Usage guides

point cloud data management systems: DBMS and file-based solutions

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

October 28, 2015

1 Introduction

In this document we describe how to use the available point cloud data management systems (PCDMS). We include both DataBase Management Systems (DBMS) and file-based solutions. In the latter points are stored in files in a certain format, and accessed/processed by specific software. Within DBMS solutions two storage models can be distinguished:

- Blocks model: nearby points are grouped in blocks which are stored in a database table, one row per block
- Flat table model: points are directly stored in a database table, one row per point, resulting in tables with many rows.

The DBMS with native point cloud support based on the blocks model are Oracle Spatial and Graph and PostgreSQL-PostGIS, the latter via the pg_pointcloud extension. Even though in principle any DBMS with spatial functionality can be used as a PCDMS with the flat table model, we tested Oracle Spatial and Graph, PostgreSQL-PostGIS and the column-store MonetDB.

In addition to using native blocks solutions, it is also possible to use third-party blocking solutions. Point Data Abstraction Library (PDAL) is a software for manipulating point cloud data and it is used, in general, as an abstraction layer on management operations. Thus the same operations are available independently on which system (DBMS or file-based) actually contains the data. It has recently released its version 1.0.0.

PDAL has database drivers for Oracle Spatial and Graph, PostgreSQL-PostGIS and SQLite, i.e. tools to load/retrieve data to/from a database using the blocks model. Unfortunately the PDAL point cloud format used for Oracle Spatial and Graph is not fully compatible with the native format provided by the Oracle SDO_PC package. On the other hand, PDAL is compatible with the native format provided by the PostgreSQL-PostGIS pg_pointcloud extension. SQLite does not have native point cloud support, thus the functionality is limited to the PDAL features.

Regarding file-based solutions, all of them use a type of blocks model because points are grouped in files. The most popular choice is to use a combination of tools from the LAStools toolset by Rapidlasso to create a PCDMS. An alternative is to use PDAL, which has also recently released some tools that enable the creation of a stand-alone file-based PCDMS. However, we have not yet tested this approach, thus it is not included in this text.

In this document we include for various PCDMS the following information:

- The procedure to load a set of LAS/LAZ files into the PCDMS. For each PCDMS, there are many parameters to tune and different loading options. In this document we propose the combination of parameters and options that we consider most useful. We will also mention which other parameters or options are available even though these will not be illustrated with examples. Also note that in the provided examples we assume that only X,Y and Z attributes are loaded even though in all the cases more attributes are also possible.
- The procedure to retrieve data, i.e. the SQL statements for DBMS solutions and the command-line commands for file-based solutions. We only show how to query 2D polygons, i.e. get all the points inside a certain 2D region. In the case of DMBS we dump the selected points in a new table (in the case of using the blocks model, the points are extracted from the blocks). For the file-based solutions the selected points are written in a new LAS file. Note that in the query examples we assume the several query regions are pre-loaded in a table called *query_polygons*.

2 pointcloud-benchmark repository: installing and using PCDMS

The repository https://github.com/NLeSC/pointcloud-benchmark has several components. The most important one is the *pointcloud* Python package (in folder *python/pointcloud*) which contains tools to load and query point cloud data in various PCDMS. The original motivation of this Python package is to be used as a platform for the execution of benchmarks for the several PCDMS. However, its loaders and queriers can also be used for more generic purposes and we recommend it because their performance is boosted by using faster loading tools, smarter procedures and parallel (multi-processing) methods which in some cases are not natively possible. For example, parallel loading is not natively possible in the PostgreSQL-PostGIS and LAStools PCDMS's, in such cases the loaders in the *pointcloud* package are much faster than the native loaders (by just adding a simple parallelization layer). Similarly two parallel query algorithms have been developed for the PCDMS's without native parallel querying capabilities. Hence, while this document describes the basics for loading/querying data into/from a PCDMS, for efficiency it is recommended to use the loaders/queries in the *pointcloud* Python package in the *pointcloud-benchmark* repository. The execution of the loaders and queriers is configured through *ini* files (see templates in the *ini* folder) where different parameters can be chosen to tune the loading and querying procedures. For the actual execution use the *load_pc.py* and *query_pc.py* scripts in the *python/pointcloud/run* folder.

