Efficient LiDAR point cloud data encoding for scalable data management within the Hadoop eco-system

Anh Vu Vo
School of Computer Science
University College Dublin
Dublin, Ireland
anhvu.vo@ucd.ie

Chamin Nalinda Lokugam Hewage
School of Computer Science
University College Dublin
Dublin, Ireland
chamin.lokugamhewage@ucdconnect.ie

Gianmarco Russo
Department of Computer Science
University of Salerno
Fisciano, Italy
g.russo159@studenti.unisa.it

Neel Chauhan
Center for Urban Science + Progress
New York University
New York, USA
nc1682@nyu.edu

Debra F. Laefer
Center for Urban Science + Progress
New York University
New York, USA
debra.laefer@nyu.edu

Michela Bertolotto
School of Computer Science
University College Dublin
Dublin, Ireland
michela.bertolotto@ucd.ie

Nhien-An Le-Khac
School of Computer Science
University College Dublin
Dublin, Ireland
an.lekhac@ucd.ie

Ulrich Oftendinger
School of Natural and Built Environment
Queen's University Belfast
Belfast, Northern Ireland
u.ofterdinger@qub.ac.uk

***Abstract*—This paper introduces a novel LiDAR point cloud data encoding solution that is compact, flexible, and fully supports distributed data storage within the Hadoop distributed computing environment. The proposed data encoding solution is developed based on Sequence File and Google Protocol Buffers. Sequence File is a generic splittable binary file format built in the Hadoop framework for storage of arbitrary binary data. The key challenge in adopting the Sequence File format for LiDAR data is in the strategy for effectively encoding the LiDAR data as binary sequences in a way that the data can be represented compactly, while allowing necessary mutation. For that purpose, a data encoding solution, based on Google Protocol Buffers (a language-neutral, cross-platform, extensible data serialisation framework) was developed and evaluated. Since neither of the underlying technologies is sufficient to completely and efficiently represent all necessary point formats for distributed computing, an innovative fusion of them was required to provide a viable data storage solution. This paper presents the details of such a data encoding implementation and rigorously evaluates the efficiency of the proposed data encoding solution. Benchmarking was done against a straightforward, naive text encoding implementation using a high-density aerial LiDAR scan of a portion of Dublin, Ireland. The results demonstrated a 6-times reduction in data volume, a 4-times reduction in database ingestion time, and up to a 5 times reduction in querying time.**

***Index Terms*—LiDAR, point cloud, Big Data, Hadoop, HBase, Google Protocol Buffers, spatial database, data encoding, distributed database, distributed computing**

I. Introduction

Globally, three-dimensional (3D) data of the Earth's topography is being collected at an unprecedented rate using aerial Light Detection and Ranging (LiDAR). As an example, the United States is undertaking its first nationwide LiDAR mapping. As part of that, more than 53% of the country has been mapped and more than 10 trillion collected data points have been made available publicly [1]. Similarly, in Europe, many countries, including Czech, Denmark, England, Finland, the Netherlands, Poland, Slovenia, Spain, and Switzerland, have completed nation-wide LiDAR mapping with many more in progress [2]. Extensive, country-scale, LiDAR mapping projects have also been undertaken in Asia in Japan and the Philippines [3] [4]. These projects highlight the need for an efficient LiDAR data access system that [1] stores data and serves retrieval requests (transactional querying), [2] streams data to other applications (data streaming), and [3] periodically analyses the accumulated data (batch processing) [5].

This paper focuses on presenting a robust and scalable data encoding solution. The data encoding is a component within a complete data storage system that integrates the encoding with other components including data indices, search algorithms, and cache strategies. Readers may consult the authors' previous works for information explicitly on those other topics [6], [7]).

978-1-7281-0858-2/19/$31.00 © 2019 IEEE

II. Background

LiDAR data are most often available in the format of point clouds, which are collections of discrete, sampling points of visible surfaces. The essential component of each LiDAR data point is its coordinates (i.e. x, y, and z). Apart from the point coordinates, each point may have other attributes such as the timestamp and the intensity of the reflected laser signal. The exact number and type of point attributes depend on the sensing platform and the data processing procedures. For example, aerial LiDAR data often contains the scan angle rank and edge of flight line, which do not exist in terrestrial LiDAR data. Additionally, depending on how the data are processed, a LiDAR point cloud can be enriched with attributes derived from the post-processing such as classification tags, colour integrated from imagery data, and physical simulation data (e.g. [8], [9]). LiDAR data density varies significantly within a dataset as the density depends on range, incident angle, and other project-specific or equipment-specific factors. Thus, an assemblage of LiDAR data points collected by multiple platforms (i.e. different aerial and terrestrial sensors) and/or processed by different procedures are highly likely to be heterogeneous in data schema and distribution. Such heterogeneity must be accommodated, especially when data integration is needed. The variation of data density can be even more extreme when multiple datasets are aggregated from disparate sensing platforms (e.g. different aerial and terrestrial sensors).

As a LiDAR point cloud dataset is essentially a collection of point records, each of which contains the point's coordinates and a set of numeric attributes, a straightforward method to encode the data is a textual data encoding method. Most LiDAR and point cloud software (e.g. Leica's Cyclone, Riegl's RiSCAN Pro, Riegl's RiPROCESS, PointCloudLibrary, CloudCompare) support text formats. A typical text-based point cloud data file has each point record stored as a separate line in the file. The coordinates and attributes are separated by defined delimiters such as commas and spaces. Each numeric digit is represented as a character via a standard coded character set (e.g. ASCII, Unicode). A coded character set is a protocol for transforming each character in a defined set to a unique number and, ultimately, to a sequence of digital bits (i.e. code unit) for storage in digital computing systems. Depending upon the coded character set being selected, the code units can be fixed-length[1] or variable-length[2]. Decimal digits and delimiters between the numbers can be encoded as characters in the same way. Text-based encoding is straightforward. The encoded data are human-readable and can be easily manipulated with a wide range of text displaying and editing software. However, text-based encoding is inefficient in terms of file size. Furthermore, parsing numeric values from a text file is usually slower than parsing an equivalent binary file.

