Working Efficiently with Large Geodata Files using Ad-hoc Queries

Pascal Bormann[1,2][a], Michel Krämer[1,2][b] and Hendrik M. Würz[1,2][c]
[1]*Fraunhofer Institute for Computer Graphics Research IGD, 64283 Darmstadt, Germany*
[2]*Technical University of Darmstadt, 64277 Darmstadt, Germany*

Keywords: Information Retrieval, Searching, Geospatial Data, Building Models, Point Clouds.

Abstract: Working with large geospatial data such as building models or point clouds typically requires an index structure to enable fast queries. Creating such an index is a time-consuming process. Especially in single-user explorative scenarios, as they are often found in the scientific community, creating an index or importing the data into a database management system (DBMS) might be unnecessary. In this position paper, we show through a series of experiments that modern commodity hardware is fast enough to perform many query types *ad-hoc* on unindexed building model and point cloud data. We show how searching in unindexed data can be sped up using simple techniques and trivial data layout adjustments. Our experiments show that ad-hoc queries often can be answered in interactive or near-interactive time without an index, sometimes even outperforming the DBMS. We believe our results provide valuable input and open up possibilities for future research.

1 INTRODUCTION

In recent years, the global amount of geospatial data has grown due to an increased number of acquisition devices and high-quality sensors (Yang et al., 2011). At the same time, geospatial data is used in more and more applications ranging from environmental monitoring, infrastructure planning, catastrophe management to health care.

Domain experts are increasingly faced with the challenge that they receive new or updated large data sets (sometimes on a daily basis) and need immediate access to them. The classic approach is to load the data into a geospatial information system or a database. These systems offer a wide range of functionality and provide users with access to individual items through acceleration structures such as inverted indexes. Other tools reorder the raw data and optimize it for certain use cases (Schütz, 2016).

Nevertheless, in our practical work, we have observed that the larger the data becomes, the harder it is to create an acceleration structure for it, and thus, the longer it takes until the data can be used. Importing big data sets into a database or pre-processing it sometimes takes several hours or even days. We have further observed that, as a consequence, new or updated data sets are often provided as raw files without an acceleration structure on a remote server for download. To immediately use such data in their applications (or simply to evaluate it before importing it into a database), individual users could benefit from a system that allows them to work directly (i.e. *ad-hoc*) on the raw data.

In this position paper, we advocate for the usage of *ad-hoc queries* based on raw, unindexed files, as a means to make working with geospatial data in certain scenarios easier. It is our believe that creating an index is not always the best course of action, a line of thought that has been expressed by other researchers as well, for example with the *NoDB* paradigm by Alagiannis et al. (Alagiannis et al., 2012). Building on this paradigm, we investigate the computational power of modern commodity hardware, which should be capable of handling many common query types in interactive or near-interactive time even when there is no index. To this end, we conduct various experiments with different data sets and measure the performance of practical ad-hoc queries. As the domain of geospatial data is vast, we focus on two commonly-used data types, namely *city models* and *point clouds*. We compare the results with existing systems and analyze the effects of trivial data layout changes on the query performance. As it is clear that unindexed data can never compete with indexed data in large-scale

[a] https://orcid.org/0000-0001-6687-0082
[b] https://orcid.org/0000-0003-2775-5844
[c] https://orcid.org/0000-0002-4664-953X

Bormann, P., Krämer, M. and Würz, H.
Working Efficiently with Large Geodata Files using Ad-hoc Queries.
DOI: 10.5220/0011291200003269
In *Proceedings of the 11th International Conference on Data Science, Technology and Applications (DATA 2022)*, pages 438-445
ISBN: 978-989-758-583-8; ISSN: 2184-285X

multi-user scenarios, we instead address individual users who download a new or updated data set from a remote server to their local hard drive.

The results of our experiments indicate that modern commodity hardware is indeed fast enough to answer various common queries on geospatial data in interactive or near-interactive time. Taking into account the large preprocessing time for indexing, ad-hoc queries are a viable and simple alternative.

2 RELATED WORK

The queries conducted within our experiments all boil down to simple linear searches in the candidate data. For city models, text-based data formats are often used, such as the *CityGML* (Gröger et al., 2012) data format, which is XML-based. Fast text search is one of the oldest problems in computer science, with popular algorithms such as Boyer-Moore (Boyer and Moore, 1977), its improved version by Horspool (Horspool, 1980), or Raita's algorithm (Raita, 1992), which we use as the basis for our experiments due to its simplicity and good performance for large data sets. The core idea of all these algorithms is to exploit knowledge about the sequence that is to be searched to make larger jumps in the data.

