Computers & Graphics 49 (2015) 92–125

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on Processing Large Geospatial Data

Massive point cloud data management: Design, implementation and execution of a point cloud benchmark

Peter van Oosterom [a,*], Oscar Martinez-Rubi [b], Milena Ivanova [b], Mike Horhammer [c], Daniel Geringer [c], Siva Ravada [c], Theo Tijssen [a], Martin Kodde [d], Romulo Gonçalves [b]

[a] Section GIS Technology, Department OTB, Faculty of Architecture and The Built Environment, TU Delft, The Netherlands
[b] Netherlands eScience Center, Amsterdam, The Netherlands
[c] Oracle Spatial & Graph and MapViewer, Server Technologies, Nashua, NH, USA
[d] Fugro, GeoServices B.V., Leidschendam, The Netherlands

ARTICLE INFO

Article history:
Received 9 September 2014
Received in revised form
7 January 2015
Accepted 29 January 2015
Available online 7 February 2015

Keywords:
Benchmark
DBMS
Point cloud data
Parallel processing
Space filling curve
Vario-scale

ABSTRACT

Point cloud data are important sources for 3D geo-information. An inventory of the point cloud data management user requirements has been compiled using structured interviews with users from different background: government, industry and academia. Based on these requirements a benchmark has been developed to compare various point cloud data management solutions with regard to functionality and performance. The main test dataset is the second national height map of the Netherlands, AHN2, with 6–10 samples for every square meter of the country, resulting in 640 billion points. At the database level, a data storage model based on grouping the points in blocks is available in Oracle and PostgreSQL. This model is compared with the 'flat table' model, where each point is stored in a table row, in Oracle, PostgreSQL and the column-store MonetDB. In addition, the commonly used file-based solution Rapidlasso LAStools is used for comparison with the database solutions. The results of executing the benchmark on different platforms are presented as obtained during the increasingly challenging stages with more functionality and more data: mini (20 million points), medium (20 billion points), and full benchmark (the complete AHN2).

During the design, the implementation and the execution of the benchmarks, a number of point cloud data management improvements were proposed and partly tested: Morton/Hilbert code for ordering data (especially in flat model), two algorithms for parallel query execution, and a unique vario-scale LoD data organization avoiding the density jumps of the well-known discrete LoD data organizations.

1. Introduction

The work presented in this paper is part of the project Massive point clouds for eSciences. The main goal of this project is to develop an infrastructure for the storage, the management, the analysis, the processing, the dissemination, the visualization and the manipulation of massive point clouds. The eScience project investigates solutions in order to better exploit the rich potential of point cloud data. The project partners are Rijkswaterstaat (RWS), Fugro, Oracle, Netherlands eScience Center and TU Delft.

Modern Big Data acquisition and processing technologies, such as laser scanning from airborne, mobile, or static platforms, dense image matching from photos, multi-beam echo-sounding, or from autotracking seismic data, have the potential to generate point clouds with millions, billions, or even trillions of 3D points with in many cases one or more attributes attached. This is especially true with the available and expected repeated scans of the same area where the temporal dimension becomes significantly important. These point clouds are too massive to be handled efficiently by common geo-ICT infrastructures. Initial implementations are available in both commercial and open source database products, illustrating the user needs for point cloud support; Oracle Spatial and Graph SDO_PC and SDO_PC_BLK object/data types [1] and PostgreSQL-PostGIS PCPATCH object/data type [2]. Both systems implement a storage model based

* Corresponding author.
E-mail addresses: P.J.M.vanOosterom@tudelft.nl (P. van Oosterom), O.Rubi@esciencecenter.nl (O. Martinez-Rubi), M.Ivanova@esciencecenter.nl (M. Ivanova), Mike.Horhammer@oracle.com (M. Horhammer), Daniel.Geringer@oracle.com (D. Geringer), Siva.Ravada@oracle.com (S. Ravada), T.P.M.Tijssen@tudelft.nl (T. Tijssen), M.Kodde@fugro.nl (M. Kodde), R.Goncalves@esciencecenter.nl (R. Gonçalves).

http://dx.doi.org/10.1016/j.cag.2015.01.007
0097-8493/

on grouping of spatially co-located points in blocks. These new point cloud object types are available in addition to the existing spatial object types like the raster type [3,4].

A benchmark has been designed based on user requirements [5], as this did not yet exist for point cloud data. We compare the loading and querying of different datasets in several Point Cloud Data Management Systems (PCDMS) implementing different storage models. The blocks model implementations in Oracle and PostgreSQL are compared with flat relational table model implementations, i.e. one row per point, in both systems and also in the column-store MonetDB [6]. The DataBase Management System (DBMS) solutions are compared with a file-based approach using the scriptable tools provided by Rapidlasso LAStools [7].

However, the aim of the project is to identify the features of a generic system that can offer a good performance and rich functionality for point cloud data, together with efficient management of other spatial (vector, raster) and non-spatial data in an integrated and scalable environment.

1.1. DBMS approach

One of the most discussed topics regarding point clouds is whether they are suitable to be handled in DBMS environments. A common practice is to use a collection of files and specific tools to select, analyze, manipulate and visualize the point cloud data. The sampling nature of point cloud data is the same as raster data and the DBMS suitability was also raised in that context. Thus, similar arguments can be given. Both raster data and point cloud data are often quite massive, but relatively static. As there is limited requirement for update, delete, or insert of individual points in a multi-user context, there is not so much need for DBMS transaction support. It is indeed true that it is possible to develop specific file based solutions based on LAS, Rapidlasso's LAZ [8] or Esri's ZLAS [9] file formats for point cloud data management. However, with the new acquisition technologies the datasets are growing significantly and an increasing number of data management challenges can be expected. To cope with them the file-based solutions would have to re-develop large parts of a DBMS. For example, our test data, AHN2 [10], is stored and distributed in more than 60,000 LAZ files, which is not really efficient for simple point cloud data selection purposes even with the quite reasonable tools of Rapidlasso LAStools. At this point a DBMS could provide an easier to use and more scalable alternative. Further, a specific file-based solution may be good at one aspect, handling point clouds, but in reality users have a range of different datasets and types: administrative data, vector data, raster data, temporal data, etc. Therefore, a standardized and generic DBMS solution would be preferable when users want to combine, for example, vector and point cloud data in their queries. In this environment it should then possible to pose integrated queries such as select points from a point cloud dataset, which overlap with a set of 3-meter buffered polygons around buildings owned by a certain person. In addition, file-based solutions offer a limited range of functionalities while DBMS have, in principle, unlimited functionality thanks to the ad hoc query support and the declarative programming models. Moreover, DBMS use optimized disk IO strategies, i.e. scanning, pre-fetching, improving cache management. Finally, most modern DB systems offer automatic parallelization in the query executions.

Point cloud data resemble both vector (object) and field (raster) data. Naturally, the question arises whether some of those existing data types can be used for implementation of point clouds. One could argue that a point cloud could be stored as standard vector data; e.g. in a Simple Feature Specification (SFS) multipoint in a single row or in many rows, each with a single point [11]. Even though this is theoretically correct, in practice the point clouds are too large to be stored efficiently in this manner.

A work-around could offer a pragmatic solution – store point clouds in several spatially coherent multipoint objects, but this is non-trivial to implement and would put a serious work load on the user to realize this grouped multipoint storage. One could also argue that point cloud data is similar to raster data as it is based on sampling; e.g. as encoded in GeoTIFF coverages [12]. This is also correct, but the result in the case of point cloud data is not a regular grid, so the available raster data storage solutions are also unsuitable. Therefore, we propose to add a *third type of spatial representation* to the geographic information processing systems and standards: a point cloud data type. Based on user requirements the point cloud data type and its operators should cover or aim to cover during the design and perhaps during standardization phase of this data type:

1. *xyz*: specify the basic storage of the coordinates in a spatial reference system (SRS) using various base data types: int, float, double, number, varchar.
2. Attributes per point: 0 or more. Example attributes are intensity *I*, color *RGB*, classification, observation point position in addition to resulting target point, etc. This might conceptually correspond to a higher dimensional point.
3. Data organization based on spatial coherence: blocking scheme in 2D, 3D, etc.
4. Efficient storage with compression techniques exploiting the spatial cohesion.
5. Data pyramid support: level of detail (LoD), multi-scale or vario-scale and support for perspective selections.
6. Temporal aspect: options for time per point (costly) or per block (less refined).
7. Query accuracy: for 2D, 3D or *n*D query ranges or geometries specify to report points or storage blocks and refine subsets of blocks with/without tolerance value.
8. Operations/functionalities in the following initially proposed categories: (a) loading, (b) selections, (c) simple analysis (not assuming 2D surfaces in 3D space), (d) conversions (some assuming 2D surfaces in 3D space), (e) towards reconstruction, (f) complex analysis (some assuming 2D surfaces in 3D space), (g) LoD use/access, and (h) updates. See Appendix A for more information regarding the proposed functionalities.
9. Indicate the use of parallel processing for the operations listed in the Point 8.

Given the huge volumes of point cloud data, the performance is critical, both with respect to data storage and to speed for loading and querying. Therefore, it is a real challenge to make an efficient implementation. However, several techniques and theories are available and there are not any fundamental reasons why this cannot be done in a DBMS context. The initial implementations such as the Oracle spatial's SDO_PC and SDO_PC_BLK data type and the PostgreSQL-PostGIS PCPATCH data type are steps in the right direction. It is important to assess the current implementations as fairly as possible via tests or benchmarks and cross-comparisons.

1.2. Related work

The majority of point cloud data management is currently done at the file level. Many different formats exist, some of them are vendor specific, while others are vendor neutral. The ISPRS LAS file format [13] is one of the most used file formats. The compressed counterpart, the Rapidlasso's LAZ format [8], is often used in the case of large datasets. The Esri's ZLAS format is also compressed and it is well integrated with the Esri GIS software. In order to improve the performance various spatial data structures are used in these file formats and their related software; e.g. quadtree/octree [14–16], hashing-based virtual grid [17], kd-tree, rtree. There is not any standard for point cloud data in the DBMS, similar to standards for vector [11] and raster data [12]. Otepka et al. [18] give a comprehensive overview of the point cloud data management issues, but do not really cover the DBMS aspect.

As explained in Section 1.1 both open source and commercial DBMS developers are working on point cloud data support at the DBMS level; e.g. PostgreSQL [2] and Oracle [1]. In addition, hybrid file/database solutions are applied; e.g. Boehm is using MongoDB with GridFS [19]. The key challenges are handling the large volumes of data, while providing standardized functionalities. Therefore, compression techniques to decrease the storage requirements and blocking techniques to exploit spatial coherence and avoid too high row level overhead are heavily investigated by industry and academia.

Ott [20] describes three basic approaches for storing point cloud data in an object-relational database, PostgreSQL: (a) store the content of a LAS file in single tuple as a BLOB (binary large object) or multipoint; (b) store each point with its attributes in a tuple by storing separate *x, y, z* attributes, or an object of point geometry type; and (c) splitting data into coherent blocks (which can be represented by a BLOB, multipoint or other data type). Each approach has its own advantages and disadvantages. Hoefsloot [21] made similar investigations, but in the context of Oracle Spatial and Graph. She proposed to use the POINTCLUSTER data type (GTYPE 3005 of SDO_GEOMETRY) to store the blocks of point cloud data. In parallel with these investigations both PostgreSQL-PostGIS and Oracle Spatial and Graph have developed their own flexible object types for point cloud data; respectively PC_PATCH and SDO_PC_BLK.

In this paper we also explore the application of space filling curves for point cloud data which is already being partly done in some of the tested systems. Lawder [22,23] already showed the benefits of using space filling curves in *N* dimensional data. Terry [24] describes how to integrate in RDBMS an approach based on space filling curves for more efficient accessing of *N*-dimensional data.

1.3. Paper outline

The outline of this paper is as follows: Section 2 summarizes the user requirements and describes the design of the benchmark including the phased approach (mini, medium, full) with regard to both dataset size and query functionalities. The used datasets are also presented. Section 3 details the used hardware and software, and describes the tested PCDMSs. Sections 4 and 5 describe respectively the medium and full benchmark executions where all the loading and querying details can be found.

Section 6 describes the principles of the improved data organization which is useful for both flat model and blocks model. Moreover, it describes the first tests with the proposed Morton-based data organization which is illustrated with the PostgreSQL flat table model, but this should also be as useful in the other flat table model platforms (Oracle, MonetDB). The details of the parallel query algorithms we propose for speeding-up the queries are given in Section 7. Section 8 describes the LoD principles, both the well known discrete LoD's (as in the data pyramids), but also our new continuous or vario LoD approach without fixed LoD levels.

The conclusions extracted from this benchmark are listed together with future work in Section 9. Finally, the appendices contain the details, such as the loading and querying scripts, which are very important for a benchmark.

2. User requirements and conceptual benchmark

Based on an inventory of user expectations with regard to the available point cloud data processing capabilities (Section 2.1), a platform independent conceptual benchmark has been designed (Section 2.2). Given the challenging aspects of the benchmark, mainly the amount of data and the tested functionalities, we organized the development and the execution of the actual benchmark in several stages (Section 2.3).

2.1. User requirements

The basic functionalities needed for a PCDMS are identified in the User Requirements document [5]. This is achieved by (a) literature research, (b) discussions with the project [25] members, and (c) customer surveys. A wide range of point cloud data users participated, varying from scientists at the Delft University of Technology (TU Delft) and Utrecht University to government users at Rijkswaterstaat (Ministry of Infrastructure and the Environment) and partners and customers from Fugro's network, one of the world leading geospatial companies with various research and production applications related to point clouds.

The resulted User Requirements document provided a good overview of the relevant functionalities. In addition, in a group session on the 31st May 2013 with staff members from Oracle, TU Delft, and the Netherlands eScience Center, the importance of the various requested point cloud functionalities was assessed. See Appendix A for a detailed list of the functionalities.

2.2. Design of conceptual benchmark

A ‘conceptual’ benchmark was designed based on the user requirements at a relative high abstraction level to assess various solutions for point cloud data management. It covers the range of functionalities as well as the performance in terms of storage space needs, loading times and query response times. The ‘conceptual’ benchmark had to be translated into a platform specific benchmark which can be executed. The benchmark includes the specifications of the test data and the queries, including the type of query geometries when relevant.

Table 1
Datasets description.

Dataset name	Benchmark	Points	Files	Format	Disk size (GB)	Area (km^2)	Description
20M	Mini/Medium	20,165,862	1	LAS	0.4	1.25	TU Delft campus
210M	Medium	210,631,597	17	LAS	4.0	11.25	Major part of Delft city
2201M	Medium	2,201,135,689	153	LAS	42.0	125	City of Delft and surroundings
23090M	Medium	23,090,482,455	1492	LAS	440.4	2000	Major part of Zuid-Holland province
639478M	Full	639,478,217,460	60,185	LAZ	11,644.4	40,000	The Netherlands

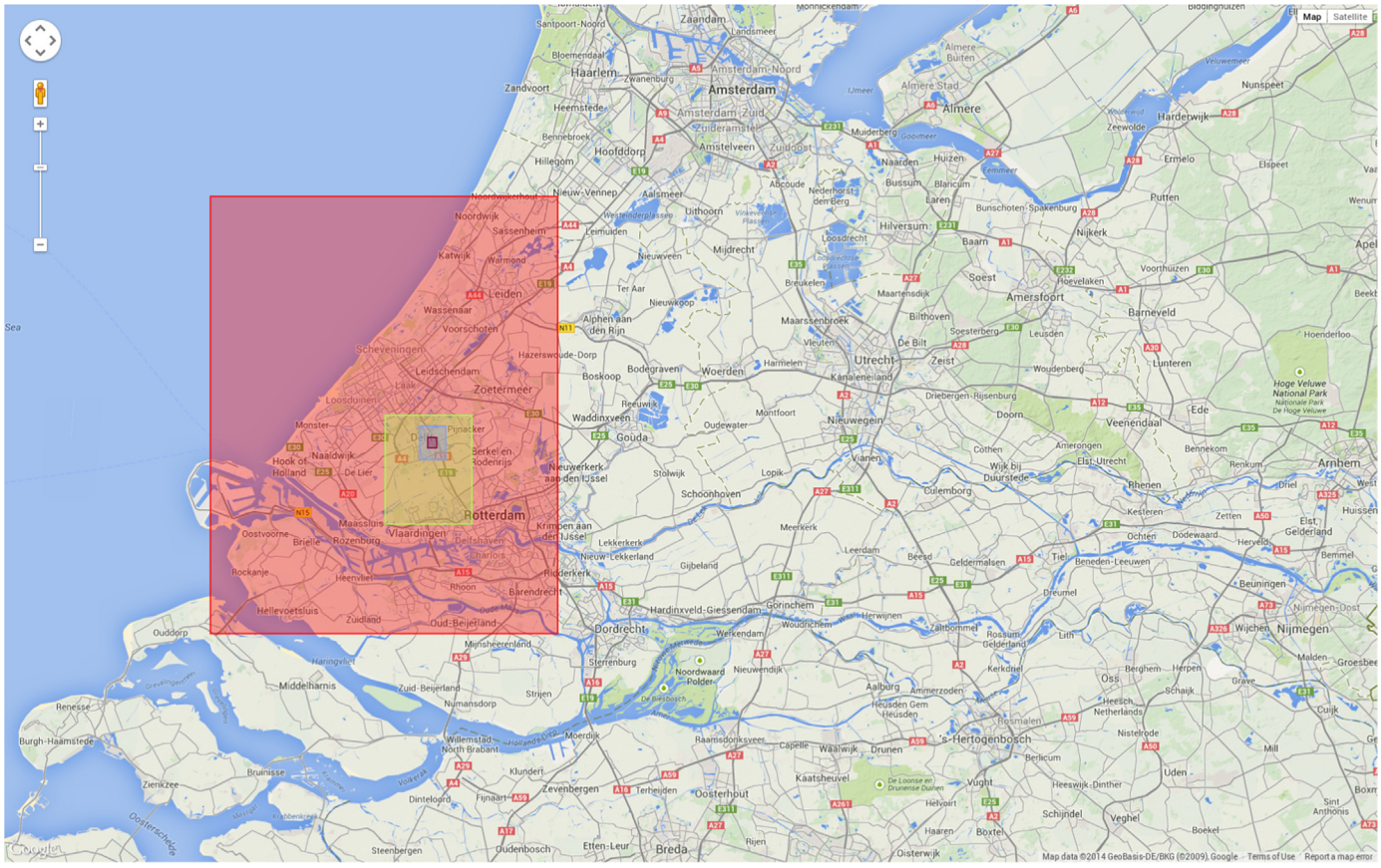

Fig. 1. Approximated projection of the extents of the used datasets in Google Maps: Purple area is for 20M dataset, cyan area is for 210M dataset, green area is for 2201M dataset and red area is for 23090M dataset. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

The structure or specification of the conceptual benchmark covers the following aspects:

- *Datasets*: range of different data sizes, points with/without additional attributes.
- *Storage platform*: Oracle, MonetDB, PostgreSQL; *storage model*: flat table, blocks; *storage parameters*: compression, clustering, block size, etc.
- *Query return method*: count, create table as select, local front-end, web services.
- *Query accuracy level*: storage blocks, individual points, 2D/3D query rectangle, 2D/3D geometry, etc.
- *Number of parallel users*: 1–10,000 with mainly selections.
- *Query types*: functionalities based on the user requirements. See Appendix A.
- *Manipulations*: update/add/delete data, including the computation of attributes; e.g. classify point. Note that this will usually involve a larger number of points and individual point manipulations are considered rare.

