Lessons learned with laser scanning point cloud management in Hadoop HBase

Anh-Vu Vo[1][0000-0002-6471-4905], Nikita Konda[2], Neel Chauhan[1], Harith Aljumaily[1,3], Debra F. Laefer[1][0000-0001-5134-5322]

[1] New York University, Brooklyn, NY 11201, USA
[2] University at Buffalo, Buffalo, NY 11260, USA
[3] Carlos III University of Madrid, Madrid, Spain
debra.laefer@nyu.edu

Abstract. While big data technologies are growing rapidly and benefit a wide range of science and engineering domains, many barriers remain for the remote sensing community to fully exploit the benefits provided by these emerging powerful technologies. To overcome these barriers, this paper presents the in-depth experience gained when adopting a distributed computing framework – Hadoop HBase – for storage, indexing, and integration of large scale, high resolution laser scanning point cloud data. Four data models were conceptualized, implemented, and rigorously investigated to explore the advantageous features of distributed, key-value database systems. In addition, the comparison of the four models facilitated the reassessment of several well-known point cloud management techniques founded in traditional computing environments in the new context of the distributed, key-value database. The four models were derived from two row-key designs and two columns structures, thereby demonstrating various considerations during the development of a data solution for high-resolution, city-scale aerial laser scan for a portion of Dublin, Ireland. This paper presents lessons learned from the data model design and its implementation for spatial data management in a distributed computing framework. The study is a step towards full exploitation of powerful emerging computing assets for dense spatio-temporal data.

Keywords: LiDAR, point cloud, Big Data, spatial data management, Hadoop, HBase, distributed database

1 Introduction and background

Three-dimensional point cloud is increasingly considered as an important geospatial resource for a vast range of applications. Point clouds are being collected at an unprecedented rate even at national scale [1]. Yet efforts to harness the usefulness of such datasets is increasingly threatened by the data's expanded scale, intensified density, and enhanced complexity. Effective storage, querying, and visualization are essential to successfully address these data challenges. While traditional relational database management systems (RDBMSs) have been in service for decades, recently there has been

The final publication is available at Springer via
http://dx.doi.org/ 10.1007/978-3-319-91635-4_13

the advent of non-relational alternatives with a wide range of attractive prospects. Many non-relational data systems are demonstrated as capable of handling petabytes of data emerging from the Big Data regime. To begin exploring the capability of the powerful computing assets, this paper presents an investigation of HBase – a distributed, non-relational, key-value storage platform within the Hadoop ecosystem for point cloud storage and querying.

To achieve this, the good practices established for point cloud data management in traditional environments are implemented and evaluated in the non-relational database context with 4 hypothetical data models. Throughout the paper, comparisons against previous RDBMS implementations are highlighted. The main aim is to share the lessons learned from the migration from an RDBMS context to a non-relational alternative with the prospect of building an integrated distributed, spatio-temporal database system for urban data at a future date. At the time of writing, the system is capable of providing concurrent access to laser scanning point data in the forms of exact match and three-dimensional (3D) range search. Data compression is supported by HBase's in-built compression mechanisms (e.g. Snappy, LZO, GZIP). The query accuracy of range searches can be set at the point or block level so that users can prioritize either accuracy or querying speed. Additional functionalities such as level-of-detail is currently not supported but will be considered in future research.

To provide the necessary background for the work presented in the paper, the remaining of this section introduces essential concepts behind Big Data and several technologies for handling Big Data, including non-relational databases. This includes an introduction of HBase – a non-relational database system on which this paper is based.

1.1 Big Data challenges and Hadoop technologies

According to the in-development ISO standard, ISO/IEC DIS 20546, Big Data are datasets of extensive volume, variety, velocity, and/or variability, that require scalable technologies for efficient storage, manipulation, management, and analysis. While the specific traits attributable to the nature of Big Data are still a subject to debate [2], the main technological challenge incurred by Big Data is the profound demand on performant and scalable computing power to handle the data's growth in (1) size of individual data sets, (2) speed of accumulation, and (3) complexity. The two common solutions to source the increasingly needed computing power involve a more powerful stand-alone computer (i.e. a supercomputer); or distributing the computation over multiple computers (i.e. a computing cluster). The two approaches are referred to as scale-up and scale-out solutions. The scale-out approach, also known as distributed computing, is often more cost effective and more sustainable when the data growth continues. The hardware configuration (i.e. scale-out or scale-up) must be accompanied by an appropriate programming framework. Dominant amongst existing distributed programming paradigms for scale-out computing clusters are the Message Passing Interface (MPI) and MapReduce. MPI suits tightly-coupled problems that require certain intensive communication between computing nodes to share data and the computational states. In contrast, MapReduce (which falls under the shared-nothing category) is restricted to computations that can be decoupled into independent components that require highly limited exchanges between them.

Critical to the recent advancements in Big Data technologies are the increasing popularity of low-cost hardware and open-source software enabling parallel programming across large amounts of data. Amongst the existing parallel, distributed computation frameworks, Hadoop is perhaps one of the most familiar names. The name Hadoop originated from a MapReduce web indexing project lead by Doug Cutting in 2002 that replicated the distributed data storage system and processing framework developed at Google [3-4]. Today, the name Hadoop is used beyond that initial single project to indicate an entire ecosystem of software and hardware solutions supporting distributed computing on commodity computing clusters. Facebook, Google, Yahoo, IBM are amongst the prominent Hadoop cluster owners, but there is speculation that these powerful computing assets may soon be as accessible as personal computers became in the 1990s. In fact, cloud computing has already made the technology available to anyone with a reliable internet connection and a credit card, irrespective of physical geo-positioning.

Hadoop is neither the only nor the first distributed computing technology. Parallel computing and distributed computation were well developed field long before the emergence of Hadoop. However, Hadoop is amongst the most-used distributed computing technologies today [5]. Other critical distinguishing features behind the popularity of Hadoop is attributable to its accessibility via open-source, its suitability for a wide variety of generic applications, and its friendliness to non-expert users. Hadoop abstracts most of the complexity of distributed computing away from users while only exposing a rather high-level, highly-simplified programming interface to the users. Users need not directly handle all of the internal complexity of distributed computation to be able to exploit its power.

1.2 HBase - a distributed data management system within Hadoop

Within the Hadoop ecosystem is HBase, an open-sourced replica of Google's BigTable [6]. This data management system allows random data retrieval on data at a petabyte scale distributed over thousands of servers. Unlike the Hadoop Distributed File System (HDFS), the original Hadoop data storage system, which only supports batch processing, HBase allows random access to the distributed data. Since the data are distributed, HBase databases are inherently highly parallelized. Thus, data retrieval is highly efficient. Compared to traditional relational database management systems (RDBMS), HBase provides much higher flexibility, as it does not require a rigid data schema or even data types. All HBase data are maintained in their arbitrary binary form and can be interpreted at the time of reading.

