remote sensing

MDPI

Article

Strategies for the Storage of Large LiDAR Datasets—A Performance Comparison

Juan A. Béjar-Martos, Antonio J. Rueda-Ruiz *, Carlos J. Ogayar-Anguita and Rafael J. Segura-Sánchez and Alfonso López-Ruiz

Cntro de Estudios Avanzados en TIC, University of Jaén, 23071 Jaén, Spain; jabm0010@red.ujaen.es (J.A.B.-M.); cogayar@ujaen.es (C.J.O.-A.); rsegura@ujaen.es (R.J.S.-S.); allopezr@ujaen.es (A.L.-R.)
* Correspondence: ajrueda@ujaen.es

Abstract: The widespread use of LiDAR technologies has led to an ever-increasing volume of captured data that pose a continuous challenge for its storage and organization, so that it can be efficiently processed and analyzed. Although the use of system files in formats such as LAS/LAZ is the most common solution for LiDAR data storage, databases are gaining in popularity due to their evident advantages: centralized and uniform access to a collection of datasets; better support for concurrent retrieval; distributed storage in database engines that allows sharding; and support for metadata or spatial queries by adequately indexing or organizing the data. The present work evaluates the performance of four popular NoSQL and relational database management systems with large LiDAR datasets: Cassandra, MongoDB, MySQL and PostgreSQL. To perform a realistic assessment, we integrate these database engines in a repository implementation with an elaborate data model that enables metadata and spatial queries and progressive/partial data retrieval. Our experimentation concludes that, as expected, NoSQL databases show a modest but significant performance difference in favor of NoSQL databases, and that Cassandra provides the best overall database solution for LiDAR data.

Keywords: LiDAR; point clouds; databases; NoSQL

Citation: Béjar-Martos, J.A.; Rueda-Ruiz, A.J.; Ogayar-Anguita, C.J.; Segura-Sánchez, R.J.; López-Ruiz, A. Strategies for the Storage of Large LiDAR Datasets—A Performance Comparison. *Remote Sens.* **2022**, *14*, 2623. https://doi.org/10.3390/rs14112623

Academic Editors: Jorge Delgado García, Fayez Tarsha Kurdi and Tarig Ali

Received: 12 April 2022
Accepted: 29 May 2022
Published: 31 May 2022

1. Introduction

LiDAR scanning has become an indisputable tool in fields such as civil engineering, surveying, archaeology, forestry or environmental engineering. The widespread use of terrestrial and airborne LiDAR, powered by the fast evolution of scanning technology, is generating an unprecedented amount of data. For instance, the scanning speeds of one million points per second with an accuracy in the range of 3–5 mm have become common today in terrestrial scanning [1]. Handling such a massive amount of data poses multiple challenges related to its storage, transmission, organization, visualization, edition and analysis, which are actually common to most Big Data applications [2].

LiDAR information has traditionally been stored and exchanged through system files usually from standard formats such as LAS/LAZ. However, the use of databases potentially has many advantages such as the centralized access, complex queries based on metadata, spatial organization of data and distributed storage in databases that support sharding. NoSQL databases are extensively used in Big Data applications because of their performance in simple retrieval operations, their flexible schema and their ability to scale horizontally.

Storing large raw LiDAR data files in a database does not provide any advantages over system files. However, as stated above, the design of an appropriate database scheme or the use of a spatial extension [3] allows organizing LiDAR data into a spatial data structure such as a regular grid, quadtree or octree. This enables spatial data queries and the selective/progressive transmission of data to clients. This organization requires splitting

the original LiDAR dataset into data blocks that are associated with the corresponding tree nodes or grid cells. The size of the data block can have a remarkable impact on the performance of the database.

The main objective of the present work was to provide guidance to those responsible for the design and implementation of LIDAR information storage and processing systems on the performances of different database management systems. Four different NoSQL and relational systems (Cassandra, MongoDB, MySQL PostgreSQL) were compared in terms of the performance of their upload and retrieval operations in single and concurrent scenarios. A complex database schema was used, with two nested spatial data structures and three different data block sizes, following the conceptual model of SPSLiDAR [4].

The rest of this paper is organized as follows: Section 2 reviews some of the extensive existing literature related to the storage of LiDAR information; Section 3 describes the dataset used in the experimentation, the conceptual data model of SPSLiDAR, the features of each database management system evaluated and the implementation of the conceptual data model in them; Section 4 provides the details of the experimental comparison carried out and outlines the results that are discussed in depth in Section 5. Finally, Section 6 summarizes the conclusions and proposes some future work.

2. Previous Work

LiDAR point clouds are currently a very valuable resource for all types of decision-making processes involving spatial data from the real world. The evolution of LiDAR systems enables the acquisition of massive spatial data models [5] whose volume is steadily increasing. Current sensors allow, using various technologies, obtaining dense information at different scales [6] from small objects to large digital terrain models, integrating them into the Geospatial Big Data [7–10]. Recently, many research papers related to this matter have been published [5,10–13].

The processing of LiDAR point cloud data involves a series of steps that range from its acquisition to the extraction of relevant features, including registering, filtering, segmentation, classification and conversion to other representation schemes. Each of these steps has its own research topic, and in particular, the storage of massive point clouds is one of the most notable. In recent years, with increasing processing power, memory and communication bandwidth, the main challenge with LiDAR data is to make massive information available for use in different applications [14]. In this sense, there are different approaches focusing on mass storage in secondary memory, in a cluster of servers or in the cloud [15–19].

Organizing 3D point clouds with data structures that subdivide space is a common solution for increasing the performance of spatial queries. However, from a storage point of view, it also brings the benefit of dividing a potentially massive amount of data into smaller chunks that can be more efficiently managed, which is of key importance when transmitting data over a network. The most widely used data structure for out-of-core point cloud storage is the octree [20–23]. Other widely used structures are the Kd-tree [24], variants of the R-tree [25], a sparse voxel structure [16] and simple grids [26].

Most spatial data structures can be adapted for out-of-core storage, and in this regard, there are different variants and approaches. This is of utmost importance, since today it is common to process LiDAR datasets that do not fit into main memory. The simplest approach is direct storage in secondary memory using local files [15,19]. Another variant consists of using storage in distributed file systems [27,28] typically indexed through a spatial structure hosted in the form of a master index or in a database. LiDAR datasets usually comprise one or more point cloud files encoded in non-standard ASCII formats, the LAS format of the American Society for Photogrammetry and Remote Sensing (ASPRS), its compressed variant LAZ [29], SPD [30], PCD, HDF5 and other general 3D data formats such as OBJ or PLY. In addition, there are some proposals for specific formats, closely related to applications which involve additional point attributes and custom compression

algorithms. Understandably, file formats that focus on data compression are the most useful [11,29,31].

Although file-oriented storage is the simplest and most common option, using databases is the most versatile solution. An original way of storing a point cloud model in a database is to use one row for each point in a relational database, including all its attributes. In the past, this approach was common among GIS applications where sets of points of moderate size were stored in a spatial database to support spatial queries, mainly 2D. However, this scheme is not valid for a large amount of data [32], so the next logical solution is to store a group of points in each row, as systems such as Oracle Spatial or PostGIS do. In addition to this, there are other solutions based on relational databases to work with point clouds and spatial information [8,21].