2.3. Benchmark stages

In Table 1 we detail the used datasets, all of them are subsets of the Actueel Hoogtebestand Nederland 2 (AHN2) dataset. The sample density is 6–10 points per square meter of the country, resulting in 640 billion points organized in 60,185 files. The reference system used is the Amersfoort/RD New, which EPSG or SRID (Spatial Reference System Identifier) is 28,992. Since the AHN2 is open data, interested readers can request copies, thus all the presented tests can be fully reproduced. The benchmark datasets have different extents and sizes and each dataset extends the area covered by the previous smaller one. Note that while the full AHN2 dataset (639478M) is provided in LAZ format, for the smaller subsets we used the LAS format. In Fig. 1 we depict the approximated projection of the extents of the used datasets.

Many of the aspects of the conceptual benchmark as introduced in Section 2.2 are orthogonal to each other, which results in a very large number of tests to be performed. Hence, it was crucial to define the best possible benchmark strategy and to prepare experimental infrastructure for automated benchmark execution and result processing.

In order to gain experience we started with the execution of a simplified functional test with a small dataset. We call this initial test the *mini-benchmark*. It uses the *20M* dataset with about 20 million points. The queries of the *mini-benchmark* are the first seven queries listed in Appendix B and they are selections of points within different query regions.

The *mini-benchmark* is also used to explore the effects of using different block sizes, compression levels, etc. The results of these initial tests served as a basis to decide on test options and systems configurations to be used in the next scaled-up benchmark, called *medium-benchmark*. In order to study the scaling behavior, this benchmark uses several datasets up to the *23090M* with about 20×10^9 (20 billion) points. It also extends the functionality and covers the first twenty queries listed in Appendix B. To avoid unrealistic caching scenarios the query regions cover different parts of the datasets spatial domain.

The *medium-benchmark* is a half-way sanity check before loading and querying the complete AHN2 point cloud data [10] in the *full-benchmark* where even more queries or functionalities are considered (see the complete list in Appendix B). At the moment of writing two PCDMS were tested with the *full-benchmark*: Oracle flat table in a Exadata hardware and the file-based LAStools approach in a 'normal' server.

In the future, scalability will be further tested by replicating the AHN2 data in the *upscaled-benchmark*. We plan to perform tests with about 20×10^{12} (20 trillion), about 30 times the current AHN2 dataset.

The strong and weak points of the various PCDMS are identified by assessing various systems with different databases and configurations, and with different hardware platforms. While benchmarking, the databases and their parameter settings are analyzed and, when needed, improved; for example, the Oracle developers have implemented a faster blocking approach. The execution of the benchmark has been repeated for these new, improved versions of the involved PCDMSs.

These tests will ease the path towards finding an 'optimal' solution, and it has already provided inspiration for the proposal of various improvements; see Section 6.

2.4. Query functionalities

The queries that we have executed are detailed in Appendix B. Many of them are selection queries, i.e. select all the points within a query region. Several types of query regions are tested, including rectangles, circles, simple polygons, diagonal lines, etc. Moreover, we also consider nearest neighbor queries and statistical queries such as finding the point with the maximum elevation value. For the PCDMSs using databases the query return method in the selection queries is a CTAS (Create Table As Select) so for each query the returned points are dumped in a new table. In the case of the LAStools PCDMS we dump the points in the query regions in new LAS files. Note that in the blocks storage model, which is implemented in PostgreSQL and Oracle blocks, the blocks need to be unpacked both for checking individual points and for delivering the final result.

The usual approach for the selection queries is to do a pre-selection of points contained in the bounding box of the query region and then check if each of them lies in the query region. In both PostgreSQL and Oracle implementations of the blocks model and in LAStools the pre-selection procedure is automatically done and transparent to the user. In the case of the PCDMS with flat tables it has to be manually done. Obviously in the case of rectangular regions the second filtering is not necessary. In the case of circular regions we use the distance method for the PCDMS using flat tables and LAStools, for the PCDMS using blocks we use an approximated polygon.

Regarding the nearest neighbor computations, for PCDMS using flat tables we use a pre-selection of a sufficient number of points (based on being within a manually estimated maximum distance to the query point) in the first phase, and in the second phase sort these points on distance and report the nearest N points.

3. Used hardware and software

3.1. Hardware

For the tests described in this document we have used a server with the following details: HP DL380p Gen8 server with 2×8-core Intel Xeon processors, E5-2690 at 2.9 GHz, 128 GB of main memory, and a RHEL 6 operating system. The disk storage which is directly attached consists of a 400 GB SSD, 5 TB SAS 15 K rpm in RAID 5 configuration (internal), and 2×41 TB SATA 7200 rpm in RAID 5 configuration (in Yotta disk cabinet).

In order to guarantee fast data loading we have split the SATA disks into two sets, one is used to store the raw data, i.e. the LAS files, and the other one contains the bulk of all the databases data and the file-based structure in the case of the LAStools PCDMS. In addition, in the case of Oracle and PostgreSQL we use the SSD disks to store the system data, (i.e. catalogs and logs); the faster SAS disks to store temporal data, the bulk of data is stored in the first SATA RAID 5 set, with the indexes stored in the second set (in fact, they are stored in the same set as the raw data). In this way we minimize heavy and mixed read/write IO to the same disk. This data distribution is achieved using different tablespaces for different types of data (Figs. 2 and 3).

In addition to the 'normal' HP DL380p Gen8 server, we also tested the Oracle Exadata X4-2 hardware [26]. The Exadata machine is Oracle SUN hardware specifically for the Oracle database software. The machine, illustrated in Fig. 4, consists of a *Database Grid* with multiple Intel cores for normal database computations and is available in configurations of a eighth, quarter, half or full rack with respectively 24, 48, 96 or 192 Intel Xeon cores. It also has *Storage Servers* with blocks of 12 Intel Xeon cores for compression, massive parallel smart scans/predicate filtering, lesser data transfer and better performance. An important feature of Exadata is the hardware hybrid columnar compression (HCC) offering five modes for compression: none, query low, query high, archive low and archive high.

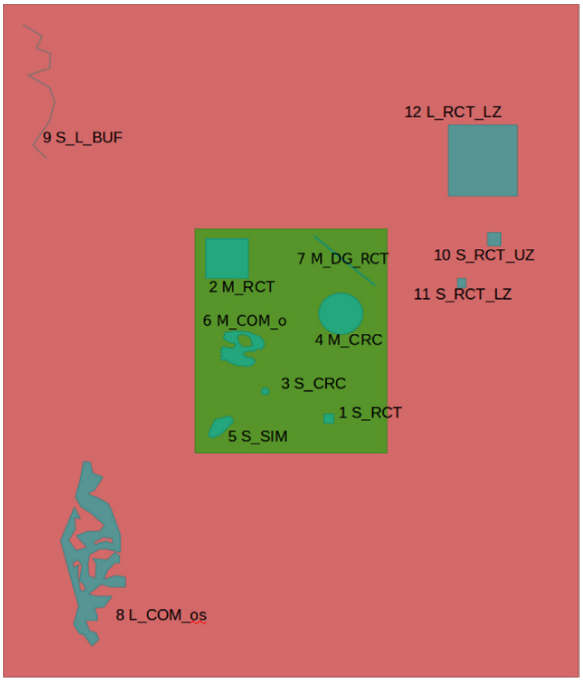

Fig. 2. Queries used in the medium benchmark (up to 210M extent).

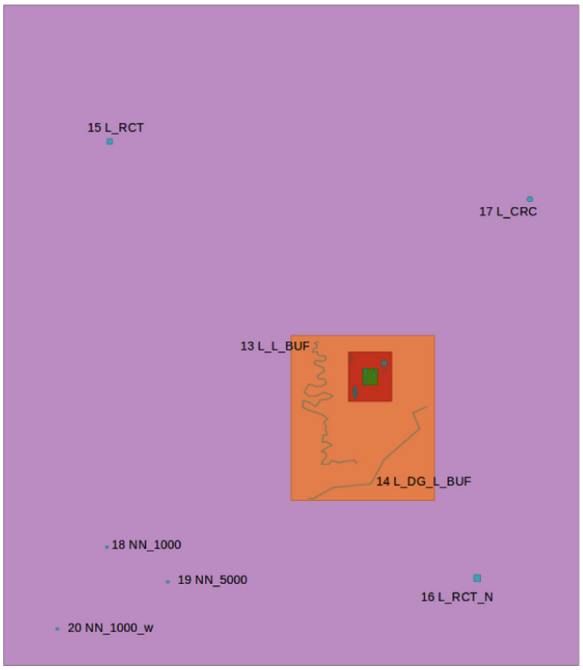

Fig. 3. Queries used in the medium benchmark.

3.2. Software

The versions of the different software used for these tests are libLAS 1.7.0, laszip 2.1.0, PostgreSQL (PostgreSQL 9.3.2, PostGIS 2.1.1, libgeotiff 1.4.0, GDAL 1.10.1 and PDAL), Oracle (Oracle Database 12c Enterprise Edition Release 12.1.0.1.0 - 64bit Production, with patch 24th July 2014 for faster blocking), MonetDB (20 June 2014), LAStools (15 July 2014), and lasnelsc (binary converters from LAS/LAZ to PostgreSQL and MonetDB; 30 June 2014).

3.3. General PCDMS description

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

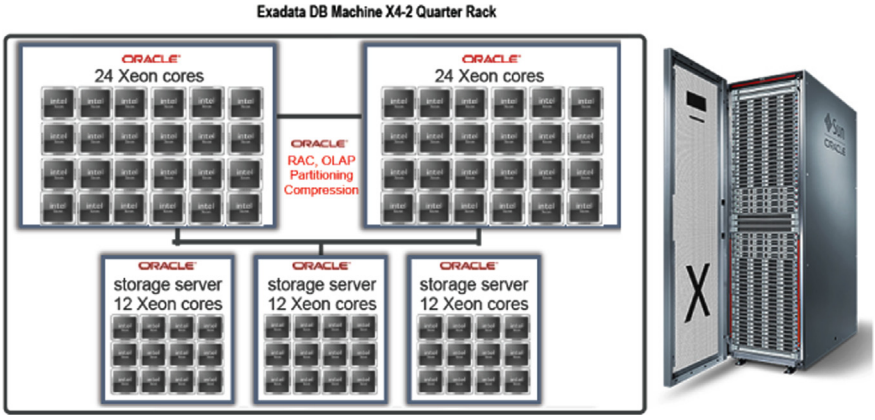

Fig. 4. The Exadata configuration (quarter rack).

Table 2
PCDMS descriptions.

Name	Software	Software description	Type
pf	PostgreSQL	Open-source RDBMS	Flat table
pb	PostgreSQL	Open-source RDBMS	Blocks
of	Oracle	Commercial RDBMS	Flat table
ob	Oracle	Commercial RDBMS	Blocks
oe	Oracle Exadata	Commercial RDBMS	Flat table
mf	MonetDB	Open-source column-store	Flat table
lt	LAStools	Commercial set of tools	File-based

All tests were executed on the HP DL380p Gen8 server, with the exception of the Oracle Exadata PCMDS, which ran on an Exadata X4-2 server.

PostgreSQL flat, Oracle flat, Oracle Exadata and MonetDB flat are PCDMS implementing the flat table storage model, i.e. the points are loaded in a regular table, one row per point. PostgreSQL blocks and Oracle blocks are PCDMS implementing the blocks storage model, i.e. the points are grouped into blocks which are stored in a DB, one row per block. In this case the loading process includes spatial reorganization for better spatial data clustering. Finally, using the LAStools software we have implemented a PCDMS that directly reorganizes and indexes the LAS or LAZ files. In fact, the LAStools PCDMS should be considered a hybrid approach because it also uses a database to manage the spatial extents of the files. This is used to avoid unnecessary processing of files which do not overlap with the queries. However, this support could also be done without database software, for example by using the Python shapely module.

In addition to gain experience with the involved systems, the *mini-benchmark* phase was also used to test many different configuration options for each PCDMS solution in order to determine the best configurations for the higher-scale benchmarks. In the following paragraphs we describe some of the investigated options.

One of the issues addressed in the *mini-benchmark* was the performance of the available compression techniques. Regarding the blocks-based PCDMS, PostgreSQL blocks have two compression modes: 'none' and 'dimensional'. In the first one, the point coordinates within each block are stored in the same way that in the LAS files, i.e. as 32 bit integers with certain offset and scale values, thus the name is misleading because there is indeed a sort of compression. In the 'dimensional' mode a certain compression algorithm per dimension is also applied. For more information see [2]. On the other hand, in Oracle blocks the compression is not point-cloud-tailored, the point coordinates are stored as floating points using the *binary double* type. There are three compression modes which are generic for the BLOB objects which are used to store the blocks: 'nocompress', 'medium' and 'high'. In the *mini-benchmark* we tested the different compression modes and we decided that it was worthy to use the available compression techniques. The test showed that in PostgreSQL blocks the 'dimensional' compression offers 2.5 better storage while only degrading 10% the query response times. In Oracle blocks using the 'high' compression mode offers 2 times better storage and the same query response times.

Regarding the flat tables PCDMS, a different conclusion was extracted from the *mini-benchmark*. In the PostgreSQL flat case we tested different data types, concretely *double precision* which is the more direct one, (scaled) 32 bit *integer* and *numeric* type. With the scaled 32 bit *integer* we were expecting a significant decrease of the storage but due to the huge row overhead added by PostgreSQL (mainly 23 bytes per row) that was not the case. Moreover, the coordinates had to be re-scaled during query execution which slowed the response times in around 10%. The *numeric* type has a very poor performance in algebraic operations that significantly degraded the query response times. Other database systems with less overhead per row may benefit more from using scaled 32 bit integers to store the point coordinates. However, in order to have a fair comparison we decided to store *X,Y,Z* as *double precision* numbers (*number* type in the case of Oracle), i.e. without compression.

In the LAStools PCDMS we store the data in LAS format. The *mini-benchmark* yielded the finding that even though LAZ files obviously offer much better compression (up to 12 times) the queries to LAS files are much faster (up to 8x) than queries to LAZ files so the compression did not compensate the query downgrade. The queries that were most affected by using LAZ were the smallest regions. The main reason is that LAZ uses a certain chunk size for its compression algorithm. When querying regions much smaller than the chunk size a lot of points are uncompressed that are not used. In fact, a similar behavior is also happening in the DB blocks-based PCDMS.

Another explored issue in the *mini-benchmark* was the size of the blocks in the blocks-based PCDMSs, i.e. the number of points per block. Concretely, the *mini-benchmark* demonstrated that PostgreSQL blocks benefits from smaller block sizes (compared to Oracle) but,

in order to avoid having an excessive number of blocks, we choose a value of 3000 points per block for the higher-scale benchmarks. From the *mini-benchmark* results we determined that Oracle blocks offer better performance with larger blocks (compared to PostgreSQL) but, in order to avoid having too large blocks, we choose a size of 10,000 points per block. Therefore, each system has to be configured differently to enhance its performance.

