MSc thesis in Geomatics

The efficient web handling of CityJSON

Jordi van Liempt

June 2020

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Geomatics

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

JSON JavaScript Object Notation 1
blob Binary Large OBject 25
CBOR Concise Binary Object Representation 2
BSON Binary JSON 8
glTF Graphics Library Transmission Format 17
OGC Open Geospatial Consortium 1
I3S Indexed 3D Scene layer 2
BSON Binary JSON 8
CBOR Concise Binary Object Representation 2
UBJSON Universal Binary JSON Specification 8
LZ77 Lempel-Ziv 1977 8
GML Geography Markup Language 1
b3dm Batched 3D Model 2
LoD Level of Detail 2
DBMS Database Management System 32
REST Representational state transfer 31
CRS coordinate references system 18
CQL Contextual Query Language 19
WFS Web Feature Service 1
WMS Web Map Service 1

1 Introduction

1.1 Motivation and problem statement

The possibilities of 3D city models for the analysis of the built environment are increasingly explored, and there is a continuous development in improvements on their inner workings. A list of applications domains and use cases for which they are exclusively suitable, or more so than traditional 2(.5)D data, includes estimation of solar irradiation, visibility analysis, energy demand estimation Biljecki et al. [2015]. While many of these applications foremost need information on the geometries of the objects that are included in the study area, some require (detailed) semantics on their parts as well. Especially in the latter case this makes for a rapid increase in file size when the area of study is expanded. At the same time, there is an increase in popularity of the dissemination and direct use of geoinformation on the web [Dempsey, 2018; Mango, 2017]. This presents challenges in efficiency, as the network becomes a potential new bottleneck in performance.

In this thesis I focus on the improvement of the inner workings of 3D city models to attempt to relieve this problem, in specific for spreading and using them more efficiently on the web. I do this by investigating and testing different compression techniques on 3D city models stored in the CityJSON [CityJSON, 2019] format. These techniques decrease the file sizes of datasets allowing for faster transmission over a network, but on the other hand add additional steps to process them. The goal of using a compression technique is to result in a net speed gain, meaning that the time that is saved on download time should be larger than the additional time that it costs to decompress the file after receival.

The creation of CityJSON in 2017 exemplifies the continuous developments on 3D city models, it being an encoding that is used to store the geometries and semantics of 3D cities and landscapes [Ledoux et al., 2019]. It follows the Open Geospatial Consortium (OGC) standardised CityGML data model but aims to be a better alternative to it by using the JavaScript Object Notation (JSON) encoding, which for example has the advantage of making it more suitable as an international 3D geoinformation standard for web use. The main reason for its creation is that Geography Markup Language (GML) in general is verbose and more complex to work with. CityJSON is more compact since JSON files are already smaller than GML, but also because it has better compression possibilities than CityGML (by storing geometries in the Wavefront OBJ [Reddy] style), resulting in on average 6 times smaller file sizes. In addition to that, JSON is smoother to use than GML files as it is a simpler file format, making it easier to read for humans and to be used by developers.

The compactness is an especially useful characteristic for web use since it decreases the time of data transfer over a network, and the use of compression techniques can decrease the sizes of datasets even further. Besides the potential improvement in data transmission efficiency that I examine in this thesis, CityJSON has not yet been fully tested in a web environment. Currently, 2D geoinformation is often disseminated through Web Feature Service (WFS) and Web Map Service (WMS) services, and the OGC recently has initiated the OGC API [Open Geospatial Consortium, 2019c] that specifies how online access should be provided to collections of geospatial data. Because it is comprised of multiple blocks and allows for extensions, its capabilities can flexibly be increased, giving way to the support for 3D geoinformation and performing (spatial) operations on it through the web [Open Geospatial Consortium, 2019c]. Efforts are already made to make CityJSON compliant with the OGC API specifications [Ledoux, 2020]. With this potential in mind, in this thesis I already consider

the influence of the use of different compression techniques on the performance of a variety of data operations with CityJSON.

In order to further improve the efficiency of CityJSON, this thesis investigates compression techniques that can be implemented to decrease its average file size and potentially improve its web use speed. Compression techniques can be all-purpose, but can also be created for a specific type of data. An example of the former that I test is zlib, with which a complete CityJSON dataset can be compressed at once. The downside of using such a technique is that the file has to be decompressed completely before it is useable. This can be inefficient in cases where this is not needed, for example when querying only one feature. In addition to that, the decompression time can be relatively long as shown in Section 6.2. The decompression time is more important than compression time because the combination of a small file size and good decompression performance decreases the time it takes for a file to be useable after receival, while compression is (in most cases) part of dataset preparation.

CityJSON has several characteristics for which specific compression techniques can be suitable. The main ones that are considered are the the geometries of features, their attributes and semantics, and the JSON encoding. Using specific techniques on them can allow for partial decoding of compressed datasets and yield different performance results. With this it is important that no information is lost, as techniques can be lossy or lossless. The decompression speed of an algorithm is presumed to be more important than the compression speed, because decompressing is always necessary before a file is useable, while the compression is mostly part of data preparation.

A way that could be incorporated for the compression of the geometries is compression with the Draco library [Google, 2019b], which is an open-source library and can be used to compress and decompress 3D geometric meshes. The library makes use of several lossless compression techniques to encode the vertices and connectivity information of meshes, such as delta and entropy coding for the former and the Edgebreaker algorithm and parallelogram prediction for the latter (see Sections 2.3.3 and 2.3.4). Draco is specifically designed to improve the efficiency of their storage and transmission [Google, 2019b]. However, it assumes simple triangular geometries as input, while CityJSON can contain complex collections of volumes that potentially have holes and cavities. Therefore, it requires testing to find out if it is suitable to use.

In addition, the attributes and semantics can be compressed using techniques such as zlib, Huffman coding, and smaz (see Subsections 2.2.5, 2.2.4, 2.2.6). These reduce the redundancy in data in different ways, which I explain in Chapter 2. Lastly, the JSON encoding is human-readable which comes at the cost of a slower performance and higher file size than a binary format. There are binary formats such as Concise Binary Object Representation (CBOR) into which JSON can be encoded that approach the latter's key-value structure, which would yield performance benefits at the cost of being able to easily inspect a file without decoding it.

Different file formats already exist for the web use of 3D city models—Batched 3D Model (b3dm) and Indexed 3D Scene layer (I3S) [Analytical Graphics, Inc., 2019; Esri, 2019b], see Sections 3.3.1 and 3.4—and they focus on high performance, which this thesis aims at as well. But for performance they do not only regard the time that it simply takes to load a file but also incorporate other features that benefit user experience such as tiling, which enables only specific parts of the dataset to be loaded, or rendered in a different Level of Detail (LoD). In addition, they combine binary structures and JSON and allow for the use of Draco geometries. However, the focus on high performance makes them relatively complex to use for developers, as opposed to CityJSON that aims to be developer-friendly. While CityJSON or compressed variants might not be able to outperform these other file types, it is still suitable to use on the web when the best performance is not necessary, because of its ease of use. This does not mean that its performance should be neglected—compression can be of added value, but only if CityJSON's simplicity is mostly retained.

Besides that, the purpose of tiling is mostly to improve the streaming and visualisation of geospatial data [Analytical Graphics, Inc., 2018]. There are however more use cases for which a geoinformation

web service could be utilised, some of which could be easier to implement with CityJSON. Users might want to download a subset of the data per specified attributes or bounding box to avoid having to download unneeded information, editing functionalities can come in handy for crowd-sourced initiatives such as OpenStreetMap, and the ability to do spatial analysis can help it function as a cloud-based GIS application. Therefore the influence of using compression techniques is tested for several types of data operations.

Concretely, I test a variety of combinations of compression techniques on multiple CityJSON datasets, and assess them on their performance when executing different data operations with them. The operations are divided into five categories: visualisation, querying, spatial analysis, editing of attributes, and editing of geometries. Besides that the difference in file size is considered for lower storage space and the lossiness induced by the combination of compression techniques as all important information should be retained. The tested datasets have different characteristics since they can influence the workings of compression techniques, and I introduce a web application that is created for the testing of implementations. This testing platform is a Flask application that acts as a server to process requests, which in turn renders a web page that works with Three.js and Harp.gl to render datasets.

1.2 Research question

The main research question of the thesis is as follows: *To what extent can the implementation of different compression techniques improve CityJSON's web use experience, considering file size, visualisation, querying, spatial analysis, and editing performance?*

To assess the impact that different compression techniques have, I take uncompressed CityJSON performance as the baseline and performance indicators are defined as derived from Section 1.1, shown in Table 1.1. I name it uncompressed for simplicity but all tested CityJSON datasets include the `"transform"` member, meaning that the origin of the set of vertices is moved to (0, 0, 0) and that their coordinates are represented as integers (see Section 3.1 for a more detailed explanation)—which is in fact a form of compression already. Some datasets originally had `"transform"` already and Draco requires vertices to be stored in this way to compress them, which is why I have chosen to make the datasets more uniform as to have one factor less to consider when interpreting the results.

For the visualisation, Three.js is used and the performance depends on the speed of the parsing of CityJSON geometries to a Three.js mesh and rendering it. As for querying and spatial analysis, the performance can be measured by the difference between the moment that a request has been made to perform the operation and the return of results. The main difference here is that the former needs object attributes to be decoded, while the latter requires their geometries. Since the decoding time and the ability to decode objects separately are characteristics that set compression methods apart, it does not matter which type of query or spatial analysis is tested. However, because of the latter characteristic it is fairer to test the performance of this on both a single and on all objects of the dataset. The editing is done in a similar way, with the difference being that the edited information needs to be compressed again, for which the execution time of the algorithm becomes relevant. For these four performance indicators the execution time of corresponding operations—which are defined in Chapter 5—is benchmarked.

Besides these, there are two characteristics that do not involve a time benchmark. The file size can be measured by the amount of bytes that a file takes up, with smaller files having the advantage that less storage space is needed and that they are transmitted over the web more quickly. The last point is the assessment of the lossiness of a compression type. No vital information from the original dataset should be lost—for instance loss of object ordering is acceptable while reduced coordinate accuracy and precision is not—as this would render the compression useless in this context.

Compression type performance indicators	Section
Visualisation time	6.4
Querying time	6.5
Spatial analysis time	6.6
Editing time	6.7 & 6.8
File size compression	6.3
Lossiness	5.6.1

Table 1.1: The six performance indicators on which variants of compressed CityJSON are assessed, and the sections where they are addressed

1.3 Scope

The average execution time of the (de)compression algorithms are already indirectly assessed with the indicators. Decompression is always done before a file can be used, and compression is done after editing a file. For this reason these are not seen as separate performance indicators. They are still benchmarked however because it can give more insight on what causes the differences in performance between compression types. Additionally a significantly longer execution time can make an implementation less desirable.

What I do not cover in the thesis is the compression of textures and materials. They are structured differently from the other parts of CityJSON and would therefore have to be investigated thoroughly to find suitable compression techniques, and most existing datasets (as retrieved from CityJSON [2020]) do not contain them. For this reason I see it as future work.

The OGC API is mostly introduced (in Section 3.5) to give context about the current developments on the dissemination of geospatial information. It is not implemented in this thesis, but Section 5.6 shows an experiment on the compression of datasets that is useful when streaming datasets. Features can be compressed individually and streamed following the specification, so I did initial compression implementations in this way to see if it is worth it to investigate it further.

Similarly tiling is explained (see Section 3.3). While being an important topic related to efficient web use of geoinformation, it is too complex to assess different ways to implement it into CityJSON within the time frame of the research and see if it is worthwile to do it. However, because of the relevance, it is discussed whether or not it is viable to incorporate a tiling specification into CityJSON as an idea for future work, and how compression could be related to it.

Lastly, the "web use experience" part of the main research question does not encompass the assessment by humans. I am assuming that the six aforementioned objective main criteria of table 1.1 can sufficiently represent the user experience, with the presumption that they prefer faster performance.

1.4 Thesis overview and preview of results

Chapter 2 discusses theory on compression and techniques that are either used in the thesis or concepts that are needed to explain other techniques. Subsequently Chapter 3 gives an explanation on CityJSON and existing file formats and concepts that are related to efficient web use of 3D models. Chapter 4 gives an overview of the methodology used to prepare CityJSON datasets, compress them, and assess their performance. Then, Chapter 5 explains with more details how the research is carried out by explaining variants of compressed CityJSON and the testing platform, and experiments that require further exploration, are done to make decisions in the implementation, or

help explaining the results. After that Chapter 6 shows the results of the six performance indicators for different compression implementations. Finally, I discuss in Chapter 7 what has been achieved, what I think would be the best compression options to consider to actually implement, and how CityJSON could evolve with tiling and the OGC API.

To do: give a preview of the results.
Throughout the whole thesis: refer to the tools part in the appendix when mentioning tools. Still have to write that part.
Also some figures should be included as vector images.

2 Theoretical background on compression

2.1 Data compression in general

Data compression has the goal to arrive at a compact representation of data by reducing its redundancy, while being suitable to use for specific needs [Sayood, 2017]. This latter part indicates that not always the most compact representation is the most optimal, as (de)compression speed can be considered of importance, as well as the technique being lossless or lossy (which is explained in Section 2.1.1). With geoinformation in mind, a high decompression speed can give a better user experience as the rendering speed is increased, but loss of precision and accuracy that alters topological relationships is not acceptable.

All data compression techniques exploit either the structure or redundancy of the data or both. This means that techniques have to be designed in such a way to optimally make use of this, and prior knowledge on the data allows for the use of a more suitable technique (in speed, lossiness, or storage space) rather than using an all-purpose method [Sayood, 2017].

2.1.1 Lossless versus lossy compression

Lossless compression methods will always result in encodings that are able to be restored to their original form. This means that both the full information that it contained and its exact structure need to be retained for a method that is truely lossless. Text for example should virtually always be compressed losslessly, as small deviations can give outcomes that have a considerably different meaning [Sayood, 2017]. Examples of losslessly compressed file formats are RAR and ZIP, and run-length encoding, Huffman coding, and LZ77 are examples of lossless compression techniques which are mentioned in Section 2.2.

With lossy techniques however, data can potentially be compressed further to an even smaller representation. A method is labeled as such when information that is deemed as unimportant will be lost in the process, resulting in data that is different from the original as it is encoded and subsequently decoded. Examples are the removal of audio and visual frequencies that (most) humans are not able to perceive [Sayood, 2017]. Popular file formats such as MP3 and JPG are lossy, while examples of their lossless counterparts are FLAC and PNG. In the context of geoinformation, it may be acceptable for coordinates to be compressed in a lossy manner when they are of higher precision than necessary.

2.2 Generic compression techniques

In this Section, techniques that can be used or taken inspiration from for the compression of CityJSON are discussed, as well as basic methods that need to be covered since they are used to explain other techiques.

2.2.1 Binary file structure

A binary file can be defined as "any file that contains at least some data that consists of sequences of bits that do not represent plain text" [The Linux Information Project, 2006]. It usually is more compact than human-readable files, which is why it is suitable to use for compressed data. Additionally, by the addition of the type and length of a part of the data it can be read faster by machines [BSON, 2019].

There are several existing options to almost directly encode JSON into binary. The following ones are included in the JSON for Modern C++ library [Lohmann, 2020] and can therefore be considered as the most important ones: Binary JSON (BSON) [BSON, 2019], CBOR [CBOR, 2019], MessagePack [MessagePack, 2019], and Universal Binary JSON Specification (UBJSON) [UBJSON, 2020].

It is possible to write an original binary format specification to encode CityJSON, but to keep it close to the original format it is chosen to only investigate the aforementioned methods that can directly encode the format. In section 3.6 CBOR is further explained, which is chosen as the one to test on CityJSON since it turned out to be the most efficient one regarding both file size and processing speed according to a benchmark of Zderadicka [2017].

2.2.2 Run-length encoding

Run-length encoding takes advantage of the occurrence of the same symbol in sequence. Following its basic principles, such a repeated character is encoded by the character itself, preceded by an integer indicating the number of times that it is repeated. In order to make the encoding more sophisticated, a negative integer can be used within the code to denote the amount of next characters that only occur once. This avoids having to always place a "1" in front of characters that are not repeated [Bourke, 1995].

As an example, the first string below would be encoded as the second string (adapted from Bourke [1995]).

Original string:

```
abcddddddcbbbbabcdef
```

Encoded string:

```
-3 a b c 6 d -1 c 4 b -6 a b c d e f
```

2.2.3 LZ77

Similarly to run-length encoding, Lempel-Ziv 1977 (LZ77) exploits the sequential repetition of the data input. However, rather than working with single characters, it finds repeated blocks. While reading the data, the algorithm keeps record of a set amount of previous characters named sliding window. As the window slides, a new block identical to a block elsewhere in the window is replaced by two numbers that represent a distance (D) and a length (L). The former states the distance into the sliding window where the identical block starts, and the latter describes the length for which the block is the same [Feldspar, 1997].

An example taken from Feldspar [1997], below it is shown how a string is encoded in LZ77 with D being the amount of characters back where the repetition starts, and L being the number of characters from that part that is repeated.

Original string:

```
Blah blah blah blah blah!
```

Encoded string:

```
Blah b[D=5, L=18]!
```

2.2.4 Huffman coding

Huffman coding is a general purpose and lossless method that attempts to find a minimum-redundancy code for a text that contains a minimised amount of symbols. As redundancy is eliminated, it results in a code that can be decoded back into the original text. This is done by first using the algorithm to produce a full binary tree (Huffman tree) with the to be coded text as input, which then outputs a tree containing all symbols that are present in the text based on the frequency in which they occur. Subsequently, the text is encoded based on this tree with more frequently occurring symbols being represented by less bits using binary code [Huffman, 1952].

On figure 2.1 an example of a Huffman tree is shown, with the frequencies enclosed in boxes and the characters underneath it. It is accompanied by table 2.1 in which all characters are shown with their frequencies and their binary encoding. Since the encoding of characters differs per Huffman tree, the tree needs to be available when data needs to be decoded. The tree uses up additional space and thus to be efficient, the encoded text should either be large or a general tree has to be created that is deemed suitable for to-be encoded strings.

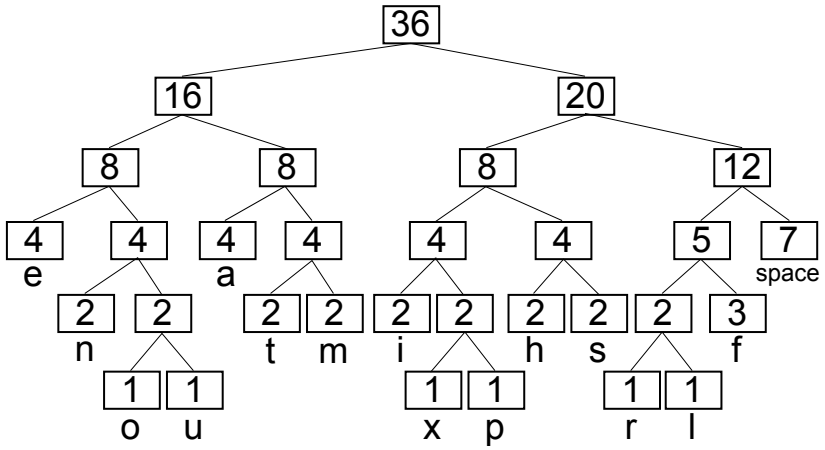

Figure 2.1: Visualisation of an example Huffman tree, created with the string "this is an example of a huffman tree" as input. Source: Dcoetzee [2007]

Char	Freq	Code	Char	Freq	Code	Char	Freq	Code
space	7	111	m	2	0111	p	1	10011
a	4	010	n	2	0010	r	1	11000
e	4	000	s	2	1011	u	1	00111
f	3	1101	t	2	0110	x	1	10010
h	2	1010	l	1	11001			
i	2	1000	o	1	00110			

Table 2.1: Table that accompanies figure 2.1. It contains all characters of the original string with their frequencies and the binary code with which they are mapped. Source: Dcoetzee [2007]

To do: create own Huffman tree + table

2.2.5 DEFLATE and zlib

Zlib is a free general purpose compression library that uses the Deflate method, which in turn is based on Huffman coding and LZ77 compression. The input data is split into a series of blocks. Each block is compressed separately by using LZ77. Subsequently, Huffman coding is used on the encoded blocks. For the latter part, zlib has two different implementations that are performed on individual blocks: either with creation Huffman trees with the algorithm which is then stored alongside the data, or by using default trees that are defined in DEFLATE which removes the need of storing the extra information [Feldspar, 1997].