On the other hand, NoSQL databases have some advantages over relational databases in Big Data applications. Among these, document-oriented databases (MongoDB, Cassandra, Couchbase, etc.) have gained popularity due to their capability to efficiently handle large volumes of data and are scalable through the use of sharding. Because of this, they are usually the preferred option for storing point clouds with a Big Data approach for both semi-structured and unstructured datasets [26,28,32]. However, the everlasting discussion about the theoretical convenience of using the modern NoSQL approach instead of relational databases for Big Data problems should not diminish the relevance of the latter, particularly considering their maturity and widespread adoption [33]. In the case of point cloud datasets, including LiDAR, several approaches use relational database management systems (DBMSs) for processing information [21,34]. The most relevant part is the data model of a DBMS that defines the logical structure of a database and that determines how the data can be stored, organized, and retrieved. In this work, we use SPSLIDAR [4], a data model with a reference implementation for a LiDAR data repository that can be used with any spatial indexing.

3. Materials and Methods

In order to study the performance of the different database engines integrated into SPSLiDAR, various datasets belonging to the city of Pamplona (Spain) were used. We chose this particular area because of the public availability of LiDAR data acquired at different moments and densities (0.5–10 points/m^2). The areas covered by the LiDAR datasets are located in the UTM zone 30N and were captured over 6 years to include a variety of terrains (urban, rural, roads, etc.). Table 1 summarizes the characteristics of the four datasets, and Figure 1 shows a partial rendering of Dataset 1.

Table 1. Datasets characteristics.

	Dataset 1	**Dataset 2**	**Dataset 3**	**Dataset 4**
Name	Pamplona 2011	Pamplona 2017b (reduced version of dataset 4)	Pamplona 2017c (reduced version of dataset 4)	Pamplona 2017
Number of points	30,401,539	244,894,563	534,073,172	1,068,146,345
Density (points/m^2)	0.5	2.5	5	10
Grid size (meters)	10,000	10,000	10,000	10,000
Size (MBs)	138.29	1850.21	4085.94	8054.36
Point data record length	34	38	38	38
LAS point data record format	3	8	8	8

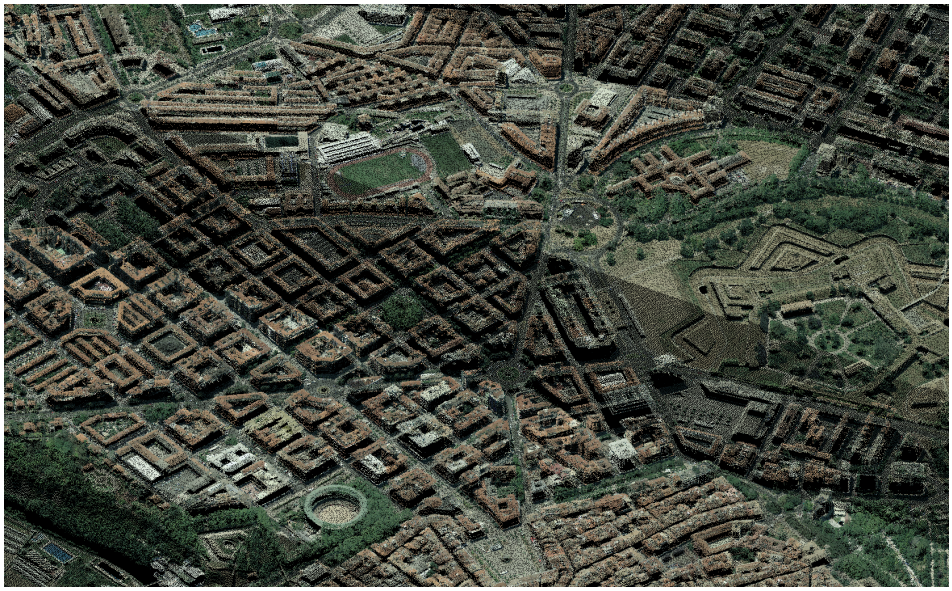

Figure 1. Partial view of Dataset 1 (Pamplona 2011).

3.1. SPSLiDAR

In previous work, a high-level conceptual model for a repository for LiDAR data, namely SPSLiDAR, was proposed [4]. The conceptual model of SPSLiDAR, depicted in Figure 2, comprises three entities: *workspace*, *dataset* and *datablock*. A workspace represents a set of related datasets, such as, for instance, those generated by the LiDAR coverage of several counties or provinces, or different scanning campaigns from related archaeological sites. A dataset comprises one or more point clouds acquired at a particular site. The points of a dataset are organized into a structure of datablocks starting at one or more root datablocks. SPSLiDAR exposes its functionalities to the clients through a restful API, allowing queries by spatial and temporal criteria, and the progressive download of partial or complete datasets.

The original paper of SPSLiDAR proposed a reference implementation that used a regular grid to organize datasets and an octree for the datablocks within a dataset. MongoDB was chosen as the underlying database engine due to its flexibility, high performance and sharding capabilities that allow handling up to petabytes of information distributed across multiple data servers. The characteristics of the database management system have a significant influence on the performance of SPSLIDAR, since each time a dataset is sent to the system, one or more octrees are generated, and each one may comprise hundreds of thousands of nodes and LAZ files that must be efficiently stored in the database—both in terms of occupied time and space. Subsequently, the SPSLiDAR API enables the navigation of the dataset octrees and the retrieval of the point clouds in the LAZ files associated with the nodes of interest. The ability of the database to fetch the desired datablocks and transmit the content of the associated point cloud files has a direct impact on latency times and system throughput, especially considering that usually several clients may interact with the system at the same time.

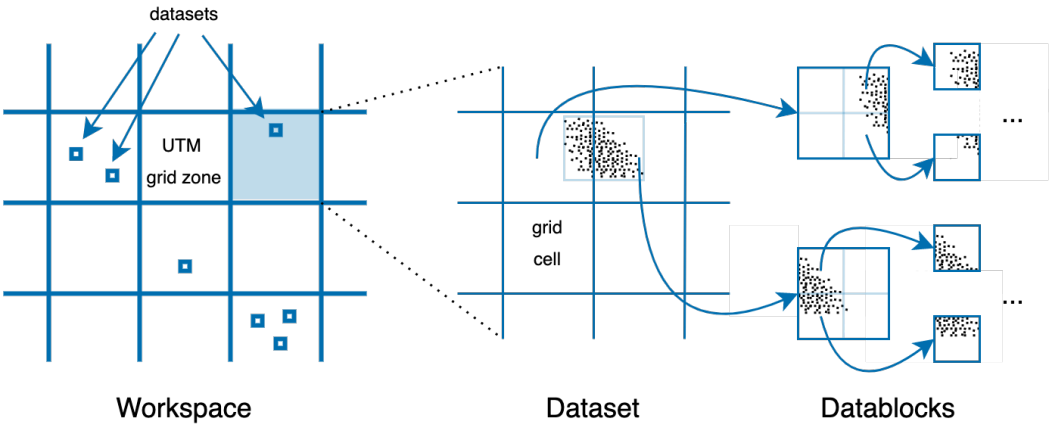

Figure 2. Spatial data structures at the workspace and dataset levels. For the sake of clarity, octrees are depicted as quadtrees at the dataset level.

