Computers & Geosciences 51 (2013) 430–438

Contents lists available at SciVerse ScienceDirect

Computers & Geosciences

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

Octree-based indexing for 3D pointclouds within an Oracle Spatial DBMS

Bianca Schön[a], Abu Saleh Mohammad Mosa[a], Debra F. Laefer[b,*], Michela Bertolotto[a]

[a] *School of Computer Science & Informatics, University College Dublin, Belfield, Dublin 4, Ireland*
[b] *School of Civil Engineering, University College Dublin, Belfield, Dublin 4, Ireland*

A R T I C L E I N F O

Article history:
Received 27 June 2011
Received in revised form
19 June 2012
Accepted 30 August 2012
Available online 12 September 2012

Keywords:
LiDAR pointcloud data
Laser scanning
Three-dimensional indexing
Spatial databases
Octree

A B S T R A C T

A large proportion of today's digital datasets have a spatial component. The effective storage and management of which poses particular challenges, especially with light detection and ranging (LiDAR), where datasets of even small geographic areas may contain several hundred million points. While in the last decade 2.5-dimensional data were prevalent, true 3-dimensional data are increasingly commonplace via LiDAR. They have gained particular popularity for urban applications including generation of city-scale maps, baseline data disaster management, and utility planning. Additionally, LiDAR is commonly used for flood plane identification, coastal-erosion tracking, and forest biomass mapping. Despite growing data availability, current spatial information systems do not provide suitable full support for the data's true 3D nature. Consequently, one system is needed to store the data and another for its processing, thereby necessitating format transformations. The work presented herein aims at a more cost-effective way for managing 3D LiDAR data that allows for storage and manipulation within a single system by enabling a new index within existing spatial database management technology. Implementation of an octree index for 3D LiDAR data atop Oracle Spatial 11g is presented, along with an evaluation showing up to an eight-fold improvement compared to the native Oracle *R*-tree index.

1. Introduction

Recent developments in LiDAR technology have produced accurate, detailed and truly three-dimensional (3D) datasets. Consequently, LiDAR data have been widely used in critical urban applications, including large-scale city map generation and change detection in urban environments. While LiDAR datasets provide high accuracy and resolution, they also pose significant size-based challenges. As an example, typical low-density aerial scanning generates 30–50 points/m^2 (a 2001 flood plane mapping generated approximately 5.6 billion points for the entire state of North Carolina) (Laefer and Pradhan, 2006), with as much as 225 points/m^2 resulting in 225 million points for one square kilometer (Hinks et al., 2009).

Current spatial information systems do not provide suitable support for 3D data manipulation and evaluation. So while a specific system can store the data, another must process it, which requires format transformations with consequent potential accuracy losses. The work herein aims at providing a more efficient and cost-effective way that allows storage and manipulation of 3D LiDAR datasets within a single system. The proposed solution exploits current spatial database management system (SDBMS) technology and its extensibility capabilities that allow additional functionality development. While extensive support for 2D data is available from several database management system (DBMS) vendors including Postgres and Oracle, very limited support is provided for 3D data handling. To achieve efficiency, suitable 3D indexing mechanisms are essential (Hongchao and Wang, 2011). Presently, Oracle is currently the only SDBMS that provides native 3D spatial data types and 3D index implementation, but it is based on an *R*-tree and possesses inherent inefficiencies when applied to LiDAR data, as will be demonstrated. This paper shows how octree-based indexing can greatly facilitate the storage and indexing of 3D pointcloud data within an SDBMS. Presented herein is an implementation of an octree index atop Oracle Spatial 11g and benchmarking demonstrating its outperformance of the native *R*-tree index commercially provided.

2. Background and technologies

This section outlines various technologies and approaches currently used to store and index 3D pointcloud data. First, support of SDBMSs is investigated, followed by an overview of indexing approaches for 3D pointdata. Subsequently, octree indexing implementation is explained.

2.1. SDBMS support for 3D pointcloud data

Many of high-resolution LiDAR's benefits remain relatively unexploited, as the data cannot be efficiently managed in a traditional Geographical Information System (GIS), because of an inability of GISs

* Corresponding author. Tel.: +353 1 716 3226; fax: +353 1 716 3297.
E-mail addresses: debra.laefer@ucd.ie, Debra.Laefer@ucd.ie (D.F. Laefer).

0098-3004/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cageo.2012.08.021

to fully support 3D objects (Zlatanova et al., 2004). For example, GIS systems are not designed to support finite element meshes, which are often the intended end products for a LiDAR scan (Laefer et al., 2011). A desirable alternative would overcome the need for multiple programs with their required import and export transactions (to eliminate the potential loss of accuracy through format conversions) and would not be file-based (due to the datasets' very large sizes). The solution proposed herein integrates all required functionality within a single SDBMS.

A DBMS controls the organization, storage, management, and retrieval of all data within a database and ensures that data inconsistencies and data redundancies are significantly reduced compared to a file-based, storage system. A DBMS also facilitates data integrity, as well as multi-user control on shared data. Initially, traditional DBMSs did not support storage and querying of spatial data (i.e., data with a spatial component). Later, an integrated approach was developed to store the spatial extent (together with the attribute data) directly into the database (in the same table). Current SDBMSs, such as Oracle Spatial or PostGIS are based on the extensibility (i.e., the ability to add new types and operations) of relational database management systems. This allows for all the data management within the same engine. Additionally, retrieval and manipulation are facilitated through structured query language (SQL).

For a spatial system to be fully 3D, it must support 3D data types including volumes in 3D Euclidean space. Such data types are based on a 3D geometric data model (i.e., vector and/or raster data with underlying geometry and topology). A 3D spatial system must also offer operations and functions embedded into its query language operable with its 3D data types (Breunig and Zlatanova, 2011). Until recently, SDBMSs have not provided support for 3D data management, as extensively reviewed by Schön et al. (2009a). However, with Oracle Spatial's release of 11g, 3D pointclouds can be stored with a built-in data type. Previously, Oracle Spatial relied heavily on SDO_GEOMETRY. More recently Oracle Spatial has moved to SDO_PC as the main data type employed for the storage of multi-dimensional pointcloud data. With that, a set of points are grouped and stored as the BLOB object in a row. Although there is no upper bound on the number of points in an SDO_PC object, the current version offers only nine placeholders for information storage alongside locational attributes. The main drawback is the inability to update SDO_PC objects. Consequently, the SDO_GEOMETRY data type still offers the greatest flexibility for 3D data points and was, thus, used in the proposed implementation. For 3D pointclouds, it is desirable to store locational information together with attribute information (e.g., colour, intensity) in the same table, as semantic information often times directs feature recognition processes. This would typically occur later in the workflow. One example for this is a spatial query that operates within a particular colour characteristic region, such as for example a row of buildings that are uniquely discernible by color. Being able to avail of this simple information should result in significant query performance improvements. Indexes are employed to avoid traversing a complete table when performing spatial queries. Thus, indexes are used to organize the space and the objects within that space. Given the large size of LiDAR datasets, efficient indexing mechanisms are essential (Breunig and Zlatanova, 2011), as discussed in Section 2.2.

