MSc thesis in Geomatics

Using Foreign Data Wrapper in PostgreSQL to Expose Point Clouds on File System

Mutian Deng

November 2020

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Geomatics

The work in this thesis was carried out in the:

Geo-Database Management Centre

Faculty of Architecture & the Built Environment
Delft University of Technology

Supervisors: Dr. Martijn Meijers
Prof.dr. Peter van Oosterom
Co-reader: Dr. Hans Hoogenboom

Abstract

In the recent years, the point clouds data becomes an important source of geographic information thanks to the rapid development of data acquisition techniques and the wide range of the applications. Light Detection And Ranging (LiDAR) is the main survey campaign that generates highly accurate point clouds data in an efficient manner. LiDAR point clouds data has two main features: on the one hand, a point clouds data has a big volume containing million and even billion point records, on the other hand, it is multi-dimensional which means each point records consist of several fields, there can be Red,Green and Blue (RGB) values, intensity and gps time next to the 3D position. In order to release the great potential of this kind of geospatial data, the point clouds data should be made full and deep use. In this context, an efficient solution for handling LiDARdata is required. There are two main solution to manage LiDARdata, one is file-based solution using files for storage and file tools for processing; another one is Database Management System (DBMS) solution storing data in the database and supporting capabilities on LiDAR data. Both solutions have it own benefit and problems, thus a hybrid solution is proposed in this research.

This research is aimed to use a hybrid solution to combine the advantages of file system solution and DBMS solution, therefore the Foreign Data Wrapper (FDW) is introduced and developed in this research. FDW is an extension of PostgreSQL that can access data residing outside the PostgreSQL using Structured Query Language (SQL) language. Thus be means of FDW, we can use LiDAR point clouds on file system directly in PostgreSQL. Since in the real-world applications, different LiDAR files are collected in a file system rather than using only massive file, it is necessary to develop a FDW of Point Cloud Data Management System.

The aim of this research is to answer the main research question: to what extent we can use LiDAR point clouds directly in the PostgreSQL by means of FDW, and thus a FDW supporting the Point Cloud Data Management System is implemented. Then, the range and performance of its functionality are evaluated. The results shows this FDW solution is feasible while the querying time is relevant to the number of returned points. The benefit and problem of this FDW are analyzed.

Acknowledgements

My first gratitude is expressed to my mentors: Martijn and Peter who help me a lot in the thesis. The graduation cannot be finished without their support. Martijn provides me many technical support and direction when I got lost in dealing with troubles and when I missed some important points to consider in the research. Peter gives me a lot helpful advice from his broad knowledge which give me direction when I am unsure about the method. Due to the Corona, the research of graduation thesis is carried out online and it became super difficult to have face-to-face meetings. Martijn, Peter and I hold the progress meeting once two weeks and they both attended the meeting and gave me very valuable supervise during every meeting.

Finally I want to thanks to my parents who gives me financial support that let me gain this great study and life experience in Delft. And I also want to thanks to my friends for their accompany and help.

Contents

List of Figures

List of Tables

Acronyms

GDAL Geospatial Data Abstraction Library . . . 12
DEM Digital Elevation Model . . . 2
DTM Digit Terrain Model . . . 2
GIS Geographical Information System . . . 2
FDW Foreign Data Wrapper . . . v
DB Database . . . 3
DBMS Database Management System . . . v
LiDAR Light Detection And Ranging . . . v
TLS Terrestrial Laser Scanning . . . 1
ALS Airborne Laser Scanning . . . 1
GeoDBMS Spatial Database Management System . . . 13
ASCII American Standard Code for Information Exchange . . . 3
ASPRS American Society for Photogrammetry and Remote Sensing . . . 8
SQL Structured Query Language . . . v
PDAL Point Cloud Abstraction Library . . . 3
GNSS Global Navigation Satellite System . . . 1
INS Inertial navigation system . . . 1
MED Management of External Data . . . xi
GI Geographical Information . . . 7
AoI Area of Interest . . . 2
RDBMS Relational Database Management System . . . 13
CSV Comma Separate Values . . . 34
AHN Acteel Hoogtebestand Nederland . . . xiii
EPSG European Petroleum Survey Group . . . 38
SRID Spatial Reference System Identifier . . . 13
UAV Unmanned Aerial Vehicle . . . 1
PDAL Point Data Abstraction Library . . . 3
API Application Programming Interface . . . 12
CRS Coordinate Reference System . . . 8
HTML Hypertext Markup Language . . . 17
DDL Data Definition Language . . . 19
DML Data Manipulation Language . . . 31
BBX bounding box . . . 41
VLR Variable Length Record . . . 9
EVLR Extended Variable Length Record . . . 9
OGC Open Geospatial Consortium . . . 14
WKT Well-Known Text . . . 14
WKB Well-Known Binary . . . 13
SFC Space Filling Curve . . . 25
I/O Input/Output . . . 3
RGB Red,Green and Blue . . . v
LAS LASER . . . 9
LAZ LAZip . . . 9
GUI Graphic User Interface . . . 11
NoSQL Not Only SQL . . . 13
JSON JavaScript Object Notation . . . 12
GiST Generalized Search Trees . . . 25
XML eXtensible Markup Language . . . 14

1 Introduction

1.1 Background

Nowadays, point clouds data becomes an important source of geographic information because of the development of point clouds data acquisition technologies and wide range of applications. Point clouds data is simply a collection of three-dimensional points embedded in Cartesian space while each point record has a series of fields containing the 3D coordinates and attributes like intensity and classification. Same as vector data and raster data, point clouds data gives an approach to represent objects in the natural and built environments.

The advanced surveying technologies provide the possibility for the popularity of point clouds data, while the attribute fields of point clouds depend on the measurement method as well. Point clouds data can be generated from laser scanning namely LiDAR, and photogrammetry [Van Oosterom et al., 2015]. Photogrammetry and LiDAR are both advanced methods to produce point clouds data but in different manner, photogrammetry obtains 3D information from the images of the same target like dense image matching, while laser scanner uses laser beams to detect and measure the target, which is more widely used. There are different platforms for LiDAR surveying, including Terrestrial Laser Scanning (TLS) terrestrial platform with fixed tripods and mobile vehicles, Airborne Laser Scanning (ALS) airborne platform with aircraft, helicopter and Unmanned Aerial Vehicle (UAV) like drones. The modern data acquisition technologies support the generation of point clouds in large scale i.e. having million and even billion point records with rapid rate i.e. in several seconds and high accuracy i.e. calculated coordinate and attribute values.

In this thesis, we study the point clouds produced by LiDAR instruments. The LiDAR system utilizes flight planning system, a Global Navigation Satellite System (GNSS) for positioning, an Inertial navigation system (INS) for navigation, and laser scanning system with laser transmitter and receiver for detection and ranging, and photogrammetric camera, all together to achieve the high measuring accuracy and frequency [Chrószcz et al., 2016], thus LiDAR can capture target objects in an effienct and georeferenced manner.

LiDAR measures the target by lighting the target with signal sent by a laser scanner and measuring the reflection with a laser sensor. Scanning is based on the measurement of distance from scanner to the test area as this determines the time delay between sending and receiving the laser pulse when range is time difference multiplied by speed of light [Chrószcz et al., 2016]. The pulse detecting objects will be used to record the time from its emitting to receiving and to calculate the range distance between target and scanner, and then stored in a discrete three-dimensional points set where the 3D coordinates of the reflected target in object space are transformed from distance measurement by analyzing the distance to the scanner combined with laser scan angle and scanner position and orientation information [Ott, 2012].

A set of 3D points with records of target is collected, the information of position and scanning measurement is attached with point records as x, y, z coordinates and attributes respectively. These attributes are highly related to the way that the point clouds data are collected. The number of points obtained per square meter of the test area is dependent on the flight parameters such as altitude and the number of passes [Chrószcz et al., 2016]. Additional attribute information is stored besides every x, y and z positional value. LiDAR point attribute values are maintained for each recorded laser pulse like intensity, classification, GPS time stamps,RGB values, scan angle and scan direction, return number, number of returns, edge of the flight line, etc.

The frequent use of point clouds data also comes from the wide variety of applications. Point clouds data can be used to depict the object in the reality environment. It can also be used to generate Digital Elevation Model (DEM) and Digit Terrain Model (DTM). Further point clouds data can be used as essential geographic information resource in many fields, like built environment, agriculture and autonomous vehicles, vegetation and soil science. Point clouds data contains great potential in society and science. The value of point clouds data can be realized in the usage stage.

The characteristics of point clouds data are derived from the data measurement nature, on the one hand, the point clouds data is always massive with millions and even billion point records, resulting from that sensing technologies can collect a big volume of points in a short period of time. For example, AHN dataset has 64 billion points describing the elevation of whole country of the Netherlands. On the other hand, the point clouds data is multi-dimensional, each point record is attached with several fields, one part is x, y, z coordinates representing the 3D position, and another part is attributes of measured point like intensity and classification. Point clouds data offers abundant information that can be used in different analysis [Pajić et al., 2018]. Therefore, LiDAR data require storing in appropriate structures.

In order to release the potential of point clouds data, it should be made as full and wide range use as possible. Due to the large-amount and multi-dimensional natures of point clouds data, the management of point clouds still remain challenging, while the availability and usage of point clouds are increasing.

1.2 Problem statement

The efficient management solution for point clouds data is required for further use of this geospatial data which contains different potential use. The management of point clouds data includes the storage, loading/accessing, filtering/selecting, manipulating and processing, visualization. In addition to traditional geographical data like raster data and vector data, point clouds data is getting more appreciated as a new type of geographic data because of its true three dimensional nature and high acquiring efficiency and availability [Cura et al., 2017], while the specialties of this data type also make the traditional spatial data management solutions not suitable enough.

Point clouds data resembles the features of Geographical Information System (GIS) data, both raster data and vector data. LiDAR share with the gridded model i.e. raster data the sampling nature, and they also share with the object model i.e. vector data the form to represent arbitrary georeferenced points [Janecka et al., 2018]. In this context, as similar to vector data and raster data with their original nature, the similar management approaches may be used by point clouds data.

A common practice for geographic data management is using files in GIS file formats and processing tools. Point clouds data can also utilize this file-based solution, both original form as points and converted GIS form in general can be used for point clouds data while specific tools are developed for processing. Therefore, the majority of GIS data transformation and processes can be applied to point clouds data as well. The point clouds data can be extracted and stored as raster data, which is normally the form in which point clouds data is given to the end users, like DEM.

An alternative DBMS is gaining discussion in terms of its rich functionalities like combination and querying, and geographic extensions of developed DBMS like PostGIS have already been able to support the storage and functions of vector data and raster data. In this DBMS context, because of the integrated features of both rasters and vectors, point clouds data can reuse the existing data type for GIS data. On the one hand, it is possible for point clouds data to be stored in a standard data type that vector data uses since the point clouds data is a collection of points, for example, in a *Simple Feature* multipoint in one row or in several rows, each with a single point [Van Oosterom et al., 2015]. However, using a geometry multipoint collection type to store point clouds may not be really feasible in practice although it is theoretically correct, because it exceeds the maximal number of rows which usually less than to one million rows, and also it is not possible to search and access a points set within Area of Interest (AoI) because the total set of points is stored as one single collection, each time a query is executed on this collection, the entire dataset need to be loaded into memory [Ott, 2012]. On the other hand, storing point clouds data as raster data also seems reasonable because they are both based on sampling of real word objects, for

example, as encoded GeoTIFF coverage [Van Oosterom et al., 2015]. However, point clouds data is not a regular grid while raster data uses equally sized and contiguous pixels.

Therefore, an appropriate solution for point clouds data management is required to support this specific data. Based on the description in [Van Oosterom et al., 2015] point cloud benchmark, the point clouds data type and its operations need to cover the following aspects.

1. x,y,z specify the basic storage of the coordinates in a spatial reference system using various base data types like int, float.
2. Attribute per point: 0 or more. The attributes are the fields of point record..
3. Data organization based on data coherence is referred as blocking schema.
4. Efficient storage with compression algorithm.
5. Data pyramid support: level of detail(LoD), multi-scale or vario-scale.
6. Query accuracy: geometries to report points.
7. Operations/functionalities.
8. Parallel processing for operations.

How to handle the point clouds data is still a challenging question, because of on the one hand, not only its allied nature between raster and vector data, but also its massive-volume and multi-dimensional features; on the other hand, the management need to fulfill the requirements in further operations of point clouds data.

First of all, point clouds data sets are massive containing million and billion points, a storage mode concerning both space saving and fast reading is needed at the basic stage. File system approach provides several dedicated data type like in byte format and American Standard Code for Information Exchange (ASCII) format aiming at efficient storage. Secondly, the point clouds data has great potential because of its very different usage, the users always need to access just a part of the data at once, thus efficient selecting a subset is important. DBMS provides unified SQL language for data retrieving within a AoI. In terms of spatial data selection, it is useful to apply spatial access method, the technologies of spatial indexing and clustering can be helpful for more efficient selection. Point clouds data usually are geographic data as having 3D coordinates, thus point clouds can be used conjointly to other data types,i.e. combined with other GIS data, either directly or by converting point clouds to vectors and raster [Cura et al., 2017]. In addition to the position fields, the fields of point clouds can be different depending on their source regarding the number and type of the attributes. Therefore, point clouds data can be related to not only spatial data but also non-spatial data, such as raster, vector and administrative data. The users of point clouds data vary in different domains, it is necessary to share data rather than duplicating them, thus several uses can simultaneously read and write the same data, this concurrency issue requires the data manipulation of management schema. Another problem is about handling point clouds data set which contains not only points content but also metadata and other information. Managing such datasets includes asking which datasets are available and where those are [Cura et al., 2017]. In the real application, point clouds data come from different sources, thus having different data specification, accuracy and resolutions need to be handled, requiring using on the common environment. The DBMS establishes a layer of abstraction over the file system [Cura et al., 2017].

The efficient and effective ways for management of point clouds data have been researched these years. The methods can be divided into two main parts: file-based solution and DBMS solution. File-based solution uses a collection of files and specific tools to store, select, manipulate and visualize the point clouds data. The DBMS solution creates a layer of abstraction over the file system, with a dedicated data retrieval language like SQL [Cura et al., 2017]. The traditional file-based solution supports efficient data access in their original formats, and Input/Output (I/O) processing by powerful point clouds processing tools like LAStools and Point Data Abstraction Library (PDAL), but it has major drawbacks like data isolation since the data are stored in individual files separately, and as a result data redundancy and inconsistency, as well as application dependency [van Oosterom et al., 2017] DBMS solutions can obstacle these drawbacks, the database community, commercial and open-source, like object relational DBMS PostgreSQL, Oracle, and column-stored Database (DB) like MonetDB can offer support, those have been

	DBMS solution
Advantages	1. unified SQL language for querying 2. combination with other geographic data 3. updates, inserts and deletes 4. multi-user access
Disadvantages	1. efficient storage model still in need 2. no fast data importing

Table 1.1: Advantages and disadvantages of DBMS solution

	File system solution
Advantages	1. efficient and standard storage 2. powerful processing tools 3. lossless compression algorithm 4. ease of use
Disadvantages	1. no fast access to subset of data 2. limited to data combination

Table 1.2: Advantages and disadvantages of file system solution

used with images and objects mode for a long time [Cura et al., 2015]. They also provides point clouds specific data structures like Oracle/Spatial and Graph and PostgreSQL/PostGIS [Van Oosterom et al., 2019].

Both file-based solution and DBMS solution have its advantages and disadvantages as showed in table 1.2 and table 1.1, and also both of them satisfy part of the aspects of point clouds data management. In this case, a hybrid solution for point clouds data management is proposed, using FDW to expose point clouds data in PostgreSQL stored on file system.

1.3 Research question

This research is aimed to figure out the solution of foreign data wrapper takes the advantages of both file-based solution and DBMS solution to offer efficient handling of point clouds data. The research will be carried out by answering the main research question:

To what extent we can use LiDAR point clouds directly in the PostgreSQL by means of Foreign Data Wrapper (FDW).

In order to answer the main research question, the following sub-questions which are relevant will also be answered.

1. How does the FDW make connection between the LiDAR file system and PostgreSQL to realize the accessing and functionalities of point clouds data?
2. What kinds of queries and what levels of selection can be executed on the LiDAR data?
3. What sizes of AHN3 datasets can work well by FDW method?
4. What are the benefits and problems when using hybrid FDW solution?

1.4 Thesis outline

The thesis is structured containing the following chapters as follow.

- Chapter 2 will give a detailed introduction about the related work of the thesis and present an overview of developed methods for point clouds data management along with their main benefit and problems. Additionally, to summarize the standard of SQL to manage external data which is supported by foreign data wrapper, the stated hybrid solution of this research.
- Chapter 3 will introduce the underlying theory to answer the research questions including the methodology used to achieve this goal and the design of a Point Clouds Data Management System to be put into practice. This research is aimed to answer the question that to what extent we can use point cloud data directly in PostgreSQL by means of foreign data wrapper. Foreign data wrapper is proposed as a combined method for handling point cloud data between file system and database management system. The steps of using point clouds data are divided into data storage, data preparation/organization, data access and data functionalities.
- Chapter 4 will describe the implementation of FDW for Point Clouds Data Management System in details and the set up of benchmark to evaluate this Point Clouds Data Management System FDW in the spactes of feasibility, efficiency and scalability. The design and concept of methodology are introduced in Chapter 3. This chapter focuses on the details of implementation and benchmark of the developed method.
- Chapter 5 will present the results of the benchmark in three aspects as the proof of concept and design, and provide an analysis of them as well.
- Chapter 6 will summarise the main result and give an answer to the research question. Concludes the method of this Point Clouds Data Management System FDW. Finally, the future work is give that can be carried out to address more question arose in this research or improve the performance.

2 Related work

This chapter will give an overview of presented methods for point clouds data management. Section 2.1 introduces the main characteristics of point clouds data which cause problems when handling point clouds data in the procedure. Section 2.2 and section 2.3 introduce the related file-based solutions and database management system solution using various tools in details respectively along with their advantages and disadvantages. 2.4 describes SQL MED which is the underlying concept of the proposed hybrid solution proposed in this research.

2.1 Point Clouds Data Management

As described before, point clouds data, being an array of 3D points, resembles both vector data and raster data. Because of the georeferenced feature, point clouds data can be used as a Geographical Information (GI) resource to represent the objects in the reality. Point clouds data can be the third type of spatial representation in the GIS [Van Oosterom et al., 2015]. In this context, as similar to vector data and raster data of their original nature, the similar management approaches of GIS data can be used by point clouds data. Management of point clouds data is being researched and improved in these years, it can be divided into two main approached: file-based approach and DBMS approach.

Reusing the existing handling approached that have been applied by GIS data, i.e. vectors and rasters, is not sufficient and suitable for point clouds data. On the one hand, the management should settle the point clouds data features of massive quantity and multi-dimensions produced by the high accuracy and high speed acquisition technologies. On the other hand, the management need to fulfill the requirements in further operations of point clouds data in the variety of applications.

2.2 File system

The management of 3D point clouds is currently usually file-based [Otepka et al., 2013] using one file or collection of files and specific tools to manage and process.