[1]Such as, 7 bits in the original ASCII, 8 bits in the extended ASCII, and 32 bits in the UTF-32

[2]From 8 to 32 bits in UTF-8

Binary encoding is an alternative to text-based encoding. Depending on the expected values and distribution of each attribute, a data type can be selected for each attribute (e.g. 1-bit Boolean value, 1-byte, 2-, 4-, or 8-byte integer, single, or double-floating point number). Binary encoding offers greater flexibility, file size compactness, and better data parsing speed. The main challenge in using binary format is in interoperability. Data encoding and decoding require an agreed file format specification. Additionally, multiple encoders and decoders might have to be implemented and maintained, if data are intended to be transferrable across software written in various programming languages. A variety of binary file formats have been developed for LiDAR data. In fact, raw data captured by the LiDAR sensors are typically stored in proprietary binary formats, which can only be interpreted by proprietary software provided by the sensors' manufacturers (e.g. Leica, Riegl). There are also many vendor-neutral, binary file formats created for LiDAR point cloud data. Most common among them is the LAS file format developed by the American Society of Photogrammetry and Remote Sensing (ASPRS) for aerial LiDAR data exchange. As of version 1.4 revision 14, the LAS file specification [10] provides 10 different formats for point data records. Each of the point record formats has a fixed structure of attributes and an optional block of extra bytes. The extra bytes are provided to make the file format more extensible and to permit storage of user-defined data. Presently, users must choose one amongst the 10 point data record formats to match their data. If the selected point record format does not perfectly match the actual data to be stored, the mismatch will leave storage space unoccupied, which unnecessarily increases the file size, as well as the file parsing time. This problem is especially common when the LAS format is adopted for storing point cloud derived from non-aerial systems. In addition, the LAS format is intended for storing LiDAR sensing data only. Furthermore, LAS does not support derived data from post-processing (e.g. the shadowing and solar potential calculations shown in [8] and [9]).

Other binary formats commonly used for storing LiDAR point clouds include the E57 format [11] and the HDF5 format [12]. The E57 format, specified by the American Society for Testing and Materials, serves as a format for data derived from generic 3D imaging systems. Unlike LAS, which mostly relies on fixed length attribute structures, E57 mixes both fixed sized and XML-based, self-documenting, variable-length structures. The hybrid solution offers a balance between multiple criteria. Variable length structures are more flexible and provide a higher level of extensibility but at a cost of higher complexity and slower implementation. While E57 can encode extremely large files up to 9 Exabytes in length [11], the solution was not meant to serve Big Data analytics, in which parallelisation is of paramount importance. In terms of parallelisation supports, Hierarchical Data Format version 5 (HDF5), originally developed by the National Center for Supercomputing Applications, is a more suitable candidate. At the logical level, an HDF5 file is a container that can possess a heterogeneous collection of datasets of different types, which

may include images, tables, graphs, and documents. Even though HDF5 was not specifically designed for 3D imaging data or point cloud data, the file format has been successfully employed for storing LiDAR point clouds [13], as well as LiDAR full waveform LiDAR data [14]. HDF5 has certain high scalability, high performance computing capabilities. For example, there is no limit in the size of a HDF5 file, and the parallel version of the file format (i.e. Parallel HDF5) was designed to natively support distributed-memory programming with Message Passing Interface (MPI).

As LiDAR data are being aggregated at extremely large scales (e.g. billions to trillions of data points), there is an obvious demand for scalable data storage and management solutions for LiDAR data. Amongst the technologies that have emerged in the recent decades, distributed-memory systems have the greatest prospects for management and analysis of datasets at extreme scales [5]. MPI and Hadoop are two of the most common programming frameworks to program distributed memory systems. While HDF5 appears to be a promising LiDAR file format for MPI, there is not an equivalent Hadoop format rigorously designed for LiDAR data. Systems such as Geowave by Whitby et al. [15] allows storage of point cloud data in a scalable Accumulo key-value data store, which is also built atop Hadoop distributed file system. However, Geowave treats point cloud data as a standard vector data type. More specifically, point cloud data are sequentially parsed by PDAL (PDAL contributors, 2018) and ingested into Accumulo as a set of 2D MultiPoint features according to the Simple Features specification (ISO 19125). While a thorough evaluation of such a vector-based solution is not within the aim of this paper, the previous research by van Oosterom [16] suggests that such a solution is impractical because of inherent inefficiencies.

Generally, Hadoop offers several different data formats, including text encoding formats and a framework called Sequence File for encoding data in binary. A Sequence File is essentially a data structure composed of a sequence of binary key-value pairs. Sequence Files are splittable, meaning multiple processes can work concurrently on the same file to reduce the data processing time. The file format has certain built-in compression mechanisms that allows both individual record compression and block compression to reduce file sizes and I/O costs. In addition to the key-value sequence, which is the main content of a Sequence File, each file also contains a header block that stores metadata (e.g. format version, file version, compression mechanism, and user-defined metadata records). In this paper, the Hadoop Sequence File format is adopted to develop a distributed file format for efficiently encoding LiDAR data in the Hadoop eco-system. Adoption of a high scalability platform such as Hadoop for LiDAR data management and analysis is not always straightforward. The core characteristics of LiDAR must be taken into account. At the same time, consideration must be paid to maximising the computational efficiency of the adopted technologies, which has mostly developed for generic and unrelated use cases and data types. Through an adoption of a generic Hadoop file format for storage of LiDAR point cloud data, those considerations are described, justified, and evaluated in this paper via a set of performance criteria.

III. Criteria for a compact and flexible LiDAR data encoding solution for Hadoop-based data systems

As a step towards building a high-scalability LiDAR data system, this paper introduces a data encoding solution, which aims to meet 4 criteria, as described below.