Point clouds are often stored in binary formats due to their large size (often billions of points), which prevents the same optimizations that text-based search algorithms employ. Instead of exploiting existing structures within point cloud file formats—as we demonstrate later in this paper—the standard approach for querying point cloud data is to create an index structure. This is often done by *file grouping*: points are grouped together by some primary key, where points close to each other in the domain of the key are put into the same file. This approach is used by the *Potree* system (Schütz, 2016) for spatial queries, as well as for queries by object class as proposed by El-Mahgary et al. (El-Mahgary et al., 2020). One of the most common point cloud file formats is the standardized *LAS* file format (American Society for Photogrammetry and Remote Sensing (ASPRS), 2013), which has its own set of command-line tools for data manipulation, called *LAStools* (Rapidlasso GmbH, 2021; Isenburg et al., 2006). As LAS is a fixed-width format (each point record has the same byte size), it is a good candidate for writing highly efficient search algorithms based on skipping over irrelevant parts of the point records.

Besides working with raw files, relational database management systems (RDBMSs) have long since been the de-facto standard for geospatial applications. The most noteworthy examples for RDBMSs with spatial support are the *PostGIS* project (PostGIS Project, 2021), Oracle Spatial (Oracle, 2021) and Microsoft SQL Server (Microsoft, 2021). *3DCityDB* (Yao et al., 2018) and *GeoRocket* (Krämer, 2020) are open-source applications that manage 3D building models. While 3DCityDB uses a relational database as backend, GeoRocket is based on NoSQL solutions. For point cloud data, there are specialized products such as the *Point Cloud Server* (Cura et al., 2017) or the *pgPointcloud* extension for *PostgreSQL* (Ramsey et al., 2021). For high-dimensional point cloud data, the *HistSFC* (Liu et al., 2020) approach gives good results.

An alternative approach to querying geospatial data is described by Sanjuan-Contreras et al. who present a data structure called *c*B*i*K (Sanjuan-Contreras et al., 2020). This data structure allows various spatial queries (e.g. k-nearest neighbor searches or bounding box queries) to be performed on a compressed data set stored in main memory. Their approach is based on the work by Navarro on compact data structures (Navarro, 2016) and avoids I/O overhead at the expense of more complex computions during search. Although *c*B*i*K is very fast, the data structure has to be created upfront, so it is a spatial index. This is in contrast to our idea to perform ad-hoc queries on disk (and not in memory) without creating an index.

3 WORKING WITH UNINDEXED GEODATA

Of the vast amount of geospatial data that is freely available online, both building models and point clouds are most often distributed as raw, unindexed files through open data portals of individual land survey agencies and municipalities (DoITT, 2021, for example) or through platforms such as *OpenTopography* (OpenTopography Facility, 2022). Depending on the use case, working with this data requires different approaches, where an index is often the first thing that is created. Index creation is highly domain-specific—an index for a point cloud visualization is quite different from one used for analyzing buildings—and often both time- and resource-intensive. Especially in explorative scenarios, where neither the specific region of interest nor the relevant query parameters are known upfront, index creation can be difficult or even impossible (Holanda et al., 2020). Working with frequently changing data also makes index creation difficult, as many of the index structures used for visualization (e.g. kd-trees, octrees) do not lend themselves well to frequent data updates.

In scenarios where users want to work quickly and straightforwardly with the data, *ad-hoc queries* might be the answer: queries issued on the raw data using fast and memory-efficient search algorithms. The reason for using sophisticated index structures is scalability, but there are many single-user scenarios where this does not play as big of a role. We therefore propose to instead focus on the common file formats of building models and point clouds and exploit their inherent structure to speed up ad-hoc queries to get as close to interactive speed as possible. Due to the large volume of geospatial data, this only becomes viable if we can identify and ignore as much data as possible during these ad-hoc queries. For the CityGML format, we achieve this by searching for potential chunks that might match the query parameters with a fast keyword search and only perform XML-parsing on these candidate chunks. For point cloud data, we can use the bounding box information in the LAS headers to quickly discard irrelevant files during spatial queries, exploiting the fact that large point cloud data sets are mostly already split up into many smaller files. In addition, we propose a data-transposed memory layout for the LAS point records which groups all attributes together in memory, making it easy to skip over large portions of the data during querying.

3.1 Fast Search in Raw XML-based Building Model Data

This section describes our approach to perform ad-hoc queries on building data. We focus on *CityGML*, which is a standardized XML-based file format commonly used to exchange 3D building models.

The classic way to work with XML is to parse the file into memory and interpret the elements according to the schema. This can be slow and memory-intensive for data sets that are potentially several gigabytes large. Instead, our approach is purely textual. It is based on a fast string matching algorithm and a simple but effective way to extract individual building models from the CityGML data set.

