MOBILE TERRESTRIAL LIDAR DATA-SETS IN A SPATIAL DATABASE FRAMEWORK

Conor P. Mc Elhinney, Paul Lewis and Timothy McCarthy

National Centre for Geocomputation (NCG),
National University of Ireland Maynooth (NUIM),
Maynooth, Co. Kildare, Ireland
conormce@cs.nuim.ie, paul.lewis@nuim.ie
http://www.eursi.com/

KEY WORDS: LIDAR, other

ABSTRACT:

Mobile Mapping Systems (MMS) have become important and regularly used platforms for the collection of physical-environment data in commercial and governmental spheres. For example, a typical MMS may collect location, imagery, video, LiDAR and air quality data from which models of the built-environment can be generated. Numerous approaches to using these data to generate models can be envisaged which can help develop detailed knowledge in the monitoring, maintanence and development of our built-environment. In this context, the efficient storing of this raw spatial data is a significant problem such that bespoke and dynamic access is possible for the generation of modeling requirements. This fundamental requirement of managing these data, where upwards of 40 gigabytes per hour of spatial-information can be collected from an MMS survey, poses significant challanges in data management alone. Existing methodologies mantain bespoke, survey oriented approaches to data management and model generation where the original MMS spatial data is not generally used or available outside these requirements. Thus, there is a need for an MMS data management framework where effective storage and access solutions can hold this information for use and analysis in any modeling context. Towards this end we detail our storage solution and the experiments where the procedures for high volume navigation and LiDAR MMS-data loading are analysed and optimised for minimum upload times and maximum access efficiency. This solution is built upon a PostgreSQL Relational Database Management System (RDBMS) with the PostGIS spatial extension and pg_bulkload data loading utility.

1 INTRODUCTION

Infrastructural mapping and monitoring has become an integral part of the academic, commercial and governmental sphere where detailed knowledge of the built environment is easily accessible. To this end Mobile Mapping Systems (MMS) play an important role in generating these environment-model data sources. They are particularly suited to the road-network infrastructural management case, as multiple environmental modelling sensors can be integrated, calibrated and transported on a single collection platform. Typically, high accuracy near 3D geospatial data can be recorded from which detailed, bespoke and comparative analysis can be performed in order to monitor, understand and plan a road-networks status and/or requirements. This paper concerns the LiDAR data collected from an MMS van which is equipped with a Global Positioning System (GPS), an Inertial Navigation Sensor (INS), six progressive scan cameras, a Light Detection and Ranging (LiDAR) unit, a multispectral camera and a thermal-imaging camera. Upwards of 40 gigabytes of data per hour can be acquired from this MMS platform with the average hourly LiDAR data file outputs containing 18 million points.

When dealing with large datasets such as MMS data, it is not sufficient to deal with the storage, access and subsequent processing of this data in isolation. These three elements in MMS data handling are interlinked and require a holistic approach. We are building a mobile mapping data framework based around a spatial database management system (SDBMS). In this paper we will deal with our initial investigations into the storage, access and processing of LiDAR data in a SDBMS with particular focus on the efficient uploading and storage of this data. Both the storage and post-survey processing of these data present a number of computing challenges because of the high volumes of detailed geospatial information involved. However, it is the storage and accessing of these data that is particularly problematic as no existing integrated framework solution can exploit not only vast data sets such as LiDAR, but also the broader spectrum of spatial information that is being collected, for example video.

Towards this requirement there exists a significant desire to store vast 3D spatial data in a database management system (DBMS) (Schön et al., 2009, Nandigam et al., 2010, Breunig and Zlatanova, 2011). DBMSs offer transaction guarantees and multi-user, random access of potentially very large datasets, in addition to advanced features, such as back-up and restore capabilities. However, the typical work flow with regards to LiDAR often does not provide the user with the actual raw LiDAR data. Instead, users have to decide on a format for the data that they wish to perform certain analysis on, for example a Digital Elevation Model (DEM). For the MMS context in particular, there typically exist two vast 3D point data sets: one is the navigation points that describe the GPS track of the MMS throughout the survey, and the other being the actual LiDAR survey point cloud. Preserving this information has the potential to empower several queries, where the collection of navigation points can be employed in order to describe the actual LiDAR data set. Consequently, users are currently prohibited from exploiting the full range of opportunities that typical MMS surveys offer. As a result Spatial DBMSs (SDBMSs) appear particularly attractive in this context.