Criterion I: Compatibility to the Hadoop eco-system
The proposed file format must be fully compatible to the Hadoop distributed computing platform, which allows storage, querying, and processing the data in parallel on distributed-memory computing clusters. The critical criterion is to ensure the system can cope with the growth in data volumes and computational loads. This criterion is primary.

Criterion II: Compactness
Data compactness is needed to restrict storage space requirements, as well as to reduce the input/output (IO) cost, which often controls overhead in computing systems.

Criterion III: Versatility
Being versatile means that the file format can be conveniently mutated to accommodate point attributes emerged anytime during the data processing pipeline. That feature enables the proposed format to be used as a working data format by allowing storage of data produced from post-processing. Examples of derived point attributes include classification tags, colour values integrated from external imagery sources, and per-point labelling and simulation data. This feature distinguishes the proposed format from the majority of other LiDAR data formats (e.g. LAS, E57), which are aimed at hosting sensing data only. In addition, data heterogeneity should be allowed, as multiple point attribute structures may co-exist in a single point cloud dataset. Such complexity commonly occurs when data are integrated from multiple sources (e.g. different terrestrial and aerial sensing platforms) or partial data mutation is performed on the dataset.

Criterion IV: Neutrality in programming languages
Data stored in the data system are intended to be consumed by applications written in multiple languages. For example, as most of the software in the Hadoop eco-system is written in Java and supports Java as the primary language, data ingestion and low-level data manipulation are performed most efficiently through a Java client. However, once data reside in the database, they may be requested to serve disparate applications written in different languages (e.g. Python, Javascript). Typically, implementation of a data format in multiple languages is time consuming and requires significant maintenance costs.

IV. Compact and flexible LiDAR data encoding solution for Hadoop-based data systems

The proposed data encoding solution is a strategic fusion of the Hadoop Sequence File format and a binary data encoding solution based on Google Protocol Buffers. As described in Section II, Sequence File is the primary binary data encoding

framework built-in Hadoop that possesses all features of the Hadoop Distributed File System (i.e. distributed, scalable, fault tolerant, and suitable for high-throughput data processing) [Criterion I]. Sequence Files are inherently splittable, meaning a file can be partitioned into multiple splits (i.e. segments) for processing in parallel when necessary. Thus, the file format is suitable for high-throughput data processing in Hadoop, typically via a MapReduce or a Spark job [Criterion I]. At the logical level, a Sequence File consists of a sequence of key-value pairs, in which the keys and values can be any arbitrary byte sequences. Encoding and decoding the binary keys and values are up to the users, typically via custom encoders and decoders. Binary data encoding allows the data to be stored in a compact manner [Criterion II]. The major challenge in adopting the Sequence File format for LiDAR data storage is in devising a data encoding method to effectively represent LiDAR data records as binary sequences within the structure of a Sequence File.

Instead of establishing an ad-hoc binary format for encoding LiDAR data as binary key-value pairs in Sequence Files, the authors opted to base the proposed binary encoding solution on Protocol Buffers, a binary data serialisation framework developed by Google originally for serialising inter-machine communication messages. Google Protocol Buffers (GPB) is a convenient solution to ensure the file format is compact [Criterion II], highly versatile [Criterion III], and language-neutral [Criterion IV]. All of those criteria are fully present in the LiDAR data encoding solution proposed in the paper. While GPB was selected for the implementation presented in the paper, there are several binary encoding alternatives such as Thrift (https://thrift.apache.org/), and Avro (https://avro.apache.org/) that are based on analogous principles and offer similar capabilities; readers interested in an in-depth comparison of the data encoding alternatives may consult [17].

An implementation of GPB data serialisation for LiDAR data requires a message format file (i.e. proto file) to map all foreseeable point attributes to GPB data types (e.g. Boolean, 32-bit integer, unsigned 64-bit integer, double-floating number, string). The `PointP` message in Listing 1 (lines 8-50) serves as an example of such a format. In that example, the point data are assumed to be sourced from an aerial LiDAR dataset in LAS format. The attributes are derived from point record format 1 defined within the LAS specification [10]. Notably, GPB allows the format to evolve. Therefore, other point attributes not envisioned at the initial time of defining can be appended to the format. Once that happens, the extended format remains back-compatible with all earlier versions of the format, which have fewer attributes.

The `spatialCode` on line 24 of Listing 1 is an example of an extra attribute added after the initial definition. The attribute is added to hold spatial indexing codes (e.g. Morton, Hilbert, Peano codes) when they are computed during post-processing or analysis of the original sensing data. Inapplicable attributes (i.e. attributes with empty values) do not occupy storage space in a GPB file. Another important feature is that GPB optimises the data compactness by automatically adjusting the code unit lengths according to the numeric values being encoded. In addition to the `PointP` message, the example format in Listing 1 also consists of `PointSequenceP`, which allows packing an arbitrary point set into a single message. The `PointSequenceP` message contains the point sequence itself (line 60) together with metadata about the sequence (lines 52-59). In the example format, the point coordinates and timestamps are internally stored as integral numbers, which are transformed to real coordinates via the offset and scale factors. As explained by Isenburg [18], the data encoding strategy is effective in preserving the uniformity of the spatial and temporal sampling, as well as avoiding the precision loss that happens when large numbers are encoded as floating-point numbers. The offset and scale factors are retained as part of the metadata in the `PointSequenceP` message (lines 52-59 in Listing 1).