The loading/preparation of the data can be improved by using multiple processes. MonetDB and Oracle support it natively, while in PostgreSQL others we had to do the parallelization out-of-core. LAStools can use multiple cores for the data preparation tasks if the license is purchased. Since our tests were with the unlicensed version we also had to do an out-of-core parallelization.

About the querying, in all the tests the query results are dumped in a database table, one row for each selected point. We use the SQL statement *CREATE TABLE [name] AS SELECT*. In the case of LAStools the results of each query are dumped in a LAS file. Note that the PCDMSs implementing the blocks model are specially affected by this requirement because the blocks need to be unpacked which obviously requires CPU cycles. Below the first query of the benchmark in Oracle blocks:

```
CREATE TABLE query_res_1 AS
  SELECT PNT.X, PNT.Y, PNT.Z FROM
    TABLE (SDO_PC_PKG.CLIP_PC((SELECT PC FROM AHN_BASE),(SELECT GEOM FROM QUERY_POLYGONS WHERE ID=1),
      NULL,NULL,NULL,NULL)) PCBLOB,
    TABLE (SDO_UTIL.GETVERTICES(SDO_PC_PKG.TO_GEOMETRY(PCBLOB.POINTS,PCBLOB.NUM_POINTS,3, NULL))) PNT;
```

In the previous query statement the SDO_PC_PKG.CLIP_PC method is used. It selects the blocks that overlap with the query region with ID=1 stored in the QUERY_POLYGONS table. The selected blocks are processed and the points outside the query region are not included. Finally the blocks are unpacked with the SDO_PC_PKG.TO_GEOMETRY and the SDO_UTIL.GET-VERTICES methods. Only *X*, *Y* and *Z* are dumped in the table with the results.

See Appendix D for the complete list of different query statements for the various PCDMS.

About how the actual query measurements, the queries are executed in a “cold” environment, i.e. when no data is in memory/OS or DB cache, and also in a “hot” environment, i.e. when data has been already queried and it is in memory/OS or DB cache. However, in this paper we mostly present the response times of the “hot” queries. The reader can find the excluded measurements in the technical report available in the project web site [25].

We have also measured the query response times in the case of using one single process or in the case of using multiple ones. In this paper we only present the times when using multiple processes because they offer, in general, a better performance. Note that in some PCDMSs the query parallelization is native and/or automatic while in some others we had to implement it out-of-core. We implemented two different parallel algorithms: Parallel Query with Gridded Region (PQGR), and Parallel Query with Candidate Blocks (PQCB), described in Section 7. The algorithms offer similar performance, but for the experiments we chose the PQGR because it can be used in all the PCDMSs (while PQCB only works with blocks-based PCDMSs).

In the next subsections we present more details for each one of the used PCDMS.

3.4. PostgreSQL flat

This is a PostgreSQL implementation of the flat table storage. In other words, the points are loaded in a normal flat table in a PostgreSQL database, one row per point. We have developed a tool to convert LAS and LAZ files into PostgreSQL binary dump files. This is the fastest way of loading the point cloud data (up to five times faster than regular ASCII/CSV loading). PostgreSQL does not provide parallel loading tools but allows for several processes writing in the same table. We created a loading script that can process (convert and load) multiple files independently and in parallel. After all the data is loaded we created a B-Tree index on *X* and *Y* coordinates. We also tested a GIST index [27], but its creation was much more expensive and the query performance advantage was limited.

The creation of the index, contrary to Oracle and MonetDB, does not happen in parallel. One option would be to use partitions and index them separately but that is out of the scope of this test. Similarly, to compensate the lack of native parallel queries in PostgreSQL we used the Parallel Querying with Gridded-Region (PQGR) algorithm.

3.5. PostgreSQL blocks

In this PCDMS we use the point cloud extension developed by P. Ramsey [2] on a PostgreSQL database. It creates blocks of points and stores each block in a row of a table using the *PCPATCH* data type. The blocks are defined in the 2D (*X* and *Y*) space containing the 3D points. Note that the blocks themselves will be stored in a TOAST table, a feature of PostgreSQL that separates and stores the large objects in different tables called TOAST tables [28]. The creation of the blocks and the actual loading into the DB are done with an external tool called PDAL [29]. The input is processed per file, resulting in block creation per file. Block creation on whole dataset should result in a better blocking structure especially in the case of input files which are overlapping. As in the flat table case, PostgreSQL and PDAL are single-core processes but we use a script that can process multiple LAS or LAZ files in parallel and independently. The obvious drawback of this method is that it is sensitive to a good spatial organization of the input files. If the spatial extents of the input files are overlapping this method will also generate overlapping blocks. For the parallel query execution we used the Parallel Querying with Gridded-Region (PQGR) algorithm.

3.6. Oracle flat

This implementation of the flat table storage model in Oracle is similar to PostgreSQL one, the points are loaded in a normal flat table, one row per point. We considered using the Index Organized Table (IOT) feature of Oracle [30], since it decreases considerably (around 60%) the required storage and provides faster queries (thanks to the data clustering). However, in our experimental system the native

query parallelization of Oracle was not working with an IOT. This issue has been reported and it is under investigation. Loading was performed via an external table loader with a pre-processor script that uses the *las2txt* tool of the *liblas* toolset [31]. This loading method natively supports multiple processes. Once the data is loaded we create a B-Tree index which is also done in parallel.

3.7. Oracle blocks

This PCDMS is the Oracle implementation of the blocks model, i.e. the points are grouped into blocks defined in the 2D space. Each block is stored in a *SDO_PC_BLK* object (the points are stored in a BLOB). The point cloud metadata, including the global blocks extent, is stored in a separate object, the *SDO_PC*, also in a separate table. Similar to the Oracle flat table load, we used the external table loader to populate a staging table. In addition, the pre-processor script computed the 2D Hilbert value [32] for each point, which was also stored in the staging table. The staging table is an IOT table which has the Hilbert value as the primary key. As this was just staging table, we did not mind that parallel query was not yet working, but we benefited from efficient storage. When the entire set was loaded, the blocking process used the staging table to create the blocks. The chosen blocking method is a new Hilbert R-Tree blocking which creates blocks of points with consecutive Hilbert values. This blocking method generates overlapping blocks. Note that contrary to PostgreSQL it requires all the points to be loaded in the staging area. However, an incremental loading similar to the PostgreSQL blocks is under development and will be available in a future release.

3.8. Oracle Exadata

The Exadata solution is similar to the Oracle flat PCDMS except that is executed in an Oracle Exadata hardware. No indexes are defined and required because of the associative searching capabilities provided by the system. For the loading we use the same procedure as Oracle flat, except no index is created. From the different compression modes we chose to use the *query high* mode because it showed to offer the best trade-off between storage and query response times.

3.9. MonetDB flat

We also implemented the flat table storage model in MonetDB. Since this is a column-store RDBMS, each column of the three column (*X*, *Y*, and *Z*) is internally stored separately. We have developed a binary converter that converts a set of LAS or LAZ files into the MonetDB native binary columnar format. This converter works natively in parallel. MonetDB uses a novel indexing strategy, called Imprints [33] that creates index structures on-the-fly when the columns are queried. Such index structures are kept in memory-mapped files and loaded in memory depending on the usage. However, we detected that they disappeared after some time. After a request from the benchmark team, an option was added to MonetDB to make Imprints persistent in order to obtain more reliable benchmark results.

3.10. LAStools

We used a combination of LAStools tools together with a PostgreSQL database to create this file-based PCDMS. LAStools is a set of tools developed and maintained by Rapidlasso GmbH [7] that process LAS, compressed LAZ, Terrasolid BIN [34], Esri Shapefiles [35], and ASCII/CSV.

First we used *lassort* to reorder the points of each input file according to their position in a 2D Morton space filling curve [36]. Then we used *lasindex* to create an index file, based on a Quad-Tree, which is used to speed up querying. Finally we inserted the extent of each file in a PostgreSQL database. This step can also be done without database, using for example the *shapely* Python module. All these operations were executed in parallel for different files using a Python script. The tools can also work with the LAZ format but the *mini-benchmark* revealed that the best performance in data preparation and mainly in querying is when the LAS format is used. The performance difference in queries is more noticeable in simple and small query regions.

For the actual querying we used the tools *lasmerge* for rectangles and circles, and *lasclip* for the rest of geometries. Before executing these tools we executed a database query to locate the files overlapping with the query region. Only these files are given as input to the LAStools tools. Note that from the used tools, *lasindex* and *lasmerge* are free and open source but *lassort* and *lasclip* are commercial tools, the unlicensed version used in the tests added some distortion to the points.

lasclip has an option to use multiple processes (limited to 2 cores with the unlicensed version) but when tested it did not really improved so only one processes is used in *lasclip* and *lasmerge*. Note that, as in PostgreSQL, this system works nicely when the input data has already good spatial organization, i.e. the spatial extents of the files have none or very small overlapping. If that is not the case a re-organization of the data should be done in advance with the commercial tool *lastile*.

4. Medium benchmark execution

In this section we present the main results of the *medium-benchmark*, i.e. loading of datasets and execution of the first 20 queries described in Appendix B. For more details we refer the reader to the technical report available on the project web site [25].

4.1. Loading

In this test all the input datasets are in the LAS format. However, note that all the tested PCDMS are compatible with the LAZ format as well.

The loading or preparation procedure is split into three phases: (a) *initialization*: the databases and tables are created and the required extensions are loaded, (b) *loading*: the bulk of the data is loaded or prepared, and (c) *close*: the required indexes are built. These three phases are perfectly defined in the flat table PCDMSs but in the other ones the exact operations in these phases are PCDMS-dependent. In

Table 3
Times and sizes of the data loading procedure for the different PCDMSs and datasets. The names of approaches encode the PCDMS name (o for Oracle, p for PostgreSQL, etc.), flat or blocked model (f and b, respectively), and the dataset name, as described in Section 2.3. For example *ob2201M* stands for the dataset 2201M loaded in the Oracle blocks PCDMS.

Approach	Time (s)				Size (MB)		Points	Points/s
	Total	Init.	Load	Close	Total	Index		
pf20M	44.07	2.00	13.86	28.21	1558.71	551.83	20,165,862	457,587
pf210M	771.63	2.09	71.44	698.10	16,249.24	5762.20	210,631,597	272,970
pf2201M	9897.37	1.15	722.91	9173.31	169,776.13	60,214.46	2,201,135,689	222,396
pf23090M	95,014.05	3.64	8745.90	86,264.51	1,780,963.91	631,663.71	23,090,482,455	243,022
of20M	128.43	0.51	123.92	4.00	879.50	453.85	20,165,862	157,018
of210M	275.72	0.32	230.01	45.39	9124.50	4739.94	210,631,597	763,933
of2201M	1805.68	0.31	1228.81	576.56	95,825.00	49,533.74	2,201,135,689	1,219,007
of23090M	15,825.53	0.14	9226.37	6599.02	997,062.50	519,621.48	23,090,482,455	1,459,065
mf20M	7.15	1.14	3.70	2.31	475.78	14.09	20,165,862	2,820,400
mf210M	29.15	1.13	16.63	11.39	4888.05	66.95	210,631,597	7,225,784
mf2201M	304.14	4.91	198.76	100.47	50,661.30	281.17	2,201,135,689	7,237,245
mf23090M	8490.90	0.93	5466.99	3022.98	529,448.70	949.37	23,090,482,455	2,719,439
pb20M	153.41	22.03	130.53	0.85	101.77	0.69	20,165,862	131,451
pb210M	129.14	0.81	121.00	7.33	1009.13	5.18	210,631,597	1,631,033
pb2201M	754.22	0.86	687.49	65.87	10,245.61	53.05	2,201,135,689	2,918,427
pb23090M	12,263.05	0.87	7450.10	4812.08	106,781.48	552.77	23,090,482,455	1,882,931
ob20M	296.22	0.35	228.17	67.70	226.50	0.20	20,165,862	68,077
ob210M	1246.87	0.25	557.70	688.92	2244.50	1.44	210,631,597	168,928
ob2201M	16,737.02	0.86	7613.39	9122.77	21,220.50	13.25	2,201,135,723	131,513
ob23090M	192,612.07	0.31	96,148.06	96,463.70	220,085.50	165.55	23,090,482,953	119,881
lt20M	9.49	0.03	9.46	0.00	384.65	0.02	20,165,862	2,124,959
lt210M	30.07	0.02	28.68	1.37	4021.37	3.88	210,631,597	7,004,709
lt2201M	218.06	0.02	216.18	1.86	41,992.78	9.40	2,201,135,689	10,094,174
lt23090M	2129.30	0.03	2116.64	12.63	440,484.48	68.04	23,090,482,455	10,844,166
oenc21090M	508.71				537,600.00	0.00	21,090,482,455	41,458,753
oeql21090M	619.14				218,504.00	0.00	21,090,482,455	34,064,157
oeqh21090M	928.84				94,240.00	0.00	21,090,482,455	22,706,269
oeal21090M	1149.55				93,992.00	0.00	21,090,482,455	18,346,729
oeah21090M	1767.53				59,096.00	0.00	21,090,482,455	11,932,177

the PostgreSQL blocks, the blocks are created by the PDAL program and dumped in the DB during the loading phase. In Oracle blocks, the staging IOT table is filled in the loading phase, but the blocks are not yet created. This means that the sorting of the points happens in the loading phase but the actual blocks creation, writing and indexing happens in the close phase; In the LAStools-based PCDMS we do *lassort* and *lasindex* in the loading phase and in the close phase we fill the database with the extent of the files. The exact steps followed during the loading or preparation of the data are described in detail in Appendix C.

Prior to the execution of the loading procedure for each dataset and PCDMS, the server memory and OS cache was flushed so we can guarantee that the loaded data was not in memory when the loading started. As previously stated the loading was done in parallel. In the case of MonetDB and Oracle this is automatically done. In the case of PostgreSQL and LAStools we performed an out-of-core parallel loading with 16 processes using a Python script based on the *multiprocessing* module.

Table 3 presents the details of the loading performance for different datasets and for 7 PCDMS. In addition to the times and sizes presented in this paper, the CPU and the memory were also monitored and their average values can be found in the *medium-benchmark* report in the project web site where detailed graphs with CPU, memory and IO usages for each case can also be found.

The results show that LAStools offers the fastest way to load or prepare the data among the solutions run on the main test hardware.

Oracle Exadata is faster, but it runs on different hardware (Exadata X4-2 half rack). It is a special case also with respect to the dataset used. The set 21090M is a bit smaller compared to 23090M used by the other PCDMSs. We tested the five levels of compression in Oracle Exadata described in Section 3.1. Each level has different load times and final storage sizes. After some exploratory testing, it was decided that query high compression (*oeqh*) provides the best balance among load time, storage size, and query times.

The loading times of the three flat table solutions are in the same order of magnitude. Nonetheless PostgreSQL and MonetDB, which were using the binary loaders, are slightly faster than the ASCII-based loading in Oracle. However, MonetDB suffered from a degrade in the loading performance in the 23090M dataset. This issue was reported and it has been already addressed. The memory management during the loading process has been improved by reducing the virtual memory utilization by the intermediate structures. The Imprints creation has been improved through a reduction on the computational complexity of the sampling function. Index creation in PostgreSQL flat was very slow in comparison to the other systems (typically a factor of 10–20 slower), because it works with one single process.

Regarding the storage requirements of the flat-table-based PCDMSs, MonetDB requires the least total storage mainly due to the columnar organization and the small amount of storage required by the Imprints index. Oracle requires less storage for the flat table itself but much more for the B-Tree index. This can be solved by using an IOT but, as previously stated, we discarded this solution because parallelization of queries did not work in the tested system when using an IOT. The PostgreSQL flat solution requires storage twice larger than Oracle due to the large overhead per row.

Among the block-based solutions, PostgreSQL blocks together with PDAL and the parallel loading offered loading one order of magnitude faster than the current Oracle blocks loading. However, it must be noted that the Oracle blocks are created globally, i.e. after entire set is loaded and sorted, while PostgreSQL–PDAL blocks are created independently for each file. PostgreSQL–PDAL benefited from the fact that our test data was already spatially split into different non-overlapping and well organized files. A blocking method that supports incremental blocks creation for each file in Oracle is currently being implemented.

The creation of the staging IOT in Oracle blocks, which happened during the loading phase, took five times more than we expected. The procedure itself is similar to the table loading and indexing (loading and close phases) in the Oracle flat PCDMS, with the exception that the Hilbert value is also computed and that the indexing uses the Hilbert value instead of X and Y. One of the possible reasons for this slowness is that the Hilbert value needs to be re-factored in order to be unique and this converts it into a large number with many digits and the comparisons with such numbers are much slower. The close phase, which created the blocks, was also slower than expected. In theory, it should be a straight forward procedure, since it is only "chopping" parts of the already sorted staging IOT. This issue is currently under investigation.

Observing the memory usage in Oracle blocks during the close phase of the dataset 23090M loading, i.e. when the blocks were actually being created, we realized that it was above 100% which meant that a considerable amount of data swapping between disk and memory was happening. Oracle blocking required all the points to be simultaneously available and in this case they did not fit the system memory. This behavior was one of the reasons that made Oracle developers explore the incremental loading. In addition, Oracle blocks in 2201M and 23090M had a slightly higher number of points when compared to the other PCDMSs. This issue is also under investigation.