However, with regards to the storage of LiDAR data while DBMSs have been used in this context, (Schön et al., 2009, Nandigam et al., 2010, Sharma et al., 2006, Rottensteiner et al., 2005, Chen et al., 2010, Ming et al., 2009), no significant solution currently exists to support this approach over and above existing LAS file format solutions. A number of methodologies have been proposed that attempt to define a comprehensive LiDAR-data management framework, (Ferede et al., 2009, Hongchao and Wang, 2011, Hongchao and Wang, 2011), where the storage, access and analysis of these data are defined and all propose at some level the requirement for a database core. In (Nandigam et al., 2010), it is suggested that alternative support that includes LAS file formats should form an integral part of their implementations where they only store metadata related to the point data in the DBMS, while the actual data remains stored across several files. However, retrieving bespoke sets of raw point cloud data using this methodology is not optimal when spatial constraints are required. In, (Jaeger-frank et al., 2006), a Grid computing solution is considered while in, (Mumtaz, 2008), an object-relational database solution is presented as part of the CityServer 3D application. (Sharma and Parikh, 2008, Parikh et al., 2004b, Parikh et al., 2004a) define a web-based LiDAR analysis, experimentation and educational platform built around a MySQL database.

In our approach we use the popular PostGIS SDBMS (2001), which is an implementation and extension of OCG standards and provides a spatial extender to PostgreSQL. PostGIS enjoys wide spread support and substantial integration with GIS software, such as Mapserver, Geotools, FDO and many more. However, the advantages of a system like PostGIS remain relatively unexploited with regards to MMS surveys. Thus, in this paper we discuss the specific problem of inputting large volumes of LiDAR point cloud data into an integrated SDBMS framework for MMS data. We do this by detailing the results of our comparisons in performance scaling for populating PostGIS tables with LiDAR data and building spatial indices. This is achieved by comparing the PostgreSQL's COPY functionality to the pg_bulkload extension, which we have adapted for spatial data in this case. A significant performance increase has already been achieved through our ongoing test implementations.

2 MOBILE MAPPING SYSTEMS DATA

Empirical experience with MMS geospatial data, in particular LiDAR data, suggests that the primary obstacles in the processing of these data is their considerable size and the inability to easily constrain the data based on spatial attributes. Towards a solution to this problem we have implemented a SDBMS approach that in this instance handles the type of MMS data detailed in Table 1. The first stage in our solution is the developement of a platform and methodology where large volumes of data can be stored in a accessible form. However, populating a SDBMS from a data source that is generating, on average, 36GB's of spatial information every hour is no trivial task. In this paper we present the results of our testing the feasibility of batch processing the MMS into a well structured, spatially-indexed database.

Navigation	LiDAR
10kHz - 250kHz	75kHz - 400kHz
GPS Time	GPS Time
Latitude, Logitude, Altutude	Latitude, Logitude, Altutude
Roll, Pitch, Yaw	Pulse Width
	Amplitude, Reflectance
	Target Number
	Scanner X,Y,Z
	Target Type
	Channel Descriptor
	Scanner angles

Table 1: Properties of the spatial data collected from the XP1 MMS survey vehicle.