Listing 1. Google Protocol Buffers messages for point and point sequence

```
1
syntax = "proto2";                                                    2
                                                                      3
option java_package = "umg.core.lidar.protobuf                        4
                                                    .laspoint";       5
option java_outer_classname = "LASPointProtos";                       6
                                                                      7
message PointP{                                                       8
  required int32 x = 1;                                               9
  required int32 y = 2;                                               10
  required int32 z = 3;                                               11
  optional uint32 intensity = 4;                                      12
  optional uint32 returnNumber = 5;                                   13
  optional uint32 numberOfReturns = 6;                                14
  optional bool scanDirectionFlag = 7;                                15
  optional bool edgeOfFlightLine = 8;                                 16
  optional PointClassP classification = 9;                            17
  optional bool synthetic = 10;                                       18
  optional bool keyPoint = 11;                                        19
  optional bool withheld = 12;                                        20
  optional int32 scanAngleRank = 13;                                  21
  optional uint32 pointSourceID = 14;                                 22
  optional uint64 gpsTimestamp = 15;                                  23
  repeated bytes spatialCode = 16;                                    24
                                                                      25
  enum PointClassP {                                                  26
    NEVER_CLASSIFIED = 0;                                             27
    UNCLASSIFIED = 1;                                                 28
    GROUND = 2;                                                       29
    LOW_VEGETATION = 3;                                               30
    MEDIUM_VEGETATION = 4;                                            31
    HIGH_VEGETATION = 5;                                              32
    BUILDING = 6;                                                     33
    NOISE = 7;                                                        34
    MODEL_KEY_POINT = 8;                                              35
    WATER = 9;                                                        36
    RAIL = 10;                                                        37
    ROAD_SURFACE = 11;                                                38
    WIRE_GUARD = 12;                                                  39
    WIRE_CONDUCTOR = 13;                                              40
    TRANSMISSION_TOWER = 14;                                          41
    WIRE_STRUCTURE_CONNECTOR = 15;                                    42
    BRIDGE_DECK = 16;                                                 43
    HIGH_NOISE = 17;                                                  44
    OVERLAP_POINT = 18;                                               45
    RESERVED = 19;                                                    46
    USER_DEFINABLE = 20;                                              47
  }                                                                   48
}                                                                     49
                                                                      50
```

```
message PointSequenceP{
  required double xOffset = 1;
  required double yOffset = 2;
  required double zOffset = 3;
  required double xScale = 4;
  required double yScale = 5;
  required double zScale = 6;
  required double tOffset = 7;
  required double tScale = 8;
  repeated PointP point = 9;
}
```

Once the GPB message formats are defined, a code compiler can be used to automatically generate data encoders and decoders in the selected language with minimal efforts. GPB version 2 supported languages include C++, C#, Java, Python. The list is extended in version 3 to include Javascript, Objective C, PHP, and Ruby. An encoder generated by a GPB compiler allows encoding a data source (e.g. a LiDAR point cloud) in the format defined by the message format and serialising the encoded data to a persistant format (e.g. a file, or a binary object in a database). This GPB encoding is the first step (LAS2Proto) in preparing a LiDAR dataset for distributed computing in Hadoop. The complete workflow is presented in Figure 1.

In that first step, the input point clouds are re-encoded into the GPB message format defined in Listing 1. The point data are partitioned into segments of a pre-defined length (e.g. 1 million points/segment). Each segment is encoded as a `PointSequenceP` message and stored as a binary file. Subsequently, the point Sequence Files are wrapped into a Hadoop Sequence format (via PCProto2SEQ), in which each point record is stored as a key-value pair. The key can contain any of the point attributes depending on the intended use of the data or can be left as empty, to reduce the file sizes. In the particular implementation in this paper, the point timestamps are used as keys. Both LAS2Proto and PCProto2SEQ are multi-threading Java applications, which can parallelise the computation on multiple CPU threads. Even though these data preparation steps are not capable of making use of the Hadoop multiple nodes, these steps are only required once during the life-cycle of a dataset.

Immediately after the two preparation steps, the LiDAR data in Hadoop Sequence format can be uploaded into the Hadoop Distributed File System (HDFS) and fully exploit the advantages of the scalable computing framework. For example, the Sequence Files can be indexed and ingested in HBase (i.e. a key-value database within the Hadoop ecosystem) for data retrieval. Multiple applications can retrieve the GPB data residing in the database and may decode them using a decoder automatically generated by a compatible GPB compiler. Additionally, the Sequence Files can be used as input for any MapReduce or Spark job, which may perform batch processing or data analytic tasks on the data. New point clouds derived from MapReduce and Spark jobs can also be stored in the proposed file format or any extension of the format. In all of those scenarios, the encoded data are effectively structured to best use the efficiency and scalability built-in the Hadoop framework.