The first step is to quickly search the data set for a string of interest. For this, we implemented Raita's enhancement to the Boyer-Moore-Horspool fast string searching algorithm (Raita, 1992) (see also Section 2). The main idea is as follows: if you can quickly identify a byte position p_i of what you are looking for in the data set, you can then create two cursors c_1 and c_2 that search the file from p_i backward and forward to find the start and the end of the XML element to extract respectively.

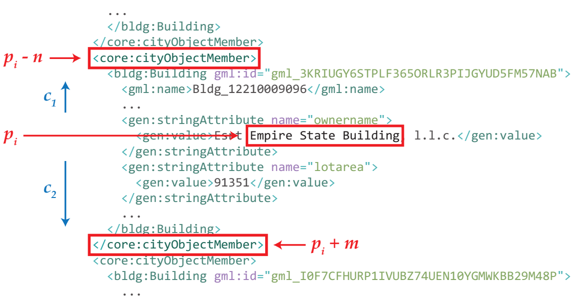

```
    ...
    </bldg:Building>
  </core:cityObjectMember>
  <core:cityObjectMember>
    <bldg:Building gml:id="gml_3KRIUGY6STPLF365ORLR3PIJGYUD5FM57NAB">
      <gml:name>Bldg_12210009096</gml:name>
      ...
      <gen:stringAttribute name="ownername">
        <gen:value>Esrt Empire State Building, l.l.c.</gen:value>
      </gen:stringAttribute>
      <gen:stringAttribute name="lotarea">
        <gen:value>91351</gen:value>
      </gen:stringAttribute>
      ...
    </bldg:Building>
  </core:cityObjectMember>
  <core:cityObjectMember>
    <bldg:Building gml:id="gml_I0F7CFHURP1IVUBZ74UEN10YGMWKBB29M48P">
      ...
```

Figure 1: General approach to search in a text file and extract a CityGML building containing the match at position p_i and ranging from $p_i - n$ to $p_i + m$

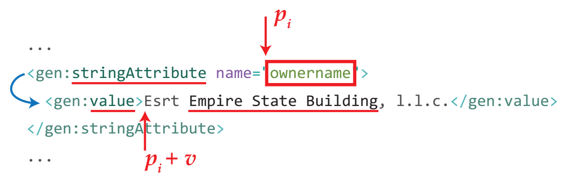

```
...
<gen:stringAttribute name="ownername">
  <gen:value>Esrt Empire State Building, l.l.c.</gen:value>
</gen:stringAttribute>
...
```

Figure 2: Search for the attribute name, then skip forward to compare the value.

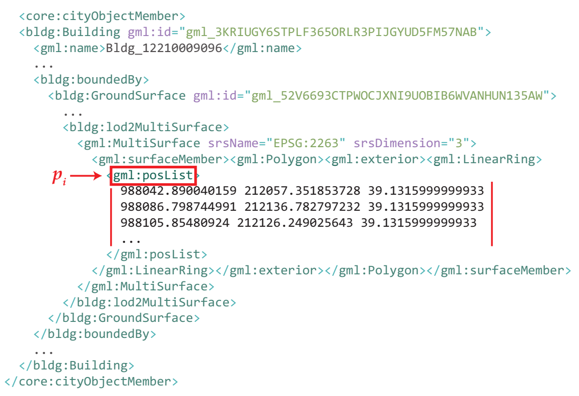
Figure 1 depicts this procedure. p_i points to the matched string "Empire State Building". Cursor c_1 searches the data set backward from $p_i - 1$ to $p_i - n$ until it finds the start of a building. In CityGML, this is the "cityObjectMember" tag. Cursor c_2 searches forward until the end of the building denoted by the closing "cityObjectMember" tag. The string between $p_i - n$ and $p_i + m$ is then extracted.

Based on this, it is also possible to search for buildings with certain attributes (e.g. addresses, owner names, use classes). In CityGML, attributes are described with the elements "stringAttribute", "intAttribute", "doubleAttribute", etc. The elements have a "name" and a child element called "value".

In order to find a building that matches a specific attribute name-value combination, our approach first searches the data set for the string representing the name at position p_i (see Figure 2). It then checks if the match belongs to a generic attribute, moves forward to find its value at a position $p_i + v$, and then compares it with the search value. In case of a positive match, the building is extracted as described above.

This idea even allows for more advanced queries. As soon as the location $p_i + v$ of the attribute value has been identified, it can be parsed into memory. This enables exact string comparisons as well as type conversions (string to number) and, hence, less than or greater than comparisons. This is useful, for example, to find all buildings that have more than n storeys or those within a given zip code range.