The other components of the repository are: (1) *lasnlesc* contains C binary loaders which are used by the PostgreSQL-PostGIS and MonetDB loaders in the *pointcloud* Python package; (2) *sfcnlesc* contains code used by alternative storage models based on Morton space filling curves also used in the *pointcloud* Python package. These are used for research purposes and will not be covered in this document.

For the installation of the several PCDMS's we recommend following the steps described in the official documentation. However, this can be quite cumbersome in some cases. In the *install* folder of the repository there is guideline on how to install the *pointcloud* Python package as well as the several PCDMS's in various Operative Systems (OS's).

3 Usage considerations

Regarding the data loading or preparation, the DBMS systems implementing the blocks model are faster and offer a better compression of the data when compared to the systems implementing the flat table model. However, the latter offers a more flexible data model, the user can modify the table definition or the data values like in any regular database table which is slightly more complicated in the blocks model. In both cases, the integration with other types of data which is important in real applications is straightforward and all the key features of DBMS systems are also present, i.e. data interface though the SQL language, remote access, advanced security, etc.

Nevertheless, the preparation of the point cloud data with LAStools is faster than any DBMS system because the data does not need to be loaded, only resorted and indexed. In addition, if using the LAZ format the storage requirements are also lower when compared to the DBMS solutions. The main disadvantage is that by using a fixed file format the flexibility of the data model is limited to what the specific format allows, for example the LAS format allows only 1 byte for user data.

Regarding the data retrieval, the DBMS solutions implementing the storage model are very performant when dealing with complex queries (larger areas or polygons with many vertices)

and offer the same performance independently of the size of the point cloud. However, the blocks handling adds an overhead in the queries processing which is more noticeable in simple queries as small rectangular queries. On the other hand, the DBMS solutions implementing the flat table model offer a good performance for simple queries but only for small point clouds due to the inefficiency of the native indexing methods for large amounts of spatial data. We explored alternative flat table models based on space filling curves that enormously improve the query performance with larger point clouds.

The file-based system using LAStools offers the best performance for simple queries. We noticed a significant difference between using LAS or LAZ. The queries to LAZ data are slower due to the need of uncompressing the data. In addition, for large point clouds the system requires external aid by a DBMS to maintain its high performance.

The conclusion is that, like in other domains, if an existing file-based solution can entirely fulfill the user requirements it is recommended to use that solution. However, if more flexibility and/or functionality is required the DBMS solutions offer a good alternative and their point cloud support is actively being improved (Oracle, MonetDB and PDAL).

Notwithstanding, in most of the systems we have identified two important missing features. The first one is that some of the systems are single-process-minded. However, this is somehow compensated by the parallel methods developed in the *pointcloud* Python package. The second important missing feature is that there is not efficient support for level of detail in the used data structures of the PCDMS's which is a crucial feature for some applications such as the ones related to the visualization of point clouds. This is the reason why point cloud renderers do not use the solutions described in this document and use their own tailor-made data structures for visualization purposes.

4 PostgreSQL-PostGIS

We have tested the two storage models for point cloud data in PostgreSQL-PostGIS, i.e. the first one is using blocks to group the points and the second one is using a regular flat table. PostgreSQL uses a single process for the loading and querying of the data. As previously mentioned, we recommend using the loader and querier in the *pointcloud* Python package to be able to load and query using parallel methods.

4.1 PostgreSQL-PostGIS using blocks

In this approach the points are grouped in blocks. We use PostgreSQL, PostGIS, pg_pointcloud (the extension developed by Paul Ramsey that adds support for point clouds in PostgreSQL) and PDAL.

4.1.1 Loading

In this section we detail the loading procedure for point clouds in PostgreSQL-PostGIS using blocks. We use PDAL to load the LAS/LAZ files into PostgreSQL-PostGIS. PDAL is also in charge of creating the blocks.

- Create a DB (COMMAND-LINE):

  ```
  createdb pointclouds
  ```

- Initialization (SQL): Load the required extensions and create the table for the blocks.