The compression achieved by both block-based PCDMSs is very good. In fact, they were even better than the LAStools-based solution. However, LAStools solution could also use LAZ format, which would decrease size in one order of magnitude, but it would also make the data preparation and the simple and small queries slower. PostgreSQL blocks offered the best storage thanks to the dimensional compression. See [2] for more information regarding the compression.

4.2. Querying

In this section we show the number of points and the response times of the first seven queries of the *medium-benchmark* as presented in Appendix B. The results of the rest of the 20 queries can be found in the *medium-benchmark* report [25]. For all the queries more detailed graphs with CPU, memory and IO usages were generated and can be found in the project web site. Queries 1 and 2 are rectangles, 3 and 4 are circles and the rest are polygons. The actual query statements for the different PCDMSs can be found in Appendix D.

For all the queries we executed a cold run and a hot run. A cold run means that any data is in memory while in a hot run the queries have already been executed so caching effects are taking place. Therefore, the exact order in the execution of the queries was Q1, Q2 … Q20, Q1, Q2 … Q20.

In MonetDB and Oracle flat the parallelization of the queries was automatic. In all the queries executions we monitored the CPU, memory and IO usages and, in the case of MonetDB and Oracle flat, we indeed observed that more than one core was used, the exact number depended on the system and the query. In the cases where the parallelization was not automatic we also executed both with a single process and with multiple processes using the PQGR algorithm with eight processes. In general using more than eight processes did not improve the queries, the bottlenecks causing these behaviors depend on several factors: region size, region complexity, cold or hot run, storage model, etc. Moreover, additional tests revealed that in few cases using multiple processes offered worse performance than using a single process.

In the LAStools queries we use *lasmerge* for rectangular and circular regions and *lasclip* for other generic polygonal regions. These tools process the set of input LAS files through their related LAX index files and create new LAS files with the points in the query regions. However, before each query we use the DB to determine which files are required and only those are provided as input to *lasmerge* and *lasclip*.

Table 4 presents the details of the hot run of the first seven queries for the different approaches using multiple processes. The number of selected points in the case of the Oracle Exadata PCDMS is lower, due to the fact that a different data was used. However, it gives an indication of the Oracle Exadata query performance using the query high compressed storage compared to the other PCDMSs. Based on this experience, the same compression is going to be used in the full benchmark of Oracle Exadata.

Moreover, comparing the rest of PCDMS we observed that the numbers of points were slightly different in many other cases. The list of reasons for this behavior is:

- LAStools has a slightly different number of points in all the cases mainly due to the added noise in *lassort* and *lasclip*. In the case of using a licensed version of LAStools we still noticed a lower number of points. For example, in the rectangles, the points in the right and the top edges were excluded.
- LAStools, concretely *lasclip*, ignores the holes in the polygons, so that query 6 has more points than the rest of the approaches.
- The PCDMSs using the PQGR algorithm, i.e. PostgreSQL flat, PostgreSQL blocks and Oracle blocks had some duplicated points due to the nature of the algorithm, the points in the edges of the sub-regions are duplicated. Note that this could be relatively easy cleaned afterwards by a 'select distinct' (but this was not yet included in the tests). Another solution would be to develop a different 'overlap' function, which in the case of points on boundary only includes these in result when selection area is to lower-left of edge (not yet implemented).
- The flat-table-based PCDMSs use the distance method for the circle areas, while the blocks-based PCDMSs use a polygon approximation of the circle.
- The PostGIS intersection methods used in PostgreSQL flat PCDMS excluded some points on the edges of the query regions.
- In PostgreSQL flat query 6 we observe that the same query has a different number of points in different datasets. MonetDB also has some differences between datasets in query 6. This should be further investigated. Note that query 6 is a polygon with a hole.

Regarding the query response times we observed:

- In rectangles and circles (queries 1–4) LAStools using *lasmerge* was clearly faster in all the cases. In other query regions (queries 5–7) the performance of LAStools using *lasclip* was quite degraded and the PCDMSs using blocks presented better performance.

Table 4
Comparison of number of points returned and response times by the hot queries 1 to 7 in the different approaches.

Approach	Number of points							Time (s)						
	1	**2**	**3**	**4**	**5**	**6**	**7**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
pf20M	74,947	718,131	34,637	562,919	182,792	387,134	45,805	0.35	2.25	0.24	1.90	1.18	1.72	0.84
pf210M	74,947	718,131	34,637	562,919	182,792	387,135	45,805	0.42	2.50	0.27	2.26	0.91	1.65	1.34
pf2201M	74,947	718,131	34,637	562,919	182,792	387,135	45,805	4.92	19.03	2.90	18.28	9.37	17.71	10.16
pf23090M	74,947	718,131	34,637	562,919	182,792	387,134	45,805	5.17	18.02	3.32	17.75	9.42	15.46	13.50
of20M	74,872	718,021	34,691	563,037	182,861	387,145	45,813	0.24	0.37	0.28	1.85	0.75	1.32	1.32
of210M	74,872	718,021	34,691	563,037	182,861	387,145	45,813	0.45	0.58	0.52	1.27	1.12	1.47	1.79
of2201M	74,872	718,021	34,691	563,037	182,861	387,145	45,813	1.47	3.87	1.29	4.26	4.38	6.60	5.24
of23090M	74,872	718,021	34,691	563,037	182,861	387,145	45,813	1.25	18.20	2.34	22.75	6.99	27.18	635.06
mf20M	74,872	718,021	34,691	563,037	182,861	387,134	45,813	0.06	0.13	0.06	0.20	9.96	187.16	38.71
mf210M	74,872	718,021	34,691	563,037	182,861	387,135	45,813	0.13	0.26	0.15	0.28	9.95	185.65	38.56
mf2201M	74,872	718,021	34,691	563,037	182,861	387,135	45,813	0.64	0.90	0.64	0.77	10.37	186.38	39.17
mf23090M	74,872	718,021	34,691	563,037	182,861	387,134	45,813	7.21	16.74	9.70	9.88	17.94	198.51	43.96
pb20M	74,947	718,131	34,697	563,108	182,930	387,142	45,821	0.32	2.14	0.20	1.69	0.61	1.72	0.41
pb210M	74,947	718,131	34,697	563,108	182,930	387,142	45,821	0.32	2.15	0.20	1.65	0.64	1.62	0.46
pb2201M	74,947	718,131	34,697	563,108	182,930	387,142	45,821	0.31	2.19	0.21	1.67	0.67	1.63	0.41
pb23090M	74,947	718,131	34,697	563,108	182,930	387,142	45,821	0.32	2.19	0.21	1.68	0.68	1.68	0.44
ob20M	74,947	718,131	34,697	563,110	182,930	387,145	45,821	0.41	1.38	0.34	1.21	0.62	1.38	0.53
ob210M	74,947	718,131	34,697	563,110	182,930	387,145	45,821	0.38	1.28	0.36	1.22	0.62	1.29	0.54
ob2201M	74,947	718,131	34,697	563,110	182,930	387,145	45,821	0.39	1.36	0.36	1.23	0.60	1.33	0.50
ob23090M	74,947	718,131	34,697	563,110	182,930	387,145	45,821	0.40	1.30	0.34	1.21	0.60	1.40	0.53
lt20M	74,840	717,931	34,695	563,049	182,849	460,068	45,834	0.04	0.12	0.03	0.09	0.66	1.48	0.51
lt210M	74,840	717,931	34,695	563,049	182,849	460,068	45,834	0.06	0.14	0.05	0.14	0.67	1.51	0.51
lt2201M	74,840	717,931	34,695	563,049	182,849	460,068	45,834	0.06	0.12	0.05	0.14	0.70	1.51	0.51
lt23090M	74,840	717,931	34,695	563,049	182,849	460,068	45,834	0.05	0.15	0.05	0.14	0.68	1.50	0.52
oeqh21090M	40,368	369,352	19,105	290,456	132,307	173,927	9559	0.18	0.35	0.59	0.72	0.66	0.67	0.46

- The LAStools PCDMS and the blocks-based PCDMSs did not have noticeable scaling effects but that was not the case in the flat-tables-based PCDMSs where there was a clear effect of having a larger dataset. In the case of PostgreSQL there is an abrupt drop in the performance with the 2201M dataset while in MonetDB and Oracle the abrupt drop is with the 23090M dataset.
- For rectangles and circles and small datasets up to the 2201M dataset, MonetDB was faster than the other flat-table-based PCDMSs. The polygon-based queries in MonetDB showed at the moment bad response times due to a inefficient implementation of the function checking if a point is in a polygon. This issue has been reported to the MonetDB developers.
- In general Oracle flat was faster that PostgreSQL flat for the smaller datasets.
- Both PCDMSs using blocks offered similar performance. Note that the query accuracy is at point level, which means that the blocks-based PCDMSs had a significant overhead due to the need to unpack the blocks for the actual points processing, i.e. to check whether individual points were within the query regions and also when dumping the selected points in the results table. For this reason the blocks-based PCDMS were only worthy with complex and large-area queries. When many blocks were involved the described overhead was compensated by the fact that much of the spatial filtering could be done at block level, checking individual points could be avoided in the cases where the block extents were completely inside the query region.
- In parallel tests with LAStools and LAZ we observed that a significant overhead due to the compression was added in the queries. The data chunks used in the LAZ compression algorithm made that even for small query regions the tools still had to unpack large data chunks before the individual points could be checked. This extra overhead was somehow compensated when dealing with large query regions.

Even though not shown in Table 4 the difference between the cold and hot queries were also analyzed:

- In LAStools the difference was significant and we think it was due to the data organization done in the data preparation phase. *lassort* reorders the data in each file according to a Morton curve which has a direct relationship with a Quad-Tree, and the LAX file maintains which parts of the file are in each Quad-Tree cell. Thanks to the reordering done by *lassort* each Quad-Tree cell will be a single and continuous part of the file. Hence, during the cold run of the queries the parts of the LAS files indicated by the LAX files were loaded and kept in the OS cache. The hot queries already had the LAX files and the small parts of the LAS files in memory so the query re-runs were done much faster.
- The differences were also quite noticeable in the flat table PCDMSs: PostgreSQL flat presented the largest difference, followed by Oracle and finally MonetDB which only showed the effects on some cases. These PCDMSs were not as efficiently organized as the LASTools approach. Their index structures, B-Tree in Oracle and PostgreSQL, were indexing unsorted data. Hence, the number of parts of the main table that needed to be fetched for each query was larger than in the LAStools case. For example, in PostgreSQL the points were spread in more DB pages than if they were clustered. And even though these parts could be reused in the hot queries the individual point retrieval was much slower than if the points were sorted.

Table 5
Full-benchmark loading results for the LAStools and Oracle Exadata PCDMSs.

System	LAStools	Oracle Exadata
Total load time	22:54 hours	4:39 hours
Total size	12,181 Tb	2,240 Tb
Points	638,609,393,087	639,478,217,460

Table 6
Full benchmark query results of LAStools and Oracle Exadata. Notes: (a) Nearest neighbors queries (18, 19 and 20) were not executed as functionality was not implemented, and (b) Oracle Exadata query 25 was also re-run using an MBR instead of a geometry close to an MBR with and improved the time to 353.93 s with 3.6546E+10 selected points.

Query	LAStools			Oracle Exadata	
	Points	Time (s)		Points	Time (s)
		DB	No DB		
1	74,861	0.07	0.90	74,863	0.48
2	718,057	0.16	0.87	718,070	0.79
3	34,700	0.07	0.78	34,675	1.22
4	563,119	0.16	0.92	563,082	1.69
5	182,871	0.70	33.24	182,875	1.43
6	460,140	1.52	32.79	387,201	1.29
7	45,831	0.55	32.29	45,815	1.71
8	2,365,925	3.72	36.21	2,273,469	2.86
9	620,568	2.34	34.76	620,719	1.58
10	2413	0.08	0.88	2434	0.40
11	591	0.05	0.84	591	0.44
12	343,168	0.26	1.03	343,171	0.60
13	897,042	412.29	829.49	897,359	23.34
14	765,989	102.19	424.91	766,029	15.05
15	3,992,330	0.49	1.39	3,992,290	2.23
16	0	0.04	0.75	0	0.00
17	2,203,066	0.32	1.18	2,201,280	2.51
21	382,395	2.28	20.74	382,335	0.95
22	12,148,049	142.09	1115.27	12,147,802	113.37
23	691,422,551	313.14	828.51	691,422,526	330.85
24	2,468,239	234.40	4261.77	2,468,367	393.21
25	–	–	–	3.5319×10^{10}	1193.10
26	2,124,162,497	282.89	1124.04	2,124,162,754	25.79
27	27,443	0.13	923.59	27,459	1.23
28	809,097,723	1553.87	1885.54	866,802,585	67.67
29	1,509,662,615	3438.34	5697.79	1,509,662,411	120.02
30	–	–	–	1.3443×10^{10}	3569.54

In addition, using a B-Tree on XY used in PostgreSQL flat and Oracle flat is not a very suitable indexing technique for point clouds. An improvement could be done by using a better data organization in the flat tables based on space filling curves like the proposed in Section 6.

- The blocks-based PCDMSs presented minor differences, in the case of Oracle blocks they were unnoticeable. As in the previous cases we hypothesized the reason for such behavior based on the way the data was organized. The data was organized in blocks but the query accuracy was at point level, thus causing that the actual processing affected the points and what was kept in the DB cache were the points and not the blocks. When the query was repeated the blocks had to be read again, the DB cache system did not know that the points that were in cache belonged to the blocks that were being retrieved again. In other words, it was a matter of the granularity, i.e. how the data was stored and how the data was queried.

5. Full benchmark execution

Two *full-benchmark* executions have been completed until moment of paper submission: one with the LAStools-based PCDMS and other with the Oracle Exadata PCDMS. During the *medium-benchmark* these platforms showed very promising performance for both loading and querying. Note that the full AHN2 dataset is provided in LAZ format, contrary to the data-sets used in the *medium-benchmark*. Also contrary to the previous test, an Oracle Exadata X4-2 full rack hardware was used. Other PCDMS have not been yet tested in this benchmark stage mainly due to either unaffordable data loading times or very bad scaling perspectives.

Given the amount of data, the Oracle Exadata machine used High Capacity disks instead of High Performance disks, similar to the disk storage attached to the HP DL380p Gen8 sever, which also are more focused on high capacity and not high performance. Moreover, we used partitions based on a grid with 20,655 cells. This somehow implied a form of spatial organization. In the future we plan to also perform tests without partitions. For more information regarding partitions in Oracle see [37].

LAStools PCDMS storage was based on uncompressed LAS files while Oracle Exadata worked on compressed data with the query high compression mode. This choice was justified by the *mini-benchmark* test results, namely that using compressed LAZ files slows down the

query performance by a factor of 2–3, and sometimes up to a factor of 8. Tests with LAStools over compressed data could be included in the future activities.

5.1. Loading

For the load of the full AHN2 dataset in the Oracle Exadata PCDMS an elegant approach based on piping was used: starting from the compressed LAZ files, decompressing, conversion into Oracle's ASCII load format, loading and finally storing compressed in a flat table. LAStools data preparation consisted of: (a) *lassort* where the data was re-sorted and uncompressed, i.e. the format was changed from LAZ to LAS, (b) *lasindex* where the LAX files were created, and (c) the supporting DB was populated with spatial extents of the files for faster querying.

Table 5 shows the times and sizes of the data loading and preparation. LAStools data preparation took 22:54 h and exposed several issues. *lassort* crashed for files with more than 60 million points and in few other cases the *lassort* processes remained hanging and had to be manually terminated. These issues have been reported to the LAStools developers and they also explain the lesser number of points loaded in the LASTools PCDMS. Without these problems the data preparation time should be around 17:00 s. The data loading in Oracle Exadata with full rack took 4:39 hours and no problems were found.

LAStools total size was around 12 TB, while LAZ storage would have been around 1 TB, but was dismissed due to impairing query performance. The LAStools indexes, i.e. the LAX files, are relatively modest in size, less than 1 GB. On the other hand, Oracle Exadata takes only 2.2 TB and does not use any indexes.

5.2. Querying

Like in the *medium-benchmark* the queries were executed in cold and hot environments. LAStools queries were executed with one single process because the multi-core support is not really beneficial for the queries while Oracle Exadata natively exploits all the available cores. All the queries were executed as CTAS. However, in future tests the regions 22, 23, 25, 26, 28, 29 and 30 will be used to test aggregate queries, i.e. minimum, maximum or averages.

Table 6 shows the number of selected points and the response times for hot execution. Similarly to what we observed in the *medium-benchmark*, the number of selected points was often not exactly the same due to earlier mentioned reasons.

In order to see the effect of using DB support with the extents of the LAS files, two timings are shown for LAStools: with DB support, as also shown in the *medium-benchmark*; and without DB support. It is remarkable that even with over 60,000 LAS files and the same number of related LAX files, small and simple queries like 1, 2, 3, and 4 could be responded in less than 1 s in the case of the hot queries and without DB help. Only the very first query in the case of LAStools 'No DB' approach was significantly slower in the cold run, 738.15 s (not shown in Table 6 which contains only hot runs). Most likely the time was spent to access all the LAX files for the first time. The files remained afterwards in the OS/file system cache and were readily available for the subsequent queries.