With a little more effort, it also enables queries for all buildings whose geometry relates to a cer-



```
  <core:cityObjectMember>
  <bldg:Building gml:id="gml_3KRIUGY6STPLF365ORLR3PIJGYUD5FM57NAB">
    <gml:name>Bldg_12210009096</gml:name>
    ...
    <bldg:boundedBy>
      <bldg:GroundSurface gml:id="gml_52V6693CTPWOCJXNI9UOBIB6WVANHUN135AW">
        ...
        <bldg:lod2MultiSurface>
          <gml:MultiSurface srsName="EPSG:2263" srsDimension="3">
            <gml:surfaceMember><gml:Polygon><gml:exterior><gml:LinearRing>
              <gml:posList>
               988042.890040159 212057.351853728 39.1315999999933
               988086.798744991 212136.782797232 39.1315999999933
               988105.85480924 212126.249025643 39.1315999999933
               ...
              </gml:posList>
            </gml:LinearRing></gml:exterior></gml:Polygon></gml:surfaceMember>
          </gml:MultiSurface>
        </bldg:lod2MultiSurface>
      </bldg:GroundSurface>
    </bldg:boundedBy>
    ...
  </bldg:Building>
</core:cityObjectMember>
```

Figure 3: Example of a “gml:posList” element from a geometry of a CityGML building.

tain bounding box. Figure 3 shows an example of a CityGML building geometry. The x-y-z tuples can be found in the “gml:posList” element. A geometry typically has more than one of such elements. Our approach here is to look for the string “gml:posList”, extract the tuples one by one, convert their items to floating point numbers, and then compare them with the bounding box. This has to be repeated for all “gml:posList” elements found inside the building. As soon as a tuple falls within the bounding box, a match has been found and the building can be extracted. All remaining tuples and “gml:posList” elements of the building can be skipped.

3.2 Fast Search in Binary Point Cloud Data Formats

The main difference between point clouds and textual data is the size of a single data entry. In our building model experiments, a single building entry was on the order of 20 KiB, whereas in common point cloud formats, a single point is rarely larger than a few dozens of bytes. Coupled with the significantly larger number of point records in a typical point cloud (often ranging into the billions), queries on point cloud data are heavily bound by I/O (input/output) throughput.

The LAS file format stores point data in fixed-sized records, which enables skipping over irrelevant attributes during querying. We build on this observation with a custom variant of the LAS file format—dubbed *LAST*—which transposes the memory layout of the point records, storing all attributes together in memory. Other file formats such as *3D Tiles* (Cesium Team, 2018) use a similar transposed memory layout.

A big hurdle for fast ad-hoc queries with point cloud data is compression. The LAZ file format is the compressed variant of LAS and is frequently used due to its great compression ratio (Isenburg, 2013), which comes at the cost of vastly increased computational overhead especially for decompression. We thus investigate whether using a faster compression algorithm such as *LZ4* (LZ4 Team, 2021) can make ad-hoc queries on compressed point cloud data feasible. We test this with a second custom variant of LAS, dubbed *LAZER*, which uses the same memory layout as *LAST* but stored in blocks of a fixed number of points where each block is compressed using LZ4, using one compression context per attribute. Figure 4 shows the memory layout of these LAS variants.

4 EXPERIMENTS

To evaluate our approach, we conducted various experiments on building model data and point clouds. This section summarizes the results.

All experiments were performed on a standard laptop, a 16” MacBook Pro 2019 with a 2.6 GHz 6-Core Intel Core i7 CPU, a 1 TB SSD hard disk, 32 GB of RAM, and macOS 11. We executed each ad-hoc query five times, recorded the time taken each, and then calculated the median. In order to get consistent results, we also used the shell commands `sync` and `purge` on macOS to flush the disk page cache prior to every run. The runtimes of these commands were not included in the measured time.

4.1 Building Model Experiments

For our approach to fast object search and extraction in XML-based building data, we performed several example queries on the enhanced New York CityGML 3D Building Model (version 20v5), which is a combination of the NYC 3D Building Model (DoITT, 2021) and the PLUTO data file (Primary Land Use Tax Lot Output) (Department of City Planning (DCP) of the City of New York, 2021), both provided free of charge by the City of New York. The data set consists of 20 files, has a total size of 20.91 GiB, and contains 1,083,437 buildings with up to 90 semantic attributes per building. It can be downloaded from GitHub (Fraunhofer IGD, 2021b). We used the raw, extracted files. No indexing was applied. All experiments were run two times to test different data sizes: once on a single file (`DA12_3D_Buildings_Merged.gml`, 736.3 MiB, 24,038 buildings), and once on the whole data set.