2.2.6 Smaz

Even though it can be used on any kind of natural language text, Smaz is a compression library for the specific purpose of working well on small strings. General purpose compression techniques tend to have a larger overhead in order to be able to work well with dynamic input, which is why Smaz is potentially more suitable for this specific case [Sanfilippo, 2019a]. It works with a static codebook containing frequent-occurring English (fragments of) words and bigrams. This makes it likely to work best on English strings, however, this is subject to testing. The parts are encoded in binary, with the most frequent parts being in the smallest representation [Sanfilippo, 2019b]. Its general idea is thus similar to Huffman coding (see 2.2.4).

2.3 Compression of geometries, meshes, and graphs

2.3.1 Quantisation

Quantisation is a lossy compression technique that is used for signal and image processing, but can also be applied to other types of data, such as a list of coordinates as is exemplified by Draco. It works by mapping a dictionary containing all values that the data encompasses to a dictionary reduced in size, to which original values are joined many-to-one. This means for example that a value that occurs once will be mapped to a value that is close to it but occurs more frequently, losing some of the original information [Sayood, 2017]. Vector quantisation can be used to compress coordinates [Rossignac, 2001].

2.3.2 Delta encoding

Another used name for this concept is delta compression. It is for instance used for version control, the distribution of software patches, and other transmission of data over the web. The idea behind it is to only store or transmit the delta (difference) of the data to data that is already stored or received. The complete data can subsequently be reconstructed by adding the delta to the existing data that it is connected to [Suel, 2019]. The technique can be applied within a complete set of data as well as is is for example done in Deering [1995], where vertices are stored in an array as a vector of the delta difference with the previous vertex. Both transmitting deltas and storing deltas in a complete dataset are relevant for this thesis in the context of streaming and the compression of geometry as well as feature attributes.

2.3.3 Edgebreaker

Edgebreaker is one of the techniques used by Draco to compress 3D triangle meshes. A mesh that consists of triangles can be represented by the vertex data, which is made up of the coordinates of all vertices that the mesh encompasses, and the connectivity, which defines the relation between the triangles (or faces) and the vertices that belong to it. This is how Wavefront OBJ files are structured as well, which is followed by CityJSON. The Edgebreaker algorithm compresses the connectivity data of meshes [Rossignac et al., 2003].

Compression

To encode this information, first a spiraling triangle spanning tree is created for the mesh [Rossignac et al., 2003]. Analogised with peeling an orange, it is done by cutting around the peel in a spiraling way, after which it can be laid out flat. This is visualised in figure 2.2. With a mesh this would result into a curled string of triangles. Ultimately, the dual graph of this string represents the spanning tree [Taubin et al., 1998].

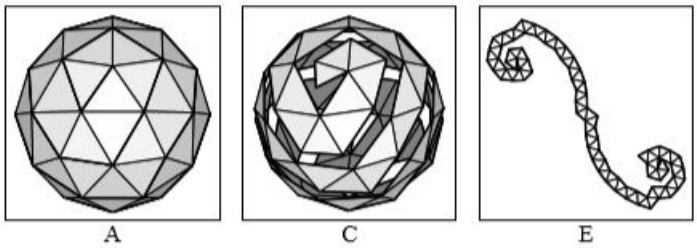

Figure 2.2: Visualisation of the creation of a triangle spanning tree, not showing the dual graph representation [Taubin et al., 1998]

Consequently, the algorithm traverses the triangles of the spanning tree, keeping track of all visited triangles and the vertices they are bounded by. Each triangle is assigned one character of the set `{C,L,E,R,S}`, which denotes one of the five cases (see figure 2.3) that are defined in the specification of the algorithm. C means that the shared vertex v, between the current triangle and the ones to its left and right, has not been visited at all. As for the other four cases, vertex v has been visited and the triangle left or right (L or R), both of them (E) or neither of them (S) has been traversed [Rossignac et al., 2003; Rossignac, 2001].

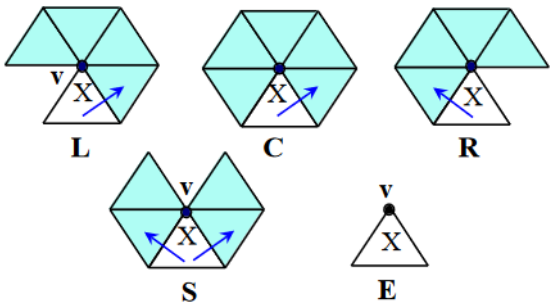

Figure 2.3: The five cases defined for the Edgebreaker method [Rossignac et al., 2003]. X is the current triangle in the iteration, v the vertex that is shared between X and the triangles to its left and right, and blue triangles are unvisited.

During the traversion of the triangles two tables (M and U) are created to respectively store the visited vertices and triangles, and two others (V and O) are used to keep references to vertices and their opposite corners.

Optionally, a prediction method for CLERS symbols of succeeding unencoded triangles can be used, which is based on the vertex valences [Google, 2020c]. The valency (or degree) of a vertex is the amount of edges that is emanating from it. The addition of this method can decrease the file size in similar fashion to parallelogram prediction (see Section 2.3.5), as symbols would not have to be stored explicitly.

Decompression

Decompressing the mesh the V (vertice references) and O (opposite corners references) tables will be reconstituted, and after that the table containing the vertex positions. The CLERS string is iteratively decoded to create a triangulated polygon that approaches C in figure 2.2, adding one triangle at a time. An S triangle will invoke the start of the decompression of one string of triangles (as there are multiple ones because of recursive compression). Encountering an E means that the end of the string is reached in the current recursion iteration. Depending on the other symbols that are encountered, the bounding edges shown in C in figure 2.2 are zipped up by matching pairs of edges. In this way, ultimately shape A in figure figure 2.2 is reconstituted. After the decompression of the connectivity information, the positions of the vertices are decompressed [Rossignac et al., 2003].

2.3.4 Sequential connectivity compression

An alternative method to edgebreaker that Draco can use is their own sequential connectivity method. It either uses delta encoding (storing the difference of vertex coordinates to the ones of the previous vertex [Deering, 1995] to store the IDs of vertices after which they are compressed by entropy encoding (of which Huffman coding is an example— 2.2.4), or the IDs are simply stored using everytime the amount of bits that the largest vertex id takes up [Google, 2019a]. In contrast with edgebreaker, this method preserves the vertex order within triangles. On the other hand the resulting encoding will not be as small.

2.3.5 Parallelogram prediction

Parallelogram prediction compresses meshes by predicting the missing vertex of a triangle, knowing the two vertices from the edge shared with a previously decoded triangle [Touma and Gotsman, 1998; Google, 2018]. A basic prediction can be made with the so-called parallelogram rule, as defined by formula 2.1, with the variables referring to the vertices in figure 2.4.

$$r^p = v + w - u \tag{2.1}$$

However, this prediction can be improved by taking an estimated crease angle between the two triangles (the "crease" being the shared edge—the angles are depicted on figure 2.4 with the lines crossing edges (s, v) and (v, u)) into account, as opposed to assuming that they are co-planar. This angle is estimated based on at least one set of adjacent triangles, that thus has to be decoded already. If there are more adjacent triangles available, it is taken as the average of the crease angles between the two sets of triangles that are closest to parallel to the shared edge of the current triangle and the predicted one [Touma and Gotsman, 1998]. As example, $r_c a^p$ on figure 2.4 is the predicted vertex of the next triangle by taking the crease angles between triangles (s, v, t) and (s, v, w).

Ultimately, the mesh is compressed by storing the error of the predicted vertex to the actual vertex (as a vector) rather than the vertex itself. It is therefore a completely lossless technique.

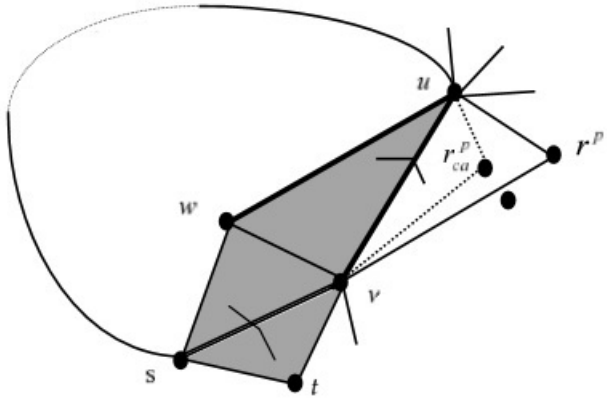

Figure 2.4: Illustrated how a new vertex can be predicted using parallelogram prediction. r^p is the vertex as predicted with the parallelogram rule, $r_c a^p$ is as predicted with the crease angle taken into account. Adapted from Touma and Gotsman [1998]

3 Related work

3.1 CityJSON

CityJSON [CityJSON, 2019] is an encoding for storing digital twins, containing geometries and attributes of real world features such as buildings and terrains. The file format has four main characteristics that can be exploited for compression purposes: the JSON encoding, geometries, attributes, and textures. As mentioned in Section 2.1, prior knowledge on the input data allows for more suitable compression techniques to be chosen. Furthermore, the explanation helps in understanding the difference with other file formats that are introduced in Sections 3.3.1 and 3.4.

The JSON format is easy for humans to read, while at the same time being relatively lightweight (in comparison to for example GML) [Introducing JSON, 2019]. In its specification several data types are defined: boolean values, numbers, strings, arrays (ordered lists of elements), and objects (containing key-value pairs). These data types can be combined and nested [Ledoux et al., 2019]. In Listing 3.2 it is shown how this looks like with example snippet of a CityJSON dataset. However, the readability comes at the cost of verbosity. This is exemplified by the binary JSON-like format CBOR which is introduced in Section 3.6.

A CityJSON file is a JSON object contains 4 mandatory keys:

1. `"type"` (which is always of value `"CityJSON"`)
2. `"version"`
3. `"CityObjects"` (a JSON object containing objects that represent geographical features)
4. `"vertices"` (an array containing the coordinates of all vertices)

In addition, there are optional keys for extensions, metadata, coordinate transform, appearance, and geometry-templates. Metadata can contain for example the geographical extent of the data, and coordinate transform gives the option to define a coordinate offset, which enables compression through the conversion of coordinates to integers and moving the origin to (0, 0, 0) [Ledoux et al., 2019; CityJSON, 2019]. The basic structure of a CityJSON file is shown in listing 3.1, including `"metadata"` and `"transform"`.

A City Object can have several types as derived from the CityGML data model, such as Building, Bridge, or WaterBody. It always has a "geometry" key which is an array containing more than 0 Geometry Objects, which can represent several LoDs. A Geometry Object in turn can be of several types of 3D geometric primitives. The vertex coordinates of a 3D model are stored in one array as shown in the above list, but the connectivity information is stored in Geometry Objects. Here, references to vertices are stored. This is inspired on the Wavefront OBJ format, and these parts are depicted in the green in listings 3.2, and 3.3. City Objects may be structured with a parent-child hierarchy [Ledoux et al., 2019; CityJSON, 2019].

Optionally, a City Object can have an "attributes" object as member. Here, all attributes can be stored as defined in CityGML's data model. It is allowed to add custom attributes as well if desired [Ledoux et al., 2019; CityJSON, 2019].

Lastly, it can contain materials and textures belonging to objects. These are respectively stored following the X3D specifications and with additional COLLADA files [Ledoux et al., 2019; CityJSON, 2019]. This is however out of the thesis scope (see 1.3).

To do: explain CityJSON in more detail. Important: "transform" is said to be quantisation in the CityJSON specs, but I think it means something else than quantisation in Section 2.3.1

Listing 3.1: Snippet of CityJSON structure, with mandatory members in orange

```
{
    "type": "CityJSON",
    "version": "1.0",
    "CityObjects": {..},
    "vertices": {..},
    "metadata": {
        "geographicalExtent": [
            78248.66,
            457604.591,
            2.463,
            79036.024,
            458276.439,
            37.481
            ],
        "referenceSystem": "urn:ogc:def:crs:EPSG::7415"
        },
    "transform": {
        "scale": [
            0.001,
            0.001,
            0.001
            ],
        "translate": [
            84616.468,
            447422.999,
            0.47
    }
}
```

Listing 3.2: Snippet of basic structure of CityObjects. Attributes in orange, geometries in green

```
{
    "type": "CityJSON",
    "version": "1.0",
     "CityObjects": {
        "b0a8da4cc-2d2a-11e6-9a38": {
            "type": "BuildingPart",
            "attributes": {
            "creationdate": "2014-07-09",
                "terminationdate": "",
                "function": "garage"
                },
                "geometry": [
                {
                    "type": "Solid",
                    "boundaries": [
                        [
                            [
                                [
                                    [10],
                                    [11],
                                    [12],
                                    [13]
                                ]
                            ]
                        ]]
                    },
                    "lod": 2,
                },
            "parents": ["b0a8da4cc-2d2a-11e6-9a39
                "]
            },
        "b1105d28c-00ba-11e6-b420": {..},
        "b1126a169-00ba-11e6-b420" {..}
    }
}
```

Listing 3.3: Snippet of vertex array of CityJSON, highlighted in green

```
{
    "type": "CityJSON",
    "version": "1.0",
    "metadata": {..},
    "CityObjects": {..},
     "vertices": [
          85012.343,
          447455.577,
          -0.27
          ],
          [
          85010.804,
          447448.808,
          -0.27
          ],
          [
          85013.832,
          447447.447,
          -0.27
          ],
          ...
}
```

3.2 Draco

Draco, an open-source library by Google for compression and decompression of geometric meshes [Google, 2019b], can be used to compress the geometrical part of the CityJSON structure. Generally, it works with the Edgebreaker algorithm (which is explained in Subsection 2.3.3), their own sequential connectivity method (Subsection 2.3.4), parallelogram prediction (Subsection 2.3.5), and the quantisation of coordinates (Subsection 2.3.1). It can compress both 3D geometrical meshes and point clouds, but only the former is relevant for this application. On average, it compresses meshes (that are in Wavefront OBJ format) to about 2 percent of their original file size [Google, 2019c]. It has encoders and decoders written in C++ and JavaScript (and exclusively a decoder in WebAssembly), enabling it to be suitable for both offline (in this case preparation of compressed CityJSON files) and online (decoding the compressed geometries in the browser) purposes.

Not just the savings on file storage itself are beneficial, but since files are transferred over the web for browser use, time is saved as the data is in a slimmer format. The network speed can form a

bottleneck, and it is exemplified by Cesium that using Draco can potentially improve the rendering speed as it is decoded asynchronously, meaning that parts are already rendered while the complete dataset is still being downloaded [Cesium, 2018].

There are three main configurations that can be set when encoding a mesh. First, there is the compression level parameter which goes from 0 to 10. Based on this parameter, different compression features that the library encompasses are used, with 10 giving the most compressed result but at the same time having the worst decompression time. Second, a parameter that specifies the amount of bits to which the input values should be quantised can be chosen (which would make the compression lossy if used—see Subsection 2.3.1), and lastly, there is the option to retain the separation between objects Google [2019c].

Draco compressed meshes are stored in their native .drc binary format. The format generally consists of four parts: a header, metadata (object id for example), connectivity data, and attributes, as shown on figure 3.1.

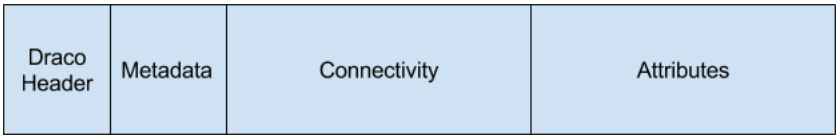

Figure 3.1: The general structure of the Draco file format (.drc) [Galligan, 2019]

3.3 Cesium 3D Tiles

Despite this part being out of scope for this thesis (as stated in the introduction), it is still important to notice its existence as it can make a contribution to the efficient streaming of 3D geoinformation. Besides that, it is implemented by the b3dm format (see Subsection 3.3.1), which is an alternative to CityJSON for web visualisation.

Created for the sharing, visualisation, and analysis of 3D geoinformation, the 3D Tiles specification [Analytical Graphics, Inc., 2018] is a JSON object in which the data is separated into tiles. Every tile is described by a bounding volume and the tiles are organised into a tree, which could be a quadtree, octree, k-d tree, or grid. Tiles can have children, creating a hierarchical structure with which different LoD can be represented. The tile parent would contain the 3D model in a low LoD, and every child would contain either more details that can be rendered when needed, or again a full 3D model but with a higher LoD [Cesium, 2020].

Its implementation can thus benefit the streaming of massive datasets as it allows the loading of smaller parts (tiles) as they are needed, e.g. when they are zoomed into, and in the LoD that they are wanted. In section 3.3.1 an implementation of 3D Tiles is explained, giving more details about the structure.

3.3.1 Batched 3D Model

b3dm is a binary file format that can store heterogeneous 3D models and incorporates the 3D Tiles specification [Analytical Graphics, Inc., 2019]. It roughly consists of three parts: a feature table, batch table, and Graphics Library Transmission Format (glTF) geometry. In the specification a feature is a 3D model and the feature table can thus be seen as a collection of tiles. Every feature may have a position property indicating the coordinates of its centre.

In the batch table the properties of the contents of features are contained. These properties are stored in JSON format, but in a different fashion than CityJSON's CityObjects. It is an array value

with the name of the property as key, with the array containing values for the ordered objects of the feature. This means that if there is an n amount of objects, the array for a property will contain n elements. An example is as follows:

```
"height" : [10.0, 20.0, 15.0]
```

Alternatively, the properties can be stored in a binary body to which a reference is placed within the aforementioned JSON structure. Since this way of storing attributes is inefficient when there are properties that only apply to a subset of objects (for example "amount of leafs" for trees, which buildings would not have), it is possible to define object types which have their own set of property names.

Lastly, the geometries of the objects from all features are stored as glTF, which allows for the use of Draco compression as well since it can be embedded therein.

The format therefore is similar to the aimed end-result of this thesis, with the additional benefit of having the option to structure the data into tiles. On the contrary, it lacks the benefits of the CityGML data structure, the tools that exist for CityJSON, and (further) compression of attributes [Analytical Graphics, Inc., 2019].

To do: explain glTF—add that glTF can not store complex geometry. Add example of file structure.

3.4 Esri I3S

I3S [Esri, 2019b] is an open file format by Esri. The scene layer that it encodes in turn contains large volume heterogeneous geographic data (e.g. 3D objects, point clouds, and buildings). It is formatted in JSON and specifically designed for the efficient streaming of massive 3D datasets. Similarly to CityJSON, Scene Layers encompass the geometry, texture, and attributes of features. Features are grouped into nodes with the nodes forming a regular or irregular tree by their bounding sphere (or oriented bounding box), akin to respectively a quadtree or an R-tree. Parent nodes in a tree contain the features with a lower LoD. Because of this structure, specific nodes can be found faster and loaded more efficiently as well as simpler representations of geometries can be shown when they are looked at from afar [Esri, 2019a,b].

A node in itself does not contain complete features, but rather IDs which point to geometry buffers (encoded with Draco), textures, and attributes. The standard allows any geodetic coordinate reference system to be used, with the node having the same coordinate references system (CRS) as the vertices that it encompasses. The vertex position are stored with the centre of the node's bounding area as offset, allowing for compression and easier visualisation in for example three.js. LoD are not the same in I3S as in CityJSON—it uses thinning, clustering, and generalisation of 3D objects or meshes, as well as the downsampling of textures. The to-be visualised LoD (thus which leaf of the tree is chosen by the viewer) is determined by screen size, resolution, bandwith, and memory [Esri, 2019b,a].