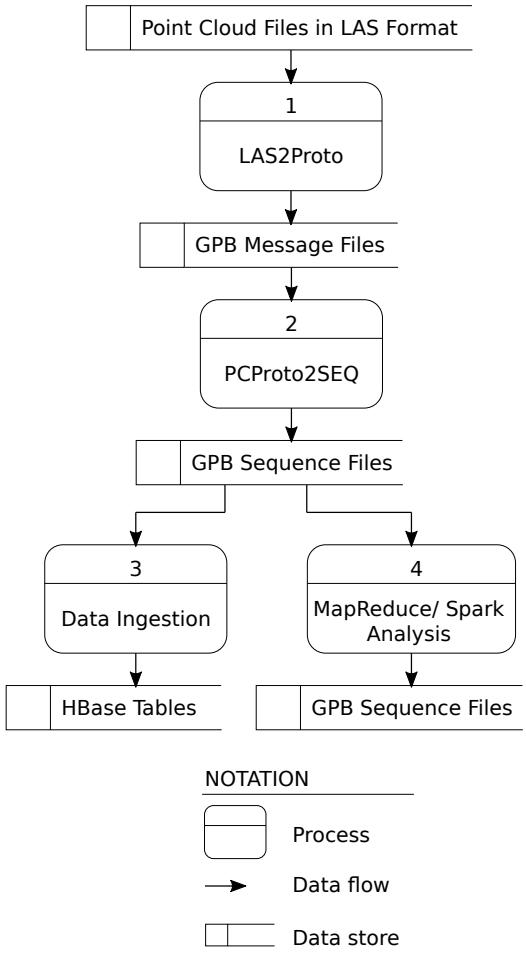

Fig. 1. GPB Sequence File preparation and possible uses

V. Evaluation

To evaluate the efficiency of the data encoding solution presented in Sections III–IV, the encoding is applied to 1.5 km^2 of real aerial LiDAR data from Dublin, Ireland (Figure 2) [19]. The data were collected with a Riegl LMS-Q680i scanner operating at 400 kHz in March 2015. This data set is known to be the densest publicly available aerial laser scanning data with its density of more than 300 points/m^2. The original dataset contains more than 1.4 billion LiDAR points. The tests presented in this paper are based on 4 subsets of the original data (with a maximum of 812 million points). The complete dataset of 1.4 billion data points was not used for testing since the required time for ingesting it in the text format was excessive. The original point cloud data are available in both LAS and text formats. The data in their text format are used in this paper as a benchmark to evaluate the binary encoding solution. The data in their LAS format are re-encoded using the proposed binary format. Both the text and binary data are ingested into HBase. Data sizes, data ingestion times, and querying times are recorded and used as the basis for performance evaluation of the proposed file format. All tests were conducted on a 35-node Hadoop cluster provided by New

York University's High-Performance Computing Center. The cluster specifications are described in Table I.

Fig. 2. 2015 aerial LiDAR point cloud data of Dublin city

TABLE I
HADOOP CLUSTER TESTING CONFIGURATION

Number of data nodes	35
Disk	16 × 2 TB per node
Memory	128 GB per node
CPU	2×8 cores Intel Haswell of 2.5 GHz
Network	10 Gb Ethernet
Operating System	Linux (Centos 6.9)
HBase version	1.2.0
Hadoop version	2.6.0
HBase replication factor	3

The pivotal objective of this work was to evaluate the performance and effectiveness of the proposed data encoding technique. To focus on that data encoding, this paper employs a minimal data indexing strategy, in which the point records are indexed only by their timestamps. In the original encoded according to the LAS format [10], each timestamp in the testing dataset was a double floating point number representing the time at which the corresponding laser pulse was emitted from the LiDAR sensor. The timestamps were in GPS Week Times (i.e. the number of seconds counted from the midnight Saturday night to Sunday morning of the week). When represented in the text format, each timestamp has 6 significant digits and 6 decimal digits (i.e. temporal precision up to 1 milli-second).

Four subsets [i.e. D1-D4 in Tables III–IV] were selected from the 2015 high-resolution aerial LiDAR dataset of Dublin city to serve the performance and scalability evaluation. Each subset was indexed and ingested into a HBase database sequentially in the proposed binary format and in the reference text format. Data ingestion times and file sizes were recorded for comparison. Finally, point and range queries were performed on each of the subsets to evaluate the effects of the data encoding on querying response times. The following subsections provide details of the testing results.

A. Influence of data encoding format on data ingestion speed and data volumes

To ingest a LiDAR point cloud subset into HBase (Process 3 in the data flow in Figure 1), the dataset (either in the text or the binary format) was copied from the local file system to Hadoop Distributed File System (HDFS). Once copied to HDFS, the dataset was automatically partitioned into chunks and was replicated multiple times (3 times by default) for fault tolerance and parallelisation. A MapReduce job was performed to index the data and to transform the point cloud data into HFiles, the internal distributed file format in HBase, for bulk ingestion. Subsequently, the data were loaded into the database. Among the three steps, generating HFiles was the most time consuming and dominated the difference in data ingestion time between the two encoding formats. After data ingestion, the data in both formats were compressed using the generic Snappy data compression library (https://github.com/google/snappy) to investigate the effect of data compression on data volumes and querying times. The data ingestion times and the data sizes inside the database under various formats and settings are presented in Tables II–IV, and Figures 3–4.

Table II compares the proposed file format (i.e. data generated by Process 2 of the data flow in Figure 1) with several common LiDAR file formats in terms of the data volume. To ensure a fair comparison, the formats were evaluated using one single dataset (i.e. D4 - the largest testing dataset of 812 million points) and with the same set of point attributes. The proposed binary file format had 12.3 bytes per point, which is less than half of the size of the data in the ASPRS' LAS format (i.e. 29.1 bytes/point, including the overheads). Compared to the reference text format, the proposed binary format was 5 times more compact. Among the investigated formats, the compressed LAZ format was the only one capable of encoding the point cloud with less disc consumption (with 4.2 bytes/point).

Tables III–IV describe the difference between the proposed file format and the referenced text format in terms of data ingestion times and data sizes. Ingesting point cloud data in the proposed GPB Sequence File into a HBase database took approximately a quarter of the time taken for ingesting the corresponding datasets in the text format. The ratio is consistent for all of the variously sized data subsets (Table III and Figure 3). For both of the binary and the text formats, the data volumes increased significantly when being ingested into the database. The binary data increased by less than 7 times, while the text data grew more than 8 times. As a result, the data stored in the proposed file format were even more compact compared to the text counterpart after being ingested into the database even without compression (Figure 4). The increase in data volume due to data ingestion is attributable to the HBase data index, and other Hadoop and HBase metadata and overheads. The proposed binary file format reduced the data volume difference by more than 6-fold inside the database when compared to the reference

text format prior to data compression. The database ingestion time was also significantly reduced (a quarter of that of the reference text format), while the splittable characteristic needed for distributed computing was retained. When Snappy compression was performed, the per-point disc consumption of the text format was approximately 100 bytes, whereas each point in the binary format cost under 30 bytes. The difference in terms of data volumes between the text and the binary formats was a reduction of 3.5 times.