Our small test application searches the single file sequentially. For the whole data set, we implemented multi-threading to process files in parallel (one thread per file up to the number of available CPU cores).

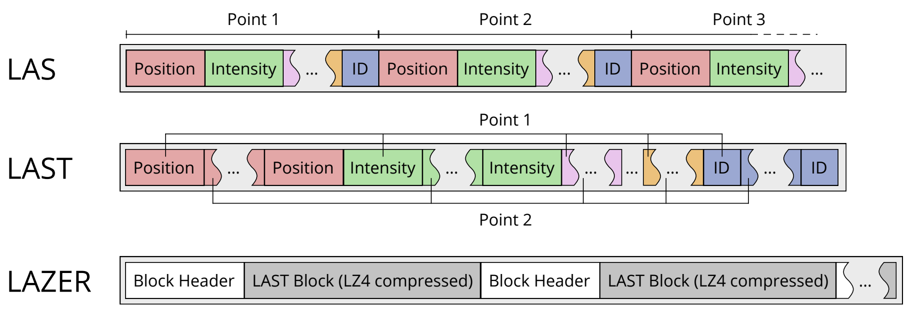


Figure 4: The memory layout of points in the *LAS*, *LAST* and *LAZER* file formats.

Table 1: Results of searching raw unindexed building data.

Query	Files	Runtime (s)	Hits	Misses	Bldgs/s
"ownername" = "Empire State Building"	Single All	0.52 6.74	1 1	23,993 1,082,658	46,227 160,747
"zipcode" = "10019"	Single All	0.96 6.71	1,179 1,196	22,844 1,081,992	25,040 161,466
"numfloors" ≥ 4	Single All	3.59 9.84	18,947 76,533	5,048 1,006,142	6,696 110,105
10018 ≤ "zipcode" ≤ 10020	Single All	0.96 6.75	1,821 1,838	22,202 1,081,350	25,040 160,509
"bldgclass" starts with "F" (Factory)	Single All	0.49 6.65	69 4,167	23,927 1,078,514	49,057 162,923
bounding box ∩ (987700, 211100, 987900, 211300)	Single All	1.96 12.89	13 13	609,408 12,965,283	12,264 84,053
bounding box ∩ (950000, 210000, 1000000, 220000)	Single All	2.71 12.60	9,892 10,983	308,466 12,641,654	8,870 85,987

Table 2: Times it took to import the 3D building model into 3DCityDB and GeoRocket.

	Files	3DCityDB	GeoRocket
Import	Single All	2 m 25 s 2 h 37 m 29 s	1 m 51 s 1 h 07 m 06 s
Create additional indexes	Single All	12 s 2 m 49 s	— —

Table 1 shows the results of our experiments. It contains the example queries and the median runtimes it took to execute them on the single file and on the whole data set. The table also shows the number of objects extracted (Hits), the number of fast text search matches that did not lead to an extraction after attribute value comparison (Misses), as well as the median throughput in buildings per second (Bldgs/s). Bounding boxes are specified in the spatial reference system NAD83 (EPSG code 2263), which uses meters as unit.

To compare the performance of ad-hoc queries with that of well-known existing database-driven solutions, we imported the data into 3DCityDB 4.1.0 (Yao et al., 2018) and GeoRocket 1.3.0 (Krämer, 2020). Table 2 shows the runtimes for importing as well as the time for creating additional indexes (applies to 3DCityDB only; GeoRocket indexes in the

Table 3: Results of executing the example queries using 3DCityDB and GeoRocket. Runtimes of ad-hoc queries have been copied from above for comparison. Fastest times set in bold.

Query	Data	Runtime (s) 3DCityDB	Runtime (s) GeoRocket	Runtime (s) Ad-hoc
"ownername" = "Empire State Building"	Single All	0.92 0.98	**0.02** **0.02**	0.52 6.74
"zipcode" = "10019"	Single All	2.42 2.52	**0.75** **0.79**	0.96 6.71
"numfloors" ≥ 4	Single All	15.31 97.53	11.53 47.00	**3.59** **9.84**
10018 ≤ "zipcode" ≤ 10020	Single All	3.19 15.74	1.16 **1.21**	**0.96** 6.75
"bldgclass" starts with "F" (Factory)	Single All	1.17 4.36	**0.06** **2.50**	0.49 6.65
bounding box ∩ (987700, 211100, 987900, 211300)	Single All	0.88 0.89	**0.04** **0.03**	1.96 12.89
bounding box ∩ (950000, 210000, 1000000, 220000)	Single All	9.28 11.46	6.42 **6.99**	**2.71** 12.60

background during import). Further, Table 3 shows the runtimes of performing the example queries in these two systems in comparison with ad-hoc queries.