To do: Add example of file structure.

3.5 OGC API

The OGC API [Open Geospatial Consortium, 2020] is a new specification that defines how to provide online access to geospatial data. This is done in a modular way, with several building blocks that can be used together as needed when creating a RESTful Web API for geospatial information.

It succeeds the older well-known OGC standards such as WMS and WFS. With CityJSON being an encoding for 3D features we focus on the OGC API Features block, which could be seen as the successor of WFS2 (and it was formerly named WFS3) [OGC API, 2020; Open Geospatial Consortium, 2020].

This block consists of multiple modules as well (on CRS, Contextual Query Language (CQL), and simple transactions) and has the possibility to be extended with more modules. In contrast with WFS, OGC API Features is created with other file encodings than XML in mind. This enables to make it work with JSON and HTML easily, which is encouraged but not mandatory as any encoding may be used [Open Geospatial Consortium, 2020, 2019a]. Because of this, it is possible to make CityJSON compliant with the specifications of the API. Ultimately, any kind of geospatial data consisting of features can be served following the guidelines of the OGC API, as long as a specification is created for it.

The API works through visiting specifically formatted links that indicate what kind of data is requested from a server. The base URL of the server (with path `"/"`) needs to provide links the to:

1. The API definition—the capabilities of the server
2. The Conformance declaration—URIs referring to the standards that the server implements
3. The Collections—the datasets that are available on the server

By visiting specifically formatted links, Collections can be queried as desired and are served in the requested encoding. Information on the Collections is found following the `/collections` path, with `/collections/{collectionId}` showing more detailed information on a specific Collection. `/collections/collectionId/items` will return the full dataset, and with `/collections/collectionId/item/itemId` one specific feature can be retrieved.

It is mandatory to provide a function to query the features of a Collection on a bounding box in for example the following way: `/collections/{collectionId}/?bbox=160.6,-55.95,-170,-25.89`. Similarly to WFS, it should be possible to include a `limit` parameter in the link to only receive a maximum amount of features. Combining this with an `offset` parameter, it is possible to stream the data. For instance `/collections/{collectionId}/item/{itemId}/limit=10&offset=0` would return the first 10 features of the requested dataset. By increasing the offset by 10 in every new request, the full dataset is traversed. Other specific queries could be formatted in a similar manner as well. These are the most important requirements, but not all are listed and there are also more recommended implementation details [Open Geospatial Consortium, 2019b].

Currently the use of the API has been mostly described in liaison with GeoJSON [GeoJSON], which is recommended, and the WGS84 CRS which is the default. A server would return a GeoJSONCollection or GeoJSONFeature when data is requested. It is however possible to use a different file encoding and a different CRS when the CRS module is used [Open Geospatial Consortium, 2019b]. Section 3.5.1 therefore explains how CityJSON could be used in accordance with the OGC API.

I might move this section to the discussion in the conclusion chapter. Still an introduction on the OGC API is needed as it initiated the creation of an initial CityJSON streaming structure (explained in next section)

3.5.1 Streaming CityJSON

GeoJSON [GeoJSON] does not support 3D data, and for OGC API to work with 3D data a new specification would have to be written. A CityJSON object can already be seen as a collection of features. It can therefore be argued that its original structure is already OGC API compliant since the specification allows for flexibility in data structures that servers return. However, the

streaming of datasets is part of the OGC API specification since it is possible to add `limit` and `offset` parameters to a link.

For this purpose a draft [Ledoux, 2020] has been made to make CityJSON compliant to the OGC API as it is implemented now (with GeoJSON), and to enable streaming. Because CityJSON has the list of vertices at the end of the file rather than per City Object, it is not possible to stream it in the fashion that it is done with GeoJSON. With GeoJSON, the receiver of the stream is fed with GeoJSON Features or FeatureCollections in line-delimited JSON. Once received, it can be used immediately as it already includes the complete geometry.

In order to enable CityJSON to stream in a similar way, the structure of listing 3.4 is proposed by Ledoux [2020]. The `"type"` is now a CityJSONFeature rather than a CityJSON object. `"CityObjects"` contains a parent and its children, with the `"id"` of the CityJSONFeature being the one of the parent. All their vertices can now be accessed immediately, without needing to have the whole file. A CityJSONFeature can thus be loaded in the same manner as a CityJSON, thus avoiding the need to adapt the software that will use it. In Section 5.6.4 I investigate how compression could be implemented for streaming.

Listing 3.4: Snippet of OGC API compliant CityJSON structure. Source: Ledoux [2020]

```
{
  "type": "CityJSONFeature",
  "id": "myid", //-- to clearly identify which of the CityObjects is the "main" one
  "CityObjects": {},
  "vertices": [],
  "appearance": {},
}
```

3.6 CBOR

CBOR is a binary data format that is based on the data model of JSON. Since JSON can directly be encoded into CBOR it is interesting to investigate using it to compress CityJSON [CBOR, 2013]. It has two clear purposes, which are storing binary strings into a JSON-like format and compression of data [CBOR, 2019].

Occasionally it is beneficial to store binary data in JSON format, such as encryption keys or graphics. JSON natively does not allow for this and requires such data to be encoded in Base64 format. This is a family of binary-to-text encoding schemes that is utilised when binary data needs to be stored, while working with media that are created to handle ASCII rather than binary data [CBOR, 2019]. The downside of this is that a string in Base64 encoding can be 33% larger than its binary form. This is related the second purpose of CBOR, which is the compression of JSON data: as mentioned in Section 2.2.1, a binary representation of text is more concise and quicker to process, and with CBOR Base64 becomes redundant.

Like JSON, it follows a key-value structure. Roughly explained, every data item starts with an initial byte the contains information about the major type of the item and supplementary information. The major type can be one of the following [CBOR, 2013]:

- Unsigned integer
- Negative integer
- Byte string
- Text string

- Array
- Map (JSON-style object)
- Optional semantic tagging of other types (such as date/time string, URI, base64)
- Floating-point number

When reading the data, knowing the type and length of a value increases the performance as mentioned in section 2.2.1.

4 Methodology

The network speed is always a bottleneck when transmitting complete CityJSON datasets over the web and doing something with the data, as the receiver can only do something with it once the data has arrived. The assumption we thus make is that compressed datasets will have a better performance when the time that is gained in network transmission speed is larger than the time that is lost decoding the data after arrival. This chapter lays out the plan to benchmark the performance of several types of compressed CityJSON compared to uncompressed CityJSON, to find the advantages and disadvantage of techniques and give recommendations on what the best compression types would be given specific scenarios.

Shown in Figure 4.1 is the structure of the methodology, divided into four general parts:

- The implementation of compression techniques for CityJSON
- The preparation of datasets to test compression techniques with
- Preparation of a testing platform that can work with both uncompressed and compressed CityJSON files
- Testing and benchmarking the performance of different file types on the performance indicators of Table 4.1, which were introduced in Section 1.2

The first three are finished prior to the benchmarking, in order to avoid having to redo work for part 2 when something had been changed in part 1. The tasks in part 1 are grouped together because there is no necessary order and are implemented simultaneously. All four are explained further in the next sections.

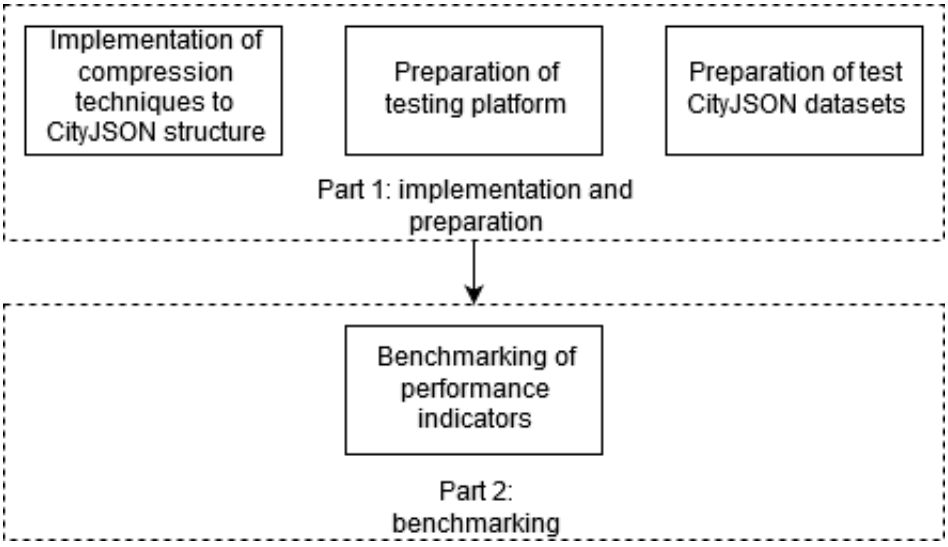

Figure 4.1: The general research workflow

Compression type performance indicators
Visualisation time
Querying time
Spatial analysis time
Editing time
File size compression
Lossiness

Table 4.1: The six performance indicators on which variants of compressed CityJSON are assessed

4.1 Implementation of compression techniques

In Chapter 2 a multitude of compression techniques has been introduced. Different combinations of these techniques are applied to the CityJSON structure and be compared on the six defined performance indicators (see Table 4.1), with the performance of uncompressed CityJSON as the baseline. A possible outcome is that certain combinations of techniques are more suitable for specific purposes, making it difficult to choose one that is the best. Chapter 6 shows which compression types For this reason, there is a depthly discussion about it in chapter 6.

Table 4.2 shows which combinations of compression techniques I implement on CityJSON and benchmark on time performance. The table has a divide between compression methods where the geometries are kept as is, and where they have been replaced by Draco-compressed geometry. All other names besides "original" and "draco" indicate either the inclusion of a binary format or a compression technique.

The only compression technique that has not been explained in chapter 2 is the so-called replace technique, which I have created. It means that all keys and values of "CityObjects" from the JSON structure are copied to a separate array. The elements of the array are sorted by frequency, and the keys and values in the JSON are replaced with a reference to the array. This has three supposed advantages:

1. The JSON structure is retained, of which parts could be reconstituted as wanted. It would for instance be possible to only decode the first City Object rather than all of them. When using a compression technique directly on the JSON structure, all data has to be decoded prior to use.
2. The array allows for greater compression as the keys and values can be compressed at once with for example zlib, as opposed to compressing all of them individually.
3. Redundancy is removed, as keys and values that appear more than once are only stored as one array element.

The array can be stored as a member of the CityJSON object and thus optionally be included in a CBOR representation. An example of how a dataset compressed in this way looks like is found in Subsection 5.4.3.

Ultimately there are 20 different types of CityJSON tested, of which 19 are compressed variants and 1 is original CityJSON. However, "original CityJSON" is defined as including the "transform" member (as stated in Section 1.2, with "transform" being explained in Section 3.1), which is done for the uniformity of datasets and because Draco needs coordinates to be stored in this manner before compression. It is however already a form of compression, but included in the CityJSON specification [CityJSON, 2019].

To do: explain somewhere why Draco needs this, and refer to that Section here.

	Original geometry	**Draco geometry**
1	original	draco
2	original-zlib	draco-zlib
3	original-cbor	draco-cbor
4	original-cbor-zlib	draco-cbor-zlib
5	original-cbor-smaz	draco-cbor-smaz
6	original-replace	draco-replace
7	original-replace-zlib	draco-replace-zlib
8	original-cbor-replace	draco-cbor-replace
9	original-cbor-replace-zlib	draco-cbor-replace-zlib
10	original-cbor-replace-huffman	draco-cbor-replace-huffman

Table 4.2: Combinations of compression methods, separated by type of geometry.

They tested CityJSON types are divided into two categories: having geometry compressed by Draco, or geometry as originally implemented in CityJSON. So there are actually 10 pairs, within which the difference is the use of Draco. Table 4.2 shows all CityJSON implementations, and in the following enumeration a brief description is given for all 10 combinations:

1. The dataset is in the style of regular CityJSON, but needs to include "transform". "original" is the baseline to which all other combinations are compared. As for "draco", the Draco Binary Large OBject (blob) is placed underneath the JSON structure, with a delimiter inbetween. It can not be added as a member of the CityJSON object as JSON does not allow binary values. This means that the result is a binary file.
2. The same as 1, but with the complete file compressed with zlib (see Subsection 2.2.5).
3. Similar to 1, but with the CityJSON object parsed into CBOR (see Section 3.6). In this case, the Draco blob is added as a member of the CityJSON object as it does allow for binary values. This would make it easier and quicker to read as well, since the file does not need to be split by a delimiter.
4. Encoded in CBOR like 3, but all keys and string values of the CityJSON object are compressed by smaz. This only works in combination with cbor, because strings are binary-encoded by smaz.
5. Same as 3, but with the resulting CBOR (Section 3.6) compressed with zlib (Subsection 2.2.5).
6. Similar to 1, but with all keys and values copied to an array and replaced by references.
7. Same as 6, but with the array compressed with zlib.
8. Same as 6, but with the CityJSON object parsed into CBOR.
9. Same as 8, but with the array compresed with zlib.
10. Same as 8, but with the array compressed with Huffman coding (see Subsection 2.2.4).

Decompression functions have to be written as well, both to be able to assess the loss of information due to the compression process and to allow for the data to be decompressed for use in the web application. Loss that is expected to see mainly concerns the precision (or even accuracy) of coordinates, the ordering of vertices, and the ordering of City Objects. The former is a large problem as the thesis concerns geographical data that can be used for analysis, as opposed to visualisation only. It can happen when Draco is not used in the correct way (see Section 3.2). The second one can happen with Draco as well, but since its geometries are triangulated it only means that vertices of triangles are ordered differently which is not problematic. Lastly, a different ordering of City

Objects is acceptable because all information is retained, but it should still be noted as it is possible that a dataset is ordered with a purpose.

Encoding JSON into CBOR is likely to give a better performance than other compression methods (such as directly compressing the datasets with zlib), as explained by UBJSON [2020]. While zlib should decrease the file sizes further than CBOR, the processing—encoding and decoding—of zlib-compressed data is less efficient. In addition, a binary JSON-like format enables a person to still inspect human readable parts (such as strings), which can help with debugging. Another option however is to zlib a CBOR file, which can be done if storage space is an important issue.

A possibility is to create an original binary format for CityJSON. However, since JSON can already be directly encoded into CBOR, this is not necessary as it would overcomplicate it. It could be done in a similar way as b3dm (see Section 3.3.1) by retaining the JSON structure and having a separate part for binary values. However, in this way only values are stored in binary and not keys, and when creating a dataset it needs to be chosen which values are stored in binary.

4.2 Performance of (compressed) datasets

The performance of the different compression method combinations are assessed following the six main performance indicators from Table 4.1. For each of these, specific data operations are defined that are tested, with original CityJSON datasets as the baseline to which the performance of compressed CityJSON files is compared. Below follows an explanation on all operations, and an overview can be seen on figure 4.2.

The file size performance is defined as the compression factor. This is the size of a compressed dataset divided by the size of the uncompressed one. For all other operations, the performance is defined as the time that it takes to complete the operation with the compressed dataset divided by the time that it takes to finish the same operation with the original dataset.

1. Querying. Querying one City Object of a dataset (by its ID) and querying all City Objects (by an attribute field that all City Objects have in common). The query returns the matching City Objects including the vertices that their Geometry Objects refer to.

2. Visualisation. The rendering of all City Objects of a dataset.

3. Spatial analysis. Buffering one City Object and buffering all City Objects by 1 metre in 2D, so their geometries are first projected to 2D.

4. Editing of attributes. Adding "1" at the end of the ID of one City Object, and doing the same thing for all City Objects. If the file had to be decompressed before the operation, it has to be compressed again.

5. Editing of geometry. Raising the height of all vertices of one City Object by 1 metre, and doing the same thing for all objects. If the file had to be decompressed before the operation, it has to be compressed again.

These are performed with all datasets, which are compressed as described in Section 4.1. All combinations of operations, datasets, and compressions are thus assessed. The assessment is done by benchmarking the time that is needed to complete an operation, taken as the average over ten attempts to eliminate abnormalities.

Being secondary indicators, lossiness and possibility of asynchronous loading are described separately. The former can involve loss of object order, loss of vertex order, and triangulation of a file that was originally non-triangulated

Ultimately, by putting all results in a table, the best (combination of) methods can be found for every separate criterion. On each of the indicators, the compression method is evaluated and ranked. An overview of the benchmarking workflow can be found on figure 4.2.

4.3 Testing platform and benchmarking

To test the performance of using CityJSON and its different compressed variants on the web, a web application has to be set up that enables this. While two options already exist (Boersma [2019] and CityJSON [2020]), it is better to create one that is of a simple structure so that it can easily be adapted for experimental functionalities for the research.

Therefore, I have set up a Flask application which acts like a server, which integrates functionalities of Draco [Google, 2019c], cjio [3D geoinformation research group at TU Delft, 2019], self-created functions, and harp.gl [HERE, 2019] through JavaScript. An explanation of the mentioned tools is given in chapter C. The platform enables to do the operations of Figure 4.2 on all (compressed) CityJSON variants by visiting corresponding Flask routes in a browser. The operations have short names as shown in Table 4.3 to easily identify them in the results.

Every dataset (8 of them, having different characteristics—see Section 4.4) is compressed by every CityJSON implementation type (20) and then benchmarked on the time that it takes to complete every individual operation (9) on them. This means that there are 1440 planned resulting time benchmarks. Every time benchmark is performed multiple times in order to retrieve more accurate results, as there are some uncertainties in the timing meaning that every iteration will have a slightly different time value. The time values are in milliseconds and outliers within the 10 iterations are removed based on having a z-score of 2. The z-score is the amount of standard deviations that a measurement differs from the average of the set.

The benchmarks are performed through a web testing library that visits the specific Flask route for a combination of dataset, operation, and compression type in a browser with a throttled network speed to simulate practical conditions.

Visualisation	visualise
Querying	queryone queryall
Spatial analysis	bufferone bufferall
Editing (attributes)	editattrone editattrall
Editing (geom)	editgeomone editgeomall

Table 4.3: Names for all operations in the results.

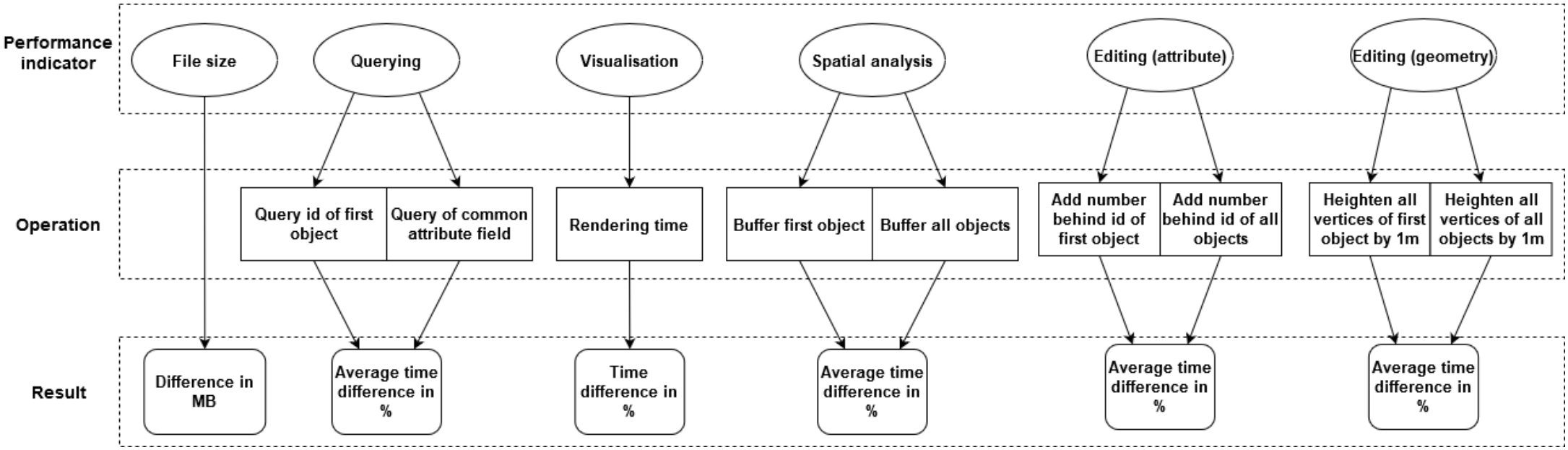

Figure 4.2: The workflow of the benchmarking. This is done for every compression method and all datasets.