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

- Loading:
  - First we need to specify the format of the data that we are going to load. This is done by adding to the *pointcloud_formats* table an entry with a XML that contains characteristics of the dimensions to load, mainly the offsets and scales of X,Y,Z to be used when packing the blocks. For each input file it is mandatory to have a valid entry 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
([formatID], [SRID], '<?xml version="1.0" encoding="UTF-8"?>
<pc:PointCloudSchema xmlns:pc="http://pointcloud.org/schemas/PC/"
                     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</pc:description>
    <pc:scale>0.01</pc:scale>
    <pc:offset>0</pc:offset>
    <pc:name>X</pc:name>
    <pc:interpretation>int32_t</pc:interpretation>
    <pc:active>true</pc:active>
  </pc:dimension>
  <pc:dimension>
    <pc:position>2</pc:position>
    <pc:size>4</pc:size>
    <pc:description>Y coordinate</pc:description>
    <pc:scale>0.01</pc:scale>
    <pc:offset>0</pc:offset>
    <pc:name>Y</pc:name>
    <pc:interpretation>int32_t</pc:interpretation>
    <pc:active>true</pc:active>
  </pc:dimension>
  <pc:dimension>
    <pc:position>3</pc:position>
    <pc:size>4</pc:size>
    <pc:description>Z coordinate</pc:description>
    <pc:scale>0.01</pc:scale>
    <pc:offset>0</pc:offset>
    <pc:name>Z</pc:name>
    <pc:interpretation>int32_t</pc:interpretation>
```

```
    <pc:active>true</pc:active>
  </pc:dimension>
  <pc:metadata>
    <Metadata name="compression" type="string">dimensional</Metadata>
  </pc:metadata>
  <pc:orientation>point</pc:orientation>
</pc:PointCloudSchema>');
```

Note that in order to add a new format we have to specify a new *formatID*. This is the value that is referenced in the blocks to specify how to pack/unpack them. Note that we are using dimensional compression. If you do not want compression specify *none*.

If you need to specify another format with more attributes you can use the PDAL tool *pdal info –schema* which will give you information regarding the different attributes.

– After the format has been properly, we use the PDAL tool to load the data from the input file (COMMAND-LINE):

```
pdal pipeline [xmlFile]
```

Where the XML file contains information to drive the loading procedure and it contains:

```
<?xml version="1.0" encoding="utf-8"?>
<Pipeline version="1.0">
<Writer type="writers.pgpointcloud">
    <Option name="connection">host='[DB host]' dbname='[DB name]'
        password='[password]' user='[DB user]'</Option>
    <Option name="table">blocks</Option>
    <Option name="column">pa</Option>
    <Option name="srid">[SRID]</Option>
    <Option name="pcid">[formatID]</Option>
    <Option name="overwrite">false</Option>
    <Option name="capacity">[block size]</Option>
    <Option name="compression">dimensionsal</Option>
    <Option name="output_dims">X,Y,Z</Option>
    <Option name="offset_x">[offsetX]</Option>
    <Option name="offset_y">[offsetY]</Option>
    <Option name="offset_z">[offsetZ]</Option>
    <Option name="scale_x">[scaleX]</Option>
    <Option name="scale_y">[scaleY]</Option>
    <Option name="scale_z">[scaleZ]</Option>
    <Filter type="filters.chipper">
        <Option name="capacity">[block size]</Option>
        <Reader type="readers.las">
            <Option name="filename">[input file path]</Option>
            <Option name="spatialreference">EPSG:[SRID]</Option>
        </Reader>
    </Filter>
</Writer>
```

```
</Pipeline>
```

Note that the *formatID* is the same value than we previously described when inserting a new format. If none was inserted (there was a file inserted with the same set of scales and offsets) then we need to get the *formatID* of the one that was already inserted. For the offsets, scales and compression use the same values as in the format XML. *blocksize* specifies the number of points per block and it has to be specified twice. Our tests indicated that a value between 800 and 3000 is recommended.

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

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

Note that PDAL loads each file independently so this loading procedure, which is called incremental, may produce overlapping blocks in the end if there was some overlapping in the input files.

Use the loader class in *pointcloud.postgres.blocks.Loader* from the *pointcloud* Python package for more efficient loading, i.e. using multiple processes. For large dataset we recommend using *pointcloud.postgres.blocks.LoaderOrdered* which decreases possible de-clustering effects produced by parallel loading.

4.1.2 Querying

For the querying of all regions (rectangles, circles and polygons) we use:

