Mapping the Via Appia in 3D

Linking to Massive Point Clouds for eSciences

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

March 19, 2014

1 Overview

This document describes the current activities, results and prospects of the project "Massive Point Clouds for eSciences". In addition it reports on the usability in the project "Mapping the Via Appia in 3D" of the investigated and/or created technologies of the project "Massive Point Clouds for eSciences".

2 Massive Point Clouds for eSciences

The main goal of the project "Massive Point Clouds for eSciences" is to create a system for the storage, the management, the analysis and the processing, the dissemination, the visualization and the manipulation of massive point clouds.

In the first part of the project which will conclude in the end of 2014 we are focussing on the development of a database system for the storage and the management of the point clouds data. For this purpose we are testing different database management systems (DBMS) and different approaches within each DBMS.

Currently we are testing with Postgres and Oracle because they are the only ones that have support for spatial types as well as point clouds. Both are based on grouping (and compressing) points in blocks and storing these blocks in rows of tables. In this solutions we can query any type of 2D/3D polygons.

In addition we are also testing MonetDB which offers a very good performance for flat tables (one row per point) but does not offer decent support for spatial types and point clouds. This means that only rectangular/cubic or circular/spherical queries are possible. We also tested flat tables in Oracle and Postgres.

Even though we are focussing on DBMS solutions, we are also testing using a solution that is not a DBMS, this means that we are directly using the LIDAR data (in their original format which is the LAS format) without loading it in any database. This is possible thanks to the LASTools package.

When dealing with massive point clouds in databases there are many issues that should be taken into account and other ones that may enormously help. In this project we want to consider all this issues and find the most optimal combination.

- Hardware: There are many possible hardware configurations when dealing with point clouds: single server, computer cluster, cloud system.
- Software: Which OS system and which DB software is installed.
- Loading: How the data is loaded into the DB.
- Indexing: Creating spatial indexes on the data
- Clustering: Re-ordering of the data based on indexes or space filling curves
- Blocking: Grouping the points in blocks.
- Compressing: Using the right data types, scaling the values, using offsets, store all position information in a single code (Morton or Hilbert codes).
- Partitioning: Dividing the large point clouds in smaller pieces probably by spatial range.
- Querying: Specifying the queries in the most efficient way.

- Parallel processing: Use multiple cores
- Level of detail: Allow to select only some points of certain area.

3 Current Activities

For our current activities we are using subsets of the AHN2 dataset that we load and query in different DBMS. Concretely we are using:

- dataset *20M*: single LAZ file with 20,165,862 LIDAR points of the area surrounding the TU Delft campus in the Netherlands. The size of this file is 37 MB.
- dataset *210M*: This dataset consists of 17 LAZ files with 210,631,597 LIDAR points of part of the city of Delft in the Netherlands. This dataset contains the previous *20M* dataset and its full size is of 366 MB (combining all the LAZ files).
- dataset *2201M*: This dataset consists of 153 LAZ files with 2,201,135,689 LIDAR points of part of the city of Delft in the Netherlands. This dataset contains the previous *210M* dataset and its full size is of 3310 MB (combining all the LAZ files).
- dataset *21686M*: This dataset consists of 2136 LAZ files with 21,686,319,598 LIDAR points of the city of the Delft and surroundings. This dataset contains the previous *2201M* dataset and its full size is 33969 MB (combining all the LAZ files). This dataset has not been cleaned so it may contain inconsistent points (the previous datasets are cleaned, i.e. they do not contain erroneous points like for example points with a height of 100 km).
- dataset *23090M*: This dataset consists of 1492 LAZ files with 23,090,482,455 LIDAR points of the city of the Delft and surroundings. This dataset contains the previous *2201M* dataset and its full size is 35673 MB (combining all the LAZ files). This dataset is clean.