3.2. Databases

In our experimentation, we used the original MongoDB-based implementation of the SPSLiDAR and three different adaptations corresponding to the rest of the database management systems evaluated. Even though the storage of large LiDAR datasets fits the definition of Geospatial Big Data, we did not want to restrict ourselves only to non-relational databases, usually considered the standard storage solution for applications in this domain. Relational databases have also been used for the storage of massive amounts of data [9,35] and may therefore meet the needs of our use case.

MongoDB, Cassandra, PostgreSQL and MySQL were the four technologies adopted. All of them are well established products. Among non-relational databases, MongoDB provides a document-oriented solution while Cassandra follows a wide-column approach. In the category of relational databases, PostgreSQL and MySQL are among the most widely adopted. In order to make the fairest comparison, the SPSLiDAR conceptual model was implemented in each database in the most straightforward way.

3.2.1. MongoDB

MongoDB is a non-relational document-oriented database that uses JSON to encode information. Documents in MongoDB present a flexible schema so that two documents representing a similar concept can have different fields and new attributes may be added or removed at runtime. MongoDB introduces the concept of collection as a means of grouping documents which represent the same concept or type of information.

The GridFS specification is provided by MongoDB as a means of storing binary files that surpass the 16 MB per document limit imposed by the system. GridFS uses two collections to store files. One collection stores the file chunks (*fs.chunks*) and the other stores file metadata (*fs.files*). Each file is decomposed into multiple chunks stored in the *fs.chunks* collection, each one containing the binary data of the corresponding section of the file, an order attribute that specifies its position in the sequence of chunks and a reference to the file document in the *fs.files* collection. By default, the maximum amount of data per chunk is set to 255 KB. The GridFS collections are depicted in Figure 3.

The storage of LAZ files in the database is one of the key points of the implementation as it has a direct effect on performance. From a technical point of view, we could save these files through GridFS or embedding the content in a document. We decided to use GridFS as the standard way of storing files, since this enables the construction of octrees with any arbitrary node size. A hybrid approach could also be followed to store the LAZ file through an embedded field if it does not surpass the 16 MB size limit for MongoDB documents, or via GridFS otherwise. This hybrid approach can easily be implemented in MongoDB thanks to its flexible document schema, which allows encoding the file data in a BSON

(Binary JSON) field or alternatively, to include a *DBRef* field with the *_id* of the document saved in the *fs.files* collection of GridFS.

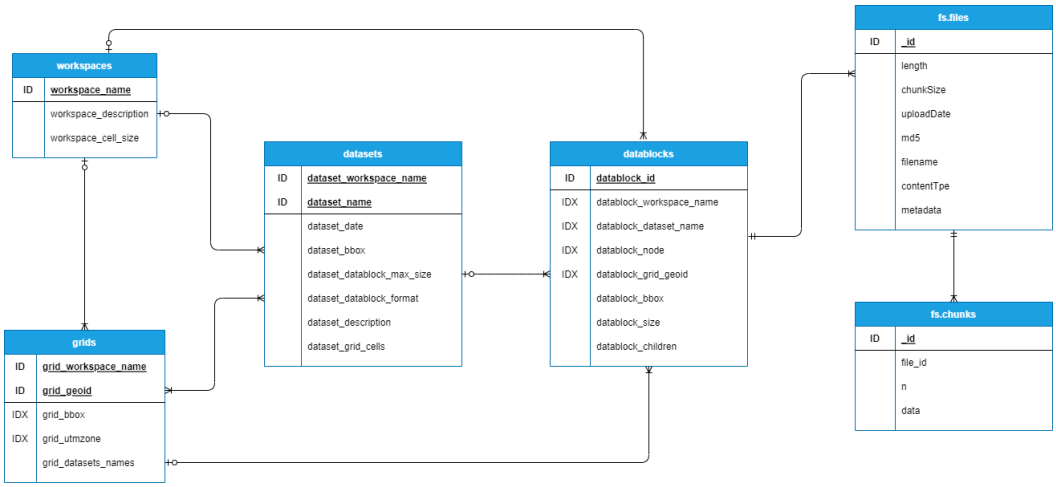

Figure 3. Database design for MongoDB. PK refers to primary key and FK refers to foreign key.

The final structure of documents and collections defined in MongoDB is depicted in Figure 3, showing the relationships among the different entities. The details of the data model, including the description of the *workspace*, *dataset* and *datablock* entities can be found in an original paper describing SPSLiDAR [4]. A mainly denormalized approach was followed: for instance, a datablock document will contain redundant information that refers to the workspace, dataset, and grid cell it belongs to. These fields were indexed in order to accelerate queries since datablock fetching is a crucial operation to access LAZ data files.

3.2.2. Cassandra

Cassandra is a non-relational database with a partitioned row model. Rows are stored in different partitions, identified by the partition key (PK) which may consist of one or more columns. A partition may contain more than one row; in these cases, a clustered key (CK) is also needed, which may also consist of more than one column and uniquely identifies each row inside a partition. Therefore, a single row is defined by a composite key formed by a partition key and optionally a clustered key.

No analogous specification to MongoDB GridFS exists in Cassandra; therefore, we implemented a custom system that follows the same basic concepts. A new entity called chunk was defined, which contains a column with binary data and is uniquely identified by a composite key formed by the partition key of the datablock it belongs to and an additional *chunk_order* attribute which identifies the position of the chunk content in the file data. As a result, all the chunks belonging to the same file are stored on the same partition and the *chunk_order* attribute facilitates a complete recovery of the chunks in ascending order, enabling a straightforward reconstruction of the file. The maximum size of content for each chunk was set to 255 KB—the same default value used by GridFS.

Figure 4 shows the tables and the columns defined for the persistence of our data model in Cassandra. As we did in MongoDB, a denormalized approach was followed. Cassandra recommends a model design based on the queries that will be performed, with data duplication encouraged to increase reading efficiency.