At the lower level, HBase data are maintained as a large multidimensional sorted map, which can be expressed programmatically as in Figure 1 [7]. According to that data structure, an HBase table is a sorted map ① of pairs of RowKey ② and List ③. Each element of List ③ is called a column family in HBase. A row key is a user-defined, unique identifier of each row in the HBase table. Notably, the row key plays an important role in HBase indexing as it is the primary key for sorting and also distributing the data. As a result, deciding upon the row key design is of utmost importance in HBase table design, as will be demonstrated in the latter part of this paper. Each column family [i.e. SortedMap ④] is composed of pairs of the table column ⑤ and a list ⑥ of table value and timestamp pairs [i.e. ⑥ and ⑧]. The value is the actual data content stored in the table, while the timestamp denotes the creation time of the content. The timestamp allows storage of multiple versions of the content in HBase. The data structure of a HBase table is sometimes viewed at a higher level as a collection of key-value pairs, in which a key is composed of a row-key, a column family name, a column name, and a timestamp. The value is the actual datum.

```
SortedMap<RowKey, List<SortedMap<Column, List<Value, Timestamp>>>>
   ①          ②        ③          ④          ⑤      ⑥    ⑦         ⑧
```

(a) Low-level data storage structure in HBase

```
(Table, RowKey, Family, Column, Timestamp) → Value
```

(b) A high-level view of HBase data structure

Fig. 1. HBase's data storage structure

Despite all of its favorable characteristics, HBase is not a replacement for a traditional RDBMS. While aiming for higher performance and greater flexibility, the HBase design (as with most other non-relational database systems) loosens parts of the relational features such as the compliance to Codd's 12 rules and the guarantees against transaction validity (a.k.a. ACID) – the traditional, widely-adopted RDBMS standards [7]. Even though these trade-offs are not acceptable in domains such as banking and medical databases, they are not fatally problematic in many applications such as web searching or point cloud visualization. Another feature that may defer the use of HBase is the lack of capability to model data relations. Notably, each HBase table is independent and contains no explicit relation with other tables. Powerful functionalities in RDBMS including foreign key and join are not inherently supported in HBase. In summary, HBase is introduced in this section as being representative of a new generation of high-performance, highly scalable, cost-effective non-relational data management systems that serve as alternatives to traditional relational databases. While HBase and other non-relational systems surpass traditional RDBMS with respect to many important criteria, they are not the definitive choice in every scenario. The decision between an RDBMS and a more relaxed non-relational option must be based on a rigorous justification of the features of the candidate technologies with respect to the specific data storage and retrieval demands. Some of the rationales for the selection of non-relational solutions for point cloud data storage and management are presented in Section 1.3.

1.3 Laser scanning point cloud as a growing source of Big Data

One fast growing area where a Big Data solution is clearly needed is in the storage of Light Detection And Ranging (LiDAR) data. The LiDAR technology (also known as laser scanning) [8-9] samples visible surfaces of physical objects in a 3D space. In its most basic form, the data resulting from laser scanning is a collection of discrete, densely sampling points in 3D, commonly referred to as a point cloud. A Big Data solution is needed for LiDAR data sets as they are being acquired at a national level with increasingly high density and frequency at large scale in many parts of the world including Denmark, England, Finland, Japan, the Netherlands, the Philippines, Slovenia, and Switzerland [1]. In addition, periodic repetition of national and regional LiDAR scans is becoming a more common practice for purposes such as change monitoring. All of these factors contribute to an increasingly significant burden for data storage, management, and processing.

Point cloud data are inherently spatial and share common characteristics with both raster and vector data. However, traditional vector and raster solutions are arguably unsatisfactory for point cloud data storage requiring a distinctive data representation strategies [10]. A point cloud data management system is often required to enable access to a large amount of data. A basic example is a data retrieval system that allows users to extract subsets (e.g. using range search) of a large point cloud. Data management systems are also frequently used as the backend of point cloud visualization engines. Point subsets need to be fetched from the database for rendering by the visualizer. Range search is also a relevant query in such a scenario. Additionally, point cloud processing systems can be integrated with a supporting database to retrieve the data needed for their processing workflows. Depending on what is needed for the particular processing, different kinds of query (e.g. range, neighbor, temporal search) may be required. The database, in this case, can allow the processing systems to scale to larger datasets.

As set forth by van Oosterom et al. [10], a point cloud data representation should be able to represent the point coordinates (i.e. x, y, z) together with the point attributes (e.g. intensity, color values, classification tags). There should be mechanisms to organize the point data based on spatial coherence, to compress the data, to support concurrent data access by multiple levels-of-detail (LoD), and to control the query accuracy. The authors also suggested a rich set of needed functionalities on point data that include data loading, data querying, simple and complex analysis, data conversions, object reconstruction, LoD use/access, and data updates. Additionally, parallel processing should be considered for all point cloud data operations with the performance of data loading and querying of the utmost importance for a point cloud database implementation. These criteria are the basis for recent developments including the point cloud server by Cura et al. [11], which is a full-fledge, functionality-rich point cloud management system in PostgresSQL built atop the pgPointCloud project.

Given the sheer size of point cloud data being generated, and the importance of parallelizing point cloud operations, significant research has been undertaken to exploit Big Data approaches for both point cloud analytics and management. They include various applications of tools such as MapReduce and Spark for point cloud processing [12-16]. Such research has proven that generic Big Data analytics frameworks are best-suited for computing problems that are perfectly parallelizable (a.k.a. embarrassingly parallel [17]). Examples include to assign a data point to a raster grid [12, 14] or to treat a large point cloud dataset as a group of wholly independent tiles with certain spatial buffer allowance [13]. For computing problems not obviously parallelizable, more complicated strategies such as a master-slave distributed method are needed [15]. Notably, such arrangements may impede the efficiency of the parallelization or even preclude the feasibility of the solution formulation. Research efforts in this area [18-21] are discussed in detail in the next section showing both the state of the art and the general assumption that a Big Data approach will have to be at least part of the solution for future LiDAR point cloud management.

2 Related works on relational and non-relational point cloud data management

This section provides a comprehensive review of major techniques successfully employed for point cloud data management in RDBMS and recent attempts in embracing emerging, distributed, non-relational database technologies. The motivation behind consideration of non-relational databases is also discussed.

In response to the demand for efficient management of spatial data, spatial capabilities have been integrated into a number of RDBMSs including IBM DB2 Spatial Extender, MySQL Spatial, Oracle Spatial, and PostGIS. Some of these spatial DBMSs provide support for point cloud data, often in form of a purpose-built point cloud data representation augmenting the existing spatial capability (e.g. Oracle's SDO_PC, and pgPointCloud's PCPATCH for PostGIS). Without such extensions, generic spatial systems appear to suffer from various performance and storage penalties under significant data volumes [22-24]. The key strategy behind those RDBMS point cloud extensions is the reduction in the indexing granularity. Namely, points are grouped into blocks (a.k.a. chunks or patches), which are handled inside the data system as atomic data units. That reduction significantly decreases the number of data items (e.g. by as little as hundreds to as many as millions of times depending on the specified block size), which in turn decreases the storage, indexing, and management overheads. The drawback of the method is that access to points within a block requires reading and parsing of the entire block. In addition, by grouping points into blocks, certain levels of flexibility in data updating and insertion are lost. Nevertheless, this strategy has been a defacto standard for point cloud storage in RDBMSs [e.g. 11, 23, 25, 26].

In addition to the aforementioned point grouping method, the use of space filling curves (SFC) is another important strategy increasingly adopted for point cloud storage. A space filling curve is a continuous, surjective mapping from a one-dimensional (1D) space to a higher-dimensional space [27]. Since physical storage devices are essentially 1D and the majority of database systems natively structure data by a singular key, the internal query resolving engine needs to re-formulate multi-dimensional data operations as a 1D problem. SFC is one of the approaches enabling such dimensionality reduction, thus it can be exploited to facilitate multidimensional queries on essentially 1D data systems. The use of SFC for point cloud data querying is rather a specific case of a broader class of data retrieval solutions. Though SFC usage is not restricted to the relational database technology nor point cloud data, there have been many successful attempts to utilize SFC within RDBMSs for point cloud storage and retrieval. Examples include the works by Psomadaki et al. [29], van Oosterom et al. [10]; Vo [26], and

Wang and Shan [28]. Interestingly, to store point clouds within an Oracle Index Organized Table, Psomadaki et al. [29] used an SFC, thereby, integrating both space and time as indexes for the data points. Since the SFC-based index already encodes the point coordinates that are selected for indexing (e.g. x, y, z, timestamp), the authors chose not to explicitly store the indexed point coordinates to minimize the storage costs. Even though the storage method allocates one point per row, the method appeared to be highly scalable. That may be attributable to the architecture of the Index Organized Table, which sorted the data by the primary key (i.e. SFC order in this particular implementation). The non-standard architecture is distinguishable from typical RDBMS tables that store the data in their original, unsorted state and maintains separately, rather large data indexes to support the data retrieval.

