ABSTRACT NON-CLUSTERED SPATIAL INDEXES FOR POINT CLOUDS

September 2012

Kyle Martin
Geospatial computer scientist
kyle.martin@mosaicsgis.com

Keywords: point clouds, spatial index, LAS, spatial partition, lidar, laser scanning, non-clustered spatial index

Abstract:

Laser scanners have always produced large datasets that require some form of spatial partitioning for efficient use. Recent advancements in the lidar industry illustrate that spatial partitions are evolving to handle higher point densities and to provide better support of new laser scanning technologies. However, the speed at which existing lidar software systems can adopt new spatial partition advancements is sluggish. The sluggish adoption of spatial partition advancements can be attributed primarily to the spatial partitioning in the core software, which when written for a specific niche market is not written abstractly thus not extendible. We propose a non-clustered spatial index abstraction that allows a point cloud engine to use multiple spatial indexes that may be tailored to different users or application needs without code modifications within the application or the point cloud engine. We then conducted a case study to validate the quality of the non-clustered spatial index abstraction within an implementation of the LAS file format and a quad tree spatial index. The results found that when the non-clustered index architecture is coupled with a quality point cloud database implementation, the spatial index architecture does not limit the overall performance. The spatial index abstraction is a step towards more optimum design strategies to allow lidar software systems to consume innovations in spatial partitioning without expensive and lengthy changes to the software core or applications themselves.

Introduction

Laser scanners have always produced large data sets. Since the inception of laser scanning, large datasets have placed significant and even over-powering demands on software systems particularly existing CAD and GIS applications (Fowler, 2007; Romano, 2007). To a large extent, limitations still remain prominent within CAD and GIS systems despite an overwhelming need to integrate the benefits of laser scanner data with traditional CAD and GIS datasets. Among many of the challenges imposed by laser scanner data, the lack of adequate spatial partitioning plays a significant role in the limitations of any system that uses laser scanner data.

The voluminous nature of laser scanner data has always governed that the use of spatial partitioning was required for the successful processing of datasets. Whether the need was for accelerating visualization, or for quick spatial searches of points in algorithm development for filters and extraction, spatial partitioning was and is a required component for efficient use of the data. Examples of specific uses of spatial partitions in the literature for extraction of features include Sohn et al. (2008) who used binary space partition trees to construct building models from the point cloud, and the segmentation of point clouds into coplanar point clusters using Octree based methods by Wang, and Tseng (2010).

Recent advances in laser technologies are producing more dense and complex point clouds, which have been found to produce better results and higher accuracies of algorithms that filter or extract information (Sohn et al., 2008; Sampath, 2007; Solberg, 2006). With these advances in laser technology and the benefits of higher point densities to applications, come larger database sizes. Higher point densities and database sizes will likely increase a reliance on quality spatial partitioning and likely drive innovations in spatial partitioning. Recent innovations involve Gong, et al. (2012) who created a hybrid spatial index from the R-Tree and Octree structures to handle the challenges of high point densities in mobile laser scanning data. Gong et al. (2012) called this new index structure the 3DOR-Tree which also was extended to consider level of details (i.e., LoDs). Another recent study by Mosa et al. (2012) involved making improvements to the Octree based index structure which out-performed an existing R-Tree index structure within a commercial spatial database. Mosa et al. (2012) also described alternative index approaches where indexes can be populated with attributes or semantics of the point records other than just the spatial information (i.e., generic multivariate indexing). An example would include grouping point clouds based on color information to provide efficient look ups of points based on queries involving spectral inputs.

There have been numerous systems developed over the past decade that sufficiently process point clouds (Romano, 2007; author experience). And to a certain extent are architected much differently than existing CAD and GIS applications which allow them to manage point clouds efficiently whether for visualization, or the execution of filtering or extraction algorithms. However, the outcome of these systems result in spatial partitioning that is hard wired into the system, thus preventing easy adoption of innovations in spatial partitioning such as the work by Gong et al. (2012) or Mosa et al. (2012). This not only effects how well the application may consume higher point densities but also how well a point cloud engine can be calibrated for specific applications. Applications typically drive the specific type of spatial index and the chosen spatial index may be calibrated even further using parameters. It can be easily understood that one spatial index structure cannot always be well suited for all applications, and given the breath of applications that point clouds impact a more flexible, extendible, and abstract architecture is optimum.

