Massive point clouds for eSciences

Mini-benchmark: description

O. Martinez-Rubi[1], *T.P.M. Tijssen*[2], *P.J.M. Van Oosterom*[2], *M. Ivanova*[1] *and E. Verbree*[2]

1 Netherlands eScience Center,
Science Park 140 (Matrix 1), 1098 XG Amsterdam, the Netherlands
2 OTB Research Institute for the Built Environment, TU Delft,
Jaffalaan 9, 2628 BX Delft, the Netherlands

October 28, 2014

Contents

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 mini-benchmark focussed on the storage and management of the data. We present several approaches using different databases and methods. We use the term "mini" to reflect the fact that we use a small test dataset and a small subset of test queries. The goal of the project is to develop a system that can deal with a much larger amount of points while providing a much larger set of functionalities.

The outline of this report is as follows: In section 2 we describe the input dataset while in sections 4, 5 and 6 we give details of the different approaches and how they load and query the data. Section 7 summaries the additional tests that have been carried out related to the different used approaches.

2 Input data: Test dataset

We use a small test dataset which consists on 20,165,862 LIDAR points of the area surrounding the OTB building in the TU Delft campus in the Netherlands. Figure 1 and 2 depict the 3D and 2D spatial representations of the area. The points in this dataset belong to one of the more than 60000 blocks of the Actual Height Model of the Netherlands (AHN2).

The input data is provided as a LAZ file (compressed LAS) and its size is 37 MB. Each point has 12 dimensions which are listed in the table 1. It must be noted that in the AHN2 dataset all the dimensions, except the spatial ones (X, Y and Z), currently contain dummy data, they all have the same default value. The size of an uncompressed point is 20 bytes. Note that the spatial coordinates, which are given in meters, are stored in 32 bits. Actually they are stored as scaled integers, which can also have an offset. The scale is 0.01 which means that the coordinates are stored in centimetres. In this case, the Y coordinate also has an offset of 400000.00 meters.

When uncompressing the input LAZ file into a regular LAS file, the size of the latter one is 385 MB (i.e. exactly 20,165,862 points * 20 bytes/point). We have also converted the LAZ file into an ASCII file with only the spatial coordinates. The size of the ASCII file is 457 MB.

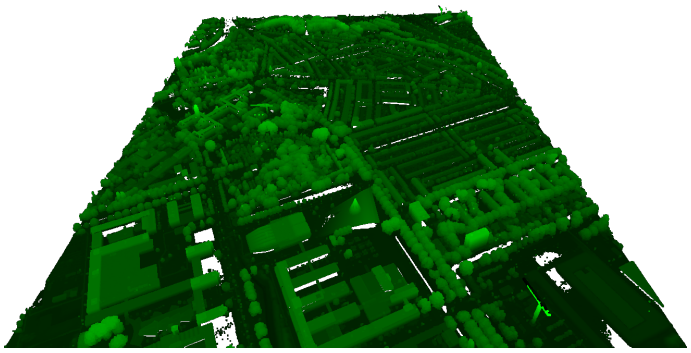

Figure 1: 3D representation of the area used in the mini-benchmark

Name	Size[bits]	Offset[bytes]
X	32	0
Y	32	4
Z	32	8
Intensity	16	12
Return Number	3	14
Number of Returns	3	14
Scan Direction	1	14
Flightline Edge	1	14
Classification	8	15
Scan Angle Rank	8	16
User Data	8	17
Point Source ID	16	18

Table 1: Dimensions of the points in the input LAZ file. Only X, Y and Z contain valuable data

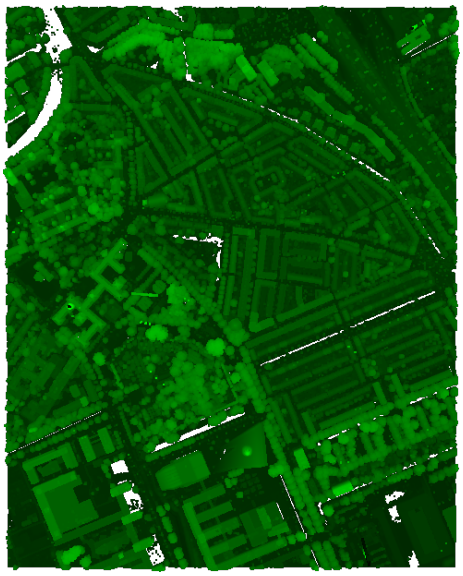

Figure 2: 2D representation of the area used in the mini-benchmark

3 Environment

This benchmark has been performed in a server with the following details

- HP DL380p Gen8 server
  - 2 x 8-core Intel Xeon processors (32 threads), E5-2690 at 2.9 GHz
  - 128 GB main memory
  - RHEL 6 operating system
- Disk storage (directly attached)
  - 400 GB SSD

- 5 TB SAS 15K rpm in RAID 5 configuration (internal)
- 88 TB SATA 7200 rpm in RAID 6 configuration (in Yotta disk cabinet)

4 Approaches description

We have implemented different approaches for the storage and querying of the test dataset presented in the previous section.