4.4 Dataset preparation

Since different dataset characteristics can influence the compression factor and benchmarking performance of compression types, it would be good to use a varied selection of CityJSON datasets to test with. For example, a compression method that makes use of the repetition of data will work better when there are more repeated elements in the dataset that can be taken advantage of. The file size of the to be compressed dataset can be an indication of this, as larger files tend to have more repeated parts. For example the keys `"type"` and `"boundaries"` are there for every City Object, and possibly repeated attributes. For this reason it is expected that larger datasets will generally have a lower file size multiplier.

The LOD is interesting because it indicates a difference in the geometrical complexity of the features. It is possible that there is a difference to be seen in the impact of Draco compression depending on it. The variety of feature types is another characteristic because similarly to LOD, terrains can potentially have more complex geometries than buildings. As for attributes, the amount of them per feature could impact the performance of attribute compression, as it would be expected that more attributes are percentually compressed further. Lastly, the language they are in can potentially make a difference when the smaz compression technique is used.

The file size will however be the most important about datasets, since the bottleneck is assumed to be the network speed when using large datasets on the web. If a dataset is already small, for example 1 MB, compressing it to 0.7 MB would only have the file be downloaded 0.06 seconds faster on a 5 MB/s (or 40 Mbit/s) internet connection (which is the average in The Netherlands Ċompressing a 100 MB file to 70 MB on the other hand will improve the transmission speed by 6 seconds.

An overview of important dataset characteristics can be seen below, and the chosen datasets can be found in Section 5.5.

- File size
- Repetition of data
- LOD
- Variety of feature types
- Size of attributes
- Attribute language

4.5 Overview

Concludingly, the following results are presented in chapter 6:

1. The resulting compression factors of individual datasets as compressed by every compression type, and the average compression factor per compression type.
2. The encoding time of individual datasets with every compression type, and the average encoding time per compression type.
3. Time performance benchmarks on all compression type, dataset, and operation combinations. As conclusion, a plot is made showing the average performance of a compression type per individual dataset, meaning that the operations are averaged. With this you can answer the question: "I have this dataset and want to do anything with it, which compression types are the most suitable?". Taking averages over all datasets is not done since they are not a representative selection, as opposed to the operations.

5 Implementation and experiments

This chapter firstly describes how the testing platform is set up that I use to assess the performance of different types of compressed CityJSON. Then, I explain the way in which the operations defined in Section 4.3 are performed on the testing platform, and subsequently give information on the performance benchmark,. After that comes the implementation of the compression types of Section 4.1 and a description of the datasets on which these are tested. Lastly I present the results of experiments that do not directly belong in the benchmarking chapter (Chapter 6) but analyse the lossiness of compression types, helped with making implementation choices, or are relevant in other ways.

5.1 Flask server

The Flask server is the base of the testing platform introduced in Chapter 4. Flask is a lightweight Python web framework with which web applications can be built [Flask, 2020]. It enables combined use of Python and JavaScript, where the former handles the transmission of datasets (which are stored on the server) to the client, while the latter handles the processes that are performed in the client, such as the visualisation. In this way, CityJSON files (and compressed variants) can be used on the web, with operations performed on it on either the server or in the client.

The testing platform is set up in Representational state transfer (REST)ful [Fielding, 2000] style—with a clear separation between the client and the server, and having stateless communication between these two, which means that every request to the server needs to contain all necessary information to fulfill the request as the server does not remember previous interactions with the client. In Flask, routes (in the form of URLs) are defined in the server's code and coupled with a function that is executed when the specific route is visited, i.e. the client makes a request to the server. For the testing platform I have created routes and corresponding functions that enable to perform the operations laid out in Table 4.3. When such a function is finished, it returns an html template that is rendered in the client, a dataset (that is prepared on the server if applicable), and variables with information on the current operation so that the client knows how the process the data.

Uncompressed CityJSON files are always processed on the server (depending on the operation—for example queried) before they are transmitted to the client. This is because they can be directly manipulated using for example cjio functions, avoiding having to transmit the full dataset if it is not requested as such. Compressed files however can generally not be manipulated before decompressing the data. There are however exceptions to this, which is discussed in Subsection 5.6.2. But in general, since the presumed advantage of using compressed files on the web is the mitigation of the data transfer time from server to client, decompressing and performing an operation on a dataset on the server is unwanted because it would diminish this potential speed gain. Therefore, the server transmits the full compressed dataset to the client, which will proceed to decompress it and execute the specified operation.

This means that the functions with which datasets are processed are thus not the same, and that the difference in execution time of used algorithms influences the benchmarking results. While this may seem to make for unfair comparisons, it is actually not a problem since this simulates the situation in practice of using a specific file type - in reality, the advantage or disadvantage of processing the data either server-side or client-side would count as well.

5.1.1 Three.js

Three.js [Three.js, 2020a] is used for the visualisation of the datasets.

It is a higher level 3D library for JavaScript that helps visualising 3D geometries by providing functions that ultimately render graphics through the lower level WebGL. WebGL has the ability to visualise 3D graphics in the browser using hardware acceleration, allowing for greater performance compared to using the CPU only [WebGL, 2011]. In its coordinate sytem, geometries are being centered around the origin and their coordinates normalised, such that all fall within the range `[-1, 1]`. Objects can thus have negative coordinates and they are not georeferenced in Three.js [Three.js, 2020a].

The CityJSON viewer [Boersma, 2019] contains code that parses CityJSON geometries into a Three.js mesh. Using this the CityObjects can be visualised. Part of this code is used in the testing platform, to be used for data that has its geometries stored in CityJSON's native manner rather than as a Draco blob.

A Three.js mesh is always formed only of triangles and can contain one material [Three.js, 2020b]. Within a mesh object, the geometry can be stored in two manners. It can be stored in the mesh similarly to OBJ and CityJSON (see Section 3.1): as a combination of three-dimensional vectors as coordinates and triangular faces with index references to the vectors. Such an object is named a `Geometry` in the library and this is what the CityJSON viewer parses geometries of datasets into. The alternative is a `BufferGeometry` object [Three.js, 2020c], which is less user-friendly but processed more efficiently. It simply contains an array of coordinates as floats, which would represent a matrix. This is one similarly for the faces. The Draco decoder for JavaScript decodes a Draco blob into a BufferGeometry. This difference can have an impact on the benchmarking results.

To do: show examples of Geometry and BufferGeometry to make the difference clearer

5.1.2 cjio

cjio[3D geoinformation research group at TU Delft, 2019] stands for CityJSON input/output. It is a Python command-line interface that has several functions that can be used to work with CityJSON files. It can be used to help preparing CityJSON datasets for the testing part, but more importantly, it can facilitate the querying and analysis of the datasets by integrating it with Flask and possibly Python libraries such as Shapely [3D geoinformation research group at TU Delft, 2019]. I chose to host files on the server and process queries with cjio and other tools, but another option is to use a Database Management System (DBMS) such as 3DCityDB [Technische Universität München, 2020].

5.1.3 Draco

As explained in Section 3.2, Google's Draco library is an open-source library to compress and decompress 3D geometric meshes for the purpose of improved storage and transmission efficiency. It is used for the compression of the geometrical part of CityJSON's file structure[Google, 2019b].

The Draco library includes an encoder and decoder for JavaScript, which can conveniently be used in the testing platform. Three.js has a function which decodes Draco using the previously mentioned decoder and loads it as a mesh that can subsequently be visualised. However, as opposed to Draco's C++ decoder, the one in JavaScript lacks the functionality for separating sub-objects (as features or CityObjects are called in Draco)—it would create one large mesh out of the Draco geometry. This makes it impossible to query objects which is necessary for several of the operations (see Section 5.2) in this thesis.

Therefore, I have altered the inner workings of the JavaScript decoder. I have added a part that is supposed to enable the passing of one or more object IDs, making it only return these specified objects as a mesh. This is only possible after the Draco blob has been fully decoded, adding time to the process. This is done by iterating over all faces of the decoded Draco geometry, querying the sub-object ID from its first vertex from a metadata querying function of Draco, and checking if it equals one of the IDs that had been previously passed to the decoder.

Additionally, the three.js function does not allow passing a Draco blob directly but requires a link instead. This link is subsequently downloaded into a BytesArray after which it is decoded. For this reason, Draco files are downloaded separately by visiting a Flask route. The Draco parts are taken out of the compressed CityJSON files in order to not affect the transmission speed and thus the benchmarking results.

The encoder, which is used after editing a file, can be used unaltered. However, it is not possible to choose a compression level between 0 and 10 like the C++ encoder can, but instead a choice can only be made between using Edgebreaker (Subsection 2.3.3) and the sequential connectivity encoding method (Subsection 2.3.4).

5.1.4 Testing platform overview

The diagram of Figure 5.1 shows on overview of the testing platform. The user (who is simulated by Selenium for the benchmarking—see Section 5.3) sends a request through the client to the Flask server. Subsequently the server processes the request and data if needed (when original CityJSON is requested), and starts a timer for the benchmarking. It then returns data (the CityJSON file or a compressed variant) and information on the requested operation to the client. The client parses the data (uncompressing it if necessary) and performs the operation. Both the client and server are supported by several libraries. Once the operation is finished, it will emit a message with the end time of the benchmark to SocketIO [Socket.io, 2020], which allows communication between JavaScript and Python. Socket.io in turn sends the message to the Flask server. The latter calculates the time that has lapsed and stores this, and a report of the benchmark is created when requested by the user.

5.2 Operations

The way in which operations are implemented influences the time benchmarking results. Additionally, they can be different for varying data types, and understanding what is happening under the hood helps with reasoning why the results of compression types turned out as they did.

In every case, once the client makes a request to the server, the server stores the starting time of the process. It then processes uncompressed CityJSON if needed and stores the result as a file. Flask will then render a HTML, while submitting information to the client about the process—the name of the requested dataset, the compression type, and the operation name. The information is used to enable the client to parse the correct URL with which the needed file can be downloaded.

5.2.1 Visualisation

As the file that needs to be visualised is downloaded, it is decoded as described in Section 5.4 if necessary. If it contains original CityJSON Geometry Objects, these are converted into a three.js mesh using code based on the CityJSON viewer. Otherwise, the JavaScript Draco decoder is used to convert the Draco geometry to a three.js mesh. In that case also the object IDs to which every vertex belongs has to be queried using `MetadataQuerier()` of the Draco decoder. This is necessary

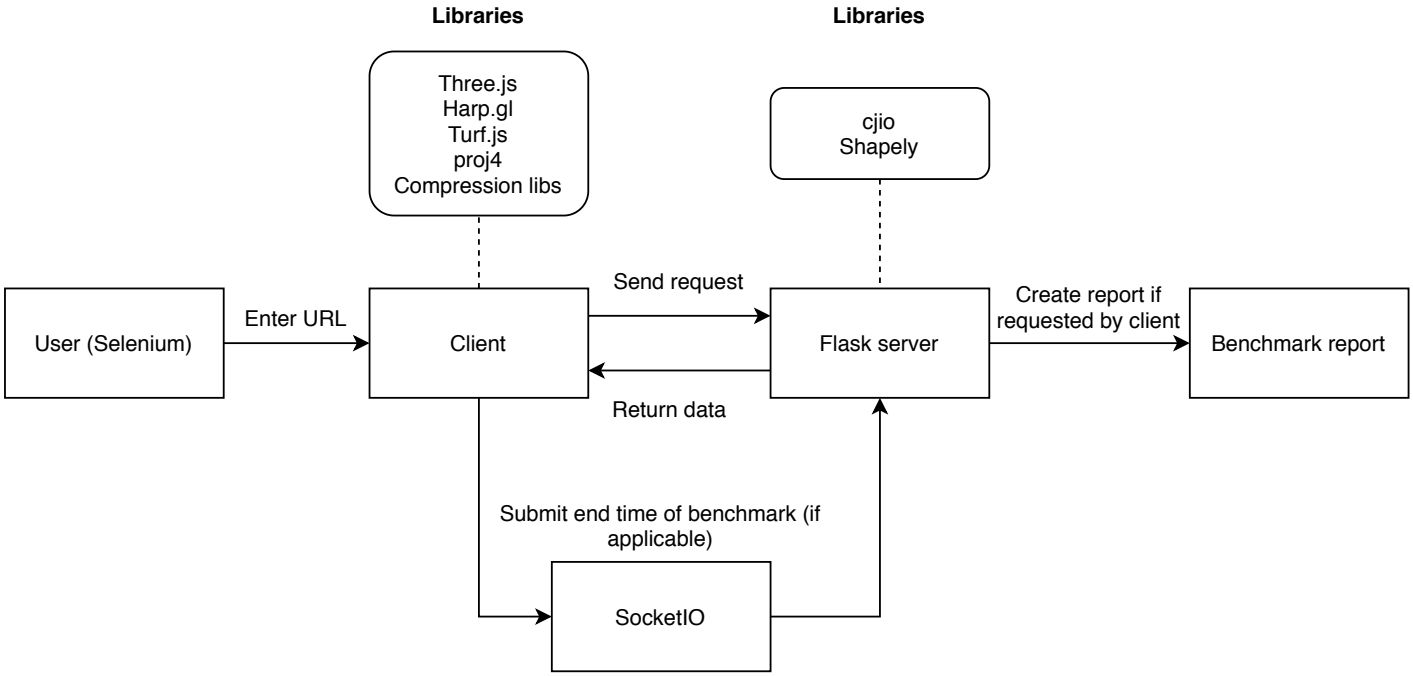

Figure 5.1: Diagram showing an overview of the structure of the testing platform.

because the Draco decoder creates one large mesh out of all objects, making it impossible to link attributes to separate objects. When visualising a 3D city model it is important that this can be done, and it makes the comparison to the visualisation of Geometry Objects fair.

When this is finished, the three.js mesh is added to the harp.gl map and visualised. The client then returns the current time to the server, which will calculate the time that has lapsed and add it to the benchmarking report.

5.2.2 Querying

Querying one object

The first object of the dataset is queried on its ID. For uncompressed CityJSON this is done on the server. The queried City Object is taken by accessing the value of the ID as key in the `"CityObjects"` member of the CityJSON object. It is placed in a new CityJSON object, and its geometry is then processed by cjio's "subset.process_geometry" function, retaining only the vertices that belong to the object and updating the vertex indices in the Geometry Object to start with 0. The bounding box is updated, and the CityJSON object is then stored in a file and can be downloaded by the client. As the download is completed, the client will submit the end time to Flask.

As for compressed CityJSON, the complete file needs to be downloaded by the client first before it can be compressed. For some compression types this is always necessary, but others have the possibility to directly be queried. That could be done on the server and would decrease the size of the transmitted data. I have not implemented this.

Instead, the file is decoded and queried in the client. For compression types with original geometry, this is done in the same way as explained before. Since cjio does not exist for JavaScript I have translated the "subset.process_geometry" function to it.

With Draco geometries, the difference is that the vertices corresponding to the object ID need to be found with `"MetadataQuerier()` of the Draco decoder. As explained in Section 5.1.3 however this does not go well. Still, in any case all vertices would have to be traversed and this is being simulated.

Querying all objects

The functions do not know beforehand that all objects are queried, in order to simulate the querying process properly.

Uncompressed, all City Objects are traversed and checked on containing the attribute field. If so, its ID is appended to an array. This array is passed to cjio's "get_subset_ids" function which will update the CityJSON object's City Objects, `"transform"`, geometry, and bounding box. The resulting CityJSON object is stored as a file and can be downloaded by the client.

With compression types with original geometries, this is done in the same way. When using Draco, again all vertices of the Draco geometry have to be traversed.

5.2.3 Spatial analysis

Buffering one feature

Uncompressed, the server will get the first City Object from the CityJSON object. The vertices have to be transformed back to original with cjio (thus removing `"transform"`), which means that they become floating points again and are not offset from the origin. This is necessary because the buffer result would be different if the coordinates are not represented in the same units (e.g. metres). The geometry of the object is converted to 2D Shapely polygons after which the library's buffering function can be used. The resulting buffer polygon is parsed into a CityJSON Geometry Object and replaces the geometry of the City Object. Accordingly the array of vertices is replaced, and thus contains the vertices of the buffer. The result is a CityJSON object containing the buffered City Object which could then be visualised, but that is not include in the time benchmark.

With compressed datasets it is done in similar fashion in the client. The difference is that the buffering is done by Turf.js rather than Shapely. This means that the coordinates have to be transfored to WGS84 first (using proj4), since Turf.js does not work with other coordinate systems. Another difference is that Draco geometries are not parsed into a Geometry Object but rather in the same way as Draco geometries are structured—a Three.js BufferGeometry (see 5.1.1).

Buffering all features

This is exactly the same as Subsection 5.2.3, but all features are iterated and the results are parsed into one CityJSON object or Draco geometry.

5.2.4 Editing (attributes)

Editing one feature

With uncompressed CityJSON, the corresponding City Object is popped from the CityJSON object and added to the `"CityObjects"` array with they key being the ID with "1" added at the end. The file is stored and can be downloaded by the client.

As for compressed CityJSON, again the complete file has to be downloaded. Normally the same thing as above is done, and the dataset is subsequently encoded again. For compression types using the "replace" method howeverI tried to do it more efficiently by directly finding the ID in the array of attributes and editing that as well. In this way the CityJSON skeleton would not have to encoded again. However, I realised that this is not necessarily possible to do when it is something other than an object ID being edited. For example when a string is edited that actually occured twice in the dataset, you can not just edit that sting in the attributes array but would have to add a new string and update the correct reference in the JSON skeleton. It is therefore not a completely fair comparison as methods using "replace" are now compressed much quicker than they should.

When using Draco, the Draco geometry is not decoded as we know only attributes are edited. It can be handled the same as other compression types.

Editing all features

This is done in the same way as explained in Subsection 5.2.4, but for every feature.

5.2.5 Editing (geomety)

Editing one feature

With all compression types, the file is transmitted as a whole in order to simulate a user editing a complete file in the browser. This means that they need to be able to see the complete data.

When editing a geometry of uncompressed CityJSON, all vertices that appear in the geometry of the feature are increased in height by 1 unit. This is actually not a completely correct implenetation, as the vertices could be shared with other features. Instead, the vertices should be copied before editing, and the original ones checked on being used by other features, otherwise removed.

With compressed datasets that have original geometries this is done in the same way. If using Draco, the vertices are edited within the decoded Draco geometry. This means they have the same problem as the geometry editing implementation of uncompressed CityJSON.

Editing all features

It is done in the same way as with one feature (see Subsection 5.2.5), but now all vertices of the dataset are edited. This means that the problem of editing shared vertices is not there as all features are moved up by 1 unit.

5.3 Testing and benchmarking with Selenium

The Selenium WebDriver [Selenium, 2020] enables the simulation of web browser use, which can be utilised to test web applications. It is used to iteratively visit the Flask routes that correspond to a test with a specific file type, dataset, and operation. By having it control Google Chrome, it is possible to configure network settings that simulate an actual network connection rather than letting the server run with local computer speed. This will mostly impact datasets that are larger than the set download speed limit. Performing an operation on them will take longer as their downloading is simulated.