Not only having geospatial information and being able to represent the reality objects, point clouds data also share the sampling nature with raster and arbitrary nature with vector. Therefore, point clouds data can be converted to these traditional GIS models. Raster models can be derived from raw LiDAR data. Such converted image model makes it possible for point clouds data to be analyzed in common GIS tools and offers a reduction of data i.e. thinning and extracted information, however it also causes a irreversible loss of information [Meyer and Brunn, 2019]. While vector points represent unstructured, original point measurements with higher degree of details [Höfle et al., 2006]. However, GIS software packages are unable to manage and process billions of vector points in an efficient and easy-to-use way [Höfle et al., 2006].

A point clouds data is essentially an array of 3D points, while also indicates how it is stored in files. Despite the format, a point clouds file can often been regarded as an array of point records, each of which contains the coordinates and attributes of one point. A point record consists of several fields representing information of a point, each of them stores a single values like integer or float. The order and meaning of the fields in a record is fixed for all the point records in one file. The specific format defines the way that the point records are structured and stored in the file and whether additional metadata is provided.

The file formats for storing point clouds data and also processing tools need to be appropriate for the structure of point records. The filed-based solutions for point clouds data store raw LiDAR point clouds in file, which is output of laser scanning campaign [Samberg, 2007]. Many different file formats exist, some of them are vendors adopt their own. Some are in common exchange formats as either ASCII or binary files, those are usually organized by flight strips, scan positions or tiles of each data acquiring campaign [Rieg et al., 2014]. Since the publication of the *LAS specification 1.0*, the LAS format of the American Society for Photogrammetry and Remote Sensing (ASPRS) has appeared as a standard for LiDAR data [Rieg et al., 2014]. Typical file-based solutions use desktop applications usually vendor specific and command line executables [Boehm, 2014], like LAStools and the PDAL. The workflow includes reading one or more files, conducting some degree of processing and writing one or more files back to use [Van Oosterom et al., 2019].

2.2.1 File formats

When designing the processing tools, the primary issue at first is to clearly make the definition of the inputs and outputs. For LiDAR data, how to specify the format of input and output file is therefore important as the basis of file system management of point clouds data [David et al., 2008]. How exactly the point records are structured and stored in the file, and what additional metadata is available, depends on the specific file format that is used. The different solutions of defining LiDAR data is heavily dependent on the laser scanning manufactures, software companies, data providers and end users [David et al., 2008]. On the one hand, such file format must be compatible with existing file formats taht have been standardized; On the other hand, it has to be flexible and has to take account into more information both position and attributes [David et al., 2008].

ASCII format

LiDAR data in its raw form is a series of 3D points stored as x, y, z coordinates where x and y can be longitude and latitude and z is the height in meters. A simple ASCII file with extensions of *.txt*, *.csv* or *.xyz* could be enough where each line has a coordinates x, y, z separated by a delimiter e.g. comma, tab, etc. to represent data [Ott, 2012]. Storing LiDAR data in plain text files as ASCII formats, the information is thus stored as a sequence of ASCII characters, usually one point record per line.

In general ASCII formats are text files containing lists of 3D points arranged in columns. Any regular columned ASCII format can be used if it consists of the main information: number of lines to skip at the beginning of the file, 3D coordinate columns and optionally other attribute columns [Samberg, 2007]. Given that several ASCII formats exist depending on the information (columns) available, it can be fundamental *xyz* file or extended version with classification, *xyzc*, other features can be added as ASCII columns and in any possible orders [David et al., 2008]. One benefit of ASCII files is that the user can simply open them in a text editor. The biggest downside is that they are not standardised at all, i.e. the type, order, and number of attributes varies and also the used Coordinate Reference System (CRS) is usually not documented in the file.

The PLY format is introduced and used as a standardised ASCII format. There is a header file in the PLY file, which specifies the structure of the point records stored the file, i.e. the number of attributes, their order, as well as their names and data types. Thus PLY files apply a flexible standard, since the user can decide on the composition of the point record. PLY files can be read by many softwares. Additionally, how to specify the CRS in a PLY file has not been standardised, although the user could add comment stating the CRS in the header. The PLY format aims at providing a simple but enough general to be useful for a variety of systems.

ASCII formats have been used frequently because they are both human-readable and machine-readable, but can become very huge size. Compared to the ASCII encoding, the binary encoding results in a smaller file size and fast reading and writing from and to the file. Although generic and simple ASCII format is suitable for most of point clouds data storage and processing, the large file size still would be a limiting factor. Besides, the undefined file construction like the number and order of columns as well as name,

data type and size of each column and different ASCII file formats produced by scanners, this can cause ambiguity and data loss when working between multiple software packages for processing [Kumar et al., 2019]. Therefore, an uniform file format maintained by standard for LiDAR data I/O processing in multiple users and tools enrolments is required.

LAS format

The need for LiDAR file standard is raised to support the different inputs and outputs as well as various tools environment. There are several reasons why the standard is proposed [Samberg, 2007].

- Manufactures use different file specifications from different surveying systems.
- There exist several LiDAR file formats and this leads to difficulties when exchanging data.
- Different inputs and outputs need to have unified software support.

The pubic LASER (LAS) format is the most widely used standard for the distribution of point cloud data. The LAS standard is maintained by the ASPRS organisation and, as the name of format implies, it was designed for datasets that originate from laser scanners. However, in practice it is also used for other types of point clouds data, e.g. the datasets derived from dense image matching. LAS format is a binary-encoded standard and compared to the PLY format it is rather strict when encoding because it prescribes exactly what attributes exist and how many bits each attribute uses.

The LAS format stores binary data which consists 4 sections [ASPRS, 2019]. The Public Header Block contains generic information such as the total number of points and the bounding box of the point clouds data. The CRS of the point clouds data can be stored in the header of the LAS file as well. The Variable Length Records is where projection information, metadata, waveform packet information, and user application data are include as variable types of data.

* Public Header Block
* Variable Length Record (VLR)s
* Point Data Records
* Extended Variable Length Record (EVLR)s

Point Data Record Format 0, as list in table 2.1, contains the core 20 bytes that are shared by point data record format 0 to 5. It is the simplest record type available for LAS file. Other record types can include more fields i.e. RGB colour values or the GPS time which is the time a point was measured by the scanner, but all records types include at least the fields of Point Data Record Format 0 as show in table 2.1. The X, Y and Z values are stored as long integers. The X, Y and Z values are used in conjunction with the scale values and the offset values to determine the coordinate for each point, i.e. the X, Y and Z fields need to be multiplied by a scaling factor and added to an offset value as described on the Public Header Block. While the LAS standard clearly specifies that all these fields are required, some of the fields are very specific to LiDAR survey campaign, and they are sometime neglected in practice, for example when a point clouds data generated from dense image matching is stored in LAS format. The unused fields will still take up storage space inn each records. In addiction, each point record has an attributes of classification indicating which which object type the point represents, as listed in table 2.2.

LAZ format

LAZip (LAZ) format is a compressed version of LAS format, i.e. the output of LASzip, a lossless compressor for LiDAR data stored in the LAS format. The compression tool LASzip is and maintained and developed by an organization named rapidlasso GmbH, which is not an official organization like ASPRS who maintains the LAS format specification.

As introduced in the previous sections, the LiDAR data is stored in vendor-defined, binary format when it is generated from the scanning system. In order to facilitate the data interoperations between different users and software packages, the LiDAR data is converted to ASCII which simply represent the attributes

Item	Format	Size
X	long	4 bytes
Y	long	4 bytes
Z	long	4 bytes
Intensity	unsigned short	2 bytes
Return Number	3 bits(bits 0-2)	3 bits
Number of Returns	3 bits(bits 3-5)	3 bits
Scan Direction Flag	1 bit (bit 6)	1 bit
Edge of Flight Line	1 bit (bit 7)	1 bit
Classification	unsigned char	1 byte
Scan Angle Rank	char	1 byte
User Data	unsigned char	1 byte
Point Source ID	unsigned short	2 bytes

Table 2.1: LAS file Point Data Record Format 0

classification value	meaning
0	created, never classified
1	unclassified
2	ground
3	low vegetation
4	Medium vegetation
5	High vegetation
6	Building
7	noise
8	mass point
9	water

Table 2.2: LAS classification values (the first 10)

of a single reflection by listing in each line. Although the ASCII format is easy to understand and flexible, it is problematic to store million and billion of LiDAR records in a textual format because the size grows large and parsing the data is of low efficiency, and it is also impossible to search inside the file. ASPRS addressed these issues by defining a simple binary , exchange and public format, LAS format, which has developed a standard for the storage and dissemination of LiDAR data [Isenburg, 2013].

One of the significant features of LAS format is that it stores the 3D coordinate values as the scaled and offset integers. In this case, before the storing the coordinates, it is required to consider the needed precision level in the surveyed data sample and to choose the suitable actual increments. It prevents the unnecessary storing of scanning noise. The absence of incompressible noise makes it possible to efficiently compress the LiDAR points in a completely lossless manner [Isenburg, 2013]. LAZip delivers high compression rates at unmatched speeds and supports streaming, random access decompression [Isenburg, 2013].

LAZ format is an open standard and it is widely used for, especially for very large dataset. Through the use of lossless compression algorithms, a LAZ file can be packed into a fraction of the storage space required for the equivalent LAS file without any loss of information. The LAZ format closely resembles the LAS format, i.e. the header and the structure of the point records are virtually identical. In a LAZ file the point records are grouped in blocks of 50,000 records each. Each block is individually compressed, which makes it possible to partially decompress only the needed blocks from a file (instead of always needing to compress the whole file). This can save a lot of time if only a few point from a huge point cloud are needed. Notice that the effectiveness of the compression algorithms depends on the similarity in information between subsequent point records. Typically information is quite similar for points that are close to each other in space. Therefore, a greater compression factor can often be achieved after spatially sorting the points.

LAZ does not compress the LAS header or any of the VLRs, but just copies them unchanged from LAS file to LAZ file. LAZ allows the user to use compressed LAZ format files just like the standard LAS format files. The users can load them directly from compressed into the intended application without needing to decompress them onto disk first [Isenburg, 2013]. The LASzip compressor encodes the points in group of points, allowing seeking in the compressed file. The default grouping size is 50,000 points. Grouping make it possible to support AoI queries that decompose only the relevant parts of a compressed LAZ file [Isenburg, 2013].

2.2.2 File tools

As there have been existing several file formats, on the one hand, from the points of software design, the developers should consider the variety of raw data format and make the processing tools work in all possible cases. One the other hand, from the points of making an agreement with commercial vendors, users should be aware of the merit and demerit of different data formats so as to consume the satisfactory data for extracting useful information indented. [Chen, 2007].

Current LiDAR point clouds processing workflow normally uses classic desktop software packages or command line interface executables. Many of these programs read one or multiple files, perform some degree of processing and write one or multiple files. There are several free or open source software collections for LiDAR processing like LAStools and some tools from GDAL and PDAL [Boehm, 2014].

LAStools

LAStools is a collection of highly efficient, batch-scriptable, multicore command line tools [rapidlasso, 2019]. All of the tools can also be run through a native Graphic User Interface (GUI) and can be used as a LiDAR processing tool boxes for ArcGIS, for QGIS , and for ERDAS IMAGINE. "LAStools are the fastest and most memory-efficient solution for batch-scripted multi-core LiDAR processing and can turn billions of LiDAR points into useful products at high speeds and with low memory requirements" [rapidlasso, 2019].

LAStools is a stand-alone application where all these tools are combined with a framework managing project and controlling quality [Hug et al., 2004]. LAStools give ability to efficient handling of different kinds of project data including to import and geocode raw LiDAR and image data, as well as system calibration, filtering and classification of LiDAR data, generation of elevation models, and export of the results in various common formats [Hug et al., 2004]. "LAStools supports an integration LiDAR processing environment including multi-user, networked production process, workflow management, and quality control for operational and efficient LiDAR data production" [Hug et al., 2004].

Additionally, LASlib in LAStools is a C++ programming Application Programming Interface (API) for reading and writing LiDAR data stored in standard LAS or in compressed LAZ format.

PDAL

PDAL is an open source library and applications for translating and manipulating point cloud data which show the similarity with Geospatial Data Abstraction Library (GDAL) library which handles raster data and vector data. PDAL allows the user to compose operations on point clouds into pipelines of phases. JavaScript Object Notation (JSON) syntax and the available API can be used to write these pipelines.

When compared to LAStools, in PDAL all components of are published with open source licence and allows application developers to use their extensions that perform as phases of processing pipelines. PDAL can operate on point clouds data of any format not just LAS format. While LAStools can read and write formats other than LAS, but it still relates all data to its internal handling of LAS data, thus limiting it to field numbers and types supported by the LAS format.

libLAS is a C/C++ library for reading and writing the LiDAR file in LAS format. libLAS support the ASPRS LAS format specification versions: 1.0, 1.1, 1.2. And it has been replaced by PDAL project.

2.2.3 Pros and Cons

File-based solution has proven to be a east to use and reliable form to store and exchange LiDAR data [Boehm, 2014]. The idea of using file-centric storage in LiDAR data management is the observation that large collections of LiDAR data are typically delivered as large collection of files, rather than single files of terabyte size [Boehm, 2014]. the AHN dataset is a good example, not storing the whole Netherlands point clouds data in the single file, instead of using several files to manage it. File-based solution for point clouds data storage is therefore applicable. Another strength is that the LiDAR file tools like LAStools and PDAL, make it possible to conduct the data manipulation and processing with the originated format file as input, and the user can decide the output formats.

However, file-based solution has limitation during use in the heterogeneous environment. For example, our benchmark dataset AHN3 is stored and distributed in more than 60,000 LAZ files with different sizes and extents.It is not really efficient for simple point cloud selection purposes. In this context, DBMS provides an easier to use and more scalable alternative [Van Oosterom et al., 2015]. File-based management systems are normally built around one file format [Cura et al., 2017], and are not necessarily compatible with other formats. Thus, using several point clouds together can be difficult, since multiple point clouds can come from different vendors and scanners and have different formats. A specific file-based solution with the dedicated file format and processing tools can be efficient handling point clouds data, but in the reality, users need to utilize different type and range of data including spatial data and no-spatial data, e.g. raster data, vector data, administration data and temporal data. Therefore, DBMS supports a generic and standardized way to combine other data in query, for example, if a integrated query is given that select points form a point clouds data. which overlap with the 3-meter buffered polygons of buildings owned by one person [Van Oosterom et al., 2015]. Furthermore, file-based solution provides limited functionalities which mainly developed by vendors. While DBMS offers unlimited functionalities thanks to the SQL support, like multiple users can share data by AoIselection, update/delete/insert operations, dealing with metadata. In addition, DBMS use optimized disk I/O strategies. Finally, most recent DB systems support automatic parallelization when executing queries [Van Oosterom et al., 2015].

2.3 Database Management System

As an alternative to an exclusively file-based handling, it is meanwhile possible to manage point clouds data within Spatial Database Management System (GeoDBMS). There are several DBMS supporting point clouds data management, both Relational Database Management System (RDBMS) and Not Only SQL (NoSQL) DBMS, either open source or commercial.

Using DBMS to management point clouds data, the storing and importing come to the beginning place, the first initiative is to reuse the existing data types like *simple feature* geometry. *POINT* is proposed since the point clouds data is essentially a collection of 3D points, [Höfle et al., 2006] store the Cartesian coordinates of one LiDAR point as a *POINT* in a single table field, hence the SQL can be used to apply geometric functions and spatial queries on the geometry objects. [Höfle et al., 2006] build the storage of point geometry which consists a description of byte order and Well-Known Binary (WKB) type, three coordinates, Spatial Reference System Identifier (SRID) and bounding box. [Zlatanova, 2006] argued that the point clouds can be organized in DBMS by either using the supported data types like *POINT* and *MULTIPOINT* or creating a user-defined type. Th*POINT* data types make it possible to handle all these attributes of each point record. The major problem comes from data storage and indexing, which are very consuming because one point takes on records. The benefit of *MULTIPOINT* data types is efficient indexing, however the points are no longer identified because of grouping. Depending on size of the point clouds and the point distribution, it costs time for the operations and thus difficult to manage. [Wijga-Hoefsloot, 2012] proposed a *POINTCLUSTER* which is comparable to *MULTIPOINT* of and it showed optimal performance in large point clouds database.

Addition to the predefined data types for point clouds data storage, creating the dedicated storage and management for point clouds data has been the centre in the research fields. Currently, the DB community provides their support for point clouds data specially, like PostgreSQL with PostGIS and PgPointcloud, Oracle SDP_PC package. In these supported databases, the storage modes can be distinguished into two part: flat table and blocks. The first model is to store each point separately in one row, while block model is group the spatially close points together.

2.3.1 Databases

PostgreSQL

PostgreSQL

is an open source and powerful RDBMS, which is popular because of its variety of features and great performance. PostgreSQL is the research tool of this research. PostgreSQL conform with the SQL standard. It provides support for a wide range of data types including primitives (integer, numeric and string, boolean) and customization like composed types and custom types. PostgreSQL is also highly extensible, the users can create user-defined data types and functions, and use different programming languages to write code in . The supported features consist data integrity, concurrency, reliability and security, and also gain ability for extensibility like stored functions and procedures, procedural language and foreign data wrapper for connection to other source, many extensions that provides additional functionality.

In order to store and query the geographic and spatial objects the extension PostGIS is introduced. PostGIS adds extra types including raster, geometry, geography and others to the PostgreSQL database. It also adds functions, operators, and index enhancements that apply to these spatial types [OSGeo, 2019b]. Another extension named PgPointcloud is aimed at storing the point clouds data efficiently. After storing LiDAR points in a PostgreSQL database, PgPointcloud eases many problems and allows a good integration with other geospatial data like vector and raster into one common framework PostGIS [Ramsey et al., 2020].

PostGIS

is a spatial database extension for PostgreSQL, which supports handling for geographic objects in PostgreSQL thus allowing position queries to be run in SQL. PostGIS conform to the Open Geospatial Consortium (OGC)'s *Simple Features* for SQL Specification. The GIS objects supported by PostGIS are a superset of the *Simple Features* defined by OGC. PostGIS gives support to all the objects and functions that have been specified in OGC *Simple Features* for SQL specification and extends the standard with support for 3DZ, 3DM and 4D coordinates [OSGeo, 2019a]. Firstly, there are two standard ways defined to express the spatial objects: the Well-Known Text (WKT) and the WKB form. Both WKT and WKB include information about the type of the object and the coordinates which form the object [OSGeo, 2019a]. There are 7 types supported in PostGIS: POINT, MULTIPOINT, LINESTRING, MULTILINESTRING, POLYGON, MULTIPOLYGON and GEOMETRYCOLLECTION [OSGeo, 2019a]. The internal storage format of spatial objects are also required to include a SRID which is necessary when insertion spatial objects into the database. Secondly, the geography types provides native support for spatial features represented on geographic coordinates which is spherical coordinates with basis of a sphere. PostGIS allows GIS objects to be stored in the database and includes support for spatial indexes, and functions for analysis and processing of GIS objects.

PgPointcloud

is a PostgreSQL extension which enables storing point clouds data in PostgreSQL database and allows different kind of queries and analysis. The reason why it is difficult to handle LiDAR data is the necessity to cope with multiple dimensions of each point.

Each point contains a number of fields, and each field can be in any data type, with scaling and offset. PgPointcloud uses a schema document describing the contents of any specifc LiDAR point to cope with all this dimensions. The schema document format used by PostgreSQL PgPointcloud is as same as the PDAL library [Ramsey et al., 2019].