For larger queries, such as queries 13, 14, 26, 28, 29, Oracle Exadata was faster. In queries 25 and 30 LAStools *lasclip* crashed, this has been reported to the LAStools developers. For small queries LAStools was faster partly due to the database meta-data management added to the implementation. When comparing the Oracle full rack Exadata query times of the first 7 queries (full benchmark) to the Oracle half rack Exadata query times as reported in previous section (medium benchmark), it can be observed that the full benchmark query times are slower (factor 2–3), while one would expect that these times would be better (up to factor 2 due to double amount of hardware). Currently, we have no explanation for this issue, it is still under investigation.

One total overall performance indicator of how fast after getting data a user can actually do something with it is summing the load and query times.

5.3. Benchmarks outcome

Our benchmark shows that different systems have different advantages and drawbacks and the choice of the best PCDMS depends on the exact requirements. If a database solution is preferred, the recommended option is to use flat tables up to few billion points, the exact limit is related to the amount of memory of the system. For relatively small datasets the response times are, in general, better than the blocks-based PCDMSs. For larger datasets we recommend using the blocks-based PCDMSs since they offer more constant response times. However, for rectangular and circular queries, the fastest option is using LAStools but take into account that the unlicensed version of this toolset is currently distorting the points when resorting the data and when using *lasclip*.

LAStools offers a range of impressively fast tools to directly work on LAS and LAZ data with some interesting, but limited, functionalities, i.e. some database-like functionalities are missing, i.e. total integration with other types of data, real multi-user environment support, etc. (see Section 1). In addition, with large datasets own solutions had to be developed in order to avoid processing all the files for every query. Unfortunately LAStools uses only one single process, using multiples processes in a more efficient manner could be very beneficial for selecting large volumes of point cloud data. Parallel algorithms like the ones proposed in Section 7 could be used for this purpose.

Another possible limiting factor in LAStools could in the cases where there is lot of overlap between the content of the input LAS or LAZ files. In such cases LAStools might give a inferior query performance. Note that also the blocks approaches that do not avoid overlap may also see their query performance degraded. This adds an option for future testing and it can be easily simulated with artificial repetition of files to create overlap.

On the other hand, using DB approaches one can get a more flexible range of functionalities provided by all the features of the DB technologies. The disadvantage is that performance in some specific tasks will not be as good as file-based solutions like the LAStools. The most clear example in our benchmarks is in small and simple queries where LAStools outperformed the rest of DB solutions. In addition, in many of the tested DB systems the loading times are excessively large but thanks to the benchmarks we could identify the main loading bottlenecks, these have been reported to the DB developers.

6. Improved data organization

Given the size of point cloud data, several techniques have to be applied to work efficiently with these large datasets. Besides using the new point cloud data types implementing the blocks storage model, for reference and comparison the flat table model has also been benchmarked. It was used to measure the improved efficiency of the point cloud data types in the databases, in terms of amount of storage needed and query performance.

The flat table storage can also be considered as a flexible intermediate stage, where it is relatively easy to organize and sort points for various purposes. The sorting of the points can be done, for example, according to their Hilbert-code [32] or Morton-code [36], i.e. the position of the points in the space filling curves; see Fig. 5.

Sorting the point using space filling curves is useful to exploit the spatial coherence of the data creating, for example, spatial location codes [38]. LAStools with *lassort* and *lasindex* is also based on these principles. After preparing the points in the flat table, it is easier to create more efficient and compact blocks over the sorted rows. Such blocks have several benefits: (a) they are spatially coherent, (b) they can be used for simple compression, for example the first digits of all coordinates are equal and could be stored at block level, (c) they have less tuple overhead compared to flat table model, (d) they may be used for caching, etc. The new Hilbert blocking algorithm already uses space filling curves for faster blocks creation. However, the used compression techniques do not exploit the spatial coherence of the blocks. On the other hand, PostgreSQL is already using compression techniques based on the spatial coherence of the blocks offering better storage when compared to Oracle. However, the option to use space filling curves for the blocks creation is missing.

A different approach is to use the flat table model directly in the case where no point cloud data type is available or in the case where the database hardware and software can handle very well compression and fast selections on flat tables, like the Oracle Exadata machine.

Therefore, one of the possible improvements of point cloud data storage in the database comes from a better data organization. In this section we detail a new flat-table-based PCDMS using an alternative data organization technique which has been tested in PostgreSQL. For each point a 64 bit 2D Morton code [36] with x-y bit interleaving is computed. The "unscaled" integer values of *X* and *Y* coordinates in the LAS files are used. The Morton code is used to index and cluster the points in an efficient manner, enabling faster queries.

6.1. Loading procedure

The loading procedure in this new PCDMS has the following phases: (a) initialize the DB and the table, (b) after adapting our binary loader to compute also Morton codes, we load in the DB the usual x, y, z and also the Morton code, (c) create a B-Tree index on the Morton code, and (d) cluster the main table, i.e. reorder the points, according to the created index. The actual SQL and command-line statements are

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

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

- Parallel processing using Python and its *multiprocessing* module to convert and load the files (Command-line example for a certain file):

```
las2pg -s tile_84000_446000.las –stdout
–parse xyzk | psql [connection parameters]
-c "copy ahn_flat from stdin with binary"
```

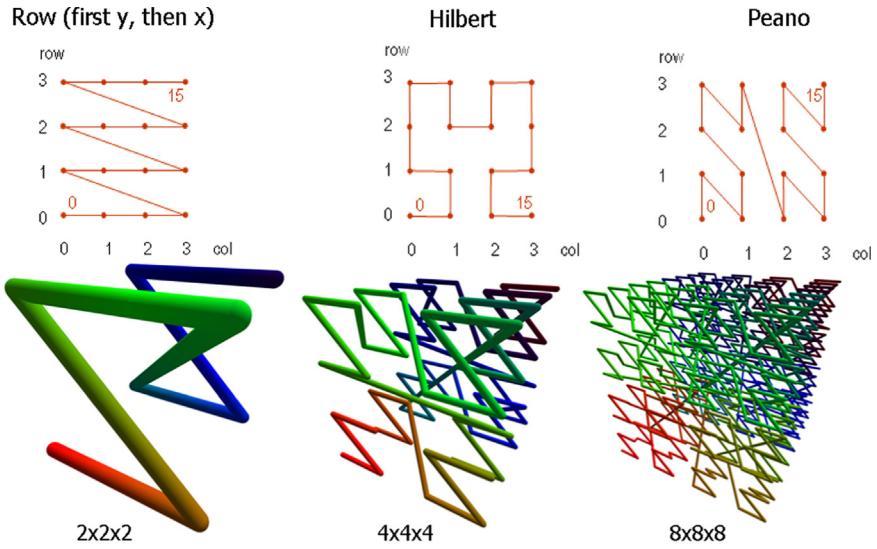

Fig. 5. Space filling curves. Top: 2D row, Hilbert and Peano/Morton curves; bottom: 3D Peano/Morton curves of various sizes (source: http://asgerhoedt.dk).

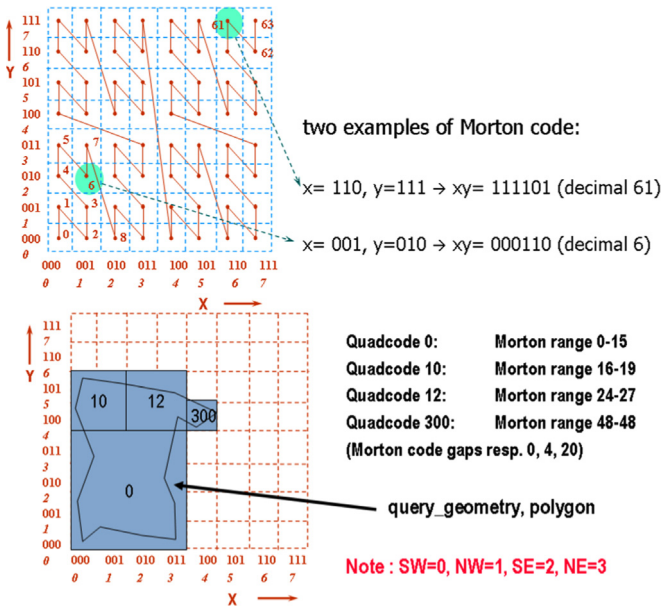

Fig. 6. Top: Morton code computation of single cell by bitwise interleaving *x* and *y*; bottom: ranges of Morton codes for polygon (Quad-Tree relationship).

Note that the parse option also includes *k*, the 2D Morton code.

- Create the 1D B-Tree index on the Morton code and cluster the data based with the index (SQL):

```
CREATE INDEX afm_btree_idx on   ahn_flat (m) WITH (FILLFACTOR=99);
CLUSTER ahn_flat USING afm_btree_idx;
```

6.2. Query algorithm

We modified the spatial selection predicate in the queries to make use of the Morton code. This is based on the direct relationship between the Morton space filling curves an the Quad-Tree structures [39], see Fig. 6 for an example.

The actual query procedure is as follows: (a) first we find the Quad-Tree cells, down to a certain level, overlapping with the query region. Fig. 7 shows an example of the overlapping Quad-Tree cells for the first seven queries. (b) Since each Quad-Tree cell has a direct mapping into a range of Morton codes we convert the overlapping Quad-Tree cells into ranges of Morton codes. (c) We extract all the points in these ranges. Since the data is clustered on the Morton code, each range is a set of consecutive physical pages on disk. (d) Finally we filter out points outside of the query region using the standard PostGIS intersection methods as used in the PostgreSQL flat PCDMS described in the medium benchmark.

For example, the actual query statement for query region 1 (rectangle) is:

```
CREATE TABLE query_results_1 AS (
SELECT x,y,z FROM (
SELECT x,y,z FROM ahn_flat WHERE (
(m between 1341720113446912 and 1341720117641215) OR
(m between 1341720126029824 and 1341720134418431) OR
(m between 1341720310579200 and 1341720314773503) OR
(m between 1341720474157056 and 1341720478351359) OR
(m between 1341720482545664 and 1341720503517183) OR
(m between 1341720671289344 and 1341720675483647) OR
(m between 1341720679677952 and 1341720683872255))) a
WHERE (x between 85670.0 and 85721.0) and
(y between 446416.0 and 446469.0));
```

Note that in the queries we basically replace the bounding box pre-selection in the regular flat-table with the Morton ranges pre-selection. We modified the Python script for the queries execution to automatically compute the Morton codes and generate the respective SQL query (see Appendix D.1).

6.3. Comparison with PostgresSQL flat PCDMS

We loaded the datasets 20M, 210M, 2201M, and 23090M with the new PCDMS and we compared it to the PostgreSQL flat table PCDMS used in the medium benchmark where a XY B-Tree index was used. In both cases the loading was parallelized using 16 processes. However, the most expensive parts were done with one single process, i.e. the indexing and, in the case of the new Morton-based PCDMS, also the clustering.

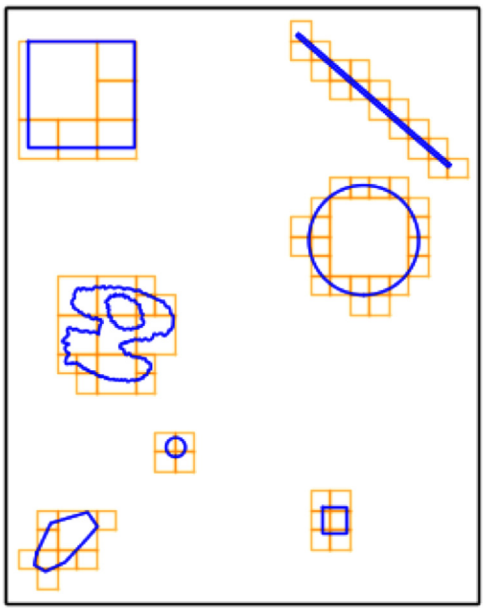

Fig. 7. Overlapping Quad-Tree cells for the first seven queries.

Table 7
Comparison of the time in DB loading and required storage requirement for the PostgreSQL flat PCDMS described in the medium benchmark (pf20M for example) and the new alternative Morton-indexed-clustered flat table (pf20Mk for example).

Approach	Time (s)				Size (MB)		Points	Points/s
	Total	Init.	Load	Close	Total	Index		
pf20M	44.07	2.00	13.86	28.21	1558.71	551.83	20,165,862	457,587
pf20Mk	45.90	1.66	17.56	26.68	1554.05	392.48	20,165,862	439,343
pf210M	771.63	2.09	71.44	698.10	16,249.24	5762.20	210,631,597	272,970
pf210Mk	813.50	23.94	113.39	676.17	16,200.72	4097.88	210,631,597	258,920
pf2201M	9897.37	1.15	722.91	9173.31	169,776.13	60,214.46	2,201,135,689	222,396
pf2201Mk	15,676.81	0.79	898.77	14,777.25	169,268.91	42,821.84	2,201,135,689	140,407
pf23090M	95,014.05	3.64	8745.90	86,264.51	1,780,963.91	631,663.71	23,090,482,455	243,022
pf23090Mk	172,045.23	1.08	10,487.29	161,556.86	1,775,643.19	449,210.67	23,090,482,455	134,212

Table 7 shows the loading details for both PCDMSs. The loading in the Morton-based PCDMS took longer because there is one more column to compute and load. It is also worth mentioning that the current computation of the Morton code is not optimal and could be done much faster. The indexing goes faster which is expected because it is a 1D indexing instead of 2D, but in the new PCDMS we also do a clustering operation which is expensive and does not scale well. The storage requirements are hardly affected. When data become larger, operations such as indexing and clustering in PostgreSQL become very expensive and would definitively benefit from the usage of multiple processes which is not currently possible in PostgreSQL, but is possible in other DBs like Oracle and MonetDB. One straightforward way of improving the loading would be using partitions which could also be done using the Morton code in order to have non-overlapping partitions.

We executed all the queries of the *medium-benchmark* in both PCDMSs except the nearest neighbor queries which have not been implemented yet in the new PCDMS. Contrary to the *medium-benchmark*, we ran the queries with one single process. We ran the queries with both cold and hot environments. ces of the *medium-benchmark* report in the project web site [25]. In Fig. 8 we show the comparison of the query response times for the two PCDMSs. Note that only queries 1, 4 and 7 are shown. The rest of queries can be found in the appendices of the *medium-benchmark* report in the project web site [25].

The queries in the Morton-based PCDMS are much faster and scale almost perfectly in both cold and hot environments thanks to the better spatial data organization and the smarter data accessing strategy. Note the large difference in query 7, a diagonal line. The queries in the initial Postgres flat PCDMS use the axis-aligned bounding box of the query region, which in some cases causes a lot of false positive points to be preselected. The Morton-based PCDMS uses the overlapping Quad-Tree cells, as shown in Fig. 7, which limits the number of false positive points.

As can be seen from the results above, creating the Morton code, indexing it, using it to cluster and rewriting the queries in order to use it together with Quad-Tree structure, considerably improves the query performance. One could think that clustering on the B-Tree on XY in the PostgreSQL flat PCDMS would also increase the performance, but this is not the same situation, because it still implies that for each X value, all the Y values have to be scanned (and skipped before arriving at needed neighbor X values). Clustering on XY B-tree might help a little bit (as it is effective to have data and index in same organization), but it remains a 'scan line' approach and therefore slow with larger datasets.

In fact, the proposed model with Morton or Hilbert curves could work for other types of N dimensional data as shown in [22,23]. The only requirement is that the significance of the dimensions is similar in the way that X and Y are similar in significance terms, in other

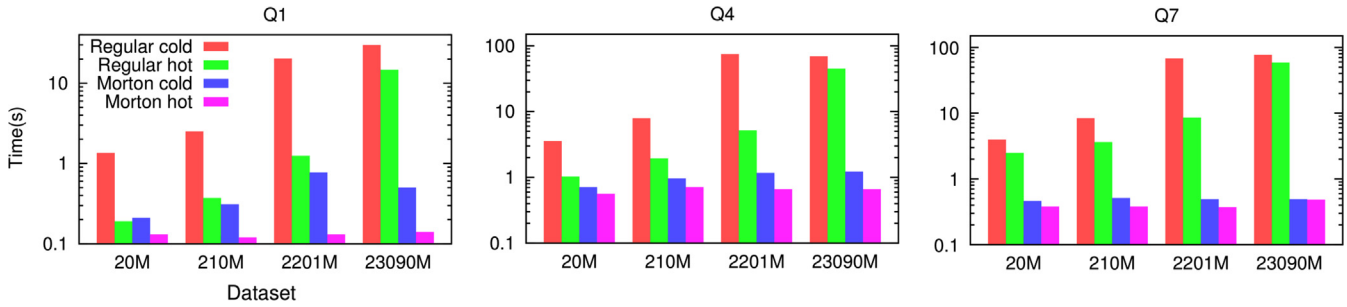

Fig. 8. Comparison of the response times of queries 1, 4 and 7 in the regular PostgreSQL flat table PCDMS and the new Morton-based PCDMS. The returned number of points is the same in both PCDMSs.