Even though improvements such as granularity reduction and use of a space filling curve as presented above have made RDBMS point cloud storage viable up to a certain level, there is a certain motivation for considering alternative storage solutions outside the relational database domain. The prime reason is the demand for greater scalability and performance.

As explained in Section 2, Big Data technologies including many non-relational data architectures are not absolute replacements for traditional RDBMSs. The prospects of better scalability, performance, availability, and/or functionality of most non-relational data systems come at a cost of lower assurance against violation of traditional database standards such as ACID. The main question while considering a non-relational database implementation is whether some relaxation is tolerable with respect to the specific application requirements. In the context of point cloud data storage, there is an ample space for such compromise. Namely, LiDAR point clouds are often static and are rarely require updating. Thus, maintaining data consistency is not as demanding as that for frequently updated data sets in domains such as banking. In addition, point clouds are weakly relational. Except for the relation between a point cloud and its metadata, most relationships of a point cloud with other point clouds (as well as other geo-spatial datasets) can be implicitly represented via the data's spatial, temporal, and/or spatio-temporal properties. While the potential losses may not be consequential, most of the flexibility provided by non-relational alternatives is meaningful for point cloud management. For example, the schema-less feature allows efficient handling of the heterogeneity of point data derived from different sources. The high level of inherent parallelism, and the potentially high compression rate are amongst the most relevant, favorable traits one can expect from a non-relational database solution.

Given the above context, there have been several attempts to employ distributed, non-relational databases for point cloud data management. They include work by Baumann et al. [18], Boehm and Liu [19], Martinez-rubi et al. [20], and Whitby et al. [21]. For example, selecting MongoDB - a document data store - as its basis, Boehm and Liu [19] stored LiDAR data tiles in their original formats as GridFS files and built a spatial index for the files' spatial extents. That system handles various metadata of the point cloud (e.g. project ID, file type) as BSON (i.e. Binary JSON) documents. Even though this approach is capable of handling a large number of LiDAR files, its usefulness is restricted to file selections since data management is only performed at the file abstraction level. Another investigation of non-relational database for point cloud data management is presented by Martinez-Rubi et al. [20]. In that research, three different approaches for point cloud storage in MonetDB – a column data store – were investigated. All three followed the one point per row method. The first approach indexed point data by the native Imprints indexing in MonetDB while the second and third approaches used two-dimensional (2D) Morton order (i.e. an SFC approach) to sort the point data. The third differed from the second in the way it replaces the indexed coordinates (i.e. x and y) with the Morton code to achieve a 30% reduction in storage overhead. The authors concluded that SFC-based approaches consumed more time for indexing but were faster and more scalable in querying responses. The authors also emphasized that keeping point data in their binary formats enormously reduced data loading time.

Unlike the works by Boehm and Liu [19] and Martinez-Rubi et al. [20], which are implementations of point cloud storage using existing non-relational databases, EarthServer by Baumann et al. [18] and Geowave by Whitby et al. [21] are full-fledged geo-spatial data systems capable of accommodating point cloud data. Built atop a multi-dimensional array database architecture, EarthServer is capable of handling and integrating a vast range of Earth observation data types derived from climatic, oceanic, and geological fields for the purpose of spatio-temporal data analytics for data at a petabyte scale. Parallelization across multiple servers is inherently supported at both the inter-query and the intra-query levels (i.e. distributed query processing). As of 2015, point cloud data were experimentally considered as part of the coverage support that also encloses regular grids (i.e. raster), irregular grids, and general meshes. Another system that fits in the same category of integrated geo-spatial database is Geowave [21]. Geowave is an open-sourced, geo-spatial software library capable of augmenting distributed, key-value databases (i.e. Apache Accumulo and Apache HBase) with spatio-temporal functionalities. The main technique backing Geowave is the use of an SFC-based index for row key construction. That technique is the backbone of Geowave, thereby enabling multi-dimensional querying within the key-value data stores. Notably, Geowave is rich in functionality (e.g. data integration, multiple LoD support, MapReduce and Spark integration) and is highly extensible. For example, the concept of Data Adapter (i.e. user-defined data encoder) allows users to model a customized data type of their choice. The point cloud is supported in Geowave via Point cloud Data Abstraction Library (PDAL).

3 Schematic designs and implementation of point cloud data storage in HBase

This paper presents the first steps towards building an integrated distributed, spatio-temporal database system for urban data that takes a point cloud as the central data component. Instead of continuing an existing platform, the authors decided to gather the good practices learned from the existing works to construct several hypothetical point cloud data models atop a representative key-value data store – Apache HBase. The primary purpose is to evaluate the advantages and limitations of each model, as well as to understand the core differences of a non-relational database implementation versus the previous RDBMS works.

To guide the database design and later to aid in evaluating the proposed point cloud data storage models (as described in Section 4), an aerial laser scanning dataset of 1.5km^2 area of the city center of Dublin, Ireland was employed throughout the paper (Figure 2). The data acquisition was conducted in March 2015, by a Riegl LMS-Q680i scanner. The total number of discrete points captured was 1,420,982,142. The typical local point density on horizontal surfaces is approximately 335 points/m^2 with an approximate vertical surface density about 1/10th of that. The LiDAR point data were delivered in the LAS 1.2 format and occupies approximately 30 GB of disk space. The main data content consists of a series of point records, in addition to the file-level metadata encoding information such as the spatial extent, the data creation date, and the coordinate system. Each point data record is composed of 28 bytes. The first 12 bytes represent the 3 point coordinates (x, y, z). The subsequent bytes compactly encode the LiDAR intensity (2 bytes), return number (3 bits), number of returns (3 bits), scan direction flag (1 bit), edge of flight line (1 bit), classification flag (1 byte), scan angle rank (1 byte), user data (1 byte), point source ID (2 bytes), and timestamp (8 bytes). Notably, some of the attributes can be left blank. Examples include the user data, the point source ID, and the classification attributes.

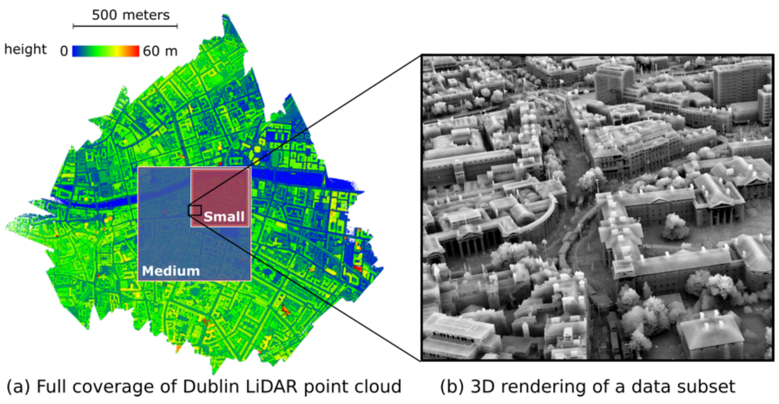

500 meters
height 0
60 m
Small
Medium
(a) Full coverage of Dublin LiDAR point cloud
(b) 3D rendering of a data subset

Fig. 2. 2015 Dublin point cloud

Given the data structure described in Section 2.2, an HBase data model must be constructed from decisions on (1) row-key, (2) column family, (3) column, (4) data cell content, and (5) versioning. Amongst those, the decisions on column family allocation and versioning are relatively obvious, while the others require rigorous evaluation. Since data stored in HBase are physically separated by column family, the column family should be used to group the data that are frequently accessed together. In this implementation, only one column family is allocated given that there is no prior assumption about querying patterns. Regarding the versioning, since the point cloud is static without updating or insertion requirements, only one version is needed. In other words, the versioning function is deactivated presently in this HBase design.