For the system that we aim to build in the project Massive point clouds for eSciences we are planning to use database technologies. Hence, for this benchmark we have implemented four different approaches using existing database technologies, i.e. we have not developed new software. Regarding the four implemented approaches, two are using Oracle and two are using PostgreSQL. Two of them consist on grouping the points in blocks.

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

- The same input file format will be used. Since the AHN2 dataset is provided with the LAZ (compressed LAS) file format we decided to use this one. In some of the approaches this will mean a data conversion during the data loading process. In such case it will be indicated in the description of the approach.
- The data will be stored without compression, i.e. using the available float data types.
- In the approaches that use blocks the block size will always be the same, 5000 points.
- Only the 3 spatial dimensions will be stored in the database.
- Only one process/thread will be used during the loading and querying.
- 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*.

In addition to the database approaches we have also implemented a "not-database" solution using the LASTools package. In this case, since the data is directly queried from the LAS files, many of the requirements presented above can not be met. Thus, the comparison in this case must be done taking the latter into account.

In the next paragraphs we present the different approaches. For each implemented approach the execution of the benchmark is split in two parts:

- Loading: The test dataset is loaded into the database (this obviously do not apply in the LASTools approach). The spent time, the required storage and the used resources (CPU and memory) are monitored. See section 5 for more information regarding the loading procedure for each approach.
- Querying: A set of (2D) queries is executed. The spent time, the returned points and the used resources (CPU and memory) are monitored. See section 6 for more information regarding the querying for each approach.

4.1 PostgreSQL PointCloud

The first approach is based on using the PointCloud extension developed by P. Ramsey (https://github.com/pramsey/pointcloud) on a PostgreSQL database. We create blocks of points and we store 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 used versions are:

- PostgreSQL:

```
PostgreSQL 9.3.2 on x86_64-unknown-linux-gnu, compiled by gcc (GCC) 4.4.7
 20120313 (Red Hat 4.4.7-3), 64-bit
```

- PostGIS:

```
POSTGIS="2.1.1 r12113" GEOS="3.4.2-CAPI-1.8.2 r3921" PROJ="Rel. 4.8.0,
6 March 2012" GDAL="GDAL 1.10.1, released 2013/08/26"
LIBXML="2.7.6" LIBJSON="UNKNOWN" RASTER
```

- PDAL:

```
(PDAL 0.9.9 (50ed6d) with GeoTIFF 1.4.0 GDAL 1.10.1 LASzip 2.1.0 Embed )
```

- PointCloud:

```
1.0.0
```

4.2 PostgreSQL flat

In this approach we store each point in a row of a table. We make a view where we use a method to create a PostGIS 2D point (on the X and Y coordinates), in which we add a GIST index. Note that, in PostgreSQL, it is not possible to directly add an index on a column of a view. Instead, we create an index on the method used to create the PostGIS points.

The used versions are:

- PostgreSQL:

```
PostgreSQL 9.3.2 on x86_64-unknown-linux-gnu, compiled by gcc (GCC) 4.4.7
 20120313 (Red Hat 4.4.7-3), 64-bit
```

- PostGIS:

```
POSTGIS="2.1.1 r12113" GEOS="3.4.2-CAPI-1.8.2 r3921" PROJ="Rel. 4.8.0,
6 March 2012" GDAL="GDAL 1.10.1, released 2013/08/26"
LIBXML="2.7.6" LIBJSON="UNKNOWN" RASTER
```

4.3 Oracle PointCloud

This approach is similar to the PostgreSQL PointCloud but using Oracle, i.e. it also groups the points in blocks (also 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 block extent) is stored in a separate object, the *SDO_PC*, also in a separate table.

The version of Oracle used in this test is:

```
Oracle Database 12c Enterprise Edition Release 12.1.0.1.0 - 64bit Production
```

With 1st patch provided by Mike Horhammer (mike.horhammer@oracle.com) on the 4th November 2013.

4.4 Oracle flat

The last database approach is based on storing the points in a regular flat table and use special functions within Oracle Spatial for intersection with the queried polygons. These functions treat X,Y number columns as a "geometry"..

The version of Oracle used in this test is:

```
Oracle Database 12c Enterprise Edition Release 12.1.0.1.0 - 64bit Production
```

With 1st patch provided by Mike Horhammer (mike.horhammer@oracle.com) on the 4th November 2013.

4.5 LASTools

In this approach we do not use any database, we use the software package LASTools. In the queries the data is directly acquired from the input LAZ file. It is based on creating a LAS Index file (LAX file) with the tool *lasindex*. This file is used by the tool *lasclip* together with the LAZ file to select points given a query region defined in a *ShapeFile*. The selected points are written into a new LAZ file.

Some of the LASTools tools only work in Windows environment. In order to make them work in Linux environment we need to use the tool Wine (*http://www.winehq.org*). LASTools is downloaded from *http://lastools.org/download/lastools.zip* when the last modification to its code was done the 25th October 2013 (see *CHANGES.txt*).

5 Loading procedures

In the next sections we describe the steps followed for the loading of the data in the different approaches.

PostgreSQL PointCloud

- Initialization (SQL), i.e. load the required extensions and create the table that will contain the blocks:

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

  The *blocks* table is set to use the default table space which will store the data in the 88 TB SATA discs.