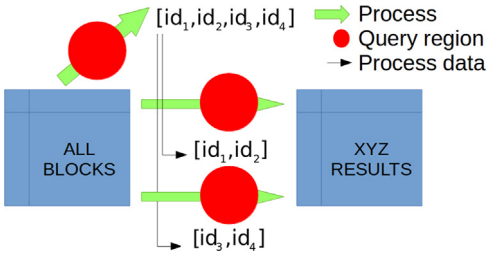

Fig. 9. Description diagram for the PQCB algorithm.

words that the queries ranges on the several dimensions have a similar length. If that is not the case, but the data distribution is known, we could also use a weighted Morton key and still use the model.

We are going to investigate the effect of the Morton code for other database solutions.

7. Parallel algorithms

In this section we present the two parallel querying algorithms we implemented to parallelize the query execution in some of the PCDMS (see Section 3.3). Both algorithms have been implemented in Python and in some cases they have provided speed up factors even higher than the used number of processes. The proposed algorithms could be further improved for example using pools of workers instead of fixed assignments. Another improvement could be combining both strategies into one algorithm. However, the results obtained with the current implementations already give an idea of the performance increase that in-core parallel algorithms would yield.

7.1. PQCB (Parallel Querying with Candidate Blocks) algorithm

This algorithm targets blocks-based solutions. First we find a list of identifiers of the candidate blocks, i.e. the ones that overlap the query region. This list is split in as many chunks as the number of processes. Each process takes a sublist and processes the related blocks extracting the points that overlap the query region. All the points are dumped in the final query result. Fig. 9 gives a graphical description of the steps of this algorithm.

7.2. PQGR (Parallel Querying with Gridded query Region) algorithm

This algorithm splits the query region into smaller pieces. We create a grid from the bounding box of the query region. Next, we create subqueries by intersecting each cell of the grid with the query region. We create as many pieces as the number of processes. Each process is in charge of one subquery, which is processed as a normal region. All the points are merged into the final query result. This algorithm may produce some duplicate points at the edges of the subquery regions. They can be removed using a "cleaning" step after all the processes are finished with a "select distinct" SQL query:

```
CREATE TABLE CLEANED_QUERY_RESULTS AS
  SELECT DISTINCT * FROM QUERY_RESULTS
```

Fig. 10 gives a graphical description of the steps of this algorithm.

Note that this algorithm can be used in both blocks-based and flat-table PCDMS solutions.

8. Vario LoD/data pyramid

Given the amount of data, it is not always needed and efficient to send all the points overlapping a query region from the server to the client. For instance, during visualization too much detail is not visible anyhow, but processing all the points takes a significant data

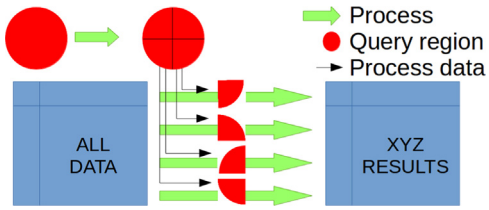

Fig. 10. Description diagram for the PQGR algorithm.

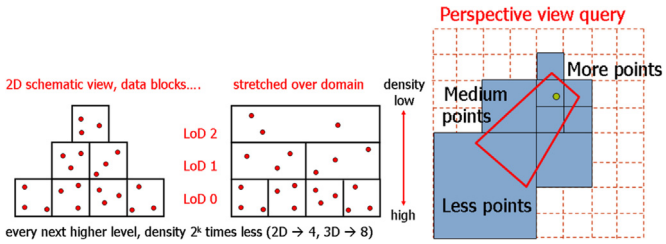

Fig. 11. Left: multiple LoDs with the same number of points in all the blocks. Right: perspective view query using blocks from multiple LoDs.

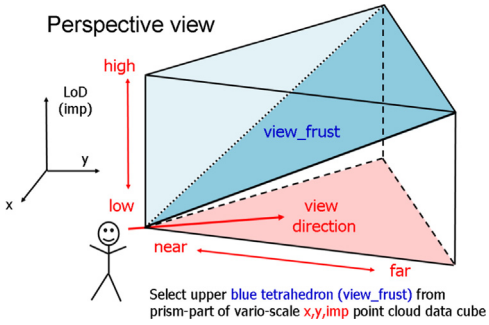

Fig. 12. Perspective view queries using vario-scale LoDs.

communication and processing time. Therefore, data pyramid solutions need to be implemented, allowing, for example, to select only a certain percentage of the points in a query region.

One alternative for the blocks storage model is to organize the blocks in a hierarchy with multiple levels of detail (LoD) as for example the data pyramid as available in Oracle's SDO_PC (see Usage notes of the SDO_PC_PKG.INIT method in the Oracle SDO_PC_PKG documentation [1]).

The idea is similar to the spatial location code [38], offering higher storage levels for more important objects. Typically, at each level the edge size of a block is doubled and in the case of 2D blocking, the four lower level blocks are covered by one upper level block. One forth of the points in the bottom level blocks can be repeated at the higher level block, or in case of no duplication, one fifth of the points of the bottom level blocks are moved to the higher level block. In both cases, all the blocks contain approximately the same amount of points; see the left image in Fig. 11.

This data pyramid approach can be used in perspective view queries as shown in the right image in Fig. 11. In this type of query we define the position of a viewer, the further away from the viewer the lesser points are selected by using the points from higher level blocks. The drawback of this multi-scale approach is that there are a discrete number of levels and the viewer may notice the difference in the point density between neighboring blocks at different levels. Therefore we investigate the possibility of vario-scale or continuous LoDs and related storage techniques.

In this work in progress we intend to apply techniques from the vario-scale research [40] to the point cloud data: add one dimension to the geometry, i.e. 2D data with vario-scale LoDs are represented by a 3D geometry. The added dimension is called importance and, in the visualization example, points with higher importance are more visible further from the viewer, while points with lower importance are only visible when close to the viewer. However, in this case there are not abrupt changes in the point density. In the implementation for point cloud data the proposed steps are (a) Compute the importance value of each point, which is comparable to the LoD level but not limited to discrete levels. An initial implementation could be to use a random distribution for the importance but this should be further refined after initial testing, probably by a 'data pyramid' distribution, that is many points at the bottom and few at the top. (b) Use this value as an additional importance dimension, which should have values from a range similar to x and y dimensions. Hence, the point cloud datasets become either x,y, importance with z and the rest of fields treated as attributes, or x,y,z, importance (in case z dimension also selective) with the rest of fields as attributes. (c) Sort and cluster, the 3D (or 4D) points using Hilbert or Morton codes. (d) Create blocks and index them with, for example, a 3D (or 4D) R-tree. This is more or less identical to the normal non-LoD point cloud blocking, but with one more dimension, the importance.

For the usage of point cloud data with vario-scale LoDs we need to implement the perspective view queries: the classic view frustum, i.e. the field of view of the viewer, gets the additional importance dimension, lower importance points are gradually ignored when getting further from the viewer. This new view frustum is a 3D polyhedron as shown in Fig. 12, or a 4D polychoron in the 4D case. In any case it can be selected quickly from the blocked and indexed vario-scale point cloud data either at block level or at exact overlap-level.

Visualizations using the vario-scale LoD approach do not have the density shocks that discrete-LoD-based data pyramids have. In a moving view position, the delta points sets can be efficiently selected and transferred from the server to the client: *New* selected points must (a) overlap with the new vario-scale view frustum and (b) do not overlap with the previous frustum as the visualization client already has these points. *Old* points must be dropped in the visualization client when (a) they overlap with the previous vario-scale view frustum and (b) they do not overlap with the new frustum.

Both discussed data pyramid solutions, i.e. multi-scale LoD and vario-scale LoD, are relevant for data storage and querying, but they are also relevant in the context of web-services. We investigate a specific web-services protocol for point cloud data supporting progressive transfers based on data pyramid solutions as point clouds datasets can be massive and the users might want to have an initial 'overview' of the data.

9. Conclusion

In this paper we presented the design, development, and execution of a benchmark for point cloud data. We investigated and benchmarked several point cloud data management systems (PCDMS) and the main results of their comparison were presented: Oracle (blocks and flat model), PostgreSQL (blocks and flat model), MonetDB (flat model), and a LAS file based solution with LAStools. Most results in this paper are based on the medium benchmark (23 billion AHN2 points), in addition a few platforms where tested with the full benchmark (640 billion AHN2 points). The experience gained with the smaller test datasets was used to tune the various platforms (block size, compression, etc.). During the several phases of this benchmark various issues were detected and reported to the developers involved in the different systems. Especially the Oracle Spatial and Graph and MonetDB developers responded promptly, resulting in several improved versions of the systems, which in turn resulted in the re-execution of the benchmark. In this paper, the results of benchmarking the latest versions of these systems (at the time of writing this paper) were presented.

9.1. Main findings and results

We tested a limited number of PCDMSs with a relatively small set of functionalities in two different hardware platforms. Thus, extracting generic conclusions is a delicate issue. The best PCDMS depends on the dataset size, the available hardware and software and the exact required functionalities. All the tested PCDMS have their pros and cons and most of them have been analyzed in this work. For example, if the query accuracy is at point level, which is the case for all the tested functionalities, the blocks-based PCDMSs need to unpack the blocks for the actual points incurring a significant overhead in the queries response times. LAStools with compressed LAZ files has a similar issue.

For the queries tested in this benchmark it is clear that the LAStools PCDMS offers the best performance for simple queries, i.e. rectangular and circular query regions. For other types of queries, the PCDMS based on databases using blocks are more suitable. For small datasets fitting in memory and simple queries, i.e. rectangles and circles, the flat-table PCDMS are more convenient than the blocks-based ones. However, they show important un-scaling issues. LAStools also presents un-scaling issues but they can be solved with the proposed hybrid system, using a database for the spatial extents of files. It is worth reminding that there are some important differences between the cold queries and the hot queries, being LAStools and the flat tables the ones that presents the most noticeable differences.

The Oracle Exadata with flat table model proved to be a very effective environment, both w.r.t. data loading and querying. Due to the massive parallel hardware engineered towards DBMS support, it was possible to load full AHN2 in less than 4:39 hours and storing the 12 TB data from LAS files into a 2.2 TB database (using 'query high' compression). In case of queries returning a very large number of points (from 10 million to over 1 billion), the system outperformed the other platforms.

Moreover, further point cloud data management improvements are presented: Morton or Hilbert codes for ordering the data, especially in the flat table model, parallel query execution algorithms, and vario-scale LoD data organizations.

9.2. Standardization

Implementing and executing the benchmark was not trivial as several different systems were used and a standard for point cloud data management is not yet available. We hope that the proposed benchmark will facilitate further developments in the area of the point clouds data management and standardization.

Recently more actors have become involved and provide partial solutions for point cloud data processing: data acquisition, storage, management, visualization, analysis, etc. The end-users, from industry, government or research, want interoperability: to be able to combine point cloud data from various sources, and to combine functionalities provided by different vendors, or even with their in-house developed specific purpose applications.

At least two closely related levels of standardization must be considered: (a) Database Structured Query Language (SQL) extension for point clouds, and (b) Web Point Cloud Services (WPCS) for progressive transfer of point clouds which should support progressive transfer based on multi-scale or vario-scale LoDs.

9.3. Future work

Several issues will be addressed in the future work:

- Compare the blocks incremental loading, i.e. file by file, with the blocks global loading for AHN2, which has optimally organized tiles into the files, and also for a less optimal case with a lot of overlap in the input files. After the most recent update in a beta version, Oracle now offers both approaches. Hence, this is the most obvious platform to analyze the effects.
- Complete the full benchmark of all the PCDMSs after resolving several loading bottlenecks and implementing functionalities. After that, the testing goes towards the scaled-up benchmark with 20 trillion points.

- In the Morton-based flat table PCDMS as presented in Section 6, do not store *X* and *Y* since the Morton key is a coded combination of both of them. Compare the change in the performance.
- Extend the executable benchmarks with multiple users loading and querying simultaneously and analyze the scalability when adding more hardware.
- Add to the benchmark different types of hardware and software solutions for point cloud data management, e.g. SpatialHadoop, LAStools/Esri format tools, cloud computing environments (MS Azure, Amazon EC2), etc.
- Explore true vario-scale LoDs implemented on blocked, higher dimensional, quadrant recursive, space filling curves at least including a 'continuous' importance value per point.
- Implement and test advanced functionalities outside of our current scope: surface and volume reconstruction, temporal difference queries, etc.
- Explore the use of higher dimensional point clouds: storing, structuring and indexing point clouds as 4D, 5D, 6D, *n*D points instead of 3D points with a number of attributes; investigate advantages and disadvantages of higher dimensional representations using both the blocks model and the flat table model.

Acknowledgments

The authors would like to thank the three anonymous reviewers (for Computers & Graphics), Milan Uitenthuis (from IntellinQ, The Netherlands), and Martijn Meijers (our TU Delft colleague) for their constructive remarks. Rod Thompson (from Queensland Government, Australia) is acknowledged for carefully proof-reading the draft of the revised version. Of course, any remaining error is the sole responsibility of the authors. The massive point cloud research is supported in part by the Netherlands eScience Center under project code: 027.012.101.

Appendix A. Functionality in the various categories

Based on the User Requirements document [5] and discussions with the project members we have compiled a list of functionalities in the various categories of a PCDMS.

A.1. Loading

The user should be able to specify:

- The input format: LAS, LAZ, ZLAS, CSV, etc.
- The dimensions on which the storage blocks are based.
- Data pyramid: discrete or continuous.
- Compression options: none, lossless or lossy.
- Spatial clustering within and between blocks: Morton, Hilbert, etc.
- Spatial indexing within and between blocks: Rtree, Quad-Tree, Oct-Tree, etc.
- Validation: format (for example, no attributes omitted), geometry or topological validation, outlier detection, etc.

A.2. Selections

The system should be able to provide methods to do point selections on:

- Simple 2D range/rectangle filters of various sizes.
- 2D polylines with different buffer sizes.
- 2D polygons.
- Spatial joints with other tables; e.g. overlapping points with polygons stored in secondary tables.
- Geographic information such as addresses, postal codes or other textual geographic names (for example gazetteers).
- Other attributes such as intensity, RGB, classification, etc.
- Space and time ranges.
- Combinations of multiple point clouds. For example, Laser and MBES or classified and unclassified datasets.

A.3. Simple analysis (not assuming 2D surfaces in 3D space)

Simple computations should be possible:

- Local or total density of points.
- k-nearest neighbors.
- Temporal differences.
- Attribute statistics: Minimum, maximum, average, median, etc.
- Cross profiles: intersections with vertical planes, 'nearby' points projections, etc.

A.4. Conversions (some assuming 2D surfaces in 3D space)

Data conversions should be supported:

- Uncompressed to compressed and vice versa.
- Change block size including optional data pyramid with multi-scale and vario-scale support.
- Coordinate transformations.
- Point clouds to contours and vice versa.
- Point clouds to TIN and vice versa.
- Point clouds to (height) raster/grid and vice versa.
- Point clouds to image and vice versa (e.g. oblique view on point cloud).
- Point clouds to higher dimensional objects/spaces.

A.5. Towards reconstruction

Some methods to ease object recognition or reconstruction should be considered:

- Computations of normal vectors of points.
- Computations of slope orientation or steepness.
- Flat plane detections, segmentation of points, addition of identifier attributes, etc.
- Detection of curved surfaces: cylinder, sphere patches, NURBS.
- Computation of buildings polygon using point clouds (difference inside-outside).

A.6. Complex analysis (some assume a 2D surfaces in 3D space)

Complex computations will present interesting challenges:

- Computations of areas of implied surface by point clouds.
- Computations of volumes below surfaces.
- Volume differences between design (3D polyhedral) surfaces and point could surfaces.
- Detections of break lines in point clouds surfaces.
- Hill shading relief (generate image).
- View shed analysis.

A.7. LoD use/access

Selections using the LoD concepts should be available:

- Multi-resolution/LoD selections. For example, select top 1%, or top 0.01% of points, or key points.
- Sort points on relevance/importance with streaming support.
- Selections based on perspective view and point cloud density: at viewer position M points per m^3 and (linear) reducing to 0 points per m^3 at distance d.
- Delta selections of perspective-view queries after moving to next view position: give additional points where densification is needed due to closer view range.

A.8. Updates

The system should support updating the data:

- Update point geometries, some small changes to many points.
- Update point geometries, larger changes to few points.
- Update attributes. For example while classifying points.
- Insert new data of the same type (0.1%, 1%, 5%, 10%, 25%)
- Delete points (0.1%, 1%, 5%, 10%, 25%)

Appendix B. Description of the different queries

Table B1 shows all the query geometries that are used in the benchmarks. The first seven queries are used in the *mini-benchmark*, the first 20 are used in the *medium-benchmark* and all of them are used in the *full-benchmark*. All the queries are executed as CTAS (Create Table As Select) except in the LAStools PCDMS were the points are dumped in a new LAS file. However, in future tests in the *full-benchmark* the regions 22, 23, 25, 26, 28, 29 and 30 will be aggregate queries, i.e. minimum, maximum or average. Note that the circular regions are

queries with the distance method in the flat table PCDMS and LAStools but with an approximated circular polygon in the blocks-based PCDMS.