As seen in the literature, representations of a point cloud in binary formats are much more efficient than in text-based formats [10,23]. As such, the compact LAS encoding is preserved for the point record representation in this paper. The only exception is that whenever possible, empty fields are excluded from storage. The remaining concerns about the data model design are about (1) the row-key design: construct a proper row-key to facilitate the needed queries on the point data; and (2) the column structure: whether all the attributes should be grouped in one column or separated into multiple columns. As there is no insightful reasoning known to the authors, two hypothetical row-key designs (so called Single-Hilbert and Dual-Hilbert) and two column structures (i.e. Separate-Attributes, and Grouped-Attributes) are implemented and experimentally evaluated in this paper. Combinations of these options results in four data models are shown in Figure 3. Details about the row-key and columns designs are elaborated in Section 3.1 and 3.2.

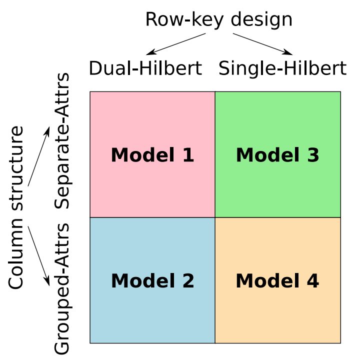


Fig. 3. Conceptual design of four data models for point cloud storage in HBase

3.1 Row-key design

Access to data in a key-value storage system is most efficiently performed via data lookup by keys (i.e. row-keys). Consequently, row-key design is the single most important element of any key-value storage solution. A row-key design is driven by dominant data access patterns. Within the scope of this project, 3D spatial queries (i.e. exact point match and range query) were selected as the primary means to access a point cloud data. Given the successful implementations of space filling curves in the authors' previous work [26] as well as in related research [e.g. 20, 21, 29], a 3D Hilbert curve was selected as the primary index to support the specified queries. Hilbert curves are defined to index given 3D spaces (e.g. [4 km × 4 km × 0.5 km]) enclosing the entire spatial extents of the geographical sites of interest at specific resolutions (e.g. 1 meter). In the first row-key design named Single-Hilbert, the LiDAR points are indexed by the Hilbert order of the voxel (e.g. [1 m × 1 m × 1 m]) containing the point. All points sharing the same voxel carry the same index, thus multiple points share the same row-key and are stored on the same row of the database. This multiple-point-per-row method is an approximate analog to the well-founded point cloud storage approaches used in many relational database management systems such as Oracle's SDO_PC and pgPoint-Cloud's PCPATCH.

To evaluate the suitability of the aforementioned traditional solution relative to the simple and more intuitive one-point-per-row approach (a.k.a. flat model), a second row-design called Dual-Hilbert was developed. With the Dual-Hilbert solution, the 1-meter resolution Hilbert index (i.e. coarse Hilbert index) is concatenated with a second spatial index computed locally within the voxel at a finer resolution (e.g. 1 millimeter) [i.e. local Hilbert index]. The fine resolution is set to be finer than the point data resolution (i.e. centimeter range) so that the concatenated Hilbert code is unique for each point. This flat model approach abandoned by the relational database systems has the potential to surpass more traditional multiple-point-per-row approaches since vertical databases such as HBase perform best for "tall tables" (i.e. large number of rows with small amount of data stored per row). Another potential benefit of using the fine-grained Hilbert index is that it can be used as a more compact replacement for the point coordinates to reduce total disk and network I/O costs. The comparisons of Model 1 versus Model 3, and Model 2 versus Model 4 (Figure 3) aim to provide evidence to assess these hypotheses.

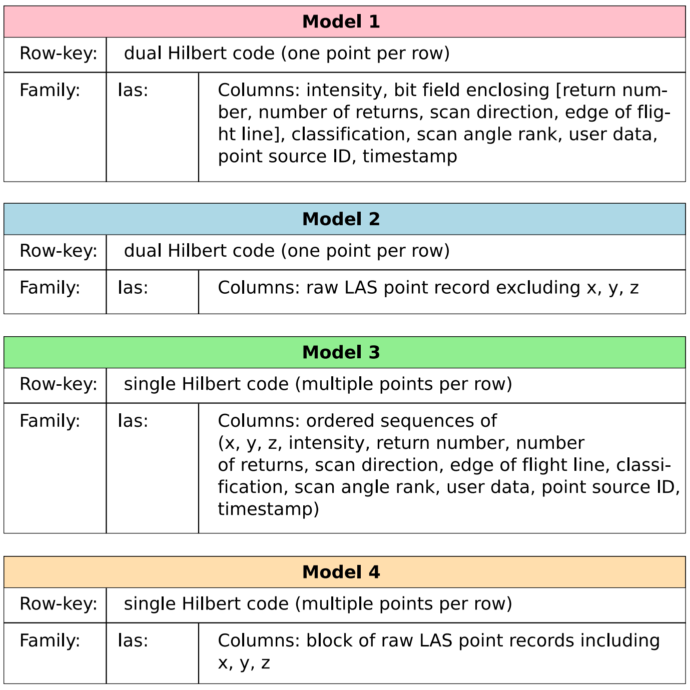
3.2 Column structures

As mentioned previously, each LiDAR data point contains a range of attributes in addition to the point coordinates. There are 10 such attributes for each point in the Dublin scan, some of which contain empty values. However, the structure and the number of point attributes are not the same for every LiDAR scan. File-based approaches including the widely used LAS data exchange format handle that semi-structured situation by providing a set of fixed templates to cover common data patterns. Each template contains a pre-determined set of placeholders for point attributes. Users can then choose the template that best matches their data, amongst the limited choices offered by the format specification. This lack of versatility can be completely alleviated when column family data stores such as HBase are used, as they are schema-less. Column family stores have no restriction on data type, data name, or the number of the attributes stored in each row. Two rows in the same table can have completely different sets of attributes. However, the flexibility and versatility do not come without a cost (e.g. storing the attribute names for each value). To better analyze the gains and costs of using HBase to provide a flexible point attribute structure, two column structures are investigated in this study. The first structure (i.e. Separate-Attributes) separates the point attributes into different columns so that the points in a table do not have to conform to any fixed template. The second structure (i.e. Grouped-Attributes) assimilates the LAS approach, in which all attributes are maintained as a binary array in a fixed structure. Comparisons of Model 1 versus Model 2, and Model 3 versus Model 4 aims to provide evidence for the assessment of column structures as shown in Section 4.

3.3 Query processing

This section presents the query resolving strategies with respect to the four data models constructed in Section 3.1 and 3.2, which are depicted in a less abstract way in Figure 4.

Point query (a.k.a. exact point match) is the most basic type of query on point cloud data. Point query aims to search for an exact match of a given point [i.e. an (x, y, z) triplet] and return all the associated attributes of the found point record. For the Dual-Hilbert data models, the exact match can be directly computed by transforming the (x, y, z) triplet into a dual Hilbert code, and then looking up the corresponding HBase table for a row-key matching the Hilbert code. Such lookup by key in HBase is termed `Get`.



Model 1		
Row-key:	dual Hilbert code (one point per row)	
Family:	las:	Columns: intensity, bit field enclosing [return number, number of returns, scan direction, edge of flight line], classification, scan angle rank, user data, point source ID, timestamp

Model 2		
Row-key:	dual Hilbert code (one point per row)	
Family:	las:	Columns: raw LAS point record excluding x, y, z

Model 3		
Row-key:	single Hilbert code (multiple points per row)	
Family:	las:	Columns: ordered sequences of (x, y, z, intensity, return number, number of returns, scan direction, edge of flight line, classification, scan angle rank, user data, point source ID, timestamp)

Model 4		
Row-key:	single Hilbert code (multiple points per row)	
Family:	las:	Columns: block of raw LAS point records including x, y, z

Fig. 4. Detail data schema of the four data models

Resolving point queries for Single-Hilbert models is slightly more complicated and less efficient. Since only the coarse Hilbert code (1 m resolution) is available for lookup, the returned data from a `Get` function is a block of points, which must be parsed and validated to return the ultimate querying result. The point block parsing can be done at either the client side or the server side in HBase. A server-side implementation (e.g. a HBase Custom Filterer) is more complicated but is more efficient as it avoids sending the entire block of points through the network, and the computing power on the server side is more available to handle the computation. Ultimately, from the theoretical perspective, a design that separate attributes into distinct columns (i.e. Model 1 and Model 3) should result in better querying performance in cases where only a subset of attributes is requested.