The download speed is set at 40Mbit/s (or 5MB/s) as this is the average network speed in The Netherlands [Cable, 2019]. The network latency, which is the time it takes for a request to be

transmitted from a client to a server [KeyCDN, 2018], is set to 0ms as it is harder to determine an appropriate number for this and it would add up the same amount of time for every test iteration anyway.

During long processes on either the server or in the client a time out exception can occur, interrupting the test. This would especially happen while buffering large datasets. The time out could happen for multiple parts of the test platform: Flask-SocketIO, Chrome, or Selenium. When the problem occurs it is not always certain at which point the test iteration breaks.

Attempting to relieve this problem, I set Selenium to wait a long time for an alert to pop up with the message being that the current task has been finished before continuing with the next test iteration. Flask-SocketIO has a function to alter its time out settings as well, but this turned out to not be important anymore after changing the structure of the test platform, which is downloading a file after the html template starts rendering, rather than transmitting the data directly with render_template(). Chrome also has a time limit before it returns a time out. It however does not support the alteration of this parameter. Firefox on the other hand does support this, but in liaison with Selenium it can not be throttled.

Ultimately, after attempts to fix the problem, it only remained to pose a problem for buffering all geometries of the large datasets, which is seen in Section 6.6.2.

5.4 Compression and decompression

The compression techniques are implemented using Python and the Draco compression library. Decompression always takes place on the testing platform in JavaScript. This section follows the order of table 4.2.

5.4.1 original-zlib, original-cbor, and original-cbor-zlib

These are the simplest compression types. It simply involves to encode the full dataset with zlib, cbor, or first cbor and then zlib. For zlib, the input needs to be a Unicode-encoded string. It requires the zlib and flunn libraries for Python to compress the data, and the pako and CBOR libraries for JavaScript for decompression. The implementations are as follows: "cm.j" being the CityJSON data:

```
import zlib
encoded = json.dumps(cm.j).encode() # encoding the JSON into a Unicode string
compressed = zlib.compress(encoded) # compressing the encoded JSON with zlib
```

```
import flunn
compressed = flunn.dumps(cm.j) # parsing the JSON into CBOR
```

```
import zlib, flunn
cborcompressed = flunn.dumps(cm.j) # parsing the JSON into CBOR
finalcompressed = zlib.compress(cborcompressed) # compressing the CBOR with
zlib
```

Decompresion is done in the browser client in the following ways:

```
import pako
decompressed = pako.inflate(compressed) # decompressing zlib-compressed data
```

```
import CBOR
jsonData = CBOR.decode(compressed) # parsing CBOR into JSON
```

```
import pako
import CBOR
decompressed = pako.inflate(compressed) # decompressing zlib-compressed data
jsonData = CBOR.decode(decompressed) # parsing decompressed data (which is
now CBOR) into JSON
```

5.4.2 original-cbor-smaz

The idea is to recursively replace all keys and string values of the CityJSON object by its smaz-encoded equivalent. It can be done using the smaz library for Python, and it seemed to work well. However, decoding the strings in JavaScript with a port of the Python library gave wrong results that could not be used.

Firstly, the binary code (which would be of type "Uint8Array" in JavaScript) has to be converted to a string. It can then be given to the "decompress" function of the smaz library, which would retrieve the character code of every character of the string. The character code is then compared to the smaz codebook (containing fragments of words or bigrams, see 2.2.6) and converted. However, it seemed that the character codes of some characters of the string are much bigger than the size of the codebook, thus not returning a proper value.

5.4.3 original-replace, original-replace-zlib, original-cbor-replace, and original-cbor-replace-zlib

For the "replace" compression method, all keys and values are traversed with a recursive function, counting the amount of times that strings appear in the dataset. Numbers are ignored because they normally are small or unique. All strings are then placed in an array ordered by frequency of appearance. All keys and (string) values are subsequently replaced by the index of which they appear in the array. The index is stored as a string as well because numbers are not replaced and they would otherwise get mixed up with values that are intended as index. This means that the dictionary is now more like a JSON-skeleton with references to values in the array. The advantage is that strings that occur multiple times are now only stored once, resulting in a compressed representation of the data.

The result is a dictionary with three members: "cm", "str", and "geom", containing respectively the JSON-skeleton, the array of strings, and the original geometry of the dataset. An example is shown in Listing 5.1.

Listing 5.1: Example of dataset compressed with replace method

```
{
    "cm": {
        "type": "Compressed CityJSON",
        "CityObjects": {
            0: "1",
            2: {
                3: "4",
                5: "6",
                7: "8"
            }
        },
    "str": ["type", "building", "attributes", "bgt_status", "bestaand",
            "creationdate", "2014-07-09", "class", "dek"],
    "vertices": [..]
}
```

For the zlib, cbor, and zlib+cbor versions, the same thing is done on the aforementioned dictionary as was done on the original datasets in Section 5.4.1 . These would be decompressed in the same way as well. The created dictionary itself is decompressed by recursively traversing all keys and values in the JSON skeleton and replacing them all with the string in the array that the keys and values are refering to.

5.4.4 original-cbor-replace-huff

For this combination of compression methods, the array containing all keys and values of the JSON would be Huffman-encoded (see 2.2.4). It can be done by using the array as input for the creation of a frequency tree, and encoding the array based on this tree. Both the encoded array and the tree would have to be transmitted in order to be able to decode the array. However, the structure of the tree differs greatly between the Python and JavaScript implementations. In Python, it is a dictionary with every character and the corresponding binary code. In the JavaScript implementation on the other hand, a full tree is needed as input. It is a nested dictionary where every key (which is a leaf of the tree) contains its left and right child. Because a conversion is needed, I could not finish the implementation.

5.4.5 Draco-compressed equivalents

The difference with the counterparts with uncompressed geometry is that all vertices and Geometry Objects are stripped from the CityJSON object prior to compressing the attributes. The geometries are converted to OBJ using cjio, which in turn is compressed with Draco. This is done by calling the terminal with the appropriate command. The compression level is always 5 as it is in the middle, and metadata always has to be included since otherwise the separation between different objects is lost. Quantisation is turned off since it would result in the loss of coordinate precision (see **??**).

The geometries are converted to OBJ with the following line of code:

```
from cjio import cityjson
obj = cm.export2obj()
```

Subsequently, the OBJ is compressed by Draco with a terminal command akin to:

```
draco_encoder.exe -i dataset.obj -o dataset.drc -cl 6 -qp 0 –metadata
```

Where `-cl` indicates the compression level, `-qp` the quantisation, and `-metadata` the separation between objects (rather than compressing them into one full mesh).

Since it is troublesome to load a Draco geometry directly from a blob, they are not embedded in the files as they are benchmarked (but still included in the compression time and file size results). These files are downloaded separately by passing a URL to the Flask server that will initiate the downloading of the Draco file belonging to the tested dataset. This means that Draco geometries are not compressed further when using zlib, at least when testing the compressed datasets on time performance. It has to be kept in mind when interpreting the results, because the files would actually have been even smaller. Compressing the Draco files separately would not give the same results either, as there will always be extra overhead when two files are compressed separately. Additionally, it would be complicated to implement decompression in the Draco loader for JavaScript, as the explanation on the decompression of Draco below will imply.

For decompression `THREE.DRACOLoader` is used, a function that is included in the three.js library. It needs a URL or the local file path to the to-be decoded Draco file.

5.5 Dataset description

A selection of datasets that are available on the CityJSON and Random3DCity webpages has been made, with the addition of a city model of Rotterdam (see table 5.1). As mentioned part 4.2, the data should have varying characteristics. There is a difference in the inclusion of terrains, language (Dutch and English), file size, and LOD. The four last ones have been chosen to make a fairer comparison between datasets of different LODs, as they do not contain attributes which can influence the results. This is unwanted, because it is the geometrical compression (Draco) only that should make a difference in that regard.

	File size (MB)	LoD	Attributes	City Objects	Other characteristics	Source
Delft	1.1	1	Medium	570	Dutch, contains terrain	CityJSON [2020]
Den Haag	2.9	2	Few	2498	Dutch, contains terrain	CityJSON [2020]
Rotterdam	5.3	1 & 2	Medium	654	Dutch, multi-LOD	Municipality of Rotterdam [2020]
Montréal	5.4	2	No	294	No attributes	CityJSON [2020]
Singapore	52.8	1	Many	10966	English	Biljecki et al. [2017]
TU Delft campus	69	1	Medium	17654	Dutch	Ledoux [2019]
New York	105	2	Few	23777	English	CityJSON [2020]
Zürich	292.9	2	Few	198699	English	CityJSON [2020]

Table 5.1: Chosen datasets and their characteristics, sorted by file size

To do: add images/listings showing parts of datasets

5.6 Experiments

Another possibility is to stream features to the client. To do this, the dataset needs to be prepared such that every feature contains its own geometry, rather than having references to the vertices list at the end. The OGC API CityJSON prototype describes how such a dataset would look for CityJSON. To compress it, it would be possible to replace the geometries with a Draco object. However, testing this it turned out that Draco does not perform as well when it has to encode every object separately - the file size would be bigger. Since Draco geometries would have to be decoded client-side, this

means that it would only take longer to use it in this way. For this reason performance tests have not been completed with this streaming implementation.

5.6.1 Data loss

Zlib, CBOR, smaz, and Huffman encoding are explained in Chapter 2 and are completely lossless compression techniques.

Draco is mostly lossless when not using quantisation (see Sections 2.3.1 and 3.2). If you do use it, some vertices are lost and precision is decreased, of which the extent depends on the set quantisation level. This is not acceptable and therefore not used. The compression levels that can be set in Draco (Section 3.2) are separate from this, and only determine the used compression techniques that the library encompasses (see Section 3.2).

I have compressed OBJs of every dataset (thus converted from CityJSON) into all Draco compression levels, and decoded the Draco geometries again. Comparing the original OBJs to the decoded OBJs shows that they are completely identical, so Draco compression itself does not induce any loss of information. However, geometries are triangulated when converting CityJSON to OBJ with cjio [3D geoinformation research group at TU Delft, 2019], so if the input file dataset is not triangulated the decoded result is not the same.

The replace method (explained in Subsection sec:implreplace) is supposed to be lossless. Since I have created it myself, it needs testing to be sure. I have done this using the DeepDiff library [Dehpour, 2020] which can thoroughly compare Python dictionaries, and as such also two JSONs.

To do: I have investigated loss with DeepDiff, and there is some small information lost. I need to investigate this further to see if I can fix it before I can describe it into detail.

5.6.2 Querying compressed CityJSON

The possibility of querying compressed datasets gives an advantage when not all features are requested. With the current implementation of the testing platform, a compressed dataset has to be transferred as a whole, whereas original CityJSON can be prepared on the server before transmitting the data. This means that when for example only one City Object is queried, the transmitted data is smaller when using orginal CityJSON as opposed to a compressed variant, losing the benefits of compression.

It is not possible to query a Draco geometry directly, at least such a function does not exist in its JavaScript decoder nor its C++ one (see Google [2020a,b]. It has to be decoded fully before separate features can be retrieved. Additionally, zlib is a general purpose compression library (see Subsection 2.2.5) that has the purpose of compressing files completely and provides no possibility to partially decompress files.

CBOR (see Section 3.6) on the other hand might have the potential to be queried in similar fashion to JSON, as it has a key-value structure as well. The libraries that I use for CBOR [Yura, 2016; Hildebrand, 2020] in this thesis do not provide clear functions to do this however. This needs to be investigated further and can be future work. It could also be done with other binary JSON formats that are mentioned in Section 2.2.1.

To do: explore how to query CBOR? Also explain possibilities of querying compression types using replace, smaz, and huffman

5.6.3 Draco compression level

Draco has several compression levels that can be set when encoding an OBJ. In order to avoid having too many variables to test in the benchmarking chapter, one compression level is chosen for every compression type that includes Draco compression. For this purpose I have run tests using Pytest to pick one that is the most suitable on average. The variables are encoding time, file size, and decoding time.

Theoretically a higher compression level will result in a lower file size, at the cost of a longer encoding and decoding time. The encoding time is the least important in this case, because it is mostly part of the dataset preparation. The most ideal option is the compression level that has a good ratio between file size and decoding time, because this will increase the performance on the web the most.

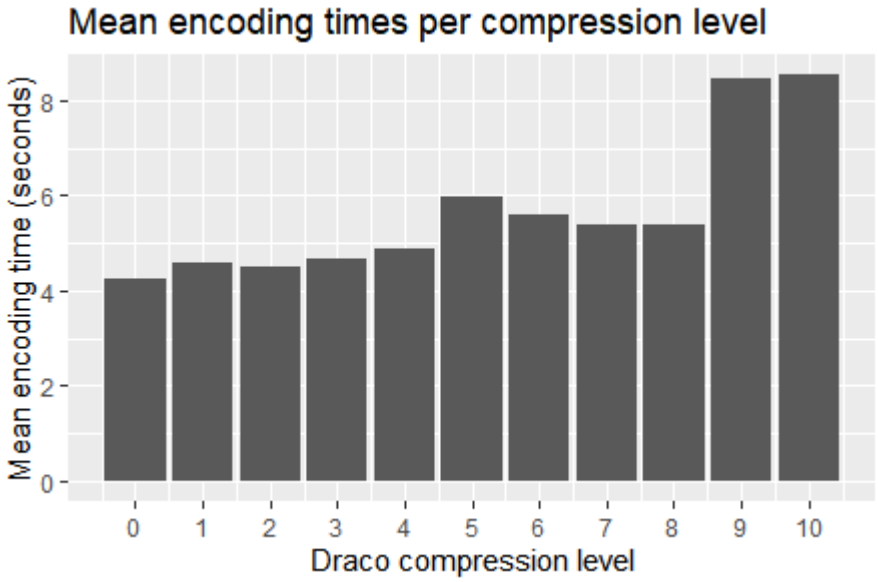

Figure 5.2: Benchmark of mean encoding times per different Draco compression level

Figure 5.2 shows that a higher compression level is not always slower than its predecessor. This is counterintuitive, but the reason for it is likely that individual objects are being encoded, rather than one big mesh. The different levels are probably determined based on their performance on larger meshes. Small meshes are less complex and compression techniques will therefore behave differently on t. Especially level 5 seems to be a bad choice if the encoding time is important, as it is even higher than 8 and 9. It is not known why level 5 in specific performs badly, as it is not specified in the Draco specifications which techniques are used per level.

As for the file sizes, figure 5.3 shows the average file size multiplier of the Draco geometry compared to the original OBJ. There are three groups: 0, 1 to 4, and 5 to 10. Between the second two groups, the difference is probably because of the first one using the sequential connectivity method, and the second one using edgebreaker as this is said to be a main difference between compression levels. Compression level 0 might compress the vertices only and not the connectivity. Interestingly, despite level 5 taking relatively long to encode, its file size multiplier is not lower than several higher compression levels that encoded faster

Figure 5.4 visualises the mean decoding times. It is tested with the Draco C++ library however, as opposed to the decoder for JavaScript which might perform differently. Between compression levels 1-6 there is not much of a difference, but 5 is again worse than 6. Despite it being the middle option,

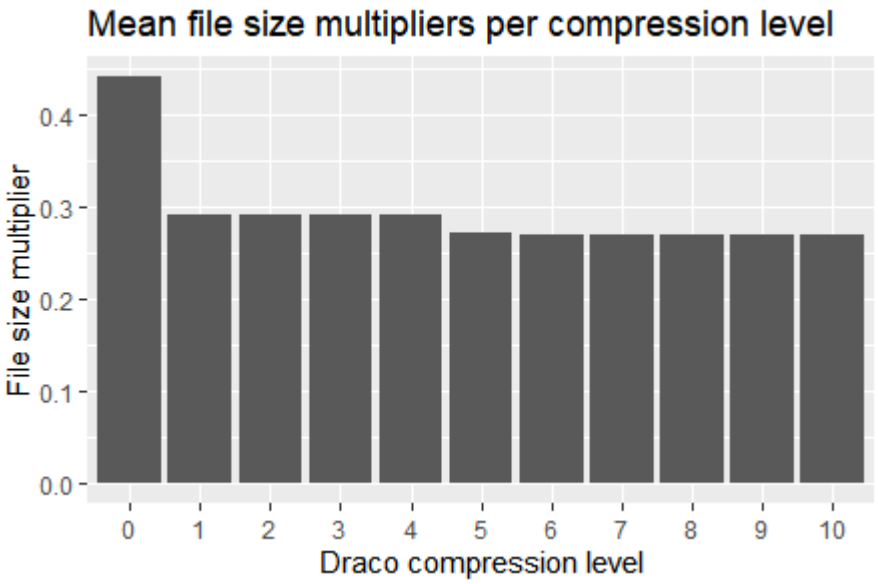

Figure 5.3: Mean file size multipliers per Draco compression level

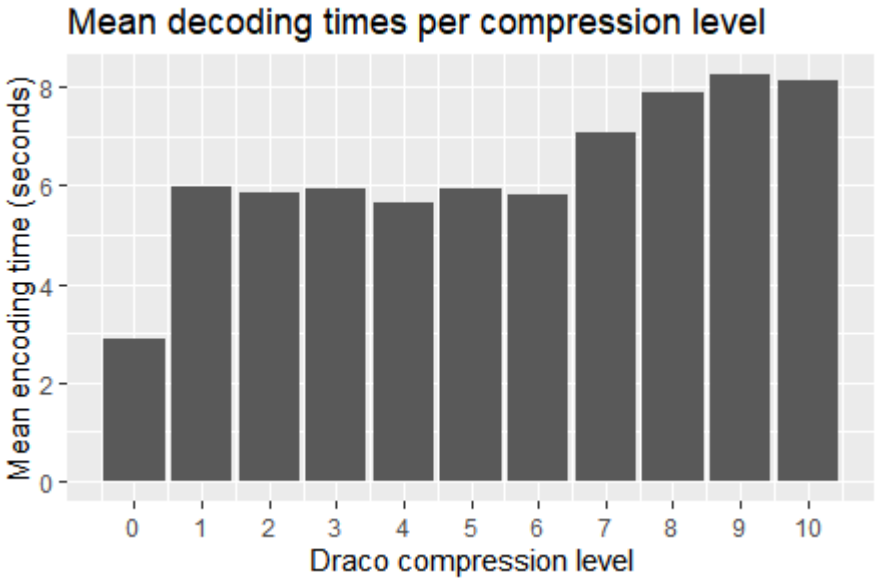

Figure 5.4: Benchmark of mean decoding times per different Draco compression level

it turns out to not be an optimal choice. Level 0 is fast but it yields a relatively high file size. Levels 7-10 take longer to decode than 6 despite resulting in similar file sizes.

Compression level 6 has just a slightly longer decoding time than 4, but it is smaller in size. Concludingly, by having a low decoding time, small file size, and reasonable encoding time, it is the best option to use.

5.6.4 Streaming CityJSON and compression

Subsection 3.5.1 has introduced a proposal for a structure in which CityJSON could be streamed over the web. Streaming is related to the efficient web use of 3D geoinformation, but it is not

within the scope of this thesis to test streaming performance. Despite that I have looked into how compression can potentially contribute to efficient streaming to see if it is worthwhile to investigate it further in the future.

Listing 3.4 shows how CityJSONFeatures look like, which can be streamed individually as they contain their own array of vertices. This gives the possibility to compress the geometry of a CityJSONFeature with Draco. There are two ways to do this: compressing every City Object of a Feature into individual Draco geometries (which would then be a parent with its children), and compressing all City Objects of a Feature together in one Draco geometry. The former option removes the need of having to separate objects within the Draco file, thus object IDs not needing to be stored inside. With the second option this does have to be done (just like when compressing complete CityJSON datasets)—but only if the Feature contains multiple City Objects—, but having one Draco file might be more efficient as the header of the file is relatively smaller.