The main objective of this paper is to define an interface between a point cloud database and spatial partitions so that multiple implementations of spatial indexes can be used against a single point cloud database (i.e., the database does not require specific ordering or reordering to conform to the sequence of the index as defined by a clustered design). This abstraction between point cloud and index allows injections of newer spatial partition technologies without any change to other parts of the software or application layer. This transparency has obvious benefits where applications can take advantage of different spatial indexes which could boost overall performance for the end user.

The paper is concluded with a case study that examines the performance and running time of the non-clustered index architecture. The case study used an implementation of the architecture for a quad tree index matched with an implementation for LAS files. The study used a moving window operation to examine the performance and running time characteristics of the architecture. Performance metrics were compiled for varying query sizes and for varying file sizes.

Conceptual Framework

Conceptually the relationship between the point cloud database and a spatial index is simple. Consider the following function:

$$P_R = f(R) \qquad (1)$$

where R is a bounding region of the point record variable(s) used to construct the index and P_R is a set of objects organized by the index. Ideally, an index given a region R can return a set of objects efficiently. We refer to regions as ranges of the variables used to construct the index. Although, a well-designed system would follow such a generic structure, we will only consider for the remainder of this paper an index in the traditional meaning where coordinates in $\mathbb{R}^n, where\ 2 \leq n \leq 3$ space are used to construct the index (i.e., a spatial index is a special kind of multivariate index).

The elements in set P_R that are returned from the spatial index can be pointers to objects, memory addresses, or unique identifiers to spatial entities in a database. Pointers to objects and memory addresses are values that would be included in a clustered index (i.e., the spatial entities are stored inline in the index data structure). The inline storage of the spatial entities in the index would be contrary to the objectives of this paper because multiple index implementations would not be able to be applied on a single database. Therefore, pointers to objects and memory addresses will not be considered further in this study. Unique identifiers, i (i.e., non-clustered index strategy) are used as elements in P_R. A non-clustered index simply stores pointers to the spatial entities as unique identifiers therefore multiple indexes could exist with different organizations of unique identifiers without modification of the underlying point cloud.

The unique identifier of a spatial entity is defined by the point cloud implementation and is obtained by the index when bulk loading (i.e., creation and insertion of the points) of the index takes place. It is up to the point cloud implementation to define the unique identifier and examples would include the sequential number of the point in the database, or the byte offset to the beginning of the point structure in the database.

The P_R returned from an index is the set of unique identifiers that satisfy R. A point cloud implementation would iterate the set of unique identifiers, i in P_R, and perform point look-ups (PL) for each i to obtain the point record address or object (i.e., $PL_i\!: \forall\, i\ \in\ P_R$).

Case Study

The spatial index abstraction defines the interaction between spatial index and point cloud using a set of pointers (i.e., P_R) or numbers that uniquely identify the point records (i.e., i) that satisfy a region (i.e., R). For each i in P_R, a point look-up (PL) is performed to obtain the point record in a point cloud database. The point look-up using the unique identifiers is a fundamental difference between a clustered and non-clustered index. Given the quantity and magnitude of the potential point look-ups for a query, one would question how well an implementation of the architecture could perform. As an additional step, would the point look-up operation ultimately limit the performance or exhibit poor growth characteristics in terms of increasing query or file size?

Test Data

A series of 12 LAS files were obtained from the USGS Click web site (USGS Click, 2012) in Somerset County, Pennsylvania, USA. The content of the files would be described as rural covered mostly by deciduous forests and open farmland. The files are identified in Table 1. A series three of datasets was constructed from the original files to be used in the tests as files of increasing size such that $E_{d_0} \cap\ E_{d_1}$ and $E_{d_1} \cap\ E_{d_2}$, where E_{d_n} is the extent or bounding box of the nth dataset. The three datasets are outlined in Table 2. Each dataset (d_0, d_1, d_2) was indexed using a quad tree with 10, 11, and 12 maximum levels respectively. The balance count used for each index was 100.