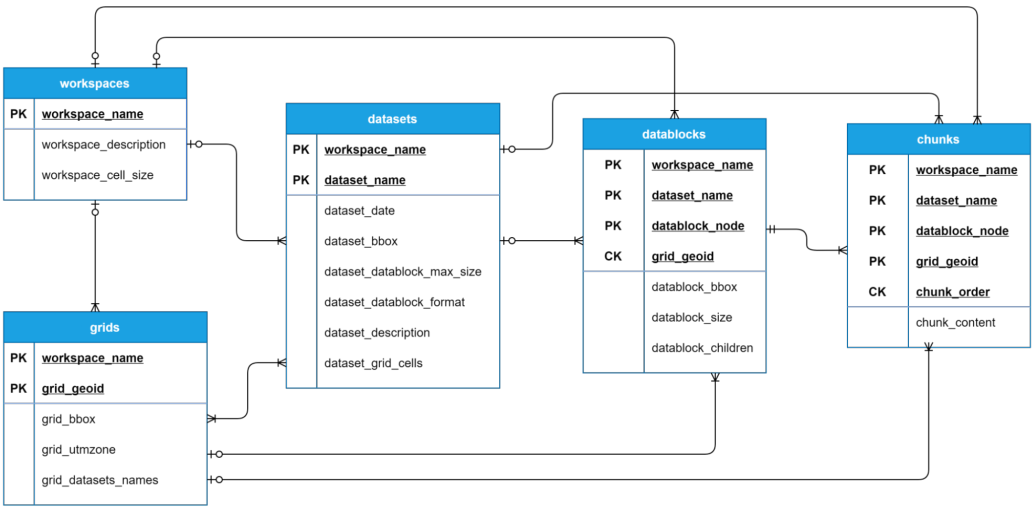

Figure 4. Database design for Cassandra implementation. PK refers to primary key and FK to foreign key.

3.2.3. Relational Databases

Regarding relational databases, we integrated two different alternatives into the system: namely PostgreSQL and MySQL. Due to their similarities, we used almost the same implementation on the persistence layer, only modifying the types so that they adjust to each database specification.

In the same way as with Cassandra, we implemented a tailored solution for the storage of files. The proposed solution defines a new entity called *chunk*, with a composite primary key formed by the primary key of its associated datablock together with an additional attribute that represents the position of the attached binary data block in the file. Normalization is usually a requirement in relational database schemas, but in order to avoid table parameters that may adversely affect performance, we denormalized both the datablocks and chunks entities. We consider that the time performance outweighs the disadvantages of a denormalized schema for these particular tables since they are expected to contain a large number of rows and support a high number of queries. In Figure 5, an entity-relational diagram of the model is shown for both relational databases.

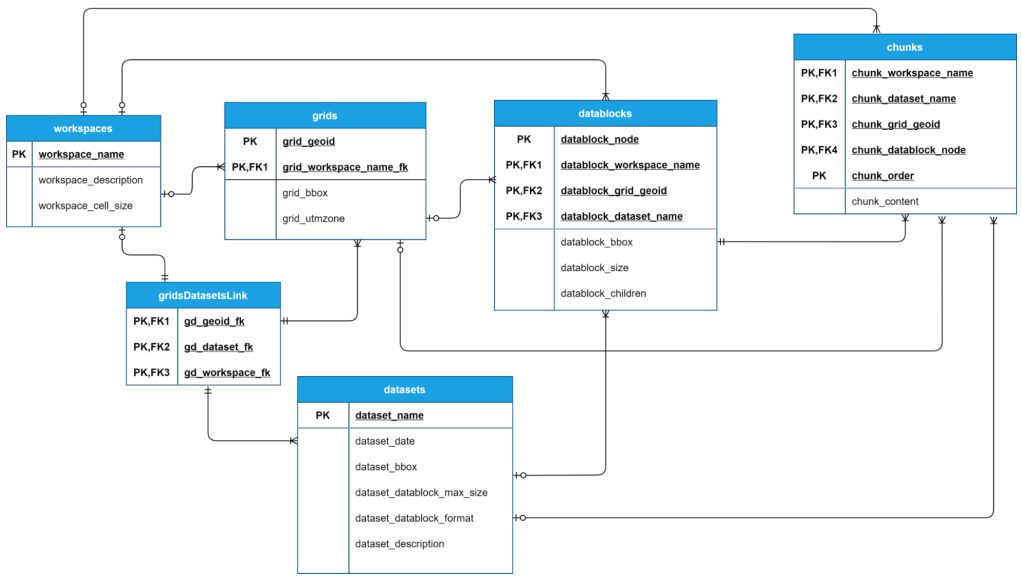

Figure 5. Database design for MySQL and PostgreSQL implementations. PK refers to the primary key and FK to foreign key.

4. Experiments and Results

To evaluate the performance of each of the databases with large LiDAR datasets, three experiments were performed. These experiments are based on those proposed in [36]. The first one (upload test) evaluates the upload time and total storage space required by the four datasets. The second test (simple requests test) measures the average download time for 1M points from a single client. For this purpose, multiple random LAZ files are requested until completing 20 M points, computing the average from the total download time. The third test (concurrent requests test) evaluates the response of the system to high workloads by measuring the average download time for multiple concurrent requests of 1M points from several clients.

The upload test and simple requests test were both implemented through a Python 3.9 script that uses the *Requests* library [37] in order to perform the petitions to the server. The concurrent requests test was developed using the *LoadTest* JavaScript package for Node.js [38]. All operations were performed through the SPSLiDAR API. The tested data correspond to the four datasets described in Table 1. For each of the datasets, three different maximum datablock sizes were tested: 10,000, 100,000, and 1,000,000 points. These values define the maximum number of points of the LAZ file stored at each node of the octree, and consequently determine its complexity since a lower or higher number of levels would be required to store the dataset.

All experiments were carried out by running the different implementations of the SPSLiDAR repository on a server with Intel i7-10700 Octa-Core at 2.9GHz, 16 GB RAM, and 1 TB SSD drive, running Windows 10 Enterprise 64. All databases were deployed through Docker using images with MongoDB 4.2.10, Cassandra 3.11.0, MySQL 8.0.24, and PostgreSQL 13.2. The client computer was a Mackbook Pro laptop with an Intel Core i7-4770HQ Quad-Core at 2.2 GHz with 16 GB RAM and a 256 GB SSD Drive, running macOS Big Sur. The connection between the server and client was made through a 1 GB/s Ethernet cable network.

4.1. Upload Test

A dataset upload comprises two stages. First, the original dataset is preprocessed and subdivided into a set of small LAZ files whose size is limited by the maximum datablock size. These files are indexed through octrees that represent the original point cloud. Second, the files and the octree metadata are stored in the database. The first stage is highly computationally demanding and consumes most of the time. Therefore, the influence of the database performance on the overall time is limited. The preprocessing stage is deterministic and is external to the persistence layer, resulting in the insertion of the same number of files in each database. Furthermore, note that, in practice, this upload operation is only to be carried out once per dataset. For this reason, we consider that the results of this test should have the least weight in the choice of the database.

Table 2 shows the average upload times for each combination of dataset and the maximum datablock size through three executions. The biggest differences among databases occurred in Dataset 4 as the octree structure generated is the most complex, with the highest number of LAZ files to store. Figure 6 shows a relative comparison of the upload times of Cassandra, PostgreSQL and MySQL with respect to MongoDB. We compared them against MongoDB because this was used in the reference implementation of SPSLiDAR.

Table 2. Dataset upload times (in seconds) for the implementations based on MongoDB, Cassandra, PostgreSQL and MySQL.

