Massive point clouds for eSciences

Medium benchmark

O. Martinez-Rubi
Netherlands eScience Center,
Science Park 140 (Matrix 1), 1098 XG Amsterdam, the Netherlands

October 28, 2014

Contents

1 Overview **2**

2 Introduction **2**
2.1 Datasets . . . 2
2.2 Point Cloud Data Management systems . . . 3
2.3 Hardware . . . 4
2.4 Data distribution . . . 4

3 Software versions **4**

4 General PCDMS description **5**
4.1 PostgreSQL flat . . . 6
4.2 PostgreSQL blocks . . . 6
4.3 Oracle flat . . . 6
4.4 Oracle blocks . . . 7
4.5 MonetDB flat . . . 7
4.6 LAStools . . . 7

5 Functionalities **9**

6 Loading/Preparation scripts **11**
6.1 PostgreSQL flat . . . 11
6.2 PostgreSQL blocks . . . 11
6.3 Oracle flat . . . 15
6.4 Oracle blocks . . . 15
6.5 MonetDB flat . . . 17
6.6 LAStools . . . 18

7 Querying scripts **19**
7.1 PostgreSQL flat . . . 19
7.2 PostgreSQL blocks . . . 20
7.3 Oracle flat . . . 20
7.4 Oracle blocks . . . 21
7.5 MonetDB flat . . . 22
7.6 LAStools . . . 22

1 Overview

The work presented in this report is part of the project Massive point clouds for eSciences. The main goal of this project is to develop an infrastructure for the storage, the management, the analysis and processing, the dissemination, the visualisation and the manipulation of massive point clouds.

This document contains the description of a benchmark called medium-benchmark focussed on the storage and management of the data. We compare the loading and querying of different datasets in several Point Cloud Data Management Systems (PCDMS) including Oracle, PostgreSQL, MonetDB and LAStools. The set of functionalities (queries) that is tested is a sample of the main functionalities compiled in the User Requirements document[1]. However, the aim of the project is to provide a system that can offer a good performance with all the requirements presented in that document.

The outline of this report is as follows: Section 2 describes the used datasets, introduces the tested PCDMSs and details the used hardware. In section 3 we detail the exact software that has been used. Section 4 contains a deeper description of the PCDMSs while in section 5 we present the queries that have been executed. Section 6 and 7 contains the loading and querying scripts.

2 Introduction

2.1 Datasets

In table 1 we detail the used datasets, all of them are parts of the Actueel Hoogtebestand Nederland 2 (AHN2) dataset, its reference system (EPSG or SRID) is 28992 and each dataset contains the previous smaller one.

Name	Points	LAS files	Disk size	Area	Description
20M	20,165,862	1	385 MB	1.25 km2	TU Delft campus
210M	210,631,597	17	4018 MB	11.25 km2	Major part of Delft city
2201M	2,201,135,689	153	41984 MB	125 km2	City of Delft and surroundings
23090M	23,090,482,455	1492	440420 MB	2000 km2	Major part of Zuid-Holland province

Table 1: Datasets description

In figure 1 we depict the approximated projection of the extent of the used datasets.

[1] http://www.gdmc.nl:8080/mpc/documents/reports/user-requirements-massive-point-clouds-for-esciences/at_download/file

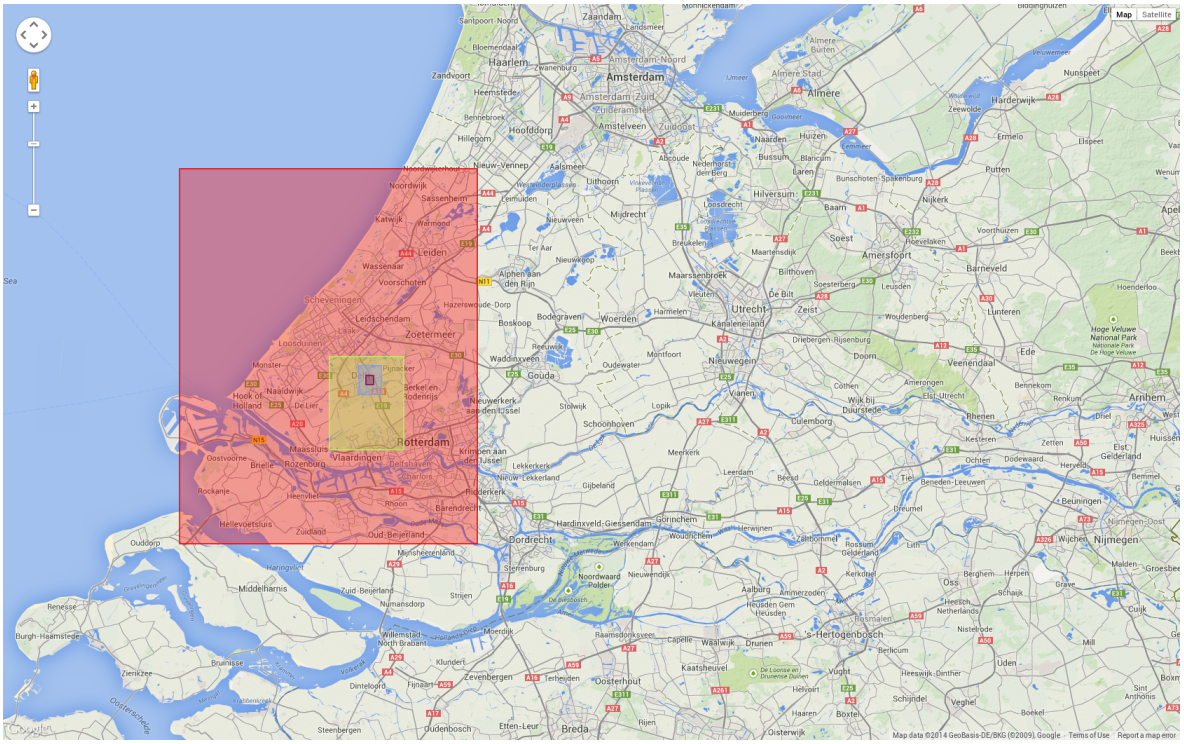

Figure 1: Approximated projection of the extent of the used datasets in Google Maps. Purple area is for 20M dataset, cyan area is for 210M dataset, green area is for 2201M dataset and red area is for 23090M dataset.