In summary, the main observations from these experiments are:

- The ad-hoc approach is fast enough to search the whole 20.01 GiB large data set within only ∼ 6.75 seconds. Note that multi-threading only speeds up I/O performance slightly but most importantly allows the string searching algorithm to work in parallel with reading new data.
- Queries that require string to number conversion execute slower than simple comparisons. Finding buildings with at least four storeys is the slowest of those queries because it yields the most hits. Extracting buildings therefore seems to be rather time-consuming.
- The queries based on bounding boxes lead to many misses because a huge number of x-y-z tuples need to be parsed and compared. The performance of these queries therefore highly depends on the data and how many buildings actually match.

- Comparing to existing solutions, ad-hoc queries perform very well. Of course, the database-driven products are almost always faster, but, given they require hours of importing and indexing in advance, the difference is not that much. In some cases, ad-hoc queries are even faster. This is due to the fact that we can burst copy extracted buildings instead of having to serialize and transfer them over an HTTP or TCP/IP connection.

4.2 Point Cloud Experiments

For point clouds, we conducted three experiments based on common query types:

- Querying points by bounding box (chosen to yield approximately 1%, 25%, and 100% of the points)
- Querying points by bounding box and maximum density (simulating level-of-detail)
- Querying points by a non-spatial attribute (object class)

We used two different publicly available data sets for these experiments:

- *doc*: The District of Columbia 2018 scan (22.2 GiB, 319 LAS files, 854 million points) (Washington, DC, 2018)
- *ca13*: A subset of the PG&E Diablo Canyon Power Plant data set (12.7 GiB, 412 LAZ files, 2.6 billion points) (Watershed Sciences, Inc, 2013)

For these experiments, we converted both data sets into all four file formats LAS, LAZ, LAST, and LAZER, the latter two being the LAS variants described in Section 3.2.

As a reference for our queries, we also loaded all data sets into a PostGIS database with version 3.1.3 using the *pgPointclouds* extension with version 1.2.1. Data upload was done using *PDAL* (PDAL Contributors, 2018) with the default configuration of grouping points into patches of size 400. Afterwards, we manually created a spatial index on the patches. This process took 1h 58m for the *doc* data set, and 7h 34m for the *ca13* data set.

The results of the three point cloud experiments are depicted in Tables 4, 5, and 6 respectively. All tables show the median runtime in seconds for each query, as well as the median point throughput in million points per second (Mpts/s), obtained as the number of points divided by the runtime of the query.

In summary, we can observe the following:

- Compressed file formats, even using the fast LZ4 compression algorithm, are an order of magnitude slower than uncompressed files, and thus are almost exclusively unsuited for ad-hoc queries.

Table 4: Results of point cloud experiment 1.

Experiment 1 - Bounding Box query							
	Source	Runtime (s)			Throughput (Mpts/s)		
		1%	*25%*	*100%*	*1%*	*25%*	*100%*
doc	LAS	0.39	**2.69**	9.87	2200.12	**317.56**	86.54
	LAZ	1.77	18.14	53.47	482.49	47.08	15.97
	LAST	0.37	2.85	**8.36**	2337.68	299.12	**102.10**
	LAZER	0.81	11.45	37.68	1054.32	74.59	22.66
	PostGIS (patches)	**0.09**	5.91	21.50	**9064.78**	144.49	39.71
	PostGIS (points)	18.99	1023.41	3771.85	44.97	0.83	0.23
ca13	LAS	1.02	6.18	44.53	2552.66	421.99	58.57
	LAZ	5.44	39.27	194.20	479.41	66.41	13.43
	LAST	**0.90**	**5.18**	**36.89**	**2884.43**	**503.77**	**70.70**
	LAZER	3.67	34.70	155.00	719.63	75.16	16.83
	PostGIS (patches)	1.07	14.90	132.05	2436.96	175.01	19.75
	PostGIS (points)	192.47	2662.58	13578.03	13.55	0.98	0.19

Table 5: Results of point cloud experiment 2.