The survey-processing methodology that prevails in industry standard software suites has proven to be a significant constraining factor in a number of aspects of our LiDAR analysis work, not least the ability to easily segment LiDAR data across a number of different surveys. These suites provide no context for spatial optimisation across numerous surveys, where data segmentation for road-edge detection, for example, can be easily implemented based on where the interest area is rather than which survey it belongs too. Therefore, numerous runs of the same algorithm have to be performed on separate surveys. Alternatively, difficult data-assimilation processes have to be followed to generate the single source data set within these software solutions. This is because the generally accepted commercial approach to using LAS files to store LiDAR on a survey based approach, visualised in Figure 1, does not easily lend itself to spatial segmentation and analysis.

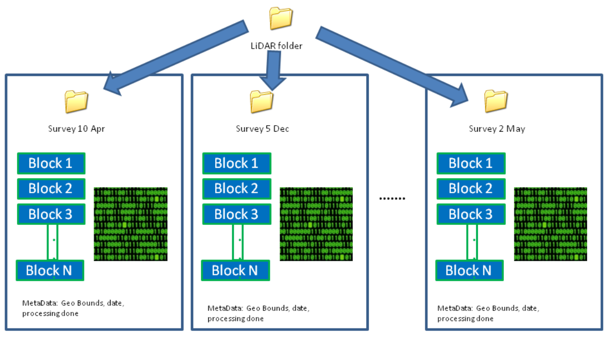


Figure 1: This example of a Survey-based approach for LiDAR data management in commercial software is typical.

However, approaching this problem with a spatial-constraint perspective, it is possible to optimise the LiDAR data being output. This can be achieved through procedures that leverage the power of a platform such as PostGIS and its numerous, integrated, spatial API's. The geo-referenced raw LiDAR is stored in a database where optimised spatial indexes can be generated in order to facilitate efficient querying of the data set. Consequently, optimally located LiDAR data can be output in a user requirements spatial context where use cases, such as the road detection algorithm, can operate on a reduced target data set.

Drawing on the spatial variables collected from an MMS survey, shown in Table 1, a selective LiDAR data segmentation can be generated. Given the known calibration information for the MMS platform a spatial extent query can be performed using the low resolution navigation data to segment the high resolution LiDAR. As an example of segmenting an optimised road surface data set, a bounding box can be constructed where, in the altitude plane, it defines 3D space that is below the GPS track, in the orthogonal plane to the traversal direction, it is extended to a adjustable distance likely to cover beyond the road edge and, in the traversal plane, can extend to an adjustable distance along the GPS track. Based on this bounding box a 3D spatial query can isolate from the larger LiDAR data store all points contained within, thus reducing the amount of points that need to be processed by any process or algorithm that is relevant to LiDAR road-data analysis. Using this selective example it has been shown in (Mc Elhinney et al., 2010, Lewis et al., 2010) that a road-edge detection algorithm can produce results more efficiently, however, and importantly, it can be applied across multiple different surveys much easier.

3 SDBMS ARCHITECTURE

The navigation and LiDAR data described in the previous section are stored in a PostgreSQL relational database management system. Integrated into this database solution are the PostGIS spatial extensions and the pg_bulkload data loading utility. It is through the use of the pg_bulkload utility that LiDAR data input and spatial-index generation have been optimised. Figure 2 gives a broad overview of the MMS data management framework being developed on this platform.

