Towards Storing Point Clouds in PostgreSQL

Seminar Database Systems

Master of Science in Engineering
Major Software and Systems
HSR Hochschule für Technik Rapperswil
www.hsr.ch/mse

Supervisor: Prof. Stefan Keller

Author: Michel Ott

Date: January 15, 2012
Abstract

A lot of efforts are in progress to store point cloud data in a database. Especially Oracle introduced in 11g a new data type called SDO_PC, which has the ability to split point cloud data into chunks and storing these chunks in a separate table including additional information like the minimum and maximum resolution and so on. Microsoft’s SQL Server and PostgreSQL missed to provide a dedicated point cloud data type. Both database’s give the ability to run only spatial functions; on the one hand with the spatial extender on SQL Server 2008 and on the other hand by using the PostGIS extension on PostgreSQL.

The difference between a dedicated point cloud data type and the use of the spatial data types are, that the data of a point cloud data type are stored optimally on the disk, means that unnecessary disc accesses or exhaustive RAM load are avoided. This paper discusses some approaches how does a dedicated point cloud data type can be implemented in PostgreSQL and makes some succession which approach should be preferred. In addition it gives an example how does PostgreSQL can be extended so that the ramp up to implement the desired approach is minimal.

The only feasible approach is to split a LAS file / point cloud data into a set of different blocks / chunks. The blocks should be stored in a separate point cloud data table and referenced to its meta data table, which holds additional information like the the whole resolution and so on. In addition, PostgreSQL functions need to be provided to do spatial analysis on the point cloud data. Hence the PostGIS extension can be reused, so that only some wrapper functions need to be created, which calls the appropriate PostGIS C libraries. The approach has the benefit, that not all points need to be scanned in PostgreSQL and that we not run against a data type limit. Hence, the most of PostgreSQL’s standards has been used without violating any design issue of PostgreSQL.

**Keywords:** Point Cloud, PostgreSQL, C-functions, Extension, Implementation, Concept, storing massive data volume, large objects, pg_largeobject
# Contents

1 Introduction .................................................. 2  
1.1 Vision ...................................................... 2  
1.2 Scope ...................................................... 2  
1.3 Restrictions on the scope .................................. 2  

2 Point Cloud (LiDAR) ........................................... 3  
2.1 Formats of LiDAR Point Clouds .............................. 3  
2.2 LAS Format .................................................. 4  

3 State of the Art ................................................ 6  
3.1 Libraries .................................................... 6  
3.1.1 libLAS ................................................... 6  
3.1.2 PDAL ...................................................... 7  
3.2 Databases .................................................... 7  
3.2.1 Oracle 11g ............................................... 7  
3.2.2 Microsoft SQL Server 2008 ............................ 11  
3.2.3 PostgreSQL and PostGIS .............................. 11  

4 PostgreSQL Extending SQL ..................................... 13  
4.1 Terms ....................................................... 13  
4.2 C-Language Functions ...................................... 14  
4.3 Compiling and Linking ..................................... 15  
4.4 Register C-Functions ...................................... 16  
4.5 How to use it ................................................ 16  

5 Implementation Approaches .................................... 17  
5.1 Approach 1: One Tuple .................................... 17  
5.2 Approach 2: Each Point in a Tuple ......................... 18  
5.3 Approach 3: Splitting Data ................................ 20  

6 Critical acclaim ................................................. 22  

References ....................................................... 23  

A PostgreSQL reverse string extension .......................... 24  
A.1 C function .................................................. 24  
A.2 Control file ................................................. 24  
A.3 SQL file ..................................................... 25  
A.4 Compile script ............................................. 25  

B Lists ................................................................ 26  
B.1 List of Figures ............................................... 26  
B.2 List of Tables ................................................ 26  
B.3 Listings ....................................................... 26  
B.4 Nomenclature ............................................... 27
Introduction

1.1 Vision

As part of this term paper a project evolved to evaluate, propose, and if time is available implement a prototype for storing point cloud data to PostgreSQL also known as Postgres. PostgreSQL does currently not support point cloud data out-of-the-box, that means, it does not provide a dedicated data type for this type of data and special functions at all. Furthermore it is not possible to query a specific n-dimensional shape out of a set of data. In this paper we keep ourself on 3-dimensional boxes means on cuboid to select a subset out of a set of data.

1.2 Scope

The goal of this paper is to show the following items:

1. What is currently on the market, and how does this solutions work?
2. How can PostgreSQL be extended to fulfill point cloud suitable data type?
3. Possible approaches to store point clouds in PostgreSQL and discussion of needed functions.

1.3 Restrictions on the scope

All the following explanations are based on the LAS format captured by LiDAR instruments.

---

1 PostgreSQL is an open source relational database, even an object-relational database. [http://www.postgresql.org/]
A point cloud is just a bunch of points. Although it is sometimes useful to talk about point clouds in any dimensional space, but usually we talk in the 3-dimensional space. The recorded data are discrete and needs enormous computing power because of the large data sets. Point clouds are recorded by so-called laser scanners that analyzes a real-world object or environment by collecting data on its shape and possibly its appearance like the color. Extremely short laser pulse hits an obstacle by which it is reflected and is then received by the scanner. The measured time of the impulse is proportional to the distance between scanner and object. During the recording procedure, the head of the scanners can rotate by 360° along the horizontal pane.

In this paper we take a point cloud produced by LiDAR instruments, which is a very good representation for our goal described in chapter 1. LiDAR is an acronym and stands for "Light Detection and Ranging". The pulse being sent and received can be equated to the distance and stored in a discrete 3-dimensional data set where the coordinates are calculated from the angles and the distance to the scanner. That means, each points stores at least the the x, y, and z coordinates as well as the angle of the laser.

Infoterra (2011) describes when using such as LiDAR instrument on an aircraft, the points can be enhanced by the coordinates of a high accuracy GPS positioning and inertial attitude measuring instrument. Together with a powerful computing software, it allows to produce a very accurate model of the ground.

2.1 Formats of LiDAR Point Clouds

As described before LiDAR in its raw form is a series of points stored as x, y, z coordinates where x and y can be longitude and latitude and z is the elevation in meters. A simple ASCII file could be enough where each line has a coordinate (x,y,z) separated by a delimiter e.g. comma, tab, etc. to represent the data.