2.2. Indexing of 3D pointcloud data

Stanzione and Johnson (2007) argue that a tree structure is inherently efficient for indexing due to its binding with the internal data storage structure. Thus, various tree structures have been explored for indexing pointcloud data. Spatial indexing techniques evolved in the mid-1980s, with Guttman's *R*-tree (Guttman, 1984) being one of the most popular and enduring. Combining an *R*-tree with an importance value (Van Oosterom, 1990), which is called *V*-reactive tree (Li et al., 2001) is one approach to indexing LiDAR data. The *V*-reactive tree is an *R*-tree structure in 4D, optimized for 3D visualization. However, to date, this has not been tested for realistically large pointcloud datasets. Hua et al. (2008) proposed a hybrid approach for visualization by combining an octree with a *k*–*d* tree (Bentley, 1975) by building a local *k*–*d* tree at each octree level node but only evaluated visualization speed for 3D pointclouds for up to 100,000 points (Hua et al., 2008). In an alternative approach, De Floriani and Facinoli (2010) extended two-dimensional (2D) quadtree indexes to work with TIN structures and argued that their mechanism could be generalized to support Tetrahedral Irregular Networks (TENs) on an octree basis to support true 3D functionality.

Hierarchical space-division based structures (e.g., octrees) are critical for 3D surface representations and queries, as they are volume based. Combined approaches, such as the volume–surface tree (*V*–*S* tree) aim to avoid a strong imbalance with regards to clustering of points by applying a 3D octree globally and a 2D quadtree locally (Boubekeur et al., 2006). However, this method tends to collapse for non-smooth surfaces, which then requires pure octree indexing. Another interesting LiDAR indexing approach is based on the Hilbert space-filling curve (Wang and Shan, 2005). Space-filling curves preserve spatial proximity at a local level and map points in an *n*-dimensional space into a linear order (Sagan, 1994). This approach was implemented in MySQL and the Microsoft Access Database for evaluation purposes and tested by Wang and Shan (2005) on 1.4 million LiDAR points from a terrestrial scan of a bridge. Notably, Microsoft Access currently does not provide any spatial support, while MySQL Spatial offers only rudimentary spatial support by providing spatial data types, functions, and a spatial index. Due to the limited functions in MySQL Spatial, this database is best used for simple retrieval by bounding box operations.

Presently, Oracle Spatial provides an *R*-tree based spatial index and a deprecated 2D quadtree based on minimum bounding rectangles (MBRs), and the 3D extension consists of minimum bounding boxes (MBBs). Implementing a bounding box on a dense pointcloud is, however, non-trivial and may introduce inefficiencies due to overlapping of sibling nodes and uneven node sizes (Zhu et al., 2007). An alternative is to map spatial objects onto a one-dimensional space to enable use of a standard index, such as a *B*-tree (Bayer, 1971). Another option is PostgreSQL, which supports the Generalized Search Tree (GiST) index (www.geospatial.org/), a template data structure for abstract data types that offers more robust support for spatial indexing.

Several strategies have been developed for indexing of multi-dimensional data, although there is limited vendor support for these, and true 3D index creation is still an ongoing research problem (Schön et al., 2009b). In most cases, indexes only support two-dimensionality with simple 3D extensions (Arens et al., 2005). An octree offers an alternative, but currently no commercially-available SDBMSs support octree indexing, and to the best of the authors' knowledge, no meaningful benchmarks have been provided thus far on this approach. This paper rectifies these deficiencies.

2.3. Octree indexing for spatial 3D pointcloud data

An octree structure offers distinct advantages over the frequently implemented *R*-tree for indexing LiDAR datasets. First, octrees can index point geometries directly, as opposed to the *R*-trees that solely rely on bounding boxes. Furthermore, octrees generate disjointed, non-overlapping tree nodes, whereas *R*-tree bounding boxes are often overlapping, which reduces query efficiency. Moreover, storing the logical tree structure into a SDBMS is complex. The tree structure can be stored in a table where each node of the tree structure corresponds to a row in the table. In that case, one column is needed to store node identifiers (nodeID) and another to store the list of node identifiers (nodeIDs), as pointers to the children nodes. The node identifier of the root node can be stored in a table, called the metadata table for that index. Oracle Spatial's *R*-tree index implementation stores the tree structure in a table and selects a node using an internal SQL statement, as each node is visited (Kothuri et al., 2002). Thus, query operations involve the processing of many recursive SQL statements, which increases query processing time (Kothuri et al., 2002). The octree, in contrast, can divide the entire space according to a specified tiling level requiring only the tiling level to be stored, as the tree structure can be rebuilt during querying; details are described in Section 3.

A further advantage of the octree lies in its support for optimized 3D pointcloud visualization (Koo and Shin, 2005). Rendering of 3D pointclouds is computationally expensive, and an SDBMS causes further delays due to I/O operations. However, an octree can be utilized to filter visible points for rendering according to a specific view frustum, instead of rendering all points at once. Nonetheless, selection of an appropriate spatial index depends on many factors, such as data distribution and data type. Octrees provide an approach highly applicable to all 3D pointcloud object types. The following section outlines how an octree index can be implemented in Oracle Spatial.

3. Design of an octree index atop Oracle Spatial 11g

Oracle's extensibility framework requires that a data cartridge be implemented, to provide a new index structure. Data cartridges are re-usable, server-based components, which utilize object types and features such as large objects, external procedures, extensible indexing, and query optimization. Oracle's extensible indexing framework defines a set of interface methods. These must be implemented in an object type, which is called indextype. An indextype is an object that specifies the routines that manage a domain (application-specific) index. It has two major components: (1) methods that implement the index's behavior and (2) operators that the index supports.

This paper describes a new data cartridge implemented in Oracle's extensible indexing framework that enables octree indexing, which is subsequently referred to as OCTREEINDEX. To facilitate the analysis of 3D pointclouds, a window query operator OT_CLIP_3D was also implemented, which performs a window query on a given 3D point geometry stored in an Oracle SDO_GEOMETRY data type. Spatial metadata information is stored in the USER_SDO_GEOM_METADATA view provided by Oracle Spatial (Kothuri et al., 2007, p. 45). The following presents the index and related window query operator implementation.

3.1. Implementation of the octree index

An octree's structure dictates that each internal node contains exactly eight child nodes regardless of its many variants. In the implementation herein a bucket point region (PR)-octree approach was adopted, where the space is decomposed into cubic blocks (or cells) through recursion, until a block is homogeneous (Samet, 2006). While use of a bucket PR approach might seem an obvious solution, this is not the direction that has been adopted to date (in either research or commercial solutions). In fact, as of version 11g Oracle deprecated the quadtree. The absence of continued support would lead users to believe that further exploitation of this class of indexing structures might not be useful. The contrary is shown in this paper.