```
CREATE TABLE results AS (
  SELECT * FROM (
    SELECT pc_explode(pc_intersection(pa,geom)) AS qpoint
    FROM blocks, (SELECT geom FROM query_polygons WHERE id = %s) A
    WHERE pc_intersects(pa,geom)) AS qtable);
```

Use the querier class in *pointcloud.postgres.blocks.Querier* from the *pointcloud* Python package for more efficient querying. Parallel querying algorithms are available.

4.2 PostgreSQL-PostGIS flat table

In this approach we use a flat table to store all the points. This approach is not recommended for production since the required storage is much higher than in other flat table approaches (MonetDB or Oracle). This is partly due to the overhead per row that is added by PostgreSQL. In addition the approach does not scale properly in query performance terms, i.e. the same query in a large data set is slower than in a smaller one.

In addition to PostgreSQL, PostGIS and the *lasnlesc* (https://github.com/NLeSC/pointcloud-benchmark/tree/master/lasnlesc) toolset need to be installed.

4.2.1 Loading

In this section we detail the loading procedure for point clouds in the PostgreSQL-PostGIS flat table approach. We use the binary loader developed by NLeSC, which is available in the *lasnlesc* toolset, to import the data from LAS/LAZ to a flat table in PostgreSQL-PostGIS.

- Create a DB (COMMAND-LINE):

```
createdb pointclouds
```

- 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);
```

- Loading: for each LAS/LAZ file that we want have to use the las2pg command-line tool of the *lasnlesc* toolset (COMMAND-LINE):

```
las2pg -s [LAS/LAZ file] --stdout --parse xyz \
 | psql [DB connection params] -c "copy 'flat' from stdin with binary"'
```

- After all the files have been loaded, we create a regular B-tree index on x,y:

```
create index flat_index on flat (x,y) WITH (FILLFACTOR=99);
```

Note that we do not use the PostGIS *ST_Point* data type neither the GIST indexing method. This is due that by using them we would require more storage and more indeixng time without having a significant gain.

Use the loader class in *pointcloud.postgres.flat.LoaderBinary* from the *pointcloud* Python package for more efficient loading, i.e. to use multiple processes.

4.2.2 Querying

In order to speed-up queries we need to use the B-tree index on x,y, i.e. we have to do range selections in x and y. This is directly done in rectangle queries. In circles and generic polygons we do this by a pre-selection step where we filter points within the bounding box of the query geometry.

- For rectangles:

```
CREATE TABLE results AS (
  SELECT * FROM flat
  WHERE x between [minx] and [maxx] AND
        y between [miny] and [maxy]);
```

- For circles:

```
CREATE TABLE results AS (
  SELECT * FROM (
    SELECT * FROM flat
    WHERE (x between [cx]-[rad] and [cx]+[rad]) AND
          (y between [cy]-[rad] and [cy]+[rad])) A
  WHERE (x - [cx])^2 + (y - [cy])^2 < [rad]^2)
```

- For generic regions (polygons):

```
# Extract the bounding box of the query geometry
SELECT st_xmin(geom), st_xmax(geom), st_ymin(geom), st_ymax(geom)
FROM query_polygons WHERE id = [queryId];

CREATE TABLE results AS (
  SELECT * FROM (
    SELECT * FROM flat
    WHERE (x between [minx] and [maxx]) AND
          (y between [miny] and [maxy])) A, query_polygons
  WHERE query_polygons.id = [queryId] AND
       _ST_Contains(geom, st_setSRID(st_makepoint(x,y),[SRID])));
```

Use the querier class in *pointcloud.postgres.flat.Querier* from the *pointcloud* Python package for more efficient querying. Parallel querying algorithms are available.

5 Oracle

We have tested the two storage models for point cloud data in Oracle, i.e. the first one is using blocks to group the points and the second one is using a regular flat table. They both rely on the Oracle Spatial and Graph component of Oracle.

5.1 Oracle using blocks

There are two ways of using Oracle with blocks. The first one is by using the *SDO_PC* native Oracle package which has a native format for the blocks and provides several loading methods and several SQL special functions for the queries. The second way is by using PDAL to take care of the data format, the loading and the queries. The two described ways are not currently compatible (though this has been reported to the involved parties which will hopefully in the future address this issue). We have decided to use PDAL because it offers better performance in storage and in queries. In this case also laz-perf needs to be installed before PDAL to have advanced compression.