A caveat is that datasets need to have `"transform"` to be able to be compressed properly with Draco, while the streaming implementation currently does not allow this Ledoux [2020]. Additionally, since Draco blobs can not be stored in JSON it is necessary to parse the file into a binary format, for which CBOR is already used throughout the thesis. CBOR on its own however can also potentially improve the performance.

The file sizes of four streaming variants of the datasets used in this thesis are shown in table 5.2. The CityJSONFeatures of every dataset are aggregated in a CityJSONFeatureCollection, akin to GeoJSON's FeatureCollection. The "collection" column this represents uncompressed CityJSON datasets that can be used for streaming. "collection-cbor" is compressed with CBOR, "collection-cbor-draco" includes individual Draco blobs for every City Object of a CityJSONFeature, and "collection-cbor-draco2" has one Draco blob per CityJSONFeature.

Dataset	collection	collection-cbor	collection-cbor-draco	collection-cbor-draco2
Delft	1.2	0.7	0.6	0.6
Den Haag	3.3	2.4	2.4	2.3
Rotterdam	3.9	2.5	2.6	2.6
Montréal	5.4	3.5	2.9	2.9
Singapore	54.5	34.4	32.1	32.1
TU Delft campus	70.6	41.8	25.5	25.5
New York	109.8	69.7	70.2	70.2
Zürich	284.2	182.5	163.2	165.6

Table 5.2: File size in MB for CityJSONCollections of datasets and compressed versions

It is shown that using CBOR already compresses the files, and possibly increases the reading and loading speed as compared to JSON (see Section 3.6 for an explanation on this). The addition of Draco compression is only worth it for datasets in which City Objects have parent-child relationships. The way in which Draco is incorporated is almost completely insignificant—the second Draco implementation gives a slightly better result for Den Haag, but a worse result with Zürich.

Concludingly, compression can be beneficial for streaming as well. But while the implementation with CBOR could easily be tested, this is not the case when using Draco. Out of the box it is not possible to directly load Draco blobs in the JavaScript decoder, so this would need to be implemented first.

5.7 Computer specifications

The specifications of the used computer are listed in table 5.3 below as the time benchmark results are influenced by it.

Model	Lenovo Thinkpad P1 Gen 2
CPU	i7-9750H@2.6Ghz
GPU	Nvidia Quadro T1000
OS	Windows 10 Home 64-bits
RAM	16 GB DDR4

Table 5.3: Specifications of computer used for benchmarking

6 Benchmarking results and analysis

The encoding, decoding, and file size performance are presented first in this chapter as these results can help explaining the results in other parts, but they are naturally factored within the benchmarks of the operations (see Section 5.2) as well. After this the time benchmarks of the remaining performance indicators (see Table 1.1) are shown in separate sections. These Sections are subsequently divided per operation. Lastly there is a conclusion including a plot where all operations are averaged.

The datasets (see Section 5.5) are divided in two classes: bigger (Singapore, TUDelft, New York, and Zürich) and smaller datasets (Delft, Den Haag, Rotterdam, and Montréal). I did this because showing the results per dataset makes it too complex, while the file size is the clearest divide between the datasets and also a factor that is expected to be significant for the performance.

Since the results are averaged by dataset group, box plots are presented. These show some statistics as well on the variety of the datasets within a dataset group. In the charts that show the performance of an operation on a compressed dataset, the baseline (uncompressed CityJSON) is shown with a dotted line. The time multiplier on the x-axis shows how much time it took to complete the specific task as compared to the baseline—a time multiplier of 0.5 means that it took half the time of doing the same thing with uncompressed CityJSON, and a time multiplier of 2 means that it took twice as long. The scales of the plots are not always fixed, which means that the two plots (bigger and smaller datasets) within one figure can have different scales. This requires extra attention when interpreting the results, but it is necessary to make them readable when the time multipliers between the two plots are very different.

Besides these plots, for all operations a table is added that shows the mean performance of every compression type per dataset type. It is sorted ascedingly so that it can easily be seen which compression types are the best for the corresponding operation.

To do: rounding the numbers in these tables. Also should fit the figures within the margins of the page I think.

The conclusion (Section 6.9) gives an overview of the results, indicating the advantages and disadvantages of every compression type. The full results table is found there (Table 6.19), as well as a plot that shows an overview of the performance of the compression types (Figure 6.13). Again per dataset group, but with all operations averaged. Through this it can more easily be noticed how different compression types compare when the wanted operation is unknown. Thus answering the question: "when you want to compress a dataset of this size and use it for any kind of purpose, which compression type would be advisable?".

6.1 Encoding performance

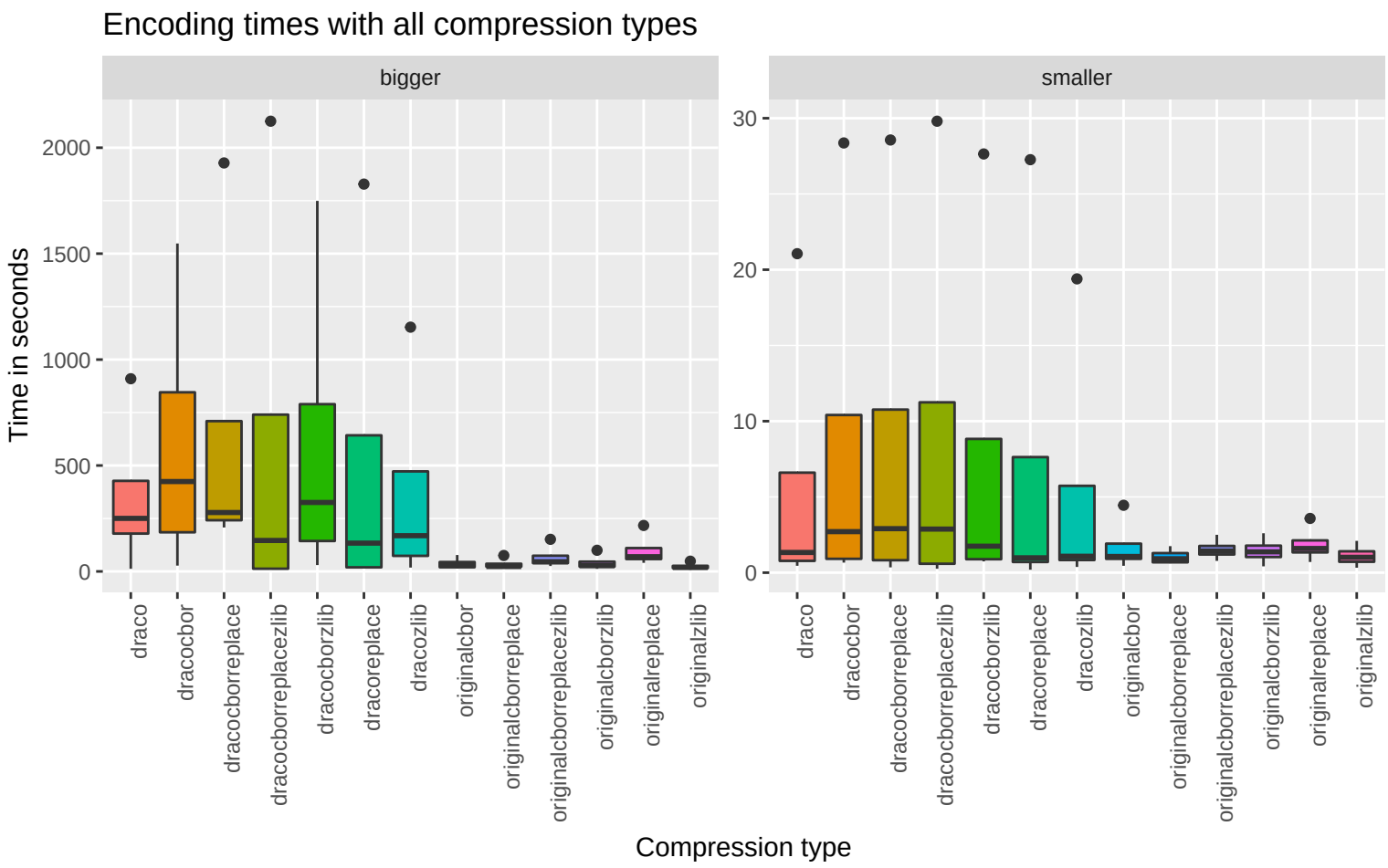

Figure 6.1: Boxplots of the encoding time performance of different compression types.

The encoding times shown here only apply to the preparation of datasets in Python. Therefore they are not an influence on the other performance results. After editing datasets they also need to be encoded, but this is done in the client and not in the server. Encoding times in the client are incorporated in the editing benchmark results.

Compression types that include Draco always take considerably longer to encode, which is true for both bigger and smaller datasets. That the impact is this large is likely because of a CityJSON dataset having to be converted to OBJ first before it can be decoded by Draco. For larger datasets, Draco compression only already takes four minutes. It could become a large share of the execution time of the preparation of datasets. For smaller datasets this is not a problem because it is still fast in absolute terms, but it is still more expensive.

Draco-cbor on average takes longest for bigger datasets. Combining it in draco-cbor-replace-zlib it is faster. Possibly this is because CBOR can encode the keys and values of a dataset faster in this structure, where all strings are contained in one array and encoded in zlib which results in a binary value. For smaller datasets these two compression types have similar performance, so the difference in JSON-structure (which is parsed into CBOR) is only noticed with a large dataset. Draco-cbor-replace-zlib is the fastest of the Draco ones together with draco-replace.

I had expected that the replace method would be expensive, but there is no clear pattern in it. Draco-replace is faster than the other Draco types for bigger datasets, but original-replace is the

slowest one without Draco. The other types that use replace but also are in CBOR are encoded faster, despite having more steps in the compression. It could possibly explained by CBOR being written to a file faster than JSON, or it is just simply written faster due to its size being smaller.

Original-zlib is however the fastest on the bigger datasets. Zlib thus seems to be the fastest compression technique to use on bigger datasets, but with smaller ones it is not significantly faster than others. Based on both plots, original-cbor, original-cbor-replace, or original-zlib would be the best choices if the encoding time is important.

6.2 Decoding performance

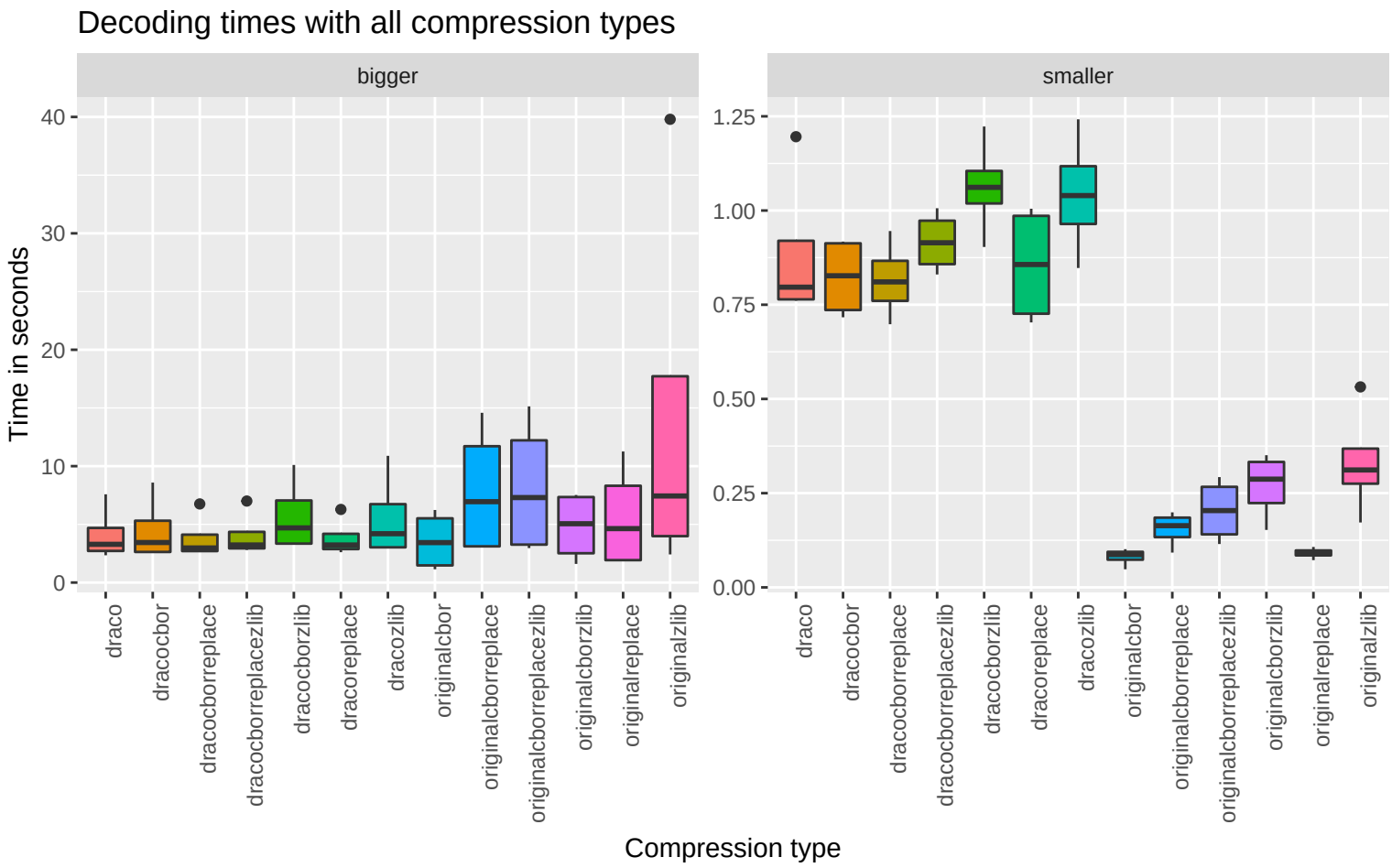

Figure 6.2: Boxplots of the decoding time performance of different compression types.

The decoding times are shown with a normal scale as the results do not have a large spread as there was with the encoding performance. These statistics should be a significant explaining factor throughout the results chapter as it is a main property that explains the difference in performance between varying compression types.

While encoding Draco took a comparatively long time, it is generally decoded quicker with the bigger datasets than the other compression types. This is not the case for the smaller datasets, suggesting that the larger a dataset is, the more suitable it is to use Draco.

Original-cbor on the other hand has a low mean decoding time as well, for both dataset types. Combining CBOR in any way with replace will decrease the performance, probably because it is expensive to traverse all keys and values and replace them.

Original-zlib takes long to decode, contrasting its encoding results. Its outlier on the left plot is the Zürich dataset, which took a considerably longer time than decoding the same dataset compressed in another way. The extent of the corresponding box plot also indicates that the larger a dataset is, the longer it takes to decode it as compared to the other compression types. This can potentially pose to be a disadvantage of original-zlib.

6.3 File size

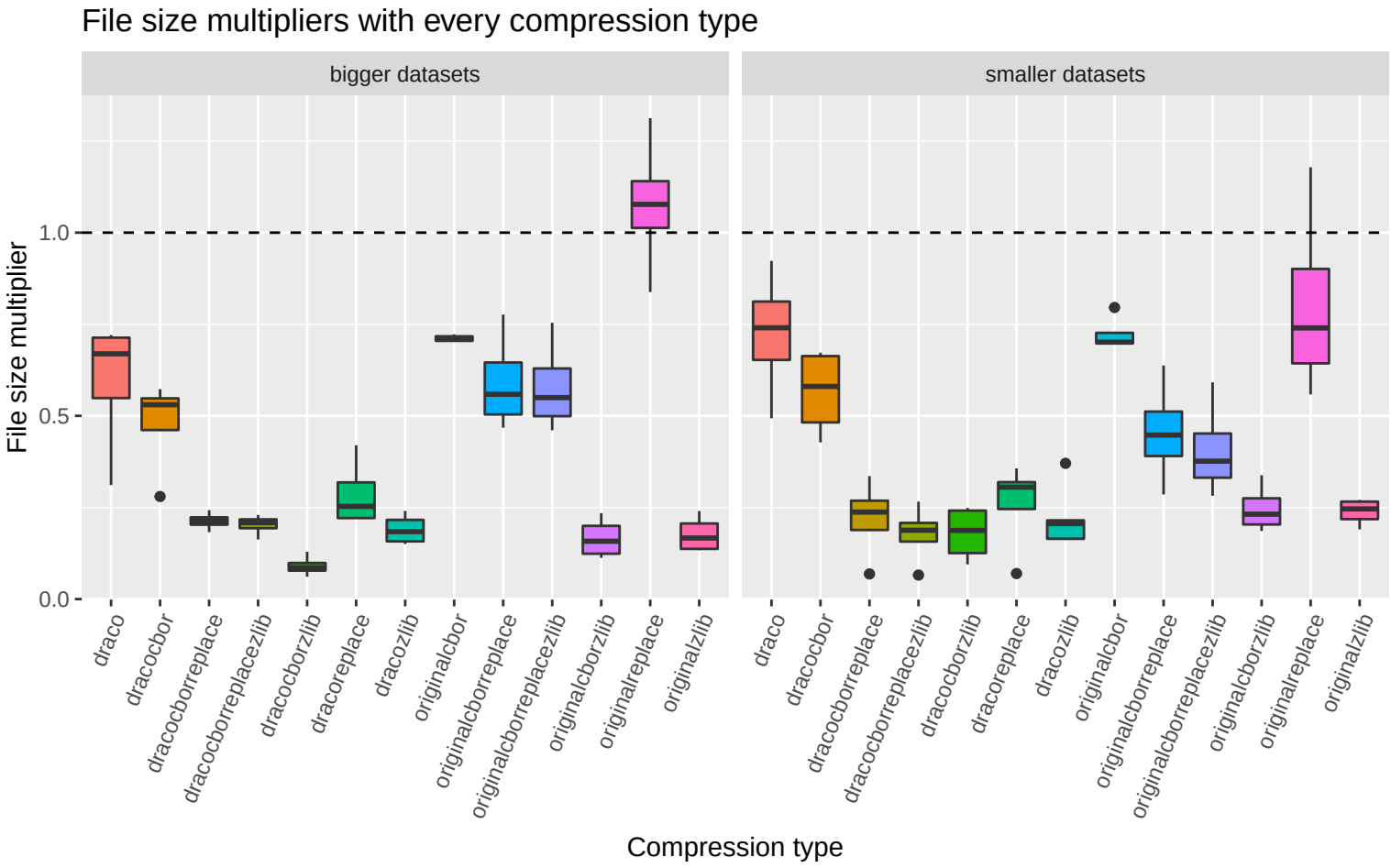

Figure 6.3: Boxplots of file size multipliers indicating to which extent compression types compress the data.

The file size multipliers are similar between the bigger and smaller datasets. It is again shown that compressing geometry with Draco is mostly suitable for bigger datasets. These datasets have a lower file size multiplier than the smaller datasets compressed with Draco.

Original-replace is the only compression type that in some cases makes a file larger than the original. The array of attributes is then larger than the removed redundancy. The replace method in general does not yield very favourable results, even when the array of attributes is encoded.

Combining CBOR and zlib or using zlib only gives the smallest file sizes when compressing bigger datasets, for both Draco compression types and the ones with original geometry. This trend is seen for the smaller datasets as well. So compression types with zlib are both encoded quickly (section 6.1) and yield small files. On the other hand zlib takes long to decode as shown in section 6.2. Further performance benchmarks will thus demonstrate whether or not the decreased file sized outweighs the increased decoding time.