Table 1: Las files used in the case study (CRS: NAD_1983_StatePlane_Pennsylvania_South_FIPS_3702_Feet)

Variable Name	File name
f0	PA_Statewide-S_2007_16152S082007
f1	PA_Statewide-S_2007_16153S082007
f2	PA_Statewide-S_2007_16154S082007
f3	PA_Statewide-S_2007_16155S082007
f4	PA_Statewide-S_2007_17152S082007
f5	PA_Statewide-S_2007_17153S082007

f6	PA_Statewide-S_2007_17154S082007
f7	PA_Statewide-S_2007_17155S082007
f8	PA_Statewide-S_2007_18152S082007
f9	PA_Statewide-S_2007_18153S082007
f10	PA_Statewide-S_2007_18154S082007
f11	PA_Statewide-S_2007_18155S082007

Table 2: Dataset series used in the tests

Variable Name	File size (KB)	Point Count	Extent (feet)	Original Files
d_o	137,777	5,038,593	10000 X 10000	*{f5}*
d_1	571,906	20,915,364	20000 X 20000	*{f0, f1, f4, f5}*
d_2	1,789,695	65,451,632	30000 X 40000	*{f0 : f12}*

Test Methods

A moving window mechanism was used over a selected region of the test data. The selected region was randomly chosen from d_0 in order to insure overlap with d_1, d_2. The random extent was sized to contain approximately 100,000 points (based on average point density). The random extent used in the test was as follows in units of feet: x_{min} = 1530447, y_{min} = 161420, x_{max} = 1531857, y_{max} = 162830. The points within the random extent were used as the focal points for the moving window operation. For each point from d_1 in the random extent, the set of points within a square window was iterated. The window or query size of 25, 50, and 100 feet were used. The same test was then executed using a constant window size of 50 feet on each dataset: d_0, d_1, d_2.

Each executed test collected the total time in milliseconds (T), total accumulated time attributed for point lookups (PLT), total outer points, and the total accumulative number of points (Table 3, and 4). One additional derived field was computed as the ratio of point look-up time to total time (i.e., *PLT/T*).

Each executed test was preceded by a cold boot of the computer (Dell Precision T1500, Intel Core i5 CPU 2.67 GHz, 4GB RAM, Windows 7 64 bit OS), and used a single core. All code (test program, index and point cloud libraries) was written in C++ by the author, and was compiled in the 64 bit architecture (CloudyDay, 2012). The test data was located on a local drive (i.e., SATA, 7200 rpm) during the tests.

Spatial Index Implementation

An implementation of the spatial index abstraction was performed for a quad tree index. The quad tree index implementation required two inputs from the user when building indexes for LAS files: the maximum number of tree levels, and balance count. File based storage was used to persist the index between uses of the LAS files. The running time analysis of a spatial find (i.e., a bounding box around a single point) operation is *O(n)*, where n is the number of levels in the index (Goodrich et. al., 2004). The memory footprint of the index file based data structure is *O(nN + nMb + nMb$_L$)*, where *nN* is the number of nodes in the index, *nMb* is the number of memory blocks or pages for the index file, and *nMb$_L$* is the number of loaded memory blocks or pages. *nMb* is determined by the number of nodes or bytes to assign to each memory block. *nMb$_L$* is governed by a caching system and is typically capped at a user specified level such as 256 MB.

LAS File Implementation

The spatial index abstraction was integrated into an implementation for LAS files. The running time analysis for a single pointer is worse case *O(log nMb)*, where *nMb* is the number of memory blocks or pages for the LAS file. The majority case running time for a single pointer would be characterized as *O(1)* because the pointer would be located on the same memory block as the current point record in the iterator, thus preventing a look up of the memory block. The degree to which a single pointer would be found in *O(1)* is determined by the amount of fragmentation of the points in the memory blocks with respect to the order they are retrieved by an index. The memory footprint analysis for the LAS file implementation was *O(nMb + nMb$_L$),* where *nMb* is the number of memory blocks or pages for the LAS file, and *nMb$_L$* is the number of loaded memory blocks or pages. *nMb* is determined by the number of points or bytes to assign to each memory block. *nMb$_L$* is governed by a caching system and is typically capped at a user specified level such as 512 MB. The LAS file implementation used the byte offset (starting from the point offset in the header) to the beginning of each point record for the unique identifier.

