SN Computer Science (2024) 5:647
https://doi.org/10.1007/s42979-024-02986-z

ORIGINAL RESEARCH

Executing Ad-Hoc Queries on Large Geospatial Data Sets Without Acceleration Structures

Pascal Bormann[1,2] · Michel Krämer[1,2] · Hendrik M. Würz[1,2] · Patrick Göhringer[1,2]

Received: 29 November 2022 / Accepted: 22 May 2024

Abstract

In this case study, we investigate if it is possible to harness the capabilities of modern commodity hardware to perform ad-hoc queries on large raw geospatial data sets. Normally, this requires building an index structure, which is a time-consuming process. We aim to provide means to individual users who receive a new or updated geospatial data set and want to directly start working with it without having to build such an index structure first. To this end, we conduct various experiments on two distinct types of data: 3D building models and point clouds. For the former, we demonstrate that well-known algorithms such as fast string search allow a wide range of queries to be answered in at most a few seconds on data sets with over a million buildings. The usage of progressive indexing additionally improves query run time by more than a factor of two. Regarding point clouds, we achieve similar run times using the popular LAS file format and a query throughput of up to a billion points per second when using a columnar memory layout. The run time of ad-hoc queries is often on par with that of database-driven solutions, sometimes even outperforming them. Considering that ad-hoc queries require no preprocessing, our results show that they are a viable alternative to acceleration structures when working with geospatial data.

Keywords Information retrieval · Searching · Geospatial data · Building models · Point clouds

Introduction

Geospatial information is important for a wide range of applications. Professional users from mapping agencies, municipalities, or companies require timely access to up-to-date data for use cases such as environmental monitoring, infrastructure planning, catastrophe management, or health care. Simultaneously, an increasing number of data acquisition sensors and devices produce large amounts of data in a short time [1]. For example, terrestrial mobile mapping systems mounted on cars [2, 3] are deployed in urban areas to continuously monitor the environment or to support planning processes. They collect 3D point clouds and panorama images on a daily basis. Point clouds produced by airborne laser scanners are in turn used to generate 3D building models [4], which are further enriched with semantic information. Earth observation initiatives such as the Sentinel missions [5] from the Copernicus programme [6] constantly produce imagery that is immediately made available to the public and is free of charge.

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 way to work with such data is to load it into a geospatial information system or database. These systems offer a wide range of functionality. In order to provide users with access to individual items from large data sets, they create acceleration structures such as inverted indexes. Other tools reorder the raw data and optimize it for certain use cases. Potree, for example, processes point clouds and creates an acceleration structure for fast web visualization [7].

However, in our practical work, we have observed that the larger the data becomes, the harder it is to create an

✉ Pascal Bormann
pascal.bormann@igd.fraunhofer.de

✉ Michel Krämer
michel.kraemer@igd.fraunhofer.de

Hendrik M. Würz
hendrik.martin.wuerz@igd.fraunhofer.de

Patrick Göhringer
patrick.goehringer@stud.tu-darmstadt.de

[1] Fraunhofer Institute for Computer Graphics Research IGD, 64283 Darmstadt, Germany

[2] Technical University of Darmstadt, 64289 Darmstadt, Germany

Published online: 13 June 2024

acceleration structure for it, and thus, the longer it takes until the data can be used. Importing big data sets into a geospatial information system or a database or processing the data with a tool like Potree sometimes takes several hours or even days, which prevents (or at least hinders) timely access to up-to-date information. We have also 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 the local disk. To immediately use such data in their applications (or simply to evaluate it before importing it into a database), individual users therefore would benefit from a system that allows them to work directly (i.e. *ad-hoc*) on the raw data.

The necessity of creating an acceleration structure—a process called *indexing*—is a fundamental insight of the computer science discipline rooted in algorithmic complexity analysis. If either the data size scales up or the maximum allowed query time scales down, the jump from linear to logarithmic lookup time is crucial. Modern computer architectures are, however, hardly as simple as the models we use for algorithmic complexity analysis. As such, there is a fundamental difference between a system that is algorithmically efficient and one that is actually usable—i.e. one that meets the user expectations in terms of run time. Something that is efficient might still not be usable, and something that is usable might not be efficient. At the same time, something that was considered large data a few years ago might be perfectly manageable by commodity hardware today.

Our main aim in this case study is, therefore, to empirically investigate if it is possible to perform *ad-hoc queries on large geospatial data* using *raw files without indexing*. To this end, we conduct various experiments with different data sets and measure the performance of practical ad-hoc queries. We compare the results with existing systems and critically discuss benefits and drawbacks of working without an index. Since the main challenge of querying unindexed geospatial data are the large data volume and high number of individual data points, we explain strategies for increasing query throughput using the power of modern hardware.

Specifically, the goals of our study are as follows:

- We aim to provide means to individual users who get a *new or updated data set and want to quickly start working with it*. For example, they should be able to timely access single data items, to analyse the data, or to quickly visualize small parts of it without having to load the whole data set into a geospatial information system or a database first.
- Since we target individual users, we want to investigate the *possibilities of modern commodity hardware* instead of large compute clusters. The results of the experiments presented in this paper were all collected on a standard laptop.
- We want to evaluate if *simple but well-known algorithms* such as a fast string search can be used to replace complex and time-consuming indexing. It should still be possible to achieve *reasonable access times* for *common geospatial queries*.

Note that our non-goals are as follows:

- We do not want to create a universal approach that applies to all kinds of data sets with different formats or schemas. In this paper, we only investigate 3D city models and point clouds, which are two of the most common kinds of geospatial data. They are also distinctively different in structure and therefore suitable to represent a range of other formats (including textual as well as binary ones).
- The programs we created for our experiments do not support scalability in terms of numbers of queries per second or number of users working in parallel. We address individual users who download a new or updated data set from a remote server to their local hard drive.
- We focus on common geospatial queries (e.g. based on attributes or bounding boxes) and on extracting single data items or small parts of a large data set. We do not cover more complex data analyses or visualizations.

The main contribution of this case study are the results of the experiments as well as strategies for improving the query throughput on raw geospatial data files, both for textual and binary data. They provide useful and interesting insights for researchers and software developers of geospatial information management and analysis systems who need to work with large and up-to-date data sets. The results demonstrate that ad-hoc queries are a viable alternative to indexing for geospatial data management.

The paper starts with an overview of related work ("Related Work" section) and an introduction into the methodology of our study ("Methodology" section). It is then structured in two parts illustrating ad-hoc queries on building models ("Querying Building Data" section) and point clouds ("Querying Point Cloud Data" section). Based on the experimental results, we critically discuss the implications on applications working with geodata ("Discussion" section). The paper finishes with conclusions and an outlook on future research possibilities ("Conclusion" section).

Differences to the Conference Paper

This paper is a significant extension of our conference paper presented at DATA 2022 [8]. The previous work was a position paper where we presented our idea and first results of experiments with building and point cloud data. In the meantime, we were able to explore new research aspects. In summary, the extended paper covers the following additional topics:

- We have extended our position paper to a *comprehensive case study* that covers *many more experiments* than before. The results now also contain *more details* and comprehensive in-depth discussions (see "Methodology"–"Discussion" sections).
- The "Related Work" section *has been completely revised*. In particular, we have included a list of existing works on dynamic indexing approaches ("Dynamic Indexing Approaches" section).
- We have revised our comparison with existing tools, in particular the solutions for managing building data ("Comparison with Existing Solutions" section). We *upgraded the tool GeoRocket* to the latest alpha version, which promises a higher performance and *performed our experiments again* to get updated measurements.
- We extended our approach for the search in building data and added a *completely new section on progressive bounding box index generation* ("Progressive Bounding Box Index Generation" section). This section shows how our approach can be combined with existing ones to open up new research possibilities.

Related Work

Searching for an element in a large set of candidates is a well-studied area. The following sections summarize known search algorithms, the use cases in which they are applied, and the additional knowledge they exploit. We focus on searching in text data ("Searching Text Data" section) and binary data ("Searching Binary Data" section). After that, we given an overview of specialized solutions to search for geospatial information ("Searching Geospatial Data" section). Finally, we summarize approaches to information retrieval that either create an index on demand (or incrementally) or that try to avoid creating an index at all ("Dynamic Indexing Approaches" section).

Searching Text Data

String matching is a search for a sequence of characters (*pattern*) in a larger sequence (*text*). Knowledge about the pattern can be exploited to skip as many bytes as possible in the text and hence to improve performance.

One of the best known algorithms in this area is Boyer–Moore [9]. In contrast to a naïve linear search where the first character of the pattern is compared to every character of the text, Boyer–Moore starts at the end of the pattern. If the last character does not match with the text, the previous characters do not need to be checked and the algorithm can skip some parts of the text. The longer the pattern, the more likely it is that a larger number of characters can be skipped.

Horspool simplified Boyer–Moore's algorithm by removing comparisons related to repetitions in the pattern [10]. These comparisons provided little improvement for natural language text and did not justify the additional effort. Raita exploited another effect with regard to natural language text: the closer the characters, the more they depend on each other [11]. Similar to Boyer–Moore–Horspool, Raita's improvement first looks at the last character of the pattern. After that, however, the first character is evaluated rather than the second to last. In this way, the dependencies between the characters are as small as possible and the probability of detecting a difference early is higher.

An alternative to Boyer–Moore is the algorithm of Knuth–Morris–Pratt [12]. Unlike Boyer–Moore, it starts with the first character of the pattern and not the last one but still tries to skip as many characters as possible. Comparing the two algorithms, Boyer–Moore is usually faster when searching in natural language text. However, when the underlying alphabet is very small, Knuth–Morris–Pratt may be better [13].

The algorithms mentioned above only search for a single pattern. Aho–Corasick introduced an algorithm that can look for multiple patterns at the same time [14]. The algorithm uses a finite state machine to check each pattern simultaneously as it traverses the text. This works best when the patterns have a common prefix or suffix. Commentz–Walter combined the idea of a finite state machine with Boyer–Moore's approach to achieve better run time [15].

Searching Binary Data

Compared to the search in text data, searching binary data highly depends on the format. Sometimes, individual byte positions of items in the data can be directly calculated and sometimes not. In some cases, a fixed bit pattern can be used as a search pattern, but sometimes, other information from the file (i.e. the file header) must be evaluated first.

An intermediate stage between plain text files and arbitrary binary data is compressed text. The goal is still to find a pattern in a sequence of characters, but the sequence is now compressed. Some algorithms have been developed that exploit how the individual compression schemes work to make use of additional information during the search. An example is the approach by Navarro [16]. It searches for regular expressions in a Ziv–Lempel compressed text, where repetitions in the original text are replaced with a pointer to the first occurrence. The algorithm takes advantage of this and only searches individual blocks in the compression. This doubles the search speed compared to decompressing and searching afterwards. However, the algorithm only works with Ziv–Lempel compressed text. Ganty et al. presented a similar algorithm that can search on grammar compressed text in general [17]. It returns the number of matches to a regular expression in linear time. Another approach was introduced by Ferragina et al [18]. They combine the compression of text with the creation of an index. This enables fast searches even though the original text was compressed.