There are two point cloud objects defined in Pointcloud. The basic point clouds type is *PcPoint*. The structure of database storage makes storing billions of points as single record in a table an inefficient use of resources. A group of PcPoint are thus collected into a a PcPatch. Each patch should consist of points that are near together. Instead of a table of billions of single PcPoint records, a collection of LiDAR data can be represented in the database as a much smaller collection of PcPatch records. Usually the user will just create tables that stores PcPatch objects, and use PcPoint objects as intermediate objects for filtering, but it is still possible to create tables of both types [Ramsey et al., 2020]. The pointcloud_postgis extension adds functions that allow you to use PostgreSQL Pointcloud with PostGIS, converting PcPoint and PcPatch to Geometry and doing spatial filtering on point cloud data.

The PgPointcloud extension supports the persistent storage of 3D point clouds while effectively accessing all additional attributes of point records. PgPointcloud extends PostGIS with the new data types: PcPoint and PcPatch. Several hundreds or thousands of spatially rearby points are organized together, called patches. Each patch is stored in an individual table row and corresponds to a PcPatch object. PgPointcloud deals with the challenge of multiple dimensions by using an eXtensible Markup Language (XML) schema document, which describes the dimension structure of any specific point. Every point might contain up to dozens of attributes, and each of it can be in any data type, e.g. with scaling and offsets [Meyer and Brunn, 2019].

Oracle

Spatial and Graph

Oracle Spatial and Graph is a set of integrated functions, procedures, data types, and data models supporting spatial and graph analysis in Oracle. The spatial features support fast and efficientstorage, access and analysis of spatial data in the Oracle database. Spatial data represents the basic position feature of the target objects because those objects relate to the space where they reside. The spatial component of a spatial object is the geometric representation of its shape in its coordinate space which is called geometry. With Spatial, the geometric description of a spatial object is stored in a single row, in a single column with the data type SDO_GEOMETRY in a user-defined table.

SDO_PC_PKG Package

SDO_PC object type and *SDO_PC_BLK* object type give ability to the management of point clouds data by using *SDO_PC_PKG* Package. It defines several useful subprogram supporting the manipulation of point clouds data. The description of a point cloud is stored in a single row, in a single column of object type SDO_PC in a user-defined table. The function *CLIP_PC* performs the clip operation and returns points inside a query window or any other conditions by the parameters, an object of SDO_PC_BLK type is returned,

To store point clouds data, the users can use either an SDO_PC object or a flat table. An advantage of the flat table is its efficiency and dynamic characteristics, because updates to the point data do not require re-blocking. The general process for working with a point cloud is as follows, depending on the storage mode of point clouds data wan SDO_PC object or a flat table. To use point cloud data stored as an SDO_PC object: Initialize and create the point clouds, as needed for queries, clip the point clouds. To use point cloud data stored in a flat table: Create the table (or a view based on an appropriate table) for the point clouds data and populate the table with point data.

NoSQL Database

Currently databases are generally divided into two ways: relational databases and NoSQL databases. The former one use transparent tables in databases to store data, this way is suitable and efficient when the data is structured [Baralis et al., 2017] NoSQL databases are general distributed database [Guo and Onstein, 2020]. They are designed for structured data and scaling horizontally [Baralis et al., 2017]. Horizontal scalability is achieved by adding new commodity servers in cluster environment when the size of the data increases i.e. the number of request, which feature is useful when managing big data. NoSQL databases perform well in terms of large scale data storage and querying, as well as data processing and scalability. However, the main drawback is that currently NoSQL only have spatial functions which is much less than relational databases [Guo and Onstein, 2020]. They are thus not able to be integrated with other GIS data. Additionally, NoSQL databases has to give up a few quality control on data [Cura et al., 2017]. NoSQL databases can be categorized into: document DB, graph DB, column DB and key-value DB.

2.3.2 Storage model

Flat table

Flat table model is to store one point records in one row of database table. Usually there are a large number of tables stored in the database, and each table storing a point per row. Such a database can thus quickly reach billions of rows. Storing these rows is a problem because DBMS may have a overhead per row that could not be negligible. Moreover, indexing such a large number of row is not efficient and takes a lot of space, and the support for compression is limited [Cura et al., 2017].

Blocks

In order to explore the efficient storage mode for DBMS solution, the grouping is necessary before storing, like PostgreSQL/ PC_PATCH and Oracle/ SDO_PC_BLK. It remains a difficult problem that managing a massive amount of individual points. A solution with scalability needs to concentrate on data storage and retrieving, and leave the additional aspects of the management issues like metadata, processing and integration, . Another recent approach is being explored in PgPointcloud and other commercial RDBMS. This is block storage model groups the points that are close together into a chunk i.e. patch, and each group of points into one row. The basic idea is to gather points together instead of individual points in a databases. Creating this abstraction layer over points allows to retain all the advantages of a database, but keeps the number of rows low to avoid the scalability issues like index and compression. The way

points are grouped must be compatible with the intended usages because before the single point can be accessed, its entire group has to be accessed first [Cura et al., 2017].

2.3.3 Pros and Cons

General advantages of a DB solution are data consistency, security, reduced storage space and multi user access [Meyer and Brunn, 2019]. Another benefit is that point clouds can be combined with all types of geographic data and non-graphic data, such as raster data, vector data and administrative data, within GeoDBMS. The possiblity to define relations in the RDBMS offers a simple way to create robust models and handle metadata [Cura et al., 2017].

In the context of DBMS approach, point clouds data are imported, indexed and stored as a dedicated defined point clouds geometry or geographic type. Database provides fast access to data, but it costs time to import the data, which has to be performed if new data is added [Rieg et al., 2014]. It is usually necessary to reorganize the data to match the internal table structure and allow batch insertion i.e. to load multiple rows of data at once. generally, time consumed by query and processing is heavily depend on the size of table in the database, the dimension of point groups, the point density of the input data, the indexing strategy and the hardware and its configuration [Rieg et al., 2014].

2.4 SQL Management of External Data

This section will give detailed description of the standard of SQL/MED and is referenced to the paper "SQL and Management of External Data" by [Melton et al., 2001].

In 2000, a new standard of SQL named MED is introduced to deal with how a database management system can integrate data stored outside the database. SQL/MED has two components: Foreign Data Wrapper and DATALINK. There are two aspects of the problem happen when accessing external data that are addressed by SQL/MED. The first aspect enables to use the SQL interface to access foreign data residing outside DBMS. This part of standard is called 'wrapper interface'. The other aspect of the problem with foreign data is the management problem. This is handled by introducing a new data type called DATALINK, which intends to store URLs in SQL table columns.

2.4.1 Accessing

Overview of SQL/MED

SQL is based on the object-relational model, so the foreign data from external sources have to be displayed as relational tables when it is going to be adapt to the environment of SQL server seamlessly. SQL/MED introduces the concept of *foreign tables* representing foreign data stored externally to the SQL server. In many cases, external data sources often hold multiple data collections, and they can be accessed through the same network connection, so SQL/MED introduces the notion of a *foreign server* which allows access to a set of foreign tables. It is also the common case that multiple data sources share one interface together. Therefore, it is required to use one single code module to connect to all of these data sources, each of which is handled by a foreign server. This common code module is accomplished by a *foreign data wrapper* in SQL/MED .

Figure 2.2a, figure 2.2b illustrates the main components and figure 2.1 shows their internal relationships defined and used by SQL/MED. The SQL server and the foreign data wrapper communicate through the SQL/MED API. But there is no requirement that they have to be implemented in one single process context, in the true cases they can be distributed in separate computers and be connected by a network. The interaction between the foreign server and the foreign data wrapper is defined individually by the authorized users of these components and not by means of SQL/MED.

Notions of SQL/MED

Foreign tables are aimed to provide support for a transparent view on foreign data, which is not stored in but managed by the local SQL server. The transparent view means the users can access the foreign data and retrieve a foreign table by a `SELECT` statement as thought it were a normal table or view, while the users don not need to know the fact that the data is not stored and actually managed by the local SQL server. Foreign data is presented to the users as regular SQL data, although the source of the data and the storage exists in file system, Hypertext Markup Language (HTML), or other formats. Before the data in a foreign table can be exposed by SQL/MED features. the SQL server need to firstly be informed of the existence of the foreign table, using `CREATE FOREIGN TABLE` statement:

By running this statement, a new schema object, a foreign table is created in a schema that belongs to the SQL server. In the statement, the foreign table identifies a foreign server that manages the data to be represented as though it were a local table. Therefore, the foreign server needs to be created as the reference of the foreign tables.

Unlike other SQL virtual objects, e.g. foreign table, foreign server is not represented in a SQL schema that belong to the SQL server, rather than at a higher level: the catalog [Melton et al., 2001]. Thus, the foreign data wrapper needs to be named, which manages the foreign server. Same as foreign server, the representation of foreign data wrapper is in the SQL server catalog but not in the schema.

These statement should be executed in the sequence:

CREATE FOREIGN DATA WRAPPER ...
CREATE SERVER ...
CREATE FOREIGN TABLE ...

SQL also provides the convenience for the cases that when the foreign server has been created, the one or more foreign tables can be created based on information available from the foreign server. In such cases, foreign server presumed the existence of this schema that includes the tables. And also the importation of table definitions from the foreign server for every tables included in this foreign schema, while certain tables can be limited or omitted.

Additionally, there are also many cases, SQL give the ability to map the user identifiers of the SQL server to the foreign servers.

It is necessary to use configuration information to characterize the foreign server which is going to be accessed by means of foreign data wrapper, this information of configuration can be hose name and port number, which can make it different from other foreign servers through the same wrapper. In this context, this information can vary from wrapper to wrapper, therefore SQL/MED does not define a fixed set of attributes of configuration information. Instead the notion *generic options* which is represented as *'attribute: value'* pairs and used by wrapper is introduced for the configuration purpose. The values of *generic options* are cached in the catalogs of the SQL server ,and SQL server does not interpret them, rather than makes their values available to the wrapper upon request. The *generic options* can be related to each of the foreign data process components defined by SQL/MED, e.g. foreign servers, foreign tables, and their columns. Therefore, for example, a foreign data wrapper that display the foreign data stored in the file system as foreign tables, can use an *option* with each corresponding table that specifies the delimit in the target file.

SQL/MED defines an API, a set of functions, through which the SQL server and foreign data wrapper perform their cooperation orders. *Foreign data wrapper interface SQL **server** routines* is the SQL/MED functions that the SQL server has to provide, which refers to all the functions that must be supported by a corresponding SQL server. While the the SQL/MED functions that the corresponding foreign data wrapper has to provide is called *foreign data wrapper interface SQL **wrapper** routines* [Melton et al., 2001].

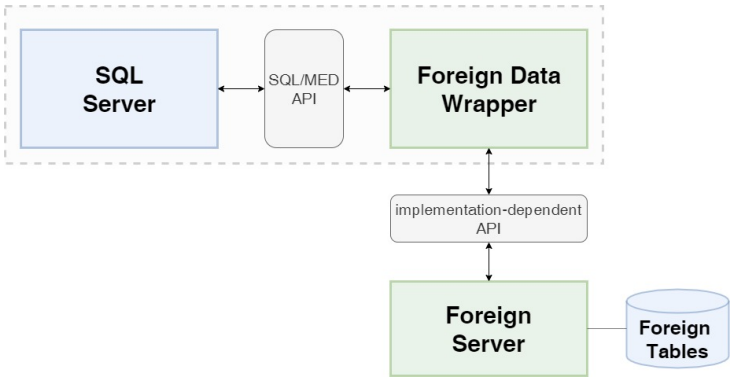


Figure 2.1: MED components

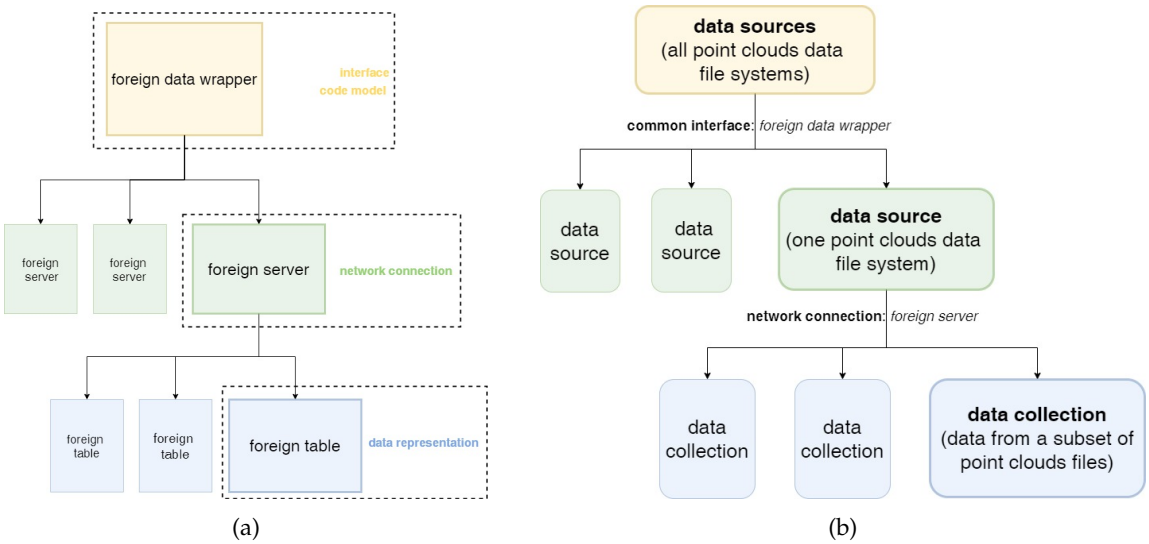


Figure 2.2: MED access.

2.4.2 Querying

Communication and collaboration between a SQL server and a foreign data wrapper can take place in two modes: *decomposition mode* or *pass-through mode* [Melton et al., 2001]. In pass-through mode, the SQL server transfers the string of query originally to the foreign data wrapper. In decomposition mode, the SQL server breaks a query into fragments, each of which is executed partly by the foreign server.

The procedure that =the SQL server and the foreign data wrapper communicate with each other can be divided into two phases:

- A *query planning* phase: The foreign data wrapper and the SQL server together produce an execution plan for the query *fragment*.
- A *query execution* phase: The execution plan with agreement is executed and foreign data is returned to the SQL server.

During both the query planning and execution phases, information should be transmitted between the foreign data wrapper and the SQL server.

Example

To take an example to illustrate the principle above, point clouds data is the data source on file system in ASCII format, users utilizes a foreign data wrapper to access the data source via a certain foreign server, using SQL statement on one or more foreign tables. The foreign data is the *TXT* file, each line of which contains one point records, each fields separated by delimiter comma. Assuming that an appropriate foreign data wrapper has been implemented for example by Multicorn the corresponding foreign server has been declared. Then using the following Data Definition Language (DDL) to declare a foreign table representing the foreign data of point clouds.

```
CREATE FOREIGN TABLE pc_txt_table (
x DOUBLE PRECISION,
y DOUBLE PRECISION,
z DOUBLE PRECISION,
classification INTEGER, ...)
SERVER pc_txt_server
OPTIONS (
filename '.../pc_files_ystem/pc_txt_file',
delimiter ',');
```

The implemented foreign data wrapper is used to access any file of this general type as ASCII format, therefore in the definition of each foreign table, the appropriate filename and delimiter have to be specified by *generic options*.

After the foreign table has been created, the foreign data can be accessed via the SQL statement on the foreign table. For example, a user want to know how many points are ground point in this 'pc_txt_file' using following SQL query:

```
SELECT COUNT (*)
FROM pc_txt_table
WHERE classification = 2;
```

The SQL server first parses and verify this query to make sure that it is syntactically and semantically correct and that the user is authorized. Then, the SQL server examines the `FROM` clause and recognizes that it includes a reference to a foreign table. Thus, the SQL server builds a *connection* to pc_txt_server and formulates a *request* of query fragment equivalent to the SQL statement `SELECT * FROM pc_txt_table`. The query excludes the predicate from the `WHERE` clause or aggregation function in the `SELECT` list.

3 Methodology

This research is aimed to answer the question that to what extent we can use point clouds data directly in PostgreSQL by means of foreign data wrapper. Foreign data wrapper is proposed as a combined method for handling point cloud data between file-based method and database management system method. The steps of using point clouds data includes data storage, data preparation, data access and data querying. This chapter will introduce the theory behind to answer the research questions including the methodology used to achieve this goal and design of a Point Clouds Data Management System to put into practice.

Section 3.1 describes the principle of foreign data wrapper along with how does it connect point clouds file system to PostgreSQL database. Section 3.2 introduces the data access schema by this means, which includes how the point clouds data is stored and prepared, and how does data content of point clouds be obtained and fetched, represented on PostgreSQL database for further operations. Section 3.3 is going to introduce the principles of the database features that can be in use on point clouds files, including querying, aggregate functions, data manipulation and type conversion; what is the implementation schema of these capabilities inside the wrapper will be demonstrated; and also it summarizes the queries, mainly spatial selection, that can be asked on point clouds data. Section 3.4 depicts a system architecture of Point Clouds Management System to package the procedures of using foreign data wrapper for exposing multiple point clouds data on file system. This system architecture utilize one single foreign data wrapper together with supporting softwares, libraries and database as well as extensions.

3.1 Foreign Data Wrapper

3.1.1 Principle of FDW

PostgreSQL has a specification named SQL/MED (SQL Management of External Data) as introduced in Section 2.4, which a standardized way to handle access to remote objects from SQL databases [PostgreSQL wiki, 2020]. Foreign data wrapper is part of support for this standard, implemented by PostgreSQL. A foreign data wrapper is a library that can communicate with an external data source, but the details of connection to the external data source and obtaining data from it are hidden [The PostgreSQL Global Development Group, 2019]. There are a variety of foreign data wrappers which make different remote data available in PostgreSQL, with these data remotely storing in various forms like other SQL or NoSQL databases, files and web form [PostgreSQL wiki, 2020]. Foreign data wrapper acts as a proxy responsible for fetching data from external data source and also transmitting data to the external data source. All operations on a foreign table are handled through its foreign data wrapper, which consists of a set of functions that the core SQL server calls [The PostgreSQL Global Development Group, 2019].

The foreign data wrapper corresponding to a foreign table is instantiated based on each process, and it take place when the first query is run upon it. Usually, SQL server process are based on a each connection. During each server process, the instance is cached, which means that if references need to keep to data sources such as connection,the user should initialize them and cache them as instance attributes [Kozea, 2014].