- Loading. In order to load point clouds in PostgreSQL PointCloud approach 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 our input file (LAS/LAZ) there needs to be a format in the *pointcloud_formats* table. 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> 400000 </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">none</Metadata>
    </pc:metadata>
</pc:PointCloudSchema>
');
```

Then, we need to use PDAL tool to load the data from the input file (COMMAND-LINE):

```
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">5000</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 5000 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 a different *TABLESPACE* space is used. We created a *TABLESPACE* for the indexes which is stored in the SAS disks that are faster than the SATA disks used for the main table.

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

```
VACUUM FULL ANALYZE blocks;
```

Figure 3 depicts a 2D representation of the several blocks that have been created after loading the compressed LAS file. Note the white areas which correspond to zones without points. It is also noticeable the several flight scans during data acquisition that produce points over-densities in the overlapping areas.

In figure 4 we zoom in the bottom left area, we also show the index of each block. We can appreciate a Morton-like ordering in the identifiers of the blocks.

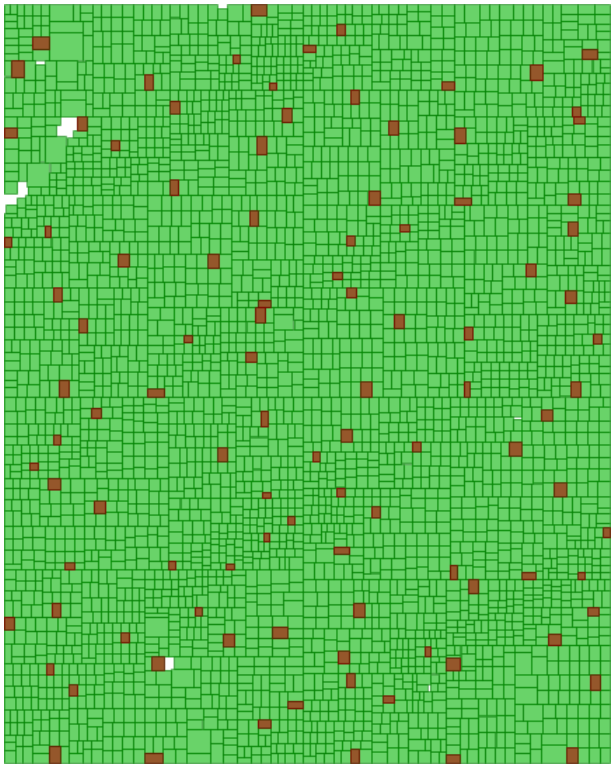

Figure 3: Blocks created using PostgreSQL PointCloud extension. The red blocks are the ones which size is different than 4999

Figure 4: Blocks identifiers in the PostgreSQL PointCloud approach

Comments/Difficulties

- PDAL will ignore the loaded point cloud format if it does not match with the one in the LAS file and it will do it without any warning. If you do not want to use all the columns

in the LAS file you need to use *filters* in the *XML* required for *pcpipeline*.

- Note that even though the block size is set to 5000 almost all the blocks have a size of 4999 points.
- It is possible to have areas that are not covered by any block in regions without any points.

PostgreSQL flat

- Initialization (SQL), i.e. load the *postgis* extension and create the flat table:

```
CREATE EXTENSION postgis;
CREATE TABLE flat (
    x DOUBLE PRECISION,
    y DOUBLE PRECISION,
    z DOUBLE PRECISION);
```

The *flat* table is set to use the default table space which will store the data in the 88 TB SATA discs.

- Loading. To load the LAZ file we first need to convert it to ASCII, we do it simultaneously while loading the data using a named-pipe (COMMAND-LINE):

```
mkfifo [TEMP PIPE]
psql [DB name] -c "COPY flat FROM '[TEMP PIPE]' (DELIMITER ' ')" 2>&1 &
las2txt --input [LAZ file] --output stdout --parse xyz
   --delimiter " " >> [TEMP PIPE]
rm [TEMP PIPE]
```

- Create the view that will be used in the queries and the PostGIS GIST index on the PostGIS method used in the view to ease the querying (SQL):

```
create view flat_view as
   select st_setSRID(st_makepoint(x,y), [SRID]) as xy, x, y, z from flat;
create index flat_xy_idx on flat using gist
(st_setSRID(st_makepoint(x,y),[SRID])) TABLESPACE indx;
```

Note that a different *TABLESPACE* space is used. We created a *TABLESPACE* for the indexes which is stored in the SAS disks.

- We vacuum and analyze the flat table:

```
VACUUM FULL ANALYZE flat;
```

Comments/Difficulties

- In this case we have tried different options of loading the data. In the end we have chosen the one presented above because it is the one that has a better compromise between the performance in data loading (speed and storage) and data querying (speed). See section 7.2.1 for more information regarding the different tried options.
- Note that in PostgreSQL it is not possible to create indexes on views. What it is possible is to create indexes on methods (and their returned values). Hence, we create an index on the same method that is used in the view to create the 2D PostGIS point.