Regarding searching in binary data in general, Gustafsson et al. present an approach to find patterns in network packets [19]. Gustafsson et al. read the length of a packet from the header information. Afterwards, they are able to filter individual segments using a decision tree.

Searching Geospatial Data

The ability to perform queries on the data is one of the central features that enable value generation from geospatial data. While data exchange often happens through specialized file formats, geospatial applications have long been working with relational database management systems (RDBMSs) as their preferred storage backend. The popular *PostGIS* project [20] adds support for spatial data to the *PostgreSQL* RDBMS and has become a de facto industry standard over the last two decades. Other RDBMSs with spatial support are Oracle Spatial [21] and Microsoft SQL Server [22]. Specialized data management solutions such as *GeoServer* [23] and *Deegree* [24] utilize these RDBMSs to store raster data and geometries.

3DCityDB [25] and *GeoRocket* [26] are open-source applications that manage 3D building models. Both are able to use various relational databases as backend, whereas GeoRocket also supports NoSQL solutions such as MongoDB. 3DCityDB and GeoRocket provide a high-level API with which users can query large 3D city models and extract individual buildings or small areas using filters based on semantic attributes or spatial areas (i.e. bounding boxes).

An area where file-based approaches are the default is point cloud data. A point cloud is a usually unordered collection of n-dimensional points, with the dimensions corresponding to predefined attributes, such as the position in 3D space, a color value or grayscale intensity, or an object classification, to name a few. Due to the limited amount of information inherent in a single point, point cloud data typically consists of millions to billions of individual points. A popular software for working with point cloud data is *LAStools* [27, 28], a set of command line tools for tasks such as splitting, merging, transforming and rasterizing point clouds, all based on the standardized *LAS* file format [29]. Many point cloud visualization applications also require acceleration structures in specialized file formats and layouts. Examples are *Potree* [30] or *Cesium* [31], for which several file-based preprocessing tools exist [32–34]. The standard approach for querying these point clouds is 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. Executing a query on these point clouds is then equivalent to looking up the matching files based on the primary key and the name of the files. This approach is used by the *Potree* system [7] for spatial queries, as well as for queries by object class as proposed by El-Mahgary et al. [35]. The downside of these file-based approaches is that they require a complete reordering of the data, resulting in a copy of the original data. Besides file-based approaches, there has been recent work studying the usage of RDBMSs for point clouds. Van Oosterom et al. conducted a study analysing the performance of common RDBMSs when storing and retrieving point cloud data [36]. There are also specific RDBMSs available for point cloud data, such as the *Point Cloud Server* [37] or the *pgPointcloud* extension for *PostgreSQL* [38]. Still, file-based systems are used most frequently since working with raw files is simple and enables more specialized data structures. These in turn can yield higher point throughput than RDBMSs and can support specialized features, such as the level-of-detail support that enables fast web-rendering of point clouds in the Potree system.

Dynamic Indexing Approaches

Optimizing the performance of queries in databases is a research area with a long history but still very active today. Various indexing methods and query optimization techniques have been investigated. While the traditional solution is to create an index in advance, there are more dynamic approaches that work on demand, incrementally, or that try to avoid creating an index at all.

An incremental indexing technique described by Idreos et al. is called *Database Cracking* [39]. Attributes that are frequently requested are clustered and partially sorted. The first query on a specific attribute initializes the index and therefore causes some overhead. With each further query, however, refined fragments may be created in the

index, which incrementally increases the query processing speed. Database Cracking was specifically designed for an environment where you have no prior knowledge of what attributes might be of interest and no time to wait for updates to be indexed.

Progressive Indexing by Holanda et al. is an adaptive indexing technique that improves Database Cracking by providing a guarantee for convergence, more robustness, more controllable indexing overhead, and lower overhead for the first query [40]. Each query triggers an incremental indexing step. The indexing overhead is calculated using a cost model and a parameter that indicates how many operations may be performed within a single indexing step.

Hohenstein presents an extended approach to Progressive Indexing, which is based on approximate query processing [41]. Hohenstein applies uniform sampling in combination with rare sub-population detection. This way, the cost of scanning the data when the index is not yet fully built can be reduced. The system renders more inaccurate results at the beginning, in exchange for more speed, but returns accurate results later when the index is fully built.

To generalize the approach of Database Cracking, Idreos et al. further address the problem that loading a large data set into a database takes a lot of time and that setting up and optimizing databases for data and workload is a complex task [42]. Their idea is that the user should only have to provide a link to the data and can start querying directly while small parts of the data are loaded into the database only *when* and *if* needed. To this end, Idreos et al. extend the DBMS MonetDB with an adaptive loading strategy. In our case study, we also aim to create a solution that allows data to be queried directly, but in contrast, our approach does not need a database.

Alagiannis et al. present a new paradigm called *NoDB*, which abstractly describes a data query solution that avoids the need of loading data but provides all features of a modern DBMS [43]. The authors claim that their approach can reduce the time from data arrival to analysis. They also see a problem in storing the data in internal, proprietary database formats, which is avoided by working on raw files. Another approach, which builds upon the NoDB paradigm, is presented by Karpathiotakis et al. who add more flexibility by describing an adaptive query engine that supports various data formats [44]. Their approach uses Just in Time compilation (JIT) for querying raw files in a more optimal way and so-called *column shreds* to read as few bytes as necessary. In contrast to the works of Alagiannis et al. and Karpathiotakis et al., the approach we present in this paper is focused on geospatial data, which requires specific operations (such as a bounding box search) but also allows us to optimize query performance to the use case.

Methodology

In order to evaluate if ad-hoc queries can be performed in reasonable time on large raw geospatial data sets, we conducted various practical experiments. We measured the time for each ad-hoc query and compared the results with existing solutions. We also identified benefits and drawbacks of ad-hoc queries compared to acceleration structures. The following sections give an overview of the different data types and data sets used in our experiments ("Data Types" section), the geospatial queries covered ("Common Geospatial Queries" section), implementation details ("Implementation" section), as well as the setup for our experiments ("Experiment Setup" section).

Data Types

As mentioned in the "Introduction" section, we focus on two types of geospatial data in our case study: 3D building models and point clouds. Figures 1 and 2 show screenshots of two applications that visualize both data types in a web browser.

We chose these data types because they are well-known but are quite different in their structure and usage. In addition, there are many large and open data sets available for these two formats. Building models are often stored in *CityGML* [47], which is a textual, XML-based format. Point

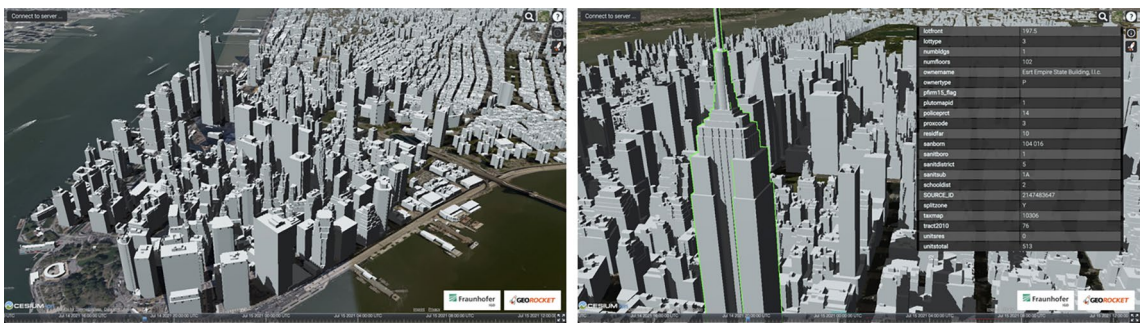

Fig. 1 Screenshots of a web viewer showing the enhanced New York City 3D Building Model [45] with semantic attributes

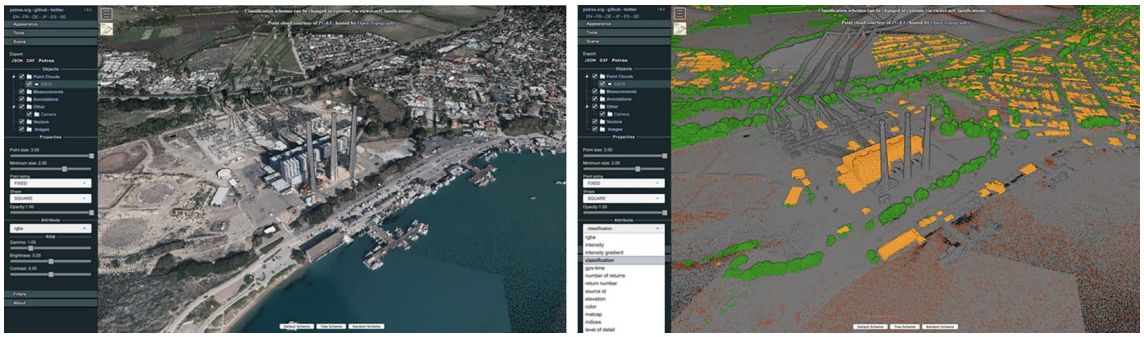

Fig. 2 Screenshots of the ca13 (PG &E Diablo Canyon Power Plant) point cloud data set [46] visualized in Potree [30] with photo-realistic RGB coloring (left) and colors based on semantic classification (right)

clouds are typically provided in the binary formats *LAS* [29] and its compressed variant *LAZ*.

For the first set of experiments (see "Querying Building Data" section), we used the enhanced New York CityGML 3D Building Model (version 20v5), which is a combination of the NYC 3D Building Model [48] and the PLUTO data file (Primary Land Use Tax Lot Output) [49], 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 [45]. 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 another time on the whole data set.

For the point clouds, we used three different data sets varying in size and provided file format. All files are publicly available and free of charge:

- *navvis3*: The *navvis_m6_3rdFloor* data set (139 MiB LAZ file with 56.2 million points) [50]
- *doc*: The District of Columbia 2018 scan (22.2 GiB, 319 LAS files, 854 million points) [51]
- *ca13*: A subset of the PG &E Diablo Canyon Power Plant data set (12.7 GiB, 412 LAZ files, 2.6 billion points) [46]

Common Geospatial Queries

When users work with a geospatial data set, they usually have a certain goal in mind: they want to explore the data, visualize it (or a subset of it), or perform some kind of analysis. For this, users typically need a way to filter the data set and to extract items based on certain criteria—i.e. to perform queries on it—in order to focus on the information that is actually necessary for the task at hand. The most common geospatial queries are as follows:

- The users have to be able to extract data items based on a given set of *user-defined attributes* (e.g. address, number of floors, classification, heat demand).
- They also need to be able to retrieve all data items that lie within a given *bounding box* (i.e. an axis-aligned rectangular spatial area), for example for a visualization or an in-depth analysis of smaller areas.

For the experiments we present in the "Querying Building Data" and "Querying Point Cloud Data" sections, we formulated corresponding ad-hoc queries on the two types of data. For the point clouds, we also defined a query based on a *dynamic property* that can only be calculated by analysing the data (i.e. the point density).

Implementation

The specific data formats of building models (CityGML) and point clouds (LAS/LAZ) require different techniques for parsing and interpreting. According to which technologies are typically used in practice, we implemented the experiments of our case study in two different programming languages: Kotlin for the building models and Rust for the point clouds.