By definition, an octree can result in an unbalanced hierarchical tree when the data distribution is not uniform. However, this requires the storage of the logical tree structure in the SDBMS for recursive reconstruction of the tree structure during query processing. While testing the presented implementation, it was found that this compromised the system's efficiency. Therefore, the proposed implementation employs a fixed, maximum tree height (also called its tiling level), thereby resulting in a balanced tree. This improves query efficiency as neither the tree structure nor recursive cells need to be stored, only the tiling level. The selection of an appropriate tiling level is a decisive factor, involving the dataset's area and size. As such, experimentation with different levels is needed to optimize performance for a specific dataset. Particulars of this problem are further illustrated in Section 4. The user can specify the tiling level through the parameter OCTREE_LEVEL during index creation. Each cell is associated with a unique code, herein referred to as the cell code. The cell code is obtained by using *z*-ordering (i.e., Morton encoding) of all cells at the specified level (Morton, 1966).

Fig. 1(a) illustrates the 3D space decomposition through an octree, and Fig. 1(b) illustrates cell code generation. All cells in the

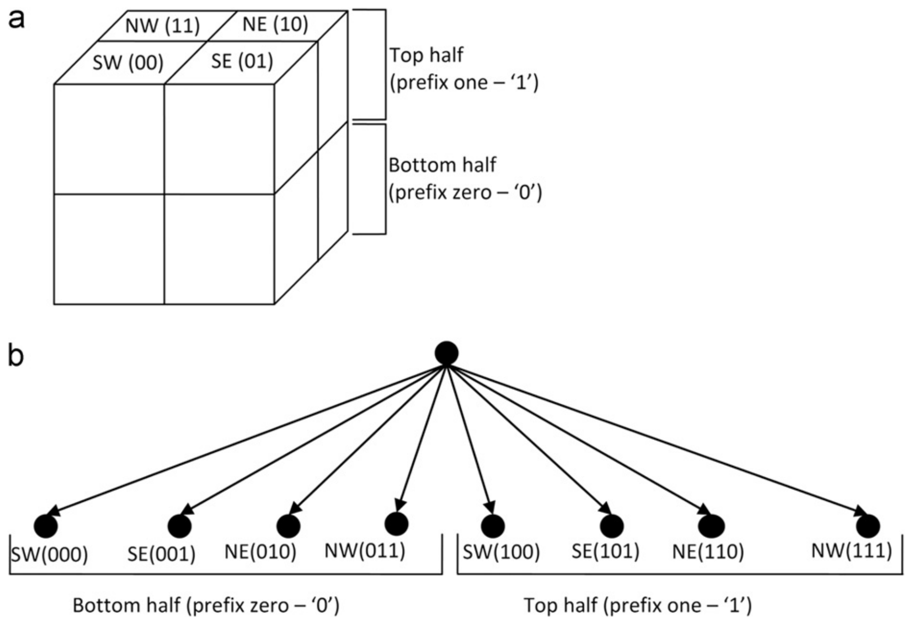

Fig. 1. Quadtree sectors. (a) 3D space decomposition. (b) Cell code generation.

bottom half are assigned with the prefix '0'—zero, and all in the top half are assigned with prefix '1'—one. Cells are marked south-west (SW), south-east (SE), north-east (NE) and north-west (NW) and associated codes are 00, 01, 10, and 11 consecutively. The associated cell code is identified by traversing the octree from root node to leaf node. For example, using B to represent the bottom half and T to designate the top half, at tiling level 5, the code for the path BNW(011)–TSW(100)–TNE(110)–BSE(001)–BSW(000) is 011100110001000. Here, it only follows the tree path where the cell associated to a node in the path contains the point. The point's ROWID and associated cell code are stored in an index storage table. The metadata (e.g., tiling level, index name, index owner, max level, min level, etc.) for the entire index are stored as a row in an table called index metadata table.

The 3D query processing using this implementation is illustrated in Fig. 2. To generate the result set for a spatial query, the octree index is used as the primary filter to find the area of interest or candidate geometries for this query. Fig. 3 illustrates use of a primary and secondary filter during querying. The area of interest is the sum of the cells of the octree that interact spatially (e.g., intersect, touch, inside, covered by) with the query geometry, as established by the primary filter. These are identified by cell code, and candidate geometries are identified by the associated cell code from the index storage table. Candidate geometries are passed through the intermediate filter and divided into two sets. Cells inside or covered by the query geometry are identified as an exact match. Points associated with these cells are sent directly to the result set. The remaining cells (those that intersect or touch the query window) are passed through the secondary filter, which is a spatial function corresponding to the spatial query.

Notably, in Oracle Spatial there is no specific operator to perform a window query. SDO_RELATE identifies all geometries

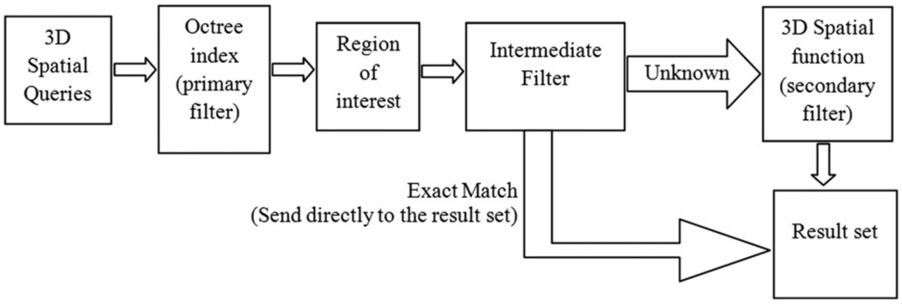
3D Spatial Queries
Octree index (primary filter)
Region of interest
Intermediate Filter
Unknown
3D Spatial function (secondary filter)
Exact Match (Send directly to the result set)
Result set

Fig. 2. Query processing steps.