6.4 Visualisation

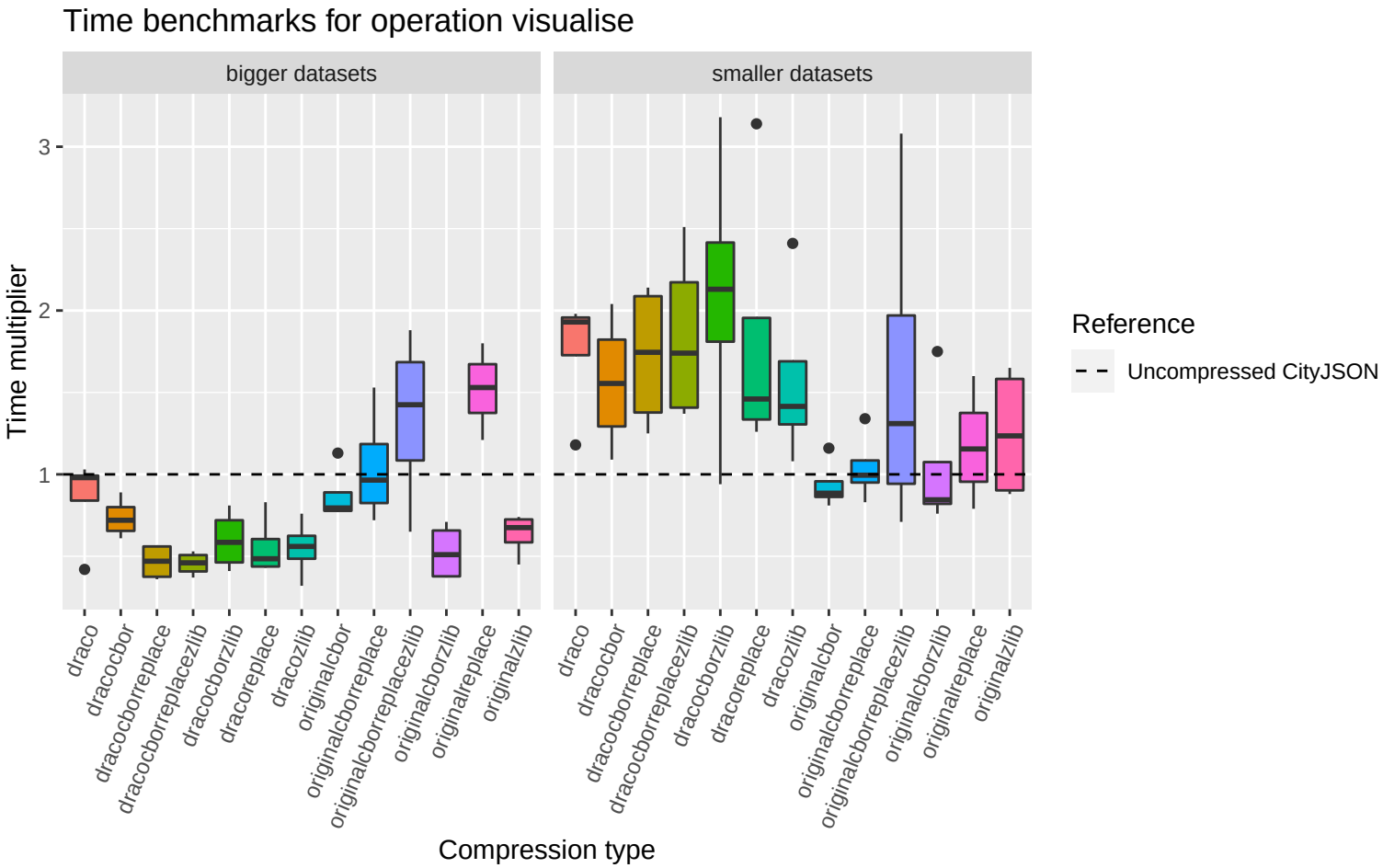

Figure 6.4: Boxplots of time performance of compressed datasets on visualisation compared to uncompressed CityJSON.

Visualising compressed smaller datasets generally gives results that are worse than the baseline. Only original-cbor original-cbor-replace, and original-cbor-zlib have acceptable performance. Other compression types have a time multiplier higher than 1, which means they are worse than uncompressed CityJSON for visualisation. With small files the saved transmission time apparently does not outweigh the time it takes for decoding. It is not a good idea to use compression in this case.

For the larger datasets, most compression types give good results compared to the baseline. Draco-compressed files in general perform better, but using Draco without further compression is not very beneficial. Despite that the combinations replace, cbor-replace, and cbor-replace-zlib do not work well with original geometries, for Draco it is fast. This is due to original-cbor-replace and original-cbor-replace-zlib having long decoding times and not so small file sizes, and original-replace giving the largest file sizes with bigger datasets (see Sections 6.2 and 6.3).

These three are the worst performers in this case. When using Draco any of the compression types that have additional compression besides Draco are good, and with original geometries original-cbor-zlib and original-zlib work well.

Table 6.1: Mean performance with visualise on bigger datasets

Compression type	mean
dracocborreplacezlib	0.4550
dracocborreplace	0.4650
originalcborzlib	0.5250
dracozlib	0.5500
dracoreplace	0.5575
dracocborzlib	0.5975
originalzlib	0.6350
dracocbor	0.7350
draco	0.8525
originalcbor	0.8725
originalcborreplace	1.0450
originalcborreplacezlib	1.3450
originalreplace	1.5175

Table 6.2: Mean performance with visualise on smaller datasets

Compression type	mean
originalcbor	0.9350
originalcborreplace	1.0400
originalcborzlib	1.0500
originalreplace	1.1750
originalzlib	1.2500
dracocbor	1.5600
dracozlib	1.5800
originalcborreplacezlib	1.6025
dracocborreplace	1.7200
draco	1.7550
dracoreplace	1.8300
dracocborreplacezlib	1.8400
dracocborzlib	2.0950

6.5 Querying

6.5.1 Querying one feature

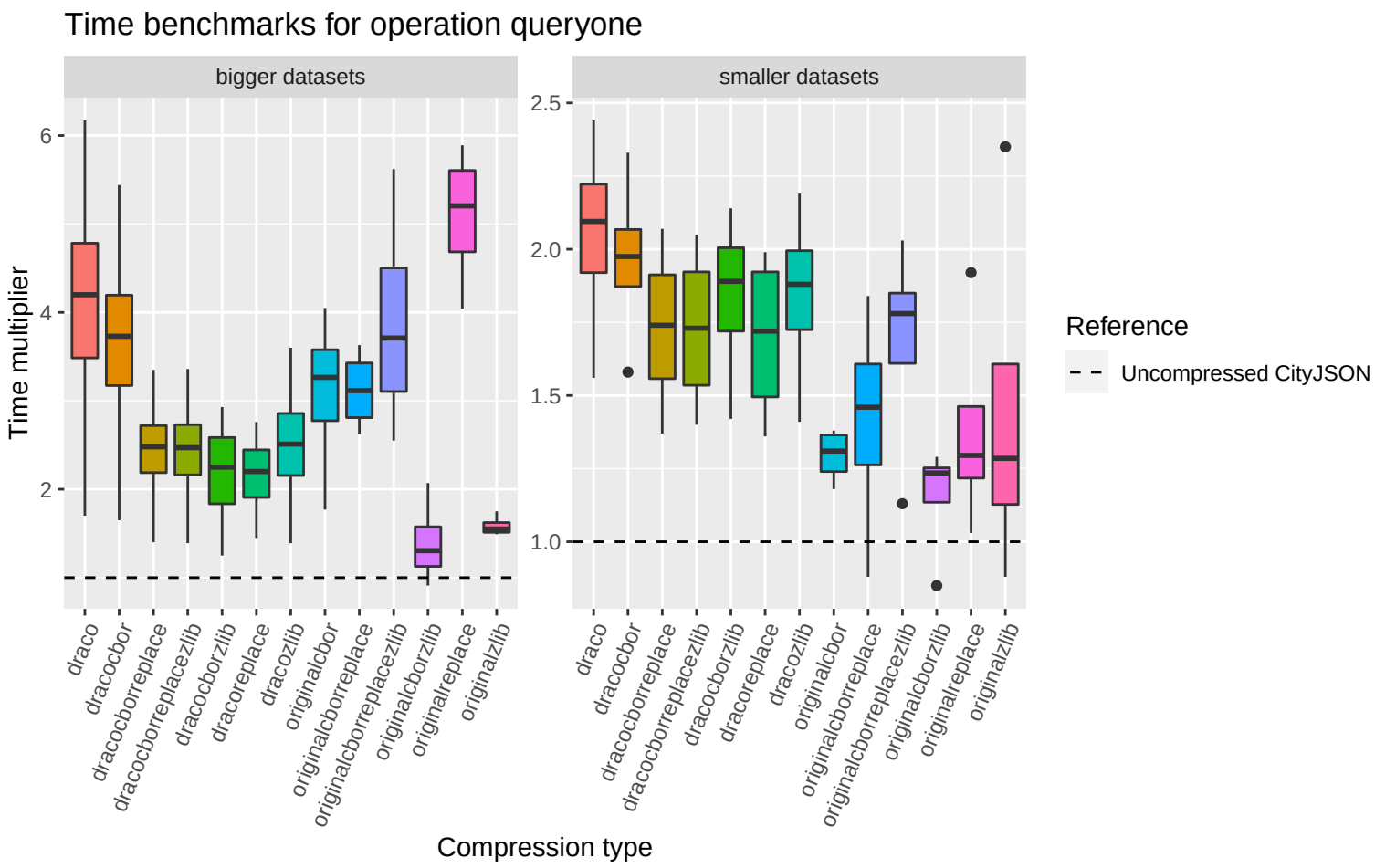

Figure 6.5: Boxplots of time performance of compressed datasets on querying one feature compared to uncompressed CityJSON.

As expected, querying one feature takes longer than the baseline for every compression type on average. The time multipliers with smaller datasets are lower relative to the ones of the larger datasets which is because there is less unneeded data (features that are not queried) to transfer. In the results of the bigger datasets it shows that the compression types that yield smaller file sizes tend to perform better. However, using Draco means that the geometries have to be queried apart from the City Objects, which adds time. Because of this and compressing files to relatively small sizes, original-zlib and original-cbor-zlib turn out to give the best performance.

While querying one feature of compressed datasets almost always takes longer, the results are not bad considering the difference in file size between a compressed full dataset and an uncompressed City Object. There are even some dataset and compression type combinations that turned out to be faster than uncompressed CityJSON as can be seen from the spread of the box plots. This indicates that the querying algorithm works faster in the client, because transmitting the full dataset first should not be beneficial.

Table 6.3: Mean performance with queryone on bigger datasets

Compression type	mean
originalcborzlib	1.3975
originalzlib	1.5850
dracoreplace	2.1525
dracocborzlib	2.1700
dracocborreplacezlib	2.4225
dracocborreplace	2.4275
dracozlib	2.5025
originalcbor	3.0875
originalcborreplace	3.1225
dracocbor	3.6375
originalcborreplacezlib	3.8975
draco	4.0675
originalreplace	5.0850

Table 6.4: Mean performance with queryone on smaller datasets

Compression type	mean
originalcborzlib	1.1525
originalcbor	1.2950
originalreplace	1.3850
originalcborreplace	1.4100
originalzlib	1.4500
originalcborreplacezlib	1.6800
dracoreplace	1.6975
dracocborreplacezlib	1.7275
dracocborreplace	1.7300
dracocborzlib	1.8350
dracozlib	1.8400
dracocbor	1.9650
draco	2.0475

6.5.2 Querying all features

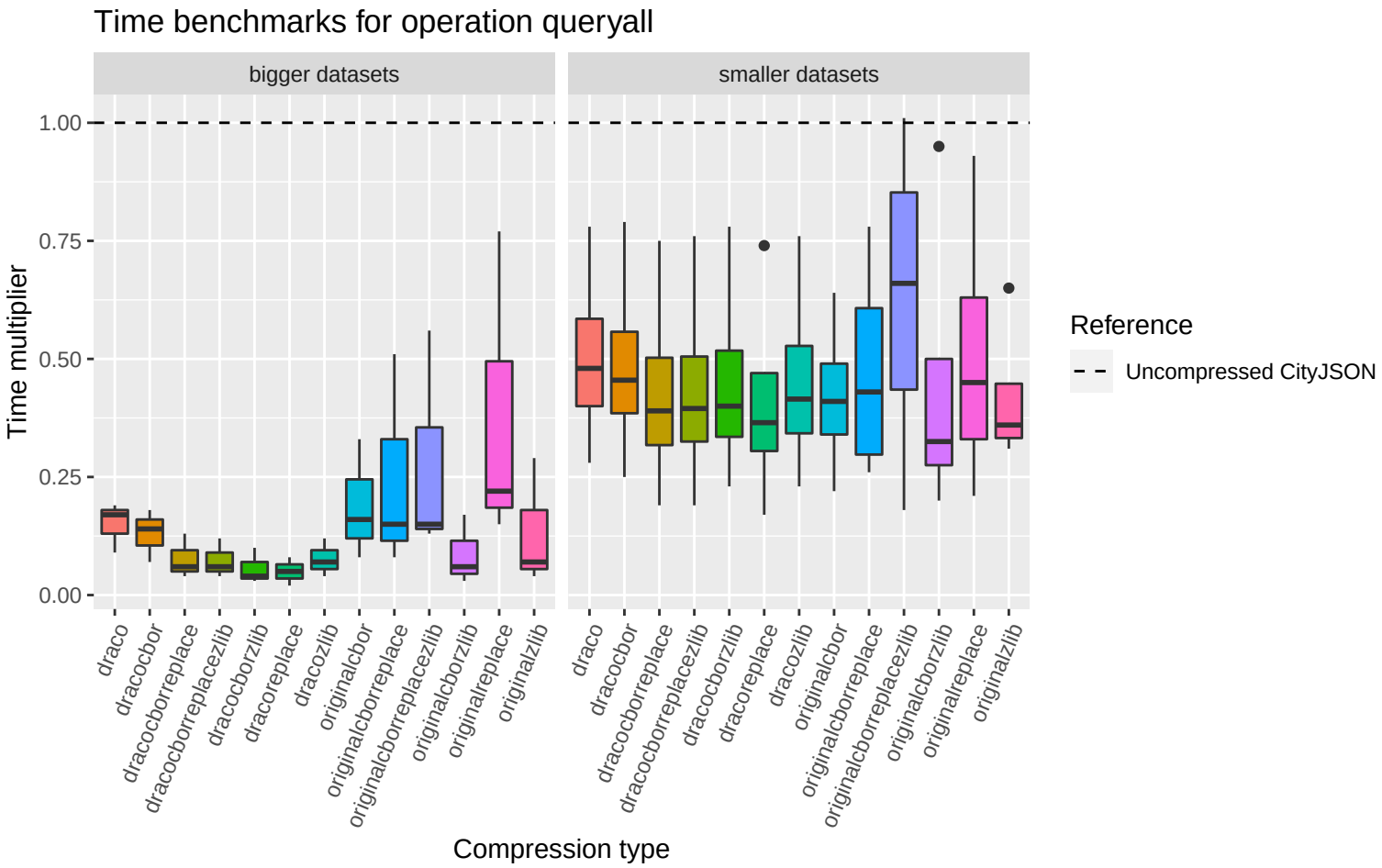

Figure 6.6: Boxplots of time performance of compressed datasets on querying all features compared to uncompressed CityJSON.

When querying all features it is almost always beneficial to use a compressed dataset, which gives the largest performance advantage when having a bigger dataset. For the smaller ones the performance is similar over the different compression types, but with original-cbor-replace-zlib having a relatively bad performance. This seems to be an anomaly, because it has a decoding time and file size multiplier similar to original-cbor-replace, yet a large difference in performance.

As for the larger datasets, compression types with Draco tend to perform well. This is despite having to traverse the Draco geometries separately, but this seemingly is made up by the low file size mulitpliers. Original-cbor-zlib and original-zlib have a good average performance as well, but draco-cbor-zlib has the best average. Original-replace, original-cbor-replace, and original-cbor-replace-zlib are relatively bad choices which is due to their comparatively bad combination of file size multiplier and decoding time.

Because original-zlib and original-cbor-zlib perform well the best with bigger datasets on both querying one feature and all features, these would be the best compression types to choose for querying.

Table 6.5: Mean performance with queryall on bigger datasets

Compression type	mean
dracoreplace	0.0500000
dracocborzlib	0.0566667
dracocborreplacezlib	0.0733333
dracocborreplace	0.0766667
dracozlib	0.0766667
originalcborzlib	0.0866667
dracocbor	0.1300000
originalzlib	0.1333333
draco	0.1500000
originalcbor	0.1900000
originalcborreplace	0.2466667
originalcborreplacezlib	0.2800000
originalreplace	0.3800000

Table 6.6: Mean performance with queryall on smaller datasets

Compression type	mean
dracoreplace	0.4100
originalcbor	0.4200
originalzlib	0.4200
dracocborreplace	0.4300
dracocborreplacezlib	0.4350
originalcborzlib	0.4500
dracocborzlib	0.4525
dracozlib	0.4550
originalcborreplace	0.4750
dracocbor	0.4875
draco	0.5050
originalreplace	0.5100
originalcborreplacezlib	0.6275

6.6 Spatial analysis

6.6.1 Buffering one feature

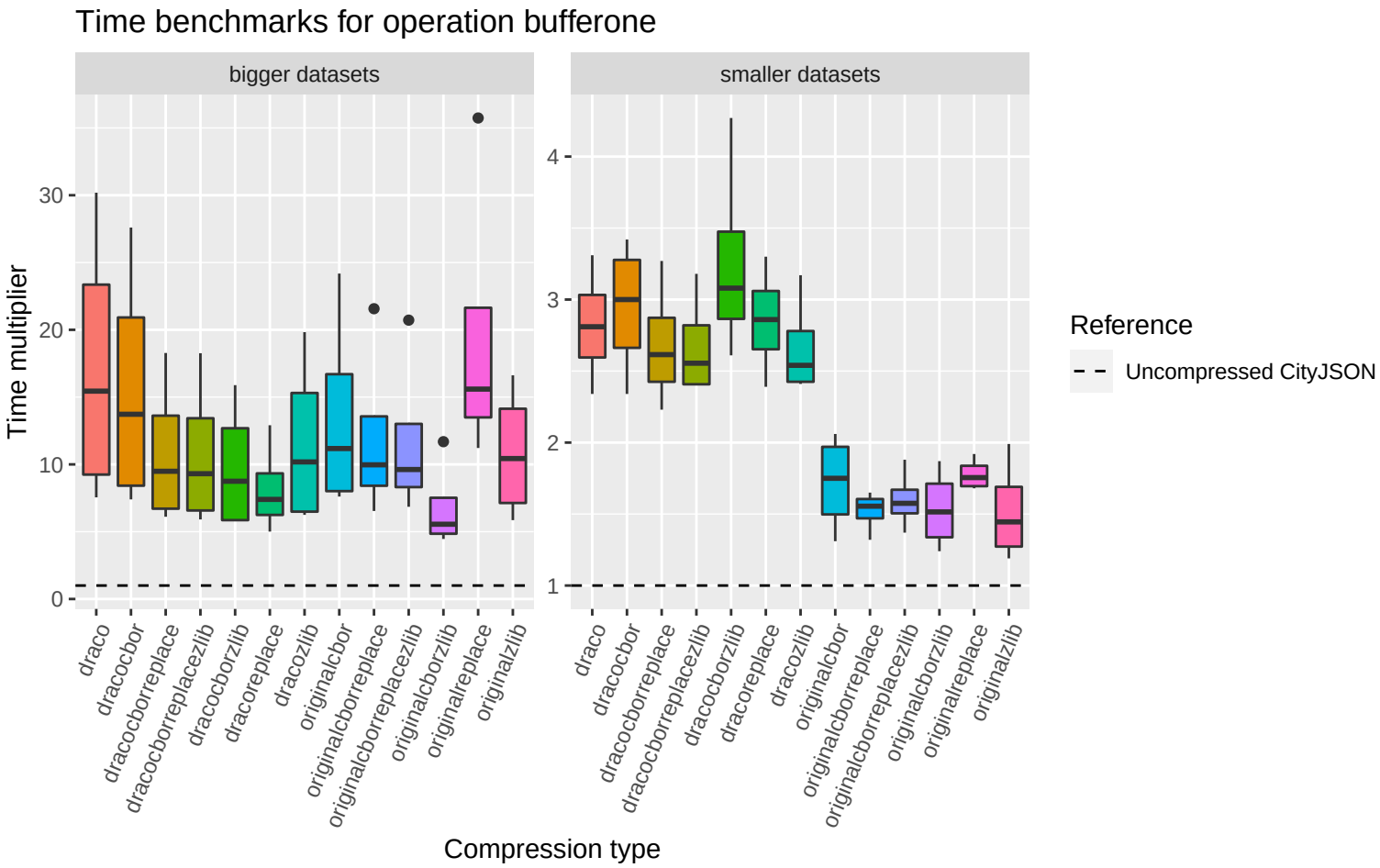

Figure 6.7: Boxplots of time performance of compressed datasets on buffering one feature compared to uncompressed CityJSON.