Another representation for LiDAR point clouds is as contours lines. They are commonly stored in vector formats and derived from a pre-constructed digital elevation model (DEM) of triangulated irregular network (TIN). These contours (see Figure 2.1) are the most commonly used representations for elevations and are found, among other, on our national swisstopo maps.

DEM data are commonly stored in raster files by using point files which can be interpolated using many different techniques from simple (e.g. nearest neighbor) to complete gridding (kriging) routines to create different surface types. According to Schmid et al. (2008, p. 16) the most common are surfaces created from the TIN or the Inverse Distance Weighted (IDW) routines.
Contours directly derived from LiDAR data are accurate but not clean in the sense of abstraction. They often need a level of interpolation to convert all the points to smoothen lines. The drawback of this process is that because of the higher level of abstraction a lot of information is lost and therefore also the accuracy of it.

2.2 LAS Format

Giving that the format can be defined depending on the usage, a committee called the imaging & geospatial information society (ASPRS) began to define an interchange format of 3-dimensional point cloud data called LAS. In spite of the fact that it was primarily developed for exchange of LiDAR point cloud data, it supports any 3-dimensional x,y,z records. LAS is a binary file format that maintains specific LiDAR information while not being too complex. The benefit compared to an ASCII file is, that it performs much better with respect to interpretation of elevation data. In addition it is much smaller, even for small data, because of its binary representation.

The binary LAS format in little-endian contains of three blocks. The first one is a header block, then the variable length records and at the end the point data records.

<table>
<thead>
<tr>
<th>public header block</th>
<th>variable length records</th>
<th>point data records</th>
</tr>
</thead>
</table>

Table 2.1: LAS Format Definition
Source: ASPRS (2010, p. 4)

In addition a LAS file can contain waveform data and it changes the format to the following:
<table>
<thead>
<tr>
<th>Public header block</th>
<th>Variable length records including waveform packet descriptions (up to 255)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Point data records</td>
</tr>
<tr>
<td></td>
<td>Extended variable length record (waveform data packets)</td>
</tr>
</tbody>
</table>

Table 2.2: LAS Format Definition Containing Waveform Data  
Source: ASPRS (2010, p. 4)

The public header block consist of generic data as the x,y,z scale factors, the point data record length, etc. and is optional but have to be zero filled if not stated. The variable length record contains variable types of data including projection information, metadata, waveform packet information and user application data. This records can be seen as metadata for the included points. At the end, the point data records, stores all the points x,y,z coordinates and additional information like intensity, classification, etc. whereas all points have the be of the same type. For further information about the LAS structure consider ASPRS (2010) on page 5ff.
As mentioned in chapter 2 we are interested in LiDAR point clouds especially in file format LAS, a binary interchangeable format. Currently many approaches are ongoing that take the LAS format into account. Some are discussing how can the LiDAR point cloud be optimized in respect of the file size and visualization, because of the high memory utilization of the huge LAS files, and others are discussing how does LAS format can be stored and analyzed in an efficient way.

In the following chapter we have a short look on the two main open source libraries libLAS and PDAL and which database implementation are on the market to store LAS files means LiDAR point clouds.

3.1 Libraries

For reading and manipulating LAS files there are not that much libraries on the market. The most of the libraries do not work or do not support the needed features. The two most interesting are libLAS and PDAL. Sadly, PDAL is still in development and does not support LAS, however, it could be a real alternative to libLAS in the future.

3.1.1 libLAS

libLAS is a C/C++ library for reading and writing the LAS format. It provides multiple application programming interfaces (API such as for C, C++, .NET, Ruby, and Python and is cross-platform compatible means runs on Windows as well as on Mac OS X and Linux and supports little- and big-endian architectures. Furthermore it can be linked against libgeotiff to set and get the spatial reference systems of LAS files using the simple proj.4 coordinate system definitions. Alternatively it is possible to link against GDAL to manipulate spatial reference systems using open geospatial consortium (OGC) well know text (WKT) or the vertical coordinate system.

Excursus: First we need to internalize what a datum in the context of surveying and geodesy is. A datum is a reference point or surface on the earth against which position measurements are made and a model for computing positions is associated. A datum is decided into horizontal and vertical from which the corresponding coordinate systems are derived.

A horizontal datum is a model to measure positions on the earth like the geographic position of the mountain Eiger. Each point on the earth’s surface can be stored in the latitude and longitude or another coordinate system, whereas a specific point can have substantially different coordinates, depending on the datum used. Around the world they are a lot of different horizontal datums to fulfill the local conditions that are referenced to local reference points. Modern datums are intended to cover large areas and are based on increasingly accurate measurements.

A vertical datum is used to measure the elevation or underwater depth of points on the Earth’s surface which is based on sea levels (tidal), the Earth’s gravimetric (geoid), or on the same ellipsoid used for computing the horizontal datum (geodetic). The elevations are often used in height above mean sea level at which the mean sea level is calculated by the arithmetic mean of the hourly water elevation taken over a specific 19 year cycle. However, zero elevation in a country does not have to be the same zero elevation
libLAS can be used in different programming languages like this:

```cpp
#include <liblas/liblas.hpp>
#include <fstream> // std::ifstream
#include <iostream> // std::cout
std::ifstream ifs;
ifs.open("file.las", std::ios::in | std::ios::binary);
liblas::ReaderFactory f;
liblas::Reader reader = f.CreateWithStream(ifs);

// access members of the public header block
liblas::Header const& header = reader.GetHeader();
std::cout << "Compressed: \"" << header.Compressed() ? "true\" : "false\";
std::cout << "Signature: \"" << header.GetFileSignature() << "\n";
std::cout << "Points count: \"" << header.GetPointRecordsCount() << "\n";
// iterate through point records
while (reader.ReadNextPoint()) {
    liblas::Point const& p = reader.GetPoint();
}
```

Listing 3.1: Example of using libLAS in C++

### 3.1.2 PDAL

PDAL is an open source library quoted under BSD license for translating and manipulating point cloud data of various formats. As opposed to libLAS which only reads and manipulates LAS formats, PDAL focuses on "reading, writing, and translating point cloud data form the ever-growing constellation of data formats that are being developed for working with multi-dimensional emitted-pulse scanning systems" (Butler, Gerlek, et al., n.d.) and is analogous to the GDAL raster library.