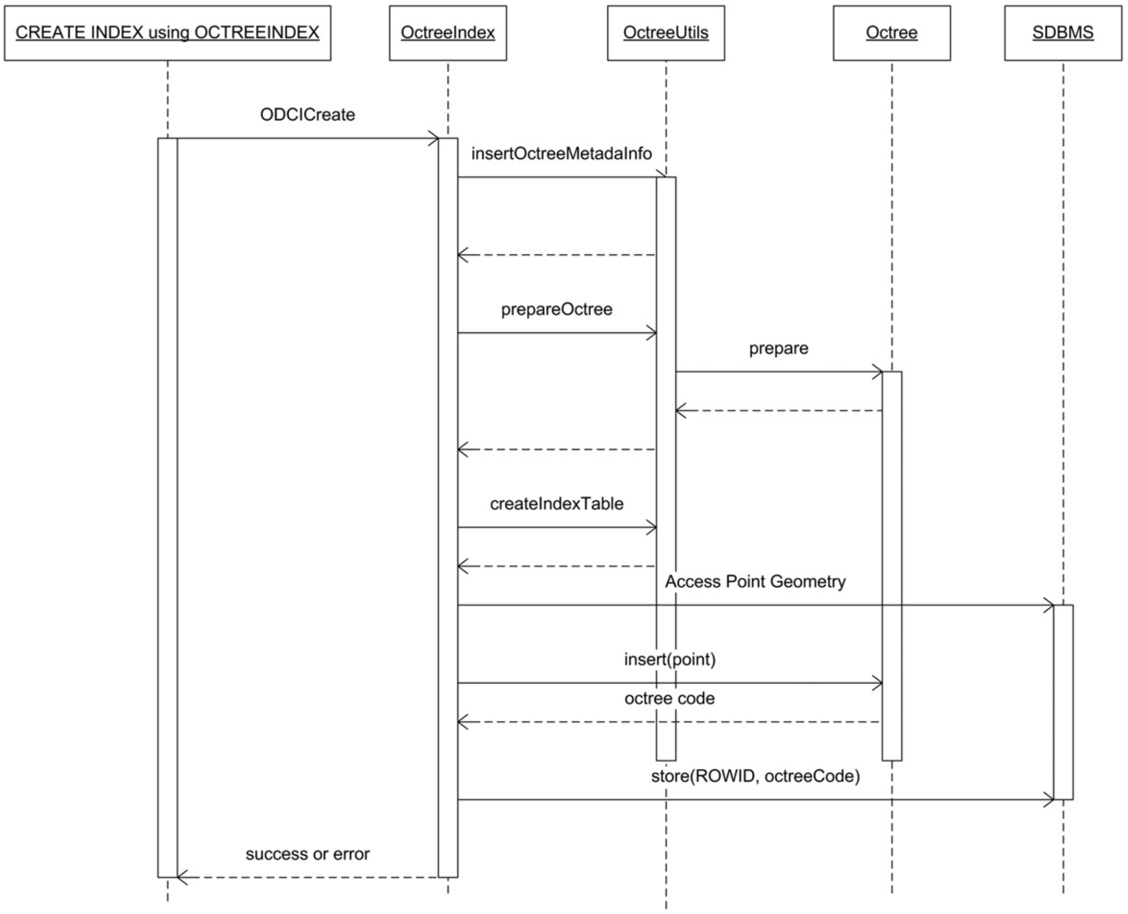

Fig. 3. Index creation sequence.

that interact in a specified manner with a query geometry. The specified type of interaction could be INSIDE, CONTAINS, COVEREDBY, ON, COVERS, TOUCH, OVERLAPPEDBYINTERSECT, EQUAL, and ANYINTERACT (Kothuri et al., 2007, pp 272–281). Since a 3D window query operator was implemented herein, utilization of the SDO_RELATE operator and a combination of interaction-type (i.e., 'QUERYTYPE=WINDOW MASK=INSIDE+COVEREDBY') was made in order to perform the window query on the *R*-tree index. This has been passed through the "param" parameter of SDO_RELATE operator. This retrieves all the points that are inside and may also touch the boundary of the query window. In Oracle, the combination of INSIDE and COVEREDBY is optimized. See Kothuri et al. (2007), pp. 278–279 for a detailed description.

The Oracle extensibility framework requires that a new index must implement a certain interface and related methods. The name of the interface is ODCIIndex. Associated methods are categorized into four classes: (1) index definition methods, (2) index maintenance methods, (3) index scan methods, and (4) index metadata method. Table 1 summarizes these methods with implementation details described henceforth.

An implementation type is required to create the indextype OCTREEINDEX and must contain the implementation of the ODCIIndex interface methods. An object type known as the implementation type and named OCTREE_IM is defined to implement the ODCIIndex interface methods. It contains the signature and return type of the interface methods. The body of OCTREE_IM contains the implementation of these, which can be implemented using PL/SQL, C, C++ or Java. In this implementation, only the ODCIGetInterfaces method is implemented in PL/SQL, while others are implemented as Java callouts, which reside in a Java class. A previously developed Java API was exploited (Kothuri et al., 2007, p. 223) to enable Java applications to access and process geometry objects managed in Oracle Spatial. OCTREE_IM contains only the implementation of the method ODCIGetInterfaces, while others are implemented in a Java class entitled OctreeIndex. Mapping of the interface methods to the Java methods is defined in OCTREE_IM. The process of index creation is outlined below. Other methods implemented for the prototype, as explained in Table 1, are created accordingly. Fig. 3 illustrates the index creation process, and Fig. 4 illustrates requisite steps.

The ODCICreate method is invoked when a user issues the "CREATE INDEX" SQL statement of indextype OCTREEINDEX. This starts the index creation process. At first, metadata information regarding the index is stored into an index metadata table named OCTREE_INDEX_METADATA. Next, the octree structure is initialized, and the 3D bounding of the 3D pointcloud sample is created as a whole. Three-dimensional points stored as point geometry data types are accessed from the base table through the Java Database Connectivity connection with the database. These are inserted into the octree structure, which returns a cell code for each point.

Table 1
ODCIIndex interface methods.

Category	Method name	Invoked by
Definition methods	ODCIIndexCreate()	"CREATE INDEX" statement
	ODCIIndexDrop()	"DROP INDEX" statement
	ODCIIndexAlter()	"ALTER INDEX" statement
Maintenance methods	ODCIIndexInsert()	"INSERT INTO" statement on the base table, which involves the indexed column.
	ODCIIndexUpdate()	"UPDATE" statement on the base table, which involves the indexed column.
	ODCIIndexDelete()	"DELETE FROM" statement on the base table, which involves the indexed column.
Scan methods	ODCIIndexStart()	At the beginning of an index-scan.
	ODCIIndexFetch()	In order to fetch the row identifiers those satisfies the operator predicate.
	ODCIIndexClose()	At the end of the index-scan In order to perform cleanup.
Metadata methods	ODCIIndexGetMetadata()	In order to write implementation-specific metadata into the export dump file using "Export" utility.

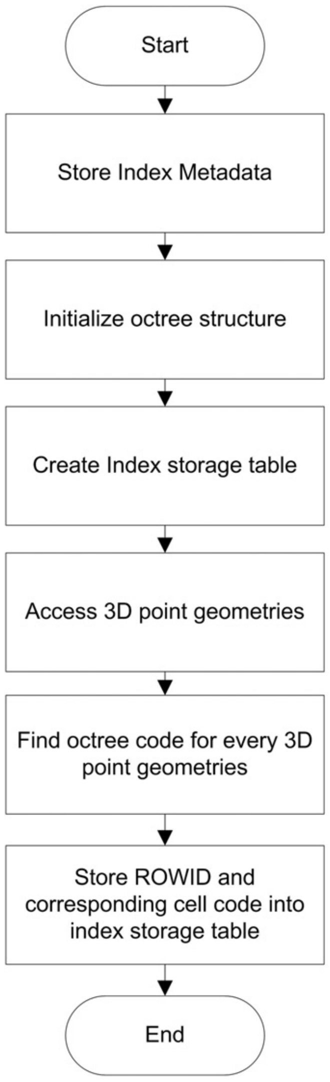
Start
Store Index Metadata
Initialize octree structure
Create Index storage table
Access 3D point geometries
Find octree code for every 3D point geometries
Store ROWID and corresponding cell code into index storage table
End

Fig. 4. Steps needed for index creation.