Oracle PointCloud

- Create a user to allocate the new tables (SQL).

```
CREATE USER ORACLE_PC IDENTIFIED BY pass
    DEFAULT TABLESPACE USERS TEMPORARY TABLESPACE TEMP;
GRANT UNLIMITED TABLESPACE, CONNECT, RESOURCE, CREATE VIEW TO ORACLE_PC;
```

- Initialization(SQL):
  - Create the table where the points will be loaded.
  - Create the table that will contain the blocks of points.
  - Create the table that will contain the metadata of the blocks.

```
create table FLAT (
    RID VARCHAR2(24) default '0',
    VAL_D1 number,
    VAL_D2 number,
    VAL_D3 number)
tablespace USERS pctfree 0;

create table BLOCKS tablespace USERS pctfree 0 lob(points) store as
securefile (tablespace USERS cache) as
select * from mdsys.SDO_PC_BLK_TABLE where 0 = 1;

create table BLOCK_META (
    pc sdo_pc)
tablespace USERS pctfree 0;
```

  The *USERS* table space uses the 88 TB SATA discs.

- Load the points in the *flat* table using the bulk loader (the data need to be converted, but the conversion is directly piped into the loader) (COMMAND-LINE):

```
las2txt --input [LAZ file] --output stdout --parse xyz | sqlldr
[connection string] direct=true readsize=8000000 control=[CONTROL FILE]
data=\'-\' bad=ahn2table.bad log=ahn2table.log
```

The *CONTROL FILE* indicates which table is going to be populated. Its content is:

```
load data
append into table FLAT
fields terminated by ','
(
VAL_D1 float external(10),
VAL_D2 float external(10),
VAL_D3 float external(8)
)
```

- Create the point cloud metadata with the *SDO_PC_PKG.INIT* function and create the point cloud blocks with *SDO_PC_PKG.CREATE_PC* procedure (this last part being the most time consuming) (SQL).

```
DECLARE
    ptn_params varchar2(80) := 'blk_capacity=5000, work_tablespace=PCWORK';
    extent sdo_geometry := sdo_geometry(
         2003,[SRID],NULL,
         sdo_elem_info_array(1,1003,3),
         sdo_ordinate_array([MIN X],[MIN Y],[MAX X],[MAX Y]));
    ptcld sdo_pc;
BEGIN
    ptcld := sdo_pc_pkg.init ('BLOCKS_META', 'PC', 'BLOCKS', ptn_params,
                              extent, 0.0001, 3, NULL, NULL, NULL);
    insert into BLOCKS_META values (ptcld);
    commit;
    sdo_pc_pkg.create_pc (ptcld, 'FLAT', NULL);
END;
```

Note the block size of 5000 and the *work_tablespace* which is set to a special table space (*PCWORK*) which is using the 5 TB SAS discs. The extent must specify the total area that contains all the points.

- Create a primary key for the blocks which is stored in the SAS disks (via using table space *INDX*), drop the flat table and compute statistics to provide the optimizer with up-to-date statistics of the data (SQL):

```
alter table BLOCKS add constraint BLOCKS_PK primary key (obj_id, blk_id)
      using index tablespace INDX;
drop table FLAT;

analyze table BLOCKS compute system statistics for table;
analyze table BLOCKS_META compute system statistics for table;
begin
    dbms_stats.gather_table_stats('ORACLE_PC','BLOCKS',
        NULL,NULL,FALSE,'FOR ALL COLUMNS SIZE AUTO',8,'ALL');
```

```
end;
begin
    dbms_stats.gather_table_stats('ORACLE_PC','BLOCKS_META',
        NULL,NULL,FALSE,'FOR ALL COLUMNS SIZE AUTO',8,'ALL');
end;
```

Figure 5 shows a 2D representation of the several blocks that have been created.

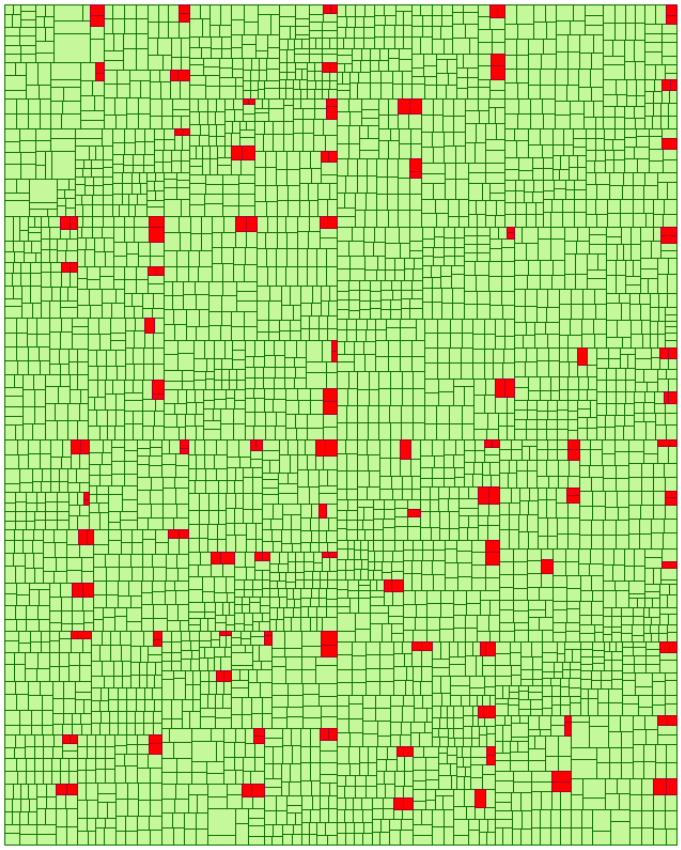

Figure 5: Blocks created using Oracle PointCloud. The red blocks are the ones which size is lower than 5000 points