3.1.2 Using FDW

Foreign data wrapper enables the access to foreign data, which is the data residing outside PostgreSQL using regular SQL queries, in this research the point clouds data is treated as foreign data. In general,

there are two steps to construct the access to foreign data after the foreign data wrapper is complied and loaded as the PostgreSQL extension. Firstly, a *foreign server* object need to be created, which defines the way to connect to a particular remote data source depending on the set of options used by its supporting foreign data wrapper. The connection information used by a foreign data wrapper in order to access an external data source is encapsulated in a foreign server.

```
CREATE SERVER server_name
FOREIGN DATA WRAPPER fdw_name
[ OPTIONS ( option_name 'value' [, ...] ) ] ;
```

Then, we need to create one or more *foreign tables*, which define the structure of external data [The PostgreSQL Global Development Group, 2019]. A foreign table can be used in queries just like a regular table, but a foreign table has no storage in the PostgreSQL server.

```
CREATE FOREIGN TABLE table_name ( [
{ column_name data_type }  [, ...] ] )
SERVER server_name
[ OPTIONS ( option_name 'value' [, ...] ) ] ;
```

Besides, it is also possible to import table definitions from a foreign server. All tables and views existing in a particular schema on the foreign server are imported [The PostgreSQL Global Development Group, 2019]. In this way, it creates foreign tables that represent tables existing on a foreign server.

```
IMPORT FOREIGN SCHEMA remote_schema
FROM SERVER server_name
INTO local_schema
[ OPTIONS ( option_name 'value' [, ...] ) ]
```

In addition, the user can define a mapping to a user to a foreign server. A user mapping can encapsulate the connection information, with which together the information encapsulated by a foreign server, a foreign data wrapper uses to access the external data sources [The PostgreSQL Global Development Group, 2019]. The author of a foreign server can define the user mapping for this server for any users.

```
CREATE USER MAPPING [ IF NOT EXIST ]
FOR { user_name | USER | CURRENT_USER | PUBLIC }
SERVER server_name
[ OPTIONS ( option_name 'value' [, ...] ) ]
```

3.1.3 Writing FDW

In this research we will use Multicorn, a Foreign Data Wrapper Library for PostgreSQL, to develop our own foreign data wrappers. The implementation purpose is that a foreign table can be used in SQL queries just like a local table, but foreign table takes no storage in the PostgreSQL server. Whenever it

is used, PostgreSQL server asks the foreign data wrapper via foreign server to carry out the operations. Foreign data wrapper is responsible for retrieving data for foreign tables.

There are some foreign data wrapper available as `contrib` module like *file_fdw* which is used to access data files in the server's file system and to execute program on the server and read their output and *postgres_fdw* which can be used to access data stored in external PostgreSQL servers. Other kinds of foreign data wrappers can be found as the third party products like which are not officially maintained by PostgreSQL Global Development Group. Users can also develop the use-defined foreign data wrapper for their specific usage. With various procedural languages, there are several foreign data wrapper as PostgreSQL extension. As specified in the SQL/MED standard, the foreign data wrapper need to interacts with the SQL server to fulfil its task of handling the content of a foreign table. For this reason, a foreign data wrapper need to be implemented in C and linked to the relevant PostgreSQL libraries in order to communicate with the SQL internals. [Roijackers et al., 2012].

Multicorn is a PostgreSQL 9.1+ extension that allows users to develop their own foreign data wrappers in Python programming language. The usage of Multicorn foreign data wrapper has no difference from any other wrapper after create the extension of multicorn in the target database, as introduced above. Multicorn gives abilities to users to access any data source in the PostgreSQL database, and then the users can leverage the full power of SQL to query the data source , and base Multicorn module contains all the python code needed by the Multicorn C extension to PostgreSQL [Kozea, 2014].

A Python script is used by Multicorn to parse the external data source and use the output of script as the content of foreign table. This Python script can yield and build its output using any capabilities like Python objects, functions and modules provided in a Python programming environment. There are several tools dedicated for point clouds data including libraries and module can be used by Python language. By using Multicorn, the users can write code in Python language to access data source outside PostgreSQL and give output of resulting records to Multicorn which forwards the content to PostgreSQL. Basically, Multicorn implements the C functions needed to be able to use a Python script which is responsible for the actual external data retrieving. It is implemented as a regular foreign data wrapper of PostgreSQL and available to be installed as an extension. Which Python script that Multicorn will use as foreign data wrapper is specified as an option when creating the foreign server with fdw name as multicorn [Roijackers et al., 2012].

Multicorn provides a simple interface for writing foreign data wrappers, which is named *multicorn.ForeignDataWrapper*. This *multicorn.ForeignDataWrapper* is the base class for all multicorn foreign data wrappers which means the implementation of the foreign data wrappers should inherit from this class . Therefore, implementing a Multicorn foreign data wrapper is as simple as implementing the *ForeignDataWrapper* class in Python, including inheriting from *ForeignDataWrapper* by *__init__()* method and implementing the *execute()* method.

One required method of this class is *__init__()* for initialization. The foreign data wrapper is initialized on the first query executed on the PostgreSQL side which commands a retrieving of data source. The parameters of *__init__()* are *fdw_options* and *fdw_columns*. The parameter of *__init__()* are:

- **fdw_options** (dict): is the foreign data wrapper options as the dictionary type, which is a dictionary that maps keys from the SQL `CREATE FOREIGN TABLE` statements `OPTIONS`. They are selected and defined by implementors what should be within these options, and what to do with them. For example, in this system wrapper, the file path of the file system is one of the required options.
- **fdw_columns** (dict): is the foreign data wrapper columns which is a dictionary that maps the columns names to their ColumnDefinition class

Another required method is *execute()* which execute a query in the foreign data wrappers and it is called at the first iteration. *execute()* is where the actual remote query execution occurs. It should return a python objects that are iterable and can be converted to PostgreSQL. These kind of iterable objects can be:

- **sequences** containing exactly same number of columns in the corresponding tables.
- **dictionaries** mapping column names to their values

The parameter of *execute()* are:

- **quals** (list): A list of Qual instances, containing the basic `WHERE` clauses in the SQL query.
- **columns** (list): A list of columns that PostgreSQL is going to need. You should return at least those columns when returning a dictionary. If returning a sequence, every column from the table should be in the sequence as structured.
- **sortkeys** (list): A list of SortKey that the wrapper announced it can enforce.

The implementation of *execute()* should necessarily:

- Initialize or reuse the connection to the remote system
- Transform the quals and columns arguments to a representation suitable for the remote system
- Fetch the data according to this query
- Return it to the C extension

These two methods are required API allowing user-defined foreign data wrapper can be used for read-only queries. There are other methods for *ForeignDataWrapper* class supporting more complex API like write capability and transnational capability, and other important classes like *quals* that could be used by *ForeignDataWrapper* class implementatin. Multicorn also provides methods *get_path_keys* and *get_rel_size* for the Python FDW implementor to affect the planner of PostgreSQL.

The detailed principles of Multicorn APIs used to implement the operations that foreign data wrappers can support will be described in details in the following sections in the specific process.

3.2 Data access

The data access schema of foreign data wrapper for point clouds includes where the LiDAR data is stored and how the data is organized using spatial data access methods including indexing and clustering, as well as how the data is displayed on PostgreSQL after being obtained and fetched by foreign data wrapper.

3.2.1 Data storage

The aim of this research is using LiDAR data directly in PostgreSQL stored in file system with the help of foreign data wrapper. Foreign data wrapper provides a bridge that point clouds data is stored in file system while is used by PostgreSQL functionalities.

The foreign data wrapper allows the querying and manipulation operations to be executed upon any foreign data regardless of its source after data source has been represented on the foreign tables. Therefore, by this means, point clouds data is still stored in original mode i.e. file formats. The research objects are point clouds data in ASCII text, LAS binary or LAZ compressed formats.

The idea of using file system storage in LiDAR data management comes from the observation that large collections of LiDAR data are typically delivered as large collection of files, rather than single files of terabyte size [Boehm, 2014]. The test datasets of this research AHN is a good example, not storing the whole Netherlands point clouds data in the single file, instead of using several files to manage it. In this research, the purpose of Point Clouds Data Management System, is in which all these point clouds files stored in different formats are collected in one file system. The access port of the foreign data wrapper is at file collection level, and then the query will be directed to each appropriate file at the file level inside the wrapper.

3.2.2 Data organization

This subsection will introduce the spatial access methods based on data organization to improve efficiency of the spatial querying. Data access methods have two aspects, one is indexing, knowing the computer, disk, memory location quickly when searching for certain points; another one clustering which is storing objects close together, which are also often selected together because of spatial proximity.

Sorting

Sorting is to store the points close in the storage medium which are close in space, since it helps locating the related data fast. It can be relatively easy to organize and sort points when using flat storage mode [Van Oosterom et al., 2015]. The aim of clustering is to make the retrieving time as less as possible by storing close objects also close in computer memory or disk. In spatial database systems, the concept of clustering is used when spatially nearby objects, which are often asked together by queries, are stored jointly in storage [Wijga-Hoefsloot, 2012]. The sorting can be done according to Hilbert code or Morton code, in which the location of point records are in the Space Filling Curve (SFC). Sorting the points by SFC can exploit the coherence of the data origin.

LAStools provides one powerful tool to sort the point clouds file based on the above principle. *lassort* can do the sorting of the points in a LAS/LAZ/ASCII file into z-order (Morton code) arranged cells of a square quad tree and saves them into LAS or LAZ format [rapidlasso, 2019].

Indexing

Before fetching the content of point clouds data to the foreign table, it is necessary to build index. Indexing is knowing storage location fast when searching for certain objects. Spatial indexing provides efficient access to subset of points that the spatial query can ask for, like query AoI (Area of Interest), getting data from neighbor area and sample elevation.

A spatial index, like any other index, provides a mechanism to limit searches, but in the case of spatial data, the mechanism is based on spatial criteria such as distance, intersection and containment [Wijga-Hoefsloot, 2012]. Point clouds data stores large amounts of elevations samples, often resulting in Terabytes of data. The generated LiDAR points are usually stored and distributed in the LAS and LAZ format files. However, managing a folder of LAS or LAZ files is not a easy work, when querying a simple AoI, all files are required to be opened and loading all those whose extent overlaps the queried AoI [Isenburg, 2012b].

One approach is to copy the data into a database such as Oracle or PostgreSQL. Another alternative that works directly on the original LAS or LAZ files proposed by LAStools [Isenburg, 2012b]. When indexing on the DBMS solution, indexes can make it possible to use a spatial database for large data sets. If there is no indexing, any search for a certain object would need a scan of every record in the sequence within the database. Indexing improves searching speed by using a search tree to organize the data into and can be quickly traversed to find a specific record. By default there are three kinds of indexes supported by PostgreSQL: B-Tree indexes, SP-GiST and GiST indexes [OSGeo, 2019a].

- B-Trees are used for data which can be sorted along one axis; for example, numbers, letters, dates. Spatial data can be sorted along a SFC curves, Z-order curve or Hilbert curve. However this representation does not allow speeding up common operations.

- Generalized Search Trees (GiST) indexes decompose data into ”things to one side”, ”things which overlap”, ”things which are inside” and can be used on a wide range of data types, including GIS data. PostGIS uses an R-Tree index implemented on top of GiST to index GIS data

In this research, we will use the *lasindex* of LAStools to index the data before FDW do the retrieve inside the foreign data wrappers of LAS and LAZ files. The reason why build indexing before fetching data on the file basis rather than after fetching into foreign table on the database basis is that the index cannot be built on PostgreSQL foreign table. *lasindex* creates a LAX file for a LAS or LAZ file that has spatial indexing information. When this LAX file is present it will be used to speed up access to the relevant parts of the LAS or LAZ file [Isenburg, 2012a].

3.2.3 Data display

By means of foreign data wrapper, the foreign data from the remote data source is represented in the foreign tables, which is the base for further operations of foreign data on the PostgreSQL. The display of foreign data is by declaring the foreign table in the schema using the following statement, which can be queried as the local table. Therefore, each field including x and y coordinate, height and attributes of point clouds data is displayed as a table column having the corresponding column name and data type.

```
CREATE FOREIGN TABLE table_name ( [
{ column_name data_type } [, ...] ] )
SERVER server_name
[ OPTIONS ( option_name 'value' [, ...] ) ] ;
```

PostgreSQL has several built-in data types that can be used when defining the foreign table for different attributes of point clouds data. PostgreSQL supports number types which consist of integers, floating point numbers, and selectable precision decimals. The integer types store whole numbers, i.e. numbers without decimal components of various range, such as the value of classification and RGB values. Arbitrary precision numbers applies the `numeric` type is used to store numbers with a very large number of digits, like the quantities requires exactness. The precision of a `numeric` is the total count of significant digits in the whole number, while the scale of `numeric` is the count of decimal digits in the fractional part. The types `decimal` and `numeric` are equivalent. Floating point types including `real` and `double precision` are inexact, variable-precision numeric types. Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and retrieving a value might show slight discrepancies. The difference between the `numeric` type and `real`, `double precision` is that When rounding values, the `numeric` type rounds ties away from zero, while on most machines the `real` and `double precision` types round ties to the nearest even number. The serial types e.g. `smallserial`, `serial` and `bigserial` are not true types, but merely a notational convenience for creating unique identifier columns.

For the position fields, i.e. 3D coordinates, according to [Van Oosterom et al., 2015] benchmark which tested different data types including `double precision`, `numeric` and `integer`. The integer does not show the storage saving, due to the huge row overhead added by PostgreSQL. Moreover, the time cost by scaling the coordinates during query execution slows down the querying times. The numeric type has a very poor performance in algebraic operations that significantly degraded the query response times. Therefore, in this research, we store X, Y and Z as double precision i.e. without compression.

Point clouds data are a set of 3-dimensional points. It is an advantage to represent the point in spatial columns. PostgreSQL supports the built-in `geometric` data types to represent two-dimensional spatial objects and PostGIS provides `geographic` data types for spatial features represented on geographic coordinates. The basis of geometry and geography is different, geometric type basis is a plane while geographic is a sphere. Therefore the calculation is different as well like areas, distance and intersection [OSGeo, 2019a]. Standard geometry type data can be autocast to geography if it is of SRID 4326 which refer to that the geographic reference system is WGS84. It should be noticed that our test dataset AHN3 applies the of 28992.

3.3 Data functionality

After the foreign data on file system has been represented on foreign table, it can be operated using database facilities on the PostgreSQL side, including query functionality, aggregate function, data manipulation and type conversion. The interaction between SQL server and foreign data wrapper is defined in SQL/MED standard as introduced in section 2.4, while the communication between foreign server and foreign data wrapper is implementation dependent. Therefore, the implementation of foreign data wrapper should participate in the process of the functionalities.

3.3.1 Query

SQL query

Querying a table is to retrieve data from a table. PostgreSQL executes the command by SQL language `SELECT` statement used to specify queries. Foreign data wrapper is responsible for the retrial of data for a foreign table. The general syntax of `SELECT` statement is:

```
[ WITH with_queries ]
SELECT select_list FROM table_expression
[ sort_specification ]
```

SQL command consists two basic parts: *table expression* and *select list*. According to the select lists, table expression can provide all columns or a subset of available columns, and make calculations using columns.

A *table expression* computes a table. The table expression contains a `FROM` clause that is optionally followed by `WHERE`, `GROUP BY`, and `HAVING` clauses. The optional `WHERE`, `GROUP BY`, and `HAVING` clauses specify a pipeline of subsequent transformed on the table derived in the `FROM` clause, i.e. computed by table expression.

The `FROM` clause derives a table from one or more tables given in a table reference list, by using foreign data wrapper, the foreign tables can be the reference table.

```
FROM table_reference [, table_reference [, ...]]
```

A table reference can be a table name, or derived table like subquery, a `JOIN` construct, or even complex combinations of these. A joined table is a table derived from two other (real or derived) tables according to the rules of the particular join type. Inner, outer, and cross-joins are available.A temporary name can be given to tables and complex table references to be used for references to the derived table in the rest of the query, which is called a table alias. Subqueries specifying a derived table must be enclosed in parentheses and must be assigned a table alias name. In our cases, one single foreign table of point clouds data, the foreign table of point clouds data can join the local table of other GIS data for data combination, or a subquery resulting potential points can join with a local table storing query polygons, can constructs a table reference, thus deriving a table for querying. The result of the `FROM` list is an intermediate virtual table that can then be subject to transformations by the `WHERE`, `GROUP BY`, and `HAVING` clauses and is finally the result of the overall table expression.

The `WHERE` clause comes along with a search condition which is any value expression that returns a value of type `boolean`.

```
WHERE search_condition
```

After the processing of the FROM clause is done, each row of the derived virtual table is checked against the search conditions. If results of the condition is true, the row is kept in the output table, otherwise it is discard. After passing the WHERE filter, the derived input table might be subject to grouping, using the GROUP BY clause, and elimination of group rows using the HAVING clause.

The *table expression* in the SELECT command constructs an intermediate virtual table by possibly combining tables, views, eliminating rows, grouping, etc. This table is finally passed on to processing by the *select list*. The select list determines which columns of the intermediate table are actually output. The simplest kind of select list is * which emits all columns that the table expression produces. Otherwise, a select list is a comma-separated list of value expressions.

Spatial query

The research object is point clouds data which is spatial data, other than attribute selection, the query can be executed is spatial selection. Spatial query is to select the points within a query region, which can be any region including simple rectangle and circle, or irregular polygons. For this purpose, the PostGIS functions will be used for Point-in-Polygon selection.

The following functions supported bu PostGIS can be used as condition depending on the spatial topological relationship.

- *ST_Contains* Returns true if and only if no points of B lie in the exterior of A, and at least one point of the interior of B lies in the interior of A. Geometry A contains Geometry B if and only if no points of B lie in the exterior of A, and at least one point of the interior of B lies in the interior of A. This function can be used as one of the search condition in WHERE clause for Point-in-Polygon selection, with A being query regions like POLYGON and B being the POINT.

```
boolean ST_Contains( geometry geomA, geometry geomB)
```

- *ST_Intersects* Returns TRUE if the Geography spatially intersect in 2D which means sharing any portion of space and FALSE if they don't which means they are Disjoint.

```
boolean ST_Intersects( geometry geomA , geometry geomB );
```

- *ST_Within* Returns true if the geometry A is completely inside geometry B. This function can be used as one of the search condition in WHERE clause for Point-in-Polygon selection, with B being query regions like POLYGON and A being the POINT.

```
boolean ST_Within( geometry A, geometry B);
```

The following functions can be used to measuring the attribute of searching condition i.e. query region.

- *ST_Extent* is an aggregate function that returns the bounding box that bounds rows of geometries.This function can be used to get the extent i.e. bounding box information of the query polygon, with extent information a pre-selection can be queried as box selection.

```
box2d ST_Extent( geometry set geomfield);
```

- *ST_Area* returns the area of a polygonal geometry, this can be used to get the area of query polygon for computation of efficiency related to ratio of area of query region and selected file extent.

```
float ST_Area( geometry geom);
```

Possible queries

With the support of PostgreSQL and PostGIS, the possible queries on the foreign data representing the foreign data from the point clouds file system, can have a wide range. The system should be able to support points selection, and the search condition in `WHERE` can be based on position and also additional attributes, in addition the aggregate function can be added in select lists.

- select nearest neighbor of one location with a buffer
- select all the points within rectangle of different sizes
- select all the points within a circle of different sizes
- select all the points within a polygon of different sizes

It is possible to make selection based on attribute field conditons

- select the ground/ water/ building points insides a region
- select points based on intensity, RGB values insides a region

It is also possible to make aggregation function

- select the highest point of a region
- select the highest, lowest and average elevation value of points in a region
- select the total point density and local point density of a region

FDW routine

When a SQL command is executed in PostgreSQL prompt, the supporting foreign data wrapper will do the retrieving from the external data source which is point clouds file, and then fetch the asked content to the foreign table. The mapping of foreign data wrapper does not copy any data but redirects any query to the remote database server and table [Stanisavljevic, 2019]. From a user perspective, there is no difference between a foreign table and any other relation in database. There is a schema holding data content and it can be used in select queries without any restrictions [Roijackers et al., 2012]. A foreign data wrapper is responsible for the retrieval of data for a foreign table. The callback functions are used for planning, explaining and retrieving data for a asked query. Retrieving data records to the table rows is implemented using an iteration function which returns the next record according to the schema of the used foreign table [Roijackers et al., 2012].