3.2. Operator implementation

Window queries are among the most commonly used first-step-analysis operations for LiDAR data. As such, this query was implemented and is referred to as OT_CLIP_3D. The operator returns all point geometries inside and on the boundary of the specified 3D cube and takes two SDO_GEOMETRY objects as input. The first is a 3D point geometry or a column of the type SDO_GEOMETRY that contains a 3D point geometry on which the operator is applied. The second is a simple solid of type SDO_GEOMETRY, which specifies the query window. Every operator must be tied to an index for index-based evaluation. Oracle's extensible indexing framework requires implementation of index scan methods to evaluate operators. These are ODCIIndexStart, ODCIIndexFetch and ODCIIndexClose. Fig. 5 illustrates the invocation sequence of index scan methods.

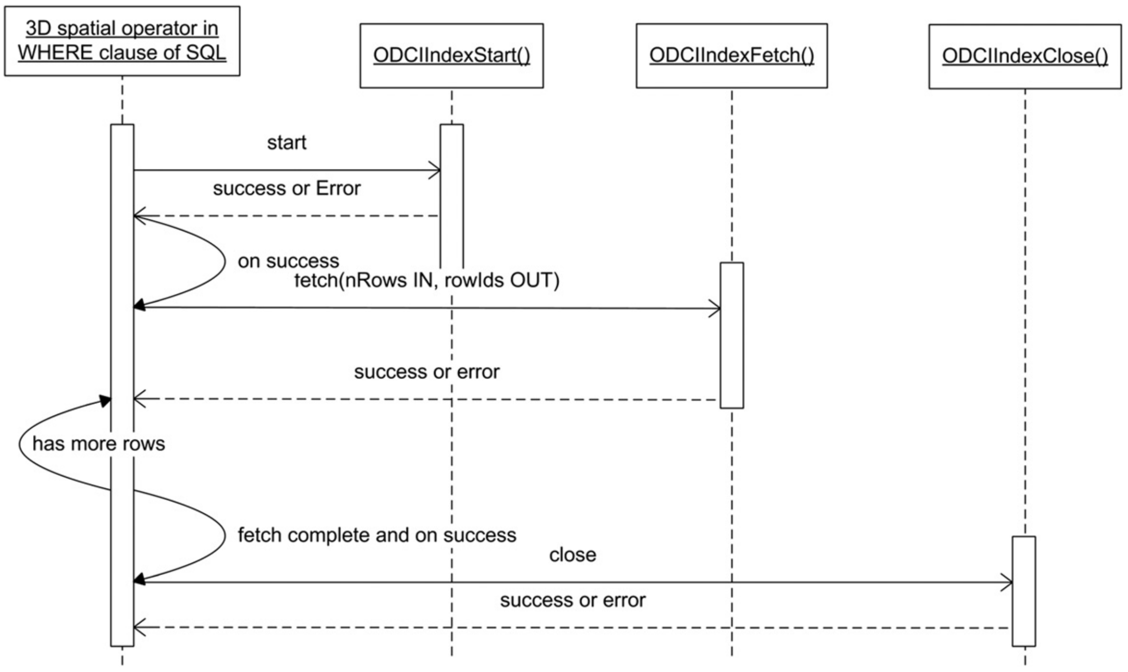

Fig. 5. Index invocation.

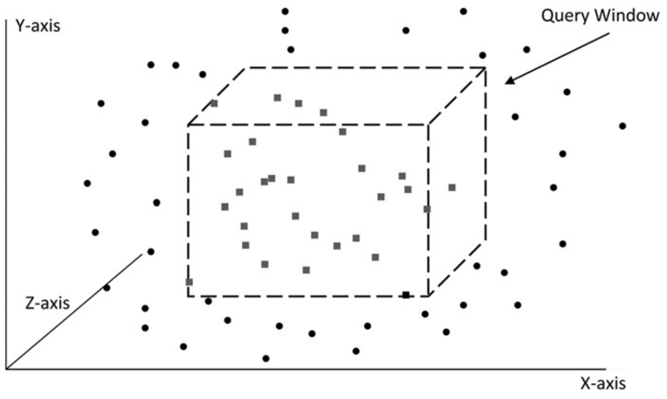

Fig. 6. Query window.

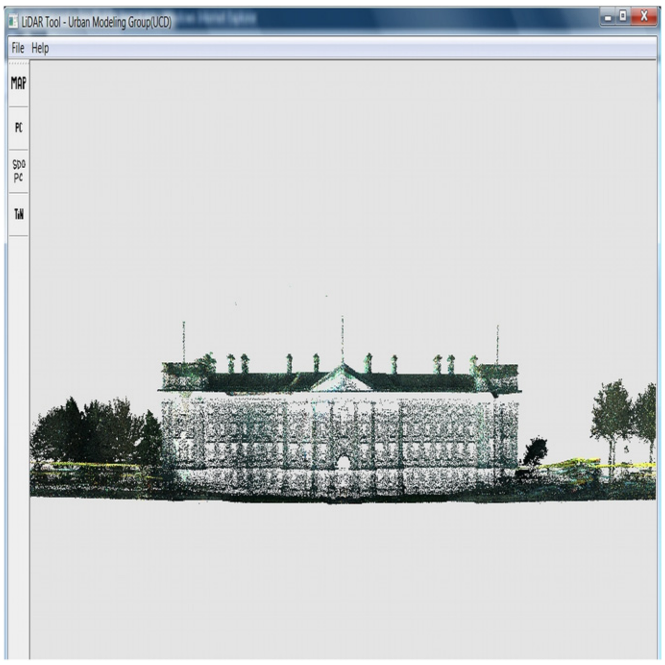

Fig. 7. Sample of 3D data.