Currently PDAL is still in development and not an alternative for libLAS.

### 3.2 Databases

Three-dimensional data tend to be relatively large, typically of the order of hundred of thousand or more of points for each scan respectively LAS file. Storing such huge 3-dimensional point clouds in a geometry multipoint collection type may not be an ideal solution for at least two reasons:

1. It exceeds the maximal amount of ordinates (typically limited to 1 million ordinates)

2. The entire set of points is stored as single collection, means it is not possible to search and access a point set by area of interest. Every time a query is applied on this collection, the whole data need to be loaded into memory what is not efficient.

#### 3.2.1 Oracle 11g

To address the above issues 1 and 2 Oracle investigated into a new data type called SDO_PC (PC is short for point cloud). The SDO_PC storage depicts the structure shown in figure 3.1.
In general two tables are needed, which separates the logical from the physical structure. The base table holds data of type SDO_PC which holds the metadata associated with the point cloud also known as logical structure. The points of the point cloud are, however, divided into subsets and stored as multiple rows of a separate table, which is the corresponding block table of SDO_PC_BLK type which represents the physical structure. Each tuple in the block table has an object identifier, a block identifier, a geometry, the resolution of the subset and the point as binary large object (BLOB) type.

As you can see in listing 3.4 the block table need to be created by yourself like the schema of the system table MDSYS.SDO_PC_BLK_TABLE. It is also possible to create the columns of the block table by your own, but by linking the system table, it is guaranteed that the table works fine with the data type even on a upgrade of Oracle.

**SDO_PC type:** SDO_PC type can be seen as a complex data type. It is not a data type like CHAR which holds a single value but rather a structure of the format defined in table 3.1.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE_TABLE</td>
<td>VARCHAR2(70)</td>
<td>Name of the base table containing a column of type SDO_PC.</td>
</tr>
<tr>
<td>BASE_TABLE_COL</td>
<td>VARCHAR2(1024)</td>
<td>Name of the column of type SDO_PC in the base table.</td>
</tr>
<tr>
<td>PC_ID</td>
<td>NUMBER</td>
<td>ID number for the point cloud.</td>
</tr>
</tbody>
</table>
### 3.2. DATABASES

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLK_TABLE</td>
<td>VARCHAR2(70)</td>
<td>Name of the table that contains information about each block in the point cloud.</td>
</tr>
<tr>
<td>PTN_PARAMS</td>
<td>VARCHAR2(1024)</td>
<td>Parameters for partitioning the point cloud.</td>
</tr>
<tr>
<td>PC_EXTENT</td>
<td>MDSYS.SDO_GEOMETRY</td>
<td>Geometry object representing the spatial extent of the point cloud (the minimum bounding object enclosing all object in the point cloud).</td>
</tr>
<tr>
<td>PC_TOL</td>
<td>NUMBER</td>
<td>Tolerance value for points in the point cloud.</td>
</tr>
<tr>
<td>PC_TOT_DIMENSIONS</td>
<td>NUMBER</td>
<td>Total number of dimensions in the point cloud. Includes spatial dimensions and any non spatial dimensions, up to a maximum total of 9.</td>
</tr>
<tr>
<td>PC_DOMAIN</td>
<td>MDSYS.SDO_ORGSCL_TYPE</td>
<td>Not currently used.</td>
</tr>
<tr>
<td>PC_VAL_ATTR_TABLES</td>
<td>MDSYS.SDO_STRING_ARRAY</td>
<td>Array object specifying the names of any value attribute tables for the point cloud.</td>
</tr>
<tr>
<td>PC_OTHER_ATTRS</td>
<td>SYS.XMLTYPE</td>
<td>XML type object specifying any other attributes of the point cloud.</td>
</tr>
</tbody>
</table>

Table 3.1: Structure of Oracle’s SDO_PC type  
Source: Godfrind (2009, p. 12f)

**SDO_PC_BLK type:** The SDO_PC_BLK type is not a data type for a single column, it is rather a table schema that need to be assigned to the `CREATE TABLE` statement like in listing 3.3 or like in 3.4.

The block table structure is defined as in table 3.2

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJ_ID</td>
<td>NUMBER</td>
<td>ID number of the point cloud object.</td>
</tr>
<tr>
<td>BLK_ID</td>
<td>NUMBER</td>
<td>ID number of the block.</td>
</tr>
<tr>
<td>BLK_EXTENT</td>
<td>MDSYS.SDO_GEOMETRY</td>
<td>Spatial extent of the block.</td>
</tr>
<tr>
<td>BLK_DOMAIN</td>
<td>MDSYS.SDO_ORGSCL_TYPE</td>
<td>Not currently used.</td>
</tr>
<tr>
<td>PCBLK_MIN_RES</td>
<td>NUMBER</td>
<td>Minimum resolution level at which the block is visible in a query. The block is retrieved only if the query window intersects the spatial extent of the block and if the minimum - maximum resolution interval of the query. Usually, lower values mean farther from the view point, and higher values mean closer to the view point.</td>
</tr>
<tr>
<td>PCBLK_MAX_RES</td>
<td>NUMBER</td>
<td>Maximum resolution level at which the block is visible in a query.</td>
</tr>
<tr>
<td>NUM_POINTS</td>
<td>NUMBER</td>
<td>Total number of points in the POINTS BLOB.</td>
</tr>
</tbody>
</table>
### Table 3.2: Structure of Oracle's block table

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_UNSORTED_POINTS</td>
<td>NUMBER</td>
<td>Number of unsorted points in the POINTS BLOB.</td>
</tr>
<tr>
<td>PT_SORT_DIM</td>
<td>NUMBER</td>
<td>Number of the dimension (1 for the first dimension, 2 for the second dimension, etc.) on which the points are sorted.</td>
</tr>
<tr>
<td>POINTS</td>
<td>BLOB</td>
<td>Contains the points.</td>
</tr>
</tbody>
</table>