Spatial range search is another important type of point cloud query. A range query returns all data points enclosed within a given querying window, which often has the form of a 3D polygonal shape. The simplest case of a querying window is a rectilinear box. Spatial range search is useful for applications including downloading a data subset or clipping point data by a viewing frustum for visualization. Resolving a range query on point data sorted by a space filling curve involves decomposing the bounding of the querying windows into several continuous Hilbert segments (i.e. 1D numeric segments). The number of Hilbert segments is equivalent to the number of 1D range searches invoked against the database. Data querying can be slow, if a querying window is highly fragmented and requires a large number of range searches. The fragmentation issue can be alleviated by loosening the continuous constraint within each Hilbert segment. Namely, if the separation between two Hilbert segments is smaller than a certain level (e.g. 500 cells), the segments are grouped to reduce the number of total segments (i.e. number of database invokes). The strategy can greatly accelerate data queries at the cost of including more false-positive results (i.e. the gaps within the Hilbert segments), which may result in higher pressure on later filtering steps. Setting the value for the allowable Hilbert gap to optimize query performance is the matter of balancing the two factors and requires empirical tuning.

The Hilbert segments are then used to retrieve the point candidates by the native 1D range search on the row-keys. The Hilbert order only facilitates a coarse filtering for range querying. Namely, the candidate points resulting from a Hilbert decomposition include not only the true result but also some false positive points. The false positive points include those that fall outside the querying window but are inside its bounding box or those share the same Hilbert order with the actual resulting points. In order to get the exact results, a final fine-filtering is needed to perform a spatial check for each and every candidate points returned by the Hilbert coarse filtering. This relatively costly fine filtering should preferably be pushed to the server side to take advantage of the parallelism and data locality. Compared to the Single-Hilbert models, the dual level Hilbert codes in Model 1 and Model 2 allow Hilbert filtering at one extra level, thus reducing the amount of data passed through the fine filtering. The fine filtering can be skipped for some applications that can tolerate false positive points such as many visualizations. Skipping the fine filtering can greatly accelerate querying speed and is done natively in some existing systems such as pgPointCloud [23]. In the current HBase implementations introduced in this paper, the fine filtering can be enabled or disabled at the time of querying.

3.4 Data ingestion and querying workflows

Starting with a large, unstructured point data set in the binary LAS format, this section introduces a workflow for loading the data into HBase, while exploiting the Hadoop distributed framework (Figure 5). One of the issues during the data ingestion is that the original data format (i.e. LAS) is not suitable for processing in parallel on a computing cluster. To address that, the original LAS formatted data are transformed into a Hadoop Sequence File format (Step 1 in Figure 5). Sequence File (SF) is a Hadoop mechanism for encapsulating arbitrary binary data into a key-value format, while making the data split-able for parallel processing on Hadoop clusters. The LAS-to-Sequence transformation parses point records from the input LAS files and encodes them as values in the corresponding SFs. In this particular case, the point cloud SF's keys are not useful and, thus, left blank. The SF's metadata, which is a set of text-based key-value pairs, is exploited to store offset and scale parameters; the information needed for parsing the LAS point data. Subsequent to the sequential transformation, which occurs outside the cluster, the point cloud is uploaded to the Hadoop Distributed File System (HDFS) (Step 2 in Figure 5). Thus, the transformation to Sequence File enables parallel processing of large point cloud data.

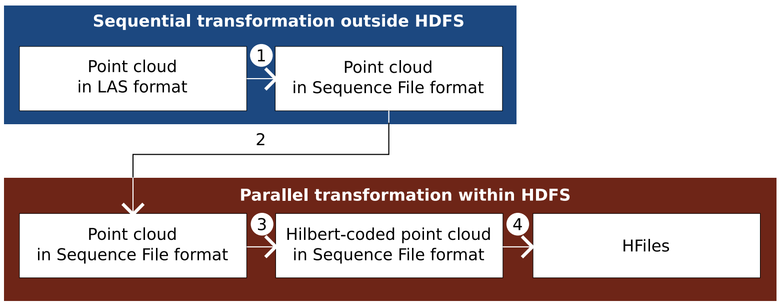

Sequential transformation outside HDFS
Point cloud in LAS format
1
Point cloud in Sequence File format
2
Parallel transformation within HDFS
Point cloud in Sequence File format
3
Hilbert-coded point cloud in Sequence File format
4
HFiles

Fig. 5. Data ingestion workflow

The data ingestion procedure continues with the Hilbert computation for every point record. A MapReduce program is used for this purpose (Step 3 in Figure 5). The mapper computes a coarse Hilbert code for each point record and outputs the result as <`coarse-hilbert; raw-las-point-record`>. The sort and shuffle process automatically handled by Hadoop MapReduce is responsible for grouping the point records by the coarse Hilbert code. Arriving at the reduce phase, the point data are grouped into blocks by the coarse Hilbert codes. The fine Hilbert codes are then computed, and the resulting codes are appended ahead of each point of the block. Ultimately, the Sequence Files resulting from the Hilbert computation have the format of <`coarse-hilbert; fine-hilbert-coded-point-block`>. Prior to being loaded into HBase, the point data need to be transformed once more into a so-called HFile format, which is the native file format underlying HBase (Step 4 in Figure 5). During this transformation, the row-key, column family, column, and data content are set. The final step of ingesting HFile data into HBase tables and distributing the data across multiple servers is facilitated by an in-built HBase function.

4 Performance evaluation

To aid in the performance evaluation, three different subsets of varying sizes (Small - S, Medium - M, and Large - L) were extracted from the 2015 Dublin point cloud. The Large dataset includes all the 2015 Dublin point cloud while the coverages of the Small and Medium datasets are shown in Figure 2. The approximate number of points in the S, M and L datasets are 90 million, 360 million, and 1.43 billion, respectively. All of the experiments were run on a 10-node Hadoop cluster. Each node consists of 2 Intel Haswell (E5-2695 v3) CPUs and 128GB DDR4-2133 RAM.

4.1 Data ingestion

Amongst the 4 steps of the data ingestion process presented in Figure 5, only Step 4 is model-dependent and needs to be run for each of the 4 models. The results of the other 3 steps are shared for all data models. Thus, they were executed only once. The LAS-to-Sequence transformation processed the data with multi-threading parallelization at a speed of 2,855,020 points/sec. The Hilbert computation operated on the cluster at a speed of 813,536 points/sec. The HFile creation speed, total data loading speed, and disk consumption of the all the experiments corresponding to the 3 datasets and the 4 models are presented in Table 1.

Table 1. Data loading speed and disk consumption

Data model	Dataset	Number of points	Size (bytes/point)	HFile creation speed (points/sec)
1	S	89,970,106	244.5	57,285
2	S	89,970,106	51.3	182,324
3	S	89,970,106	32.2	944,593
4	S	89,970,106	28.4	1,250,587
pgpc	S	89,970,106	21.0	52,698(1)
1	M	365,612,527	244.5	60,914
2	M	365,612,527	51.3	177,407
3	M	365,612,527	32.0	1,328,281
4	M	365,612,527	28.4	1,908,530
pgpc	M	365,612,527	21.0	61,939(1)
1	L	1,420,982,142	235.0	41,283
2	L	1,420,982,142	48.3	181,110
3	L	1,420,982,142	31.2	1,344,047
4	L	1,420,982,142	26.9	2,372,243
pgpc	L	1,420,982,142	21.0	57,091(1)

The corresponding storage cost and data loading speed of equivalent tests using a pgPointCloud database [23] are also included in the table for comparison. Except for Model 1, the total ingestion time for all other 3 data models, including the Hadoop Sequence file conversion and Hilbert computation, were significantly shorter [i.e. 2.5 to 8.0 times] than the time needed to load data into pgPointCloud databases. The data ingestion times for Model 1 were 1.5 times longer than that for pgPointCloud. The disk consumption of all the HBase models was higher than the pgPointCloud data.