Comments/Difficulties

- The columns in the flat table must have the same name as shown in this example, i.e. an identifier *rid* and *val_d1*, *val_d2*, *val_d3* for the spatial coordinates.
- In principle PDAL could also be used for loading the data. This must be further investigated.
- Note that in Oracle we create a new user instead of a new database. This is done to avoid the large overhead in creating databases. This also requires to use a super user of the root database (the one that will contain the tables of each user) that can creates new users.

- In Oracle the table names are internally stored in upper case. Thus, in order to avoid naming problems we recommend to always use upper case names.

Oracle flat

- Create a user to allocate the new table.

```
CREATE USER ORACLE_FLAT IDENTIFIED BY pass
    DEFAULT TABLESPACE USERS TEMPORARY TABLESPACE temp;
GRANT UNLIMITED TABLESPACE, CONNECT, RESOURCE, CREATE VIEW TO ORACLE_FLAT;
```

- Initialization(SQL), i.e. create the table where the points will be loaded:

```
create table FLAT (X number,Y number,Z number)
    TABLESPACE USERS pctfree 0;
```

  The *USERS* table space uses the 88 TB SATA discs.

- Load the points in the *flat* table using the bulk loader (the data need to be converted, but the conversion is directly piped into the loader) (COMMAND-LINE):

```
las2txt --input [LAZ file] --output stdout --parse xyz | sqlldr
[connection string] direct=true readsize=8000000 control=[CONTROL FILE]
data=\'-\' bad=ahn2table.bad log=ahn2table.log
```

  The *CONTROL FILE* indicates which table is going to be populated. Its content is:

```
load data
append into table FLAT
fields terminated by ','
(
X float external(10),
Y float external(10),
Z float external(8)
)
```

- Create B-tree index on X,Y columns which is stored in the SAS disks:

```
create index FLAT_IDX on FLAT (x, y) tablespace INDX;
```

- Provide the optimizer with up-to-date statistics (SQL):

```
analyze table FLAT compute system statistics for table;
begin
    dbms_stats.gather_table_stats('ORACLE_FLAT','FLAT',
        NULL,NULL,FALSE,'FOR ALL COLUMNS SIZE AUTO',8,'ALL');
end;
```

Comments/Difficulties

- As in the Oracle PointCloud approach we need to create a new user to allocate the flat table (otherwise it would conflict with the one create in Oracle PointCloud). Like in the previous case this requires to have a super user with the proper permissions to create new users.
- Like in the previous approach we recommend that table names are always specified in upper case.

LASTools

- Make sure it is possible to write in the folder where the LAS file is located.
- Run *lasindex* tool to generate the index files that will speed up lasclip.

  ```
  lasindex [LAS File path]
  ```

  This will create the LAX file in the same folder where the LAS file is stored.

Comments/Difficulties

- We have compiled a guideline to install LASTools and Wine in a CentOS 6 system.
  https://github.com/NLeSC/pointcloud/blob/master/install_lastools_solution/install_lastools
- Note that this is a file-based solution. In order to query points, the LAX index file needs to be read and, if the query region is within the area covered by the LAS file, this one also needs to be opened. This is fine for a small number of LAX and LAS files but for a very large number of LAX/LAS files the overhead of the massive file opening/reading would become the main bottleneck.

6 Queries

We have defined a set of 2D geometry queries:

1. Small rectangle, axis aligned, 51 x 53 m
2. Large rectangle, axis aligned, 222 x 223 m
3. Small circle at (85365 446594), radius 20 m
4. Large circle at (85759 447028), radius 115 m
5. Simple polygon, 9 points
6. Complex polygon, 792 points, 1 hole
7. Long, narrow, diagonal rectangle

In figure 6 we show the several queried geometries on top of the 2D representation of the test dataset area.

In the next paragraphs we show the SQL queries that are executed for the different approaches, we only show the query for the first polygon.

Comments/Difficulties

- The geometries are given as WKT polygons. We pre-load the query polygons in a table called *query_polygons*.
- Originally the circle geometries were given as *CURVEPOLYGON* but some PostGIS methods are not yet compatible with this data type so these geometries had to be rewritten as POLYGON data types.

Figure 6: Queried geometries on top of 2D representation of the test dataset

PostgreSQL PointCloud

```
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 blocks, query_polygons WHERE PC_Intersects(pa,geom) AND query_polygons.id = 1)
AS temp_query_results_1)
```

A closer look to the previous command reveals that the querying is based on two steps:

- Initial filter on the blocks. The blocks that overlap the query area are filtered using the *PC_Intersects* method which internally uses the PostGIS *ST_Intersects* method. This method includes a bounding box comparison that will allow the usage of indexes.
- For each of the filtered blocks we get the points that are within the query region with the methods *PC_Intersection* and *PC_Explode*.