After understanding the structure of the tables and defining the tables, the point cloud data can be initialized as mentioned in listing 3.5.

```sql
INSERT INTO base_table (pc) VALUES (
  SDO_PC_PKG.INIT(
    BASETABLE => 'base_table', -- Name of the base table
    BASECOL => 'pc', -- The column in which this SDO\_PC object is being inserted
    BLKTABLE => 'block_table', -- The block table that stores the blocks of the SDO\_PC object
    PTN_PARAMS => 'BLK\_CAPACITY=1000', -- The maximum block capacity of each point cloud block
    PC_TOL => 0.005, -- The tolerance associated with the SDO\_PC object
    PC_TOT_DIMENSIONS => 3, -- The total number of dimensions that are stored in the SDO\_PC object
    PC_EXTENT =>
      SDO\_GEOMETRY (2003, 4326, NULL, SDO\_ELEM\_INFO\_ARRAY (1,1003,3),
      SDO\_ORDINATE\_ARRAY (-74, 40, -73, 41)
      ) -- The extent of the SDO\_PC object specified as an SDO\_GEOMETRY

);```

Listing 3.5: Oracle’s initialization statement of point cloud data

Note that the **PC_EXTENT** specifies how many dimensions are to be used for partitions. Means in your case we use for the partitioning the first 2 dimension of the input data set, whereas in the SDO\_PC 3 dimensions are stored.

Next, we need to load the data into a temporary input table so that afterwards Oracle can partition the point cloud. The external file can be shown as table and defined as in listing 3.6.

```sql
CREATE TABLE input_points (
  rid VARCHAR2 (40),
  val_d1 NUMBER,
  val_d2 NUMBER,
  val_d3 NUMBER
)
ORGANIZATION EXTERNAL (  
  TYPE ORACLE_LOADER  
  DEFAULT DIRECTORY data_files ACCESS PARAMETERS (  
    FIELDS TERMINATED BY ",", (rid, val_d1, val_d2, val_d3 )  
  )  
  LOCATION ('input_points.dat')  
);```

Listing 3.6: Defining external file as input table

Lastly we now need to create the point cloud blocks as in listing 3.7.

```sql
SDO\_PC\_PKG\_CREATE\_PC(
  pc, -- Initialized PointCloud object
  'INPTAB' -- Name of input table to ingest into the point cloud
  'RESTAB' -- Name of output table that stores the points (with addl. Columns ptn_id, pt_id)  
);```

Listing 3.7: Create point cloud blocks
### 3.2. DATABASES

**Issues:** Oracle does not provide an update or insert mechanism. If you want to update or insert a new point into an existing point cloud set, the whole point cloud blocks need to be generated again by invoking the `CREATE_PC` procedure.

#### 3.2.2 Microsoft SQL Server 2008

Microsoft delivered in the SQL Server 2008 spatial support for the geodetic spatial and planar spatial model, by providing a `geography` data type for geodetic spatial data, and a `geometry` data type for planar spatial data. Both are implemented as Microsoft .Net Framework Common Language Runtime (CLR), and can be used to store different kinds of geographical elements such as points, lines, and polygons (Microsoft, 2008). Furthermore properties and methods are provided to do spatial operations such as calculating the distance between to points or finding features that intersect one another.

The spatial data types can be used as decreed in listing 3.8

```sql
-- Defining data type
CREATE TABLE Districts (
  DistrictId int IDENTITY (1,1),
  DistrictName varchar(20),
  DistrictGeo geography -- OR geography
);

-- Doing a insert
INSERT INTO Districts (DistrictName, DistrictGeo) VALUES ('Downtown',
  geography::STGeomFromText('POLYGON(((0,0),(0,150),(150,150),(150,0),(0,0)))',0)
);
```

Listing 3.8: Microsoft SQL spatial data type

Unfortunately Microsoft SQL Server 2008 does not support a dedicated point cloud support like Oracle SDO_PC data type does.

**Excursus:** Describing a location on a planetary surface brings up the problem that planets are not flat. However, the very complex object "Earth" can be approximated by an oblate spheroid, a flattened sphere. An oblate spheroid is a surface of revolution obtained by rotating an ellipse about its minor axis, means for the earth that the equatorial radius is greater than the polar radius Weisstein (n.d.). Locations are normally described in terms of their latitude and longitude on the manifested globe, which is measured in degrees from the equator and the international date-line respectively.

This approach to model geographic locations is called geodetic model, and provides an accurate way to define locations on the globe. Today a lot of different geodetic models are in use like the Swiss Terrestrial Reference System 1995 (CHTRS95).

While geodetic model provides the most accurate way to represent geographic features, calculating distances is more difficult, because we need to work with an ellipsoid and taking planetary curvature into account. It is much easier to work with two-dimensional surfaces, or planes. When working with such a planar model, a projection is needed to flatten the locations to the spheroid.

#### 3.2.3 PostgreSQL and PostGIS

Alternatively to all the commercial products, the open source community offers an extension for PostgreSQL, because it does not support spatial data types and functions out of the box. At the core, PostGIS and Oracle follows the same approach of a spatial extender to provide spatial functionalities for the RDBMS (Schön, Laefer, Morrish, & Bertolotto, 2009).

PostGIS stores each feature, a geographic location or representation, OGC simple features specification compliant, which supports the interoperability of the stored information. Seven different geographic data types are implemented: `POINT`, `LINESTRING`, `POLYGON`, `MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON` for a collection of different polygon objects and `GEOMETRYCOLLECTION` for a collection of elements, such as points, lines and polygons. In addition they provide a lot of functions to calculate and do conversions on geographic objects.
A short example is provided in listing 3.9

```
-- Creating the table
CREATE TABLE geotable(
    gid serial PRIMARY KEY,
    the_geom geography(POINT,4326), -- Defining the spatial referencing system. Here
    the_name TEXT
);

-- Defining the spatial referencing system. Here EPSG:4326

-- Inserting data
INSERT INTO geotable(the_geom, the_name)
VALUES (ST_GeomFromText('POINT(-126.4 45.32)@', 312), 'A,Place');