At first, the interface method ODCIIndexStart is invoked by Oracle with the operator name, arguments, and the lower and upper bounds describing the predicate. This method is invoked to begin the operator evaluation. A series of fetches are performed by invoking the ODCIIndexFetch method to obtain row identifiers or rows that satisfy the operator predicate. The number of expected rows (nRows) in every fetch is specified by Oracle during each invocation. The ROWIDs are placed into the placeholder array (rowIds). Finally, before the destruction of the SQL cursor, ODCIIndexClose is invoked by Oracle to end the operator processing. Fig. 6 illustrates the window query performed on a 3D pointcloud. The result set returns all point geometries inside or on the boundary of the query window. In this example, all square points are inside or on the boundary of the query window (illustrated by the box of dotted lines). The OT_CLIP_3D operator is evaluated through the octree index. Fig. 7 shows a sample of the 3D data. Fig. 8 demonstrates the evaluation process. For ease of illustration, the example is shown as a 2D case. The query window is drawn in dotted lines and the resulting geometries as solid squares. The octree is traversed to identify cells that interact or are topologically related with the query window. Possible topological relations are “inside”, “intersect”, “touch”, and “covered by” (Egenhofer and Franzosa, 1991).

Blocks that are (1) inside the query window or (2) intersect with it or (3) touch it, or (4) are covered by it are identified, and the area of interest is the union of these blocks. This area is searched to generate the result set. In Fig. 8, Block E is inside the query window, block H is covered by it, and all others intersect with it.

The intermediate filter identifies any exact match (e.g., blocks E and H). A block inside the query window, implies that all points belonging to such blocks are also inside the query window. These points are sent directly to the result set and labelled as known, and the blocks are labelled as known regions. Points covered by

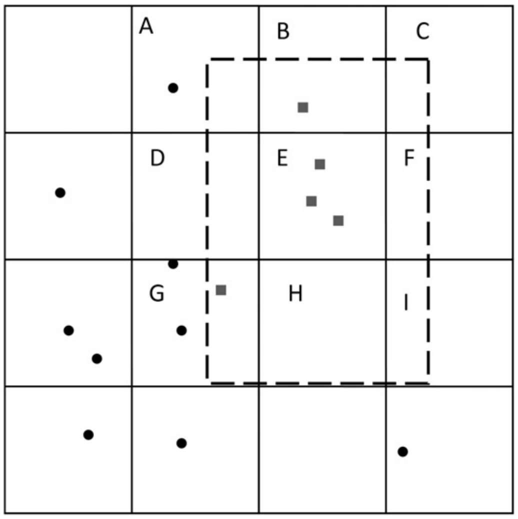

Fig. 8. Octree query window.

the other blocks are passed through the secondary filter, as those points are labelled as unknown and require further processing. The following benchmarks this approach for the purpose of validation.

4. Evaluation

This section benchmarks window query response times on a 3D LiDAR pointcloud dataset between *R*-tree and octree indexing. The evaluation was conducted on a computer with the Intel Core2 Duo CPU 2.53 GHz and 4 GB RAM, 7200 SATA harddrive using Oracle 11g release 11.1.0.6.

The 3D LiDAR pointcloud dataset was stored in Oracle's SDO_GEOMETRY data type. To benchmark performance of spatial queries on 3D pointcloud data, the dataset was indexed using the *R*-tree and the octree index, independently. For this, two randomly selected datasets from a dense aerial 3D LiDAR flyover of Dublin's city centre (Hinks et al., 2009) were queried in order to test each index. Given the dense and somewhat random distribution of the source data set this has ensured that realistic query sizes are used for evaluation. One query contained nearly 2.9 million points and the other almost 66 million points. Query response times were compared for various window sizes. The *R*-tree index was created using Oracle's existing, in-built spatial index. The octree index structure was created through the implementation described in Section 3.1 in Oracle's extensible indexing framework.

In Oracle Spatial, the *R*-tree index supports only one operator with which a 3D spatial query can be performed. Furthermore, this provides no window query functionality. Consequently, to perform a window query on a LiDAR dataset, a 2D index was created to allow for a 2D window query. Since Oracle's in-built 2D *R*-tree index is created on the 3D pointcloud, it is assumed that the index is created on the 2D projection of the 3D pointcloud data. The SDO_RELATE operator provides functionality similar to a general window query. The "inside and touch" masks (Kothuri et al., 2007, p. 274), and the 2D query window are specified to perform the window query.

The octree index implemented herein supports the operator OT_CLIP_3D, which performs a 3D window query on 3D pointcloud data. To create the octree index, it is very important to determine the tiling level for efficient query processing. For this purpose, all points are considered. The indexed points per cell and cell volume decrease as the tiling level increases, which in turn decreases the total number of candidate geometries. Conversely, the number of leaf nodes increases (leaf nodes at tiling level '*N*' is 8^N). As such, memory consumption increases at higher tiling levels. Tiling level five was experimentally selected for this dataset. This was based on Oracle's *R*-tree recommendation of a tiling level of 8, which generated a memory error. Tiling levels 4–7 were then tried, with level 5 generating the best results; there are future opportunities for automated determination of this. For the small dataset of 2,881,899 points/8^5, average number of points indexed by each octree node 87.95. For the large dataset of 65,562,235 Points/8^5, average number of points indexed by each Octree node was 2000.8.

In the octree implementation, the index storage table has two columns: OCTREE_CODE (oracle data type RAW, in order to store the cell code, requires 3 bits for each branch), and OCTREE_ROWID (oracle data type ROWID, 10 bytes in size, in order to store the ROWID of the 3D point geometry). In the experiment herein, OCTREE_LEVEL=5 and the size of index storage table is $(12 \times N)$ bytes. The implementation does not require any additional storage for temporary worktables during index creation. Consequently, for a set of *N* rows in a table, the *R*-tree spatial index roughly requires $100 \times 3 \times N$ bytes of storage space for the spatial index table. Also, during index creation, it requires an additional $200 \times 3 \times N$ to $300 \times 3 \times N$ bytes for temporary worktables. (Kothuri et al., 2007, p.253).

In this example, a fairly uniform aerial LiDAR dataset was used as it represents a portion of Dublin's city centre in Ireland, where relatively few large occlusions exist, as average building height is low. With this dataset, the response time of a 2D window query (*x*- and *y*-coordinates) using an *R*-tree index was compared with the response time of a 3D window query (*x*-, *y*- and *z*-coordinates) using an octree index. The query response times, as well as the number of resulting geometries, were expected to increase with a larger query window size (Kothuri et al., 2002). For reasons of comparability, the same number of resulting geometries for a window query were used to test both indexes. To ensure this for the octree index, the maximum and minimum values of the *z*-coordinates of the query window were set to the minimum and maximum values of the underlying space. The same values were used for *x*- and *y*- coordinates for the octree and *R*-tree indexes. Thus, the total number of resulting geometries was equal for both indexes.

Table 2 presents the average query response time with the increase of the window size for both indexes. For every window size, up to 625 queries were performed in the underlying space.

Table 2
Evaluation results.