The *execute()* method of *multicorn.ForeignDataWrapper* can execute a query in the foreign data wrapper. This method is called at the first iteration. This is where the actual remote query execution takes place. It should return a python objects that are iterable and can be converted to PostgreSQL. These kind of iterable objects can be: *sequences* containing exactly same number of columns in the corresponding tables; *dictionaries* mapping column names to their values. The parameter of *execute()* are: *quals*, a list of Qual instances, containing the basic `WHERE` clauses in the SQL query. *columns*, a list of columns that PostgreSQL is going to need. You should return at least those columns when returning a dictionary. If returning a sequence, every column from the table should be in the sequence as structured. *sortkeys*, a list of SortKey that the wrapper announced it can enforce.

multicorn.ForeignDataWrapper class provides an API for improving the retrieve efficiency, which is applying the *Qual* class. A *Qual* describes a PostgreSQL qualifier, which is defined as an expression of the type: *col_name operator value*. The attributes of *Qual* are:

- field_name (string): The name of the column as defined in the foreign table.
- operator (string or tuple): The name of the operator.
- value (object): The constant value on the right side

The initialization method for the *Qual* class happens when WHERE clause exists in the SELECT statement. A *Qual* object is constructed and referenced to the one of search conditions in the WHERE clause, after extracting the field name, operator and values from WHERE clause and assigning to the attributes *Qual.field_name*, *Qual.operator* and *Qual.value* respectively. Whenever there is one search condition in the WHERE clause, a *Qual* instance is built, therefore, if there are several union search conditions, a list of *Qual* instances is created. This constructed list of *Qual* instances is named *quals* that will be passed as the parameter of *execute()* method when the query is executed.

Then in the *execute()* method of *multicorn.ForeignDataWrapper*, the information in *Qual* can be used as a filtering when *execute()* method retrieving the data content from remote data source. By using *quals*, the wrapper will conditionally fetching the data content in the external data source, otherwise, all the content in the foreign data will be fetched to the PostgreSQL foreign table.

3.3.2 Data manipulation

Foreign data wrapper also implements the functionality of update, delete and insert data, which is important support for system architecture. Data manipulate in PostgreSQL includes insert, update, and delete table data. When a table like the metadata table of the file system is created, it contain no data. The first thing to do is to insert data. Data is conceptually inserted one row at a time. In addition, the point records in the point clouds file are outside the bounding box given in the header information, these outside points may influence the accuracy spatial query, thus they could be deleted.

SQL update

To create a new row, use the INSERT command which requires the table name and column values, listing column names explicitly or not. It is also possible to insert multiple rows in a single command.

```
INSERT INTO table_name
VALUES (col1_value, col2_value, col3_value, ...);
```

We can modify the data that is already exists, updating individual rows, all the rows in table, or a subset of all rows. To update existing rows, use the UPDATE command, which requires information including the name of table and column to update, the new value of the column and which row to update.

```
UPDATE table_name
SET col1_name = new_value
WHERE search_condition;
```

Besides adding and changing data. deleting data is also supported. If the data is no longer needed, we can delete it by DELETE command to remove rows, with providing the information of table name and which rows to delete.

```
DELETE FROM table_name
WHERE search_condition;
```

FDW routine

Multicorn also provides API to realize the write capabilities that enable the data manipulation on the PostgreSQL. The following property have to be implemented *ForeignDataWrapper.rowid_column()*, this method returns a column name which will act as a *rowid* column, for delete/update operations. It can be understood as a primary key in PostgreSQL standard. a *rowid* column has the similar role as primary key indicating that a column or group of columns can be used as a unique identifier for rows in the table. Therefore, the values of *rowid* column are required to be both unique and not null. *rowid* column can be either an existing column name, or a made-up one. This column name should be subsequently present in every returned result set. If this *ForeignDataWrapper.rowid_column()* has not been implemented, but one execute the `UPDATE, DELETE, INSERT` command on the foreign table, an error of NotImplementedError would be raised reporting "This FDW does not support the writable API", these logging and error reporting functions have been defined already by Multicorn extension.

In addition, each Data Manipulation Language (DML) operations have to be implemented for SQL command to fit while executing the command and wrapper is responsible for the **sml!** (**sml!**). *insert()* method insert a tuple defined by *values* into the foreign table. This method need one parameter named *values* which is a dictionary mapping column names to column values, and returns a dictionary containing the new values. These values can differ from the *values* argument if any one of them was changed or inserted by the foreign side. For example, if a key is auto generated.

update() method Update a tuple containing *oldvalues* to the *newvalues*. Two parameters are required: *oldvalues* is a dictionary mapping from column to previously known values for the tuple. *newvalues* is a dictionary mapping from column names to new values for the tuple. While this method is able to a dictionary containing the new values.

delete() method delete a tuple identified by *oldvalues*, thus the parameter of *oldvalues* is needed which is a dictionary mapping from column names to previously known values for the tuple. And this *delete()* method returns None.

3.3.3 Type conversion

The 3D coordinates of point records are represented in numeric data types separately, but before spatial queries like Point-in-Polygon is executed, the x, y, z in numeric type need to be cast to geography or geometry data type for spatial selection according to spatial relationship.

PostGIS provides a function *ST_MakePoint* to create 2D, 3D Point geometry. For example, in the test to create 2D geometry Point for AHN3 data, the statement need to be run:

```
geometry ST_MakePoint(float x, float y);


 ST_SetSRID(ST_MakePoint(x, y),4326)
```

3.4 System Architecture

This thesis is aimed to answer the main research question that to what what extent we can use point clouds data in the file system directly on PostgreSQL by means of foreign data wrapper. Because one single file system can store different types and sizes point clouds data, while in the reality usage multiple point clouds files derived from different sources are mixed. It is necessary to create a file system for LiDAR data collection and in the meantime the foreign data wrappers can recognize relevant files and handle the relevant LiDAR data for required operations. In this context, a **Point Clouds Data Management System** based on the method of *foreign data wrapper* is introduced here, which is aimed to handle multiple LiDAR data with different file formats, sizes and extent, as well as their metadata. Therefore multiple files manipulation is the core of this Point Clouds Data Management System.

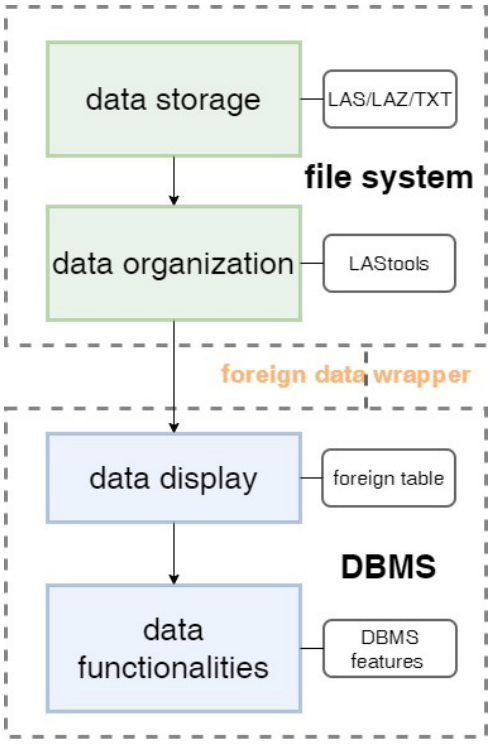


Figure 3.1: Foreign data wrapper solution for LiDAR data

In this Point Clouds Data Management System, the general requirements of LiDAR data usage will be fulfilled, including original storage, easy access, efficient and integrated filtering and uniform query language. In addition to this, this system will also deal with the metadata information for registration and pre-selection. Lastly, this management system is built on open-source technologies. The proposed point clouds data management system relies on PostgreSQL, PostGIS, Multicorn and LAStools. The systematic idea is to store several LiDAR data in a file system, utilizing foreign data wrappers to access and query the data on PostgreSQL. A metadata file is used for registration and pre-filtering. The point clouds data management system developed in this thesis is going to handle a file system includes multiple point clouds file together with a file describing their metadata information. Therefore, the building phase of this system is not only just collecting the point clouds files but also the registration of these file in a metadata file. In this metadata file, the specific features of point clouds files are recorded during the registration process, the features are the filename, file format of point clouds data, spatial extent, maximal and minimal x and y coordinates, total number of points, indexed or not. During the operation phase of this phase, the metadata information is also asked for pre-selection before the filtering and fetching upon several LiDAR files

Several point clouds of different formats and sizes, spatial extent are stored in the same file system while in their original formats like TXT, LAS and LAZip. When a new file is added into this file system, the registration happens in the meantime, which means the features of this file e.g. the filename, file format and spatial extent, etc are recorded into the metadata file. This registration file will be used for the query of this file system later. When a spatial selection is executed, this system will first check with the metadata file before the actual filtering and fetching from the point clouds files. The management system will first search for the relevant point clouds file based on the criteria whether spatial extent in the metadata file is overlapping with the query region. Afterwards, there can be one or more point clouds files are relevant to this query, thus they will be read by the foreign data wrapper.

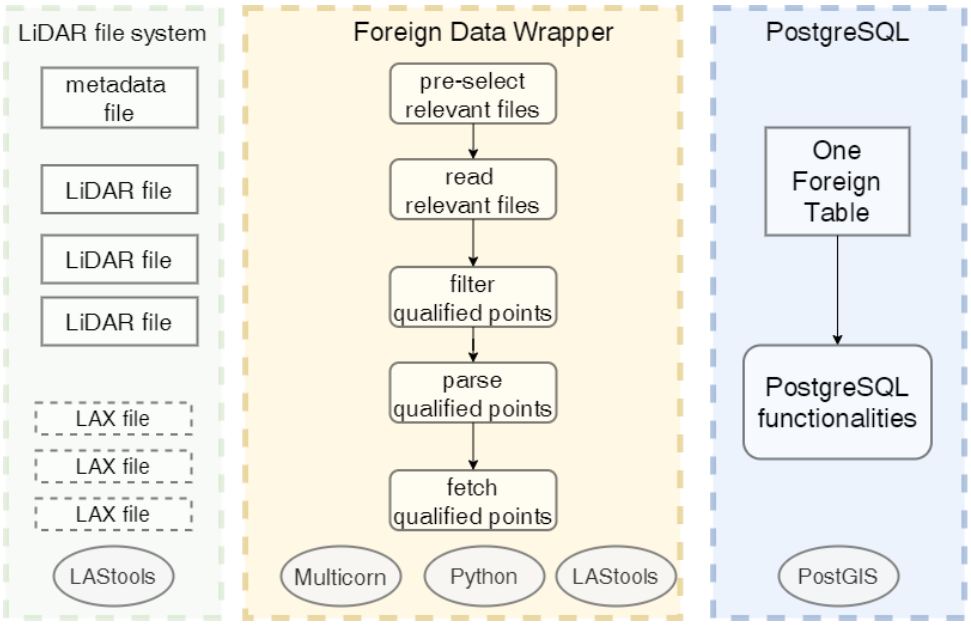


Figure 3.2: System Architecture

3.4.1 System components

The Point Clouds Data Management System based on Foreign Data Wrapper, includes the following components:

- PostgreSQL
- PostGIS
- LAStools
- Multicorn
- Foreign Data Wrapper as showed in figure 3.1
- Entry of metadata file
- Entry of point clouds files
- Support for different file formats (TXT, LAS, LAZ)
- Support for multiple files reading
- Support for querying
- Support for data manipulation
- Support for spatial data methods

These components are depicted in the figure 3.2.

3.4.2 System process

File system building

The construction of the target file system includes two part: one is the collection of different LiDAR file having different formats and sizes, spatial extent. They are mixed and stored in one single file folder, i.e. using the same file path which is the connection basis of system wrapper.

Another part is the file registration, for this purpose, one metadata file is created besides several point clouds files to record the features of every point clouds files in this file system. I uses Microsoft excel sheet to type in the information od these features: *file_id*, *filename*, *file_format*, *file_size*,*count* ,*min_x*, *max_x*, *min_y*, *max_y*, (using lasinfo of LAStools to get these in header information) of each point cloud file, and then save this metadata file as Comma Separate Values (CSV) file for foreign data wrapper as Python script to read when the query is executed.

Data access

The access schema of this foreign data wrapper is designed to read multiple LiDAR files stored on the same file system with the same file path being the connection basis between PostgreSQL side and file system side. Before the data access, the underlying foreign server and foreign tables need to be declared to bridge file system and PostgreSQL. The connection information is needed to be given in the `OPTIONS` in CREATE FOREIGN TABLE statement.

```
CREATE SERVER pc_server
FOREIGN DATA WRAPPER multicorn
OPTIONS ( wrapper 'SystemFdw' ) ;

CREATE FOREIGN TABLE pc_table (
x DOUBLE PRECISION,
y DOUBLE PRECISION,
z DOUBLE PRECISION,
classification INTEGER
...)
SERVER pc_server
OPTIONS ( filepath '.../C_37EN1' ) ;
```

Querying

When an query is executed on the foreign table at the PostgreSQL, the system FDW is actually responsible for the iterative retrieving on foreign data stored at file system. Inside the Python foreign data wrapper, the actual remote query execution runs.

Step 1: Pre-selection

The first thing python script does is to check if there is search conditions given and to pre-select out relevant files. No matter what query conditions are given in `WHERE` clause, the foreign data wrapper reads the metadata file before reading LiDAR files in order to get the information of every files in the system. Because the querying on LiDAR data is often spatial selection, the foreign data wrapper is designed only to identify the search condition based on position, which means its python script only create the *Qua* having field of x and y coordinate.

If there is no search condition or the conditions are only based on other attributes rather than position i.e. no spatial selection in the querying, no *Qual* object and thus no *quals* (a list of *Qual*) is created. Thus all the LiDAR files in the file system are selected as the group of relevant files going to be read. If the user run a querying having the search condition based on the position, the Python script is going to build

python list object named *quals* of *Qual* instances derived from `WHERE` clause, while each *Qual* instance represents one value expression.

Because the spatial selections on LiDAR data are always AoI selection, i.e. Point-in-Polygon querying, the spatial selections are divided into two main types based on query region: rectangle query and irregular region query. The designed querying process uses a selection box , which is the bounding box of querying region. Therefore, selection box is the region itself when running rectangle querying and the bounding box of the polygon when running irregular querying. When a spatial querying is executed, the selection box is used in the first stage as the search condition, thus there are four *Qual* instances representing the min_x, max_x, min_y and max_y of selection box respectively i.e. the *quals* consists of the values of box range.

Then, the Python wrapper is going to read the metadata files in order to make the pre-selection to recognize the relevant LiDAR files. When there are search conditions of given in `WHERE` clause and the field is spatial field, i.e. x and y coordinates, the *quals* list is created .Thus each *Qual* stands for the selection box range of left, right and up, down. The Python script is going to read the metadata file in the LiDAR file system and recognize the relevant files which satisfy these conditions by comparing the header information in metadata file whether the spatial extent of each LiDAR file is inside the selection box range. Then consequently the foreign data wrapper picks out the relevant files from all the files, which are the LiDAR is going to be read and group these relevant files in a Python dict objects i.e. put the information of the point clouds files that is going to be read into an Python dictionary object. During the pre-selection step, the files selected out are going to be read in sequence. When there is no query condition, all the LiDAR files in this file system are going to be read by the foreign data wrapper

Therefore, there are multiple target files are going to be accessed by the same foreign data wrapper through the same foreign server, and fetched onto the same foreign table. Because the Python foreign data wrapper works in an iteration schema, the Python script gathers the entries of multiple files in a dictionary, and then they can be entered in a sequence. This means the foreign data wrapper fist access one file and fetch the point records one by one to the PostgreSQL, after finishing this file, the next file will be entered in the same iterative procedure.

Step 2: Refined filtering

The second step is filtering the qualified points that are inside the selection box. When executing a box spatial query, i.e. query region is a rectangle, the foreign data wrapper firstly define a selection box and read the metadata file to get the overlapping files by comparing the extent of each file to the selection box. Thus the relevant files are recognized, and then the entries of these files are selected out and gathered, i.e. the information of the relevant files are stored in a Python dictionary object inside the wrapper. Next, the foreign data wrapper will continue to read the data content of each relevant files by corresponding file reader, i.e. TXT reader, LAS reader or LAZ reader, according to data format. When these readers are obtaining the point records, they will filter the qualified points using *Qual* object whether the points in each overlapping file are inside the selection box. Otherwise all points in the relevant files are selected. And then fetch these asked point records to PostgreSQL. The procedure of rectangle selection is first selecting relevant files depending on metadata file, secondly filtering the qualified points, and last step is transferring the required point records to foreign table.

If the query region is an irregular polygon, it is necessary to use a local spatial table to store the polygon geography including the table attributes: *geometry column, id , name*. The query procedure is first selecting the relevant files comparing the spatial extent of LiDAR file with selection box (bounding box of query polygon). The relevant files whose extent intersects with the selection box range are pre-selected. Secondly to filter the qualified points by **Qual** object inside selection box, i.e. maximal and minimal x and y coordinates of the query polygon, and thirdly fetching these points to foreign table. Therefore, the points inside the bounding box of query polygon are delivered to PostgreSQL. And then cast to geography point and make point-in-polygon query by PostgreSQL with the support of PostGIS.

4 Implementation

This chapter will describe the implementation of FDW for Point Clouds Data Management System in details and experiments of small test to show the feasibility of the functionalities of implemented FDW from the practice point of review. The codes is published on GitHub. The design and concept of methodology are introduced in Chapter 3. This chapter focuses on the details of implementation and benchmark of the developed method.

Section 4.1 introduces the contained tools including hardware configuration, softwares and libraries, database and extensions used in the experiments. Section 4.2 describes the details of how the Point Clouds Data Management System is built and employed. Section 4.3 presents the performance tests to conduct based on evaluation aspects and levels.

4.1 Tools and datasets

As mentioned before, the implementation is based on both file-based tools and database. The FDW is written in Python script and some libraries are also involved. Since the performance of benchmark will be evaluated and analyzed, the hardware is also relevant.

4.1.1 Hardware

The laptop used is ThinkPad S2 3rd Gen. The processor is Intel® Core™ i5-8250U CPU @ 1.60GHz × 8. Graphics is Intel® UHD Graphics 620 (Kabylake GT2).The operating system is Ubuntu 16.04 64-bit.

4.1.2 Software

The following softwares and libraries support the implementation and usage of Foreign Data Wrapper for handling point clouds data.

- PostgreSQL

  The aim of this research is to directly use point cloud data in PostgreSQL stored in file system. PostgreSQL is an open-source object relational database management system [The PostgreSQL Global Development Group, 2019].PostgreSQL provides basic features to store and access data and also more advanced features are supported. Operations can be performed by using SQL language.

- Python

  The proposed hybrid solution for handling point cloud data is Foreign Data Wrapper to encapsulate the operations. Python programming language is used to write such a FDW that PostgreSQL can depend on for foreign data operations. Python 3.7 is used in this research.

- Multicorn

  The development of FDW is based on Mulitcorn that is an extension of PostgreSQL [Dunklau and Mounier, 2017]. Multicorn is used to make writing FDW easy bu allowing the user to use the Python programming language.