TABLE II
COMPARISON OF DATA SIZES OF A POINT CLOUD IN DIFFERENT FORMATS

Format	Number of points	Data size	
		Total *(GB)*	Per point *(bytes)*
Proposed format	812,779,644	9.3	12.3
Text format	812,779,644	49.0	64.7
LAS format	812,779,644	22.0	29.1
LAZ format	812,779,644	3.2	4.2

TABLE III
DATA INGESTION TIME

Format	Data-set	Number of input points	Input data size		Ingestion *(mins)*
			Total *(GB)*	Per point *(bytes)*	
Text format	D1	51,146,333	3.1	65.1	14.2
	D2	160,402,182	9.7	64.9	42.0
	D3	272,562,565	17.0	67.0	72.7
	D4	812,779,644	49.0	64.7	224.2
Proposed format	D1	51,146,333	0.6	12.6	3.8
	D2	160,402,182	1.9	12.7	10.5
	D3	272,562,565	3.1	12.2	17.0
	D4	812,779,644	9.3	12.3	52.3

TABLE IV
DATA SIZES AFTER DATABASE INGESTION

Format	Data-set	Number of DB records	HBase table size			
			Uncompressed		Compressed	
			Total *(GB)*	Per-point *(bytes)*	Total *(GB)*	Per-point *(bytes)*
Text format	D1	45,288,694	22.6	535.8	4.2	99.6
	D2	140,118,458	69.9	535.7	13.0	99.6
	D3	241,733,335	120.5	535.2	22.5	99.9
	D4	695,979,083	347.0	535.3	65.1	100.4
Proposed format	D1	45,288,690	3.5	83.3	1.2	28.5
	D2	140,118,455	11.5	88.1	3.7	28.3
	D3	241,733,329	18.4	81.7	6.3	28.0
	D4	695,979,074	52.2	80.5	18.1	27.9

B. Influence of data encoding format on querying time

A set of point and range queries were performed on the HBase tables to evaluate the influence of the data encoding format on querying times. For point queries (i.e. look-up by key), 5 randomly selected timestamps were selected (so called P1 to P5). Each point query was executed 50 times to achieve

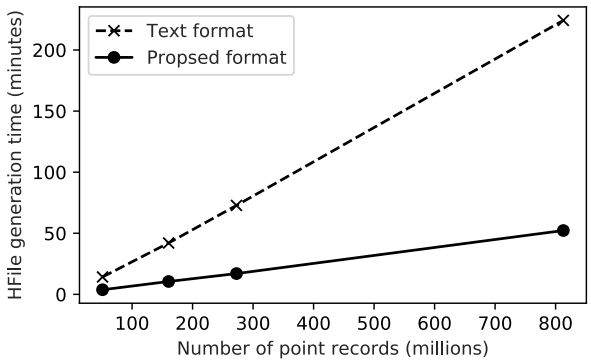

Fig. 3. HFile generation times

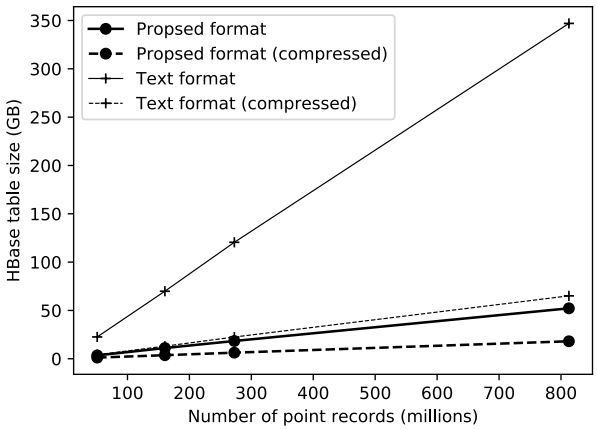

Fig. 4. Data sizes inside HBase

statistically robust response times. Similarly, 5 temporal ranges (R1 to R5) with different sizes (i.e. 1, 5, 10, 30, and 60 seconds) were selected, and each range was also executed 50 times. To analyse the influence of data compression on query times, the point and range queries were performed on both the original, uncompressed datasets and the compressed data. The response times of all queries are presented in the format of box plots (Figures 5–8).

1) Point queries: Table V presents details of the point queries (P1-P5). The response times of the point queries against the uncompressed and compressed datasets are, respectively, reported in Figure 5 and Figure 6. There is not an obvious difference between the point queries run on the binary-based and the text-based HBase tables. The median response times were between 0.85 and 0.90 seconds for both formats when the data were not compressed. The median response times slightly increased by less than 0.05 seconds when the data were compressed. Each point query always returned one point record, which cost less than 1 KB. Thus, the difference in terms of the size of the returned data was insignificant, irrespective of the difference in the data encoding formats.

The most dominant part of the querying processing time was in processing the database index to locate a record. As both the binary and the text data were indexed by the timestamps of the point records, the query processing times were similar. When the data were compressed, the cost for data decompression became an additional overhead which led to a slight increase in the total querying time.

TABLE V
POINT QUERIES

Query ID	Timestamp *(seconds)*
P1	400061.002059
P2	388610.002820
P3	388562.239010
P4	400010.003452
P5	388570.000848