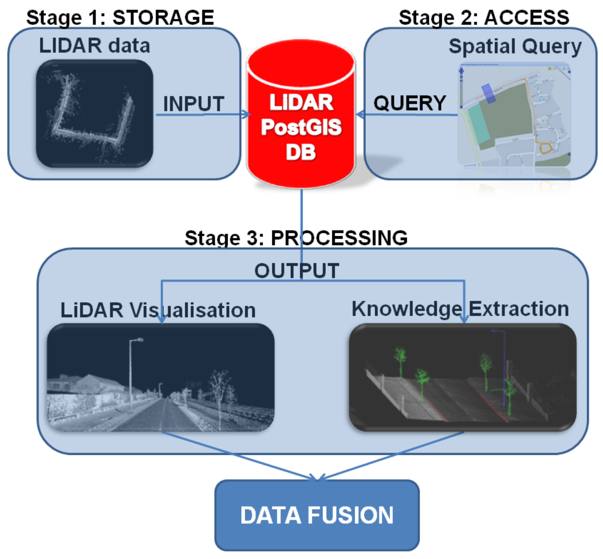
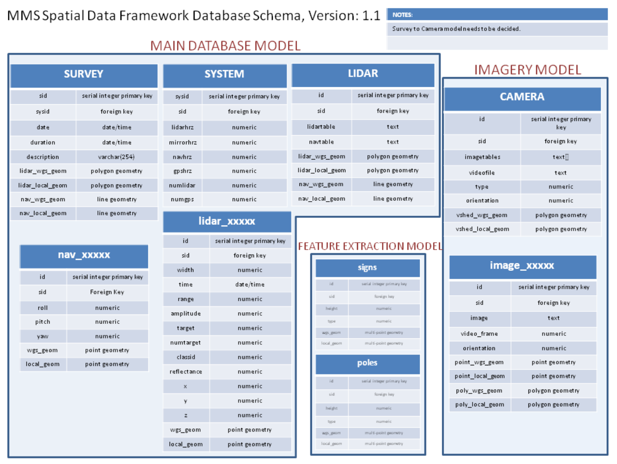


Figure 2: MMS data management framework overview.

While in Figure 3 we provide a more detailed sample from our SDB schema which incorporates a number of levels of spatial detail from which data extraction is optimised.


MMS Spatial Data Framework Database Schema, Version: 1.1

NOTES:
Survey to Camera model needs to be decided.

MAIN DATABASE MODEL

SURVEY	
sid	serial integer primary key
sysid	foreign key
date	date/time
duration	date/time
description	varchar(254)
lidar_wgs_geom	polygon geometry
lidar_local_geom	polygon geometry
nav_wgs_geom	line geometry
nav_local_geom	line geometry

SYSTEM	
sysid	serial integer primary key
sid	foreign key
lidarhrz	numeric
mirrorhrz	numeric
navhrz	numeric
gpshrz	numeric
numlidar	numeric
numgps	numeric

LIDAR	
id	serial integer primary key
sid	foreign key
lidartable	text
navtable	text
lidar_wgs_geom	polygon geometry
lidar_local_geom	polygon geometry
nav_wgs_geom	line geometry
nav_local_geom	line geometry

nav_xxxxx	
id	serial integer primary key
sid	Foreign Key
roll	numeric
pitch	numeric
yaw	numeric
wgs_geom	point geometry
local_geom	point geometry

lidar_xxxxx	
id	serial integer primary key
sid	foreign key
width	numeric
time	date/time
range	numeric
amplitude	numeric
target	numeric
numtarget	numeric
classid	numeric
reflectance	numeric
x	numeric
y	numeric
z	numeric
wgs_geom	point geometry
local_geom	point geometry

FEATURE EXTRACTION MODEL

signs	
id	serial integer primary key
sid	foreign key
height	numeric
type	numeric
wgs_geom	multi-point geometry
local_geom	multi-point geometry

poles	
id	serial integer primary key
sid	foreign key
height	numeric
type	numeric
wgs_geom	multi-point geometry
local_geom	multi-point geometry

IMAGERY MODEL

CAMERA	
id	serial integer primary key
sid	foreign key
imagetables	text[]
videofile	text
type	numeric
orientation	numeric
vshed_wgs_geom	polygon geometry
vshed_local_geom	polygon geometry

image_xxxxx	
id	serial integer primary key
sid	foreign key
image	text
video_frame	numeric
orientation	numeric
point_wgs_geom	point geometry
point_local_geom	point geometry
poly_wgs_geom	polygon geometry
poly_local_geom	polygon geometry

Figure 3: Sample from our MMS spatial database schema.

