 2007 : Eurographics / IEEE VGTC Symposium Proceedings, Eurogrpahics Association* (Aug. 2008). DOI: 10.2312/SPBG/SPBG07/037-045.

[156] Washington DC government. *District of Columbia - Classified Point Cloud LiDAR*. Dataset. Accessed: 2023-12-14. 2015–2018. URL: https://registry.opendata.aws/dc-lidar/.

[157] Aloysius Wehr and Uwe Lohr. “Airborne laser scanning—an introduction and overview”. In: *ISPRS Journal of photogrammetry and remote sensing* 54.2-3 (1999), pp. 68–82. DOI: 10.1016/S0924-2716(99)00011-8.

[158] Matthew J Westoby et al. “‘Structure-from-Motion’ photogrammetry: A low-cost, effective tool for geoscience applications”. In: *Geomorphology* 179 (2012), pp. 300–314.

[159] Lee Westover. “Interactive volume rendering”. In: *Proceedings of the 1989 Chapel Hill Workshop on Volume Visualization, VVS 1989* Vi (1989), pp. 9–16. DOI: 10.1145/329129.329138.

[160] Francis Williams. *Point Cloud Utils*. Accessed: 2023-12-14. 2022. URL: https://www.github.com/fwilliams/point-cloud-utils.

[161] Michael Wimmer and Claus Scheiblauer. “Instant Points: Fast Rendering of Unprocessed Point Clouds”. In: *Symposium on Point-Based Graphics*. Ed. by Mario Botsch et al. The Eurographics Association, 2006. ISBN: 3-905673-32-0. DOI: /10.2312/SPBG/SPBG06/129-136.

[162] Wen Xiao et al. “Geoinformatics for the conservation and promotion of cultural heritage in support of the UN Sustainable Development Goals”. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 142 (2018), pp. 389–406. DOI: 10.1016/j.isprsjprs.2018.01.001.

[163] Xiaoyuan Yang et al. “Three-dimensional forest reconstruction and structural parameter retrievals using a terrestrial full-waveform lidar instrument (Echidna®)”. In: *Remote sensing of environment* 135 (2013), pp. 36–51. DOI: 10.1016/j.rse.2013.03.020.

[164] Matei Zaharia et al. “Apache Spark: a unified engine for big data processing”. In: *Communications of the ACM* 59.11 (2016), pp. 56–65. DOI: 10.1145/2934664.

[165] Weijian Zhao et al. “Automated recognition and measurement based on three-dimensional point clouds to connect precast concrete components”. In: *Automation in Construction* 133.September 2021 (2022), p. 104000. ISSN: 09265805. DOI: 10.1016/j.autcon.2021.104000. URL: \url{https://doi.org/10.1016/j.autcon.2021.104000}.