Kotlin is a language based on the Java Virtual Machine. The Java API is often used for building models because of its sophisticated support for reading XML files. For point clouds with their binary files containing millions or billions of data items, it is often more reasonable to use a low-level programming language like Rust, which provides the developer with means to organize the memory layout and to control when resources are allocated and released.

More details on the individual implementations are given in the "Implementation" section on querying building data and the "Implementation" section on point clouds.

Experiment Setup

As described in the goals of our case study, we wanted to test if ad-hoc queries can be performed on commodity hardware of individual users instead of on large compute clusters. All experiments were therefore conducted 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 12.

We executed each ad-hoc query five times, recorded the time taken each, and then calculated the median as well as the standard deviation. 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 run times of these commands were not included in the measured time.

Since the program for the building experiments was written in Kotlin, we also performed two initial runs there (prior to the five main runs) to allow the just-in-time compiler (JIT) to warm-up. These runs were also not included in the median and standard deviation.

Querying Building Data

This section describes how we performed ad-hoc queries on building data stored in the XML-based CityGML file format. First, we introduce our basic approach ("Executing Ad-Hoc Queries on Building Data" section) and give some implementation details ("Implementation" section). We then present the results of the conducted experiments ("Building Experiments" section) and compare them with existing solutions ("Comparison with Existing Solutions" section). Finally, we present an approach to speed up the search for bounding boxes using progressive indexing ("Progressive Bounding Box Index Generation" section).

CityGML defines various modules covering a large number of object types that can appear in an urban environment (e.g. buildings, bridges, street furniture, railways). We focus on the core and the building module which address generic geometries and attributes as well as building models.

Executing Ad-Hoc Queries on Building Data

The classic approach to work with XML is to parse the file into memory and interpret the elements according to the schema. However, building a document object model (DOM) for a data set that is potentially several gigabytes large is impractical, and parsing the entire XML structure just to match a few attributes and extract a subset of objects is too much of an overhead. 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 following subsections describe operations that build upon each other and pose challenges with increasing difficulty to the query system. The most basic operation on textual data is free text search. Our assumption is that if you are able to find an arbitrary string in the data set ("Extracting Objects Based on Arbitrary String Matches" section), you should also be able to identify key-value pairs ("Searching for Key-Value Pairs" section) and to perform more advanced value comparisons ("Advanced Key-Value Queries" section). In turn, being able to find key-value pairs is a prerequisite to retrieving objects by bounding box ("Bounding Box Queries" section). Finally, multiple queries can be combined using logical operators ("Combining Multiple Queries" section).

Extracting Objects Based on Arbitrary String Matches

The first step in our approach is to search the data set for an arbitrary string. For this, we implemented Raita's enhancement to the Boyer–Moore–Horspool fast string searching algorithm [11] (see also "Searching Text Data" section).

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.

Figure 3 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.

Note that this idea is simple and very fast (see results in the "Building Experiments" section) but its simplicity comes with a caveat: It does not check if the search string actually is an attribute (or an attribute value). You can literally search for anything. Even the XML tag name "Building" would work. It would simply match all buildings in the whole data set. Additional checks are needed to make the query results more precise. Our approach therefore also supports searching for key-value pairs.

Searching for Key-Value Pairs

Generic attributes in CityGML are described with the elements "stringAttribute", "intAttribute", "doubleAttribute", etc. The elements have a "name" and a child element called "value". Based on the idea described

Fig. 3 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 $p_i + m$ (Image source: [8])

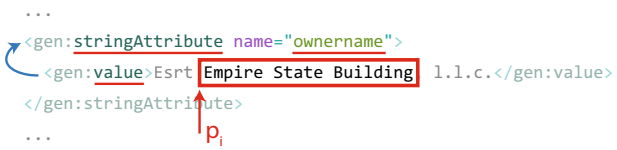

Fig. 4 Search by value, then skip backward to extract and compare key (Image source: [8])

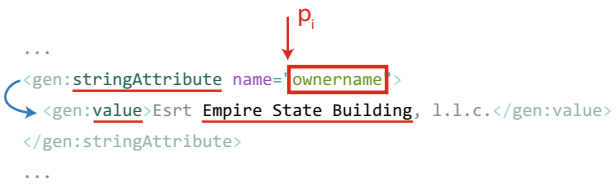

Fig. 5 Search by key, then skip forward to extract and compare value (Image source: [8])

above, there are two approaches to identify key-value pairs (i.e. matching generic CityGML attributes):

- you can either search for the string representing the *value*, check if the match actually is a CityGML value, then move backward to find the corresponding parent element, extract its name, and finally compare it with the *key* you are looking for (see Fig. 4), or
- you can search for the string representing the *key*, check if the match belongs to a generic attribute, move forward to find its value, and then compare it with the *value* you are looking for (see Fig. 5).

In case of a positive match, you then have to extract the building as described above.

Both approaches have benefits and drawbacks. For example, a very specific value (e.g. "Empire State Building") might only occur once or a few times in a large data set, whereas the key (e.g. "ownername") most probably appears as many times as there are buildings in the data set. In such a case, it can be faster to search for the value first to avoid unnecessary comparisons and to directly jump to the buildings to extract instead. Nevertheless, as we show in the "Building Experiments" section, these comparisons actually do not have a large performance impact, and searching for the key first is interestingly almost as fast as searching by value. It also provides the additional benefit that you can compare the value not only literally but also parse numbers, apply greater than or less than comparisons, etc. This flexibility is necessary for more advanced queries.

Advanced Key-Value Queries

As soon as you have found a CityGML generic attribute by key and extracted its value, you can perform advanced comparisons. The following list gives a few example comparisons that are possible with our approach:

- Convert the value to a number and check if it is less than or greater than the value you are looking for (e.g. to get all buildings that have more than *n* storeys)
- Compare the converted number to a numerical range (e.g. to find all buildings within a given zip code range)
- Compare the value to a set (e.g. to get all buildings whose "usage" is either "commercial" or "domestic")

Based on the result, you then have to extract the building as described above. We have implemented these example comparisons and evaluated their performance (see "Building Experiments" section).

Bounding Box Queries

Searching for buildings whose geometry is covered by a given bounding box is more complex than looking for attributes. In the case of CityGML, you have to iterate through the entire geometry of a building and compare its coordinates with the bounding box.

Figure 6 shows an example of a building geometry. The coordinates (more precisely, the x–y–z tuples) can be found

Fig. 6 Example of a "gml:posList" element from a geometry of a CityGML building (Image source: [8])

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" first, then extract the tuples one by one, convert their items to floating point numbers, and 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 is not within the bounding box, the corresponding building can be skipped. However, if all tuples are in the bounding box, the building is extracted as described above.

Note that there are other geospatial relations besides *covers*, such as *intersects*, *touches* or *overlaps* [52]. We have already discussed *intersects* in our previous work [8]. We did not implement other relations because they require more computational effort and it is not possible to skip individual tuples.

Combining Multiple Queries

Multiple queries can be combined using logical operators. We implemented the following two operators:

OR

There are two ways to find buildings that match at least one of a given set of criteria. As described in the "Advanced Key-Value Queries" section, for criteria that refer to the same key, you can compare the found values to a range or a set. For all other criteria, you have to perform the search multiple times (once for each criterion) and then combine the results by creating their union.

AND

Finding buildings that match all given criteria at the same time works differently. In this case, you initially have to find a building that matches the first criterion. Then, you have to repeat the search for each other criterion but only within the byte range from $p_i - n$ to $p_i + m$ representing the beginning and the end of the building respectively (see "Extracting Objects Based on Arbitrary String Matches" section). The whole query matches if all individual searches are successful.

Implementation

We implemented our approach in a single command line application written in Kotlin (running on the Java Virtual Machine JVM). For reference, we released it under an open-source license and made it available on GitHub [53].

Our application supports searching a single file (sequentially) or multiple files (in parallel) using multiple threads (one per file). It has two modes: In the default mode, it performs a single search and then prints the extracted buildings to standard out. The benchmark mode runs multiple searches (including a warm up phase) and does not print extracted buildings. It keeps them in memory per search, collects metrics (such as number of buildings extracted, number of search hits and misses, etc.), and prints statistics at the end. We used the benchmark mode for our experiments and recorded the statistics. The results are described in the following section.

Building Experiments

We used the enhanced New York 3D Building Model as test data set (see "Data Types" section). All experiments were conducted on a MacBook (see "Experiment Setup" section). We ran each ad-hoc query two times: Once on a single file, and another time on the whole data set. As described above, our application searches a single file sequentially, but multiple files in parallel. For the whole data set, up to 12 threads (the maximum for CPU of our MacBook) were used.

The following sections give the results of each experiment conducted.

Experiment 1: Search by Free Text

Our first experiment was to find the Empire State Building in the data set. For this, we applied the approach described in the "Extracting Objects Based on Arbitrary String Matches" section and were able to extract a single object. Table 1 shows the measured times of the runs with the single file and the whole data set. The column "Hits" shows how many buildings were extracted. "Misses" denotes how many locations of the search string were identified but did not lead to an extraction. Since the string "Empire State Building" only appears once in the whole data set, there were no such cases. As we show below, this column is more relevant in the other experiments (e.g. when we find a matching value but the key is different). The column "Bldgs/s" shows how many buildings were processed per second.

The SSD in the MacBook Pro is fast enough so the single 736.3 MiB file can be searched within 400 milliseconds. The whole 20.91 GiB data set can even be processed within 6.50 s. Note that multi-threading does not speed up I/O performance here but allows the string searching algorithm to work in parallel with reading new data, which increases the number of buildings processed per second from 60,095 to 166,683.

Experiment 2: Search for Key-Value Pairs

In this experiment, we evaluated the performance of key-value queries. We first searched the data set for buildings whose attribute "ownername" equals "Empire State Building". Since there is only one such building in the entire data set, we also performed another query where we extracted all buildings with a "zipcode" of "10019" (Manhattan). As described in the "Searching for Key-Value Pairs" section, there are two ways to search for key-value pairs: search by value first, and search by key first. Both strategies have benefits and drawbacks and might yield different performance. We therefore executed both queries with both strategies. Table 2 shows the results for the two queries and the two strategies applied to the single file and the entire data set.

The results of the by-value search for the Empire State Building are comparable to the ones from the previous experiment. The queries execute slightly slower because in addition to finding the string "Empire State Building", the key of the generic attribute has to be extracted and compared to "ownername". The by-key searches are also still very fast. Even though the key "ownername" appears 1,082,659 times in the whole data set and only one value matches "Empire State Building", the strategy is only less than 200 milliseconds slower than the by-value strategy.

Searching by value first appears to be faster than the by-key strategy but this only applies to cases like this where the search string does not appear very often in the data set. If we look at the results of the second query, we can see that the by-value strategy is much slower than the by-key strategy. The reason for this is not directly obvious. The string "10019" appears 352,527 in the whole data and "zipcode" appears 1,083,188 times. It seems a lot more checks need to be done when "zipcode" is the search string. However, the by-value strategy—as described in the "Searching for Key-Value Pairs" section—relies on checking if the search string actually is a value of a generic attribute and on extracting its key. If the search string is not a value of a generic attribute but part of a "gml:posList" element, for example, a large portion of the file needs to be searched backward for the string "gen:value", which is time-consuming. The performance of searching by key first, on the other hand, is more predictable and—compared to the free text search—still very fast.