Small dataset of 2881,899 million points			**Large dataset of 65,562,235 million points**		
Window size (m^2)	**Avg. query response time in ms (*R*-tree)**	**Avg. query response time in ms (Octree)**	**Window size (m^2)**	**Avg. query response time in ms (*R*-tree)**	**Avg. query response time in ms (Octree)**
25	3,720.40	1686.28	400	83,026.65	128,814.14
100	10,868.86	1975.92	1,600	166,708.00	149,541.69
225	17,170.21	3121.44	3,660	321,180.30	243,061.55
400	23,269.12	4628.71	6,400	467,678.50	245,920.08
625	30,816.40	5804.15	10,000	641,871.10	250,993.00
900	42,541.17	6934.25	14,400	864,345.50	257,525.44
1225	42,277.08	6329.17	19,600	1065,853.90	269,746.34
1600	68,354.84	7521.83	25,600	1281,446.50	286,461.88
2025	70,115.16	8328.33	32,400	1535,893.00	310,641.75
2500	83,238.25	9462.75	40,000	1933,097.50	321,632.50

The details of these queries are shown in Table 3, and the reported times (Figs. 9 and 10) represent the average of these queries. The octree index nearly consistently outperformed the *R*-tree index for all window sizes.

The dataset used in Fig. 10 is ~23 times the size Fig. 9's dataset where the octree is twice as fast as the *R*-tree for the small window of (25 m^2) and 8 times faster for the large window (2500 m^2). In Fig. 10, for the small window of 400 m^2, the *R*-tree outperforms the octree, but once the window reaches 1600 m^2, the octree is better, with a six-fold improvement for a 40,000 m^2 window.

Table 3
Number of window queries per window size.

Small dataset of 2881,899 points		**Large dataset of 65,562,235 points**	
Window size (m^2)	No. of queries performed (i.e., no. of different windows)	Window size (m^2)	No. of queries performed (i.e., no. of different windows)
25 (5 m × 5 m)	588	400 (20 m × 20 m)	625
100 (10 m × 10 m)	140	1,600 (40 m × 40 m)	144
225 (15 m × 15 m)	63	3,600 (60 m × 60 m)	64
400 (20 m × 20 m)	35	6,400 (80 m × 80 m)	36
625 (25 m × 25 m)	20	10,000 (100 m × 100 m)	25
900 (30 m × 30 m)	12	14,400 (120 m × 120 m)	16
1,225 (35 m × 35 m)	12	19,600 (140 m × 140 m)	9
1,600 (40 m × 40 m)	6	25,600 (160 m × 160 m)	9
2,025 (45 m × 45 m)	6	32,400 (180 m × 180 m)	4
2,500 (50 m × 50 m)	4	40,000 (200 m × 200 m)	4

5. Conclusions and summary

This paper implements and evaluates an octree index, intended for 3D pointcloud data from laser scanning, employing Oracle's extensible indexing framework. However, its functionality may be cross-applicable to other pointcloud datasets and implementable in other SDMSs as they expand their 3D capabilities. An operator using the proposed octree index was implemented to perform 3D window queries. This was described, along with some optimizations. The newly implemented octree index and Oracle's inbuilt *R*-tree index were compared using data from a dense, aerially-based, 3D pointcloud. The octree consistently outperformed the *R*-tree for almost every window size and more so with increases in query window size, to as much as an eight-fold difference. The considerably improved performance, while notable in itself, needs to be considered further in light of the additional functionality offered by the octree in terms of a more appropriate storage and indexing of pointcloud data in particular. As such, groupings into appropriate cells may occur according to a predefined semantic, such as for example colour, intensity, or elevation information. Furthermore, the approach is not plagued with the *R*-tree's related uncertainty when trying to select a bounding box for point. It may be argued that the under-performance of the *R*-tree is due to using unsuitable parameters

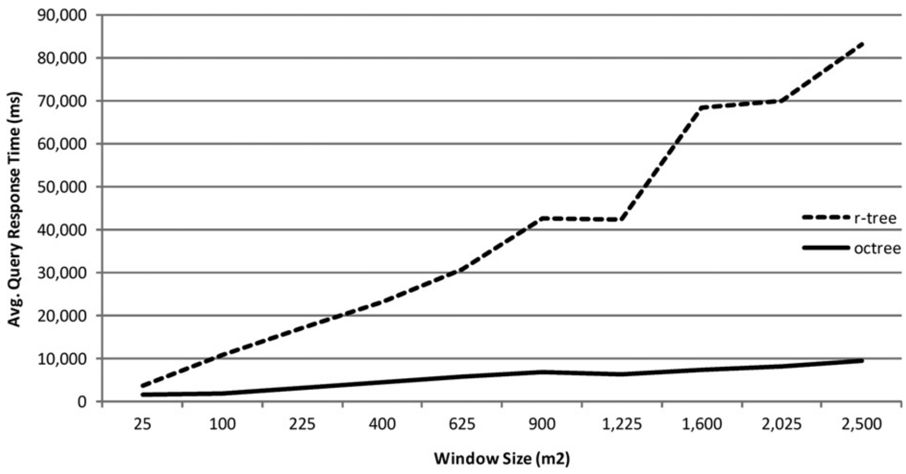

Fig. 9. *R*-tree vs. Octree 2881,899 points in the dataset.

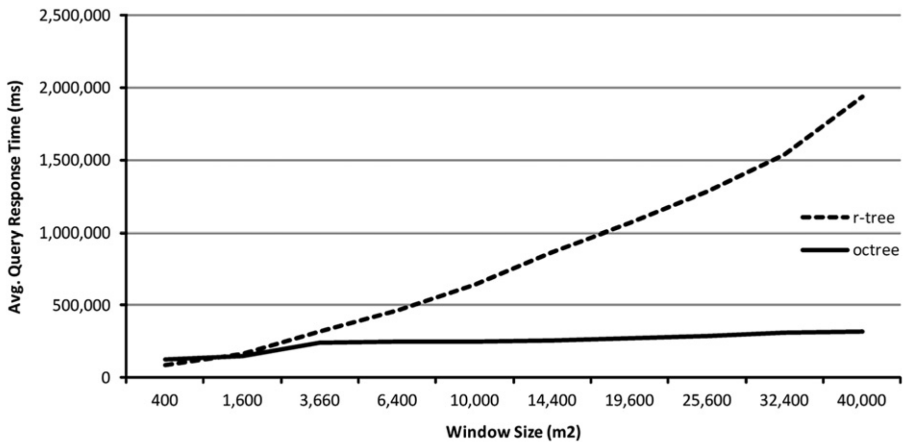

Fig. 10. *R*-tree vs Octree 65,562,235 million points in the dataset.

during bounding-box generation. However, there is always a trade-off between different options for generating an indexing structure. These might not be obvious to an inexperienced GIS technician/user and often result in a "trial and error" process. If the data set is very large, the "trial-and-error" process phase might take considerable time.

Since only one operator has been implemented so far, further work is envisioned (e.g., nearest neighbor or within distance) to more comprehensively evaluate the octree's potential. In this prototype, tiling level is user determined. Further work will incorporate a feature for automatic tiling level determination, along with exploitation of the visualization-based efficiencies that this approach will engender.