5.1.1 Loading

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

```
CREATE USER [userName] IDENTIFIED BY pass;
GRANT UNLIMITED TABLESPACE, CONNECT, RESOURCE, CREATE VIEW TO [userName];
```

- Initialization, i.e. with the recently created user create the table for the blocks and the base table for meta-data (SQL).

```
CREATE TABLE blocks pctfree 0 nologging
  lob(points) store as securefile (nocompress cache reads nologging)
  as SELECT * FROM mdsys.SDO_PC_BLK_TABLE where 0 = 1;

CREATE TABLE base (id number, pc sdo_pc) pctfree 0 nologging;
```

- We use the PDAL tool to load the data from the input file (COMMAND-LINE):

  ```
  pdal pipeline [xmlFile]
  ```

  Where the XML file contains information to drive the loading procedure and it contains:

  ```
  <?xml version="1.0" encoding="utf-8"?>
  <Pipeline version="1.0">
   <Writer type="writers.oci">
     <Option name="debug">false</Option>
     <Option name="verbose">1</Option>
     <Option name="connection">
        [userName]/[password]@//[dbHost]:[dbPort]/[dbName]</Option>
     <Option name="base_table_name">base</Option>
     <Option name="block_table_name">blocks</Option>
     <Option name="compression">true</Option>
     <Option name="store_dimensional_orientation">false</Option>
     <Option name="cloud_column_name">pc</Option>
     <Option name="is3d">false</Option>
     <Option name="solid">false</Option>
     <Option name="overwrite">false</Option>
     <Option name="disable_cloud_trigger">true</Option>
     <Option name="srid">[SRID]</Option>
     <Option name="create_index">false</Option>
     <Option name="capacity">[blockSize]</Option>
     <Option name="stream_output_precision">8</Option>
     <Option name="pack_ignored_fields">true</Option>
     <Option name="output_dims">X,Y,Z</Option>
     <Option name="offset_x">[offsetX]</Option>
     <Option name="offset_y">[offsetY]</Option>
     <Option name="offset_z">[offsetZ]</Option>
     <Option name="scale_x">[scaleX]</Option>
     <Option name="scale_y">[scaleY]</Option>
     <Option name="scale_z">[scaleZ]</Option>
     <Filter type="filters.chipper">
       <Option name="capacity">[blockSize]</Option>
       <Reader type="readers.las">
         <Option name="filename">[input file path]</Option>
         <Option name="spatialreference">EPSG:[SRID]</Option>
       </Reader>
     </Filter>
   </Writer>
  </Pipeline>
  ```

- Add a primary key in the blocks table (SQL):

  ```
  ALTER TABLE blocks add constraint BLOCKS_PK primary key (obj_id, blk_id);
  ```

- Compute the 2D bounding box of all the points that we have inserted. You can use the *getPCFolderDetails* method from the *lasops* module in the *pointcloud* Python package.
- Create the index for the blocks (SQL):

```
insert into USER_SDO_GEOM_METADATA values ('blocks','BLK_EXTENT',
sdo_dim_array(sdo_dim_element('X',[minX],[maxX],[tolerance]),
              sdo_dim_element('Y',[minY],[maxY],[tolerance])),[SRID]);

create index BLOCKS_SIDX on blocks (blk_extent)
indextype is mdsys.spatial_index
parameters ('layer_gtype=polygon sdo_indx_dims=2 sdo_rtr_pctfree=0');
```

- Compute statistics (SQL):

```
ANALYZE TABLE blocks  compute system statistics for table;

BEGIN
    dbms_stats.gather_table_stats('[userName]','BLOCKS',NULL,NULL,FALSE,
                                  'FOR ALL COLUMNS SIZE AUTO',8,'ALL');
END;
```

Note that PDAL loads each file independently so this loading procedure, which is called incremental, may produce overlapping blocks in the end if there was some overlapping in the input files.

Use the loader class in *pointcloud.oracle.blocks.incremental.LoaderPDAL* from the *pointcloud* Python package for more efficient loading, i.e. using multiple processes. In the *pointcloud* Python package there are several loaders for the other way (SDO_PC) of using point cloud blocks in Oracle.