Regarding the queries we are usually querying the following regions:

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

All the tests have 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)

Next we summarize the tests that we have performed:

3.1 Mini-benchmark

- Goal: Load and query the dataset *20M* in Postgres using blocks, Postgres flat table, Oracle using blocks, Oracle flat table. Compare it with LASTools query performance.
- Details:
  - Approaches that use blocks are not using compression and the block size is fixed to 5000 points.
  - Only X, Y and Z coordinates are loaded.
  - We only use 1 core.
- Results / conclusions:
  - LASTools offers the best performance for larger queries for the tested *20M* dataset. However, for larger datasets this may become much slower. Also LASTools would offer a limited set of functionalities.
  - Required software for the DBMS solutions is not easy to install. We gathered installation guides for CentOS/RedHat systems.
  - Flat tables require much more storage and indexes are much larger.
  - Oracle using blocks is better than Postgres.
  - Postgres using blocks is very fast in loading and requires the least storage (data values are scaled and offset and stored as integers)
  - Oracle loading is too slow.
  - Flat tables can be useful for some tests but are not a good option for very large point clouds.
  - We realized that when using blocks, updating the points would be quite cumbersome because of the way the data is organized.

3.2 Block sizes and compression

- Goal: Load and query the datasets *20M* and *210M* in Oracle and Postgres using blocks with different (built-in) compression techniques and block sizes.
- Details:
  - We tried with blocks sizes 300, 500, 1000, 3000 and 5000.

- Only X, Y, Z are loaded.
- We only use 1 core.

- Results / conclusions:
  - In Postgres using the *dimensional* compression offers a similar loading time to *none* compression while requiring 2.5 times less storage and "only" slowing down by 10 % the rectangle and circle queries. About the block sizes, we find that values between 1000 and 3000 points per block seem to offer a good compromise between loading and querying.
  - Oracle works better with larger block sizes. We only tried up to 5000. Using compression does not damage the queries, does not require longer loading time (for large block size) and decreases considerably the storage needs. However, and as already stated, it is very slow in loading.

3.3 Flat tables and spatial functionalities effect

- Goal: Load and query the *20M* and *210M* datasets in Postgres, Oracle and MonetDB using only flat tables. We also tested how using spatial functionalities in Oracle and Postgres affect the performance.

- Details:
  - Only rectangular and circular queries are possible in MonetDB so only these ones are tested.
  - Only X, Y, Z are loaded.
  - We only use 1 core.

- Results / conclusions:
  - MonetDB is much faster for the tested datasets.
  - MonetDB creates index automatically (in the first range query)
  - The penalty for using spatial types and methods (in Oracle and Postgres) is very high.
  - With the sizes of the current dataset all the data fits in memory.

3.4 Built-in clustering

- Goal: Test several built-in clustering techniques in Postgres and Oracle when loading and querying the datasets *20M*, *210M* and *2201M*.

- Details:
  - In Postgres we use *CLUSTER ON* tool which resorts a table based on an index.
  - In Oracle we use the index-organized tables
  - Only X, Y, Z are loaded.
  - We only use 1 core.

- Results / conclusions:

- Clustering Postgres flat slightly improves the query times but requires 50% extra time in loading. It does not help storage though.
- Clustering the Postgres blocks does not help.
- Oracle needs a primary key to cluster a table. This requires a 3D index/key (and points to be unique). It decreases storage in more than 50% because the index and the data is the same. Using clustering (i.e. a 3D index) we get better query times (at least with the tested datasets) but the loading times get exponential.