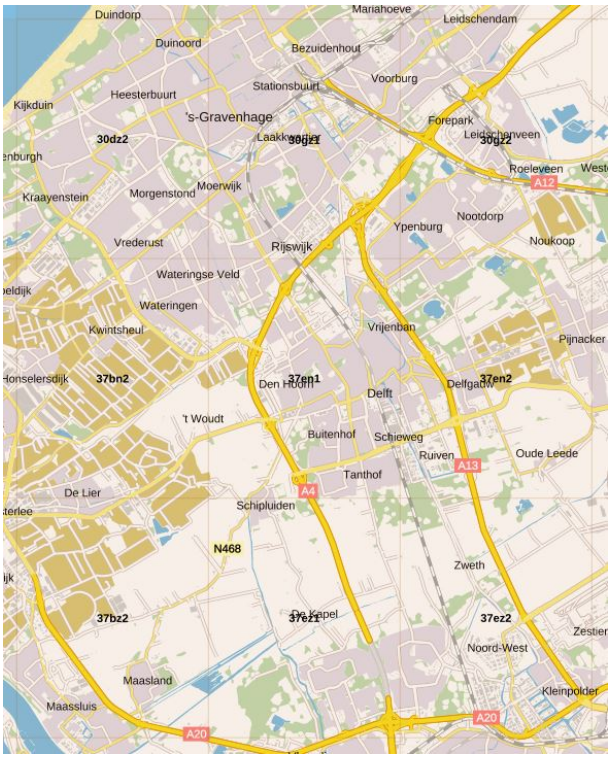


Figure 4.1: Datasets from AHN3

The usage of Multicorn foreign data wrapper is not different from other foreign data wrappers. The first step is to create the extension in the target database bu running the following SQL:

```
CREATE EXTENSION multicorn;
```

The next step is to create a server, in the `OPTION` clause, the user have to give an option named wrapper which contain the fully-qualified class name of the multicorn foreign data wrapper to use.

```
CREATE SERVER system_srv FOREIGN DATA WRAPPER multicorn
OPTIONS (wrapper 'myfdw.SystemFdw');
```

- LAStools

  LAStools software is an open source efficient tool for LiDAR data processing which is widen used because of it blazing speeds and high productivity, It has robust algorithms with efficient I/O and clever memory management to achieve high throughput for data sets containing billion of points.

4.1.3 Datases

To investigate the feasibility, efficiency and scalability of foreign data wrapper method, different datasets will be used for benchmark test. This research uses subsets of AHN with different sizes. The sample density is 6-10 points per square meter of the country, resulting in 640 billion points organized in 60,185 files. The reference system used is the Amersfoort/RD New, which European Petroleum Survey Group (EPSG) code or SRID is 28992.

filename	number of points	min_x	max_x	min_y	max_y	file size(byte)
30DZ2	803933102	75000.000	79999.999	450000.000	456249.999	4239691086
30GZ1	714550010	80000.000	84999.999	450000.000	456249.999	3682486723
30GZ2	578255743	85000.000	89999.999	450000.000	456249.999	2790939970
37BN2	506360353	75000.000	79999.999	443750.000	449999.999	2614200800
37EN1	508564458	80000.000	84999.999	443750.000	449999.999	2571777712
37EN2	524635218	85000.000	89999.999	443750.000	449999.999	2661945848
37BZ2	473420115	75000.000	79999.999	437500.000	443749.999	2252538987
37EZ1	468682253	80000.000	84999.999	437500.000	443749.999	2158133859
37EZ2	495827533	85000.000	89999.999	437500.000	443749.999	2323723409

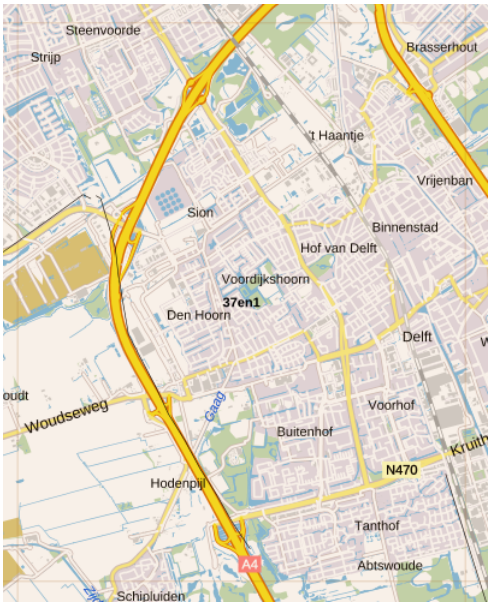
Table 4.1: Metadata info about each AHN dataset

AHN is the digital elevation map for the whole of the Netherlands, which contains detailed and precise elevation data with an average of eight elevation measurements per square meter. Height is measured using laser altimetry, also the LiDAR technique, an airplane or helicopter scans the earth's surface with a laser beam. The measurement of the transit time of the laser reflection and of the attitude and position of the aircraft together give a very accurate result. A number of products have been made of the measured heights, which can be divided into rasters and 3D point clouds. The point cloud is a LAS file where a classification has been applied to the individual points. Each point is assigned to one of the following classes: ground level, buildings, water, artwork or other. In addition, extra attributes have been included for each point. LAS is a standard binary format for storing and exchanging LiDAR data. The LAS file is compressed into an LAZ (or LASzip) file. By applying the compression, the original LAS file is reduced to approximately 10% without loss of quality.

The test datasets are divided into 3 scales based on file extent and thus size: small, medium and large, so as to evaluate the feasibility and scalability. The larger scale covers the smaller one. The whole test uses 9 AHN LAZ files as showed in figure 4.1 . The header information are given in table 4.1, including the maximal, minimal x and y coordinates, file size and total points number. In this design, the small scale test uses one AHN file (37EN1), and the medium scale test uses 4 files, finally 9 files are all involved in test of large scale.

Because in the small scale test, the AHN LAZ file is the raw data source downloading from PDOK, but the benchmark is to test a dataset which is a file system i.e. a collection of files, therefore it is necessary to split this one AHN LAZ file into several test files to building a file system for small scale test. Splitting process is only necessary in the benchmark to build the file system of multiple files collection, but in the reality applications, the operations can be run directly on the original AHN point clouds files.

Since in each scale test, different query levels depending on the number of overlapping files will be evaluated, each downloaded AHN LAZ file is divided into 16 sub datasets as the test data unit, by using *lassplit* of LAStools for experiments. *lassplit* splits the input files into several output files based on various parameters. By default *lassplit* splits a combined LAS or LAZ file into its original, individual flight lines by splitting based on the point source ID of the points. Other options based on other attributes are also supported like *'-by_classification'*, *'-by_gps_time'*, etc. Since the datasets are going to be tested spatial queries, the input point clouds files are divided on spatial basis. In this research, I use options *'-by_x_interval'* and *'-by_y_interval'* to split the LAS and LAZ based on x and y coordinates intervals, the split output point clouds files are the test datasets. The header information from the input file is used for each written output while these LAZ header attributes are updated: number of point records, number of points by return, max and min of x, y and z coordinates. The output files, i.e. split sub-files are named with a suffix that corresponds to the point source ID as in table 4.2. The data for small scale test is *C_37EN1*, it has been divided into 16 sub files based on x and y coordinates, the header information is showed in table 4.2.



(a)

48.216	49.216	50.216	51.216
48.215	49.215	50.215	51.215
48.214	49.214	50.215	51.214
48.213	49.213	50.213	51.213

(b)

Figure 4.2: Small scale test datasets. (a) 37en1 AHN file. (b) The split of 37en1 AHN file.

file name	file format	min_x	max_x	min_y	max_y	number of points
C_37EN1_0000048_0000213.laz	LAZ	79999.998	80833.351	443749.998	444791.597	10500367
C_37EN1_0000048_0000214.laz	LAZ	79999.998	80833.351	444791.598	446874.930	21311433
C_37EN1_0000048_0000215.laz	LAZ	79999.998	80833.351	446874.931	448958.263	26174412
C_37EN1_0000048_0000216.laz	LAZ	79999.998	80833.351	448958.264	450000.001	12006886
C_37EN1_0000049_0000213.laz	LAZ	80833.348	82500.018	443749.998	444791.597	21939891
C_37EN1_0000049_0000214.laz	LAZ	80833.348	82500.018	444791.598	446874.930	48441291
C_37EN1_0000049_0000215.laz	LAZ	80833.348	82500.018	446874.931	448958.263	49152590
C_37EN1_0000049_0000216.laz	LAZ	80833.348	82500.018	448958.264	450000.001	28375818
C_37EN1_0000050_0000213.laz	LAZ	82500.015	84166.685	443749.998	444791.597	25351931
C_37EN1_0000050_0000214.laz	LAZ	82500.015	84166.685	444791.598	446874.930	66377715
C_37EN1_0000050_0000215.laz	LAZ	82500.015	84166.685	446874.931	448958.263	66393726
C_37EN1_0000050_0000216.laz	LAZ	82500.015	84166.685	448958.264	450000.001	39737993
C_37EN1_0000051_0000213.laz	LAZ	84166.682	85000.001	443749.998	444791.597	14942766
C_37EN1_0000051_0000214.laz	LAZ	84166.682	85000.001	444791.598	446874.930	28772802
C_37EN1_0000051_0000215.laz	LAZ	84166.682	85000.001	446874.931	448958.263	37390101
C_37EN1_0000051_0000216.laz	LAZ	84166.682	85000.001	448958.264	450000.001	11694736

Table 4.2: Sub datasets splitted from AHN3 *C_37EN1.LAZ*

4.2 Point Clouds Data Management System

4.2.1 Metadata file

In this LiDAR file system, besides different point clouds files, there is one metadata file storing the header information of all the LiDAR data. It is simply managed in CSV file storing several features of each LiDAR file, including filename, file format, minimal and maximal x and y coordinates, number of points, and number of outside points, file size. etc, part of them are showed in table 4.2 which file system for small scale test. To get the header information of each point clouds file, *lasinfo* of LAStools is used. *lasinfo* reports the contents of the header and a short summary of the points. By default *lasinfo* reports the min and max of every point attribute after parsing all the points and also counts the points falling outside the header bounding box. And then to register these information into the metadata CSV file. When a query is run, the foreign data wrapper will first read this metadata file no matter whether and what the search conditions are. The metadata file has registered every LiDAR files residing on the file system. The attributes of filename, file format, bounding box (BBX),etc. All these information are obtained by *lasinfo* of LAStools and used by the system foreign data wrapper.

- **file name** (with file path): build the entry of the LiDAR file
- **file format** : call the according LiDAR file reader function
- **file extent** : compare with the selection box

4.2.2 Data access

Data access of this Point Clouds Data Management System includes data storage and data preparation, the recognition of relevant files and their format, reading LiDAR data from multiple files and data presentation on PostgreSQL. Part of these operations are realized on the file system side, like data sorting and indexing process uses *lassort* and *lasindex* from LAStools to organize data for each LAS and LAZ point clouds files. It is necessary to do the data organization to improve the query efficiency. While part of these operations are implemented inside the Python foreign data wrappers, like reading data content, for example to read LiDAR files in LAS and LAZ formats, the *las2txt* of LAStools is used by Python script. *las2txt* converts from binary LAS/LAZ to an ASCII text format. Thus this tool is used to parse point records in LAS and LAZ files by FDW to obtain abd fetch the information content of point clouds files.

At the data preparation phase, first I use *lassort* to reorder the points of each test LiDAR files according to their position in a 2D Morton SFC. Then I use *lasindex* to create the LAX indexing file based on a quad tree, which is used to accelerate the querying. In the benchmark, we do the lassort and lasindex to organize the point clouds files individually as the data preparation phase in still processing on file system side, to improve the selection efficiency especially the spatial selection. *lassort* sorts the points of a LAS file into z-order cells of a square quad tree and saves them into LAS or LAZ format. *lasindex* creates a spatial index LAX file for for a given LAS or LAZ file to fast spatial queries. When this LAX file is present it will be used to speed up access to the relevant areas of LAS and LAZ.

Then it is the data loading phase inside the foreign data wrapper, *las2txt* is used to convert each point record in binary LAS and LAZ to ASCII text format. The 'parse' flag defines how to order each line of ASCII file. The supported entries are a - scan angle, i - intensity, n - number returns for given pulse, r - number of return, c - classification, u - user data, p - point source ID, e - edge of flight line flag, and d - direction of scan flag, R - red channel of RGB color, G - green channel of RGB color, B - blue channel of RGB color. The '-sep' flag specifies what delimiter to use. The default one is a a space but 'tab', 'comma', 'colon', 'hyphen', 'dot', 'semicolon' are other possibilities.

The foreign data wrapper named 'SystemFdw' is built to connect the LiDAR file system and the PostgreSQL. It can parse and deliver the needed data content onto foreign table. The foreign table is designed to display the points content, it is created with following DDL to consist basic columns x, y, z representing the x and y coordinates and height in *double precision* type, classification with *integer* type, optionally adding red, green and blue value, intensity, GPS time, point source ID with *numeric* type. In

addition two extra columns is created for 2D point and 3D point for further casting to geometry and geography point type.

```
CREATE SERVER pc_server FOREIGN DATA WRAPPER multicorn
OPTIONS ( wrapper 'SystemFdw' ) ;

CREATE FOREIGN TABLE pc_table (
x DOUBLE PRECISION,
y DOUBLE PRECISION,
z DOUBLE PRECISION,
classification INTEGER
...)
SERVER pc_server
OPTIONS ( filepath '.../C_37EN1' ) ;
```

After the definition of foreign server and foreign table, the corresponding foreign data wrapper, i.e. 'SystemFdw' needs to be initialized at first query in the implementation of the Python script by *__init__()* method:

```
def __init__(self, fdw_options, fdw_columns):
    super(SystemFdw, self).__init__(fdw_options, fdw_columns)
    self.columns = fdw_columns
    if 'filepath' in fdw_options:
        self.filepath = fdw_options["filepath"]
    else:
        log_to_postgres('filepath parameter is required', ERROR)
```

Therefore, the foreign data wrapper build the bridge between PostgreSQL and the file system as the file path of this file system being the connection basis.

4.2.3 Querying

The query process of retrieving data from the database side is designed to use developed foreign data wrapper for file system management and PostGIS for geometry/geography handling. The queries are going to execute are described in Section 3.3. Most of them are selection queries, i.e. select all the points within a query region. Several types of query region are experimented, including rectangle, circle, simple polygon, etc. Additionally, the nearest neighbor queries and calculation queries are also executed.

When a query is executed, the foreign data wrapper starts to work, since this is where the actual remote query execution occurs. At first, the wrapper needs to get the information from the search condition given in the `WHERE` clause with the help of *Qual* objects, whose principle is introduced before. Now the selection box information (bounding box) of the querying region is obtained by the foreign data wrapper and will be used by wrapper for pre-selection and filtering.

```
for qual in quals:
        if qual.field_name == 'x' and qual.operator == '>' or '>=':
            xmin = qual.value
        if qual.field_name == 'x' and qual.operator == '<' or '<=':
```

```
            xmax = qual.value
        if qual.field_name == 'y' and qual.operator == '>' or '>=':
            ymin = qual.value
        if qual.field_name == 'y' and qual.operator == '<' or '<=':
            ymax = qual.value
```

When the query region is a rectangle, use the following SQL statement to command the query.

```
SELECT COUNT(*) FROM pc_table
WHERE x>80200 AND x<80600 AND y>443800 AND y<444700;
```

Before the selection based on search condition, the foreign data wrapper firstly get the information about the search condition through *Qual* object. Because in this rectangle querying, the selection box is identical to the query region, and four *Qual* instances are constructed and make up a list of *quals*. Now the foreign data wrapper get the range of selection box defined by these four *Qual* instances.

- Left range of selection box : xmin
- Right range of selection box : xmax
- Down range of selection box : ymin
- Up range of selection box : ymax

The Python scripts of FDW begins to read the metadata CSV file to select out the relevant files whose file extent overlaps with the selection box defined by four qualifiers. The foreign data wrapper stores the information of the relevant files are stored in a Python dictionary object. Then the wrapper begin to access relevant point clouds files that can have different file formats, therefore, the script needs to ask Python dictionary for header information about relevant files to obtain the file format and file name of LiDAR file the script is going to read, therefore the foreign data wrapper can choose the appropriate reader according to the file format and access the entry of file. Three reader for TXT, LAS and LAZ file are defined as three Python functions. Now, the wrapper need to call the appropriate function to read each file. The LAS and LAZ reader use the *las2txt* to parse the point records, and uses the selection box for filtering points by , by adding the following option at the end *las2txt* command. Thus, only the qualified point records in the relevant files will be parsed by *las2txt* and be fetched by the foreign data wrapper to PostgreSQL, otherwise all points in the relevant files are selected and transferred.

```
-inside xmin ymin xmax ymax
```

In the rectangle selection, because the selection box is used for pre-selection for relevant files and filtering the qualified points and it is same as rectangle query region, all the fetched point records meet the conditions given in the WHERE clause. The foreign data wrapper do all the steps of selection, when PostgreSQL query the foreign table all the points on the foreign table are the result points, there is no points need to be omitted in PostgreSQL selection step.

1. Get the selection box from *quals*, which is the bounding box of region
2. Read metadata.csv to pre-select the relevant files whose extent overlaps with selection box
3. Store the header information about relevant files in a Python dictionary
4. Call appropriate reader function to read relevant LiDAR files
5. Retrieve LiDAR file and only parse the points inside the selection box

6. Fetch the qualified points to PostgreSQL

When the query region is a n irregular polygon, the PostGIS extension of PostgreSQL is used to make spatial SQL. To query a polygon region, first to store these geometries in a spatial table. During the query on foreign table of point clouds and this local spatial table for polygons will be combined, both of them are reference table. Therefore, it is necessary to create a local table storing the query polygons using the following statement, each query polygon uses a geography column to store the spatial information, and id column as the primary key along with name column.

```
CREATE TABLE query_polygons
(id serial PRIMARY KEY,
name varchar(20),
geom geometry(POLYGON));
```

Then to fill in the query polygon table by inserting information of each polygon. For example it shows the SQL statement for creating the polygon for test of low level query on small scale data.

```
INSERT INTO query_polygons(geom, id, name)
VALUES(
ST_GeomFromText('POLYGON((80200 443800, 80500 444000,
                          80600 444400, 80400 444700,
                          80200 443800))', 28992),
11, small_low')
```

To get the selection box information about each query region (bounding box of region), since the region is an irregular polygon, the functions `ST_Extent` and `ST_XMin, ST_XMax, ST_YMin, ST_YMax` provided by PostGIS is used. In this way, the *quals* for selection box of *execute()* are assigned as the computed bounding box values of the polygon, while in the rectangle selection, the *quals* are assigned directly by the given rectangle.

```
SELECT ST_Extent(geom) FROM query_polygons WHERE id=11;
```

The `ST_Contains` function is used for polygon selection, the SQL statement below, take the low level query(*query_polygons.id=11*) of small scale datasets as an example.

```
SELECT COUNT(*) FROM
(SELECT x,y,z FROM pc_points, query_polygons
 WHERE query_polygons.id=11 AND
 x>ST_XMin(geom) AND
 x<ST_XMax(geom) AND
 y>ST_YMIN(geom) AND
 y<ST_YMax(geom)
)AS relevant, query_polygons
WHERE query_polygons.id=11 AND
ST_Contains(geom,
ST_SetSRID( ST_MakePoint(relevant.x,relevant.y), 28992));
```

If the query region is a polygon, it also contains the steps the foreign data wrapper will carry out for rectangle selection:

1. Get the selection box from *quals*, which is the bounding box of region
2. Read metadata.csv to pre-select the relevant files whose extent overlaps with selection box
3. Store the header information about relevant files in a Python dictionary
4. Call appropriate reader function to read relevant LiDAR files
5. Retrieve LiDAR file and only parse the points inside the selection box
6. Fetch the qualified points to PostgreSQL
7. *Use PostGIS function to select out the points are exactly inside the polygon*

Therefore, in the irregular polygon selection, the points inside the bounding box of the query polygon are fetched to foreign table, and then PostgreSQL/PostGIS needs to screen the points that are exactly inside the polygon by using PostGIS function `ST_Contains`

4.3 Benchmark

The benchmark includes the range of functionalities as well as the performance in terms of cost time and returned points. In the benchmark, the functionalities of this FDW for Point Clouds Data Management System. Different queries using different datasets will be tested and the performance will be evaluated. There are three primary aspects of the Point Clouds Data Management System FDW to be tested and analyzed:

- Feasibility
- Efficiency
- Scalability

The connection information is designed by the entry of file system like the directory path name. The input and output of certain point clouds files upon the file system are handled by the metadata CSV file and FDW. In order to evaluate the aspects of the foreign data wrapper, different levels of querying on different scale of datasets are designed. As showed in table 4.3, there are 3 scales of datasets will be tested, and figure 4.3 show what AHN files are included in each scale of datasets. In the small scale datasets, there are just 1 AHN file coloured with deepest red in figure 4.3 (37EN1), and after splitting it has 16 test LiDAR files; The medium scale datasets consist of 4 AHN files coloured with medium red at right-down side in figure 4.3 (37EN1, 36EN2, 37EZ1, 37EZ2), thus there are 64 test files after splitting; In the large scale test, all these 9 AHN files are involved with the most shallow red in figure 4.3, and 164 test files are collected. Because the performance of querying is also related to the querying levels i.e. the number of points are asked, in this benchmark, the number of relevant files is used to represent the level of querying. To take the small scale file system as an instance, as shoed in table 4.4the query levels are divided in 3 levels, low level query region overlaps with just one test file, as showed as the left bottom in figure 4.4; medium level query region has 4 test files relevant at right top in figure 4.4; high level query region intersects with 9 test files at center in figure 4.4.