5.1.2 Querying

Since we chose PDAL blocks, we also need to use PDAL to retrieve the data (PDAL blocks are not compatible with SDO_PC methods to retrieve point cloud data) (COMMAND-LINE).

```
pdal pipeline [xmlFile]
```

Where the XML file contains information to drive the retrieval procedure and it contains:

```
<?xml version="1.0" encoding="utf-8"?>
<Pipeline version="1.0">
  <Writer type="writers.las">
    <Option name="filename"> [output file name] </Option>
    <Filter type="filters.crop">
        <Option name="polygon">[WKT description of query region]</Option>
        <Reader type="readers.oci">
          <Option name="query">
SELECT l."OBJ_ID", l."BLK_ID", l."BLK_EXTENT",
       l."BLK_DOMAIN", l."PCBLK_MIN_RES",
```

```
       l."PCBLK_MAX_RES", l."NUM_POINTS",
       l."NUM_UNSORTED_POINTS", l."PT_SORT_DIM",
       l."POINTS", b.pc
FROM blocks l, base b, query_polygons g
WHERE
    l.obj_id = b.id
    AND
    SDO_FILTER(l.blk_extent,g.geom) = 'TRUE' AND g.id = [queryId]
          </Option>
          <Option name="connection">
             [userName]/[password]@//[dbHost]:[dbPort]/[dbName]</Option>
          <Option name="spatialreference">EPSG:[SRID]</Option>
        </Reader>
    </Filter>
  </Writer>
</Pipeline>
```

5.2 Oracle flat table

In this approach we use a flat table to store all the points.

5.2.1 Loading

There are two methods to load the data in Oracle flat approach. The first one is by using the Oracle *sqlldr* tool which requires the LAS/LAZ files to be converted to ASCII which can be done with LAStools *las2txt*. The second one is by using an external table loader with a preprocessor script to convert a LAS to a sqlldr input stream. The advantage of the second method is that it can use multiple processes natively so this is the one we use. The disadvantage is that LAZ files are not compatible.

The points are inserted in an Index-Organized table (IOT), contrary to a normal table which requires an index to be created. This is done to decrease the storage requirements and to have the data clustered. The disadvantage is that parallel queries on IOT's do not seem to work (this issue has been reported and it is under investigation).

- Create a user to allocate the new table with a super-user (SQL).

  ```
  CREATE USER [userName] IDENTIFIED BY pass;
  GRANT UNLIMITED TABLESPACE, CONNECT, RESOURCE, CREATE VIEW TO [userName];
  ```

- We need to create a Oracle directory for the input data and give permissions to the user, this also has to be done with a super-user (SQL):

  ```
  CREATE DIRECTORY INPUT_DATA_DIR as '[input data folder]';
  GRANT READ on DIRECTORY INPUT_DATA_DIR to ORACLE_FLAT;
  ```

- Initialization(SQL), i.e. with the recently created user create the external table:

  ```
  CREATE TABLE POINTS_EXT (VAL_D1 NUMBER, VAL_D2 NUMBER, VAL_D3 NUMBER)
  organization external
  ```

```
(
type oracle_loader
default directory INPUT_DATA_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 [number processes] reject limit 0;
```

Note that two Oracle directories are also required, *EXE_DIR* and *LOG_DIR*. The first one must contain the preprocessor script and the second will contain logs. The preprocessor script looks like:

```
#!/bin/sh
${ORACLE_HOME}/jdk/bin/java -classpath ${ORACLE_HOME}/md/jlib/sdoutl.jar oracle.spati
```

- Load the points into an IOT (SQL):

```
CREATE TABLE POINTS
(VAL_D1, VAL_D2, VAL_D3,
    constraint POINTS_PK primary key (VAL_D1, VAL_D2, VAL_D3))
    organization index
    pctfree 0 nologging
    parallel [number processes]
as
    SELECT VAL_D1, VAL_D2, VAL_D3 FROM POINTS_EXT;
```

- Compute statistics (SQL):

```
ANALYZE TABLE points compute system statistics for table;

BEGIN
    dbms_stats.gather_table_stats('[userName]','POINTS',NULL,NULL,FALSE,
                                    'FOR ALL COLUMNS SIZE AUTO',8,'ALL');
END;
```

5.2.2 Querying

In order to properly use the IOT the queries first need to do a range selection, i.e. a pre-filter on the bounding box of the query region. This is not necessary in rectangular regions since they are already a range selection.