Experiment 2 - Bounding box query with max. density							
	Source	Runtime (s)			Throughput (Mpts/s)		
		1%	*25%*	*100%*	*1%*	*25%*	*100%*
doc	LAS	0.46	**3.44**	**11.28**	1854.01	**248.31**	**75.68**
	LAZ	1.78	18.67	57.56	479.78	45.74	14.84
	LAST	**0.42**	3.78	11.81	**2053.87**	225.93	72.32
	LAZER	0.88	12.93	42.97	970.45	66.05	19.87
ca13	LAS	1.29	7.89	53.44	2025.09	330.51	48.80
	LAZ	5.80	42.60	204.00	449.66	61.22	12.78
	LAST	**1.20**	**7.42**	**49.57**	**2179.32**	**351.44**	**52.62**
	LAZER	3.96	34.72	172.70	658.59	75.12	15.10

Table 6: Results of point cloud experiment 3.

Experiment 3 - Query by object class					
	Source	Runtime (s)		Throughput (Mpts/s)	
		building	*non-existing*	*building*	*non-existing*
doc	LAS	8.74	8.15	97.67	104.78
	LAZ	62.74	59.39	13.61	14.38
	LAST	**3.73**	**0.83**	**229.06**	**1034.24**
	LAZER	23.66	21.35	36.09	40.00
	PostGIS (points)	165.68	5.62	5.15	151.83
ca13	LAS	44.31	42.60	58.86	61.23
	LAZ	213.10	204.47	12.24	12.75
	LAST	**2.51**	**2.25**	**1040.18**	**1158.96**
	LAZER	104.76	104.41	24.89	24.98
	PostGIS (points)	114.78	110.41	22.72	23.62

- Transposing data, such as in the LAST format, can speed up query performance significantly, allowing up to a billion points per second to be queried on consumer-grade hardware.
- Spatial queries in almost interactive time are often possible due to the presence of bounding box information in the LAS headers of typical point cloud data sets. Adding level-of-detail to the queries has only a small performance overhead.

- PostGIS becomes slow once single-point granularity (using *PC_Intersection*) is desired. In almost all cases, working with unindexed files is faster, and this is before including the substantial time for importing the point cloud into PostGIS.

5 CONCLUSION

In this paper, we demonstrated how fast search algorithms enable working with raw, unindexed geodata. We implemented several experimental applications that perform queries that are answered *ad-hoc* based on raw geodata, without using any index structure. We released these applications under an open-source license on GitHub (Fraunhofer IGD, 2021a). The experiments show that for both building and point cloud data, ad-hoc queries perform very well in general. They achieve times reasonable for practical use cases and comparable to those of the indexed-based solutions. In particular, they allowed us to directly work with the data without having to wait several hours for it to be imported into a database. This even works for data that does not fit into main memory.

For point cloud data, we also discovered potential for optimization by changing the data layout within the LAS file format so a search algorithm has to fetch less data from disk. Compression is a limiting factor that makes querying raw point cloud data slow. We were unable to achieve interactive query response times for all tested compressed formats due to the large computational overhead.

We believe that our results form the basis for developing further applications based on ad-hoc queries, harnessing the power of modern computers. Especially in the scientific community, we often encounter single-user scenarios where data is analyzed in a way that is similar to what ad-hoc queries offer. Integrating fast search algorithms in the fashion of what is presented in this paper into existing data analysis libraries could speed up these scenarios, enabling users to work more efficiently with geospatial data.

REFERENCES

Alagiannis, I., Borovica, R., Branco, M., Idreos, S., and Ailamaki, A. (2012). NoDB: Efficient query execution on raw data files. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 241–252.

American Society for Photogrammetry and Remote Sensing (ASPRS) (2013). LAS specification, version 1.4 - R13. https://www.asprs.org/wp-content/uploads/2010/12/LAS_1_4_r13.pdf. Accessed: 2022-04-06.

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772.

Cesium Team (2018). CesiumGS/3d-tiles: Specification for streaming massive heterogeneous 3D geospatial datasets. https://github.com/AnalyticalGraphicsInc/3d-tiles. Accessed: 2022-04-06.

Cura, R., Perret, J., and Paparoditis, N. (2017). A scalable and multi-purpose point cloud server (pcs) for easier and faster point cloud data management and processing. *ISPRS Journal of Photogrammetry and Remote Sensing*, 127:39–56.

Department of City Planning (DCP) of the City of New York (2021). Primary land use tax lot output (pluto). https://www1.nyc.gov/site/planning/data-maps/open-data/dwn-pluto-mappluto.page. Accessed: 2021-07-10.

DoITT (2021). Department of Information Technology & Telecommunications (DoITT) of the City of New York – NYC 3-D building model. https://www1.nyc.gov/site/doitt/initiatives/3d-building.page. Accessed: 2021-07-01.

El-Mahgary, S., Virtanen, J. P., and Hyyppä, H. (2020). A simple semantic-based data storage layout for querying point clouds. *ISPRS International Journal of Geo-Information*, 9(2).