2.2 Point Cloud Data Management systems

We have tested six different PCDMSs, five are using relational database management systems (RDBMS) and they are compared with a file-based system based on the LAStools set of tools. In table 2 we present the tested systems.

PCDMS name	Software	Software description	PCDMS description
pf	PostgreSQL	Open-source RDBMS	points in flat table
pb	PostgreSQL	Open-source RDBMS	points in blocks (PointCloud extension)
of	Oracle	Commercial RDBMS	points in flat table
ob	Oracle	Commercial RDBMS	points in blocks (Oracle Spatial, SDO_PC)
mf	MonetDB	Open-source column-store	points in flat table
lt	LAStools	Commercial set of tools	file-based solution (with DB help)

Table 2: Point Cloud Data Management Systems description

2.3 Hardware

For this test we have used a server with the following details:

- HP DL380p Gen8 server
  - 2 x 8-core Intel Xeon processors, E5-2690 at 2.9 GHz
  - 128 GB main memory
  - RHEL 6 operating system
- Disk storage (direct attached)
  - 400 GB SSD
  - 5 TB SAS 15K rpm in RAID 5 configuration (internal)
  - 2 x 41 TB SATA 7200 rpm in RAID 6 configuration (in Yotta disk cabinet)

2.4 Data distribution

In order to guarantee the fastest data loading process we have split the SATA disks in two sets, one is used to store the raw data, i.e. the LAS files, and the other one contains the bulk of all the database data and the file-based structure in the case of LAStools.

In addition, in the case of Oracle and PostgreSQL we use the SSD disks to store the system data, i.e. the catalogs, logs, etc. We use the faster SAS disks to store temporal data. The bulk of data is stored in the first SATA RAID 6 set while the indexes are stored in the second set (in fact, they are stored in the same set than the raw data). In this way we guarantee that heavy reading/writing to same disk is minimized. This data distribution is done using different tablespaces for the different types of data.

Note that in the case of MonetDB, the index structures are always kept in memory so the faster devices are not used.

3 Software versions

The versions of the different software used for this test are:

- libLAS 1.7.0, laszip 2.1.0
- PostgreSQL: PostgreSQL 9.3.2, PostGIS 2.1.1, libgeotiff 1.4.0, GDAL 1.10.1 and PDAL (repository: https://github.com/pramsey/PDAL.git,
  commit: 50ed6d4296170ec32c35e27a308a6a6d7ce1c9c9)
- Oracle: Oracle Database 12c Enterprise Edition Release 12.1.0.1.0 - 64bit Production, with patch provided by Mike Horhammer on the 24th July 2014
- MonetDB: version downloaded (from the trunk repository) and compiled on the 20th June 2014
- LAStools: version downloaded on the 15th July 2014
- lasnelsc (binary converters from LAS/LAZ to PostgreSQL and MonetDB): version downloaded and compiled on 30th June 2014

4 General PCDMS description

As previously stated we have tested six different PCDMSs. The goal of these systems is to provide a manner of querying the point cloud data. For each PCDMS we have tried many different options and the ones that we present here are the ones we considered optimal.

There are three PCDMSs using flat tables (PostgreSQL flat, Oracle flat and MonetDB flat) where points are loaded in a regular flat table in a DB, i.e. one row per point. There are two PCDMSs (PostgreSQL blocks and Oracle blocks) that group the points in blocks which are stored in a DB, i.e. one row per block. In this case the data have to re-ordered as part of the blocking process so there is some spatial data organization. The last PCDMS (LAStools) is using a file-based set of tools that directly reorganize and index the LAS files. In this case we also use a DB to help during the queries so this PCDMS can be considered an hybrid.

In order to do a fair comparison we decided that all the PCDMSs must meet the following requirements:

- The same input files and formats will be used. All input data is given in LAS format (even though all the used PCDMS are compatible with LAZ format as well)
- About compression, if the query responses are not much affected by compression we will use it. In particular:
  - In the blocks-based PCDMSs the data will be stored with compression. We choose this because from previous test we found out that in PostgreSQL the compression offers 2.5 better storage while only degrading 10% the query responses. In Oracle using compression offers 2 times better storage and same query response times.
  - In the flat tables we store the data without compression. X,Y,Z are stored as double precision numbers (NUMBER type in the case of Oracle).
  - In the LAStools PCDMS we will store the data in LAS format. In previous tests we found out that even though LAZ files obviously offer much better compression (up to 12x) the queries to LAS files are much faster (up to 8x) that queries to LAZ files so we felt like the compression did not pay off the query downgrade.
- In the PCDMSs using blocks their size is configured to help each PCDMS. In particular:
  - In previous tests in PostgreSQL we found out that it benefits from smaller block sizes (compared to Oracle blocks) but we did not want to end up having many blocks so we choose a value of 3000 points per block.
  - In previous test in Oracle blocks we found out that this system works better with larger blocks (when compared to PostgreSQL blocks) but we did not want to have too large blocks. We choose a size of 10000 points per block.
- Only the 3 spatial dimensions will be stored in the databases solutions.
- In the loading/preparation we will use multiple processes. In some cases it is natively supported while in some others we had to do the parallelization out-of-the-core.
- When querying, the results will be written in a database table, one row for each queried point. We will use the SQL statement *CREATE TABLE [name] AS SELECT*. For MonetDB this does not work so we have to separated CREATE and INSERT statements. In the case of LAStools the results of each query will be written in a LAS file. Note that the

PCDMSs using blocks are specially affected by this requirement because the blocks need to be unpacked (which obviously requires time)

- In all the cases we will present the query results with a "cold" environment (when no data is in memory / OS or DB cache) and with a "hot" environment (when data has been already queried and it is in memory / OS or DB cache).

4.1 PostgreSQL flat

In this case the points are loaded in a normal table, i.e. there is one row per point.

We have developed a tool to convert a LAS/LAZ file into a PostgreSQL binary dump file. This is the fastest way of loading the point cloud data (up to five times faster than regular ASCII loading). The loading of the several LAS files is done independently. PostgreSQL does not provide parallel loading tools but it is possible to have several processes writing in the same table so we created a loading script that can process (convert and load) multiple files in parallel.