Dataset	Maximum Datablock Size	MongoDB	Cassandra	PostgreSQL	MySQL
Dataset 1	1,000,000	67	68	69	70
	100,000	218	211	224	213
	10,000	1444	1373	1553	1411
Dataset 2	1,000,000	746	793	824	727
	100,000	2089	2027	2016	1993
	10,000	13,962	12,791	12,650	12,942
Dataset 3	1,000,000	1945	1806	1899	2191
	100,000	3906	3843	3897	3962
	10,000	25,176	24,528	26,823	23,307
Dataset 4	1,000,000	4131	4220	4150	4816
	100,000	8460	8258	8887	8575
	10,000	45,035	41,290	49,714	42,018

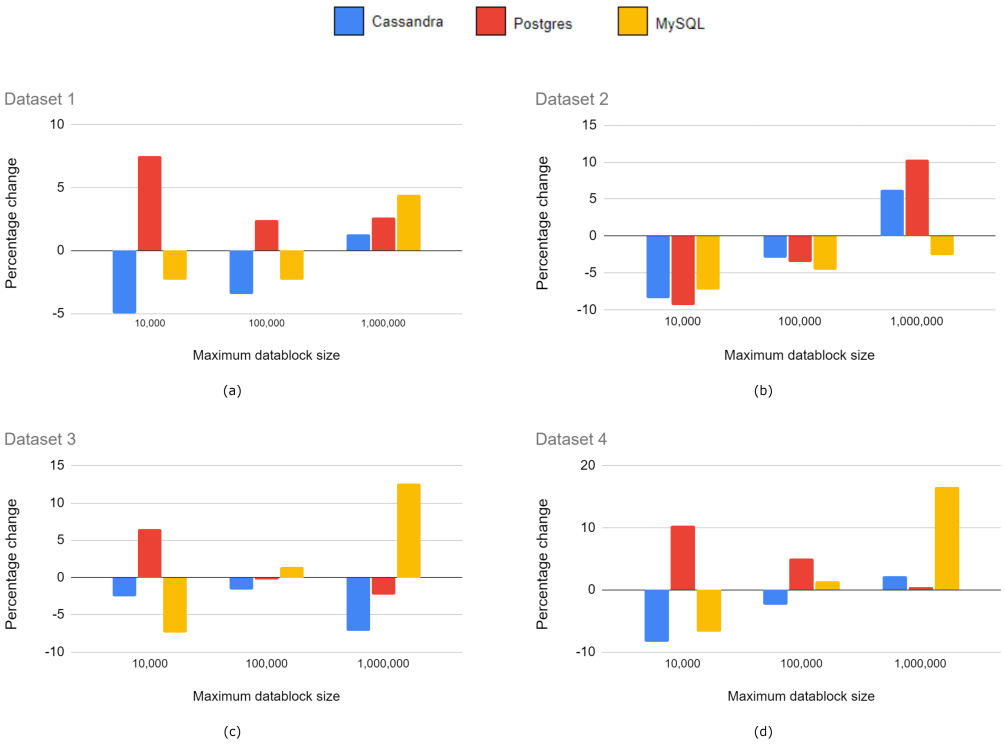

Figure 6. Relative comparison of upload times. Results for (**a**) Dataset 1; (**b**) Dataset 2; (**c**) Dataset 3; and (**d**) Dataset 4.

Table 3 shows the final storage space required at the database level by the datasets after the upload tests. This includes the LAZ files generated in the process in order for all the entities necessary for the system to work correctly. The results shown are the average of three upload operations, although we observed low variance among the results. Figure 7 depicts the information of Table 3 as a relative comparison in storage size with respect to MongoDB.

Table 3. Storage required (in MB) by the implementations based on MongoDB, Cassandra PostgreSQL and MySQL.

Dataset	Maximum Datablock Size	MongoDB	Cassandra	PostgreSQL	MySQL
Dataset 1	1,000,000	198	197	214	224
	100,000	202	200	217	220
	10,000	235	224	247	262
Dataset 2	1,000,000	1875	1882	1960	2118
	100,000	1893	1884	1970	2109
	10,000	2162	2056	2262	2603
Dataset 3	1,000.000	4233	4243	4413	4337
	100,000	4313	4308	4486	4365
	10,000	5062	4753	5377	6130
Dataset 4	1,000.000	8265	8291	8609	9116
	100,000	8388	8478	8726	9397
	10,000	9893	9160	10,270	12,406

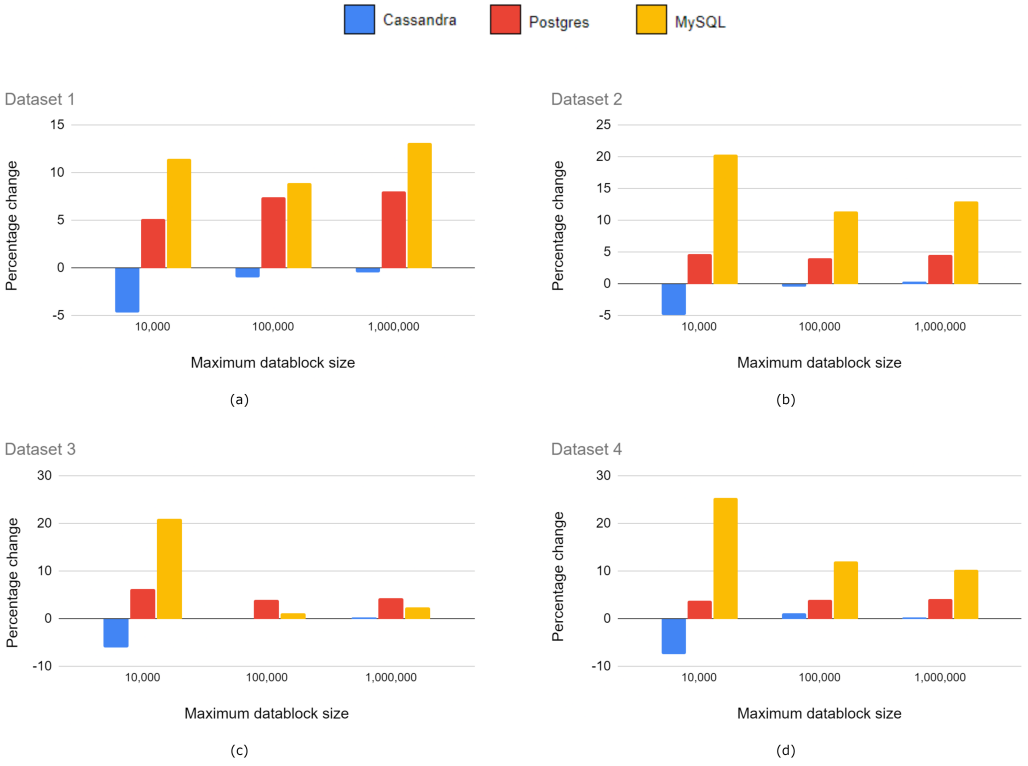

Figure 7. Relative comparison of the required storage. Results for (**a**) Dataset 1; (**b**) Dataset 2; (**c**) Dataset 3; and (**d**) Dataset 4.

4.2. Simple Requests Test

Table 4 shows a relative comparison of the average times required for reading 1M points from a single client in the different implementations tested. The results are also graphically summarized in Figure 8.