PostgreSQL flat

```
CREATE TABLE query_results_1 AS
(SELECT x,y,z FROM flat_view, query_polygons
WHERE query_polygons.id = 1 AND st_contains(geom,xy))
```

As stated in the loading procedure, we created an index on the same function that is used to create the column *xy* in the view and this index is used when we use the *xy* column in the where statement.

Note that, even if not noticeable in the query statement, the *st_contains* method internally does two steps: first it creates a bounding box of the query region (that allows to use the spatial index) to make a first selection of points. Afterwards, it checks if each pre-selected point lay in the query region.

Oracle PointCloud

```
CREATE TABLE query_results_1 AS
SELECT pnt.x, pnt.y, pnt.z FROM
  table (SDO_PC_PKG.Clip_PC(
      (SELECT pc FROM BLOCKS_META),
      (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
```

The *SDO_PC_PKG.CLIP_PC* function returns blocks (*SDO_PC_BLK* objects) which contain the points that overlap with the query region. Internally this method:

- Makes an initial selection of blocks (using the extent stored in the meta-data) that overlap with the bounding box of the query region (in this moment the spatial indexes are used).
- Checks if it can discard some blocks because they do not overlap the query region.
- Processes the remaining blocks and returns modified versions of them which only contains points in the query region.

Afterwards the points are extracted with the *GetVertices* method (after converting the blocks to geometry with the function *SDO_PC_PKG.TO_GEOMETRY*.

Oracle flat

```
DECLARE
  bbox sdo_geometry;
BEGIN
  SELECT sdo_geom_mbr (geom) INTO bbox FROM QUERY_POLYGONS WHERE id = 1;
  execute immediate 'CREATE TABLE query_results_1 AS
  SELECT /*+ PARALLEL (1) */ x, y, z
  FROM table ( sdo_PointInPolygon (
    cursor (
      SELECT x, y, z FROM FLAT WHERE
      (x between '||to_char(bbox.sdo_ordinates(1))||'
          AND '||to_char(bbox.sdo_ordinates(3))||')
      AND (y between '||to_char(bbox.sdo_ordinates(2))||'
          AND '||to_char(bbox.sdo_ordinates(4))||')
```

```
    ),
    (SELECT geom FROM QUERY_POLYGONS WHERE id = 1),
    0.0001,
    NULL
  ))';
END;
/
```

Note that in this approach, like in the PostgreSQL flat, there also two steps in the query statement, i.e first we filter the points that are within the bounding box of the query region (in this part the built binary tree is used). Afterwards we check if the selected points lay within the query region.

We use the *SDO_Geom_MBR* method since it it the fastest way to compute a bounding box.

The *PARALLEL* allows to easily use multiple processors for the query but in this benchmark we agreed on using a single core.

LASTools

```
pgsql2shp -f query1.shp -h localhost -u [user] -P [pass] dbname
"SELECT ST_SetSRID(geom, [SRID]) FROM query_polygons where id = 1;"

lasclip.exe [LAZ File] -poly query1.shp -verbose -o query1.laz
```

Note that in this case the query regions are loaded in a PostgreSQL database. For each query the query polygon is stored as a ShapeFile and this is used by the LASTools *lasclip* tool when selecting the points (together with the LAZ/LAZ files).

7 Additional tests

7.1 PostgreSQL PointCloud

We have performed some additional tests with the PostgreSQL PointCloud approach that are out of the main focus of the mini-benchmark but still provide some interesting information.

7.1.1 Different block sizes

We tried loading the data with different block sizes and it seems that a lower size would be better for the queries. In figure 7 we show the normalized times (all time values divided by the maximum value) of the most expensive queries (the ones retuning the larger numbers of points) for different block sizes. From this picture we see that a block size of 3000 is better for the current test dataset and performed queries.