Using this platform we have shown in (Lewis et al., 2010) how this technology decision has enabled us to optimise the large volumes of LiDAR data for processing by feature extraction algorithms. However, the core component to being able to implement this approach is the ability to use the extensive functionalities provided by PostgreSQL's PostGIS spatial extensions. Through PostGIS we have been able to generate and store Open Geospatial Consortium (OGC) compliant spatial objects in a standard relational database. Extending from this functionality is the spatial indexing options provided by PostGIS. However, while it is trivial to generate a PostGIS spatial object (geometry) and its associated spatial-index, doing so for huge volumes of LiDAR data is not trivial. It is at this point that the need to input these large volumes of spatial data in a efficient manner becomes important. While it has been identified that our access and analysis approach has proven optimal for LiDAR processing, this advantage may be offset by requirements to push the data through very long data uploading processes.

To this end we have completed a set of experiments where we have implemented and tested the built in PostgreSQL bulk data input functionality, the COPY command, against the pg_bulkload bulk data loading utility. The PostgreSQL COPY command moves data from a file into a DB table through a SQL statement. Its performance is better on initial data loading if the destination table is empty. Performance is also improved when the data are loaded without an index constraint; thus dropping and recreating the indexes after loading is optimal. Pg_bulkload provides an alternative method that has been developed by the Nippon Telegraph and Telephone Corporation for PostgreSQL. It is an optimised high volume data loading utility that skips some of the processing overheads used by COPY. It is designed to load huge amounts of data to a database where you can choose whether database constraints are checked, whether errors are ignored during the loading and to have the index updated as a synchronous operation. Pg_bulkload has two operation modes; Direct and Parallel. Direct uses one core of the system to upload the data while parallel attempts to distribute as much processing across the systems cores. On servers this should increase the efficiency of pg_bulkload. The following sections detail the experiments we performed and the analysed results achieved.

4 EXPERIMENTS

We selected four files and three different uploading strategies to compare COPY and pg_bulkload. For the four files, we first selected a large dataset over 6GBs in size and a small dataset over 800MBs in size. We create two files from each, as shown in Table 2. The first file has 10 columns while the second file has 14 columns. By keeping the data constant and only changing the row size we intended to examine the effect row size had on the uploading approaches. Our server is a Dell PowerEdge with 32GBs of RAM, 64-bit Ubuntu Server, a 16-core Intel(R) Xeon (R) 2.8GHz X5560 and 7TBs of storage.

In the first experiment we compared like with like. Both COPY and pg_bulkload followed the same process for uploading where the spatial index is created from the table after all the data has been uploaded. This is the most inefficient approach. This process was:

1. Create the Table - A empty table is created from the input file header fields.
2. Load Data - Populate the whole table with all the raw LiDAR file data.
3. Create Geometry Column - Add a PostGIS POINT-geometry data type field to the table.
4. Update Geometry Column - Update the geometry field from the tables raw latitude, longitude and altitude data.
5. Create Index - Create the spatial index on the PostGIS geometry field.
6. Vacuum Table - Flush the table of all empty tuples etc. to reclaim and optimise table size.

In the second experiment, we tested one of pg_bulkloads unique features. It can use an SQL filter to process the data as it is being loaded into the database. It is also one of pg_bulkload's constraints as it cannot load data from a file that does not have a structure exactly the same as the target table, whereas the COPY command can have its parameters set to reflect file and/or table constraints. This SQL filter option allows us to load a file without a spatial index and create the spatial index using this SQL filter while uploading. In this case the process was:

File	Rows	Cols	Size (MBs)	Avg. Row Size (KBs)	Python (s)	Table Size (MBs)	Index Size (MBs)
La	66,182,260	10	4359.52	0.0675	308.72	9401	3363
Lb	66,182,260	14	6757.94	0.1046	375.82	12000	3362
Sa	8,138,952	10	526.90	0.0663	38.06	1156	520
Sb	8,138,952	14	821.85	0.1034	47.67	1479	426

Table 2: File properties.