-- Selecting data
SELECT the_name, ST_AsGML(the_geom) AS the_geom FROM geotable;
```

Listing 3.9: PostGIS usage example

As you can see in the listing 3.9 on line 13, it is possible to convert the geometry object on the fly. PostGIS supports different input and output formats: binary, well-known text (WKT), GeoJSON, GML, KML, SVG, Text, etc.

**Binary Large Object:** An other thing we need to look at is the concept of binary large objects. In general PostgreSQL does not allow to store tuples that span multiple pages. Thereby it is not possible to store very large objects in a single cell directly, because per default a page has size of 8kB. PostgreSQL overcome this limitation by compressing large data and breaking it up into multiple rows what is called TOAST. In addition to that, PostgreSQL provides a catalog called `pg_largeobject`, which holds data making up "large objects" (PostgreSQL Global Development Group, 2011c). The idea behind this catalog type is the same as the TOAST mechanism. It splits up the data into blocks of size `LOBLKSIZE` and assigns each block a `OID`. In addition it stores the `pageno` and the data of type `bytea`. Both techniques, TOAST and `pg_largeobject`, are more or less the same. TOAST has a limitation of data size up to 1GB and `pg_largeobject` up to 2 GB. Also, large objects can be randomly modified using a read/write API that is more efficient than performing such operations using TOAST. A B-tree index guarantees fast searches for the correct chunk number when doing random access reads and writes on `pg_largeobject`.
PostgreSQL Extending SQL

PostgreSQL is catalog-driven and can be therefore extended. Database management systems (DBMS stores information about databases, tables, columns, etc. in so called system catalogs. This catalogs appears to the user as table, but the DBMS stores it internal bookkeeping in them. PostgreSQL stores much more information in such catalogs: not only information about tables, columns, but also information about data types, functions, access methods and so on (PostgreSQL Global Development Group, 2011b). These tables can be modified by the user to create on data types, procedures and include (own written) libraries. PostgreSQL dynamically reloads functionality as required e.g. new data types, functions, and so on.

In this chapter we do not discuss all the possibilities that PostgreSQL gives to add new functionality; we limit ourself to the structure and logic we need for this paper.

4.1 Terms

The most important part are the user-defined functions. They can be divided into query language, procedural language, internal, and C-language functions. Every kind of functions can take base types, composite types, or combinations of these as arguments, as well as returning base or composite type (PostgreSQL Global Development Group, 2011b).

A base type, e.g. \texttt{int4}, is implemented below the level of SQL language means typically implemented in low-level languages such as C and corresponds to abstract data types. PostgreSQL can only work with these types through functions provided by the user.

A composite type are created whenever a user creates a table. It is simply a list of types with associated field names and are also called row types. In addition it is possible to use \texttt{CREATE \_TYPE} to define a "stand-alone" composite type as exampled in listing 4.1.

\begin{verbatim}
-- Create composite type
CREATE TYPE person AS (
    first_name    varchar(80),
    last_name     varchar(80),
    address       text
);

-- Create table that uses the composite type "person"
CREATE TABLE customers(
    customer    person,
    card_nr     integer
);

-- Inserting a new customer
INSERT INTO customers VALUES (ROW('Zapp', 'Brannigan', 'Nimbus_Avenue'), 5724)
\end{verbatim}

Listing 4.1: PostgreSQL Composite Type
4.2 C-Language Functions

User-defined functions can be written in C or a language that can be made compatible with C like C++. Such functions are compiled into dynamically loadable objects also called shared libraries which are loaded by the server on demand.

PostgreSQL will only store and retrieve the data from disk and use the user-defined functions over a type to input, process, and output the data. It is possible to pass the data by value, fixed-length, or by reference, fixed- or variable-length.

To extend PostgreSQL with new functionality it is important to have the right semantic. Each from PostgreSQL accessible function need to be declared with:

```c
PG_FUNCTION_INFO_V1(funcname);
```

Listing 4.2: PostgreSQL declare C function

whereas "V1" defines the convention version to be used. This function can than be implemented by the following semantic:

```c
Datum funcname(PG_FUNCTION_ARGS) { }
```

Listing 4.3: PostgreSQL define C function

In the function body, each actual argument can be fetched by `PG_GETARG_XXX(number_of_argument)` macro that corresponds to the argument's data type and the result is returned by `PG_RETURN_XXX(value)` macro for the return type.

The following two listing 4.4 for basic types and 4.5 for composite types are examples from the PostgreSQL documentation1.

```c
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/* by value */
PG_FUNCTION_INFO_V1(add_one);
Datum add_one(PG_FUNCTION_ARGS) {
    int32 arg = PG_GETARG_INT32(0);
    PG_RETURN_INT32(arg + 1);
}

/* by reference, fixed length */
PG_FUNCTION_INFO_V1(add_one_float8);
Datum add_one_float8(PG_FUNCTION_ARGS) {
    float8 arg = PG_GETARG_FLOAT8(0);
    PG_RETURN_FLOAT8(arg + 1.0);
}

/* by reference, variable length */
PG_FUNCTION_INFO_V1(makepoint);
Datum makepoint(PG_FUNCTION_ARGS) {
    Point *pointx = PG_GETARG_POINT_P(0);
    Point *pointy = PG_GETARG_POINT_P(1);
    Point *new_point = (Point *) palloc(sizeof(Point));
    new_point->x = pointx->x;
    new_point->y = pointy->y;
    PG_RETURN_POINT_P(new_point);
}
```

1The example and further documentation can be found under the following url: an can be found here http://www.postgresql.org/docs/9.1/interactive/xfunc-c.html
4.3 Compiling and Linking

To compile the source under Linux the cc compiler can be used. It is important that the compiler flag 
-fpic and -shared is set to crate a shared library (see listing 4.6).
4.4 Register C-Functions

When everything is written, compiled, and placed on the right place, then the C-Functions can be registered in PostgreSQL as it is possible to call them by a SQL statement. This can be simply than with the `CREATE FUNCTION` statement as in listing 4.8 for basic types and in listing 4.9 for composite types.

```sql
CREATE FUNCTION add_one(integer) RETURNS integer AS 'DIRECTORY/funcs', 'add_one' LANGUAGE C STRICT;
```

```sql
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean AS 'DIRECTORY/funcs', 'c_overpaid' LANGUAGE C STRICT;
```

The `DIRECTORY` stands for the folder of the shared library. Note that the functions are specified as strict like in line 3, which means, that the system automatically assumes a null result if any input value is null. By doing it so, we avoid checking for null inputs in the function code if we are using pass-by-reference arguments which is mostly the case.

4.5 How to use it

When registered the functions in PostgreSQL as in listing 4.8 or 4.9, you can simple use your PostgreSQL function in your statement by passing the arguments defined in your function and you will get the value calculated by your C function of the type defined in the PostgreSQL function e.g. `RETURNS text`.