- For rectangles:

```
CREATE TABLE results AS (
  SELECT * FROM points
  WHERE val_d1 between [minx] and [maxx] AND
        val_d2 between [miny] and [maxy]);
```

- For circles:

```
CREATE TABLE results AS (
  SELECT * FROM (
    SELECT * FROM points
    WHERE (val_d1 between [cx]-[rad] and [cx]+[rad] AND
          (val_d2 between [cy]-[rad] and [cy]+[rad]) A
  WHERE POWER(val_d1 - [cx],2) + POWER(val_d1 - [cy],2)
        < POWER([rad],2))
```

- For generic regions:

```
DECLARE
  bbox sdo_geometry;
BEGIN
  SELECT sdo_geom_mbr (geom) into bbox from query_polygons where id = [queryId];
  execute immediate 'create table results as
  select val_d1, val_d2, val_d3
  from table ( sdo_PointInPolygon (
    cursor (
      select val_d1, val_d2, val_d3 from points
        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 = [queryId]),
    0.0001,
    NULL
  ))';
END;
```

Use the querier class in *pointcloud.oracle.flat.Querier* from the *pointcloud* Python package for more efficient querying. Parallel querying algorithms are available.

6 MonetDB

The column-store MonetDB has recently extended its GeoSpatial extension to deal with point cloud data. The points are stored in a flat table, i.e. one row per point. However, we use partitions to enhance data retrieval operations. MonetDB automatically uses parallel processes when possible.

6.1 Loading

We use the *las2col* tool from the *lasnlesc* toolset so this has to be installed. The steps to load point cloud data in MonetDB are:

- For each partition we create a file which contains the paths to the LAS/LAZ files that will be imported in the partition. Choose a distribution of files in partitions as spatially coherent as possible, i.e. files which contain nearby points should go to the same partition.
- For each partition we use the las2col to convert the LAS/LAZ data to columnar files which can be directly imported to MonetDB (COMMAND-LINE):

```
las2col -f [partition file list] [base name for temporal data of partition] \
         --parse xyz --num_read_threads [number of processes]
```

- Create a DB (COMMAND-LINE):

```
monetdb create pointclouds
monetdb release pointclouds
```

- Create the tables where we will import each partition (SQL):

```
create table [partition 1] (x decimal(9,2), y decimal(9,2), z decimal(9,2));
...
```

- Create the merge table, i.e. the table that links the partitions, and add the partitions to it (SQL):

```
create merge table all_points (x decimal(9,2), y decimal(9,2), z decimal(9,2));
alter table all_points add table [partition 1];
...
```

- We import the data for the partitions (SQL):

```
COPY BINARY INTO [partition 1] from
('[base name for temporal data of partition]_col_x.dat',
 '[base name for temporal data of partition]_col_y.dat',
 '[base name for temporal data of partition]_col_z.dat');
 ...
```

- Set the partitions to read-only (SQL):

```
alter table [partition 1] set read only;
...
```

- Build imprints indexes for all partitions (SQL):

```
select x from [partition 1] where x between 0 and 1;
select y from [partition 1] where y between 0 and 1;
select z from [partition 1] where z between -30000 and -1000;
analyze sys.[partition 1] (x, y, z) minmax;
```

We recommend using the loader in *pointcloud.monetdb.LoaderBinary* from the *pointcloud* Python package for more efficient loading. This loader creates spatially-coherent partitions to further enhance data retrieval operations. The input files will be imported to one or other partition depending on their 2D spatial extent. The 2D bounding boxes of the data in the partitions will be similar to a 2D grid but with some overlap between partitions.

6.2 Querying

In order to speed-up queries we need to use the imprints index on x,y,z, i.e. we have to do range selections in x, y and z. This is directly done in rectangle queries. In circles and generic polygons we do this by a pre-selection step where we filter points within the bounding box of the query geometry.