1. Create the Table.
2. Create Geometry Column.
3. Create Index.
4. Load Data.

In the third experiment, we selected the most efficient uploading approach we have found for both COPY and pg_bulkload and compared them. For our data we found that by pre-processing the file using python and adding the spatial index column directly to the file as a string we significantly decreased the uploading time. This process involved using the Latitude, Longitude and Altitude fields in the LiDAR data file to concatenate a canonically suitable representation of a PostGIS base-geometry data type. This base data type is an Extended Well Know Text (EWKT) representation, in three dimensional space, of the OGC S̈imple Features for SQLs̈pecification which only defines a two dimensional structure. For COPY the process was:

1. Python - Process the original file to add the PostGIS geometry data type representation into it.
2. Create the Table.
3. Create Geometry Column.
4. Load Data.
5. Create Index.

.

While the process for pg_bulkload differed slightly:

1. Python.
2. Create the Table.
3. Create Geometry Column .
4. Create Index.
5. Load Data.

.

The following section analyses the results from these procedures.

4.1 Results

File	SQL COPY	PG direct	PG parallel
La	98.15	95.99	95.44
Lb	108.77	103.39	103.89
Sa	9.24	8.84	8.84
Sb	10.42	9.64	9.67

Table 3: Total upload time in minutes for Experiment 1.

In Table 3 the runtime results from experiment 1 are presented. This experiment proved to be the slowest procedure in terms of runtime mainly due to the requirements to perform a full VACUUM procedure to optimize disk space once data loading and index creation had been completed. Because of the update procedure that generates the PostGIS geometries a significant amount of redundant database tuples are created which in the case of files La and Lb bloated an optimized 12GB table to 21GB before the VACUUM. Comparatively, however, in all the cases the runtimes for pg_bulkload are faster than the standard COPY command.

File	SQL COPY	PG direct	PG parallel
La	N/A	78.66	75.84
Lb	N/A	81.99	79.00
Sa	N/A	8.67	8.32
Sb	N/A	9.21	8.93

Table 4: Total upload time in minutes for Experiment 2.

In Table 4 the results show an optimized load procedure and database size as redundant data is not generated during any of the operations. In this case a comparative SQL COPY procedure could not be implemented as no similar state could be achieved at each operational stage between COPY and pg_bulkload. However, this experiment does show an improved runtime for loading these large datasets over that of experiment 1.

File	SQL COPY	PG direct	PG parallel
La	54.50	52.42	48.08
Lb	59.47	56.03	51.75
Sa	5.87	5.28	5.09
Sb	6.42	5.82	5.59

Table 5: Total upload time in minutes for Experiment 3.

In Table 5 the results show the fastest runtimes from all three experiments. In this case a comparative procedure could be generated between COPY and pg_bulkload. The only difference in the operational procedures is when the spatial- index is created; in the COPY case it is initiated after the table has been loaded with data from the input file, while, in the case of the pg_bulkload an empty spatial-index is created before loading commences. This difference is required to initiate the index update procedures within pg_bulkload, otherwise initiating the creation of the spatial-index could follow as in the COPY case, however, this would not be optimal based on pg_bulkloads inherent PostgreSQL operational efficiencies. Significantly in this case it can be seen that the speed-up gain using pg_bulkload in parallel mode is, on average, 87% that of the standard COPY procedure provided with PostgreSQL.

In an attempt to develop a method to predict the length of time uploading a file would take based on the file attributes we examined

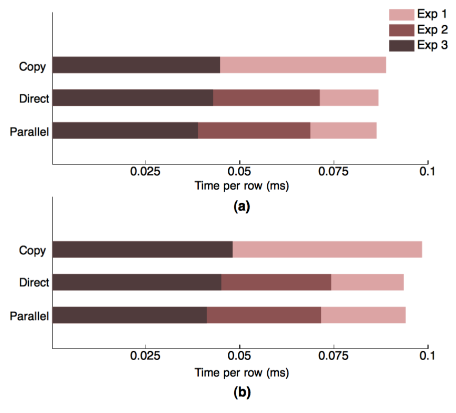


Figure 4: Timing plots for data load time per row for files (a) La and (b) Lb.