All data models including the pgPointCloud databases appear scalable, as the data

1 total data ingestion time into a pgPointCloud database using PDAL (https://www.pdal.io)

speed remains relatively constant with respect to significant increases in data size. The disk consumption appears to largely independent from the total data size. The HBase models can be sorted as Model 1, Model 2, Model 3, Model 4 in a descending order of both the data loading speed and the storage costs. Model 1, which stores one point per row with point attributes separated in different columns, is significantly larger and took much more time to load. The above experimental results can be interpreted as the separation of point attributes, and the use of dual Hilbert codes introduces significant overheads, which is understandable when considering the physical storage structure in HBase. Namely, HBase stores data as key-value pairs. While the total amount of values – the actual point data content - is unchanged among the models, the keys vary largely. A key in HBase is an aggregation of row-key, column family name, column name, and timestamp. Both the separation of attributes and the use of dual Hilbert code increase the number of key-value pairs. In addition, the former results in significantly more content stored in the keys. The empirical result shows that the flexibility in the data schema provided by HBase as maximized in the Separate-Attributed data models comes with significant overheads.

4.2 Point query

One thousand points subsampled from the Small dataset were used as querying points for the point query performance evaluation. The queries were executed consecutively. The first query was a cold query, which often takes a longer time to process compared to subsequent queries (also known as hot queries). The difference between hot and cold query speed is mostly attributable to caching.

The point query response times are presented in Figure 6 (hot queries), Figure 7 (cold queries), and Table 2. Box plots are selected to present the distribution of the performance of the 1000 hot queries. In each box plot, the notched rectangle (i.e. box) represents the middle 50% of the response time values. The bottom and the top of the box are called the lower and the upper quartiles, which bound the middle 50% of the samples. The notch itself shows the median value, which equally bisects the sample population. The crosses are outliers, which are the values exceeding 1.5 times the upper quartiles or lesser than 1.5 times the lower quartiles. The two whiskers projected from the box represents the values outside the middle 50% of the population excluding the outliers.

As expected, the cold queries were approximately 300 to 500 milliseconds slower than the hot queries. For both cold and hot queries, Model 3 appeared to be the slowest requiring 400 to 500 milliseconds for the first runs and 22 to 26 milliseconds for the subsequent invokes. For the other models, cold queries were in the range of 300 to 450 seconds, while hot queries were from 4 to 11 seconds. The lower performance of Model 3 was caused by the difference between the stored data structure and point record structure returning to the queries. More specifically, as seen in Figure 4, point data stored in Model 3 are partitioned by attributes. Each cell contains an ordered sequence of the same attributes, e.g. x_0, x_1, x_2, …, x_n. In response to a query, these attribute sequences need to be parsed and re-organized into the point record structure, e.g. x_0, y_0, z_0, intensity$_0$, …, timestamp$_0$. The restructuring overhead is the reason for the lower performance of Model 3. Nevertheless, the most significant observation extracted from the experiment demonstrated that all data models were scalable in point querying. There was no observable performance degradation with respect with the growth in data volume.

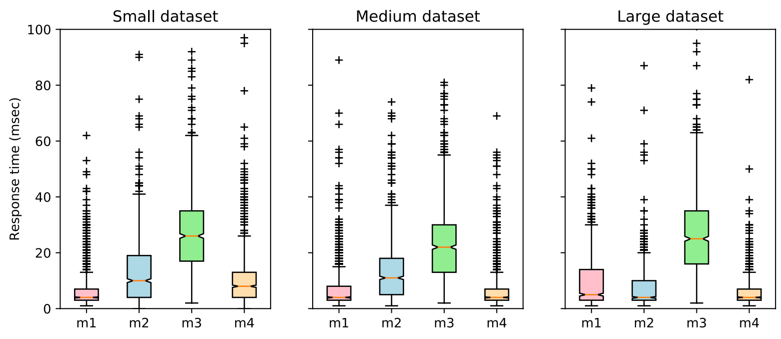


Fig. 6. Hot point query response times

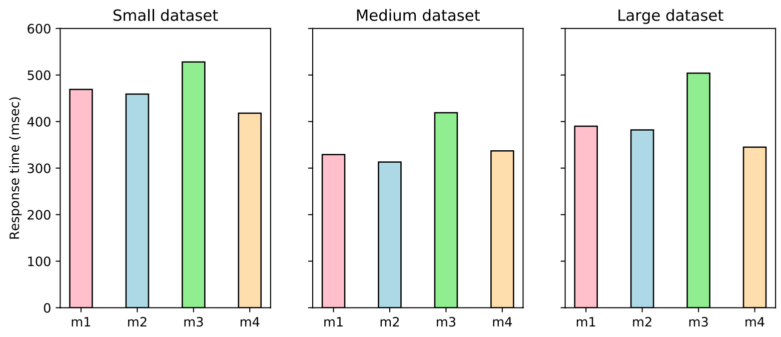


Fig. 7. Cold point query response times

Table 2 also presents the corresponding querying response times for point data stored in a PostgreSQL database with a pgPointCloud extension [23]. Since pgPointCloud does not support exact match queries, approximately equivalent range queries were used with additional filtering by the Point Data Abstraction Library (PDAL). The querying scripts can be seen in Appendix B. Compared to Model 3 – the slowest model amongst the 4 HBase candidates – exact match queries with pgPointCloud were at least 5 times slower. The differences between the pgPointCloud performance and the other 3 data models were from 12 to 34 times. Notably, there was no observable difference between the cold and hot queries in the pgPointCloud tests.

Table 2. Point query response time

Data model	Dataset	Point query response time (msec)	
		Hot (median)	Cold
1	S	4	469
2	S	10	459
3	S	26	528
4	S	8	418
pgpc	S	124	120
1	M	4	329
2	M	11	323
3	M	22	419
4	M	4	337
pgpc	M	135	130
1	L	5	390
2	L	4	382
3	L	25	504
4	L	4	345
pgpc	L	134	110

4.3 Range query

The first 50 samples of the querying points used for testing the point queries in Section 4.2 were re-used to evaluate the performance and scalability of the 4 data models in supporting range queries. Two classes of range queries were investigated. The first type considered small querying windows that are cubes with 3m long sides. The second class considered large querying windows having a side length of 50m. To alleviate the effects of data density, the range query response times were normalized by the number of returning points. The results of cold queries and hot queries corresponding to each class are plotted in Figure 8-11, and Table 3. Equivalent tests of PostgreSQL pgPointCloud databases are reported alongside the tests of the 4 data models for comparison. Notably, the HBase and pgPointCloud tests were not completely equivalent because pgPointCloud only supports 2D queries, while the HBase queries are in 3D. All the response times reported in this section include the entire costs for extracting data from the databases and exporting the resulting points to LAS files.

An important observation is that all 4 data models and the pgPointCloud databases are perfectly scalable. The querying response times remained largely unchanged despite the growth in data volume. Model 4 appears to be the best performer amongst the investigated solutions. Model 4 was 3 to 4 times faster than the other 3 data models in the cases of small, hot queries. The factors were larger (i.e. from 4 to 8 times) for the large queries. Compared to pgPointCloud, Model 4 was consistently faster (i.e. from 2 to 4 times). The difference between Model 4 and pgPointCloud was less significant in the tests with the large querying windows. The better performance of Model 4 compared to the other 3 data models is likely to be attributable to several factors. First, the aggregation of points into blocks and the attribute grouping greatly reduced the number of key-value pairs. Despite the side-effect of having larger datum per value, the number of key-value pairs reduction seems to have had a positive effect on both the querying time and the storage overhead. The second factor contributing to the better performance of Model 4 was that the data model preserved the original structure of LAS point data records, which was also the data format returned to the range queries. As such, the binary sequences stored in the database were returned directly without having to undergo restructuring as was required in the other models.