After all the data is loaded we create a B-Tree index on X and Y. We also tested using a GIST index but its creation was much more expensive and in the case of simple queries (rectangles and circles) it was offering a worse performance than the B-Tree indexing. Only for complex queries using the GIST index is worthy but even in this case it is not much better.

The creation of the index, contrary to other databases, does not happen in parallel (we could have used partitions and indexed them separately but that was out of the scope of this test)

Like for the loading, PostgreSQL does not have native parallel queries.

4.2 PostgreSQL blocks

In this PCDMS we use the point cloud extension developed by P. Ramsey (https://github.com/pramsey/pointcloud) on a PostgreSQL database. It creates blocks of points and stores each block in a row of a table using the *pcpatch* data type. The blocks are defined in the 2D (X and Y) space.

The creation of the blocks and loading is done by an external tool called PDAL (http://www.pdal.io/).

Like in the flat table case, PostgreSQL/PDAL are single-core processes but we use a script that can process multiple LAS files in parallel and independently (the obvious drawback of this method is that it requires a good input file organization, i.e. if the input LAS files are overlapping this method will also generate overlapping blocks).

There is not native parallel querying in this approach.

4.3 Oracle flat

In this case the points are loaded in a normal table, i.e. there is one row per point.

We considered using the Index Organize Table (IOT) feature of Oracle since it decreases considerable (around 60%) the required storage and provides faster queries (thanks to the data clustering) but the native parallelization of Oracle was not working in our test with an IOT.

For the loading we used an external table loader with a pre-processor script that uses the las2txt tool (of the liblas library). This method natively supports multiple processes.

Once the data is loaded we create a B-Tree index which is also done in parallel.

4.4 Oracle blocks

This PCDMS groups the points in blocks (defined in the 2D space). Each block is stored in a *SDO_PC_BLK* object (the points are stored in a BLOB). The point cloud metadata (including the blocks extent) is stored in a separate object, the *SDO_PC*, also in a separate table.

Like in the Oracle flat case we use the external table loader to load all the points in a staging table. The pre-processor script also computes and delivers the Hilbert value for each point and this is also stored in the staging table. The staging table is a IOT table which has as a primary key the Hilbert value.

When all the points are loaded, the staging table is used by the blocking process to create the blocks. The chosen blocking method is the new Hilbert R-Tree blocking which creates blocks of points where their Hilbert values are similar.

Note that contrary to PostgreSQL this method requires all the points to be loaded in the staging area. However, a incremental loading is being developed and could also be used in a future release.

4.5 MonetDB flat

MonetDB is a column-store RDBMS and each column is stored separately.

We have developed a binary converter that converts a set of LAS files to the MonetDB native format. The converter works in parallel. Because of memory limitations the sets should not be too large. If they are too large it is recommended to split them in chunks and load them sequentially (each chunk is parallelized though)

MonetDB uses a novel indexing strategy called Imprints that will create index structures on-the-fly when the columns are queried. Such index structures are kept in memory.

4.6 LAStools

We use a combination of LAStools tools together with a PostgreSQL database to create a PCDMS.

LAStools is a set of tools developed and maintained by Rapidlasso GmbH (http://rapidlasso.com/).

First we use lassort to reorder the points of each input file. Then we use lasindex to create an index file (based on Quad-Tree) that will be used when doing queries. Finally we insert in a PostgreSQL database the extent of each file (this step can also be done without databases, using for example shapely Python module)

All these operations are parallelized for the different files using a Python script.

The tools can also work with the LAZ format but the best performance in data preparation and querying is when the LAS format is used.

For the querying we use the tools lasmerge for rectangles and circles and lasclip for the rest of geometries. Before executing these tools we do a database query to know which files should be used and only these ones are given as input to the LAStools tools.

Note that from the used tools, lasindex and lasmerge are free but lassort and lasclip are commercial tools. They can be freely used but they introduce some noise in the points. In the case of lassort the same functionality could probably be obtained by using a free non-compiled tool in LAStools called lasoptimize but it has not been used in this test because it is not officially included yet in the LAStools binary folder. Probably it will be included in future versions.

lasclip has an option to use multiple processes (limited to 2 cores with the unlicensed version) but when tested it did not really improved so only one processes is used in lasclip (and lasmerge).

Note that, like in happens in PostgreSQL, this system will work nicely when the input data has some good spatial organization, i.e. the spatial extent of the files has none or very small overlapping. If that is not the case a previous re-organization of the data should be done with the commercial tool lastile.

5 Functionalities

The queries that we have executed are detailed in table 3 where *Pnts* is the number of points in the boundary of the query geometry, *Dset* is dataset in which the query geometry is located (they are cumulative) and *Area* is in km^2.

For all the PCDMSs using databases we do a CTAS (create table as select) so for each query we write the points in the area in a new table. In the case of LAStools we write the points in the query regions in LAS files.

Note that in PostgreSQL blocks and Oracle blocks the blocks need to be unpacked (both for checking individual points and for delivering the final result).

ID	Key	Pnts	Dset	Area	Type	Description
1	S_RCT	5	20M	0.0027	polygon	Small axis-aligned rectangle
2	M_RCT	5	20M	0.0495	polygon	Medium axis-aligned rectangle
3	S_CRC	97	20M	0.0013	polygon	Small circular polygon, radius 20 m
4	M_CRC	379	20M	0.0415	polygon	Medium circular polygon, radius 115 m
5	S_SIM	9	20M	0.0088	polygon	Small, simple polygon
6	M_COM_o	792	20M	0.0252	polygon	Medium, complex polygon, 1 hole
7	M_DG_RCT	5	20M	0.0027	polygon	Medium, narrow, diagonal rectangular area
8	L_COM_os	89	210M	0.1341	polygon	Large, complex polygon, 2 holes
9	S_L_BUF	94	210M	0.0213	polygon	Small polygon (10 m buffer around line of 11 pts)
10	S_RCT_UZ	5	210M	0.0021	polygon	Small axis-aligned rectangle, cut in maximum elevation
11	S_RCT_LZ	5	210M	0.0051	polygon	Small axis-aligned rectangle, cut in minimum elevation
12	L_RCT_LZ	5	210M	0.1419	polygon	Large axis-aligned rectangle, cut in minimum elevation
13	L_L_BUF	237	2201M	0.0475	polygon	Large polygon (1 m buffer around line of 61 pts)
14	L_DG_L_BUF	39	2201M	0.0499	polygon	Large polygon (2 m buffer around diagonal line of 8 pts)
15	L_RCT	5	23090M	0.2342	polygon	Large axis-aligned rectangle
16	L_RCT_N	5	23090M	0.1366	polygon	Large axis-aligned rectangle in empty area
17	L_CRC	93	23090M	0.1256	polygon	Large circular polygon
18	NN_1000	1	23090M	0.0	point	Point for NN query, 1000 nearest points
19	NN_5000	1	23090M	0.0	point	Point for NN query, 5000 nearest points
20	NN_1000_w	1	23090M	0.0	point	Point in water for NN query, 1000 nearest points