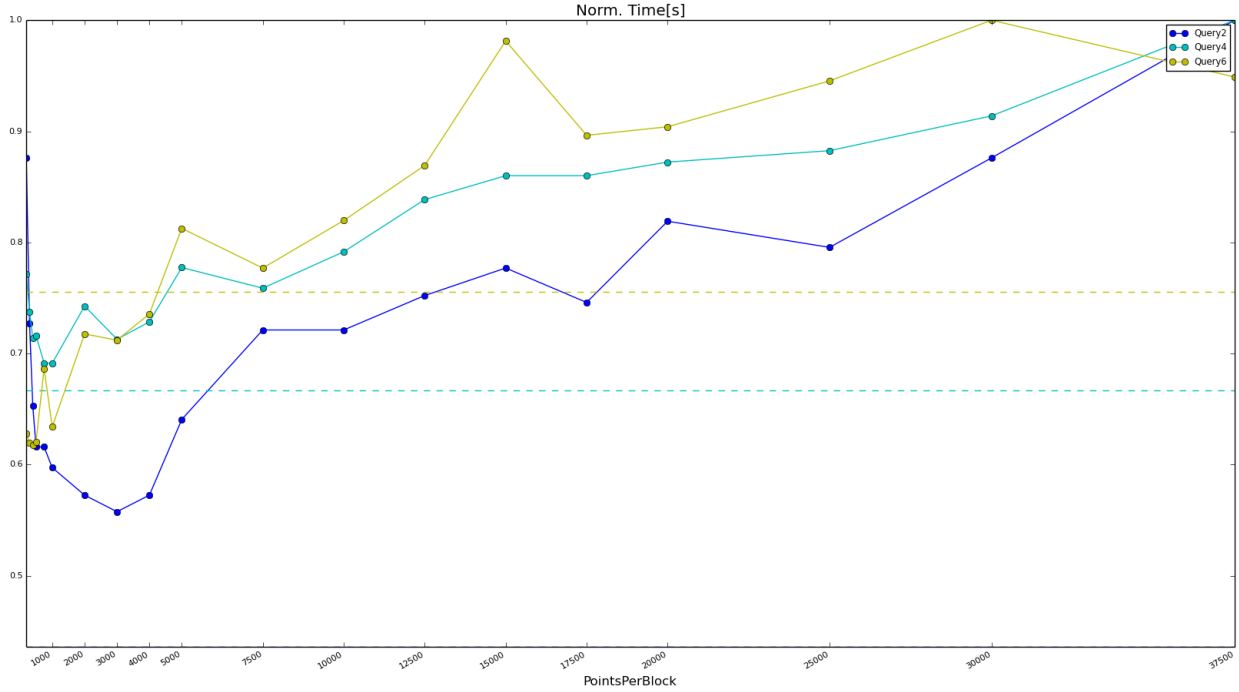

Figure 7: Normalized times of the three most expensive queries for different block sizes.

7.1.2 Multiple overlapping LAS files

When loading several LAS files using the PDAL *pcpipeline* tool, each LAS file is independently loaded in the blocks table. For each LAS file, its data is used to generate some blocks, the blocks that are already in the database are not used. So, the loading is faster because each LAS file is independently loaded but if there are some overlapping points in different LAS files they will produce blocks that are overlapping as well. However, if the data is properly acquired we do not expect many overlapping and this loading strategy really speeds up the process thus it is probably preferable.

7.1.3 Multiple cores

We have tried using a multiprocessing solution for the second step of the querying in the PostgreSQL PointCloud (where the filtered blocks are exploded) and we get much better times (4x or 5x). A similar idea could be also used in the other database solutions since all of them are based on two steps where the first one make a pre-selection of points.

7.2 PostgreSQL flat

7.2.1 Loading procedure

In this test we have tried different data schemas and loading procedures for the flat table approach in PostgreSQL. We tried using double precision and numeric types for the spatial coordinates, using views, using indexes built on methods, altering and dropping columns to create the PostGIS points, etc.

In table 2 we show the describe the different tried options.

#Op.	Description of data schema and loading procedure
1	create table with Double Precision types, load data, create PostGIS geometry column xy, fill geometry column xy, drop x, drop y, index on geom column xy
2	create table with Double Precision types, load data, create view with geometry column xy, index on original table (in the view is not possible) using same function as in the view creation, i.e. the one to create the geometry column xy
3	create table with Double Precision types, load data, index on table using function to create geometry column xy but without creating it, i.e. not using views
4	same as option #2 but using numeric types: x numeric(7,2), y numeric(8,2), z numeric(6,2)
5	same as option #3 but using numeric types: x numeric(7,2), y numeric(8,2), z numeric(6,2)
6	create table with Double Precision types, load data, create table from selecting points using function to create geometry column xy, drop first table, index on geometry column xy
7	create table with Double Precision types, load data, create second table with scaled integers as select (x*100)::integer as x, (y*100)::integer as y, (z*100)::integer as z, drop first table, make view to second table to convert back to Double Precision (also descaling) because this is what is used for queries, create index with same columns used in view
8	create table with Double Precision types, load data, create second table with scaled integers as select (x*100)::integer as x, (y*100)::integer as y, (z*100)::integer as z, drop first table, make index with methods that convert back to Double Precision (also descaling), i.e. without the view.
9	same as option #7 but using 3D index instead of 2D index, i.e. also using Z column.

Table 2: Different options of data schemas and loading procedures for the PostgreSQL flat

In table 3 we show the results of the loading procedure for each option presented in table 2. We show the size of the database before the index creation (*size_bi*) and after the index creation (*size_ai*), we also show the spent time in the same system as used in the main mini-benchmark.

The analysis of the obtained results reveals that:

#Op.	size_bi[MB]	size_ai[MB]	time[s]
1	1635	2690	570.48
2	1015	2069	534.51
3	1015	2069	535.47
4	1094	2149	566.89
5	1094	2149	568.37
6	1635	2690	560.26
7	863	1927	553.50
8	863	1927	557.13
9	863	1926	572.58

Table 3: Sizes and times of the different options

- Not storing the geometry PostGIS 2D column (i.e. using indexes on the method to create the geometry column) decreases the size (option #2 and #3 are faster and lighter than option #1).
- Using Numeric data type is not beneficial for the size and neither the time spent in loading (options #4 and #5 are slower and heavier than options #2 and #3). Using create table from (as in option #6) is faster than add/drop column (as in option #1) when we want to create the PostGIS 2D column, but none of them are a good idea for large amount of points
- The fastest in loading are the ones that are creating the index on the function (that creates the 2D point) and not using numeric types, i.e. option #2 and #3. It is not clear which one is faster (with or without view) so we will prefer to use views since it is a cleaner approach.
- Using scaled integers instead of Double Precision (options #7 to #9) only decreases 10% the size. This is due to overhead in PostgresSQL pages for the main table and because the view/index need to be build using Double Precision anyway since this is used for the queries so there is not possible decrease in the index either.
- Surprisingly the 3D index (option #9) has the same size than the 2D index (option #7). This is due to the overhead of the PostgreSQL pages. See section 7.2.2 for more information about PostgreSQL overheads.