Similar to what observed from the point queries, Model 3 was also the slowest amongst all the solutions with respect to range queries. In fact, Model 1, 2, and 3 were all slower than pgPointCloud, which has an underlying structure similar to Model 4. More specifically, the point data in both pgPointCloud and Model 4 were grouped into spatial coherent groups, while the attributes of each point record were serialized into a fixed-length binary string. The observation demonstrated that the concepts established for enhancing the performance and scalability of point cloud storage in traditional environments are also applicable to distributed databases.

Table 3. Range query response times

Data model	Dataset	Point query response time (msec per 1000 points)			
		Small queries [3×3×3]		Large queries [50×50×50]	
		Hot	Cold	Hot	Cold
1	S	69	200	50	60
2	S	56	184	39	48
3	S	81	203	62	67
4	S	17	101	9	14
pgpc	S	52	46	14	15
1	M	62	185	47	61
2	M	59	175	41	50
3	M	79	201	67	66
4	M	17	96	8	13
pgpc	M	49	44	15	28
1	L	65	163	51	49
2	L	53	153	42	43
3	L	92	200	65	67
4	L	18	89	9	10
pgpc	L	69	62	15	15

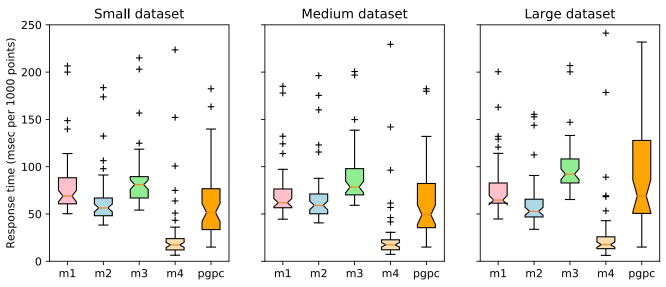


Fig. 8. Hot range query response times for small querying windows

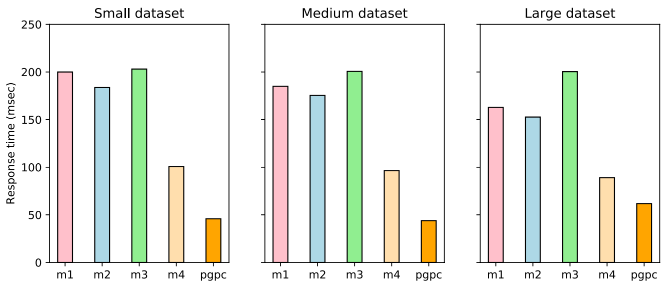


Fig. 9. Cold range query response times for small querying windows

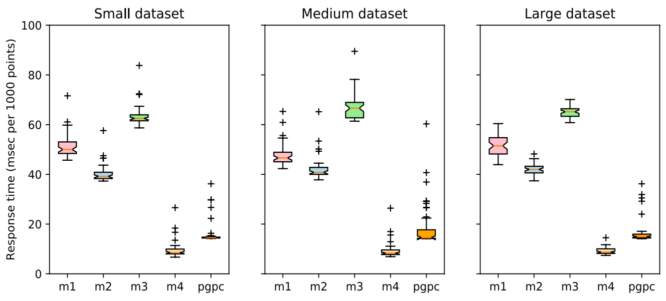


Fig. 10. Hot range query response times for large querying windows

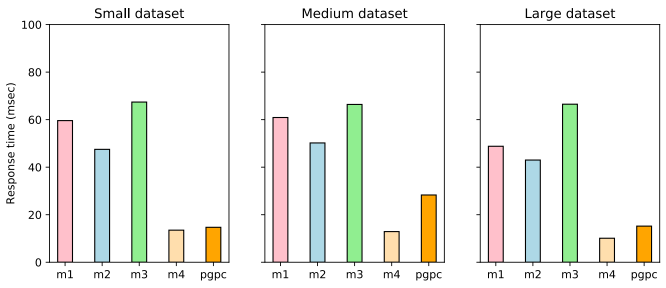


Fig. 11. Cold range query response times for large querying windows

For all 4 models and pgPointCloud, the unit querying costs per point decreased with larger querying windows. A more detailed analysis actually shows that the decrease stops after the querying size reaches to a certain level (e.g. around 30m in the investigated tests). This may be due to some overheads that are independent of the number of resulting points. When more points are returned from larger querying windows, the distribution of the overheads per point gets smaller and becomes insignificant at a certain querying size. The same logic is behind the dissimilarity of the hot and the cold query response times in the cases of large querying windows. In these large queries, the overheads needed in the first query get distributed to more points and becomes insignificant fractions. Notably, the cold queries of pgPointCloud databases does not appear to be slower than subsequent queries. Thus, the first queries of pgPointCloud were sometimes faster than the corresponding queries of all the HBase data models.

5 Concluding remarks

As a demonstration for an implementation of a distributed, non-relational, key-value store for large and high-resolution point cloud data, this paper presents four data models for storage, indexing, and querying point clouds. The four models are constructed from two row-key designs (i.e. Single-Hilbert and Dual-Hilbert) and two column structures (i.e. Separate-Attributes and Grouped-Attributes). The Dual-Hilbert models resemble the flat model approach in RDBMS point cloud storage, while the Single-Hilbert models are largely similar to the standard point block solution. In addition, the Dual-Hilbert codes were used as replacement for the point coordinates. The experimental evaluations of up to 1.4 billion points showed that the flat models are as scalable as the block models within HBase, unlike what has been observed in traditional RDBMS environments. The only notable demerit of the flat models is that they required more storage space and were slower to create initially, without any benefit in the querying speed. The two columns structures, Separate-Attributes and Grouped-Attributes, were compared to evaluate the capability of HBase in supporting flexible data schema. The separation of point attributes to different columns allowed the heterogeneity in point record structure and avoided storage of empty fields. However, doing so in HBase resulted in significant storage overhead as reflected by the sharp increase in the number of key-value pairs and the longer key content. That increase in the number of stored data entities seemed to affect the querying performance in the case of range querying.

Amongst the investigated data models, Model 4, which indexes point data at the block level and preserves the aggregation of the point attributes, appears to be the most competitive solution. The simple structure of Model 4 allows the data to be loaded 7 to 46 times faster than the Dual-Hilbert models (Model 1, Model 2) and at least 1.3 times faster than Model 3. Range queries with Model 4 are from 3 to 8 times faster than the other models while its point query performance is among the highest. Future research will investigate Model 4 further with regard to its capability to support queries that seek for only a subset of point attributes. Since Model 4 does not index the data at the point attribute level as in Model 1 and Model 3, there is a potential that it may not be as effective as the other models in supporting the attribute-specific queries. In addition, heterogeneous datasets (i.e. point data with various attribute structures) will be used to further evaluate the storage efficiency of the data models and explore the schema-less feature of HBase.

The evaluation against pgPointCloud, which is an existing relational database solution, showed that all the HBase data models were faster than pgPointCloud in supporting point queries. With respect to range queries, Model 4 was from 1.5 to 4 times faster than pgPointCloud. However, the other 3 HBase data models were slower than the traditional solution. The result shows that grouping data into blocks and preserving the point record structure are good strategies for encoding point cloud data in HBase. Notably, the unit querying speed per point of the range queries decreased with a larger querying size. The differences between the first (i.e. cold) queries and subsequent (i.e. hot) queries were also reduced when the query size got larger. Due to the built-in parallel mechanism of Hadoop, loading point data into HBase was considerably faster than the pgPointCloud data ingestion despite the requirement of some data preprocessing steps. There was an exception with Model 1 where the extreme indexing decelerated the data ingestion to as much as 1.5 times slower than the pgPointCloud. Finally, these advances do come at a cost. Namely, all the HBase data models consumed more disk space than the pgPointCloud.