Table 3: Description of the different queries

In all the cases the usual approach to solve the queries is by first doing a pre-selection of points based on the bounding box of the query region and then check if each of the preselected points lay in the query region. This procedure is, in the PCDMSs using blocks and LAStools,

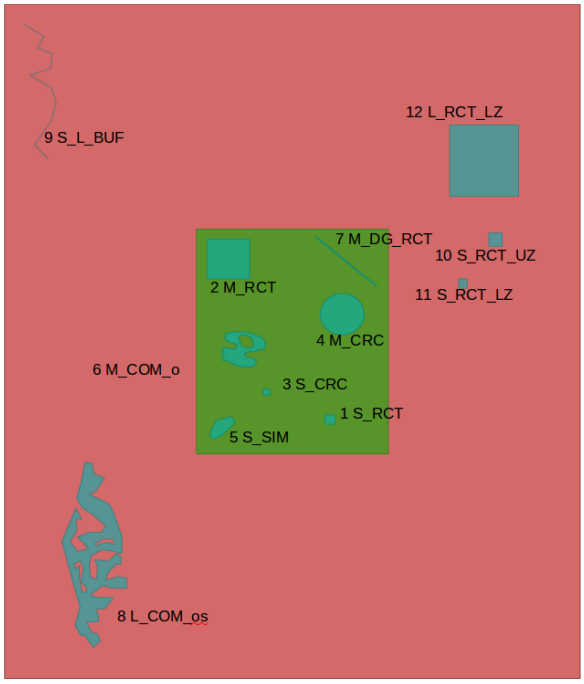

Figure 2: Queries used in the medium benchmark (up to 210M extent)

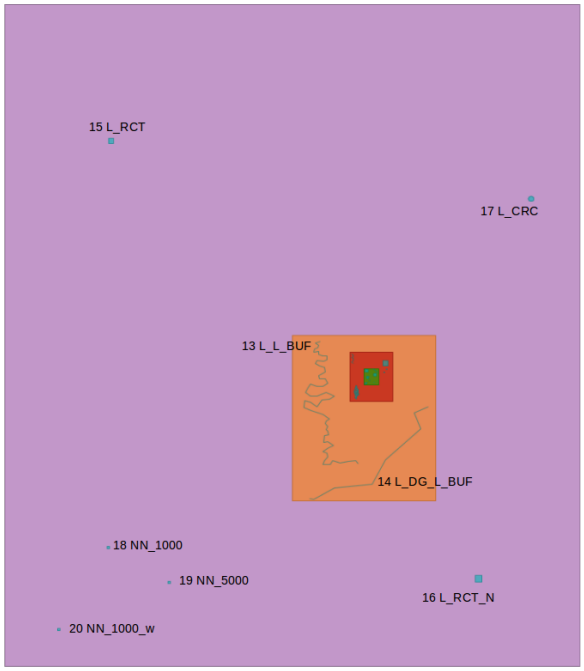

Figure 3: Queries used in the medium benchmark

automatically done and transparent to the user but in the case of the flat tables we have to do it manually. Obviously in the case of rectangles the second filtering is not necessary. In the case of circles we do the distance method in the case of flat tables and LAStools and we use an approximated polygon in the case of blocks. The last 3 queries are nearest neighbour computations. In the case of the flat tables we do an approximation algorithm that previously preselects points given a certain distance to the point (10m, 20m and 100m respectively).

6 Loading/Preparation scripts

We have developed a set of Python scripts for the loading/preparation of the data. In the following sections we detail the steps for he loading/preparation of the data for the different PCDMS. In all the cases the CPU, memory and disks usage has been monitored.

6.1 PostgreSQL flat

- Load the required PostGIS extension and create the flat table (SQL):

```
CREATE EXTENSION postgis;
CREATE TABLE ahn_flat (x DOUBLE PRECISION,
                       y DOUBLE PRECISION,
                       z DOUBLE PRECISION) TABLESPACE users;
```

  Note that we use the table space *users* which is mounted in a file system different to the one that contains the LAS files.

- Parallel processing (converting and loading) of files (Command-line, executed in parallel using Python multiprocessing module, example for certain file):

```
las2pg -s /pak2/usrdata/ahn2/benchmarks/210M/tileslas/tile_84000_446000.las \
--stdout --parse xyz | \
psql [connection parameters] -c "copy ahn_flat from stdin with binary"'
```

- Create the B-Tree index (SQL):

```
CREATE INDEX ahn_flat_xy_btree_idx on ahn_flat (x,y)
WITH (FILLFACTOR=99) TABLESPACE indx;
```

  Note that the index creation can not be done in parallel in PostgreSQL. Also note that we use the table space *indx* for the index. This one is set to sue the same file system that contains the LAS files but different to the one that contains the main table. In this way creating the index will be faster.

6.2 PostgreSQL blocks

- Load the required extensions and create the table that will contain the blocks (SQL):

```
CREATE EXTENSION postgis;
CREATE EXTENSION pointcloud;
CREATE EXTENSION pointcloud_postgis;
CREATE TABLE patches (
            id SERIAL PRIMARY KEY,
            pa PCPATCH)TABLESPACE users ;
```

  Note that we use the table space *users* which is mounted in a file system different to the one that contains the LAS files.