In Oracle Spatial, the SDO_PC data type applies an *R*-tree index only to the groups of clusters that contain point geometries. An alternative approach to the one presented in this paper may rely on a two-step index, where an octree index is applied to points inside a block, and an *R*-tree is applied as a higher-level index to the block extents, as polygons are better indexed by an *R*-tree. There may also be specific cases where the converse of ordering proves advantageous with an octree over an *R*-tree. Future work will evaluate these. Additionally, better storage and indexing of 3D pointcloud data may better enable web dissemination of substantial LiDAR datasets. Finally, recently proposed alternative solutions explore cluster and parallel computing (Hongchao and Wang, 2011; Guan and Wu, 2010). While beyond this paper's scope, combining such techniques with the approach introduced herein represents an exciting future research direction.

References

Arens, C., Stoter, J., van Oosterom, P., 2005. Modelling 3D spatial objects in a Geo-DBMS using a 3D primitive. Computers & Geosciences 31 (2), 165–177.

Bayer, R., 1971. Binary *B*-trees for virtual memory. In: Proceedings 1971 ACM SIGFIDET Workshop on Data Description, Access and Control. San Diego, California, pp. 219–235.

Bentley, J.L., 1975. Multidimensional binary search trees used for associative searching. Communications of the ACM 18 (9), 509–517.

Boubekeur, T., Heidrich, W., Xavier, G., Christophe, S., 2006. Volume–surface trees. Computer Graphics Forum 25 (3), 399–406.

Breunig, M., Zlatanova, S., 2011. 3D geo-database research: retrospective and future directions. Computers & Geosciences 37 (7), 791–803.

De Floriani, L., Facinoli, M., Magillo, 2010. Spatial indexing on tetrahedral meshes, In: Proceedings 18th ACM SIGSPATIAL International Conference on. Advances in Geographic Information Systems (ACM SIGSPATIAL GIS 2010), San Jose, CA, USA pp. 506–509.

Egenhofer, M.J., Franzosa, R.D., 1991. Point-set topological spatial relations. International Journal of Geographical Information Systems 5 (2), 161–174.

Guan, X., Wu, H., 2010. Leveraging the power of multi-core platforms for large-scale geospatial data processing: exemplified by generating DEM from massive LiDAR pointclouds. Computers and Geosciences 36 (10), 1276–1282, Elsevier, pp.

Guttman, A., 1984. *R*-trees: a dynamic index structure for spatial searching. In: Proceedings 1984 ACM SIGMOD International Conference on Management of Data, NY, USA, pp. 47–57.

Hongchao, M., Wang, Z., 2011. Distributed data organization and parallel data retrieval methods for huge laser scanner pointclouds. Computers and Geosciences 37 (2), 193–201.

Hinks, T., Carr, H., Laefer, D.F., 2009. Flight optimization algorithms for aerial LIDAR capture for urban infrastructure model generation. Journal of Computing in Civil Engineering 23 (6), 330–339.

Hua, L., Zhengdong, H., Qingming, Z., Peng, L. 2008. A database approach to very large LiDAR data management. In: The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, XXXVII Part B1 Commission I, Beijing, China, pp. 463–468.

Koo, Y.-M., Shin, B.-S. 2005. An efficient point rendering using octree and texture lookup. In: Proceedings International Conference on Computational Science and Its Applications-ICCSA 2005, Lecture Notes in Computer Science 3482, pp. 1187–1196.

Kothuri, R.K., Ravada, S., Abugov, D., 2002. Quadtree and *R*-tree indexes in Oracle Spatial: a comparison using GIS data, In: Proceedings ACM SIGMOD International Conference on Management of Data, Madison, WI, USA, pp. 546–557.

Kothuri, R., Godfrind, A., Beinat, E. 2007. Pro Oracle Spatial for Oracle Database 11g. APresss, USA, 824pp.

Laefer, D.F., Pradhan, A.R., 2006. Evacuation route selection based on tree-based hazards using LiDAR and GIS. Journal of Transportation Engineering 132 (4), 312–320, ASCE Press.

Laefer, D.F., Truong-Hong, L., Fitzgerald, M., 2011. Processing of terrestrial laser scanning pointcloud data for computational modelling of building facades. Recent Patents in Computer Science 4 (1), 16–29.

Li, J., Jing, N., Sun, M., 2001. Spatial database techniques oriented to visualization in 3D GIS. In: Proceedings 2nd International Symposium on Digital Earth, Fredericton, New Brunswick, Canada, 15 pp.

Morton, G.M., 1966. A Computer Oriented Geodetic DataBase and a New Technique in File Sequencing. IBM Technical Report, Ottawa, Canada.

Sagan, H., 1994. Space-Filling Curves. Springer-Verlag, New York 208pp.

Samet, H., 2006. Object-based and image-based image representations. In: Samet, H., Palmeiro, A. (Eds.), Foundations of Multidimensional and Metric Data Structures. Morgan-Kaufmann, San Francisco, USA, pp. 211–220.

Schön, B., Laefer, D.F., Morrish, S.W., Bertolotto, M., 2009a. Three-dimensional spatial information systems: state of the art review. Recent Patents on Computer Science 2 (1), 21–31.

Schön, B., Bertolotto, M., D.F. Laefer, 2009b. Storage, manipulation, and visualization of LiDAR data. In: Remondino, F., El-Hakim, S., Gonzo, L. (Eds.) Proceedings of 3rd International Workshop, 3D-ARCH'2009: 3D Virtual Reconstruction and Visualization of Complex Architectures, Trento, Italy, International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XXXVIII-5/W1, ISSN:1682-1777.

Stanzione, T., Johnson, K., 2007. GIS Enabled Modeling and Simulation (GEMS). In: Proceedings 2007 ESRI International User Conference, San Diego, California, USA. 14 pp. 〈http://proceedings.esri.com/library/userconf/proc07/papers/papers/pap_1115.pdf〉 [accessed June 10, 2011].

Van Oosterom, P., 1990. Reactive Data Structures for Geographic Information Systems. Ph.D. Dissertation, Leiden University, The Netherlands, 222 pp.

Wang, J., Shan, J., 2005. Lidar Data Management with 3-D Hilbert space-filling Curve. In: Proceedings ASPRS 2005 Annual Conference, Baltimore, USA. 7 pp. 〈http://www.digitalearth-isde.org/cms/upload/2007-07-27/de_a_064.PDF〉 [accessed June 10, 2011].

Zlatanova, S., Rahman, A.A., Shi, W., 2004. Topological models and frameworks for 3D spatial objects. Computers and Geosciences 10 (4), 419–428.

Zhu, Q., Gong, J., Yeting, Z., 2007. An efficient 3D *R*-tree spatial index method for virtual geographic environments. ISPRS Journal of Photogrammetry and Remote Sensing 62 (3), 217–224.