Results

A moving window operation was performed on points (i.e., focal points) from a randomly generated extent that overlapped each of the test datasets. The randomly selected extent contained 82,836 points. The moving window tests were performed by file size, and window size and the results are shown in Tables 3 and 4 respectively.

Table 3: Moving window test by file size for 50 foot window size. The number of focal points was 82,836.

Dataset	Total Time (*T*) in ms	Point Lookup Time (*PLT*) in ms	Total Point Count	*PLT /T*
d_0	2,246	390	27,917,130	0.174
d_1	2,309	420	27,935,452	0.182
d_2	2,090	358	24,843,333	0.171

Table 4: Moving window test by window size using dataset d_1. The number of focal points was 82,836.

Window Size (Feet)	Total Time (T) in ms	Point Lookup Time (PLT) in ms	Total Point Count	PLT/T
25	1,326	279	14,432,497	0.210
50	2,216	343	27,935,452	0.155
100	4,306	941	67,755,200	0.219

Table 3 shows the results for increasing file sizes. Despite file size increases of 4 and 13 times larger than d_0 (Table 2), the point look up times showed no significant indication of comparable increases for results using d_1, and d_2. Similarly, there was no indication of increases in the proportion of time attributed to the point lookups *(PLT/T*) for the results using d_1, and d_2.

Table 4 shows the results for increasing window sizes. Similar to the results in Table 3, the proportion of time attributed to the point lookups *(PLT/T*) appeared to remain steady. The decrease in *PLT/T* for window size 50 appears to be low and the exact same query in Table 3 (i.e., results for d_1) shows a value of 0.182 for *PLT/T*. The point lookup time increases as window size increases but appears to be proportional to the total point count. Examination of the proportional increases from window size 25 to 50, and 50 to 100 results in a 123% and 274% increase in time respectively. Total point counts for window size increases of 25 to 50, and 50 to 100 increased by 194% and 243% respectively. The expectation is that when more points are iterated (i.e., more elements in *P*), the total time would increase linearly. The results suggest that there is a less than linear increase in total time attributed to the point lookup as window size increases from 25 to 50 (i.e., 123% < 194 %). However, there appears to be some inefficiency since there was 274% increase in the time attributed to point look-up as opposed to a 243% increase in total points between 50 and 100 foot window sizes.

Discussion

Abstract non-clustered spatial index architecture was constructed to allow a point cloud database to use any spatial index transparently. The main mechanism used for the point cloud database to query a spatial index was defined as a set of pointers where a pointer was an integer that uniquely represented a point record in the database *(1)*. The non-clustered architecture was included in an implementation for a quad tree index and LAS files. The performance of the architecture was examined using a moving window operation. The test was performed by file size and window size. The results are shown in Tables 3 and 4.

The *PLT/T* values in both tables show that the proportion of point look up times to total time remains constant as file size, and query size increases. The time attributed to the point look-up operations are approximately 18.5% of the total time, leaving the remaining time (i.e., ~81.5%) attributable to the index implementation. Although the remaining time is directly dependent on the specific index implementation, this strongly suggests that the architecture is not a limiting factor of the performance. These results allow for significant improvement of the indexing technology used in the tests and likely other technologies and implementations before the point look-up operation limits the performance significantly.

The point look-up times did not show any evidence of increase as the file size increased. This characteristic certainly illustrates the quality of the LAS file implementation used in the study. It was expected to have some increase in the point look-up times as file size increased especially when there is a 13 time change in file size. The running time analysis for the LAS file implementation is a worst case *O(log nMb)*, where *nMb* is the number of memory blocks or pages for the file. The *nMb* for each dataset (*d0 – 268, d1 – 1118, d2 – 3496*) would suggest a small increase in running time, however it was quite evident that the majority was *O(1)* which allowed the tests to show very little growth in the point look-up running times.