Table 1 Results of building Experiment 1 (search by free text)

Query	Files	Run time (s)	Hits	Misses	Bldgs/s
"Empire State Building"	Single	0.40 ± 0.01	1	0	60,095
	All	6.50 ± 0.14	1	0	166,683

Table 2 Results of building Experiment 2 (Search for key-value pairs)

Query	Strategy	Files	Run time (s)	Hits	Misses	Bldgs/s
"ownername" = "Empire State Building"	By value	Single	0.43 ± 0.03	1	0	55,902
	By key	Single	0.52 ± 0.04	1	23,993	46,227
	By value	All	6.52 ± 0.03	1	0	166,171
	By key	All	6.74 ± 0.02	1	1,082,658	160,747
"zipcode" = "10019"	By value	Single	9.53 ± 0.03	1179	29,520	2522
	By key	Single	0.96 ± 0.03	1179	22,844	25,040
	By value	All	23.54 ± 0.28	1196	351,331	46,025
	By key	All	6.71 ± 0.04	1196	1,081,992	161,466

Experiment 3: Advanced Key-Value Queries

As described in the "Advanced Key-Value Queries" section, searching by key first and then extracting its value allows for more advanced operations such as *less than* and *greater than* comparisons or range checks. In this experiment, we performed three different queries: we searched for buildings with at least four storeys (i.e. where the attribute "numfloors" is greater than or equal to 4), buildings that are located in areas with a zip code between 10018 and 10020 (inclusive), as well as buildings classified as factories (i.e. where the attribute "bldgclass" starts with the letter "F", which stands for 'factory' according the list of building classifications of the City of New York [54]). The results are shown in Table 3.

Compared to the key-value searches from the previous experiment, the queries execute slightly slower because converting and comparing the values takes more time. Finding buildings with at least four storeys is the slowest of all three queries because it yields the most hits. Extracting buildings is therefore rather time-consuming. The search for buildings within a zip code range yields similar results as the zip code query from the previous experiment, which suggests that the range comparison affects performance only slightly.

Experiment 4: Search by Bounding Box

In the geospatial domain, a bounding box is typically specified by four ordinates (*minimum X*, *minimum Y*, *maximum X*, and *maximum Y*). In this experiment, we performed two queries and searched the data set for all buildings that are covered by the bounding boxes (987,700, 211,100, 987,900, 211,300) and the much larger one (950,000, 210,000, 1,000,000, 220,000). Since our application does not support conversion between different spatial reference systems, the query must be specified in the same reference system as the coordinates in the data set (EPSG 2263, US feet).

Table 4 shows that bounding box queries are considerably slower than the queries from the previous experiments. The reason for this is that a large number of coordinates need be extracted and compared. As described in the "Bounding Box Queries" section, our application only supports finding buildings that are covered by a given bounding box, which helps us skip buildings as soon as we find a coordinate outside the bounding box. However, buildings whose coordinates are all covered by the bounding box, need to be processed completely. The performance of bounding box queries therefore highly depends on the data and how many buildings actually match.

Experiment 5: Logical AND

In order to evaluate if multiple queries can be combined, we also searched the data set for buildings within a given zip code range *and* with at least four storeys. Table 5 shows the results of this experiment.

As described in the "Combining Multiple Queries" section, our application executes queries with multiple criteria by searching for a building that matches the first criterion and then searching this building again to evaluate if the other criteria also match. This is reflected in the table by the additional column "1st hits", which denotes the number of buildings that matched the first criterion. The column "Hits" shows the final number of buildings extracted, whereas "Misses" gives the number of buildings that contained the

Table 3 Results of building Experiment 3 (Advanced key-value queries)

Query	Files	Run time (s)	Hits	Misses	Bldgs/s
"numfloors" ≥ 4	Single	3.59 ± 0.40	18,947	5048	6696
	All	9.84 ± 0.34	76,533	1,006,142	110,105
$10018 \leq$ "zipcode" ≤ 10020	Single	0.96 ± 0.11	1821	22,202	25,040
	All	6.75 ± 0.03	1838	1,081,350	160,509
"Bldgclass" starts with "F" (Factory)	Single	0.49 ± 0.02	69	23,927	49,057
	All	6.65 ± 0.06	4167	1,078,514	162,923

Table 4 Results of building Experiment 4 (search by bounding box)

Query	Files	Run time (s)	Hits	Misses	Bldgs/s
Bounding box covered by (987,700, 211,100, 987,900, 211,300)	Single	0.66 ± 0.02	2	609,517	36,421
	All	6.76 ± 0.15	2	12,965,392	160,272
Bounding box covered by (950,000, 210,000, 1,000,000, 220,000)	Single	3.25 ± 0.07	9790	309,304	7396
	All	6.96 ± 0.07	10,763	12,644,194	155,666

Table 5 Results of building Experiment 5 (logical AND)

Query	Files	Run time (s)	1st hits	Hits	Misses	Bldgs/s
(10018 ≤ "zipcode" ≤ 10020 *AND* "numfloors" ≥ 4)	Single	1.11 ± 0.03	1821	1538	22,485	21,656
	All	6.80 ± 0.06	1838	1549	1,081,639	159,329
("numfloors" ≥ 4 *AND* 10018 ≤ "zipcode" ≤ 10020)	Single	2.97 ± 0.03	18,947	1538	22,457	8094
	All	7.38 ± 0.04	76,533	1549	1,081,126	146,807

Table 6 Times it took to import the 3D building model into 3DCityDB and GeoRocket

	Files	3DCityDB	GeoRocket
Import	Single	2 m 25 s	1 m 12 s
	All	2 h 37 m 29 s	1 h 00 m 18 s
Create additional indexes	Single	12 s	–
	All	2 m 49 s	–

search string of the first criterion but where the value did not match.

The results of this experiment indicate that the performance of multi-criteria queries depends on the order of the criteria. Searching for buildings within a certain zip code range first and then comparing the number of floors is faster than the other way round because, in the entire data set, there are less buildings in the zip code range than there are buildings with at least four storeys (which is reflected by the lower number of 1st hits). Since our application is not able to predict the outcome of such a query and therefore cannot automatically optimize the order of criteria, it is up to the user to contribute this knowledge and to specify the query accordingly.

Comparison with Existing Solutions

In this section, we compare the results collected above with two existing database-driven solutions 3DCityDB 4.1.0 (with Importer/Exporter 4.3.0) and GeoRocket (latest 2.0 alpha version from October 2022). For better comparability, we configured both solutions to use PostgreSQL as backend.

Table 6 shows the times it took to import the data into both applications. With 3DCityDB, this process took about two and a half hours for the whole data set. To speed up the queries later, we also had to manually create additional database indexes on the database relation cityobject_genericattrib containing the semantic attributes. This took another

Table 7 Results of executing the example queries using 3DCityDB and GeoRocket

Query	Files	Run time (s) 3DCityDB	Run time (s) GeoRocket	Run time (s) Ad-hoc
"ownername" = "Empire State Building"	Single	0.92 ± 0.02	**0.05 ± 0.01**	0.52 ± 0.04
	All	0.98 ± 0.07	**0.05 ± 0.00**	6.74 ± 0.02
"zipcode" = "10019"	Single	2.42 ± 0.07	**0.30 ± 0.01**	0.96 ± 0.03
	All	2.52 ± 0.11	**0.30 ± 0.01**	6.71 ± 0.04
"numfloors" ≥ 4	Single	15.31 ± 0.46	4.85 ± 0.09	**3.59 ± 0.40**
	All	97.53 ± 10.26	**7.58 ± 0.68**	9.84 ± 0.34
10018 ≤ "zipcode" ≤ 10020	Single	3.19 ± 0.03	3.77 ± 0.17	**0.96 ± 0.11**
	All	15.74 ± 0.48	**2.70 ± 0.02**	6.75 ± 0.03
"bldgclass" starts with "F" (Factory)	Single	1.17 ± 0.01	**0.09 ± 0.01**	0.49 ± 0.02
	All	4.36 ± 0.33	**2.54 ± 0.03**	6.65 ± 0.06
Bounding box covered by (987,700, 211,100, 987,900, 211,300)	Single	0.88 ± 0.00	**0.05 ± 0.01**	0.66 ± 0.02
	All	0.89 ± 0.01	**0.05 ± 0.00**	6.76 ± 0.15
Bounding box covered by (950,000, 210,000, 1,000,000, 220,000)	Single	9.28 ± 0.55	**1.67 ± 0.01**	3.25 ± 0.07
	All	11.46 ± 0.67	**1.61 ± 0.03**	6.96 ± 0.07
(10018 ≤ "zipcode" ≤ 10020 *AND* "numfloors" ≥ 4)	Single	2.98 ± 0.04	5.32 ± 0.06	**1.11 ± 0.03**
	All	28.50 ± 3.06	**2.66 ± 0.03**	6.80 ± 0.06
("numfloors" ≥ 4 *AND* 10018 ≤ "zipcode" ≤ 10020)	Single	3.01 ± 0.03	6.18 ± 0.24	**2.97 ± 0.03**
	All	28.44 ± 7.78	**2.01 ± 0.05**	7.38 ± 0.04

Run times of ad-hoc queries have been copied from above for comparison. Fastest times set in bold

couple of minutes. Although GeoRocket was considerably faster, importing still took about an hour.

Table 7 shows the measured times for performing the individual queries. Note that we copied the run times of the ad-hoc queries from above into the last column for better comparison. The fastest times are highlighted in bold. Also note that the Importer/Exporter tool from 3DCityDB measures the time an operation took in seconds and cuts off the fractional digits, which was too imprecise for our purposes. We therefore had to compile the tool ourselves to print out milliseconds.

As mentioned in the "Implementation" section, the program we implemented for the ad-hoc queries does not write extracted buildings to a file in benchmark mode. To avoid that writing to disk affected the results of GeoRocket, we redirected its HTTP responses to /dev/null. Similarly, since the 3DCityDB exporter only supports file output, we created an in-memory file system and let it write to this.

Before measuring the run times, we verified that each query returned the same number of buildings in 3DCityDB, GeoRocket, as well as with the ad-hoc approach. The only difference we noticed was with the two bounding box queries. While GeoRocket and 3DCityDB find buildings based on the intersection of bounding boxes, our approach only finds those that are completely covered by the bounding box to look for. The ad-hoc implementation therefore returns less buildings (i.e. a subset of those returned by GeoRocket and 3DCityDB). In order to get the same results, the buildings returned by GeoRocket and 3DCityDB would have to be filtered in a post-processing step.

GeoRocket does not support searching for attribute values that start with a given string. We therefore had to emulate the query regarding the building class by a comparison with all nine classes $F1, \ldots, F9$ defined in the list of New York building classifications [54]. In addition, since all attributes in the data set are stored as strings and 3DCityDB does not support automatic type conversion, we had to specify an SQL query that performs an explicit type cast.