Feasibility

In the feasibility test, it needs to be proved that whether the required functionalities can be supported by this Point Clouds Data Management System FDW, and the sub research question 2 can be answered, how does the FDW build the connection between LiDAR files system and PostgreSQL. The basic functionality is spatial query mainly including the following spatial queries. Based on the designed query levels, the rectangle and polygon selection will be executed and further statistical query. Next to spatial query, the query on the attribute fields like classification will be made, e.g. to select out the ground points among a region. The additional functionality to evaluate is the update operation. While executing the data manipulation operations like update, insert and delete, the LiDAR files can be changed synchronously

Data scale	AHN files	Test files
Small	1	(1*16) 16
Medium	4	(4*16) 64
Large	9	(9*16) 164

Table 4.3: Design of dataset scales

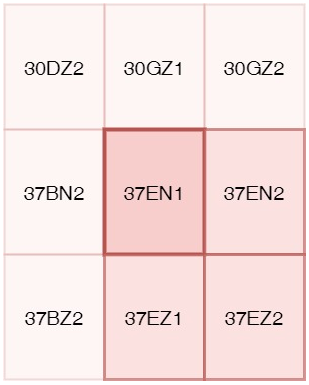


Figure 4.3: Design of dataset scales

Query level	Relevant files in each AHN file
Low	1
Medium	4
high	9

Table 4.4: Design of querying levels

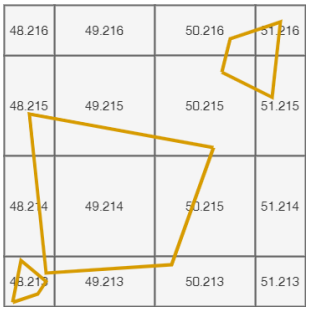


Figure 4.4: Design of Design of querying levels on small scale datasets

both in source files on the file system and the foreign table on the DBMS side. For example, the useless point records like blunders waste the storage space and decrease the query accuracy, it can be necessary to delete them.

- Select all the points
- Select the points within a rectangle
- Select the points within a polygon
- Select the points within a circle
- Select nearest neighbour of the point
- Select the ground points (based on classification attribute)
- Aggregation function

Because in the real-world application, the querying is always like search for the points inside a building area from the point clouds of a whole city and even province, to test the feasibility in the real-word usage, the mini level query on the large scale file system are designed. In the large scale file system which consists of all 9 AHN files, there are 9 tiny selection regions are designed as mini level query as showed in table 4.5. First three of tiny query regions are square having the area of 100 which is really tiny comparing to the area of used datasets. Since the really small size, these querying region just overlaps with 1, 2 and 4 test LiDAR files respectively as showed in figure 4.5. Other groups of query squares have area of 400 and 900 and also have 1, 2 and 4 relevant test files respectively as showed in figure 4.5.

Efficiency

In the efficiency test, the methods to improve the query efficiency will be tested to see whether and how it reduce as the querying time. The first factor to test the efficiency is data organization including sorting and indexing. Indexing is to know the storage location fast when searching the objects. In this

Area	Relevant files
100	1 2 4
400	1 2 4
900	1 2 4

Table 4.5: Feasibility test

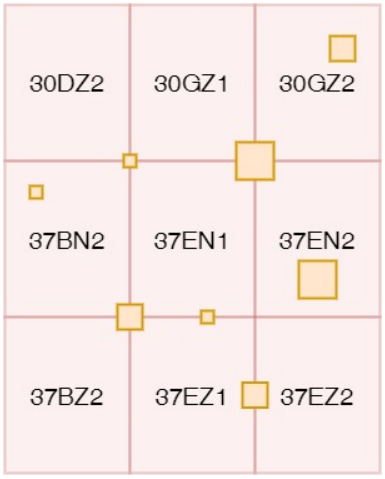


Figure 4.5: Feasibility test

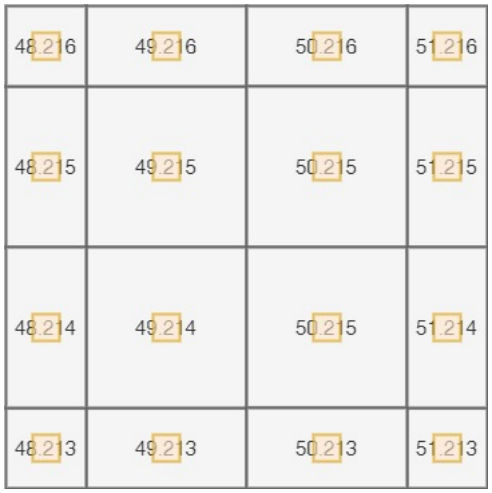


Figure 4.6: Efficiency test

management system, the *lasindex* from LAStools is used to create index for LAS and LAZ files, which creates a LAX file, with this index file present, the retrieving of data can be accelerated. Sorting is storing points close together by using *lassort* from LAStools In order to test the performance improvement, the mini level queries on small scale datasets are designed to query with tiny square selection on the small dataset showed in figure 4.6. Therefore there are 16 querying square with area of 100, each of which overlaps with one single test file in small scale file system in figure 4.6. First to query on the standalone LAS and LAZ files without sorting and indexing, then to query on these files after sorting and beside the corresponding LAX files. The time difference for querying will be compared. The second point is to test is how the qualification in the Multicorn foreign data wrappers improve the efficiency. To use the *Qual* object inside the foreign data wrappers can ask script to filter the data before fetching them to the PostgreSQL. Without the qualifications, all the points in the point clouds file will be delivered to PostgreSQL foreign table, and then execute SQL selection on it. The time difference for querying between using FDW with and without *Qual* will be compared.

Scalability

In the scalability test, different scales of datasets will be used to see how does the data scale effect the performance of the Point Clouds Data Management System FDW; and also different levels of query regions will be asked to explore the relevance between the querying level and performance of FDW. Thus three sizes of sub datasets of AHN with small, medium and large scale will be tested, the larger

Data scale	Query level	Relevant files
Small (1 AHN)	Low	1
	Medium	4
	High	9
Medium (4 AHN)	Low	4
	Medium	16
	High	36
Large (9 AHN)	Low	9
	Medium	36

Table 4.6: Scalability test

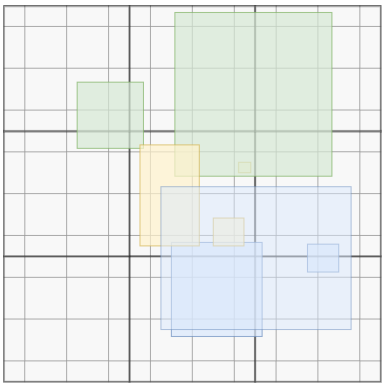


Figure 4.7: Scalability test

datasets covers the previous one. More specific, These three datasets have different file sizes, coverage and number of points as showed in table 4.3. Also three levels with low. medium and high of query according to different sizes of querying regions will be used. For example, the small scale test uses 1 AHN file and thus 16 LiDAR test files in its file system, the low level query region overlaps with 1 test file, medium level query has 4 relevant files and large level query covers 9 point clouds files, as illustrated in 4.4 and also yellow colour in figure 4.7. For medium scale file system which includes 4 AHN files and consequently 64 test files, its file system also uses 3 levels of query regions: the low level region overlaps with 1 test files in each AHN block and thus overall 4 relevant files; since medium level has 4 relevant files in each AHN block, and thus 16 intersecting files; and high query region overlaps with 36 test files. As showed in blue colour in figure 4.7. When querying on the large scale file system which collects 9 AHN files and thus 164 test files. For the low level querying which has 1 relevant files in each AHN block, low level query on large scale has thus 9 relevant files. For medium level query whose region overlaps with 4 files in each AHN block, and thus medium level query on large scale has thus36 relevant files. As showed in green colour in figure 4.7.

5 Results and Analysis

5.1 Feasibility test

In order to evaluate whether the functionality of foreign data wrapper can be successfully executed on PostgreSQL, the following kinds of the queries are executed in sequence:

- Querying on a rectangle of different levels
- Selection based on attribute field *classification* within the query rectangle
- Aggregate function `count` of point records and (`max`, `min`, `avg`) of height value within the rectangle
- Querying on a bounding box of a query polygon
- Selection based on attribute field *classification* within the bounding box
- Aggregate function `count` of point records and (`max`, `min`, `avg`) of height value within the bounding box
- Querying on a query polygon of different levels
- Selection based on attribute field *classification* within the polygon
- Aggregate function `count` of point records and (`max`, `min`, `avg`) of height value within the polygons

In order to have a comprehensive evaluation the benchmark, the benchmark is design to use three scales of test datasets, and each scale of data will be queried by regions based on three levels, as shown in table 5.1. It shows the number of relevant files of each designed query. Because in the real-world application, the querying is always like search for the points inside a building area from the point clouds of a whole city and even province, to test the feasibility in the real-word usage, the mini level query on the large scale file system are designed. In the large scale file system which consists of all 9 AHN files, there are 9 tiny selection regions are designed as mini level query as showed in table 4.5.

Table 5.3 is 9 tiny polygons as the query polygon and table 5.2 is the bounding box of those polygons as the query rectangle, thus the area of polygons are smaller than its corresponding rectangles. From table 5.2 and table 5.3, it can be seen that time is remarkably reduced by the data organization, when region is tiny now matter the query region is an irregular polygon or a rectangle. Query is done in several seconds, not all the files in the underlying large file system need to be read, instead of only relevant files, because of the pre-selection of overlapping files. Although there are still several big point clouds files relevant, only part of the points (not all the points) in the file are required to be read and delivered, because of the data organization and qualifiers used in FDW. Therefore, bu using foreign data wrapper method, it is feasible to query a region with little area like a house, a building or a park, on the LiDAR datasets of a city and even a province, if the storage space is sufficient.

Table 5.4 show the timing of the basic query on LiDAR file system, (low level query on the small scale data) which means the query region overlaps with just one test file in the file system which consists 16

	AHN files	Test files	Low query level	Medium query level	High query level
Small data scale	1	16	1	4	9
Medium data scale	4	64	4	16	36
Large data scale	9	144	9	36	81

Table 5.1: Test design of number of relevant files

Area	Relevant files	Querying time (s)	After organization (s)
100	1	27	0.48
	2	17	0.26
	4	20	0.34
400	1	17	0.50
	2	23	0.33
	4	17	0.27
900	1	23	0.74
	2	44	0.87
	4	18	0.34

Table 5.2: Querying rectangles of mini level on large scale data

Area of bbx	Relevant files	Area of polygon	Querying time (s)	After organization (s)
100	1	54.5	27	0.47
	2	52	17	0.25
	4	45.5	20	0.31
400	1	220	16	0.53
	2	230	22	0.33
	4	186	18	0.29
900	1	458	24	0.85
	2	318	43	0.93
	4	441	19	0.36

Table 5.3: Querying polygons of mini level on large scale data

Query region	Classification	System FDW (min)	After Organize (min)	No quals (min)
Rectangle	all point	2.30	2.17	9.13
	ground	2.42	2.32	9.50
Bounding Box	all points	0.55	0.49	1.47
	ground	0.58	0.54	2.05
Polygon	all points	0.58	1	2.11
	ground	1.05	1.03	2.14

Table 5.4: Timing of queries on Small scale data by Low level regions

Query region	Classification	Aggregate func	System FDW	After Organize	No quals
Rectangle	all points	COUNT	13 000 976	13 000 982	13 000 976
		MAX(z)	62.490	62.489	62.490
		MIN(z)	-8.632	-8.632	-8.632
		AVG(z)	-0.490	-0.491	-0.490
	ground points	COUNT	10 537 701	10 537 689	10 537 701
		MAX(z)	5.170	5.169	5.170
		MIN(z)	-8.632	-8.633	-8.6327
		AVG(z)	-1.574	-1.575	-1.5741
Bounding Box	all points	COUNT	4 724 593	4 724 588	4 724 593
		MAX(z)	16.950	16.949	16.950
		MIN(z)	-3.295	-3.296	-3.295
		AVG(z)	-1.383	-1.384	-1.383
	ground points	COUNT	3 841 360	3 841 359	3 841 360
		MAX(z)	1.570	1.569	1.570
		MIN(z)	-3.286	-3.287	-3.286
		AVG(z)	-2.125	-2.126	-2.125
Polygon	all points	COUNT	2 263 124	2 263 118	2 263 124
		MAX(z)	16.950	16.949	16.950
		MIN(z)	-3.257	-3.258	-3.257
		AVG(z)	-0.994	-0.995	-0.994
	ground points	COUNT	1 669 467	1 669 465	1 669 467
		MAX(z)	1.570	1.569	1.570
		MIN(z)	-3.257	-3.258	-3.257
		AVG(z)	-2.04	-2.041	-2.04

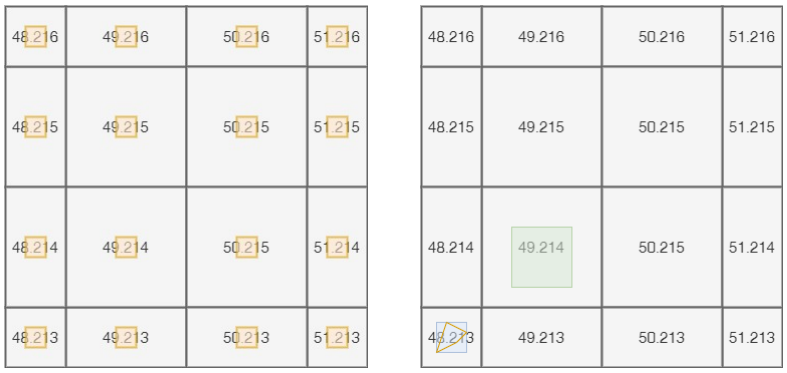
Table 5.5: Aggregate functions of queries on Small scale data by Low level regions

Query region	Classification	System FDW (min)	COUNT	MAX	MIN	AVG
Rectangle	all	5.3	24 344 532	80.138	-5.233	5.312
	ground	5.25	9 756 922	4.940	-5.233	0.274
Bounding Box	all	15.51	76 208 194	109.193	-5.9951	4.072
	ground	16.16	33 558 362	4.940	-5.995	-0.257
Polygon	all	19.23	41 749 091	80.138	-5.995	4.5191
	ground	17.08	17 526 258	4.940	-5.995	-0.110

Table 5.6: Timing and Aggregation functions of queries on Small scale data by Medium level regions

point clouds files in total, the visualization is depicted in figure 5.1. The query includes all the steps described above, it can be seen the querying time for polygon and its bounding box are close, resulting from that they have similar area of region. In addition, the rectangle has larger area, thus it consumes more time for retrieving the points inside it. The functionality of spatial queries which includes rectangle selection and polygon selection, query on attributes, aggregation functions, are all proven to be feasible through this FDW Point Cloud Data Management System, the results area showed in table 5.5 and 5.4.

Because the core of this FDW Point Cloud Data Management System is to handle multiple LiDAR files, higher level queries on small scale data are required to be included in feasibility test. Small scale data running medium level and high level queries proves that it is also feasible to handle multiple relevant files, the results points that are from different point clouds files can be displayed together on the same foreign table. Medium level query regions overlap with 4 files while high level query regions cover 9 files. All these relevant files will be read in sequence, and the content of these files will be obtained and fetched onto the same foreign table. The results are showed in table 5.6.



(a) (b)

Figure 5.1: Data organization test (a) Mini level query on small scale data. (b) Small level query on small scale data.

Filename	Query time (s)	After organization(s)
C_37EN1_0000048_0000213.laz	4	0.08
C_37EN1_0000048_0000214.laz	8	0.09
C_37EN1_0000048_0000215.laz	11	0.11
C_37EN1_0000048_0000216.laz	5	0.13
C_37EN1_0000049_0000213.laz	9	0.07
C_37EN1_0000049_0000214.laz	17	0.14
C_37EN1_0000049_0000215.laz	18	0.12
C_37EN1_0000049_0000216.laz	11	0.13
C_37EN1_0000050_0000213.laz	10	0.15
C_37EN1_0000050_0000214.laz	26	0.16
C_37EN1_0000050_0000215.laz	26	0.05
C_37EN1_0000050_0000216.laz	15	0.07
C_37EN1_0000051_0000213.laz	6	0.08
C_37EN1_0000051_0000214.laz	11	0.11
C_37EN1_0000051_0000215.laz	15	0.14
C_37EN1_0000051_0000216.laz	5	0.14

Table 5.7: Data Organization Test (Querying Time of Mini level Query on Small scale datasets)

5.2 Efficiency test

In order to evaluate the techniques that can be applied to improve the efficiency of query, the mini level queries based on small scale data are designed and tested. This is to select the points within a really tiny rectangle with x and y coordinates range as 4 like the area of a retail store, which overlaps with only one point clouds file. Because there are 16 test files in the small scale datasets, the mini level test uses also 16 the tiny rectangles that have the same really tiny extent, which are relevant to each 16 files as showed in figure 5.1 (a).

First to query on the standalone LAS and LAZ files without sorting and indexing, then to query on these files after sorting and beside the corresponding LAX files. The time difference for querying will be compared. The querying time difference is represented table 5.7. When a really tiny region is queried one LiDAR files system which consists of the sorted point clouds files (by *'lassort'*) and each of them with a LAX indexing file besides (by *'lasindex'*), the efficiency is improved remarkably around 100 times.