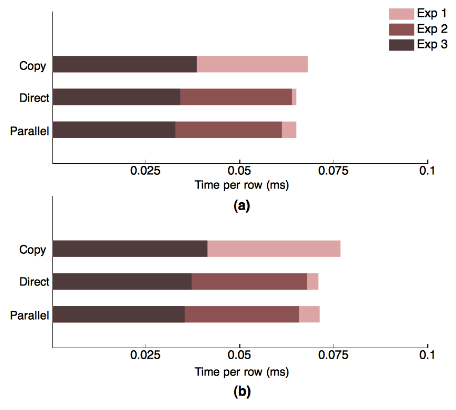


Figure 5: Timing plots for data load time per row for files (a) La and (b) Lb.

the average upload time per row for the four files. As expected, the time to upload a row of data for the two files with 10 columns, Figures 4 and 5 (a), was shorter than when uploading the data for the 14 column case, Figures 4 and 5 (b). However, uploading the smaller file had a consistently shorter per row upload time. Also, the percentage increase in row processing time for these extra 4 columns was always significantly smaller than the percentage increase in row size (kilobytes). Adding these columns to the files resulted in a file size change ranging between 33-56%, while the resulting row uploading time change ranged between only 8-15%. In Figures 6 and 7 the timing information per kilobyte is plotted. For all experiments an increase in row size resulted in a reduction in the row processing time per kilobyte. These results show that, with PostgreSQL, as the number of rows of data to upload increases the time per row increases and that an increase in row size will lead to a decrease in time per kilobyte. This implies that there is a non-linear relationship between upload time, the number of rows and row size.

5 CONCLUSION

In this paper, we have discussed in detail the different tools and strategies for uploading large LiDAR data into a PostGIS enabled PostgreSQL database. We have compared PostgreSQLs inbuilt COPY function with pg_bulkload in a number of experiments. Our comprehensive suite of experiments have shown that the best method for uploading LiDAR data and adding a spatial index to the database is to append the geometry information for each point to the file by pre-processing the files using a scripting language such as python. This resulted in a minimum 40% speedup over the traditional SQL approach used by COPY or pg_bulkload. We have also demonstrated that using pg_bulkload in the parallel mode is the fastest method for uploading large LiDAR data. In all cases we achieved a minimum of a 12% speedup over COPY using this method.

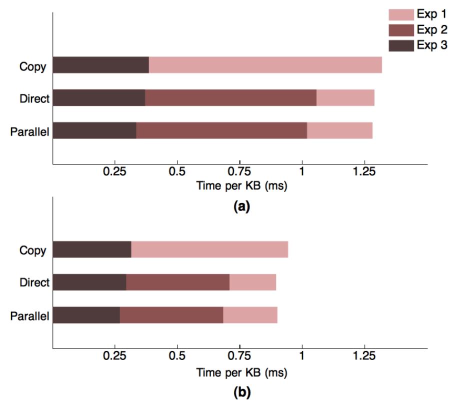


Figure 6: Timing plots for data load time per kilobyte for files (a) La and (b) Lb.

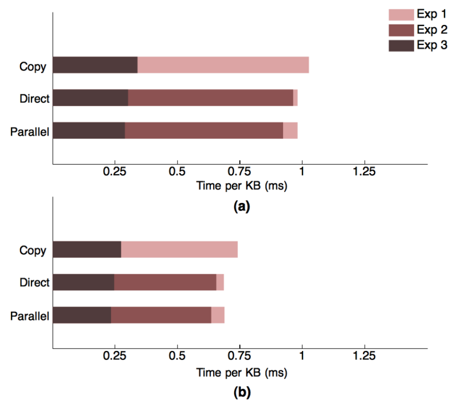


Figure 7: Timing plots for data load time per kilobyte for files (a) Sa and (b) Sb.