Table 7 shows that, in almost all cases, GeoRocket was the fastest solution. Nevertheless, ad-hoc queries were on par, sometimes even beating GeoRocket. They were also in many cases faster than 3DCityDB. This applied to most of the single-file queries but particularly to the queries regarding the number of floors, the zip code range, as well as the boolean combinations thereof. The bounding box queries, however, were faster in 3DCityDB and GeoRocket because of their spatial indexes. Our approach to compare coordinates from gml:posList elements is not very efficient. There is room for improvement here. Nevertheless, 3DCityDB was still very slow for the larger bounding box where a greater number of buildings needed to be extracted.

To summarize, ad-hoc queries performed very well in general. They achieved times that were reasonable for practical use cases and comparable to those of the other 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.

Progressive Bounding Box Index Generation

The previous section has shown that the ad-hoc approach is practical but database-driven solutions are still considerably faster when performing bounding box queries. With ad-hoc queries, all coordinates of a building have to be converted to numbers and compared with the bounding box. An index can speed this up.

In the "Related Work" section, we have discussed existing dynamic indexing approaches such as Database Cracking [39] and Progressive Indexing [40], which build an index on-demand and step by step. These approaches provide a good balance between unindexed and indexed queries.

To build upon this idea, in this section, we combine progressive indexing with our ad-hoc bounding box search. For this, we incrementally create an index and use it to speed up later queries. The concrete procedure is now as follows:

1. If there already is a (probably incomplete) index, use it to search for matching buildings
2. Search for buildings in the not yet indexed areas of the CityGML file using the ad-hoc approach
3. Expand the index

The index is a linked list with one entry per indexed building. An entry contains the bounding box as well as the first and last byte position of the building in the CityGML file. To add a building to the index, we calculate the center of its bounding box and convert it to Morton code [55, 56] with a fixed number of bits (which translates to a fixed coordinate precision). Morton code defines a space-filling curve (also known as Z-order curve), which allows us to sort the index entries and create a one-dimensional binary search tree.

To use the index for a query, we determine the Morton codes for the lower left and the upper right corners of the bounding box to search for. Using binary search, we then identify all entries in the index whose Morton code lies between these two. They are candidates for hits. Due to the fixed bit size of the Morton code and the structure of the space-filling Z-order curve, it is possible that more candidates are found than there are actual hits. Furthermore, it is not guaranteed that the entire geometry of a candidate lies within the searched bounding box. After the binary search, we therefore traverse all candidates and filter out those that do not match.

Each time the user executes a bounding box query, the index is enlarged. The first query completely uses the ad-hoc approach since there is no index yet. However, the first

2000 buildings in the CityGML file are parsed and put into the index. The second query tries to make use of the existing (most likely partial) index to find the first matches. It then performs a new ad-hoc query starting from the byte position after the last indexed building in the CityGML file. At the end of the second query, the next 2000 buildings are indexed, so the index contains 4000 entries and more bytes can be skipped in upcoming queries. This process continues until the whole file has been indexed, in which case ad-hoc queries are not needed anymore. The number 2000 has been chosen empirically and could, in future implementations, be made dynamic or be based on a time budget for example.

Figure 7 shows the run times of several sequential queries for the bounding box (950,000, 210,000, 1,000,000, 220,000) on a single file of our test data set, similar to the experiment performed in the "Experiment 4: Search by Bounding Box" section but with a progressive index. We executed 20 sequential queries five times and plotted the median run times. At the beginning, a query with index (shown in red) is slower than the pure ad-hoc query (blue). This is because, in the first query, the index for the first 2000 buildings is generated, which leads to additional effort. The same applies to the following queries, even though the process becomes continuously faster, since larger parts of the file have already been indexed. After three queries, the variant with the index becomes faster than without. This point strongly depends on the number of indexed buildings. If more buildings are indexed per query, then the first queries are even slower, but the index plays out its advantages more quickly. On the other hand, fewer indexed buildings per query lead to a lower overhead at the beginning, and the index provides its performance benefit at a later point.

The 13th query is significantly faster than the previous one. This can be explained by looking at the distribution of the computing time (see Fig. 8). At this point, the entire file has been indexed. This means that no further computing time has to be used for indexing. In addition, writing the index file can be skipped after the 13th query since it does not change anymore.

This experiment shows that a progressive index can be beneficial for ad-hoc searches. It combines fast initial search with the advantages of an index and represents a

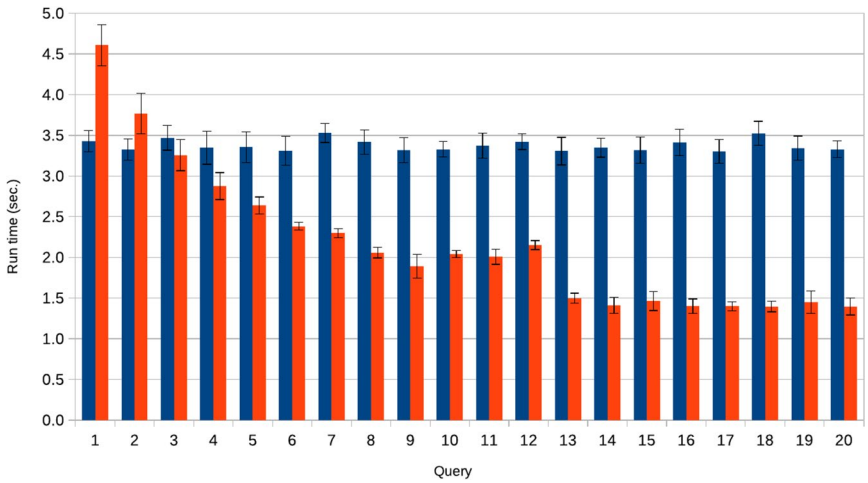

Fig. 7 Median run times (with standard deviations) of sequential queries for the bounding box (950,000, 210,000, 1,000,000, 220,000) on a single file. Red bars represent the run times of queries with progressive indexing, blue bars represent those of ad-hoc queries

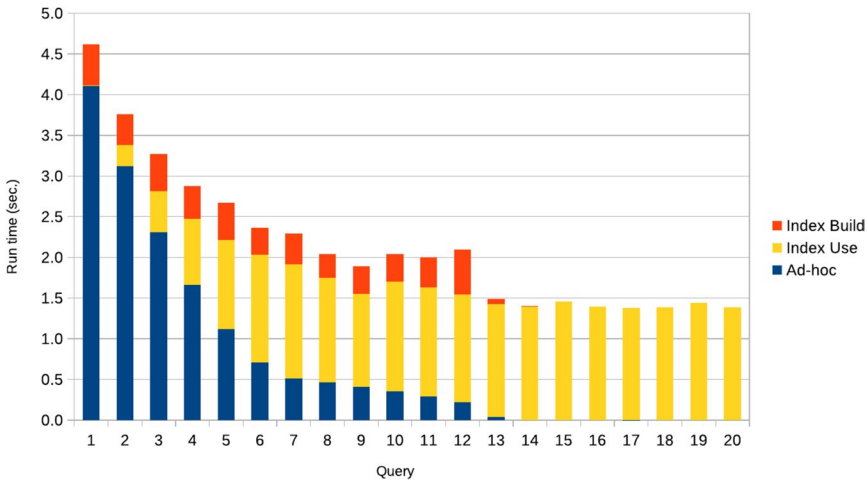

Fig. 8 Composition of the run time when using an index. In the first queries, a lot of ad-hoc search is necessary. Later, more search is done using the index. After the 13th query, the whole file is indexed and no ad-hoc search is needed anymore

compromise between the pure ad-hoc query and the use of a database-based system. In the future, we would like to build on these results and further explore progressive indexes for geospatial applications.

Querying Point Cloud Data

In this section, we illustrate our approach to conducting queries on raw point cloud data files. We first go over the general experiment setup and its differences with regard to the building queries. We then show some optimizations specific to point clouds, which we implemented to speed up the query process. Lastly, we show the results of all conducted point cloud queries and discuss them.

Executing Ad-Hoc Queries on Point Cloud Data

Compared to a textual search, since point cloud data is typically binary, each point in every file of the source data set has to be examined and compared according to the query parameters. This necessitates the following steps:

1. Loading the appropriate bytes that make up the point from the input file(s) into main memory
2. Converting the binary representation into an internal point representation that the query application can work with
3. Applying the query to the internal point representation to decide if the point is a match or not

The expected performance of the first two steps largely depends on the file format of the point cloud. Step 1 (loading the bytes) will be slower the more bytes a single point takes in a given file format. In that regard, a compressed file format will be faster to read from than an uncompressed file format, as a single point on average requires less bytes in the compressed format. Step 2 (converting the bytes to an internal point representation) will be faster the more closely the binary layout of a point in a given file format matches the binary layout of the internal point representation in memory. Here, compressed file formats are at a disadvantage, since the data first has to be decompressed before it can be converted to the internal representation. However, even uncompressed file formats can require some data transformations. As an example, the LAS file format stores point coordinates as normalized integer coordinates, often in a local coordinate system based on the bounding box of the file. Most applications use floating-point values for coordinate representation, so parsing LAS requires a conversion from normalized integer coordinates to floating-point values in world space.

The last step, applying the actual query to the internal point representation, is mostly independent of the file format and is executed through a linear search. Algorithmically, this search could only be sped up if the point data were already sorted based on the primary key of the query. In practice, this is highly unlikely since the point order is mainly dictated by the capturing process. Furthermore, even if the points were sorted by a single attribute, such as their classification, this order would be useless in speeding up a query based on a different attribute. This situation is very well understood in the context of database management systems (DBMSs) and has been covered in the literature for point cloud data by various authors [36, 37].

The goal of this case study is to identify scenarios in which ad-hoc queries on point cloud data can be conducted sufficiently fast to facilitate certain user interactions. Thus, we are aiming at the lowest possible run times for these queries. The two major factors that impact run time are the file format as shown above and data throughput. We propose performance optimizations for both areas and evaluate them in the following sections.

Implementation

For the point cloud experiments, all queries were conducted using a single command line application written in Rust. For reference, we released it under an open-source license and made it available on GitHub [57].

Since the used file formats are binary formats, we use memory-mapped files for best I/O performance. In contrast to the building models scenario, the amount of information in a single point is small, so the main factor for query performance becomes the point throughput, i.e. how many points the application can inspect in a given time. Where possible, we executed the query in parallel, inspecting one file per logical core on the target machine, similar to our implementation of the building data query.

The expected number of positive results from a point cloud query can be several orders of magnitude larger than the results from a query on building models. Gathering the matching points thus has a larger impact on query performance than gathering building data. Additionally, some queries might end up with so many matching points that the result would not fit into main memory. To still get reasonable performance measurements, we simulate the gathering process through a polymorphic `ResultCollector` type with different implementations that either count the number of matching points, print them to the standard output, or store them in an in-memory buffer. The `ResultCollector` interface expects a point structure containing all attributes for a single point. It is called once for each matching point, regardless of the underlying implementation. This way, we can simulate

the process of extracting the necessary information for each matching point independently of any target format or application. Measuring the run time of a query using this approach gives a good measure of the time that finding and extracting the relevant points from the source files takes.