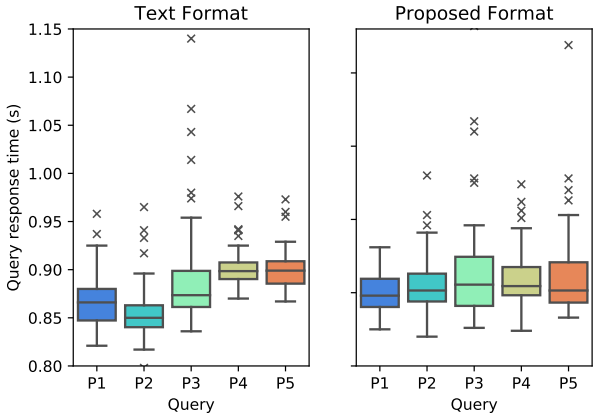

Fig. 5. Response times of point queries for uncompressed data

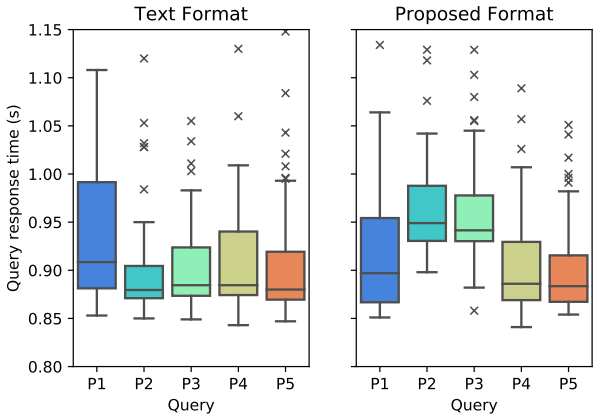

Fig. 6. Response times of point queries for compressed data

2) Range queries: Details of the five range queries (R1-R5) are presented in Table VI, along with the querying window size and the number of points returned in the querying result set. Unlike what was observed with the point queries, the data encoding format had a clear influence on the response times of the range queries (Figures 7–8). Range queries performed on the binary tables responded significantly faster compared to those performed on the text tables, irrespective of whether the data were compressed or not. The speed difference was three-fold faster for R1 (the smallest query returns 0.2 million points in a 1s time interval), and the difference was more than five-fold faster for all of the larger range queries (R2 - R5). The improvement in querying speed was mostly attributable to the compactness of the point data, which was encoded using the proposed binary format. As there were less data that needed to be fetched from the discs and transferred over the network, the range queries against the binary tables were significantly faster. Notably, the difference in querying times between the compressed and uncompressed datasets was insignificant. While Snappy reduced the data sizes inside the database (Table IV and Figure 4), in response to a query, the data has to be decompressed on the server side prior to being transferred over the network to the client. Thus, ultimately, the amount of data being carried over the network remained the same. Consequently, compressing the data inside the database did not lead to a higher querying speed as was the case of the proposed binary encoding. This highlights the benefit of data compactness achieved by the proposed binary data encoding versus data compression approaches.

TABLE VI
RANGE QUERIES

Query ID	Start timestamp *(seconds)*	Range size *(seconds)*	Number of returned records
R1	388590.000000	1	263,761
R2	399505.000000	5	1,321,484
R3	400051.000000	10	2,636,943
R4	388580.000000	30	7,904,950
R5	388570.000000	60	13,498,454

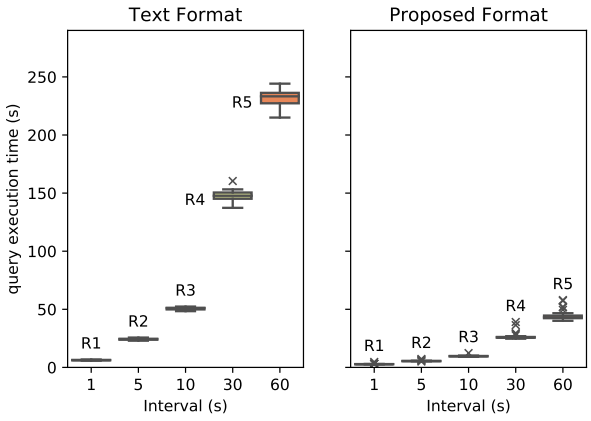

Fig. 7. Response times of range queries for uncompressed data

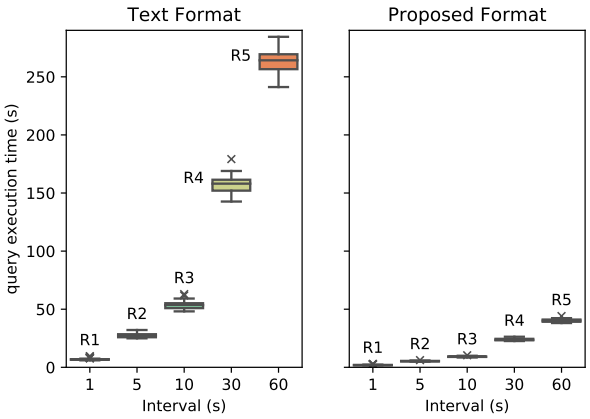

Fig. 8. Response times of range queries for compressed data

VI. Conclusions

This paper presents and evaluates the design and usage of an efficient data encoding mechanism for point cloud LiDAR data in the Hadoop distributed computing environment. The proposed LiDAR data encoding was designed to satisfy 4 criteria: (1) Compatible to Hadoop, (2) Compact, (3) Versatile, and (4) Neutral to programming languages. The encoding solution was implemented based on the Google Protocol Buffers framework. Subsequently, the encoded data were embedded in Hadoop Sequence Files to be used within the distributed computing platform. The resulting format is capable of representing every LiDAR point attributes derived from sensing platforms, as well as post processing and analyses. Data stored with the proposed format are compact, versatile, schemaless, and independent of the sensing platform. Data encoders and decoders can be automatically generated in multiple programming languages by exploiting the flexibility of GPB framework. This feature can significantly reduce implementation time, as well and minimise human errors. Most importantly, the point cloud format is fully compatible with the Hadoop distributed computing environment for both data analysis and management.

The proposed data encoding format was rigorously evaluated through a set of tests on actual LiDAR data. Prior to a data compression, point data encoded in the proposed format was twice as compact as the ASPRS's LAS format and more than 5 times smaller than the data stored in the text format. Inside HBase, the data encoded in the proposed format consumed 6 times less disc space, as compared to the text format. Data ingestion was 4 times faster with the proposed binary format. The data in both formats were further compressed with the Snappy compression library. After the compression, the binary data were approximately 3.5 times more compact than the text counterpart. The data compression did not have clear influence on the querying times. Range queries were 3-5 times faster with the proposed binary-based format irrespective of whether the data were compressed or not. Point querying was the only case where no evidence of performance enhancement was observed. In summary, the data format proposed in this paper for encoding point clouds in the Hadoop distributed computing environment was proven to have many superior characteristics in data volumes, processing speed, convenient implementation, and suitability to distributed computing. Future work may consider the integration of point cloud data from multiple platforms (i.e. aerial, mobile, terrestrial) to evaluate the efficiency of the proposed approach in handling data heterogeneity.