ACKNOWLEDGEMENTS

Research presented in this paper was funded by the NRA research fellowship programme, ERA-NET SR01 projects and by a Strategic Research Cluster grant (07/SRC/I1168) by Science Foundation Ireland under the National Development Plan. The authors gratefully acknowledge this support.

REFERENCES

Breunig, M. and Zlatanova, S., 2011. 3D geo-database research: Retrospective and future directions. Computers & Geosciences pp. 1–13.

Chen, Y., Zhang, H. and Fu, X., 2010. Organization and Query of Point Clouds Data Based on SQL Server Spatial. In: ICCSIT, pp. 178–181.

Ferede, H., Agency, N. G.-i. and Mazzuchi, T. A., 2009. Multi-dimensional data discovery. In: ASPRS/MAPPS, ASPRS, San Antonio, Texas.

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

Jaeger-frank, E., Crosby, C. J., Memon, A., Nandigam, V., Arrowsmith, J. R., Conner, J., Altintas, I. and Baru, C., 2006. A Three Tier Architecture for LiDAR Interpolation and Analysis. In: Computational Science, pp. 920–927.

Lewis, P., McElhinney, C. P., Schön, B. and Mccarthy, T., 2010. Mobile Mapping System LiDAR Data Framework. In: 3D Geo-Information 2010, Berlin, Germany.

Mc Elhinney, C. P., Kumar, P., Cahalane, C. and McCarthy, T., 2010. Initial results from European Road Safety Inspection (EURSI) mobile mapping project. In: ISPRS Commission V Technical Symposium, ISPRS, Newcastle, UK, pp. 440–445.

Ming, G., Yanmin, W., Youshan, Z. and Junzhao, Z., 2009. Research on Database Storage of Large-Scale Terrestrial LIDAR Data. In: 2009 International Forum on Computer Science-Technology and Applications, IEEE, pp. 19–23.

Mumtaz, S. A., 2008. Integrating Terrestrial Laser Scanning Models into 3d Geodatabase. In: ICAST, pp. 124–130.

Nandigam, V., Baru, C. and Crosby, C. J., 2010. Database Design for High-Resolution LIDAR Topography Data. In: Scientific and Statistical Database Management, pp. 151–159.

Parikh, N. C., Parikh, J. A., Clark, M., Damon, M., Mandable, S., Connors, M., Britain, N. and Haven, N., 2004a. An extensible system architecture for web-based lidar experimentation and data analysis. In: ILRC22.

Parikh, N., Parikh, J., Clark, M., Damon, M., Mandable, S. and Connors, M., 2004b. Remote internet-based Lidar experimentation and education. In: IEEE International IEEE International IEEE International Geoscience and Remote Sensing Symposium, 2004. IGARSS '04. Proceedings. 2004, IEEE, Anchorage, AK, pp. 4779–4782.

Rottensteiner, F., Trinder, J. and Clode, S., 2005. Data acquisition for 3D city models from LIDAR: extracting buildings and roads. In: IEEE Geoscience and Remote Sensing Symposium, IEEE, pp. 521–524.

Schön, B., Bertolotto, M., Laefer, D. F. and Morrish, S. W., 2009. Storage, manipulation, and visualization of lidar data. In: 3D-ARCH, ISPRS, Zurich, Switzerland.

Sharma, N. C. P. and Parikh, J. A., 2008. Data Analyis System Design for Lidar Experimentation. In: IGARSS 2008 - 2008 IEEE International Geoscience and Remote Sensing Symposium, IEEE, Bostan, USA, pp. 1420– 1420.

Sharma, N., Parikh, J. and Clark, M., 2006. A Lidar Collaboratory Data Management System. In: 2006 IEEE International Symposium on Geoscience and Remote Sensing, IEEE, pp. 817–820.