```sql
SELECT name, c_overpaid(emp, 1500) AS overpaid
FROM emp
WHERE name = 'Bill' OR name = 'Sam';
```

Listing 4.9: Using a C function with composite data type
Implementation Approaches

PostgreSQL does not support any spatial data type or functions at all. Together with the spatial extender PostGIS, has PostgreSQL the ability to store, read, and analyze spatial information. Because of this circumstances, it is absolutely necessary, that PostGIS need to be taken under consideration for at least one approach. In addition, the issued mentioned in section 3.2 need to be addressed unconditionally.

5.1 Approach 1: One Tuple

A LAS file consists of a public header block, a variable length records, and point data records in binary format. Each point data has to be in the same format. Such a file can grow very fast and becoming huge quickly over gigabytes. Storing such a file in a single cell of the table could be a possibility and leads to different methods.

**Binary Large Object:** The most obvious method is to store the file as it is, means creating a column of type `bytea` and storing it into this cell. The PostgreSQL table can be defined as in listing 5.1 whereas the datatype to store the LAS file is defined as `bytea` which is known on other databases as `blob`.

```sql
CREATE TABLE point_cloud(  
  file_id serial,  
  ...  
  raw_data bytea  
);
```

Listing 5.1: Creating a table using bytea datatype

In general PostgreSQL does not allow to store tuples that span multiple pages. A page is per default in PostgreSQL 8kB. Therefore it is not possible to store very large objects in a cell directly. This limitation can be overcome by compressing the large field value and / or breaking it up into multiple physical rows. This technique is known as TOAST (or "the best thing since sliced bread") and is supported only by certain data types, which must have a variable-length (varlena) representation, in which the first 32-bit word of any stored value contains the total length of the value in bytes. TOAST takes two bits of the varlena length word and limits thereby the logical size of any value to 1 GB ($2^{30} - 1$ bytes) (PostgreSQL Global Development Group, 2011a).

If any of the columns of the table are TOAST-able, what `bytea` is, a associated TOAST table will be created and linked over the OID stored in the table’s `pg_class.reltoastrelid` entry. After compressing the value with the LZ compression technique family (a lossless technique), it will be split into chunks of at most TOAST_MAX_CHUNK_SIZE bytes, which is by default about 2000 bytes so that four chunk rows fit to a page. Each chunk is stored as a separate and holds a chunk_id, a chunk_seq, and the chunk_data.

The TOAST scheme has a number of advantages compared to a more straightforward approach such as allowing row values to span pages. Most of the queries are usually qualified by comparison on relatively small key values. The big values of TOASTed attributes will only be pulled out at the time the result is sent to the client. Consequently, the main table is much smaller and more of its rows fit in the shared buffer cache. Most of the operations, such as sorting data, could then be done on the buffer.
The difficulty by this method is to query a partition of the point cloud. Hence, maybe the whole data need to be loaded into the RAM to find the intersection of the whole point cloud and the requested partition, because the spatial operation is doing a query on a TOASTed attribute. Means working with the file itself or with this method, no performance or additional functionality will be won.

**PostGIS multipoint:** The second most obvious method is to use the power of PostGIS and TOAST. PostGIS is providing a so called multipoint object which can be used in combination with the data type `geometry`.

```sql
CREATE TABLE point_cloud(
  file_id serial,
  ...
  data geometry,
  info text[]
);

INSERT INTO point_cloud(data, info) VALUES(
  ST_GeomFromText('MULTIPOINT((0 0 0, 1 1 1, ...))'),
  '{[68, 2962], [72, 2964]}'
);
```

Listing 5.2: Using multipoint object

As exampled in the listing 5.2 the whole LAS file, or the x,y,z coordinates to be more precise, is stored in a single cell as multipoint object. By the fact that each point in the point data records stores in addition to the x,y,z coordinates also information like intensity, classification, etc. (see chapter 2.2) and PostGIS can only store coordinates in the multipoint object, a second column need to be introduced to store these data. This additional column stores all the additional information of a point in the same order as in the `geometry` data type. The critical assumption is, that the order of the values in the `geometry` and `text` data type are always the same and never be changed. If it will be changed anyway, maybe by deleting, changing, or adding a point, the order of the the points and the additional information need to be changed in the same way and at the end to be assured that it manifest the same order.

Moreover, the additional information can be sourced out onto an additional table. It need to hold all the information per point. This can be done for example by storing each additional information for one point on a single tuple as an attribute. Despite that, we have still the problem that the relationship between the additional information and the point in the multipoint object is unknown.

In spite of doing it so, it does not change anything on the fact that the relationship between a point and its additional information need to be handled very carefully and the complexity of a spatial operation on the point cloud is very high, because all the data need to be loaded, before it can be processed.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy to load by implanting a little program and doing a simple <code>INSERT</code></td>
<td>Doing spatial operation costs a lot of memory</td>
</tr>
<tr>
<td>No data transformation needed</td>
<td>Maybe entire raw data need to be loaded into memory despite the TOAST technique</td>
</tr>
<tr>
<td>Raw data is stored in a lossless compressed format in separate table</td>
<td>Single cell can only store up to 1 gigabyte</td>
</tr>
<tr>
<td>PostGIS provides a comprehensive spatial library</td>
<td><code>DELETE</code>, <code>UPDATE</code>, <code>INSERT</code> of a single Point is complicated</td>
</tr>
</tbody>
</table>

Table 5.1: Advantages and disadvantages of approach 1

### 5.2 Approach 2: Each Point in a Tuple

Another obvious way is to store the files into a table, whereas each point is a single tuple including all additional information and the PostGIS geometry for the x,y,z coordinate. This is as ease as it sounds. First we need to create a table, that represents the structure of the LAS point data record. To do so, it results in a table schema as exampled in listing 5.3.
Now, that we have the LAS point data record structure representation, we can load the file into the database. One possibility is to transform the LAS file into a CSV file and load it into the database (see listing 5.4 and 5.5).