3.5 Different datasets

- Goal: Load and query the datasets *20M*, *210M*, *2201M* and *21686M* with MonetDB (using a flat table) and Oracle and Postgres (using blocks)
- Details:
  - MonetDB does not use any spatial functionality so only circular and rectangular queries are possible.
  - Postgres is using dimensional compression and a block size of 3000.
  - Oracle is using high compression and a block size of 5000.
  - Only X, Y, Z are loaded.
  - We only use 1 core.
- Results / conclusions:
  - MonetDB is better up to few billions (only circle and rect queries).
  - In MonetDB more points means worse performance. Oracle and Postgres offer a constant performance.
  - Indexing in MonetDB is magic. We have observed some strange behaviour (indexes being automatically deleted, indexes not being used in circles, etc).
  - First query in a new region in MonetDB is slower than repetitions.
  - Postgres is very bad in large regions. Oracle is much better (not as good as MonetDB for small datasets though)
  - Oracle is unusable for the datasets larger that *210M*.

3.6 Built-in parallel querying and new parallel querying algorithms

- Goal: Load and query the dataset *20M* in Oracle using flat table, Oracle using blocks and Postgres using blocks. Test the native parallel in Oracle. Test parallel blocks querying (PBQ) algorithm in Oracle and Postgres.In this algorithm several processes are started, each taking a bunch of blocks and processing them in parallel.
- Details:
  - Postgres using blocks uses dimensional compression and a block size of 3000.
  - Oracle using blocks uses high compression and a block size of 5000.

  - We defined and implemented (for Oracle and Postgres) a parallel blocks querying algorithm based on two steps: 1- Using the bounding box of the query region create a temporal table that only contains the blocks that overlap the query region. 2- Each process takes a bunch of blocks from the temporal table and process them, i.e. the points that overlap the query region are extracted.
  - Only X, Y, Z are loaded.

- Results / conclusions:
  - Oracle flat table benefits from native parallel support.
  - Oracle using blocks does not benefit from native parallel but it does from PBQ in large queries.
  - Postgres benefits from PBQ in all cases.
  - Better parallel algorithms should be explored.

4 Usability in the project Mapping the Via Appia in 3D

In the project "Massive point clouds for eSciences" we have tested loading up to few billions points in different DBMS systems with different configurations using the existing technologies. We have also tested different queries (up to 0.5 million points). The obtained performance is not what we expect and we think it can be improved so there is still more tests and new developments to be done including:

- Developments:
  - Improve querying: parallel querying, querying algorithms based on range queries, etc.
  - Clustering flat tables based on Morton/Hilbert codes.
  - Test alternative data structures based on storing only significant bits and/or Hilbert/Morton codes with/without using blocks.
  - Implement Level-of-Detail which is currently missing in the DBMS.

- Tests:
  - Different query regions (up to 2 million points).
  - Larger datasets (up to the whole AHN2, i.e. 640 billion points).
  - Different datasets like, for example, the Via Appia data.
  - Test of our new developments.

We are monitoring the developments of Oracle, MonetDB and Postgres related to point clouds so any new developments in these DBMS will also be tested.

Therefore, even though we expect a considerable improvement we can already provide the expertise to load large point clouds in DBMS that could be used for the storage of the point clouds of the Via Appia. The system would not be optimal at this stage but it could improve with the new results that would come out of the project "Massive point clouds for eSciences".

With the current expertise we can say that if the points need to be "update-able" and/or the amount of points does not exceed a **few billions** then the **MonetDB flat table** could be

used. If we are dealing with **larger datasets** and there is not need of "update-able" points then we should use **Postgres with blocks** since it offers the most stable system. Oracle is currently having issues when loading large point clouds. However, these issues should be fixed soon.

When using Postgres we could use parallel solutions to speed up considerably the queries. It also advisable to use compression when available and to use small block sizes (3000 seems a good size).

When using Oracle we could also use parallel solutions to speed up the queries. In this case it is recommendable to use larger block sizes (minimum 5000). It is also advisable to use compression.

5 Additional information

In this section we detail the loading procedures and the query statements for the different suggested solutions.

5.1 Loading

5.1.1 MonetDB

We compiled a installation guide for MonetDB. Contact *o.rubi@esciencecenter.nl* to get it.