In Table B1, *Pnts* is the number of points in the boundary of the query geometry, *Test* is the dataset name in which the query geometry is located.

Appendix C. Loading/preparation scripts

For the different PCDMS we have developed a set of Python scripts for the loading/preparation of the data. The steps followed by these scripts are detailed in the following sections. The input LAS files and the PCDMS data (after loading) are in different file systems. In the PCDMS using Oracle and PostgreSQL we use the tablespace *users* for the main data and the tablespace *indx* for the indexes. These two tablespaces are also using different file systems.

C.1. PostgreSQL flat

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

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

Parallel converting and loading of files (Command line, executed in parallel using Python multiprocessing module):

```
las2pg -s [input LAS file] -stdout -parse xyz | psql [connection parameters]-c "copy ahn_flat from stdin with
  binary"
```

Create the B-Tree index (SQL):

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

Note that the index creation can not be done in parallel in PostgreSQL.

C.2. PostgreSQL blocks

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

```
CREATE EXTENSION postgis;
CREATE EXTENSION pointcloud;
```

Table B1
Description of the different queries.

ID	Key	Pnts	Test	Area (km^2)	Description
1	S_RCT	5	20M	0.0027	Small axis-aligned rectangle
2	M_RCT	5	20M	0.0495	Medium axis-aligned rectangle
3	S_CRC	97	20M	0.0013	Small circle, radius 20 m.
4	M_CRC	379	20M	0.0415	Medium circle, radius 115 m
5	S_SIM	9	20M	0.0088	Small, simple polygon
6	M_COM_o	792	20M	0.0252	Medium, complex polygon, 1 hole
7	M_DG_RCT	5	20M	0.0027	Medium, narrow, diagonal rectangular area
8	L_COM_os	89	210M	0.1341	Large, complex polygon, 2 holes
9	S_L_BUF	94	210M	0.0213	Small polygon (10 m buffer around line of 11 pts)
10	S_RCT_UZ	5	210M	0.0021	Small axis-aligned rectangle, cut in maximum elevation
11	S_RCT_LZ	5	210M	0.0051	Small axis-aligned rectangle, cut in minimum elevation
12	L_RCT_LZ	5	210M	0.1419	Large axis-aligned rectangle, cut in minimum elevation
13	L_L_BUF	237	2201M	0.0475	Large polygon (1 m buffer around line of 61 pts)
14	L_DG_L_BUF	39	2201M	0.0499	Large polygon (2 m buffer around diagonal line of 8 pts)
15	L_RCT	5	23090M	0.2342	Large axis-aligned rectangle
16	L_RCT_N	5	23090M	0.1366	Large axis-aligned rectangle in empty area
17	L_CRC	93	23090M	0.1256	Large circle
18	NN_1000	1	23090M	0.0000	Point for NN query, 1000 nearest points
19	NN_5000	1	23090M	0.0000	Point for NN query, 5000 nearest points
20	NN_1000_w	1	23090M	0.0000	Point in water for NN query, 1000 nearest points
21	L_NARROW_RECT	5	639478M	0.0236	Large narrow axis-aligned rectangle, points
22	L_NARROW_DIAG_RECT	5	639478M	0.6399	Large narrow diagonal rectangle, min(Z)
23	XL_NARROW_DIAG_RECT	5	639478M	42.5573	Very large narrow diagonal rectangle, max (Z)
24	L_NARROW_DIAG_RECT_2	5	639478M	0.1208	Large narrow diagonal rectangle, points
25	PROV_DIAG_RECT	5	639478M	3022.0427	Provincial size diagonal rectangle, min(Z)
26	MUNI_RECT	5	639478M	236.7744	Municipality size diagonal rectangle, max(Z)
27	STREET_DIAG_RECT	5	639478M	0.0016	Street size diagonal rectangle, points
28	VAALS	1565	639478M	23.8972	Municipality Vaals, avg(Z)
29	MONTFERLAND	1565	639478M	106.6520	Municipality Montferland, avg(Z)
30	WESTERVELD	1569	639478M	282.7473	Municipality Westerveld, avg(Z)

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

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

```
INSERT INTO pointcloud_formats (pcid, srid, schema)
VALUES (1, [SRID],'
  <?xml version="1.0" encoding="UTF-8"?>
  <pc:PointCloudSchema xmlns:pc=
  "http://pointcloud.org/schemas/PC/1.1" xmlns:xsi=
  "http://www.w3.org/2001/XMLSchema-instance">
    <pc:dimension>
      <pc:position>1</pc:position>
      <pc:size>4</pc:size>
      <pc:description>X coordinate</pc:description>
      <pc:name>X</pc:name>
      <pc:interpretation>int32_t</pc:interpretation>
      <pc:scale>0.01</pc:scale>
      <pc:offset>0</pc:offset>
    </pc:dimension>
    <pc:dimension>
      <pc:position>2</pc:position>
      <pc:size>4</pc:size>
      <pc:description>Y coordinate</pc:description>
      <pc:name>Y</pc:name>
      <pc:interpretation>int32_t</pc:interpretation>
      <pc:scale>0.01</pc:scale>
      <pc:offset>0</pc:offset>
    </pc:dimension>
    <pc:dimension>
      <pc:position>3</pc:position>
      <pc:size>4</pc:size>
      <pc:description>Zcoordinate</pc:description>
      <pc:name>Z</pc:name>
      <pc:interpretation>int32_t</pc:interpretation>
      <pc:scale>0.01</pc:scale>
      <pc:offset>0</pc:offset>
    </pc:dimension>
    <pc:metadata>
      <Metadata name="compression">
       dimensional
      </Metadata>
    </pc:metadata>
</pc:PointCloudSchema>');
```

Note that we use dimensional compression. Once the format has been added we need to use the PDAL tool to load the data from the input file (Command-line, executed in parallel using Python multiprocessing module):

```
pdal pipeline [xmlFile]
```

Where the XML file contains:

```
<?xml version="1.0" encoding="utf-8"?>
<Pipeline version="1.0">
  <Writer type="drivers.pgpointcloud.writer">
  <Option name="connection"> host='[DB host]'
 dbname='[DB name]' password='[password]'
 user='[DB user]'</Option>
<Option name="table">blocks</Option>
<Option name="srid">[SRID]</Option>
<Option name="overwrite">false</Option>
<Option name="pcid">1</Option>
<Filter type="filters.chipper">
```

```
  <Option name="capacity">3000</Option>
  <Filter type="filters.cache">
    <Option name="max_cache_blocks">1</Option>
    <Filter type="filters.selector">
      <Option name="keep">
        <Options>
          <Option name="dimension">X</Option>
          <Option name="dimension">Y</Option>
          <Option name="dimension">Z</Option>
        </Options>
      </Option>
      <Option name="overwrite_existing_dimensions">
       false
      </Option>
      <Reader type="drivers.las.reader">
        <Option name="filename">[input LAS file]
        </Option>
        <Option name="spatialreference">EPSG:28992
        </Option>
      </Reader>
      </Filter>
    </Filter>
  </Filter>
  </Writer>
 </Pipeline>
```

Note that the *pcid* is the *formatID*, i.e. 1. Note that we use a block size of 3000 and that we only load *X*, *Y* and *Z* coordinates by adding a filter. Create a PostGIS GIST index on the blocks to ease the querying (SQL):

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

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

```
VACUUM FULL ANALYZE blocks;
```

C.3. Oracle flat

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

```
CREATE TABLE EXT_AHN_FLAT (VAL_D1 NUMBER,
  VAL_D2 NUMBER, VAL_D3 NUMBER)
ORGANIZATION EXTERNAL(
  TYPE ORACLE_LOADER
  DEFAULT DIRECTORY LAS_DIR
  ACCESS PARAMETERS (
    RECORDS DELIMITED BY NEWLINE
    PREPROCESSOR EXE_DIR:'HILBERT_PREP.SH'
    BADFILE LOG_DIR:'HILBERT_PREP_%P.BAD'
    LOGFILE LOG_DIR:'HILBERT_PREP_%P.LOG'
    FIELDS TERMINATED BY ',')
LOCATION ('*.LAS'))
PARALLEL16 REJECT LIMIT 0;
```

The preprocessor file uses the class *Las2SqlLdr* in the package *oracle.spatial.util* of the JAR file sdoutl.jar. The *LAS_DIR* directory must have been previously created within the Oracle environment and it must point to the folder with the input data. Directories *EXE_DIR* and *LOG_DIR* must also be created. We create the flat table by selecting from the external table in parallel using the available cores (SQL).

```
CREATE TABLE AHN_FLAT TABLESPACE USERS PCTFREE 0
NOLOGGING PARALLEL16 AS
  SELECT VAL_D1,VAL_D2,VAL_D3 FROM EXT_AHN_FLAT;
```

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

```
CREATE INDEX AHN_FLAT_IDX ON AHN_FLAT (VAL_D1,VAL_D2)
TABLESPACE INDX PCTFREE 0 NOLOGGING PARALLEL16;
```

C.4. Oracle blocks

In this case we need a staging table that will contain all the points from all the files. For this we also use an external table which also contains the Hilbert value computed by the preprocessor (SQL).

```
CREATE TABLE EXT_AHN_STAGING (VAL_D1 BINARY_DOUBLE,
  VAL_D2 BINARY_DOUBLE, VAL_D3 BINARY_DOUBLE,
  DNUMBER)
ORGANIZATION EXTERNAL(
  TYPE ORACLE_LOADER
  DEFAULT DIRECTORY LAS_DIR
  ACCESS PARAMETERS (
    RECORDS DELIMITED BY NEWLINE
    PREPROCESSOR EXE_DIR:'HILBERT_PREP.SH'
    BADFILE LOG_DIR:'HILBERT_PREP_%P.BAD'
    LOGFILE LOG_DIR:'HILBERT_PREP_%P.LOG'
    FIELDS TERMINATED BY ','
    LOCATION ('*.LAS'))
PARALLEL16 REJECT LIMIT 0;
```

This external table is similar to the one used in the Oracle flat with the addition of the Hilbert value. The directories *LAS_DIR*, *EXE_DIR* and *LOG_DIR* must be created within the Oracle environment. We create the table that will contain the blocks and the table with the meta-data information of the point cloud (SQL).

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

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

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

```
CREATE TABLE AHN_STAGING (VAL_D1,VAL_D2,VAL_D3,D,
  CONSTRAINT AHN_STAGING_PK PRIMARY KEY (D))
ORGANIZATION INDEX TABLESPACE USERS PCTFREE 0
NOLOGGING PARALLEL16 AS
  SELECT VAL_D1,VAL_D2,VAL_D3,
    D+(ROWNUM*0.0000000001) D FROM EXT_AHN_STAGING;
```

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

```
DECLARE
  PTCLD SDO_PC;
  PTN_PARAMS VARCHAR2(80) :='BLK_CAPACITY=10000';
  EXTENT SDO_GEOMETRY :=SDO_GEOMETRY(2003,28992,
    NULL, SDO_ELEM_INFO_ARRAY(1,1003,3),
    SDO_ORDINATE_ARRAY(
      [min. X],[min. Y],[max. X],[max. Y]));
    OTHER_ATTRS XMLTYPE :=XMLTYPE('
      <OPC:SDOPCOBJECTMETADATA
      XMLNS:OPC="HTTP://XMLNS.ORACLE.COM/SPATIAL/
        VIS3D/2011/SDOVIS3D.XSD"
      XMLNS:XSI="HTTP://WWW.W3.ORG/2001/
        XMLSCHEMA-INSTANCE"
    BLOCKINGMETHOD="HILBERT R-TREE">
    </OPC:SDOPCOBJECTMETADATA>'); BEGIN
  PTCLD :=SDO_PC_PKG.INIT('AHN_BASE','PC','AHN_BLCK',
    PTN_PARAMS,EXTENT,0.0001,3,NULL,NULL,OTHER_ATTRS);
  INSERT INTO AHN_BASE VALUES (PTCLD);
  COMMIT;
```

```
  SDO_PC_PKG.CREATE_PC (PTCLD, 'AHN_STAGING', NULL);
END;
```

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

```
UPDATE AHN_BLCK B SET B.BLK_EXTENT.SDO_SRID=28992;
DROP TABLE AHN_STAGING;
INSERT INTO USER_SDO_GEOM_METADATA VALUES
  ('AHN_BLCK','BLK_EXTENT',
  SDO_DIM_ARRAY(
  SDO_DIM_ELEMENT('X',[min. X],[max. X],0.0001),
  SDO_DIM_ELEMENT('Y',[min. Y],[max. Y],0.0001)),
  28992);
CREATE INDEX AHN_BLCK_SIDX ON AHN_BLCK (BLK_EXTENT)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX PARAMETERS (
    'TABLESPACE=INDX WORK_TABLESPACE=PCWORK
    LAYER_GTYPE=POLYGON SDO_INDX_DIMS=2
    SDO_RTR_PCTFREE=0')
  PARALLEL16;
```

C.5. MonetDB flat

Create the flat table (SQL):

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

We execute the MonetDB binary converter that converts LAS and LAZ files into the internal binary format used for columnar storage in MonetDB (command line).

```
las2col -i [input LAS file 1] -i [input LAS file 2]
-i ...0_tempFile -parse xyz
```

The output files are written in a different file system than the input files (to maximize IO). We make MonetDB aware of the data (SQL):

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

The files still need to be read by MonetDB and they can be quite large so this process can take a while. In the case of a large number of input files (more than few hundred) we recommend splitting the loading in chunks. Since the Imprints index in MonetDB is automatically created in the first query, we force the Imprints creation (on *X* and *Y*) by doing a rectangle query in the center of the extent (even though a query between 0 and 1 would also trigger its creation) (SQL)

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

C.6. LAStools

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

```
lassort.exe -i [input LASfile] -o [output LASfile]
lasindex -i [output LASfile]
```

The sorted files are written in a different file system than the input files. We create a DB with the PostGIS extension and a table to hold the extent of the several files (SQL).

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

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

```
lasinfo [output LAS file] -nc -nv -nco


INSERT INTO lasindex (id,filepath,num,scalex,scaley,
  scalez,offsetx,offsety,offsetz,geom) VALUES
  (0, '[output LAS file]', [num. points],
  [scale X], [scale Y], [scale Z],
  [offset X], [offset Y], [offset Z],
  ST_MakeEnvelope(    [min. X],[min. Y],[max. X],[max. Y], [SRID]));
```

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

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

Appendix D. Querying scripts

We have developed a set of Python scripts to execute the several queries. In the database solutions the query regions, provided as Well-known text (WKT), are loaded in a table called QUERY_POLYGONS. In the following sections we present some examples of the queries executed for the different PCDMS. In some queries the extraction of the bounding box can be done in the same query statement while in some other it has to be done before. In such cases the time spent in retrieving the bounding box is also taken into account in the results shown in this paper. In the flat-table-based PCDMS the nearest neighborquery has two phases: (a) We use a pre-selection of a sufficient number of points, the preselected points are within a manually estimated maximum distance to the query point. (b) We sort these points on distance to the query point and report the nearest *N* points.