Fraunhofer IGD (2021a). Ad-hoc queries on 3d building models and point clouds - benchmark implementations. https://github.com/igd-geo/adhoc-queries-building-models, https://github.com/igd-geo/adhoc-queries-pointclouds. Accessed: 2022-04-06.

Fraunhofer IGD (2021b). Enhanced NYC 3-D building model. version 20v5. https://github.com/georocket/new-york-city-model-enhanced/. Accessed: 2022-04-06.

Gröger, G., Kolbe, T. H., Nagel, C., and Häfele, K.-H. (2012). *OGC city geography markup language (CityGML) encoding standard*. Open Geospatial Consortium, 2.0.0 edition.

Holanda, P., Raasveldt, M., Manegold, S., and Mühleisen, H. (2020). Progressive indexes: Indexing for interactive data analysis. *Proceedings of the VLDB Endowment*, 12(13):2366–2378.

Horspool, R. N. (1980). Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506.

Isenburg, M. (2013). LASzip: Lossless compression of lidar data. *Photogrammetric Engineering and Remote Sensing*, 79(2):209–217.

Isenburg, M., Liu, Y., Shewchuk, J., Snoeyink, J., and Thirion, T. (2006). Generating raster dem from mass points via TIN streaming. In Raubal, M., Miller, H. J., Frank, A. U., and Goodchild, M. F., editors, *International conference on geographic information science*, pages 186–198. Springer.

Krämer, M. (2020). Georocket: A scalable and cloud-based data store for big geospatial files. *SoftwareX*, 11.

Liu, H., Van Oosterom, P., Meijers, M., and Verbree, E. (2020). An optimized SFC approach for nd window querying on point clouds. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 6(4/W1):119–128.

LZ4 Team (2021). Extremely fast compression algorithm. https://github.com/lz4/lz4. Accessed: 2021-07-16.

Microsoft (2021). SQL Server. https://www.microsoft.com/en-us/sql-server. Accessed: 2021-07-19.

Navarro, G. (2016). *Compact Data Structures: A Practical Approach*. Cambridge University Press, USA, 1st edition.

OpenTopography Facility (2022). OpenTopography. https://opentopography.org/. Accessed: 2022-03-21.

Oracle (2021). Oracle's spatial database. https://www.oracle.com/database/spatial/. Accessed: 2021-07-19.

PDAL Contributors (2018). Pdal point data abstraction library. https://doi.org/10.5281/zenodo.2556738.

PostGIS Project (2021). PostGIS - Spatial and Geographic objects for PostgreSQL. https://postgis.net/. Accessed: 2021-06-08.

Raita, T. (1992). Tuning the boyer-moore-horspool string searching algorithm. *Software: Practice and Experience*, 22(10):879–884.

Ramsey, P., Blottiere, P., Brédif, M., and Lemoine, E. (2021). pgPointcloud - A PostgreSQL extension for storing point cloud (LIDAR) data. https://pgpointcloud.github.io/pointcloud/index.html. Accessed: 2021-07-19.

Rapidlasso GmbH (2021). LAStools: award-winning software for rapid LiDAR processing. http://lastools.org/. Accessed: 2021-06-08.

Sanjuan-Contreras, C. E., Retamal, G. G., Martínez-Prieto, M. A., and Seco, D. (2020). cBiK: A space-efficient data structure for spatial keyword queries. *IEEE Access*, 8:98827–98846.

Schütz, M. (2016). Potree: Rendering Large Point Clouds in Web Browsers. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria.

Washington, DC (2018). District of Columbia - Classified Point Cloud LiDAR. https://registry.opendata.aws/dc-lidar/. Accessed: 2020-08-27.

Watershed Sciences, Inc (2013). PG&E Diablo Canyon Power Plant (DCPP): San Simeon, CA Central Coast. http://opentopo.sdsc.edu/lidarDataset?opentopoID=OTLAS.032013.26910.2. Accessed: 2020-08-27.

Yang, C., Goodchild, M., Huang, Q., Nebert, D., Raskin, R., Xu, Y., Bambacus, M., and Fay, D. (2011). Spatial cloud computing: how can the geospatial sciences use and help shape cloud computing? *International Journal of Digital Earth*, 4(4):305–329.

Yao, Z., Nagel, C., Kunde, F., Hudra, G., Willkomm, P., Donaubauer, A., Adolphi, T., and Kolbe, T. H. (2018). 3dcitydb - a 3d geodatabase solution for the management, analysis, and visualization of semantic 3d city models based on citygml. *Open Geospatial Data, Software and Standards*, 3(1):5.