The point look-up operation also exhibited positive characteristics as query size increased. The total point count increased as the query or window increased, and the ratio between a window size's total point count and the total point count for the next smaller window size should describe the expected increase in the point look-up times. However, the results were mixed where window size increases from 25 to 50 feet appeared to follow a less than linear model (i.e., 123% < 194%) and the window size increase from 50 to 100 foot appeared to be greater than linear (i.e., 274% > 243%). The unexpectedly large increase in point look up running time could indicate possible inefficiencies in the underlying implementations. But if the results from the window size 25 to 100 foot are used then a 337% increase in point look-up time and 469% increase in total point counts was yielded suggesting that a less than linear growth occurred for both increases in window size. Despite the conflicting results in increases of window size, there was strong evidence of the indication of the use of constant time when performing point look-up operations.

The running time of the LAS file implementation are represented well in the results. It is evident that the use of *O(1)* for point look-up operations is quite apparent in the results. The degree to which *O(1)* is used for point look-up operations is determined by the amount of fragmentation in the LAS file. Fragmentation occurs when the sequence of point records stored in a file, does not match the sequence of unique identifiers from an index. This results in skipping over bytes in either direction in the file possibly numerous times, thus resulting in inefficiencies when locating point records. Fragmentation can only negatively affect the performance of the queries. However, implementations may react differently depending on the choice of file IO technology, and one implementation may be more or less sensitive to fragmentation than other implementations.

As a matter of curiosity, two metrics were conceived to measure the amount of fragmentation between an index and an LAS file. The metrics were the byte distance between each unique identifier, and the number of memory block revisits per query. The byte distance for the tests presented in both Table 3 and 4 was approximately 7,304 bytes (i.e., ~261 points at 28 bytes per point record), and did not significantly differ between the varying file size, and query size tests. The memory block revisits was approximately 1.2 for the varying file size tests, and the memory block revisits had very little variability as file size increased. The memory block revisits for the varying window sizes were 0.37, 1.09, and 3.58 for the 25, 50, and 100 foot window sizes respectively. This strongly suggests that as the window size increases there is an increasing degree of fragmentation between the index and the LAS file. The 3.58 memory block revisits for the window size test of 100 foot would result in slightly poorer performance as the point cloud implementation would require nearly 4 revisits to the same memory block per query (i.e., one execution of the window query for a focal point). The only evidence to suggest poor performance for window size of 100 foot query was the unexpected increase in point look-up times (i.e., 274%) compared to the total point count increase from 50 to 100 foot (i.e., 243%). Perhaps the increase of the window size from 50 to 100 feet crossed a threshold or size for which points are stored or grouped in the file as it was obtained from the source. It would not be uncommon to experience LAS files that have point records organized into square boxes for a given size that is conducive for processing of the data during the lifecycle of the laser scanning project. It appears as if the window size of 100 feet for this particular data has frequent crossings or intersections with these special sequences of point records in LAS files.

It is unclear what effect fragmentation had on the overall performance presented in Tables 3 and 4. If fragmentation in terms of 7,304 bytes between each unique identifier and non-zero memory block revisits did not exist, would the performance increase, and if so by how much? What if fragmentation was worse, would the performance decline and by how much? Answers to these questions are outside of the scope of this paper, but would likely differ between implementations depending on factors such as the file IO mechanisms used internally. Different results could also be yielded based on hardware differences specifically the drive technology that the LAS files are tested on. Despite these challenges, improvements of performance levels even small ones are significant and worth further study and investigation.

The general formula given in Eq. *1* can be extended to specializations of indexes such as elements in set P_R that are the n-nearest neighbors of a point or indexes that have LoD's. Consider that R in Eq. *1* is a definition of an orthographic or perspective projection and that P_R contains elements that are optimum for the viewing projection. These specializations are attractive because improved indexes with LoD's can be consumed without change to the software or application layers. However, without reordering the database (i.e., at least with respect to LAS files) large degrees of fragmentation would be introduced resulting in a complete traversal of the LAS file to obtain the top (i.e., coarsest level) LoD in the index. Unfortunately, a complete traversal of the file completely negates the value of the spatial index. A possible work around is to perform a reordering of the LAS file then apply other non-LoD indexes to the reordered LAS file. The work around does warrant further investigation because the points that comprise the LoD's, will be significantly out of order introducing fragmentation between the index and LAS file.