```
#!/bin/bash
x=0
for f in $( ls *.las); do
  x=`expr $x + 1`
  echo $x $f
  las2txt --parse xyzinrcpM -sep komma $f $f.csv
done
```

Listing 5.4: Convert LAS file
Source: Mather (2010)

The new comma separated values file can now be uploaded as follow:

```
psql -U postgres -d point_cloud -f las.txt
```

Listing 5.5: Insert CSV file
Source: Mather (2010)

As we have now all the data in the database, namely, for each point in the LAS file we created an own tuple, we can generate the 2- or 3-dimensional points using PostGIS. Depending on what you want to analyze afterwards, you can choose the 2D or 3D data object (see listing 5.6). In PostGIS 2D points can be created by using the function `ST_MakePoint(x,y)` and 3D points by using `ST_MakePointM(x,y,z)` function.

```
SELECT AddGeometryColumn('public', 'point_cloud', 'the_geom', 91023236, 'POINT', 2);
UPDATE point_cloud SET
  the_geom = ST_SetSRID(
    ST_MakePointM(x,y,z), 91023236
  );
```

Listing 5.6: Creating geometry data type
Source: Mather (2010)

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• No additional upload program needed</td>
<td></td>
</tr>
<tr>
<td>• Each point can be queried separately</td>
<td></td>
</tr>
<tr>
<td>• Usage of PostGIS functions</td>
<td></td>
</tr>
<tr>
<td>• Extreme amount of tuples needed</td>
<td></td>
</tr>
<tr>
<td>• A lot of memory needed for querying a 3-dimensional bounding box</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Advantages and disadvantages of approach 2
5.3 Approach 3: Splitting Data

An other approach could be to split up the data into a set of blocks, which will be stored in a separate table. The blocks will be generated when inserting the LAS file into the database. Therefore a conversion of the raw data to fit into these blocks is needed. This should be a little program, that splits up all the points based on fixed or on users desired parameters. One of the parameter should be the block size, means how many points would be in a single block. Another parameter could be the algorithm, how does the blocks would be generated. One concept could be based on the nearest neighborhood or or another one based on the similarity of a specific point attribute. This guarantees, that the data would be stored in blocks of similar interests of the user. The advantage in such a logic is, that the most of the spatial operations on the point cloud, affects a minimum of blocks, and therefore the complexity and needed performance could be minimized.

Once the data are split into blocks they need to be uploaded. For that a table schema need to be defined (see listing 5.7). It is a manual approach, because each LAS file could have a different amount of additional information to a specific point.

```sql
CREATE TABLE point_cloud_info (
    file_id serial,
    ...
);

CREATE TABLE point_cloud_data (
    obj_id integer,
    blk_id integer,
    raw_data bytea,
    bbox geometry
);
```

Listing 5.7: Create block table

As you see in listing 5.7 each block is saved as `bytea` and the additional information to a block can be stored in separate attributes as needed. Instead of using the data type `bytea`, `pg_largeobject` can be used to store the raw data.

**Excursus:** This catalog type `pg_largeobject` automatically compresses the data and splits it up into blocks of the size defined in `LABELSIZE` variable. A B-tree index guarantees fast searches for the correct chunk number when doing random access reads and writes. If `pg_largeobject` is used, the client or server interface need to provide manipulation functions for accessing large objects and must take place within an SQL transaction. The idea behind this interface, is to stream the data to to corresponding clients. All these functions begin with `pg_`, ends like in a Unix file-system with e.g. `read`, `write`, etc. and returns an `oid` that was assigned to the new large object or an `int` that specifies the amount of bytes. For example the read function is defined as

```c
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

and reads `len` bytes into the buffer `*buf`.

Now, to query a partition of the point cloud or doing spatial operation, the involved blocks need to be found. For that, each block holds its minimum bounding box, which is a box with the smallest measure within which all the points lie and can be created by calling the PostGIS function `Box3D(geometry)`. But first, the x,y,z coordinates of the `raw_data` need to be extracted and hand it over to the `Box3D(geometry)` function and stored as the separate attribute `bbox`. Generating the bounding box, and optionally the additional information are preprocessing steps after generating the blocks.

So lets say we want to find a 3-dimensional area in the point cloud. The area need to be defined as bounding box as the second argument in listing 5.8 on line 2. Then we need to find, all blocks, that intersects with this area. The results is a list of blocks that contains at least one point that is included in the area.

```sql
SELECT raw_data FROM point_cloud_data WHERE
    ST_Intersection(bbox::geometry, Box3D(0 0 0, 10 23 30)::geometry);
```

Listing 5.8: Find an area in the point cloud
As you can imagine, the right blocks has been selected, but too many points will be returned. Here we need to extend PostgreSQL (see chapter 4) by new functions, that cuts out the unnecessary points. One possibility is to create an own C-library that will do it, given the user interested area. But why need we invent the wheel again. Why not trying to use PostGIS which provides a comprehensive spatial library. All what we need to do, is to call the intersection function again, but this time with a collection of all points of the returned blocks. Let us introduce a new PostgreSQL function called `PC_GetIntersectionFromBlocks(blocks, area)` which takes a `bytea` array called blocks and the desired area as `geometry`. With these information a C-library will cut all x,y,z coordinates out of the blocks and assemble it to a `geometry` of object type `multipointm`. This object can than be handed over to the PostGIS C-library which will calculate the intersection of the multipoint object and the given bounding box. The result from the PostGIS C-library will than be got back to PostgreSQL over our own C-library and finally returned to the user.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Using the proven comprehensive PostGIS library</td>
<td>• Tables need to be created forehanded by the user</td>
</tr>
<tr>
<td>• Only necessary blocks will be queried</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Advantages and disadvantages of approach 3
As we saw in the previous chapters, different approaches or let say database schemas could led in a solution to our goal. The difficulty is to define a right solution to fulfill the additional conditions to minimize the side effects. The only feasible approach discussed in chapter 5 is to split a LAS file into a set of different blocks. All the other approaches have at least one severe drawback. In approach 1 it could be the problem, that not all points of a single LAS file can be stored in geometry data type, because of the limit of the amount of ordinates, which can be stored. Approach 2 can lack in performance, by reason that each point is a unique tuple in the database table. To get for example a three-dimensional bounding box of the whole point cloud, PostgreSQL need to scan nearly the whole table. Therefore, further research need to be concentrated on approach 3, which splits all the data into a set of blocks that are stored in a separate point cloud data and referenced to its meta data table.