- In order to load point clouds in the PostgreSQL blocks PCDMS we need to specify the format of the data. This is a XML text that contains the sizes, scales and offsets of the different attributes. For each input file (LAS/LAZ) it is mandatory to have its format in the *pointcloud_formats* table. Every time there is an new file with a different set of scales and offsets (than previously loaded files) we need to add a new format. To add a new format (SQL):

```
INSERT INTO pointcloud_formats (pcid, srid, schema) VALUES (1, [SRID],'
<?xml version="1.0" encoding="UTF-8"?>
<pc:PointCloudSchema xmlns:pc="http://pointcloud.org/schemas/PC/1.1"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <pc:dimension>
        <pc:position>1</pc:position>
        <pc:size>4</pc:size>
        <pc:description>X coordinate as a long integer.
        You must use the scale and offset information of
        the header to determine the double value.
        </pc:description>
        <pc:name>X</pc:name>
        <pc:interpretation>int32_t</pc:interpretation>
        <pc:scale> 0.01  </pc:scale>
        <pc:offset> 0 </pc:offset>
    </pc:dimension>
    <pc:dimension>
        <pc:position>2</pc:position>
        <pc:size>4</pc:size>
        <pc:description>Y coordinate as a long integer.
        You must use the scale and offset information of
        the header to determine the double value.
        </pc:description>
        <pc:name>Y</pc:name>
        <pc:interpretation>int32_t</pc:interpretation>
        <pc:scale> 0.01 </pc:scale>
        <pc:offset> 0 </pc:offset>
    </pc:dimension>
    <pc:dimension>
        <pc:position>3</pc:position>
        <pc:size>4</pc:size>
        <pc:description>Z coordinate as a long integer.
        You must use the scale and offset information of
        the header to determine the double value.
        </pc:description>
        <pc:name>Z</pc:name>
        <pc:interpretation>int32_t</pc:interpretation>
        <pc:scale> 0.01 </pc:scale>
        <pc:offset> 0 </pc:offset>
    </pc:dimension>
    <pc:metadata>
```

```
        <Metadata name="compression">dimensional</Metadata>
    </pc:metadata>
</pc:PointCloudSchema>
');
```

Note that we use dimensional compression.

Then, we need to use PDAL tool to load the data from the input file (Command-line, executed in parallel using Python multiprocessing module):

```
pdal pipeline [xmlFile]
```

Where the XML file tells PDAL what to do and it contains:

```
<?xml version="1.0" encoding="utf-8"?>
<Pipeline version="1.0">
    <Writer type="drivers.pgpointcloud.writer">
        <Option name="connection">host='[DB host]' dbname='[DB name]'
        password='[password]' user='[DB user]'</Option>
        <Option name="table">blocks</Option>
        <Option name="srid">[SRID]</Option>
        <Option name="overwrite">false</Option>
        <Option name="pcid">1</Option>
        <Filter type="filters.chipper">
            <Option name="capacity">3000</Option>
            <Filter type="filters.cache">
                <Option name="max_cache_blocks">1</Option>
                <Filter type="filters.selector">
                    <Option name="keep">
                        <Options>
                            <Option name="dimension">X</Option>
                            <Option name="dimension">Y</Option>
                            <Option name="dimension">Z</Option>
                        </Options>
                    </Option>
                    <Option name="overwrite_existing_dimensions">false</Option>
                    <Reader type="drivers.las.reader">
                        <Option name="filename">[input file path]</Option>
                        <Option name="spatialreference">EPSG:28992</Option>
                    </Reader>
                </Filter>
            </Filter>
        </Filter>
    </Writer>
</Pipeline>
```

Note that the *pcid* is the *formatID*, i.e. 1. Note that we use a block size of 3000 and we only load X, Y and Z (we need to add a filter to only use the X, Y and Z columns).

- Create a PostGIS GIST index on the blocks to ease the querying (SQL):

  ```
  CREATE INDEX pa_gix ON blocks USING GIST (geometry(pa)) TABLESPACE indx;
  ```

  Note that we use the table space *indx* for the index. This one is set to use the same file system that contains the LAS files but different to the one that contains the main table. In this way creating the index will be faster.

- Run *VACUUM* and *ANALYZE* (SQL).

  ```
  VACUUM FULL ANALYZE blocks;
  ```

6.3 Oracle flat

- We create an external table that will use a preprocessor that convert the LAS into an ASCII stream. This preprocessor is also computing the Hilbert value for each point which is not used in the flat-table-based PCDMS so a faster loading would be possible if we would use a preprocessor that does not do extra work when not required. We define the external table with parallel 16 (SQL):

```
CREATE TABLE EXT_AHN_FLAT (VAL_D1 NUMBER,VAL_D2 NUMBER,VAL_D3 NUMBER)
ORGANIZATION EXTERNAL(
    TYPE ORACLE_LOADER
    DEFAULT DIRECTORY OF210M_LAS_DIR
    ACCESS PARAMETERS (
        RECORDS DELIMITED BY NEWLINE
        PREPROCESSOR EXE_DIR:'HILBERT_PREP.SH'
        BADFILE LOG_DIR:'HILBERT_PREP_%P.BAD'
        LOGFILE LOG_DIR:'HILBERT_PREP_%P.LOG'
        FIELDS TERMINATED BY ',')
    LOCATION ('*.LAS')) PARALLEL 16  REJECT LIMIT 0;
```

  The preprocessor file uses the class *Las2SqlLdr* in the package *oracle.spatial.util* of the JAR file sdoutl.jar.

- We create the flat table from selecting from the external table and it is also parallelized using the available cores (SQL).

```
CREATE TABLE AHN_FLAT TABLESPACE USERS PCTFREE 0 NOLOGGING
 PARALLEL 16 AS SELECT VAL_D1,VAL_D2,VAL_D3 FROM EXT_AHN_FLAT;
```

  Like in PostgreSQL we use the users tablespace which is mounted in a file system different to the one that contains the LAS files.