Optimizations

To keep query run times to a minimum, efficient point cloud formats are necessary. At the same time, there is little flexibility from a practical point of view in coming up with new, exotic file formats. The LAS file format and its compressed variant LAZ are by far the most common formats for storing point cloud data captured from a LiDAR (light detection and ranging) scanner. LAS is a standardized binary format that stores point records in an interleaved, fixed-size format. Each point entry in an LAS file has the same size and stores all attributes, such as position, intensity or classification, together in memory. Using this format, it is possible to skip over all bytes that do not belong to the attribute that is queried. Together with memory mapped files, this reduces the amount of work for converting to the internal point format to a minimum.

Special care has to be taken for queries on LAS data based on positions because of the normalized integer coordinates that LAS uses internally. The typical behavior of libraries and tools that read LAS data is to first convert from normalized integer coordinates into floating-point coordinates in world space using the offset and scale parameters in the LAS header. For bounding box queries, there is a more efficient way to do this. Instead of transforming each point into world space, the bounding box can be transformed into normalized integer coordinates once and checked against the integer coordinates. Only for a matching point do we then have to perform the transformation to world space.

For the LAZ file format, there is little potential for optimization. The compression scheme used in LAZ, a form of run-length encoding with a different encoding scheme for each attribute, is computationally expensive. The LAZ format, while being widely used in the industry, has never been officially standardized. We therefore investigated if it would be worthwhile to use an alternative compression scheme to achieve better performance. We chose the LZ4 compression algorithm [58] for this due to its good decompression performance.

We developed and tested the following variants of the LAS file format:

- *LAST*, an LAS variant where the memory layout is transposed so that data is stored attribute-wise instead of point-wise
- *LAZER*, an LAS variant using 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

The memory layout of these file formats is depicted in Fig. 9. The main idea of the *LAST* format is to keep relevant data for queries close together in memory. This is not a new concept, formats such as *3D Tiles* [59] use a similar memory layout. While it would be possible to just compress a whole *LAST* file using LZ4, this would prevent any form of random

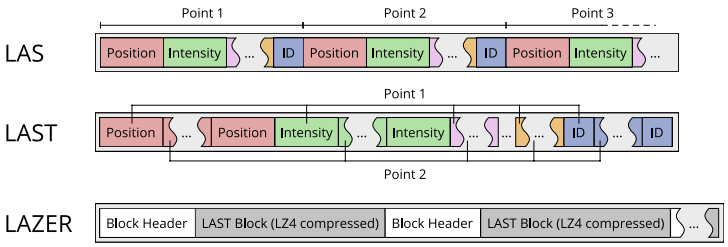

Fig. 9 The memory layout of points in the *LAS*, *LAST* and *LAZER* file formats (Image source: [8])

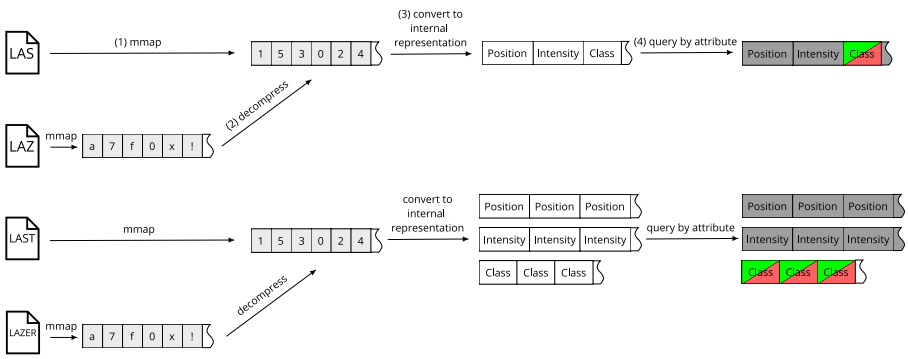

Fig. 10 Ad-hoc query process for point clouds, consisting of the four steps *mapping into memory (1)*, *decompression (2)*, *converting to internal point representation (3)*, and the final *query by attribute (4)* (Image source: [8])

access. Instead, with the *LAZER* format, we tried to stay close to the LAZ format and compressed the data in blocks of a fixed number of points, where each block uses a unique compression context for each of the point attributes within the block. Figure 10 illustrates the ad-hoc query process for point cloud data in all four tested file formats.

On top of the file-based optimizations, we employ basic parallelism by searching each file of the target data sets on a separate thread. While this is far from an optimal scheduling strategy due to differences in the file sizes, not all point cloud file formats support trivial random access so that a single file can be decoded by multiple threads.

Lastly, all of the tested file formats in this paper provide bounding box information for their points. For the bounding box queries, this allows for trivial culling of all files whose bounding boxes do not intersect with the query bounding box. Especially on large data sets made up of many small files, this can lead to a substantial decrease in query execution time.

Point Cloud Experiments

We conducted three experiments for point cloud data:

- Querying point cloud data by bounding box
- Querying point cloud data by bounding box and density
- Querying point cloud data by object class

We converted all of our three point cloud data sets *navvis3*, *doc*, and *ca13* (see "Data Types" section) into each of the formats LAS, LAZ, LAST, and LAZER, and ran each query on every data set in every format.

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* [60] with the default configuration of grouping points into patches of size 400. Afterwards, we manually created a spatial index on the patches. Table 8 shows the time that this process took for the three test data sets.

Experiment 1: Bounding Box Query

In this experiment, we tried to simulate a scenario were points are queried based on a 3D bounding box, so every point within the bounding box has to be extracted from the data set. We defined three bounding boxes with different sizes for each data set, labeled *S*, *L* and *XL*. They were selected in a way so that the *S* bounding box yields about 1% of the total points, *L* yields about 25% of the total points, and *XL* equals the full data set. For the PostGIS comparison, we used the same bounding boxes but converted to polygons in 2D, since PostGIS with the *pgPointclouds* extension does not support spatial queries against 3D bounding boxes. We ran two different queries for each bounding box, one for finding all intersecting patches using the *PC_Intersects* routine, and one for finding every intersecting point using the *PC_Intersection* routine.

Table 8 Times it took to import the point cloud data into PostGIS

Data set	Import time
navvis3	6 m
doc	1 h 58 m
ca13	7 h 34 m

Experiment 2: Bounding Box Query with Maximum Density

In this experiment, we used the same bounding boxes as in experiment 1. However, we also added a maximum density constraint d to the query result, which guarantees that there will be at most one point per d^3 cubic meters returned from the query. We used a maximum density value of 0.1 for the *navvis3* data set, 25 for the *doc* data set, and 100 for the *ca13* data set. The density constraint was applied through simple grid-based sampling: space is divided into an even grid where each grid cell has a side length equal to the maximum density value d. Based on all points that fall into a grid cell, only the closest one to the center of the cell is returned in the query result. With this experiment, we simulated the typical level-of-detail (LOD) approach through spatial subsampling that was popularized by Scheiblauer [61] and Schütz [7].

Since there is no equivalent functionality for querying points by maximum density in PostGIS using the "pgPointclouds" extension, we did not conduct any PostGIS queries for this experiment.

Experiment 3: Query by Object Class

In this experiment, we performed a query for all points with a given object classification. We ran all queries twice, once querying for objects with the object class *building* (classification ID 6 as per the LAS standard), and once for querying for an non-existing object class. We chose the classification ID 19 for the non-existing object class, as this ID was not present in any of the test data sets. Since the *navvis3* data set does not contain object classifications, we conducted this experiment only for the *doc* and *ca13* data sets. For the PostGIS comparison, we used the *PC_FilterEquals* routine which has single-point granularity.

Point Cloud Results

The results of the three point cloud experiments are depicted in Tables 9, 10, and 11 respectively. All tables show the median run time in seconds for each query, as well as the median point throughput, measured in million points per second (Mpts/s). The point throughput is not the actual number of points inspected per second but rather obtained by

Table 9 Results of point cloud Experiment 1

Experiment 1—Bounding Box query							
	Format	Runtime (s)			Throughput (Mpts/s)		
		S	L	XL	S	L	XL
navvis3	LAS	3.52 ± 0.08	3.75 ± 0.11	4.45 ± 0.04	15.94	15.01	12.63
	LAZ	17.45 ± 0.06	17.46 ± 0.04	17.45 ± 0.04	3.22	3.22	3.22
	LAST	1.86 ± 0.04	2.63 ± 0.07	5.32 ± 0.11	30.22	21.34	10.57
	LAZER	12.14 ± 0.06	13.20 ± 0.05	19.18 ± 0.09	4.63	4.26	2.93
	PostGIS (patches)	**0.01 ± 0.02**	**0.30 ± 0.08**	**1.21 ± 0.06**	**5486.89**	**190.45**	**46.46**
	PostGIS (points)	1.26 ± 0.01	39.95 ± 0.21	220.89 ± 7.59	44.49	1.41	0.25
doc	LAS	0.39 ± 0.01	**2.69 ± 0.07**	9.87 ± 1.40	2200.12	**317.56**	86.54
	LAZ	1.77 ± 0.03	18.14 ± 0.42	53.47 ± 0.60	482.49	47.08	15.97
	LAST	0.37 ± 0.01	2.85 ± 0.13	**8.36 ± 0.02**	2337.68	299.12	**102.10**
	LAZER	0.81 ± 0.03	11.45 ± 0.09	37.68 ± 0.80	1054.32	74.59	22.66
	PostGIS (patches)	**0.09 ± 1.05**	5.91 ± 11.52	21.50 ± 38.45	**9064.78**	144.49	39.71
	PostGIS (points)	18.99 ± 0.10	1023.41 ± 0.45	3771.85 ± 4.23	44.97	0.83	0.23
ca13	LAS	1.02 ± 0.01	6.18 ± 0.05	44.53 ± 1.42	2552.66	421.99	58.57
	LAZ	5.44 ± 0.08	39.27 ± 0.35	194.20 ± 2.60	479.41	66.41	13.43
	LAST	**0.90 ± 0.01**	**5.18 ± 0.18**	**36.89 ± 0.24**	**2884.43**	**503.77**	**70.70**
	LAZER	3.67 ± 0.10	34.70 ± 0.40	155.00 ± 4.90	719.63	75.16	16.83
	PostGIS (patches)	1.07 ± 0.40	14.90 ± 5.78	132.05 ± 1.83	2436.96	175.01	19.75
	PostGIS (points)	192.47 ± 2.40	2662.58 ± 30.68	13,578.03 ± 77.63	13.55	0.98	0.19

Point throughput is measured in million points per second (Mpts/s)

Table 10 Results of point cloud Experiment 2

Experiment 2—Bounding box query with max. density							
	Format	Runtime (s)			Throughput (Mpts/s)		
		S	L	XL	S	L	XL
navvis3	LAS	3.61 ± 0.08	4.05 ± 0.16	**7.17 ± 0.07**	15.56	13.86	**7.84**
	LAZ	17.30 ± 0.10	17.80 ± 0.10	17.90 ± 0.10	3.25	3.16	3.14
	LAST	**1.93 ± 0.07**	**3.01 ± 0.05**	7.63 ± 0.12	**29.10**	**18.68**	7.37
	LAZER	12.10 ± 0.04	13.60 ± 0.07	21.36 ± 0.07	4.64	4.13	2.63
doc	LAS	0.46 ± 0.02	**3.44 ± 0.08**	**11.28 ± 0.30**	1854.01	**248.31**	**75.68**
	LAZ	1.78 ± 0.01	18.67 ± 0.15	57.56 ± 0.17	479.78	45.74	14.84
	LAST	**0.42 ± 0.01**	3.78 ± 0.17	11.81 ± 0.08	**2053.87**	225.93	72.32
	LAZER	0.88 ± 0.02	12.93 ± 0.15	42.97 ± 0.80	970.45	66.05	19.87
ca13	LAS	1.29 ± 0.02	7.89 ± 0.30	53.44 ± 2.31	2025.09	330.51	48.80
	LAZ	5.80 ± 0.12	42.60 ± 0.50	204.00 ± 0.80	449.66	61.22	12.78
	LAST	**1.20 ± 0.02**	**7.42 ± 0.13**	**49.57 ± 1.94**	**2179.32**	**351.44**	**52.62**
	LAZER	3.96 ± 0.07	34.72 ± 0.39	172.70 ± 5.89	658.59	75.12	15.10

Point throughput is measured in million points per second (Mpts/s)

dividing the number of points in a data set by the run time of the query.