One of the most challenging conditions is the limitation of the amount of ordinates, which can be stored in a single geometry data type. This side effect has impact of the design of the user defined C library. First, PostgreSQL need to find the affected blocks based on the query. These blocks can be easily determined by doing a intersect on the bbox attribute of the block table. The bbox attribute holds the maximum extent of the stored raw data block, means that the bbox spans all the points of a block. Next, the blocks including the requested users bbox need to be transferred to the C library, which merges all to blocks together. Based on new merged block, the requested points, which are spanned by the users defined bbox can be determined. An simple intersect for each point need to determined. If it intersects than the point will be returned back to PostgreSQL. Lastly, all found points can be returned to the user.

The approach 3 has the benefit, that not all points need to be scanned in PostgreSQL and that we not run against a data type limit. Hence, the most of PostgreSQL's standards has been used without violating any design issue of PostgreSQL. One thing needs to be mentioned at this point. All the discussed approaches are on theoretical basis and at some points PostgreSQL specific optimization techniques can be applied. Additionally, the behavior of PostgreSQL’s internal optimization and organization algorithms need to be analyzed can lead in changes of the database schema.


A.1 C function

```c
/*
** Filename: reverse_string.c
*/
#include "postgres.h"
#include "fmgr.h" /*
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
PG_FUNCTION_INFO_V1(reverse_string);

Datum reverse_string(PG_FUNCTION_ARGS) {
    int len, pos = 0;
    VarChar *str_out, *str_in;
    /* get pointer to the argument */
    str_in = PG_GETARG_VARCHAR_P_COPY(0);
    /* calculating the size in bytes of the string */
    len = (int) (VARSIZE(str_in) - VARHDRSZ);
    /* create an empty string of same size */
    str_out = (VarChar *)palloc(VARSIZE(str_in));
    /* the resulting structure has the same length */
    SET_VARSIZE(str_out, VARSIZE(str_in));
    /* Verifies that encoding of the string argument matches encoding of DB */
    pg_verifymbstr(VARDATA(str_in), len, false);
    /* copy to the other side of the chain */
    while (pos < len) {
        int charlen = pg_mblen(VARDATA(str_in) + pos);
        int i = charlen;
        /* copy a character.
         * a character = Byte */
        while (i--) {
            *(VARDATA(str_out) + len - charlen + i - pos) = *(VARDATA(str_in) + i + pos);
            pos = pos + charlen; /* increments the counter */
        }
    }
    PG_FREE_IF_COPY(str_in, 0);
    /* returns the copy */
    PG_RETURN_VARCHAR_P(str_out);
}
```

A.2 Control file

```plaintext
# reverse_string extension
comment = 'reversing a string'
default_version = '1.0'
module_pathname = '$libdir/reverse_string'
relocatable = true
```
A.3 SQL file

SQL file need to have a file like pattern `<extension_name>--<default_version>.sql`. In this case it is `reverse_string--1.0.sql`

```
CREATE FUNCTION reverse_string(varchar)
RETURNS varchar
AS 'MODULE_PATHNAME', 'reverse_string'
LANGUAGE C STRICT IMMUTABLE;
```

A.4 Compile script

```
#!/bin/bash

#clean up
rm -f `pg_config --sharedir`/extension/reverse_string--1.0.sql
rm -f `pg_config --sharedir`/extension/reverse_string.control
rm -f `pg_config --pkglibdir`/reverse_string.so
rm -f reverse_string.so
rm -f reverse_string.o

# compile
cc -fpic -c -I`pg_config --includedir-server` reverse_string.c
cc -shared -o reverse_string.so reverse_string.o

# create symbolic link to PostgreSQL install dir
# default dir: /usr/lib/postgresql/9.1/lib/
cp `pwd`/reverse_string.so `pg_config --pkglibdir`

#default dir: /usr/share/postgresql/9.1
cp `pwd`/reverse_string--1.0.sql `pg_config --sharedir`/extension
cp `pwd`/reverse_string.control `pg_config --sharedir`/extension
```
B.1 List of Figures

2.1 Example of a digital elevation model .................................. 4
3.1 Scalable storage framework for a point cloud in Oracle ............ 8

B.2 List of Tables

2.1 LAS Format Definition ..................................................... 4
2.2 LAS Format Definition Containing Waveform Data .................. 5
3.1 Structure of Oracle’s SDO_PC type ..................................... 9
3.2 Structure of Oracle’s block table ...................................... 10
5.1 Advantages and disadvantages of approach 1 ......................... 18
5.2 Advantages and disadvantages of approach 2 ......................... 19
5.3 Advantages and disadvantages of approach 3 ......................... 21

B.3 Listings

3.1 Example of using libLAS in C++ ....................................... 7
3.2 Adding an SDO_PC column ............................................. 8
3.3 Creating an SDO_PC block table by SELECT ....................... 8
3.4 Creating an SDO_PC block table by data type ..................... 8
3.5 Oracle’s initialization statement of point cloud data ............... 10
3.6 Defining external file as input table .................................. 10
3.7 Create point cloud blocks ............................................. 10
3.8 Microsoft SQL spatial data type .................................... 11
3.9 PostGIS usage example .............................................. 12
4.1 PostgreSQL Composite Type .......................................... 13
4.2 PostgreSQL declare C function ...................................... 14
4.3 PostgreSQL define C function ....................................... 14
4.4 PostgreSQL C-language function example for basic types ......... 14
4.5 PostgreSQL C-language function example for composite types ... 15
4.6 Compiling and linking C functions .................................. 16
4.7 Registering C functions in PostgreSQL ............................ 16
4.8 Registering C functions in PostgreSQL ............................ 16
4.9 Using a C function with composite data type ..................... 16

26
B.4 Nomenclature

API Application Programming Interface
ASPRS The Imaging and Geospatial Information Society
BLOB Binary Large Object
CHTRS95 Swiss Terrestrial Reference System 1995
CLR Common Language Runtime
CSV Comma Separated Values
DBMS DataBase Management System
DEM Digital Elevation Model
LAS Binary Interchange format defined by ASPRS
LiDAR Light Detection and Ranging
OGC Open Geospatial Consortium
PC Point Cloud
Postgres PostgreSQL
TIN Triangulated irregular network
WKT Well Known Text