- Finally we create a B-Tree index on X and Y which can also be parallelized (SQL):

```
CREATE INDEX AHN_FLAT_IDX ON AHN_FLAT (VAL_D1,VAL_D2)
   TABLESPACE INDX PCTFREE 0 NOLOGGING  PARALLEL 16;
```

  Note that we use the table space *indx* for the index. This one is set to use the same file system that contains the LAS files but different to the one that contains the main table. In this way creating the index will be faster.

6.4 Oracle blocks

- In this case we need a staging table that will contain all the points from all the files. For this we also use a external table but this one must also contain the Hilbert value that is precomputed by the preprocessor file (SQL).

```
CREATE TABLE EXT_AHN_STAGING (VAL_D1 BINARY_DOUBLE,VAL_D2 BINARY_DOUBLE,
                              VAL_D3 BINARY_DOUBLE,D NUMBER)
ORGANIZATION EXTERNAL(
    TYPE ORACLE_LOADER
    DEFAULT DIRECTORY OB210M_LAS_DIR
    ACCESS PARAMETERS (
        RECORDS DELIMITED BY NEWLINE
        PREPROCESSOR EXE_DIR:'HILBERT_PREP.SH'
        BADFILE LOG_DIR:'HILBERT_PREP_%P.BAD'
        LOGFILE LOG_DIR:'HILBERT_PREP_%P.LOG'
        FIELDS TERMINATED BY ',')
    LOCATION ('*.LAS')) PARALLEL 16  REJECT LIMIT 0;
```

- We create the tables that will contain the blocks and the table with meta-data information of the point cloud (SQL).

```
 CREATE TABLE AHN_BLCK TABLESPACE USERS PCTFREE 0 NOLOGGING
  LOB(POINTS) STORE AS SECUREFILE (TABLESPACE USERS COMPRESS HIGH)
AS SELECT * FROM MDSYS.SDO_PC_BLK_TABLE WHERE 0 = 1;

CREATE TABLE AHN_BASE (PC SDO_PC)
  TABLESPACE USERS PCTFREE 0 NOLOGGING;
```

  Like in Oracle flat and PostgreSQL cases we use the users table space

- We create the stating table which is an IOT table from selecting the points from the external table. In Oracle IOTs the primary key (that defines the clustering of the data) must be unique so we need to add a re-factoring to the Hilbert value in order to make it unique. We add a hint to parallelize the process using all the available cores (SQL).

```
CREATE TABLE AHN_STAGING (VAL_D1,VAL_D2,VAL_D3,D,
    CONSTRAINT AHN_STAGING_PK PRIMARY KEY (D))
    ORGANIZATION INDEX
    TABLESPACE USERS PCTFREE 0 NOLOGGING PARALLEL 16
AS
    SELECT  VAL_D1,VAL_D2,VAL_D3,
    D+(ROWNUM*0.0000000001) D FROM EXT_AHN_STAGING;
```

- After the staging IOT table is ready we create the blocks using the Hilbert R-Tree method (SQL).

```
DECLARE
    PTCLD      SDO_PC;
    PTN_PARAMS VARCHAR2(80) := 'BLK_CAPACITY=10000';
    EXTENT     SDO_GEOMETRY := SDO_GEOMETRY(2003,28992,NULL,
       SDO_ELEM_INFO_ARRAY(1,1003,3),
```

```
       SDO_ORDINATE_ARRAY(84000.00,445000.00,87000.00,448750.00));
    OTHER_ATTRS XMLTYPE     := XMLTYPE('
        <OPC:SDOPCOBJECTMETADATA
           XMLNS:OPC="HTTP://XMLNS.ORACLE.COM/SPATIAL/VIS3D/2011/SDOVIS3D.XSD"
           XMLNS:XSI="HTTP://WWW.W3.ORG/2001/XMLSCHEMA-INSTANCE"
           BLOCKINGMETHOD="HILBERT R-TREE">
        </OPC:SDOPCOBJECTMETADATA>');
BEGIN
    PTCLD := SDO_PC_PKG.INIT ('AHN_BASE', 'PC', 'AHN_BLCK', PTN_PARAMS,
        EXTENT, 0.0001, 3, NULL, NULL, OTHER_ATTRS);
    INSERT INTO AHN_BASE VALUES (PTCLD);
    COMMIT;
    SDO_PC_PKG.CREATE_PC (PTCLD, 'AHN_STAGING', NULL);
END;
```

- Update the SRID of the blocks, drop the staging table and add the blocks index (SQL).

```
UPDATE AHN_BLCK B SET B.BLK_EXTENT.SDO_SRID = 28992;

DROP TABLE AHN_STAGING;

INSERT INTO USER_SDO_GEOM_METADATA VALUES ('AHN_BLCK','BLK_EXTENT',
   SDO_DIM_ARRAY(SDO_DIM_ELEMENT('X',84000.00,87000.00,0.0001),
   SDO_DIM_ELEMENT('Y',445000.00,448750.00,0.0001)),28992);

CREATE INDEX AHN_BLCK_SIDX ON AHN_BLCK (BLK_EXTENT) INDEXTYPE IS
   MDSYS.SPATIAL_INDEX PARAMETERS ('TABLESPACE=INDX WORK_TABLESPACE=PCWORK
   LAYER_GTYPE=POLYGON SDO_INDX_DIMS=2 SDO_RTR_PCTFREE=0') PARALLEL 16;
```

Note that we use the *indx* table space which is mounted in a different file system than the main blocks table.

6.5 MonetDB flat

- Create the flat table (SQL):

```
CREATE TABLE ahn_flat (x DOUBLE PRECISION,y DOUBLE PRECISION,
z DOUBLE PRECISION);
```

- We execute the MonetDB binary converter that convert LAS/LAZ files into the exact files that are used to store the columnar data in MonetDB (command line)

```
las2col -i /pak2/usrdata/ahn2/benchmarks/210M/tileslas/tile_84000_446000.las
 -i ... /pak1/mondata/tmp/0_tempFile --parse xyz
```

The output files are written in a different file system than the input files (to maximize IO).

- We make MonetDB aware of the data (SQL):

```
COPY BINARY INTO ahn_flat from ('/pak1/mondata//tmp/0_tempFile_col_x.dat',
'/pak1/mondata//tmp/0_tempFile_col_y.dat',
'/pak1/mondata//tmp/0_tempFile_col_z.dat');
```

  The files still need to be read by MonetDB and they can be quite large so this process can take a while. In the case of a large number of input files (more than few hundred) we recommend splitting the loading in chunks.