We have also executed the queries of the mini-benchmark for all the described flat table options. In table 4 we show the query statements for the different options. Note that some of them use views or directly the table, and also some of them have to do extra conversions to provide the results.

#Op.	Query statement
1	`select st_x(point_xy) as x, st_y(point_xy) as y, z from points, query_polygons where query_polygons.id = 1 and st_contains(geom,point_xy)`
2	`select x,y,z from ppoints,query_polygons where query_polygons.id = 1 and st_contains(geom,xy)`
3	`select x,y,z from points,query_polygons where query_polygons.id = 1 and st_contains(geom,st_setSRID(st_makepoint(x,y),28992))`
4	`select x,y,z from ppoints,query_polygons where query_polygons.id = 1 and st_contains(geom,xy)`
5	`select x,y,z from points,query_polygons where query_polygons.id = 1 and st_contains(geom,st_setSRID(st_makepoint(x,y),28992))`
6	`select st_x(xy) as x, st_y(xy) as y, z from points,query_polygons where query_polygons.id=1 and st_contains(geom,xy)`
7	`select x, y, z from ppoints, query_polygons where query_polygons.id = 1 and st_contains(geom,xy)`
8	`select x::double precision/100, y::double precision/100, z::double precision/100 from points, query_polygons where query_polygons.id=1 and st_contains(geom,st_setSRID(st_makepoint( x::double precision/100,y::double precision/100),28992))`
9	`select x, y, z from ppoints, query_polygons where query_polygons.id = 1 and st_contains(geom,xyz)`

Table 4: Different query statements used in the different options tried for the PostgreSQL flat approach. The query statement is for the first query presented in the mini-benchmark.

The results of running all the queries for all the options revealed that:

- Using numeric type is not good for the queries either. The reason is that numeric types are much slower when doing arithmetic operations.
- The fastest in queries is option #1 but the two options that use Double Precision attributes and use indexes on the function to create PostGIS 2D point(#2 and #3) are really close in performance.
- The options that use scaled-integers are slightly slower than the ones that use Double Precision (5% - 10% depending on the query). This is due to the conversion required to provide the results as Double Precision.
- Using 2D index is faster than using 3D index.

After analysing the results of both loading and querying with the different options we decided to use option #2 for the mini-benchmark because using Double Precision offers the best compromise between size and speed. In addition using views instead of large modifications in the input table is cleaner and faster and more efficient for larger number of points. Finally using 3D indexes, even though not affecting the size, makes the queries slower.

7.2.2 Overhead

In the previous test we were very surprised to find out that 3D indexes had the same size that 2D indexes when our estimation was that they should be 50% larger. This is due to the overheads introduced by the PostgreSQL pages which are also used in indexes.

A PostgreSQL page consists on:

- 24 bytes page header.
- The several items. Each item is:
  - 4 bytes for identifier
  - 23 bytes header
  - user data

Note that according to PostgreSQL documentation the 23 bytes header for each data item is for table rows, we are not sure if it is the same for indexes but we assume it is.

Also, in PostgreSQL the pages are never filled with all possible data items, normally they contain only around 80% of their maximum capacity. This is done to allocate possible future items but the full space is reserved.

So, with the latter information, we would need 1220 MB to store the 2D index and 1415 MB to store the 3D index (assuming 80% of page usage). Both sizes are approximated to 1GB in the size estimation done by PostgreSQL in the previous test.

Taking into account the PostgreSQL page is 8 KB, the computations for the previous values are:

Size of 2D index

We assume that in order to store a 2D index we need 24 bytes for each point (8 for X, 8 for Y and 8 for the index value)

So, in this case a page would fit 160 items.

$$\frac{8192 - page\_header}{item\_id + item\_header + item\_data} = \frac{8192 - 24}{4 + 23 + 24} = 160 \tag{1}$$

Taking into account that each page is only up to 80% of its capacity, to store 20 million points we would need 156250 pages.

$$\frac{20000000}{160 \cdot 0.8} \tag{2}$$

The final size of the whole index, each page being 8K, would be 1220 MB.

Size of 3D index

Now if we use a 3D index, we would need for each point 32 bytes (8 for X, 8 for Y, 8 for Z and 8 for the index value)

In this case, the page would fit 138 items.

$$\frac{8192 - 24}{4 + 23 + 32} \tag{3}$$

To store 20 million points items we would need 181160 pages

$$\frac{20000000}{138 \cdot 0.8} \tag{4}$$

The final size of the whole index, each page being 8K, would be 1415 MB.