Once the software is installed, a database created and its instance running:

- Initialization (SQL):

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

- Loading. For each LAZ file to be loaded we need to convert it to an ASCII file with (COMMAND-LINE):

```
las2txt --input [LAZ file] --output [ASCII file] --parse xyz --delimiter ","
```

Then, we load the data into the DB (SQL):

```
copy into flat from '[ASCCI file]' using delimiters ',','\n';
```

Note, that in this case we are only loading X, Y an Z.

5.1.2 PostgreSQL

We compiled an installation guide for the PostgreSQL and the related software (PDAL, point-cloud extension etc.). Contact *o.rubi@esciencecenter.nl* to get it.

- Initialization (SQL):

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

- 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 each input file (LAS/LAZ) there needs to be its format in the *pointcloud_formats* table. Every time we have a LAZ file with a different sets of scales and offsets (than previous files) we need to add a new format. To add a new format (SQL):

```
INSERT INTO pointcloud_formats (pcid, srid, schema) VALUES ([formatID], [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> [scaleX] </pc:scale>
        <pc:offset> [offsetX] </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> [scaleY] </pc:scale>
        <pc:offset> [offsetY] </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> [scaleZ] </pc:scale>
        <pc:offset> [offsetZ] </pc:offset>
    </pc:dimension>
    <pc:metadata>
```

```
        <Metadata name="compression">dimensional</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">[formatID]</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 *formatID* is the same as we previously commented. If none was inserted then we need to get the *formatID* of the one that was already inserted. 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 a different *TABLESPACE* space is used. We created a *TABLESPACE* for the indexes which is stored in a faster device.

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

```
VACUUM FULL ANALYZE blocks;
```

5.1.3 Oracle

We compiled a installation guide for Oracle. Contact *o.rubi@esciencecenter.nl* to get it.

- Initialization(SQL):

```
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 compress high cache) as
select * from mdsys.SDO_PC_BLK_TABLE where 0 = 1;

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

We create three tables, note the *high* compression specified in the *blocks* table.

- Loading. For each LAZ file to be loaded we use the *sqlldr* (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
```

Where *[CONTROL FILE]* contains:

```
load data
append into table flat
fields terminated by ','
(
val_d1 float external(10),
val_d2 float external(10),
val_d3 float external(8)
)
```

- After the last file is loaded we create the blocks (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 ('base', 'PC', 'blocks', ptn_params,
                              extent, 0.0001, 3, NULL, NULL, NULL);
    insert into base 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 in our case was set to use slightly faster 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 faster devices (via using table space *INDX*), drop the flat table and compute statistics (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 base compute system statistics for table;
begin
    dbms_stats.gather_table_stats('[USER name]','blocks',
        NULL,NULL,FALSE,'FOR ALL COLUMNS SIZE AUTO',8,'ALL');
end;
begin
    dbms_stats.gather_table_stats('[USER name]','base',
        NULL,NULL,FALSE,'FOR ALL COLUMNS SIZE AUTO',8,'ALL');
end;
```

5.2 Queries

5.2.1 MonetDB

Rectangles:

```
create table results (like flat);
insert into results select *
    from flat where x between minx and maxx and y between miny and maxy;
```

Circles:

```
create table results (like flat);
insert into results select *
    from (select * from flat
         where (x between cx-rad and cx+rad) and (y between cy-rad and cy+rad)) a
    where power(x - cx,2) + power(y - cy,2) < power(rad,2);
```

5.2.2 PostgreSQL

The different query areas are loaded in a table as *POLYGON* types. The query for region 1 is:

```
CREATE TABLE results 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);
```

For the rest of regions it is the same just changing the region identifier.

5.2.3 Oracle

The different query areas are loaded in a table as *POLYGON* types. The query for region 1 is:

```
create table results (x number, y number, z number);
insert into results select pnt.x, pnt.y, pnt.z
    from table (sdo_pc_pkg.clip_pc(
             (select pc from 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
```

For the rest of regions it is the same just changing the region identifier.