In all experiments, the compressed file formats LAZ and LAZER are about an order of magnitude slower than the uncompressed file formats LAS and LAST. This is unsurprising, as the process of decompressing the data is both computationally expensive and prevents skipping over irrelevant data. Using a compression algorithm with better decoding performance makes little difference, as the comparison between the LAZ format and our LZ4-compressed LAZER format shows. LAZER files achieve a throughput that is between 0.84 and 2.03 times that of

Table 11 Results of point cloud Experiment 3

Experiment 3—Query by object class					
	Format	Runtime (s)		Throughput (Mpts/s)	
		Building	Non-existing	Building	Non-existing
doc	LAS	8.74 ± 0.24	8.15 ± 0.21	97.67	104.78
	LAZ	62.74 ± 2.29	59.39 ± 0.64	13.61	14.38
	LAST	**3.73 ± 0.02**	**0.83 ± 0.03**	**229.06**	**1034.24**
	LAZER	23.66 ± 0.24	21.35 ± 0.11	36.09	40.00
	PostGIS (points)	165.68 ± 1.73	5.62 ± 0.03	5.15	151.83
ca13	LAS	44.31 ± 1.62	42.60 ± 2.00	58.86	61.23
	LAZ	213.10 ± 2.27	204.47 ± 3.25	12.24	12.75
	LAST	**2.51 ± 0.23**	**2.25 ± 0.06**	**1040.18**	**1158.96**
	LAZER	104.76 ± 4.62	104.41 ± 1.73	24.89	24.98
	PostGIS (points)	114.78 ± 17.44	110.41 ± 0.56	22.72	23.62

Point throughput is measured in million points per second (Mpts/s)

LAZ files, on average about 1.36 times the throughput of LAZ.

Transposing the data, as in the LAST format, has a wider range of effects. The benefits of transposed data become particularly apparent in cases where most points are not a match, and where the attribute that is queried only makes up a small fraction of the memory of a single point. On average, both LAS and LAST achieve about equal throughputs, with LAST being slightly faster. However, in certain situations, LAST vastly outperforms LAS. On the *navvis3* data set, together with the smallest bounding box, querying LAST data is about twice as fast as querying LAS data, as LAST first checks the positions for a match, and only if a match is found loads and parses that other point attributes. A single LAS point in the *navvis3* data set is 26 bytes large, a single position only 12 bytes, which explains the two-fold speedup between LAS and LAST, as LAST on average reads and parses only half as much data as LAS. This effect is amplified significantly in Experiment 3, where a single classification value is only one byte large. The *ca13* data set has a very small number of buildings, hence most points are not a match in the building query. As a result, querying LAST data is almost 20 times faster than querying LAS data, achieving point throughputs of over one billion inspected points per second on a consumer-grade laptop.

In general, the query performance depends a lot on the queried attribute as well as the query parameters. The bounding box queries of Experiments 1 and 2 benefit from the presence of bounding box information within the file headers, and the fact that both the *doc* and the *ca13* data sets consist of a large number of small files. This allows for early culling of files whose bounding boxes do not intersect the query bounding box, which yields very high effective point throughputs of between two and three billion points per second on the *S* query. The larger the bounding box, the less effective this culling becomes, as more and more files fall into the query bounding box. Even without culling, multi-file data sets benefit from multiple processor cores, as multiple files can be queried in parallel. On the reference system, this results in a 5× to 10× increase in point throughput through parallel processing alone.

Lastly, adding a density constraint to the bounding box queries only has a marginal effect on the query run time. The average run time of a bounding box query with maximum point density is only 16% higher than without a density constraint.

Looking at the PostGIS results, we found that even though PostGIS uses a spatial index, run times are often slower than simply searching the uncompressed LAS and LAST files manually. For spatial queries, finding all intersecting patches is very fast using the spatial index, but as soon as the more precise *PC_Intersection* routine is used for obtaining single-point granularity, performance drops by one order of magnitude for small queries, and up to three orders of magnitude for large queries. On top of the raw query run time, one also has to take into account the preprocessing time required for loading the point cloud data into the PostGIS database, which even for a moderately-sized data set can take hours.

Discussion

The experimental results for queries on both building models and point clouds show the potential of ad-hoc queries as an alternative way for users to work with geospatial data. With text-based building model data, our ad-hoc query application is able to answer most queries in a few seconds, oftentimes even outperforming the 3DCityDB system. Only GeoRocket is performing significantly faster, especially for simple

queries, which is unsurprising as it is using an index to answer the queries. The benefit of using an index decreases as the number of positive query results increases, as data throughput becomes a larger bottleneck. In these cases, ad-hoc queries often perform with equal performance to the reference database systems. Possibly the biggest downside to geospatial database systems is the significant time-loss due to uploading and indexing the data, a process that ad-hoc queries circumvent completely. Considering the goal of this paper to provide individual users with ways to quickly work with new or updated data sets, ad-hoc queries on building models have thus been demonstrated to fulfill this goal in many scenarios.

Concerning the point cloud experiments, we have demonstrated that modern commodity hardware is fast enough to execute a variety of query scenarios in interactive or near-interactive time on these data as well. The fact that many point cloud data sets are split into multiple small files can be exploited to make ad-hoc queries by bounding box very fast, especially if the queried region is significantly smaller than the extent of the data set. A common scenario that satisfies these criteria are address-based lookups using a geocoder, which are quite common in the geospatial domain. Combined with the fact that simple grid-based LOD representations can be computed with little overhead, we conclude that it should be possible to implement a point cloud visualization application for these data sets using only ad-hoc queries and processing, instead of the time-consuming preprocessing that is prevalent in the literature. Applying some data layout optimizations, such as the data transposition of the *LAST* format, can yield up to an order of magnitude speedup, pushing the boundary of possible ad-hoc queries further towards multi-billion point data sets.

On the other hand, many point cloud data sets are stored in compressed formats. Their decompression overhead makes them ill-suited for the kinds of ad-hoc queries that we evaluated in this paper. While there are fast compression algorithms available, even a highly efficient variant such as *LZ4* is still at least an order of magnitude slower than working with uncompressed data. Depending on the use case, resorting to storing point cloud data in an uncompressed format might be feasible. Disk space is generally assumed to be cheap but data transfer over a network might not be. A hybrid solution might store the point cloud locally in an optimized, uncompressed format, and perform on-the-fly compression whenever data has to be sent to a client over the network.

We primarily see the results of the experiments as a justification for taking ad-hoc queries into consideration when developing applications that have to work with geospatial data. Even if the use of a RDBMS or any other form of index is warranted in various scenarios, the inherent performance of modern commodity hardware should not be underestimated. We already identified individual users working with new data as one of the main areas of application for ad-hoc queries. Beyond that, we see many other areas of application, for example to quickly identify relevant subsets of larger data sets that can then be indexed more precisely. Ad-hoc queries can also be used for quality control of data, identifying potential flaws in the data before a resource-intensive indexing process has been started. Ultimately, we believe ad-hoc queries can become one more tool in the toolbox of application developers. In doing so, one has to be aware of the downsides of ad-hoc queries: They are fundamentally limited by the lack of scalability on a single system, so multi-terabyte data sets will yet be out of reach for the next couple of years, as will multi-user scenarios since a single ad-hoc query can occupy most of the system's resources. On top of that, ad-hoc query performance depends heavily on the used data formats. Especially for binary formats, an efficient memory layout can make a large difference in the expected performance—a fact which in our opinion has to be taken into account when developing and improving data formats for geospatial data in the future.

Conclusion

In this case study, we conducted a series of experiments that demonstrate that it is possible to perform common queries on raw, unindexed geospatial data in single-user scenarios on commodity hardware while achieving interactive or near-interactive response times. To demonstrate this, we wrote two specialized applications based on common search algorithms for performing queries *ad-hoc* on both buildings models in the *CityGML* format as well as point clouds in the *LAS* format and variations thereof. We gathered experimental data on the run time of a range of common geospatial queries based on user-defined attributes as well as bounding boxes and compared our *ad-hoc* solutions to common database management systems. In many of the evaluated scenarios, *ad-hoc* queries can be answered in similar or less time than with the database management systems, especially taking into account the substantial time required for uploading the test data sets into the databases. For point cloud queries, we also evaluated how incremental changes to common data formats can help achieve substantial improvements in query speed. While we found compressed point cloud formats to be unsuited for ad-hoc queries, the same queries on uncompressed formats seldom took more than a few seconds. With the trivial change of storing points in a transposed memory layout, we were able to achieve query throughputs of more than a billion points per second on commodity hardware. Applications could

thus benefit if more point cloud data were stored in such a transposed memory layout natively.

The experimental results are a strong indicator that in a single-user setup, ad-hoc queries are a viable alternative to the classic process of uploading and indexing geospatial data into a RDBMS. It is noteworthy that our approach is very simple, requiring neither sophisticated algorithms nor exotic data formats. Since the ad-hoc queries can be conducted on raw files, it simplifies and speeds up the access to geospatial data, enabling users to quickly interact with and evaluate the data. We believe our work can thus form the basis for the implementation of on-the-fly query and processing systems for various geodata.

While analysis scenarios are typically more forgiving when it comes to query run times, visualization applications have very strict interactivity requirements. Therefore in the future we want to evaluate the quality and usability of a visualization application for building models and point clouds using only ad-hoc queries. To get rid of the shortcomings of unindexed data, we also plan to evaluate how ad-hoc queries can serve as guidelines for indexing only relevant data on the fly, instead of indexing all data upfront. As an example, the sequential scan necessary for ad-hoc queries can be used to build a rough index which in turn can then be used to guide and speed up future ad-hoc queries. Furthermore, our experiment on progressive bounding box indexing has shown that the combination of ad-hoc queries and adaptive indexing provides a useful balance between quick response times and indexing overhead. In the future, we would like to explore this idea further and investigate its use for geospatial applications in particular.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data availability All data used for the experiments is publicly available.