- For rectangles:

```
CREATE TABLE results AS
  SELECT * FROM all_points
  WHERE x between minx and maxx AND
        y between miny and maxy
WITH DATA;
```

- For circles:

```
CREATE TABLE results AS (
  SELECT * FROM (
    SELECT * FROM all_points
    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)
WITH DATA;
```

- For generic regions (polygons):

```
# Extract the bounding box of the query geometry
SELECT mbr(geom) FROM query_polygons WHERE id = [queryId];

CREATE TABLE results AS
  SELECT * FROM (
    SELECT * FROM all_points
    WHERE (x between [minx] and [maxx]) AND
          (y between [miny] and [maxy])) A
  WHERE contains(GeomFromText('[WKT description of query region]',
                 [SRID]),x,y)
WITH DATA;
```

7 LAStools

Using a combination of tools from the LAStools toolset it is possible to create a PCDMS that can directly deal with LAS/LAZ files. Concretely we use the open-source *lasindex* and *lasmerge*

and the license-required *lassort* and *lasclip*. Using the LAS format will offer the best query performance but it requires around ten times more storage compared to LAZ format.

7.1 Loading

Technically speaking the data is not loaded but prepared. In order to achieve the best performance in the queries the data needs to be sorted and indexed. We also use a DB to store the extents of the various files, this is used to speed up the queries.

- We create a folder for the sorted data (COMMAND-LINE):

  ```
  mkdir [output folder]
  ```

- For each file in the input folder we run *lassort* which create a new file with the data in the input file properly sorted in Morton order (COMMAND-LINE):

  ```
  lassort.exe -i [input file 1] -o [output folder]/[file 1]
  ...
  ```

- For each sorted file we create an index file (LAX) with *lasindex* (COMMAND-LINE):

  ```
  lasindex -i [output folder]/[file 1]
  ...
  ```

  This creates a LAX file in the same folder where the sorted file is stored.

- Finally we create a DB which will contain the extents of the sorted files. We use a PostgreSQL-PostGIS but other DBs can be used. This is recommended when we have a considerable number of files (more than few hundreds).

  - Create a DB (COMMAND-LINE):

    ```
    createdb files_extents
    ```

  - Add PostGIS extension and create table for the extents (SQL):

    ```
    CREATE EXTENSION postgis;
    CREATE TABLE extents (
      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,[SRID]));
    ```

  - For each sorted file we need to insert a row in the table (SQL) with information about its contents and extent. You can use the *getPCFileDetails* method from the *lasops* module in the *pointcloud* Python package.

```
INSERT INTO extents
(id,filepath,num,scalex,scaley,scalez,offsetx,offsety,offsetz,geom)
VALUES
([counter], [output folder]/[file 1], [number of points],
[scale X], [scale Y], [scale Z], [offset X], [offset Y], [offset Z],
ST_MakeEnvelope([minX], [minY], [maxX], [maxY], [srid]));
...
```

– Create an index on the extents (SQL):

```
create index ON extents using GIST (geom);
```

Use the loader class in *pointcloud.lastools.Loader* from the *pointcloud* Python package for more efficient loading, i.e. using multiple processes to simultaneously sort and index files and insert extents in the DB.

7.2 Querying

In order to decrease the number of files that the LAStools tools need to read we do, in all the cases, a query to the DB to get a list of files that overlap with our query region.

```
psql files_extents -t -A -c \
"SELECT filepath FROM extents e,query_polygons q \
 WHERE  ST_Intersects(q.geom,e.geom ) and q.id = [queryId]" \
> file_selection
```

Then, depending on the type of query:

- For rectangles:

```
lasmerge -lof file_selection -inside [minx] [miny] [maxx] [maxy] \
  -o results.las
```

- For circles:

```
lasmerge -lof file_selection -inside_circle [cx] [cy] [rad] -o results.las
```

- For generic region. we need to convert the query region to a ShapeFile:

```
pgsql2shp -f query.shp -h localhost -u [user] -P [pass] dbname
"SELECT ST_SetSRID(geom, [SRID]) FROM query_polygons where id = [queryId];"
lasclip.exe -lof file_selection -poly query.shp -merged -o results.las
```

LAStools *lasclip* has an option to use multiple cores but it is not really efficient and its usage is not recommended.