Table 4. Average times (in seconds) of a request operation of 1M points in the implementations based on MongoDB, Cassandra, PostgreSQL and MySQL.

Dataset	Maximum Datablock Size	MongoDB	Cassandra	Postgres	MySQL
Dataset 1	1,000,000	0.107	0.11	0.264	0.268
	100,000	0.333	0.303	0.965	0.425
	10,000	2.547	1.824	5.997	1.72
Dataset 2	1,000,000	0.149	0.222	0.308	0.324
	100,000	0.701	0.438	0.992	0.4
	10,000	3.718	3.036	9.48	2.792
Dataset 3	1,000,000	0.17	0.237	0.304	0.469
	100,000	0.429	0.332	1.094	0.551
	10,000	3.559	2.453	8.991	2.223
Dataset 4	1,000,000	0.348	0.188	0.304	0.572
	100,000	0.452	0.329	1.042	0.641
	10,000	3.663	2.701	6.859	2.894

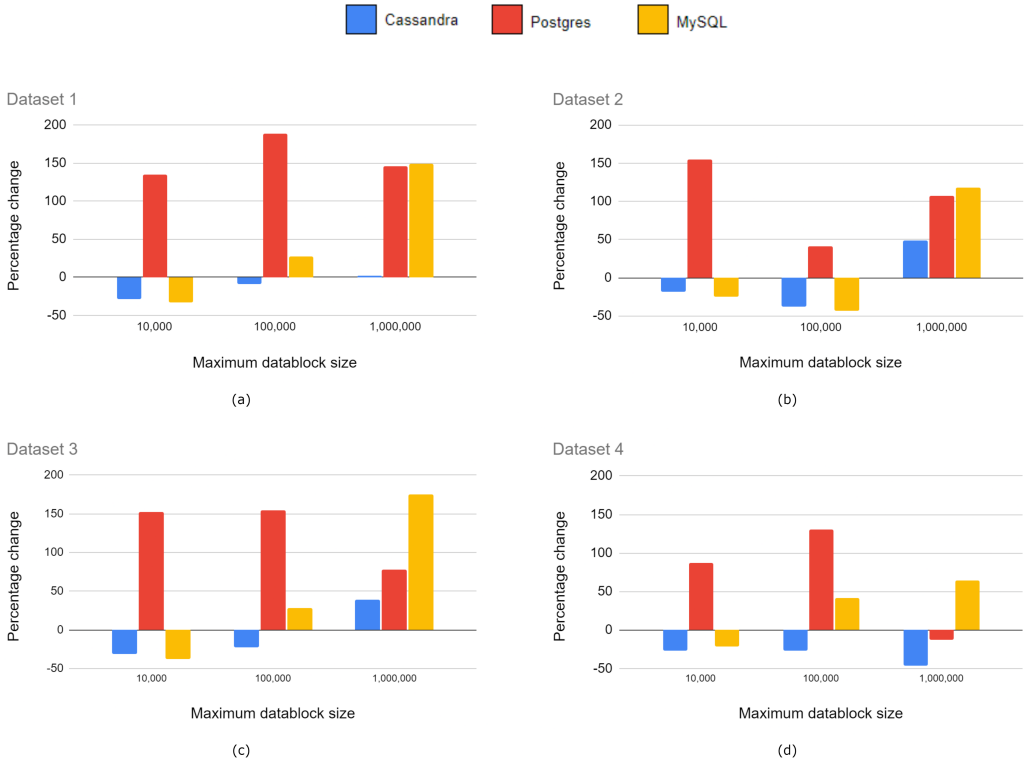

Figure 8. Relative comparison of the results of the simple requests test with (**a**) Dataset 1; (**b**) Dataset 2; (**c**) Dataset 3; and (**d**) Dataset 4.

4.3. Concurrent Requests Test

The third experiment evaluates the response time for several concurrent requests of 1M points from different clients. Tables 5–8 show the results obtained for each implementation with 10 and 100 concurrent clients, respectively. For the sake of simplicity, requests are only performed on Dataset 1. Figure 9 graphically compares the throughput of the different implementations.

Table 5. Results of the concurrent requests test in Cassandra.

Concurrent Users	Maximum Datablock Size	Average Request Time (s)	Maximum Request Time (s)	Throughput (Requests per Second)	Total Time (s)
10	1,000,000	1.085	1.512	7	1.532
	100,000	0.174	0.23	55	1.831
	10,000	0.025	0.074	386	2.59
100	1,000,000	13.36	18.572	5	18.61
	100,000	1.715	2.589	56	17.73
	10,000	0.237	0.45	419	23.86

Table 6. Results of the concurrent requests test in MongoDB.

Concurrent Users	Maximum Datablock Size	Average Request Time (s)	Maximum Request Time (s)	Throughput (Requests per Second)	Total Time (s)
10	1,000,000	1.653	1.68	6	1.691
	100,000	0.252	0.42	39	2.5705
	10,000	0.038	0.076	259	3.8628
100	1,000,000	1.614	1.657	6	16.164
	100,000	2.079	3.823	47	21.4625
	10,000	0.269	0.527	370	27.011

Table 7. Results of the concurrent requests test in PostgreSQL.

Concurrent Users	Maximum Datablock Size	Average Request Time (s)	Maximum Request Time (s)	Throughput (Requests per Second)	Total Time (s)
10	1,000,000	2.386	3.002	3	3.024
	100,000	0.224	0.503	42	2.352
	10,000	0.035	0.4	274	3.646
100	1,000,000	23.3	38.57	3	38.62
	100,000	3.763	16.51	26	38.691
	10,000	0.279	0.597	356	28.071

Table 8. Results of the concurrent requests test in MySQL.

Concurrent Users	Maximum Datablock Size	Average Request Time (s)	Maximum Request Time (s)	Throughput (Requests per second)	Total Time (s)
10	1,000,000	1.547	1.579	6	1.604
	100,000	0.232	0.296	42	2.36
	10,000	0.044	0.113	223	4.493
100	1,000,000	25.422	34.841	3	34.886
	100,000	9.316	23.556	10	95.749
	10,000	0.416	0.633	239	41.86

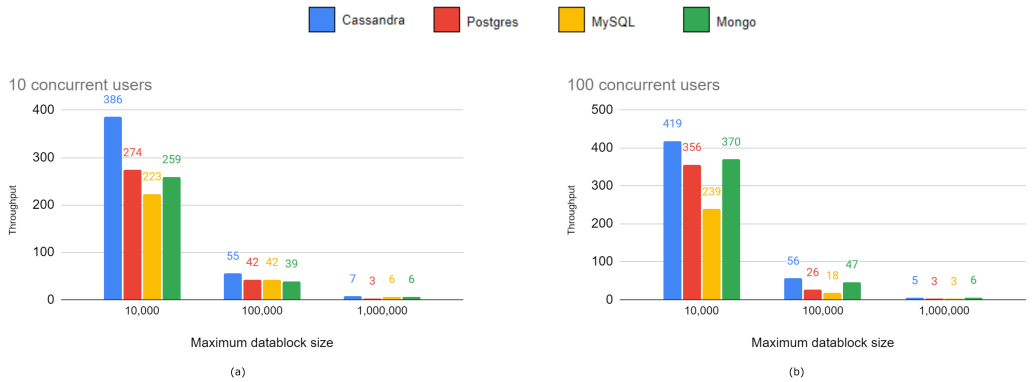

Figure 9. Comparison of the throughput of the different implementations with (**a**) 10 concurrent users and (**b**) 100 concurrent users.