Declarations

Conflict of interest The authors declare that they have no Conflict of interest. This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

1. Yang C, Goodchild M, Huang Q, Nebert D, Raskin R, Xu Y, Bambacus M, Fay D. Spatial cloud computing: How can the geospatial sciences use and help shape cloud computing? Int J Dig Earth. 2011;4(4):305–29. https://doi.org/10.1080/17538947.2011.587547.
2. Petri G. An introduction to the technology mobile mapping systems. GeoInformatics. 2010;13(1):32–43.
3. Puente I, González-Jorge H, Arias P, Armesto J. Land-based mobile laser scanning systems: a review. Int Arch Photogrammetry Remote Sens Spatial Inform Sci. 2011;XXXVIII–5/W12:163–8. https://doi.org/10.5194/isprsarchives-XXXVIII-5-W12-163-2011.
4. Arefi H. From LIDAR point clouds to 3D building models. PhD thesis, Institute for Applied Computer Science-Bundeswehr University Munich; 2009.
5. European Space Agency ESA: Sentinel Online. Accessed: 2022-11-09 (2022). https://sentinel.esa.int.
6. European Union: Copernicus Programme. Accessed: 2022-11-09 (2022). https://www.copernicus.eu.
7. Schütz M. Potree: Rendering large point clouds in web browsers. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology; 2016.
8. Bormann P, Krämer M, Würz H.M. Working efficiently with large geodata files using ad-hoc queries. In: Proceedings of the 11th international conference on data science, technology, and applications DATA. Setúbal, Portugal: SciTePress; 2022. p. 438–45. https://doi.org/10.5220/0011291200003269. INSTICC
9. Boyer RS, Moore JS. A fast string searching algorithm. Commun ACM. 1977;20(10):762–72. https://doi.org/10.1145/359842.359859.
10. Horspool RN. Practical fast searching in strings. Softw Pract Exp. 1980;10(6):501–6. https://doi.org/10.1002/spe.4380100608.
11. Raita T. Tuning the Boyer–Moore–Horspool string searching algorithm. Softw Pract Exp. 1992;22(10):879–84. https://doi.org/10.1002/spe.4380221006.
12. Knuth DE, Morris JH Jr, Pratt VR. Fast pattern matching in strings. SIAM J Comput. 1977;6(2):323–50. https://doi.org/10.1137/0206024.
13. De V, Smit G. A comparison of three string matching algorithms. Softw Pract Exp. 1982;12(1):57–66. https://doi.org/10.1002/spe.4380120106.
14. Aho AV, Corasick MJ. Efficient string matching: an aid to bibliographic search. Commun ACM. 1975;18(6):333–40. https://doi.org/10.1145/360825.360855.
15. Commentz-Walter B. A string matching algorithm fast on the average. In: Maurer HA, editor. International colloquium on automata, languages, and programming. London: Springer; 1979. p. 118–32. https://doi.org/10.1007/3-540-09510-1_10.
16. Navarro G. Regular expression searching on compressed text. J Discrete Algorithms. 2003;1(5–6):423–43. https://doi.org/10.1016/S1570-8667(03)00036-4.
17. Ganty P, Valero P. Regular expression search on compressed text. In: 2019 data compression conference (DCC). New York: IEEE; 2019. p. 528–37. https://doi.org/10.1109/DCC.2019.00061.
18. Ferragina P, Manzini G. Indexing compressed text. J ACM (JACM). 2005;52(4):552–81. https://doi.org/10.1145/1082036.1082039.
19. Gustafsson P, Sagonas K. Efficient manipulation of binary data using pattern matching. J Funct Program. 2006;16(1):35–74. https://doi.org/10.1017/S0956796805005745.
20. PostGIS Project: PostGIS-Spatial and Geographic objects for PostgreSQL. Accessed: 2022-11-09 (2022). https://postgis.net/.

21. Oracle: Oracle's Spatial Database. Accessed: 2022-11-09 (2022). https://www.oracle.com/database/spatial/.
22. Microsoft: SQL Server. Accessed: 2022-11-09 (2022). https://www.microsoft.com/en-us/sql-server.
23. Open Source Geospatial Foundation: GeoServer. Accessed: 2022-11-09 (2022). http://geoserver.org/.
24. OSGeo Foundation: Deegree. Accessed: 2022-11-09 (2022). https://www.deegree.org/.
25. Yao Z, Nagel C, Kunde F, Hudra G, Willkomm P, Donaubauer A, Adolphi T, Kolbe TH. 3DCityDB—a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. Open Geospat Data Softw Stand. 2018;3(1):5. https://doi.org/10.1186/s40965-018-0046-7.
26. Krämer M. Georocket: a scalable and cloud-based data store for big geospatial files. SoftwareX. 2020. https://doi.org/10.1016/j.softx.2020.100409.
27. Rapidlasso GmbH: LAStools: award-winning software for rapid LiDAR processing. Accessed: 2021-06-08 (2021). http://lastools.org/.
28. Isenburg M, Liu Y, Shewchuk J, Snoeyink J, Thirion T. Generating raster DEM from mass points via TIN streaming. In: Raubal M, Miller HJ, Frank AU, Goodchild MF, editors. International conference on geographic information science. London: Springer; 2006. p. 186–98. https://doi.org/10.1007/11863939_13.
29. American Society for Photogrammetry and Remote Sensing (ASPRS): LAS specification, version 1.4 - R13. Accessed: 2022-11-09 (2013). https://www.asprs.org/wp-content/uploads/2010/12/LAS_1_4_r13.pdf.
30. Schütz M. Potree. Accessed: 2022-11-09 (2022). https://github.com/potree/potree/.
31. Cesium GS, Inc.: CesiumJS - Cesium. Accessed: 2022-11-09 (2022). https://cesium.com/platform/cesiumjs.
32. Hobu Inc.: Entwine. Accessed: 2022-11-09 (2019). https://entwine.io/.
33. Bormann P, Krämer M. A system for fast and scalable point cloud indexing using task parallelism. Smart Tools Apps for Graph Eurograph Italian Chapt Conf. 2020. https://doi.org/10.2312/stag.20201250.
34. Schütz M, Ohrhallinger S, Wimmer M. Fast out-of-core octree generation for massive point clouds. Comput Graph Forum. 2020;39(7):1–2.
35. El-Mahgary S, Virtanen JP, Hyyppä H. A simple semantic-based data storage layout for querying point clouds. ISPRS Int J Geoinform. 2020. https://doi.org/10.3390/ijgi9020072.
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. https://doi.org/10.1016/j.cag.2015.01.007.
37. 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 Photogrammetry Remote Sens. 2017;127:39–56. https://doi.org/10.1016/j.isprsjprs.2016.06.012.
38. Ramsey P, Blottiere P, Brédif M, Lemoine E. pgPointcloud—a PostgreSQL extension for storing point cloud (LIDAR) data. Accessed: 2021-07-19 (2021). https://pgpointcloud.github.io/pointcloud/index.html.
39. Idreos S, Kersten ML, Manegold S. Database cracking In: CIDR, vol. 7; 2007. p. 68–78.
40. Holanda P, Raasveldt M, Manegold S, Mühleisen H. Progressive indexes: indexing for interactive data analysis. Proc VLDB Endow. 2019;12(13):2366–78. https://doi.org/10.14778/3358701.3358705.
41. Hohenstein M. Progressive indexing for interactive analytics. In: Thor A, Totzauer S, editors. GvDB. CEUR workshop proceedings, vol. 3075; 2021.
42. Idreos S, Alagiannis I, Johnson R, Ailamaki A. Here are my data files. Here are my queries. Where are my results? In: Proceedings of 5th biennial conference on innovative data systems research CIDR; 2011.
43. Alagiannis I, Borovica R, Branco M, Idreos S, Ailamaki A. Nodb: efficient query execution on raw data files. In: Proceedings of the 2012 ACM SIGMOD international conference on management of data. SIGMOD'12. Association for Computing Machinery, New York, NY, USA; 2012. p. 241–52. https://doi.org/10.1145/2213836.2213864.
44. Karpathiotakis M, Branco M, Alagiannis I, Ailamaki A. Adaptive query processing on raw data. Proc VLDB Endow. 2014;7(12):1119–30. https://doi.org/10.14778/2732977.2732986.
45. Fraunhofer IGD: Enhanced NYC 3-D building model. Version 20v5. Accessed: 2022-11-09 (2021). https://github.com/georocket/new-york-city-model-enhanced/.
46. Watershed Sciences, Inc: PG &E Diablo Canyon Power Plant (DCPP): San Simeon, CA Central Coast. Accessed: 2022-11-09. https://portal.opentopography.org/datasetMetadata?otCollectionID=OT.022013.26910.2.
47. Gröger G, Kolbe TH, Nagel C, Häfele K-H. OGC city geography markup language (CityGML) encoding standard 2.0.0. Open Geospatial Consortium, Rockville, USA; 2012.
48. Department of Information Technology & Telecommunications (DoITT) of the City of New York: NYC 3-D Building Model. Accessed: 2022-11-09 (2018). https://www.nyc.gov/site/planning/data-maps/open-data/dwn-nyc-3d-model-download.page.
49. Department of City Planning (DCP) of the City of New York: Primary Land Use Tax Lot Output (PLUTO). Accessed: 2022-11-09 (2022). https://www.nyc.gov/site/planning/data-maps/open-data/dwn-pluto-mappluto.page.
50. NavVis: NavVis M6 Point Cloud Data. Accessed: 2022-11-09. https://www.navvis.com/resources/specifications/navvis-m6-sample-data.
51. Washington, DC: District of Columbia—Classified Point Cloud LiDAR. Accessed: 2022-11-09. https://registry.opendata.aws/dc-lidar/.
52. Egenhofer MJ, Franzosa RD. Point-set topological spatial relations. Int J Geograph Inform Syst. 1991;5(2):161–74. https://doi.org/10.1080/02693799108927841.
53. Fraunhofer IGD: Ad-hoc queries on 3D building models—Benchmark implementation. Accessed: 2022-11-09 (2022). https://github.com/igd-geo/adhoc-queries-building-models.
54. The City of New York: Building Classification. Accessed: 2022-11-09 (2022). https://www1.nyc.gov/assets/finance/jump/hlpbldgcode.html.
55. Gaede V, Günther O. Multidimensional access methods. ACM Comput Surv. 1998;30(2):170–231. https://doi.org/10.1145/280277.280279.
56. Morton GM. A computer oriented geodetic data base and a new technique in file sequencing; 1966
57. Fraunhofer IGD: Ad-hoc queries on point clouds—Benchmark implementation. Accessed: 2022-11-09 (2021). https://github.com/igd-geo/adhoc-queries-pointclouds.
58. LZ4 Team: Extremely Fast Compression algorithm. Accessed: 2022-11-09 (2022). https://github.com/lz4/lz4.
59. Cesium Team: CesiumGS/3d-tiles: Specification for streaming massive heterogeneous 3D geospatial datasets. Accessed: 2022-11-09 (2018). https://github.com/CesiumGS/3d-tiles.
60. Contributors PDAL. PDAL Point Data Abstract Lib. 2018. https://doi.org/10.5281/zenodo.2556738.
61. Scheiblauer C. Interactions with gigantic point clouds. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology; 2014.