D.1. PostgreSQL flat

Rectangle (example query 1):

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

Circle (example query 3):

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

Other regions (example query 5):

```
CREATE TABLE query_results_5 AS (SELECT * FROM   (SELECT * FROM ahn_flat WHERE     (x between 85069.0 and
   85201.0) and     (y between 446334.0 and 446458.0)) a,   query_polygons   WHERE id=5 and _ST_Contains(geom,
    st_setSRID(st_makepoint(x,y),28992)));
```

Nearest neighbor approximation. This a manual estimation of a surrounding area with sufficient points (example query 18):

```
CREATE TABLE query_results_18 AS (
SELECT * FROM
(SELECT * FROM ahn_flat WHERE (x between 67195.73-10 and 67195.73+10) and (y between 433973.27-10 and
   433973.27+10)) a
ORDERBY (x-67195.73)^2+(y-433973.27)^2
LIMIT 1000);
```

Other regions with PQGR algorithm (example query 5 with 8 processes):

```
CREATE TABLE query_grid_5 AS
  WITH A AS (    SELECT geom, st_xmin(geom) as minx,
      st_xmax(geom) as maxx, st_ymin(geom) as miny,
      st_ymax(geom) as maxy
      FROM query_polygons WHERE id=5)
  SELECT B.row, B.col, ST_Intersection(A.geom,
    ST_SetSRID(B.geom, '28992')) as geom FROM A,
      (SELECT F.row, F.col, F.geom FROM A,
        ST_CreateFishnet(2, 4, (maxx-minx)/4,
        (maxy-miny)/2, minx, miny) F) AS B ;
```

```
CREATE INDEX qg_5_rowcol ON query_grid_5 (row, col );

CREATE TABLE query_results_5
  (x NUMERIC,y NUMERIC, z NUMERIC);
- The 8 processes have to insert the
- data in parallel (example for process 0)
- We use this query to extract the
- bounding box of the piece of region

SELECT st_xmin(geom), st_xmax(geom),
  st_ymin(geom), st_ymax(geom)
  FROM query_grid_5 WHERE row=1 AND col=1;

INSERT INTO query_results_5 (x,y,z)
  SELECT x,y,z FROM (
    SELECT x,y,z FROM ahn_flat WHERE
      (x between 85102.0 and 85135.0) and      (y between 446338.63 and 446396.0))     a, query_grid_5
  WHERE row=1 AND col=1 AND
    _ST_Contains(geom,st_setSRID(
    st_makepoint(x,y),'28992'));
```

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

D.2. PostgreSQL blocks

Any region (example query 1):

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

Any region with PQGR algorithm (example query 1 with 8 processes):

```
CREATE TABLE query_grid_1 AS
  WITH A AS (
    SELECT geom, st_xmin(geom) as minx,
      st_xmax(geom) as maxx, st_ymin(geom) as miny,
      st_ymax(geom) as maxy FROM query_polygons
      WHERE id=1)
  SELECT B.row, B.col, ST_Intersection(A.geom,
    ST_SetSRID(B.geom, '28992')) as geom
    FROM A,
    (SELECT F.row, F.col, F.geom FROM A,
      ST_CreateFishnet(2, 4, (maxx-minx)/4,
      (maxy-miny)/2, minx, miny) F) AS B ;

CREATE INDEX qg_1_rowcol ON query_grid_1 (row, col);

CREATE TABLE query_results_1
  (x NUMERIC,y NUMERIC, z NUMERIC);

- The 8 processes have to insert the
- data in parallel (example for process 0)

INSERT INTO query_results_1 (x,y,z )
  SELECT PC_Get(qpoint, 'x') as x,
    PC_Get(qpoint, 'y') as y,
    PC_Get(qpoint, 'z') as z FROM (
    SELECT PC_explode(pc_intersection(pa,geom))
      as qpoint FROM patches, query_grid_1
      WHERE pc_intersects(pa,geom)
      AND row=1 AND col=1) as D;
```

Nearest neighbor approximation. In this estimation we get a certain number of blocks closest to the desired point and select the closest points. The number of blocks is estimated based on the block size and the requested number of points (example query 18):

```
CREATE TABLE query_results_18 AS (
  SELECT PC_Get(qpoint, 'x') as x,
    PC_Get(qpoint, 'y') as y,
    PC_Get(qpoint, 'z') as z FROM
      (SELECT PC_explode(pa) as qpoint FROM
        (SELECT pa FROM patches ORDER BY
        geometry(pa) < >
         '010100002040710000E17A14
        AEBB67F04048E17A14D57C1A41'
        LIMIT 9) as A) as B
  ORDERBY ((PC_Get(qpoint,'x')-67195.72)^2+
    (PC_Get(qpoint,'y')-433973.27)^2)
LIMIT 1000);
```

The string represent the Python geometry object related to the point.

D.3. Oracle flat

Rectangle (example query 1):

```
DECLARE
  BBOX SDO_GEOMETRY;
BEGIN
  SELECT SDO_GEOM_MBR (GEOM) INTO BBOX
    FROM QUERY_POLYGONS WHERE ID=1;
  EXECUTE IMMEDIATE 'CREATE TABLE
    QUERY_RESULTS_2 AS
      SELECT VAL_D1, VAL_D2, VAL_D3 FROM AHN_FLAT
        WHERE (VAL_D1BETWEEN'||
          TO_CHAR(BBOX.SDO_ORDINATES(1))||' AND
        '||TO_CHAR(BBOX.SDO_ORDINATES(3))||')
        AND (VAL_D2BETWEEN'||
        TO_CHAR(BBOX.SDO_ORDINATES(2))||' AND
        '||TO_CHAR(BBOX.SDO_ORDINATES(4))||')';
END;
```

Circle (example query 3):

```
CREATE TABLE QUERY_RESULTS_3 (VAL_D1 NUMBER,
  VAL_D2 NUMBER,VAL_D3 NUMBER);
  INSERT INTO QUERY_RESULTS_3
    SELECT * FROM
      (SELECT * FROM AHN_FLAT
      WHERE (VAL_D1 BETWEEN 85345 AND 85385)
        AND (VAL_D2 BETWEEN 446574 AND 446614))B
      WHERE POWER((VAL_D1 - 85365.0),2) +
        POWER((VAL_D2 - 446594.0),2)
         <=POWER(20.0,2);
```

Other regions (example query 5):

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

```
      AND (VAL_D2 BETWEEN '||
        TO_CHAR(BBOX.SDO_ORDINATES(2))||' AND
        '||TO_CHAR(BBOX.SDO_ORDINATES(4))||')
      ),(SELECT GEOM FROM QUERY_POLYGONS
      WHERE ID=5),0.0001,NULL))';
END;
```

Nearest neighbor approximation (example query 18):

```
CREATE TABLE query_results_18(x DOUBLE,
  y DOUBLE, z DOUBLE);
INSERT INTO query_results_18
  SELECT * FROM
    (SELECT * FROM ahn_flat WHERE
    (x between 67195.73-10 and 67195.73+10) and
    (y between 433973.27-10 and 433973.27+10)) a
  ORDER BY POWER(x - 67195.73,2) +
    POWER(y - 433973.27,2) LIMIT 1000;
```

D.4. Oracle blocks

Any region (example query 1):

```
CREATE TABLE QUERY_RESULTS_1
  (VAL_D1 NUMBER,VAL_D2 NUMBER,VAL_D3 NUMBER);
INSERT INTO QUERY_RESULTS_1
  SELECT PNT.X, PNT.Y, PNT.Z FROM
    TABLE (SDO_PC_PKG.CLIP_PC(
      (SELECT PC FROM AHN_BASE),
      (SELECT GEOM FROM QUERY_POLYGONS WHERE ID=1),
      NULL,NULL,NULL,NULL)) PCBLOB,
    TABLE (SDO_UTIL.GETVERTICES(
      SDO_PC_PKG.TO_GEOMETRY(PCBLOB.POINTS,
      PCBLOB.NUM_POINTS,3,NULL))) PNT;
```

Any region with PQGR algorithm (example query 1 with 8 processes):

```
CREATE TABLE QUERY_RESULTS_1
  (VAL_D1 NUMBER,VAL_D2 NUMBER,VAL_D3 NUMBER);
CREATE TABLE QUERY_GRID_1 AS
  WITH A AS (
    SELECT GEOM FROM QUERY_POLYGONS WHERE ID=1
  )
  SELECT T.ID, SDO_GEOM.SDO_INTERSECTION(
    A.GEOM, T.GEOMETRY, 0.0001) AS GEOM FROM
    TABLE(SELECT SDO_SAM.TILED_BINS(
      SDO_GEOM.SDO_MIN_MBR_ORDINATE(A.GEOM,1),
      SDO_GEOM.SDO_MAX_MBR_ORDINATE(A.GEOM,1),
      SDO_GEOM.SDO_MIN_MBR_ORDINATE(A.GEOM,2),
      SDO_GEOM.SDO_MAX_MBR_ORDINATE(A.GEOM,2),
      NULL, 28992, 4 - 1, 2 - 1
) FROM A) T, A;

CREATE INDEX QG_1_id_idx ON QUERY_GRID_1(ID);

- The 8 processes have to insert the
- data in parallel (example for process 0)

INSERT INTO QUERY_RESULTS_1
  SELECT PNT.X, PNT.Y, PNT.Z FROM
    TABLE (SDO_PC_PKG.CLIP_PC(
      (SELECT PC FROM AHN_BASE),
      (SELECT GEOM FROM QUERY_GRID_1 WHERE ID=0),
      NULL,NULL,NULL,NULL)) PCBLOB,
    TABLE (SDO_UTIL.GETVERTICES(
      SDO_PC_PKG.TO_GEOMETRY(PCBLOB.POINTS,
      PCBLOB.NUM_POINTS,3,NULL))) PNT;
```

D.5. MonetDB flat

Rectangle (example query 1):

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

Circle (example query 3):

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

Other regions (example query 5):

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

Nearest neighbor approximation (example query 18):

```
CREATE TABLE query_results_18
  (x DOUBLE, y DOUBLE, z DOUBLE);
INSERT INTO query_results_18
  SELECT * FROM
    (SELECT * FROM ahn_flat WHERE
      (x between 67195.73-10 and 67195.73+10) and
      (y between 433973.27-10 and 433973.27+10)) a
    ORDER BY POWER(x - 67195.73,2) +
    POWER(y - 433973.27,2) LIMIT 1000;
```

D.6. LAStools

Rectangle (example query 1):

```
psql lt23090M -U oscar -h localhost
-p 5434 -t -A -c "SELECT filepath FROM
lasindex,query_polygons
WHERE ST_Intersects(query_polygons.geom,
lasindex.geom ) and query_polygons.id=1"
 > input1.list

lasmerge -lof input1.list
-inside 85670.0 446416.0 85721.0 446469.0
-o output1.las
```

Circle (example query 3):

```
psql lt23090M -U oscar -h localhost
-p 5434 -t -A -c "SELECT filepath FROM
lasindex,query_polygons WHERE ST_Intersects(
```

```
query_polygons.geom, lasindex.geom ) and
query_polygons.id=3" > input3.list

lasmerge -lof input3.list -inside_circle 85365.0
446594.0 20.0 -o output3.las
```

Other regions (example query 5):

```
psql lt23090M -U oscar -h localhost
-p 5434 -t -A -c "SELECT filepath FROM
lasindex,query_polygons
WHERE ST_Intersects(query_polygons.geom,
lasindex.geom ) and query_polygons.id=5"
 > input5.list

lasclip.exe -lof input5.list
-poly query5.shp -o output5.las -merged
```

In this case we use *lasclip* instead of *lasmerge*. The polygon is given with a Shapefile, so before the *lasclip* call we have to convert the PostgreSQL polygon into a Shapefile. This conversion is not counted in the times. We do this with:

```
pgsql2shp -f query5.shp -h localhost
-p 5434 -u oscar -P oscar lt23090M
"SELECT ST_SetSRID(geom, 28992)
FROM query_polygons WHERE id=5;"
```

References

[1] Oracle Database Online Documentation 12c Release 1 (12.1): Spatial and Graph Developer's Guide/SDO_PC_PKG Package (Point Clouds), ⟨https://docs.oracle.com/database/121/SPATL/sdo_pc_pkg_ref.htm⟩; 2014.
[2] Ramsey P. A PostgreSQL extension for storing point cloud (LIDAR) data, ⟨https://github.com/pramsey/pointcloud⟩; 2014.
[3] Oracle Database Online Documentation 12c Release 1 (12.1): Spatial and Graph GeoRaster Developer's Guide, ⟨https://docs.oracle.com/database/121/GEORS/geor_intro.htm⟩; 2014.
[4] PostgreSQL: Raster reference, ⟨http://postgis.net/docs/RT_reference.html⟩; 2014.
[5] Suijker PM, Alkemade I, Kodde MP, Nonhebel AE. User requirements Massive Point Clouds for eSciences (WP1), Technical Report, Delft University of Technology, ⟨http://repository.tudelft.nl/view/ir/uuid%3A351e0d1e-f473-4651-bf15-8f9b29b7b800/⟩; 2014.
[6] Martinez-Rubi O, Kersten M, Goncalves R, Ivanova M. A column-store meets the point clouds. FOSS4GEurope; 2014.
[7] rapidlasso GmbH, ⟨http://rapidlasso.com/⟩; 2014.
[8] rapidlasso GmbH LASzip - free and lossless LiDAR compression, ⟨http://www.laszip.org/⟩; 2014.
[9] Esri ArgGIS LAS Optimizer, ⟨http://www.arcgis.com⟩; 2014.
[10] Actueel Hoogtebestand Nederland (AHN), ⟨http://www.ahn.nl/⟩; 2014.
[11] Herring JR. OpenGIS simple features specification for SQL, Technical Report OGC 06-104r4, version 1.2.1; Open Geospatial Consortium, Inc.; 2010.
[12] Meissl S. OGC GML Application Schema - Coverages - GeoTIFF Coverage Encoding Profile, Technical Report OGC 12-100r1, version 1.0; Open Geospatial Consortium, Inc.; 2014.
[13] ISPRS. LAS 1.4 format specification, Technical Report, The American Society for Photogrammetry & Remote Sensing, ⟨http://www.asprs.org/a/society/committees/standards/LAS_1_4_r13.pdf⟩; 2011.
[14] Elseberg J, Borrmann D, Nüchter A. One billion points in the cloud an octree for efficient processing of 3D laser scans. ISPRS J Photogramm Remote Sens 2013;76:76–88.
[15] Schön B, Mosa ASM, Laefer DF, Bertolotto M. Octree-based indexing for 3D point clouds within an Oracle Spatial DBMS. Comput Geosci 2013;51:430–8.
[16] Wenzel K, Rothermel M, Fritsch D, Haala N. An out-of-core octree for massive point cloud processing. In: Workshop on processing large geospatial data Cardiff, UK, July 8th, 2014, ⟨http://rs.tudelft.nl/~rlindenbergh/workshop/WenzelIQmulus.pdf⟩; 2014.
[17] Han S, Kim S, Jung JH, Kim C, Yu K, Heo J. Development of a hashing-based data structure for the fast retrieval of 3D terrestrial laser scanned data. Comput Geosci 2012;39:1–10.
[18] Otepka J, Ghuffar S, Waldhauser C, Hochreiter R, Pfeifer N. Georeferenced point clouds: a survey of features and point cloud management. ISPRS Int J GeoInf 2013;2(4):1038–65.
[19] Boehm J. File-centric Organization of large LiDAR Point Clouds in a Big Data context. In: Workshop on processing large geospatial data Cardiff, UK, July 8th, 2014, ⟨http://rs.tudelft.nl/~rlindenbergh/workshop/BoehmIQmulus.pdf⟩; 2014.
[20] Ott M. Towards storing Point Clouds in PostgreSQL, Technical Report, HSR Hochschule für Technik Rapperswil, Seminar Database Systems, Master of Science in Engineering; 2012.
[21] Wijga-Hoefsloot M. Point clouds in a database—data management within an engineering company [Master's thesis]. TU Delft: MSc Geomatics; 2012.
[22] Lawder JK, King P. Querying multi-dimensional data indexed using the Hilbert space-filling curve. SIGMOD Rec. 2001;30:2001.
[23] Lawder J. The application of space-filling curves to the storage and retrieval of multi-dimensional data; 2000.
[24] Terry J, Stantic B, Terenziani P, Sattar A. Variable granularity space filling curve for indexing multidimensional data. In: Proceedings of the 15th international conference on advances in databases and information systems. ADBIS'11; Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-642-23736-2; 2011, p. 111–24, URL: ⟨http://dl.acm.org/citation.cfm?id=2041746.2041759⟩.
[25] Massive point clouds for eSciences, ⟨http://www.pointclouds.nl/⟩; 2014.
[26] Oracle Exadata Database Machine X4-2, ⟨https://www.oracle.com/engineered-systems/exadata/database-machine-x4-2/index.html⟩; 2014.
[27] PostgreSQL 9.3.5 Documentation: Chapter 55. GiST Indexes, ⟨http://www.postgresql.org/docs/9.3/static/gist.html⟩; 2014.
[28] PostgreSQL 9.3.5 Documentation: Chapter 58. Database Physical Storag/58.2. TOAST, ⟨http://www.postgresql.org/docs/9.3/static/storage-toast.html⟩; 2014.
[29] PDAL - Point Data Abstraction Library, ⟨http://www.pdal.io/⟩; 2014.
[30] Oracle Database Online Documentation 12c Release 1 (12.1): Database Administrator's Guide/Managing Index-Organized Tables. ⟨https://docs.oracle.com/cd/E24628_01/server.121/e41484/tables.htmADMIN01506⟩; 2014.
[31] libLAS - LAS 1.0/1.1/1.2 ASPRS LiDAR data translation toolset, ⟨http://www.liblas.org/⟩; 2014.
[32] Sagan H. Hilbert's space-filling curve. In: Space-filling curves. New York: Springer. ISBN 978-0-387-94265-0; 1994. p. 9–30, ⟨http://dx.doi.org/10.1007/978-1-4612-0871-6_2⟩.
[33] Sidirourgos L, Kersten M. Column imprints: a secondary index structure. SIGMOD; 2013.
[34] Terrasolid Software for LiDAR processing, ⟨http://www.terrasolid.com/⟩; 2014.
[35] Esri. ESRI Shapefile technical description, Technical Report, URL: ⟨http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf⟩; 1998.

[36] Sagan H. Peano's space-filling curve. In: Space-filling curves. New York: Springer. ISBN 978-0-387-94265-0; 1994, p. 31–47. ⟨http://dx.doi.org/10.1007/978-1-4612-0871-6_3⟩.
[37] Oracle Database Online Documentation 12c Release 1 (12.1): Database VLDB and Partitioning Guide, ⟨https://docs.oracle.com/database/121/VLDBG/toc.htm⟩; 2014.
[38] van Oosterom P, Vijlbrief T. The spatial location code. In: Proceedings of the 7th international symposium on spatial data handling, Delft, The Netherlands; 1996. p. 3B.1–17.
[39] Finkel R, Bentley J. Quad trees a data structure for retrieval on composite keys. Acta Inf 1974;4(1):1–9 URL: ⟨http://dx.doi.org/10.1007/BF00288933⟩.
[40] van Oosterom P, Meijers M. Vario-scale data structures supporting smooth zoom and progressive transfer of 2d and 3d data. Int J Geogr Inf Syst 2014;28(3):455–78.