- Since the Imprints index in MonetDB are automatically created in the first query, we force the Imprints creation (on X and Y) by doing a rectangle query in the center of the extent (even though query between 0 and 1 would also do the trick) (SQL)

```
select * from ahn_flat where x between 85400.0 and 85600.0 and
                             y between 446775.0 and 446975.0;
```

6.6 LAStools

- Each file is resorted and indexed. (Command-line, executed in parallel using Python multiprocessing module, example for certain file):

```
lassort.exe -i \
 /pak2/usrdata/ahn2/benchmarks/210M/tileslas/tile_84000_446000.las \
 -o /pak1/ltdata/nlt210M/tile_84000_446000.las

lasindex -i /pak1/ltdata/nlt210M/tile_84000_446000.las
```

- We create a DB with PostGIS extension and a table to hold the extent of the several files (SQL).

```
CREATE EXTENSION postgis;
CREATE TABLE lasindex (
       id integer,
       filepath text,
       num integer,
       scalex double precision,
       scaley double precision,
       scalez double precision,
       offsetx double precision,
       offsety double precision,
       offsetz double precision,
       geom public.geometry(Geometry,28992));
```

- Using lasinfo we extract the header and insert the information in the DB. We do this by getting the output of lasinfo and converting it into an INSERT statement to the DB (Command-line and SQL, executed in parallel using Python multiprocessing module, example for certain file):

```
lasinfo /pak1/ltdata/nlt210M/tile_84000_446000.las -nc -nv -nco

INSERT INTO lasindex
(id,filepath,num,scalex,scaley,scalez,offsetx,offsety,offsetz,geom)
VALUES
(0, '/pak1/ltdata/nlt210M/tile_84000_446000.las', 17004733, 0.01, 0.01, 0.01,
 0.0, 0.0, 0.0, ST_MakeEnvelope(84000.0, 465000.0, 84999.990000000005,
  446999.98999999999, 28992));
```

- Create the GIST index on the extents. We also vacuum and analyze the table.

```
create index ON lasindex using GIST (geom) ;
VACUUM FULL ANALYZE lasindex;
```

7 Querying scripts

We have developed a set of Python scripts to execute the several queries. In the database solutions the different query regions, which are provided as WKT, are loaded in a table called QUERY_POLYGONS. In all the cases the CPU, memory and disks usage has been monitored.

In the following sections we present some examples of the queries executed for the different PCDMS.

In some queries the extraction of the bounding box can be done in the same query statement while in some other it has to be done before and specify the values directly in the statement. When the values are retrieved in a previous query the time spect in that process is also taken into account in the results shown in this report.

7.1 PostgreSQL flat

Rectangle (example query region #1)

```
CREATE TABLE query_results_1 AS (
    SELECT * FROM ahn_flat where (x between 85670.0 and 85721.0) and
                                 (y between 446416.0 and 446469.0));
```

Circle (example query region #3)

```
CREATE TABLE query_results_3 AS (
    SELECT * FROM
        (SELECT * FROM ahn_flat where (x between 85365.0-20.0 and 85365.0+20.0) and
                                      (y between 446594.0-20.0 and 446594.0+20.0)) a
    where (x - 85365.0)^2 + (y - 446594.0)^2 < (20.0)^2);
```

Other regions (example query region #5)

```
CREATE TABLE query_results_5 AS (
    SELECT * FROM
        (SELECT * FROM ahn_flat where (x between 85069.0 and 85201.0) and
                                      (y between 446334.0 and 446458.0)) a,
```

```
        query_polygons
    where id = 5 and _ST_Contains(geom, st_setSRID(st_makepoint(x,y),28992)));
```

Nearest neighbour (example query region #18)

```
CREATE TABLE query_results_18 AS (
    select * from
        (select * from ahn_flat where (x between 67195.73-10 and 67195.73+10) and
                                       (y between 433973.27-10 and 433973.27+10)) a
    ORDER BY (x - 67195.73)^2 + (y - 433973.27)^2 LIMIT 1000);
```

Note that this is an approximation to a nearest neighbour query since we use a bounding box (of a certain size) to pre-filter points.

In the case of rectangle regions, the smaller pieces are also rectangles so the second filtering using *_ST_Contains* is not done in that case.

7.2 PostgreSQL blocks

Any region (example query region #1)

```
CREATE TABLE query_results_1 AS (
    SELECT PC_Get(qpoint, 'x') AS x, PC_Get(qpoint, 'y') AS y,
    PC_Get(qpoint, 'z') AS z  from (
        SELECT pc_explode(pc_intersection(pa,geom)) AS qpoint from patches,
         query_polygons
            WHERE pc_intersects(pa,geom) and query_polygons.id = 1) AS qtable );
```

7.3 Oracle flat

Rectangle (example query region #1)

```
DECLARE
  BBOX SDO_GEOMETRY;
BEGIN
  SELECT SDO_GEOM_MBR (GEOM) INTO BBOX FROM QUERY_POLYGONS WHERE ID = 1;
  EXECUTE IMMEDIATE 'CREATE TABLE QUERY_RESULTS_2 AS
  SELECT VAL_D1, VAL_D2, VAL_D3 FROM AHN_FLAT
        WHERE (VAL_D1 BETWEEN '||TO_CHAR(BBOX.SDO_ORDINATES(1))||' AND
                              '||TO_CHAR(BBOX.SDO_ORDINATES(3))||')
          AND (VAL_D2 BETWEEN '||TO_CHAR(BBOX.SDO_ORDINATES(2))||' AND
                              '||TO_CHAR(BBOX.SDO_ORDINATES(4))||')

  ';
END;
```

Circle (example query region #3)

```
CREATE TABLE QUERY_RESULTS_3 (VAL_D1 NUMBER,VAL_D2 NUMBER,VAL_D3 NUMBER);
INSERT INTO QUERY_RESULTS_3
    SELECT * FROM
        ( SELECT * FROM AHN_FLAT
```

```
            WHERE (VAL_D1 BETWEEN 85345 AND 85385)
              AND (VAL_D2 BETWEEN 446574 AND 446614)

        ) B
    WHERE POWER((VAL_D1 - 85365.0),2) + POWER((VAL_D2 - 446594.0),2)
     <= POWER(20.0,2)
```