Conclusion

A system that uses abstract spatial index architectures poses a way for applications and algorithms to be calibrated and optimized. Optimization can be performed by modifying a spatial index or even by adding new types of spatial indexes that underlines the spatial query processing power of a point cloud implementation. The optimization can occur without code modifications by changing index properties to better match the needs of an application, or with code modifications by implementing new index algorithms or implementing existing algorithms better.

In this paper, we developed non-clustered spatial index architecture to facilitate the ability to match different indexes with the same point cloud database. The abstraction defines the interaction between spatial index and point cloud using a set of pointers or numbers that uniquely identify the point records that satisfy a region. A case study was performed that used the architecture within a quad tree and LAS file implementation. The case study compiled performance metrics (Tables 3 and 4) using a moving window operation over a random extent within varying file, and window sizes. The performance metrics strongly suggested that the architecture did not limit the overall performance and that significant gains in index performance could be achieved before the architecture would be found to limit the overall performance. The results also showed that when coupled with a quality point cloud implementation such as the one used in the case study, performance does not decline by file or query size. These results are positive signs that make non-clustered spatial index architectures a smart choice for allowing lidar software systems to consume innovations in spatial partitioning without expensive and lengthy changes to the software core or applications themselves.

References

CloudyDay, 2012. *The CloudyDay SDK*. <http://www.mosaicsgis.com>.

Fowler, Robert A., A. Samberg, M. J. Flood, and T. J. Greaves, 2007. Topographic and Terrestrial Lidar, *Digital Elevation Model Technologies and Applications: The DEM Users Manual* (2nd ed.). Bethesda, Maryland: American Society for Photogrammetry and Remote Sensing.

Goodrich, Micheal T., Roberto Tamassia, and David Mount, 2004. Trees and Search Trees, *Data Structures and Algorithms in C++*. John Wiley and Sons, Inc., Hoboken, New Jersey (252 – 301, 411 – 475).

Gong, J., Q. Zhu, R. Zhong, Y. Zhang, and X. Xie, 2012. An efficient point cloud management method based on a 3d R-tree, *Photogrammetric Engineering & Remote Sensing*, 78(4):373 - 383.

Mosa, Abu Saleh Mohammed, Bianca Schon, Michela Bertollotto, and Debra F. Laefer, 2012. Evaluating the Benefits of Octree-based Indexing for Lidar Data, *Photogrammetric Engineering & Remote Sensing*, 78(9):927 - 934.

Romano, Mark E., 2007. Lidar Processing and Software, *Digital Elevation Model Technologies and Applications: The DEM Users Manual* (2nd ed.). Bethesda, Maryland: American Society for Photogrammetry and Remote Sensing.

Sampath, Aparajithan and Jie Shan, 2007. Building Boundary Tracing and Regularization from Airborne Lidar Point Clouds, *Photogrammetric Engineering & Remote Sensing*, 73(7):805 - 812.

Solberg, Svein, Erik Naesset, and Ole Martin Bollandsas, 2006. Single Tree Segmentation Using Airborne Laser Scanner Data in a Structurally Heterogeneous Spruce Forest, *Photogrammetric Engineering & Remote Sensing*, 72(12):1369 - 1378.

Sohn, Gunho, Xianfeng Huang, and Vincent Tao, 2008. Using a Binary Space Partition Tree for Reconstructing Polyhedral Building Models from Airborne Lidar Data, *Photogrammetric Engineering & Remote Sensing*, 74(11):1425 – 1438.

USGS Click, 2012. Somerset County, Pennsylvania, USA. August 2012 < http://lidar.cr.usgs.gov>

Wang, Miao, Yi-Hsing Tseng, 2010. Automatic Segmentation of Lidar Data into Coplanar Point Clusters Using an Octree-Based Split-and-Merge Algorithm, *Photogrammetric Engineering & Remote Sensing*, 76(4):407 – 420.