The efficiency of data organization is also evaluated by low level queries on small scale data, which uses larger query regions than mini level querying and the query region is around of half area to the extent of its relevant files, as showed in figure 5.1 (b). It can be seen that the time difference gets smaller when

Query region	Classification	Query time(s)	After organization(s)
polygon	all points	63	61
	ground points	65	64
bounding box	all points	55	49
	ground points	58	54
rectangle	all points	150	138
	ground points	162	152

Table 5.8: Data Organization Test (Querying Time of Low level Query on Small scale datasets)

Filename	Using quals(s)	Without qual(ms)
C_37EN1_0000048_0000213.laz	4	113
C_37EN1_0000048_0000214.laz	8	221
C_37EN1_0000048_0000215.laz	11	284
C_37EN1_0000048_0000216.laz	5	127
C_37EN1_0000049_0000213.laz	9	241
C_37EN1_0000049_0000214.laz	17	526
C_37EN1_0000049_0000215.laz	18	521
C_37EN1_0000049_0000216.laz	11	302
C_37EN1_0000050_0000213.laz	10	266
C_37EN1_0000050_0000214.laz	26	772
C_37EN1_0000050_0000215.laz	26	713
C_37EN1_0000050_0000216.laz	15	426
C_37EN1_0000051_0000213.laz	6	170
C_37EN1_0000051_0000214.laz	11	302
C_37EN1_0000051_0000215.laz	15	427
C_37EN1_0000051_0000216.laz	5	130

Table 5.9: FDW Qualifiers Test (Querying Time of Mini level Query on Small scale datasets)

the query level gets higher. When the low level selection regions are queried on the same small scale test dataset , the results of time difference is much smaller represented in table 5.8.

Although data organization methods have indeed increased the efficiency, the time difference is less great as in min level test. This is because that the area of the query region of low level test is around half of the area of point clouds files, the clustering of the search objects are not quite helpful in this kind of cases that the both areas are at the same level. And in the mini query test, the area of retrieved whole point clouds file extent is quite huge compared to mini selection rectangle.

Additionally, the efficiency of Multicorn qualifiers is also evaluated, the querying time difference is represented in table 5.9. When querying the same file system but using another foreign data wrapper without *Qual* objects inside, the consuming time increases a lot. Since without using qualification, the foreign data wrapper will read and deliver all the points in the relevant file, while *Qual* objects can help foreign data wrapper to do a filtering before fetching data content to the tables, thus only the qualified points will be delivered.

The number of returned points of three different accessing procedures is showed in table 5.10. The number of points returned from querying the original file system using or without *Quals* are exactly the same in all these 16 mini level queries. However, the querying on the organized point clouds file system shows the slight difference of point records count, it may results from the *'lassort'* alters the position of some points in the point cloud, and hence there are different query results. As table 5.5 display the statistic data of the low queries on small scale data, the difference brought by the data organization tools still exists, and the error becomes larger when the covering range is getting bigger.

In the benchmark, I decided to use *Quals* inside foreign data wrapper and not use data organization, because in the higher level query test, data organization would not improve so much efficiency, while the sorted point clouds file and LAX indexing files would take extra storage space of disk.

Filename	Query count	After organization	Without quals
C_37EN1_0000048_0000213.laz	124	124	124
C_37EN1_0000048_0000214.laz	238	238	238
C_37EN1_0000048_0000215.laz	165	166	165
C_37EN1_0000048_0000216.laz	114	114	114
C_37EN1_0000049_0000213.laz	116	116	116
C_37EN1_0000049_0000214.laz	253	252	253
C_37EN1_0000049_0000215.laz	177	177	177
C_37EN1_0000049_0000216.laz	128	128	128
C_37EN1_0000050_0000213.laz	140	140	140
C_37EN1_0000050_0000214.laz	270	270	270
C_37EN1_0000050_0000215.laz	179	179	179
C_37EN1_0000050_0000216.laz	88	87	88
C_37EN1_0000051_0000213.laz	122	122	122
C_37EN1_0000051_0000214.laz	1022	1022	1022
C_37EN1_0000051_0000215.laz	219	219	219
C_37EN1_0000051_0000216.laz	129	129	129

Table 5.10: Returned points After organization and Without quals

5.3 Scalability test

In order to have a comprehensive evaluation the benchmark, the benchmark is design to use three scales of test datasets, and each scale of data will be queried by regions based on three levels, as shown in table 5.1 which shows the number of relevant files of each designed query. And the representation of all these 9 different rectangle queries are showed in figure 4.7. Results of the scalability test of rectangle selection are showed in table 5.11, the selection regions are the bounding box of the corresponding polygons in table 5.12. Thus these polygons have the smaller area and less points to the rectangle. It can be seen from the results that time for each polygon selection is more than the time for its bounding box . The last column in table 5.11 and table 5.12 is speed which means "how many seconds needed for 1 million returned points". The speed of bounding box selection is fast than polygon. Because the difference between rectangle and polygon selection, polygon selection has a extra steps on PostgreSQL/PostGIS to select out the points exactly inside the polygon and omit the points that are inside the bounding box but not in the polygon.

In the small scale data test, the higher query level is, the more relevant files it has. Its can been seen that the high level query overlaps with more files than medium level query, but it costs less time, it is because its query region bounding box count is smaller. The analysis is that there is no relevance between the querying time and the query level (how many files it overlaps).

When the query goes to the larger scale data, I use more (64) point clouds files in one file system which is defined as medium scale in this benchmark. The low level query of the medium scale test has 4 relevant files (medium-low), compared to the test on smaller dataset which tests small scale data queried by medium level region (small-medium), both queries covers 4 relevant files in their separate file systems. It takes less time to make selection even on the larger scale data, this is because this test uses the query region having the smaller area. The analysis is the querying time is relevant to the area of query region but not the data scale.

Time difference between the polygon selection and bounding box selection (time polygon selection – time bbx selection) is the time cost by PostGIS to do the query on the PostgreSQL foreign table as showed in table 5.13 This time is also relevant to returned points count.

I also made the CTAS (CREATE TABLE AS) query procure, it create a local PostgreSQL TABLE to storing the points inside the bounding box of polygon, and then further selection by PostgreSQL/PostGIS is made on this table. The difference between the CTAS method and original method is that, the CTAS procedure stores the points of bounding box in a local table taking the space of disk, but the original procedure take the storage of memory. This alternative works well until the medium level query on

Data scale	Query level	Relevant files	Area (km2)	Count (million)	Time (min)	speed (s/million)
Small	Low	1	0.4	5	1	11.7
	Medium	4	3.8	76	16	12.5
	High	9	3.5	49	12	14.7
Medium	Low	4	1.2	30	6	12.9
	Medium	16	8.5	137	31	13.4
	High	36	31.5	502	107	12.8
Large	Low	9	13	203	42	12.3
	Medium	36	31.4	653	135	12.4

Table 5.11: Scalability test of bounding box selection

Data scale	Query level	Relevant files	Area (km2)	Count (million)	Time (min)	speed (s/million)
Small	Low	1	0.2	2	1	27.8
	Medium	4	2.0	42	19	27.9
	High	9	3.5	49	15	18.3
Medium	Low	4	0.6	17	8	27.5
	Medium	16	7.6	122	37	18.3
	High	36	31.4	501	137	16.4
Large	Low	9	8.3	132	52	23.9
	Medium	36	31.3	653	188	17.3

Table 5.12: Scalability test of polygon selection

medium scale data, which has 16 relevant files. The total cost time of directly polygon selection and CTAS selection which creates an intermediate table are similar. However, when the test goes larger and higher, this alternative is not feasible any more, in the medium scale data querying on high level region (medium-high) and the large scale data querying on medium level region (large-medium), both of these two test cost around 2 hours in original procedure (directly selection). When I tried CTAS alternative, the disk space is run out when creating the intermediate table, because this intermediate table is stored on local PostgreSQL server, it takes huge storage when parsing point records and storing them in local table. By contract, querying on foreign table does not take the local storage and also the foreign table representing the huge amount of point clouds data does not need to be stored locally. This foreign data wrapper solution proves to perform well in terms of storage, and the cost time of executing query by foreign data wrapper is relevant to the area of querying region, despite of the scale of queried file system of datasets.

Data scale	Query level	Count (million)		Time (s)		Time PostGIS (s)
		Box	Polygon	Box	Polygon	
Small	Low	5	2	55	63	8
	Medium	76	42	952	1164	212
	High	49	49	718	891	173
Medium	Low	30	17	383	458	75
	Medium	137	122	1832	2224	392
	High	502	501	6399	8197	1798
Large	Low	203	132	2490	3145	655
	Medium	653	653	8108	11265	3157

Table 5.13: Time difference between rectangle and polygon selection

6 Conclusion

The aim of this research is to explore the possibility of using LiDAR data directly in PostgreSQL on file system by means of foreign data wrapper. This chapter will summarise the main result and give the answers to the main research question and sub questions. In addition the work that have not be achieved and can be carried out in the future will be discussed.

6.1 Conclusion

The proposed method of foreign data wrapper have been implemented and tested. It is necessary to review the research question raised at the beginning of the research.

To what extent we can use LiDAR point clouds directly in the PostgreSQL by means of FDW?

The conclusion can be drawn from the results and analysis, by means of foreign data wrapper, several operations supported by PostgreSQL can be executed on point clouds files that are still stored in the file system, residing outside the PostgreSQL. Multiple LiDAR point clouds are stored on file system, with a metadata file. By means of foreign data wrapper, they can be exposed and queried on foreign table in the PostgreSQL / PostGIS. (foreign table can be queried same as local table and join with local table) Spatial selection, attributes selection, data manipulation, aggregate function are possible.

- ***How does FDW build the connection between LiDAR file system and DBMS?***

  The access scheme is building connection based on the file path of LiDAR file system, then all the point clouds files stored on this file system can be accessed and then obtained and fetched to foreign table at the side of PostgreSQL. In this point clouds data management system, multiple LiDAR files with different file format and extent are collected and mixed in one LiDAR file system. Besides a metadata file is created to register the header information of these LiDAR files for pre-selection of relevant files. While the qualifiers in foreign data wrapper can conduct a filtering of qualified points before delivering point records. Thus the potential points are fetched to PostgreSQL on the foreign table, for the rectangle selection, the potential points are identical to the asked points; for the polygon selection the potential points are the points inside the bounding box of this polygon and PostgreSQL/PostGIS will select out the asked points and omit the points that are inside the bounding box but not in the polygon.

- ***What kinds of queries and what levels of selection can be executed upon LiDAR data ?***

  The operations includes data manipulation like update, delete and insert; spatial selection of different query regions including rectangles, circle and irregular polygons; query based on attributes; aggregation function like compute the highest, lowest and average height of a region, are feasible and work well by foreign data wrapper solution.

  In the scalability results, different sizes of tested regions can be queried on LiDAR data, time is relevant to the returned points. The level of selection i.e. the number of relevant files has no relevance to the querying time.

- ***What sizes of AHN3 datasets can work well by FDW solution?***

  Different scales of tested datasets can work well by FDW solution, if the storage space for the LiDAR file system is enough.

  However the scalability test shows that the querying time has no relevance to the scale of datasets but only the number of returned points.

  In addition to this, the feasibility test shows that if the large LiDAR datasets are stored on a file system even this datasets covers a province or the whole country, the query of a tiny geometry area like a building, a campus or a park, can be quickly down, since the time cost only for parsing the potential and asked points, which is only a really small part of the whole dataset.

- ***What are the benefits and problems when using FDW method?***

  The benefit of this hybrid solution combines the advantages of both file system and database management system, the data is still stored on file system where supports standardized and exchangeable formats and the data can be processed by the powerful file-based tools. While after representing on the database management system, the capabilities of data handling becomes more various with the support of SQL database management system. For example, the data content on PostgreSQL foreign table can be combined with other information and queries together with vector data that are already in the database; it is possible to connect front ends that understand how to communicate with the database already or understand the data types of the database e.g. QGIS might be able to visualize the query results. Thus this hybrid solution can save the storage of this kind of huge point clouds data, the querying also works well, but the foreign tables used by this solution can be indexed like normal PostgreSQL tables.

  The benefit of this point clouds data management system based on FDW method is that there is no need to load data content of LiDAR files into PostgreSQL and take storage, but just build connection to the file system, they can be used by DBMS features like query. Therefore, it is efficient to execute the query like selecting the points inside a building on the datasets of whole province. The problems can be it takes time to register header information into metadata file and retrieve the metadata file to search for relevant files.

6.2 Future work

If time and storage space permit, it is possible to do higher level and larger scale and even full-scaled benchmark. Additionally, the following aspects can be researched to improve the handling features of this point clouds data management system based on FDW methods.

- Metadata management

  In terms of handling header information of multiple LiDAR files, database management system can be a good alternative, because the LiDAR file system can have a large number of different files, and the header information of all these files need to be retrieved during the process of FDW, while the index can be built on the file extent column of metadata table to save the retrieving time. This alternative can be researched in the future work.

- Data display

  When fetching and representing the point records on the PostgreSQL foreign tables, it can be an more efficient way to use the data type PcPatch of PgPointcloud to organize and display the content of LiDAR data. If the Applications understand this data type, then points retrieved from the foreign data wrapper can be directly get to.

- Data organization

  Data organization is proved to be useful, more spatial access methods and underlying parameters can be researched to improve the feature of the foreign data wrapper solution.

- Comparison

  The hybrid solution of foreign data wrapper can be compared to the both solutions:

  - file system solution: use *las2las* to filter and query the points inside an area of interest.
  - DBMS solution: use PgPointcloud to store the LiDAR data in PostgreSQL.

Bibliography

ASPRS (2019). *LAS Specification 1.4 - R15*.

Baralis, E., Dalla Valle, A., Garza, P., Rossi, C., and Scullino, F. (2017). Sql versus nosql databases for geospatial applications. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 3388–3397. IEEE.

Boehm, J. (2014). File-centric organization of large lidar point clouds in a big data context. In *IQmulus First Workshop on Processing Large Geospatial Data*, pages 69–76.

Chen, Q. (2007). Airborne lidar data processing and information extraction. *Photogrammetric engineering and remote sensing*, 73(2):109.

Chrószcz, A., Łukasik, P., and Lupa, M. (2016). Analysis of performance and optimization of point cloud conversion in spatial databases. In *IOP Conference Series: Earth and Environmental Science*, volume 44.

Cura, R., Perret, J., and Paparoditis, N. (2015). Point cloud server (pcs): Point clouds in-base management and processing. *ISPRS Annals of Photogrammetry, Remote Sensing & Spatial Information Sciences*, 2.

Cura, R., Perret, J., and Paparoditis, N. (2017). A scalable and multi-purpose point cloud server (pcs) for easier and faster point cloud data management and processing. *ISPRS Journal of Photogrammetry and Remote Sensing*, 127:39–56.

David, N., Mallet, C., and Bretar, F. (2008). Library concept and design for lidar data processing. In *Proceedings of the GEOgraphic Object Based Image Analysis (GEOBIA) Conference, Calgary, AB, Canada*, volume 58.

Dunklau, R. and Mounier, F. (2017). *Multicorn Documentation Release 1.1.1*.

Guo, D. and Onstein, E. (2020). State-of-the-art geospatial information processing in nosql databases. *ISPRS International Journal of Geo-Information*, 9(5):331.

Höfle, B., Rutzinger, M., Geist, T., and Stötter, J. (2006). Using airborne laser scanning data in urban data management-set up of a flexible information system with open source components. In *Proceedings of UDMS*, volume 2006, page 25th.

Hug, C., Krzystek, P., and Fuchs, W. (2004). Advanced lidar data processing with lastools. In *XXth ISPRS Congress*, pages 12–23.

Isenburg, M. (2012a). Lasindex.

Isenburg, M. (2012b). Lasindex - spatial indexing of lidar data.

Isenburg, M. (2013). Laszip: lossless compression of lidar data. *Photogrammetric Engineering and Remote Sensing*, 79(2):209–217.

Janecka, K., Karki, S., van Oosterom, P., Zlatanova, S., Kalantari, M., and Ghawana, T. (2018). 3d cadastres best practices, chapter 4: 3d spatial dbms for 3d cadastres. In *26th FIG Congress 2018" Embracing our Smart World Where the Continents Connect*. International Federation of Surveyors (FIG).

Kozea (2014). Multicorn.

Kumar, A. Y., Noufia, M., Shahira, K., and Ramiya, A. (2019). Building information modelling of a multi storey building using terrestrial laser scanner and visualisation using potree: An open source point cloud renderer. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 42:421–426.

Melton, J., Michels, J.-E., Josifovski, V., Kulkarni, K., Schwarz, P., and Zeidenstein, K. (2001). Sql and management of external data. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(1):70–77. cited By 13.

Meyer, T. and Brunn, A. (2019). 3d point clouds in postgresql/postgis for applications in gis and geodesy. pages 154–163. cited By 0.

OSGeo (2019a). *PostGIS 3.0.1 dev Manual*.

OSGeo (2019b). Postgis feature list.

Otepka, J., Ghuffar, S., Waldhauser, C., Hochreiter, R., and Pfeifer, N. (2013). Georeferenced point clouds: A survey of features and point cloud management. *ISPRS International Journal of Geo-Information*, 2(4):1038–1065. cited By 40.

Ott, M. (2012). Towards storing point clouds in postgresql. *HSR Hochschule für Technik Rapperswil, Rapperswil, Switzerland*.

Pajić, V., Govedarica, M., and Amović, M. (2018). Model of point cloud data management system in big data paradigm. *ISPRS International Journal of Geo-Information*, 7(7):265.

PostgreSQL wiki (2020). Foreign data wrappers. [Online; accessed 11-Aug-2004].

Ramsey, P., Blottiere, P., Brédif, M., Lemoine, E., et al. (2019). Pgpointcloud-a postgresql extension for storing point cloud (lidar) data.

Ramsey, P., Blottiere, P., Brédif, M., Lemoine, E., et al. (2020). Pgpointcloud-a postgresql extension for storing point cloud (lidar) data.

rapidlasso (2019). Lastools.

Rieg, L., Wichmann, V., Rutzinger, M., Sailer, R., Geist, T., and Stötter, J. (2014). Data infrastructure for multitemporal airborne lidar point cloud analysis - examples from physical geography in high mountain environments. cited By 30.

Roijackers, J., Fletcher, D. G. H. L., and Serebrenik, D. A. (2012). Bridging sql and nosql.

Samberg, A. (2007). An implementation of the asprs las standard. In *ISPRS Workshop on Laser Scanning and SilviLaser*, pages 363–372. Citeseer.

Stanisavljevic, V. (2019). Comparison of data aggregation from a wireless network of sensors using database sharding and foreign data wrappers. pages 247–250. cited By 0.

The PostgreSQL Global Development Group (2019). *PostgreSQL Documentation*.

Van Oosterom, P., Martinez-Rubi, O., Ivanova, M., Horhammer, M., Geringer, D., Ravada, S., Tijssen, T., Kodde, M., and Gonçalves, R. (2015). Massive point cloud data management: Design, implementation and execution of a point cloud benchmark. *Computers and Graphics*, 49:92–125.

van Oosterom, P., Martinez-Rubi, O., Tijssen, T., and Gonçalves, R. (2017). Realistic benchmarks for point cloud data management systems. In *Advances in 3D Geoinformation*, pages 1–30. Springer.

Van Oosterom, P., MEIJERS, M., VERBREE, E., LIU, H., and TIJSSEN, T. (2019). Towards a relational database space filling curve (sfc) interface specification for managing nd-pointclouds.

Wijga-Hoefsloot, M. (2012). Point clouds in a database: Data management within an engineering company.

Zlatanova, S. (2006). 3d geometries in spatial dbms. In *Innovations in 3D geo information systems*, pages 1–14. Springer.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class `scrbook`. The main font is Palatino.