Other regions (example query region #5)

```
DECLARE
  BBOX SDO_GEOMETRY;
BEGIN
  SELECT SDO_GEOM_MBR (GEOM) INTO BBOX FROM QUERY_POLYGONS WHERE ID = 5;
  EXECUTE IMMEDIATE 'CREATE TABLE QUERY_RESULTS_5 AS
  SELECT VAL_D1, VAL_D2, VAL_D3
  FROM TABLE ( SDO_POINTINPOLYGON (
    CURSOR (
      SELECT VAL_D1, VAL_D2, VAL_D3 FROM AHN_FLAT
        WHERE (VAL_D1 BETWEEN '||TO_CHAR(BBOX.SDO_ORDINATES(1))||' AND
                              '||TO_CHAR(BBOX.SDO_ORDINATES(3))||')
          AND (VAL_D2 BETWEEN '||TO_CHAR(BBOX.SDO_ORDINATES(2))||' AND
                              '||TO_CHAR(BBOX.SDO_ORDINATES(4))||')
    ),
    (SELECT GEOM FROM QUERY_POLYGONS WHERE ID = 5),
    0.0001,
    NULL
  )) ';
END;
```

Nearest neighbour (example query region #18)

```
create table query_results_18(x DOUBLE, y DOUBLE, z DOUBLE);
INSERT INTO query_results_18
    select * from
        (select * from ahn_flat where
           (x between 67195.73-10 and 67195.73+10) and
           (y between 433973.27-10 and 433973.27+10)) a
    ORDER BY POWER(x - 67195.73,2) + POWER(y - 433973.27,2) LIMIT 1000;
```

Note that this is an approximation to a nearest neighbour query since we use a bounding box (of a certain size) to pre-filter points.

7.4 Oracle blocks

Any region (example query region #1)

```
CREATE TABLE QUERY_RESULTS_1 (VAL_D1 NUMBER,VAL_D2 NUMBER,VAL_D3 NUMBER);
INSERT INTO QUERY_RESULTS_1
    SELECT PNT.X, PNT.Y, PNT.Z FROM
        TABLE (SDO_PC_PKG.CLIP_PC((SELECT PC FROM AHN_BASE),
```

```
                        (SELECT GEOM FROM QUERY_POLYGONS WHERE ID = 1),
                        NULL,NULL,NULL,NULL)) PCBLOB,
      TABLE (SDO_UTIL.GETVERTICES(SDO_PC_PKG.TO_GEOMETRY(PCBLOB.POINTS,
                                  PCBLOB.NUM_POINTS,3,NULL))) PNT;
```

7.5 MonetDB flat

Rectangle (example query region #1)

```
create table query_results_1(x DOUBLE, y DOUBLE, z DOUBLE);
INSERT INTO query_results_1 SELECT x,y,z FROM ahn_flat WHERE
(x between 85670.0 and 85721.0) and (y between 446416.0 and 446469.0);
```

Note that in MonetDB we can not directly use CTAS.

Circle (example query region #3)

```
create table query_results_3(x DOUBLE, y DOUBLE, z DOUBLE);
INSERT INTO query_results_3
SELECT x,y,z FROM (SELECT x,y,z FROM ahn_flat WHERE
(x between 85365.0-20.0 and 85365.0+20.0) and
(y between 446594.0-20.0 and 446594.0+20.0) ) a
WHERE (power(x - 85365.0,2) + power(y - 446594.0,2)
< power(20.0,2)) ;
```

Other regions (example query region #5)

```
create table query_results_5(x DOUBLE, y DOUBLE, z DOUBLE);
INSERT INTO query_results_5
SELECT x,y,z FROM (SELECT x,y,z FROM ahn_flat WHERE
(x between 85069.0 and 85201.0) and
(y between 446334.0 and 446458.0) ) a , query_polygons
WHERE  id = 5 AND contains(geom, Point(x,y)) ;
```

Nearest neighbour (example query region #18)

```
create table query_results_18(x DOUBLE, y DOUBLE, z DOUBLE);
INSERT INTO query_results_18
    select * from
        (select * from ahn_flat where
           (x between 67195.73-10 and 67195.73+10) and
           (y between 433973.27-10 and 433973.27+10)) a
    ORDER BY POWER(x - 67195.73,2) + POWER(y - 433973.27,2) LIMIT 1000;
```

Note that this is an approximation to a nearest neighbour query since we use a bounding box (of a certain size) to pre-filter points.

7.6 LAStools

Rectangle (example query region #1)

```
psql  lt23090M -U oscar -h localhost -p 5434 -t -A -c "select filepath from
```

```
lasindex,query_polygons where ST_Intersects( query_polygons.geom,
lasindex.geom ) and query_polygons.id = 1" > input1.list
lasmerge -lof input1.list -inside 85670.0 446416.0 85721.0 446469.0 \
-o output1.las
```

Circle (example query region #3)

```
psql  lt23090M -U oscar -h localhost -p 5434 -t -A -c "select filepath from
lasindex,query_polygons where ST_Intersects( query_polygons.geom,
lasindex.geom ) and query_polygons.id = 3" > input3.list
lasmerge -lof input3.list -inside_circle 85365.0 446594.0 20.0 \
-o output3.las
```

Other regions (example query region #5)

```
psql  lt23090M -U oscar -h localhost -p 5434 -t -A -c "select filepath from
lasindex,query_polygons where ST_Intersects( query_polygons.geom,
lasindex.geom ) and query_polygons.id = 5" > input5.list
lasclip.exe -lof input5.list -poly query5.shp  -o output5.las -merged
```

In this case we use lasclip instead of lasmerge. The polygon is given with a shape file, so before the lasclip call we have do convert the PostgreSQL polygon into a shape file. We do this with:

```
pgsql2shp -f query5.shp -h localhost -p 5434 -u oscar -P oscar lt23090M \
 "select ST_SetSRID(geom, 28992) from query_polygons where id = 5;"
```

This conversion is not counted in the times.