In summary, distributed, non-relational databases can be promising for point cloud data storage, because point clouds are weakly relational and do not strictly require transactional consistency. The most significant gains expected from migrating to a non-relational alternative include an improved possibility to scale the system for large amounts of data and better performance due to the inherent parallelism in the framework. The experimental results presented in this paper show that HBase, a representative distributed database, was scalable and faster than the relational PostgreSQL pgPointCloud database when similar data encoding strategies were used (Model 4). The storage of one point per row in HBase (Model 1 and Model 2) did not encounter a scalability issue as previously observed in relational databases [10]. However, they were slower than the storage scheme that groups data into blocks. Future research should consider different techniques to further optimize the performance of both the non-relational and relational solutions. Testing the databases with data of greater volumes and complexity should also be considered.

Acknowledgments. The Hadoop cluster used for the work presented in this paper was provided by allocation TG-CIE170036 - Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562 [30]. The authors would like to thank the staff at Pittsburg Supercomputing Center for the truly outstanding technical support provided during setting up the testing.

References

1. Vo, A.V., Laefer, D.F., Bertolotto M.: Airborne laser scanning data storage and indexing: state of the art review. Int. J. Remote Sens. **37**(24), 6187–6204 (2016). doi: 10.1080/01431161.2016.1256511
2. Kitchin R., McArdle, G.: What makes Big Data, Big Data? Exploring the ontological characteristics of 26 datasets. Big Data Soc. **3**(1), 1-10 (2016). doi: 10.1177/2053951716631130
3. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. In: Proc. 19th ACM Symp. Oper. Syst. Princ., pp. 29-43. New York (2003)
4. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM. **51**(1), pp. 107–113 (2004). doi: 10.1145/1327452.1327492
5. White, T., Hadoop The Definitive Guide, 4th ed. O'Reilly (2015)
6. Chang, F. et al.: Bigtable: a distributed storage system for structured data. ACM Transactions on Computer Systems. **26**(2), 4 (2008), doi: 10.1145/1365815.1365816
7. George, L.: HBase The Definitive Guide, 1st ed. O'Reilly, (2011)
8. Middleton, W., Spilhaus, A.: The measurement of atmosferic humidity. In: Meteorological Instruments, pp. 105–111, Toronto (1953)
9. Shepherd, E.C.: Laser to watch height," New Scientist, **1**(33) (1965)
10. van Oosterom, P. et al.: Massive point cloud data management: design, implementation and execution of a point cloud benchmark. Comput. & Graph., **49**, 92–125 (2015). doi: 10.1016/j.cag.2015.01.007
11. Cura, R., Perret, J., Paparoditis, N.: A scalable and multi-purpose point cloud server (PCS) for easier and faster point cloud data management and processing. ISPRS J. Photogramm. Remote Sens. **127**, 39–56 (2017). doi: 10.1016/j.isprsjprs.2016.06.012
12. Krishnan, S., Baru, C., Crosby, C.: Evaluation of MapReduce for gridding LIDAR data. In: 2010 IEEE Second Int. Conf. Cloud Comput. Technol. Sci., 33–40 (2010). doi: 10.1109/CloudCom.2010.34
13. Li, Z., Hodgson, M. E., Li, W.: A general-purpose framework for parallel processing of large-scale LiDAR data vol. 8947. Int. J. Digit. Earth, **11**(1), 26-47 (2017). doi: 10.1080/17538947.2016.1269842
14. Rizki, P.N.M., Eum, J., Lee, H., Oh, S.: Spark-based in-memory DEM creation from 3D LiDAR point clouds. Remote Sens. Lett., **8**(4), 360–369, (2017). doi: 10.1080/2150704X.2016.1275053
15. Hamraz, H., Contreras, M.A., Zhang, J.: A scalable approach for tree segmentation within small-footprint airborne LiDAR data. Comput. Geosci., 8(4), 360-369, (2017). doi: 10.1080/2150704X.2016.1275053
16. Aljumaily, H., Laefer, D.F., and Cuadra, D.: Urban point cloud mining based on density clustering and MapReduce. J. Comput. Civ. Eng., **31**(5), (2017). doi: 10.1061/(ASCE)CP.1943-5487.0000674
17. Moler, C.: Matrix computation on distributed memory multiprocessors. In: Hypercube multiprocessors 1986, pp. 181–195 (1987).
18. Baumann, P. et al.: Big Data analytics for Earth sciences: the EarthServer approach. Int. J. Digit. Earth, **9**(1), 3–29, (2015). doi: 10.1080/17538947.2014.1003106
19. Boehm, J., Liu, K.: NoSQL for storage and retrieval of large LiDAR data collections. In: ISPRS - Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci., **XL-3**/W3, pp. 577–582, La Grande Motte (2015).
20. Martinez-Rubi, O. et al.: Benchmarking and improving point cloud data management in MonetDB. SIGSPATIAL Special - Big Spatial, **6**(2), pp. 11-18, (2014). doi: 10.1145/2744700.2744702

21. Whitby, M., Fecher, R., Bennight, C.: GeoWave: utilizing distributed key-value stores for multidimensional data. In: Gertz M. et al. (eds) Advances in Spatial and Temporal Databases, LNCS 10411, pp. 105–122. doi: 10.1007/978-3-319-64367-0
22. Mosa, A. S. M., Schön, B., Bertolotto, M., Laefer, D. F.: Evaluating the benefits of octree-based indexing for LiDAR data. Photogramm. Eng. Remote Sens., **78**(9), 927–934 (2012). doi: 10.14358/PERS.78.9.927
23. Ramsey, P.: LiDAR in PostgreSQL with PointCloud. In: FOSS4G, Nottingham (2013).
24. Nandigam, V., Baru, C., Crosby, C.: Database design for high-resolution LIDAR topography data. In: Gertz M., Ludascher, B. (eds) Scientific and Statistical Database Management, LNCS6187, Springer Berlin Heidelberg, pp. 151–159 (2010)
25. Murray, C. et al., Oracle Spatial and Graph - developer' s guide, 12c Release 1. (2017). https://docs.oracle.com/database/121/SPATL/toc.htm
26. Vo, A.-V.: Spatial data storage and processing strategies for urban laser scanning. PhD Thesis. University College Dublin, (2017). doi: 10.13140/RG.2.2.12798.48962
27. Haverkort, H., van Walderveen, F.: Locality and bounding-box quality of two-dimensional space-filling curves. Computational Geometry. **43**(2), 131-147, (2008). doi: 10.1016/j.comgeo.2009.06.002
28. Wang, J., Shan, J.: Space-filling curve based point clouds index. In: Proc. 8th Int. Conf. Geo-Computation, Michigan, (2005).
29. Psomadaki, S., van Oosterom, P. J. M., Tijssen, T. P. M., Baart, F.: Using a space filling curve approach for the management of dynamic point clouds. In: ISPRS Ann. Photogramm. Remote Sens. Spat. Inf. Sci., **IV-2**/W1, pp. 107–118, (2016). doi: 10.5194/isprs-annals-IV-2-W1-107-2016
30. Towns J., Cockerill T., Dahan M., Foster I., Gaither K., Grimshaw A., Hazlewood V., Lathrop S., Lifka D., Peterson G.D., Roskies R., Scott J.R., Wilkins-Diehr N.: XSEDE: Accelerating Scientific Discovery, Computing in Science & Engineering, vol.16, no. 5, pp. 62-74, Sept.-Oct. (2014). doi:10.1109/MCSE.2014.80