5. Discussion

NoSQL databases are the primary storage option in Big Data applications. Our experiments with LiDAR datasets confirm in general terms the advantage of using NoSQL databases over relational ones, although the performance of MySQL is close or even superior in certain situations. Nevertheless, Cassandra is the clear winner, only beaten by MongoDB when the spatial data structure was organized in large datablocks (1 M). One possible explanation for this fact may be the superior performance of GridFS when handling larger LAZ files. Among all the experiments, the upload tests show less conclusive results, but in general terms, Cassandra and MySQL perform better with small or medium datablocks, and therefore, larger octrees, while MongoDB beats the rest of databases with large datablocks (i.e., smaller octrees). Regarding the storage space required, the two relational databases clearly perform the worst; most notably MySQL requires up to 20% more space than MongoDB when using small datablock sizes. This may be due to a more complex internal organization of the information or a wider use of indexes, which is necessary to be able to respond to more complex queries in the relational model.

In the simple requests test, Cassandra and MySQL showed the lowest latency with smaller and medium datablocks up to 25% better than MongoDB, although the latter was still overall the fastest database with a 1M datablock size. PostgreSQL is clearly the worst option—up to twice as slow as Cassandra in some experiments.

In the third series of experiments related to concurrent requests, Cassandra showed the highest throughput followed by MongoDB. Surprisingly, the MySQL performance was much less satisfactory in situations with concurrent requests than with single requests, coming in last position—even behind PostgreSQL. This is an important weakness of MySQL that may discourage its use in applications which need to support high concurrent workloads.

In summary, our recommendation for any project that requires storage in a database of large LiDAR datasets is Cassandra. Its performance is excellent in all operations. MongoDB can be an interesting alternative when Big Data chunks have to be stored in the database, thanks to the excellent performance of GridFS. Finally, if using relational databases is a requirement of the project, MySQL is the option of choice, although its performance under high concurrent workloads should be carefully observed.

6. Conclusions and Future Work

Databases are a versatile and robust option when storing LiDAR data. In this paper, we tried to shed some light on the most appropriate choice of database management system for this type of data. To this end, we compared the performance of four of the most popular NoSQL and relational database management systems in several areas. Our conclusion is that although NoSQL databases perform better than relational systems, the gap, particularly with MySQL, is narrow. Overall, Cassandra shines in all areas, only lagging behind MongoDB when datasets are split into large blocks.

Many approaches use a database only for storing dataset metadata or spatial indexes, keeping LiDAR data in external files. We store LiDAR data in the database because of its clear advantages: simplicity, guaranteed consistency and distributed storage when using database engines that support sharding. However, a further comparison of the performance of the in-database vs. file system storage would provide useful information for decision making when designing systems working with LiDAR data. The storage and processing of massive volumes of LiDAR data usually requires a distributed architecture. For this reason, a new experimentation similar to that carried out in this paper but using a cluster of database nodes would be relevant. However, the relational databases management systems analyzed are centralized, therefore additional solutions such as MySQL Cluster CGE would be necessary. The use of this extra infrastructure and the decisions made during the installation, configuration and tuning of the cluster and database nodes may influence the experimentation and limit the validity of the results.

Author Contributions: Conceptualization, A.J.R.-R., R.J.S.-S., C.J.O.-A.; methodology, A.J.R.-R., R.J.S.-S., C.J.O.-A.; software, J.A.B.-M.; validation, A.J.R.-R., R.J.S.-S., C.J.O.-A., A.L.-R.; formal analysis, C.J.O.-A., A.L.-R.; investigation, A.J.R.-R., R.J.S.-S., C.J.O.-A.; resources, J.A.B.-M.; data curation, J.A.B.-M., C.J.O.-A.; writing—original draft preparation, J.A.B.-M., R.J.S.-S., C.J.O.-A.; writing—review and editing, A.J.R.-R.; visualization, R.J.S.-S., C.J.O.-A.; supervision, A.J.R.-R., R.J.S.-S., C.J.O.-A.; project administration, R.J.S.-S.; funding acquisition, R.J.S.-S. All authors have read and agreed to the published version of the manuscript.

Funding: This result is part of the research project RTI2018-099638-B-I00 funded by MCIN/ AEI/ 10.13039/501100011033/ and ERDF funds "A way of doing Europe". Also, the work has been funded by the University of Jaén (via ERDF funds) through the research project 1265116/2020.

Data Availability Statement: The datasets used for the experimentation are publicly available from SITNA at the following URL: https://filescartografia.navarra.es/, accessed on 11 April 2022.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Leica Geosystems. ScanStation P40, P30 and P16 Comparison Chart. Available online: https://leica-geosystems.com/en-us/products/laser-scanners/-/media/00ac56bc2a93476b8fe3d7c1795040ac.ashx (accessed on 22 May 2022).
2. Li, S.; Dragicevic, S.; Anton, F.; Sester, M.; Winter, S.; Coltekin, A.; Pettit, C.; Jiang, B.; Haworth, J.; Stein, A.; et al. Geospatial Big Data handling theory and methods: A review and research challenges. *ISPRS J. Photogramm. Remote Sens.* **2015**, *115*, 119–133. [CrossRef]
3. Chen, R.; Xie, J. *Open Source Databases and their Spatial Extensions*; Springer: Berlin/Heidelberg, Germany, 2008; Volume 2, pp. 105–129. [CrossRef]
4. Rueda-Ruiz, A.J.; Ogáyar-Anguita, C.J.; Segura-Sánchez, R.J.; Béjar-Martos, J.A.; Delgado-Garcia, J. SPSLiDAR: Towards a multi-purpose repository for large scale LiDAR datasets. *Int. J. Geogr. Inf. Sci.* **2022**, *36*, 1–20. [CrossRef]
5. Poux, Florent. The Smart Point Cloud: Structuring 3D Intelligent Point Data. Ph.D. Thesis, Universite de Liege, Liege, Belgique, 2019. [CrossRef]
6. Bräunl, T. Lidar sensors. In *Robot Adventures in Python and C*; Springer: Cham, Switzerland, 2020; pp. 47–51. [CrossRef]
7. Evans, M.R.; Oliver, D.; Zhou, X.; Shekhar, S. Spatial Big Data. Case studies on volume, velocity, and varitety. In *Big Data: Techniques and Technologies in Geoinformatics*; CRC Press: Boca Raton, FL, USA, 2014.
8. Lee, J.G.; Kang, M. Geospatial Big Data: Challenges and opportunities. *Big Data Res.* **2015**, *2*, 74–81. [CrossRef]
9. Pääkkönen, P.; Pakkala, D. Reference architecture and classification of technologies, products and services for Big Data systems. *Big Data Res.* **2015**, *2*, 166–186. [CrossRef]
10. Deng, X.; Liu, P.; Liu, X.; Wang, R.; Zhang, Y.; He, J.; Yao, Y. Geospatial Big Data: Ew paradigm of remote sensing applications. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2019**, *12*, 3841–3851. [CrossRef]
11. Sugimoto, K.; Cohen, R.A.; Tian, D.; Vetro, A. Trends in efficient representation of 3D point clouds. In Proceedings of the 2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC), Kuala Lumpur, Malaysia, 12–15 December 2017; pp. 364–369. [CrossRef]
12. Ullrich, A.; Pfennigbauer, M. Advances in LiDAR point cloud processing. In *Laser Radar Technology and Applications XXIV*; Turner, M.D., Kamerman, G.W., Eds.; SPIE: Baltimore, MD, USA, 2019; p. 19. [CrossRef]
13. Ma, X.; Liu, S.; Xia, Z.; Zhang, H.; Zeng, X.; Ouyang, W. Rethinking pseudo-LiDAR representation. In *Computer Vision—ECCV 2020*; Vedaldi, A., Bischof, H., Brox, T., Frahm, J.M., Eds.; Springer International Publishing: Cham, Switzerland, 2020; Volume 12358, pp. 311–327. [CrossRef]