Acknowledgment

Funding for this project was provided by the National Science Foundation as part of the project "UrbanARK: Assessment, Risk Management, & Knowledge for Coastal Flood Risk Management in Urban Areas" NSF Award 1826134, jointly funded with Science Foundation Ireland (SFI - 17/US/3450) and Northern Ireland Trust (NI - R3078NBE). The Hadoop cluster used for the testing was provided by NYU High Performance Computing Center. The implementation and testing were jointly enabled through computing resources provided under sub-allocation "TG-IRI180015" from the Extreme Science and Engineering Discovery Environment (XSEDE), supported by National Science Foundation grant ACI-1548562 [20]. The aerial LiDAR data of Dublin were acquired with funding from the European Research Council [ERC-2012-StG-307836] and additional funding from Science Foundation Ireland [12/ERC/I2534].

References

[1] US Geological Survey, "USGS program updates," 2019, (Last accessed by 20/10/2019). [Online]. Available: https://ilmf-static.s3.amazonaws.com/uploads/2019/04/USGS-Program-Updates.pdf

[2] A. Vo, D. Laefer, and M. Bertolotto, "Airborne laser scanning data storage and indexing: State of the art review," *International Journal of Remote Sensing*, vol. 37, no. 24, pp. 6187–6204, 2016.

[3] A. Lagmay, B. Racoma, K. Aracan, J. Alconis-Ayco, and I. Saddi, "Disseminating near-real-time hazards information and flood maps in the Philippines through Web-GIS," *Journal of Environmental Sciences*, vol. 59, pp. 13–23, sep 2017.

[4] GSI, "Geographical Survey Institute Map Service," 2016, (Last accessed by 20/10/2019). [Online]. Available: http://maps.gsi.go.jp/

[5] M. Kleppmann, "Reliable, scalable, and maintainable applications," in *Designing data-intensive applications - The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media, 2017, pp. 3–22.

[6] A. Vo, N. Konda, N. Chauhan, H. Aljumaily, and D. Laefer, "Lessons learned with laser scanning point cloud management in Hadoop HBase," in *Lecture Notes in Computer Science*. Lausanne: Springer, 2018, pp. 231–253.

[7] A. Vo, N. Chauhan, D. Laefer, and M. Bertolotto, "Lessons learned with laser scanning point cloud management in Hadoop HBase," in *ISPRS International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XLII-4, 2018, pp. 671–378.

[8] A. Vo and D. Laefer, "A Big Data approach for comprehensive urban shadow analysis from airborne laser scanning point clouds," in *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. IV-4/W8, 2019, pp. 131–137.

[9] A. Vo, D. Laefer, A. Smolic, and S. Zolanvari, "Per-point processing for detailed urban solar estimation with aerial laser scanning and distributed computing," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 155, no. June, pp. 119–135, sep 2019.

[10] ASPRS, "LAS specification version 1.4 - R14," 2019, (Last accessed by 20/10/2019). [Online]. Available: http://www.asprs.org/wp-content/uploads/2019/03/LAS_1_4_r14.pdf

[11] D. Huber, "The ASTM E57 File Format for 3D Imaging Data Exchange," in *SPIE 7864 Three dimensional imaging, interaction, and measurement, 78640A*. International Society for Optics and Photonics, 2011.

[12] The HDF Group, "Hierachical data format, version 5," 2016, (Last accessed by 20/10/2019). [Online]. Available: http://www.hdfgroup.org/HDF5/

[13] M. Ingram, "Advanced point cloud format standards. Open Geospatial Consortium meeting presentation," 2015, (Last accessed by 20/10/2019). [Online]. Available: https://portal.opengeospatial.org/files/?artifact_id=67558

[14] P. Bunting, J. Armston, R. Lucas, and D. Clewley, "Sorted pulse data (SPD) library. Part I: A generic file format for LiDAR data from pulsed laser systems in terrestrial environments," *Computers & Geosciences*, vol. 56, pp. 197–206, 2013.

[15] M. Whitby, R. Fecher, and C. Bennight, "GeoWave: Utilizing distributed key-value stores for multidimensional data," in *Advances in Spatial and Temporal Databases*. Springer International Publishing, 2017, pp. 105–122.

[16] P. van Oosterom, O. Martinez-Rubi, M. Ivanova, M. Horhammer, D. Geringer, S. Ravada, T. Tijssen, M. Kodde, and R. Gonçalves, "Massive point cloud data management: Design, implementation and execution of a point cloud benchmark," *Computers & Graphics*, vol. 49, pp. 92–125, 2015.

[17] M. Kleppmann, "Encoding and evolution," in *Designing data-intensive applications - The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media, 2017, pp. 111–143.

[18] M. Isenburg, "LASzip: Lossless compression of LiDAR data," *Photogrammetric Engineering & Remote Sensing*, vol. 79, no. 2, pp. 209–217, 2013.

[19] D. Laefer, S. Abuwarda, A. Vo, L. Truong-Hong, and H. Gharibi, "2015 Aerial Laser and Photogrammetry Survey of Dublin City Collection Record," 2017, (Last accessed by 20/10/2019). [Online]. Available: https://geo.nyu.edu/catalog/nyu_2451_38684

[20] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. Peterson, R. Roskies, J. Scott, and N. Wilkins-Diehr, "XSEDE: accelerating scientific discovery," *Computing in Science and Engineering*, vol. 16, no. October, pp. 62–74, 2014.