14. Poux, F.; Billen, R. A Smart Point Cloud Infrastructure for intelligent environments. In *Laser Scanning: An Emerging Technology in Structural Engineering*; CRC Press: Boca Raton, FL, USA, 2019.
15. Scheiblauer, C.; Wimmer, M. Out-of-core selection and editing of huge point clouds. *Comput. Graph.* **2011**, *35*, 342–351. [CrossRef]
16. Baert, J.; Lagae, A.; Dutré, P. Out-of-core construction of sparse voxel octrees. *Comput. Graph. Forum* **2014**, *33*, 220–227. [CrossRef]
17. Richter, R.; Discher, S.; Döllner, J. Out-of-core visualization of classified 3D point clouds. In *3D Geoinformation Science*; Breunig, M., Al-Doori, M., Butwilowski, E., Kuper, P.V., Benner, J., Haefele, K.H., Eds.; Springer International Publishing: Cham, Switzerland, 2015; pp. 227–242. [CrossRef]
18. Deibe, D.; Amor, M.; Doallo, R. Supporting multi-resolution out-of-core rendering of massive LiDAR point clouds through non-redundant data structures. *Int. J. Geogr. Inf. Sci.* **2019**, *33*, 593–617. [CrossRef]
19. Schütz, M.; Ohrhallinger, S.; Wimmer, M. Fast out-of-core octree generation for massive point clouds. *Comput. Graph. Forum* **2020**, *39*, 155–167. [CrossRef]
20. Elseberg, J.; Borrmann, D.; Nüchter, A. One billion points in the cloud—An octree for efficient processing of 3D laser scans. *ISPRS J. Photogramm. Remote Sens.* **2013**, *76*, 76–88. [CrossRef]
21. Schön, B.; Mosa, A.S.M.; Laefer, D.F.; Bertolotto, M. Octree-based indexing for 3D point clouds within an Oracle Spatial DBMS. *Comput. Geosci.* **2013**, *51*, 430–438. [CrossRef]
22. Lu, B.; Wang, Q.; Li, A. Massive point cloud space management method based on octree-like encoding. *Arab. J. Sci. Eng.* **2019**, *44*, 9397–9411. [CrossRef]
23. Huang, L.; Wang, S.; Wong, K.; Liu, J.; Urtasun, R. OctSqueeze: Octree-structured entropy model for LiDAR compression. *arXiv* **2020**, arXiv: 2005.07178.
24. Goswami, P.; Erol, F.; Mukhi, R.; Pajarola, R.; Gobbetti, E. An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. *Vis. Comput.* **2013**, *29*, 69–83. [CrossRef]
25. Gong, J.; Zhu, Q.; Zhong, R.; Zhang, Y.; Xie, X. An efficient point cloud management method based on a 3D R-tree. *Photogramm. Eng. Remote Sens.* **2012**, *78*, 373–381. [CrossRef]
26. Hongchao, M.; Wang, Z. Distributed data organization and parallel data retrieval methods for huge laser scanner point clouds. *Comput. Geosci.* **2011**, *37*, 193–201. [CrossRef]
27. Krämer, M.; Senner, I. A modular software architecture for processing of Big Geospatial Data in the cloud. *Comput. Graph.* **2015**, *49*, 69–81. [CrossRef]
28. Deibe, D.; Amor, M.; Doallo, R. Big Data storage technologies: A case study for web-based LiDAR visualization. In Proceedings of the 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 10–13 December 2018; pp. 3831–3840. [CrossRef]
29. Isenburg, M. LASzip. *Photogramm. Eng. Remote. Sens.* **2013**, *79*, 209–217. [CrossRef]
30. Bunting, P.; Armston, J.; Lucas, R.M.; Clewley, D. Sorted pulse data (SPD) library. Part I: A generic file format for LiDAR data from pulsed laser systems in terrestrial environments. *Comput. Geosci.* **2013**, *56*, 197–206. [CrossRef]
31. Cao, C.; Preda, M.; Zaharia, T. 3D point cloud compression: A survey. In Proceedings of the 24th International Conference on 3D Web Technology, Los Angeles, CA, USA, 26–28 July 2019; pp. 1–9. [CrossRef]
32. Boehm, J. File-centric organization of large LiDAR point clouds in a Big Data context. In Proceedings of the Workshop on Processing Large Geospatial Data, Dallas, TX, USA, 4 November 2014.
33. Pandey, R. *Performance Benchmarking and Comparison of Cloud-Based Databases MongoDB (NoSQL) vs. MySQL (Relational) Using YCSB*; National College of Ireland: Dublin, Ireland, 2020. [CrossRef]
34. Cura, R.; Perret, J.; Paparoditis, N. Point Cloud Server (PCS): Point clouds in-base management and processing. *ISPRS Ann. Photogramm. Remote Sens. Spat. Inf. Sci.* **2015**, *II-3/W5*, 531–539. [CrossRef]
35. Migrating Facebook to MySQL 8.0. Available online: https://engineering.fb.com/2021/07/22/data-infrastructure/mysql (accessed on 22 May 2022).
36. van Oosterom, P.; Martinez-Rubi, O.; Ivanova, M.; Horhammer, M.; Geringer, D.; Ravada, S.; Tijssen, T.; Kodde, M.; Gonçalves, R. Massive point cloud data management: Design, implementation and execution of a point cloud benchmark. *Comput. Graph.* **2015**, *49*, 92–125. [CrossRef]
37. Reitz, K. HTTP for Humans(TM)—Requests 2.25.1 Documentation. Available online: https://docs.python-requests.org/ (accessed on 9 March 2021).
38. Fernández, A. Loadtest 5.1.2. Available online: https://www.npmjs.com/package/loadtest (accessed on 9 March 2021).