Buffering one feature always takes longer with compressed CityJSON due to the complete file having to be transmitted to the client, as opposed to one buffered uncompressed City Object.

With smaller datasets it tends to be more efficient when not using Draco compression, likely due to it having a relatively long decoding time (Section 6.2). Within the groups itself (Draco and original), the difference between compression types is not big. Draco-cbor-zlib is the slowest one and it does have a relatively long decoding time (see Section 6.2). Original-cbor, original-cbor-zlib, and original-zlib have a relatively large spread, indicating that the characteristics of the encoded file influence its performance more than with the other compression types with original geometry.

The bigger datasets take especially long compared to the baseline and it is a bad idea to compress a large dataset when wanting to buffer one feature. Generally the more non-geometry compression techniques are combined, the better the results are. Original-replace is the best one, but still not feasible.

Table 6.7: Mean performance with bufferone on bigger datasets

Compression type	mean
originalcborzlib	6.8150
dracoreplace	8.1800
dracocborzlib	9.7975
dracocborreplacezlib	10.6975
originalzlib	10.8375
dracocborreplace	10.8425
dracozlib	11.6125
originalcborreplacezlib	11.7025
originalcborreplace	12.0125
originalcbor	13.5375
dracocbor	15.6125
draco	17.1575
originalreplace	19.5375

Table 6.8: Mean performance with bufferone on smaller datasets

Compression type	mean
originalzlib	1.5175
originalcborreplace	1.5200
originalcborzlib	1.5350
originalcborreplacezlib	1.6000
originalcbor	1.7175
originalreplace	1.7775
dracozlib	2.6650
dracocborreplacezlib	2.6725
dracocborreplace	2.6825
draco	2.8175
dracoreplace	2.8525
dracocbor	2.9400
dracocborzlib	3.2600

6.6.2 Buffering all features

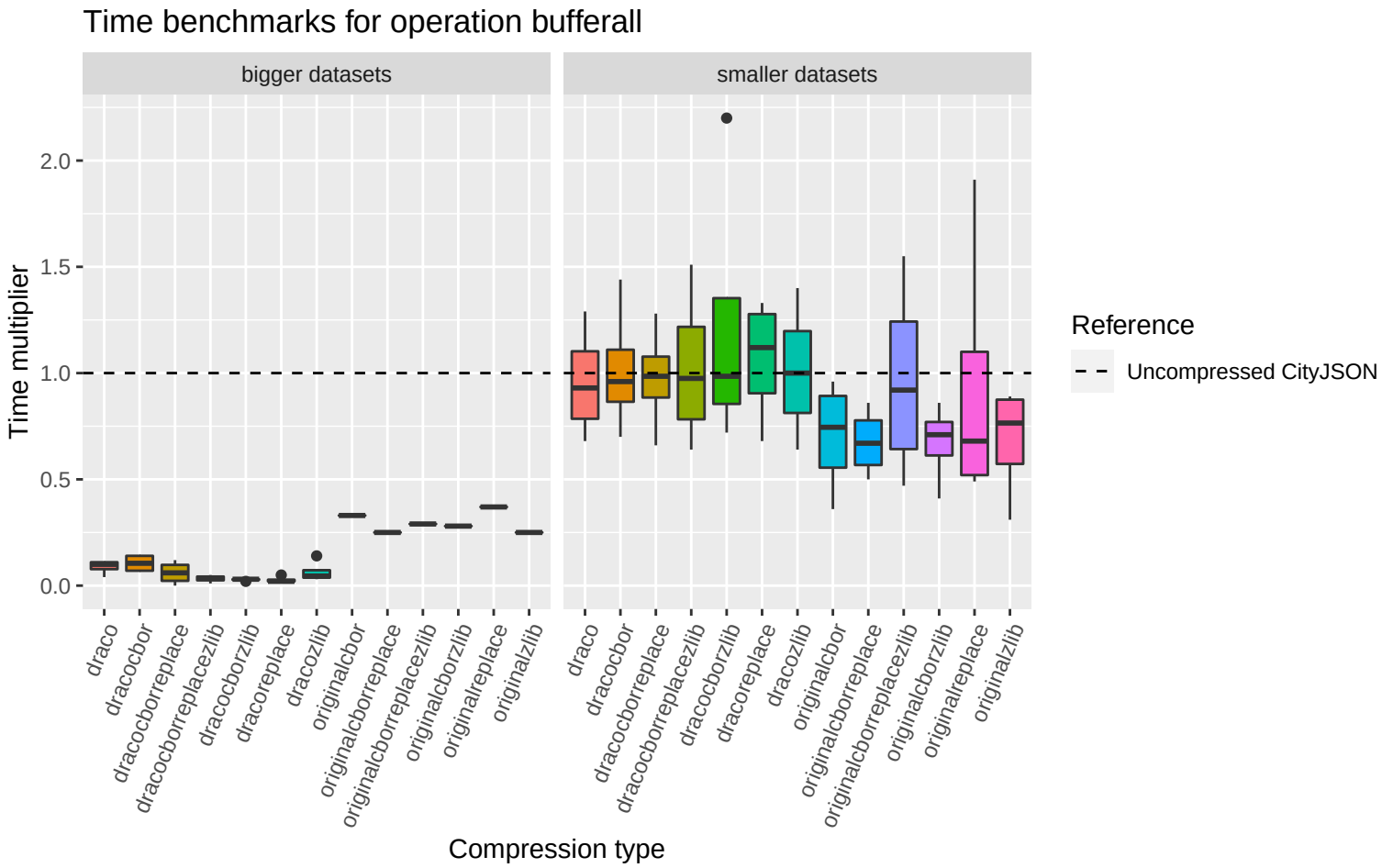

Figure 6.8: Boxplots of time performance of compressed datasets on buffering all features compared to uncompressed CityJSON.

Compression types with original geometries tend to perform better on small datasets compared to ones that use Draco, and this is again noticed. The performance can benefit from compression when buffering all features, but only when using original-cbor, original-cbor-replace, original-cbor-zlib, and original-zlib. This is despite coordinates having to be projected to WGS84 in the client (as explained in Section 5.2.3).

As for the bigger datasets, the benchmarking did not go completely well due to problems—with the New York, TU Delft Campus, and Zürich datasets the compression types using original geometry could not be tested. The testing platform has issues when the client has to process data for a long time (see Section 5.3), which is the case when buffering all features of a large dataset. For this reason the box plots do not look the same as in other graphs.

From the results that are there it can be told that the performance is increased—especially when using Draco. The reason could be that the algorithm with Draco is different, as the results of the buffer algorithm are parsed into a BufferGeometry rather than CityJSON's Geometry Objects (see sec:implbufferall).

Table 6.9: Mean performance with bufferall on bigger datasets

Compression type	mean
dracocborzlib	0.0275
dracoreplace	0.0275
dracocborreplacezlib	0.0325
dracocborreplace	0.0600
dracozlib	0.0650
draco	0.0875
dracocbor	0.1050
originalcborreplace	0.2500
originalzlib	0.2500
originalcborzlib	0.2800
originalcborreplacezlib	0.2900
originalcbor	0.3300
originalreplace	0.3700

Table 6.10: Mean performance with bufferall on smaller datasets

Compression type	mean
originalcborzlib	0.6725
originalcborreplace	0.6750
originalzlib	0.6825
originalcbor	0.7025
originalreplace	0.9400
draco	0.9575
originalcborreplacezlib	0.9650
dracocborreplace	0.9775
dracozlib	1.0100
dracocbor	1.0150
dracocborreplacezlib	1.0250
dracoreplace	1.0625
dracocborzlib	1.2225

6.7 Editing (attributes)

6.7.1 Editing attributes one feature

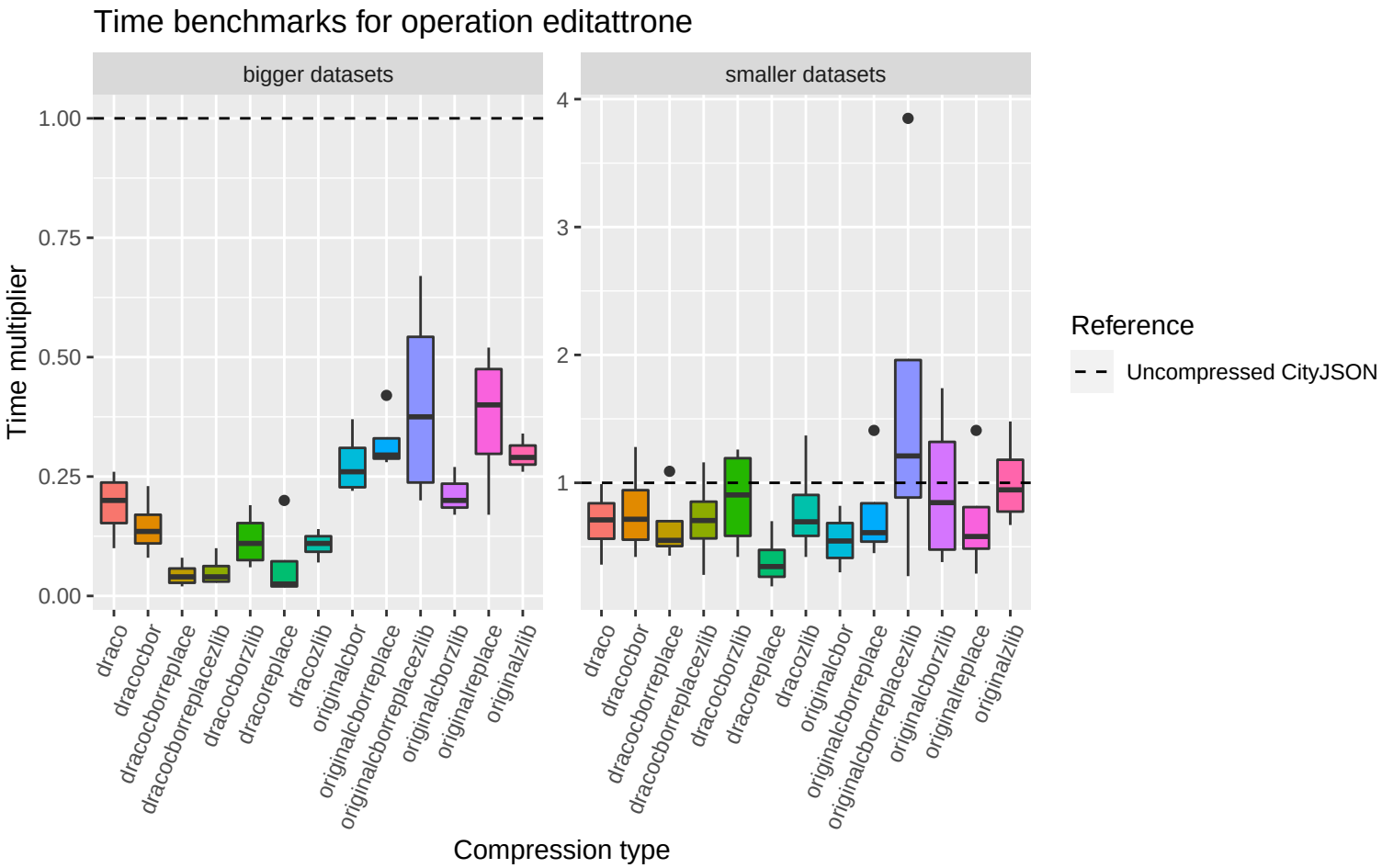

Figure 6.9: Boxplots of time performance of compressed datasets on editing attributes of one feature compared to uncompressed CityJSON.

All compression types generally perform better than expected, but the results show that the impact of data having to be compressed again is not big. This would be due to the longer time that it takes for uncompressed CityJSON to be transmitted.

In the graph showing the results with the smaller datasets, no clear trend is seen. Original-cbor-replace-zlib has the highest mean, but it does not have a significantly higher mean encoding time. Therefore I assume that it is because of the uncertainties in the benchmarking of small datasets, but it could also be that encoding it takes a longer time in JavaScript than in Python as is done in Section 6.1.

For the bigger ones, compression types with Draco are the fastest. That is because the geometries are not touched and they do not have to be decompressed again, whereas with original geometries it is, because the full file had to be decompressed. Despite compression types using replace only having to compress the array of attributes again rather than a whole JSON, they do not perform notably well.

Table 6.11: Mean performance with editattrone on bigger datasets

Compression type	mean
dracocborreplace	0.0450000
dracocborreplacezlib	0.0525000
dracoreplace	0.0675000
dracozlib	0.1075000
dracocborzlib	0.1175000
dracocbor	0.1450000
draco	0.1900000
originalcborzlib	0.2133333
originalcbor	0.2775000
originalzlib	0.2966667
originalcborreplace	0.3225000
originalreplace	0.3725000
originalcborreplacezlib	0.4050000

Table 6.12: Mean performance with editattrone on smaller datasets

Compression type	mean
dracoreplace	0.3950
originalcbor	0.5525
dracocborreplace	0.6550
draco	0.6925
dracocborreplacezlib	0.7125
originalreplace	0.7150
originalcborreplace	0.7700
dracocbor	0.7825
dracozlib	0.7950
dracocborzlib	0.8725
originalcborzlib	0.9525
originalzlib	1.0100
originalcborreplacezlib	1.6350

6.7.2 Editing attributes all features

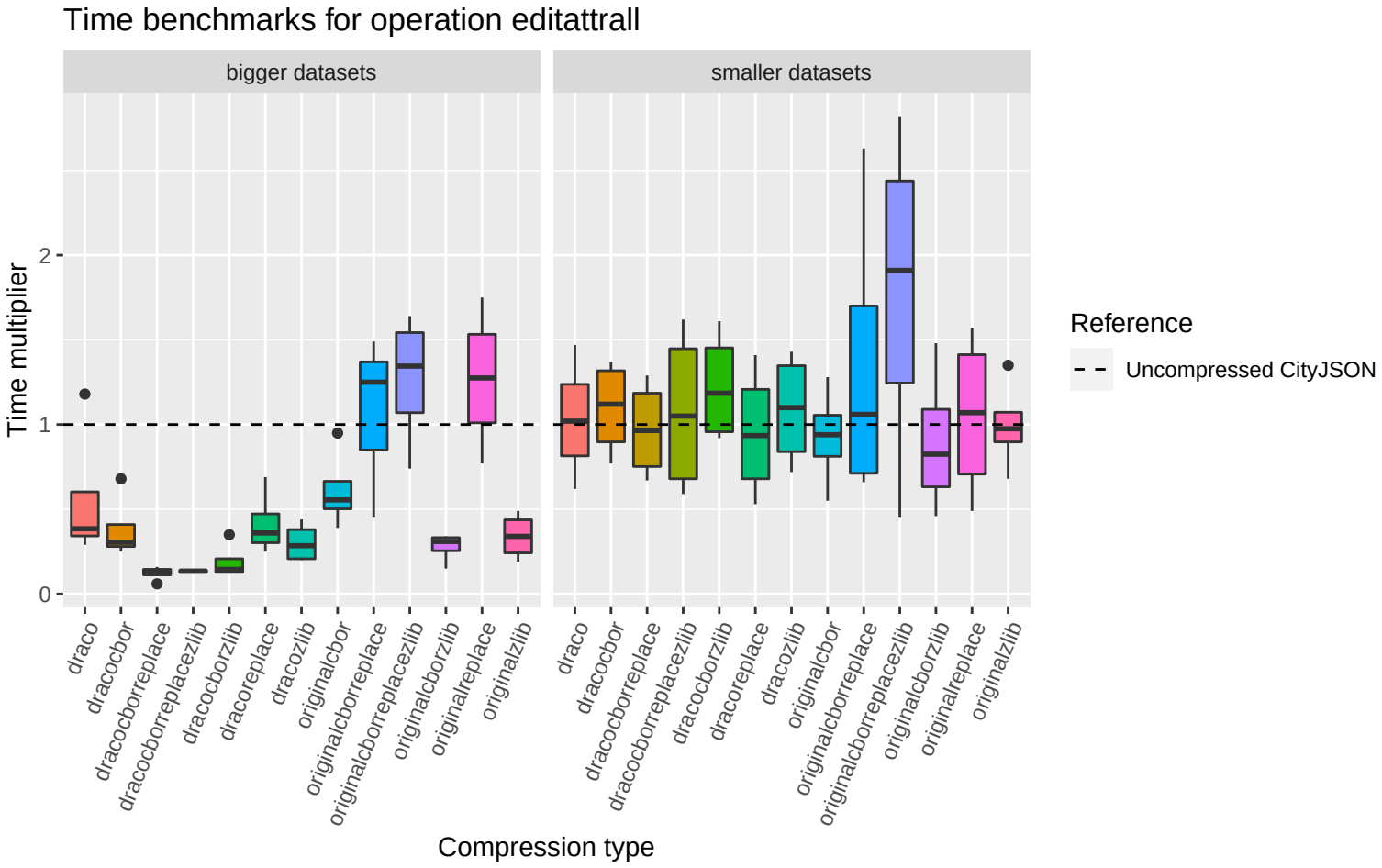

Figure 6.10: Boxplots of time performance of compressed datasets on editing attributes of all features compared to uncompressed CityJSON.

For the smaller datasets again original-cbor-replace-zlib performs remarkably worse than other compression types. Apart from that similar performances are seen and there is no compression type that would be better than uncompressed CityJSON in every case.

Original-cbor-zlib, original-cbor-replace-zlib, and original-replace have a mean above the baseline and are not suitable for this use case. This is despite the possibility of only compressing the array of attributes. Draco and original-cbor have outliers that are respectively above and just underneath the baseline, but generally they are still fine to use. The other ones however perform better, especially draco-cbor-zlib, draco-cbor-replace, and draco-cbor-replace-zlib, the latter two despite using the replace method which seems unsuitable when having original geometries. If not using Draco, original-cbor-zlib and original-zlib are good choices, of which the performance can be explained by them having a good encoding performance.

Table 6.13: Mean performance with editattrall on bigger datasets

Compression type	mean
dracocborreplace	0.122500
dracocborreplacezlib	0.132500
dracocborzlib	0.190000
originalcborzlib	0.277500
dracozlib	0.302500
originalzlib	0.340000
dracocbor	0.385000
dracoreplace	0.415000
draco	0.560000
originalcbor	0.612500
originalcborreplace	1.063333
originalcborreplacezlib	1.267500
originalreplace	1.267500

Table 6.14: Mean performance with editattrall on smaller datasets

Compression type	mean
originalcborzlib	0.8975
originalcbor	0.9275
dracoreplace	0.9525
dracocborreplace	0.9725
originalzlib	0.9950
draco	1.0325
originalreplace	1.0500
dracocborreplacezlib	1.0775
dracozlib	1.0875
dracocbor	1.0950
dracocborzlib	1.2250
originalcborreplace	1.3525
originalcborreplacezlib	1.7725

6.8 Editing (geometry)

6.8.1 Editing geometry one feature

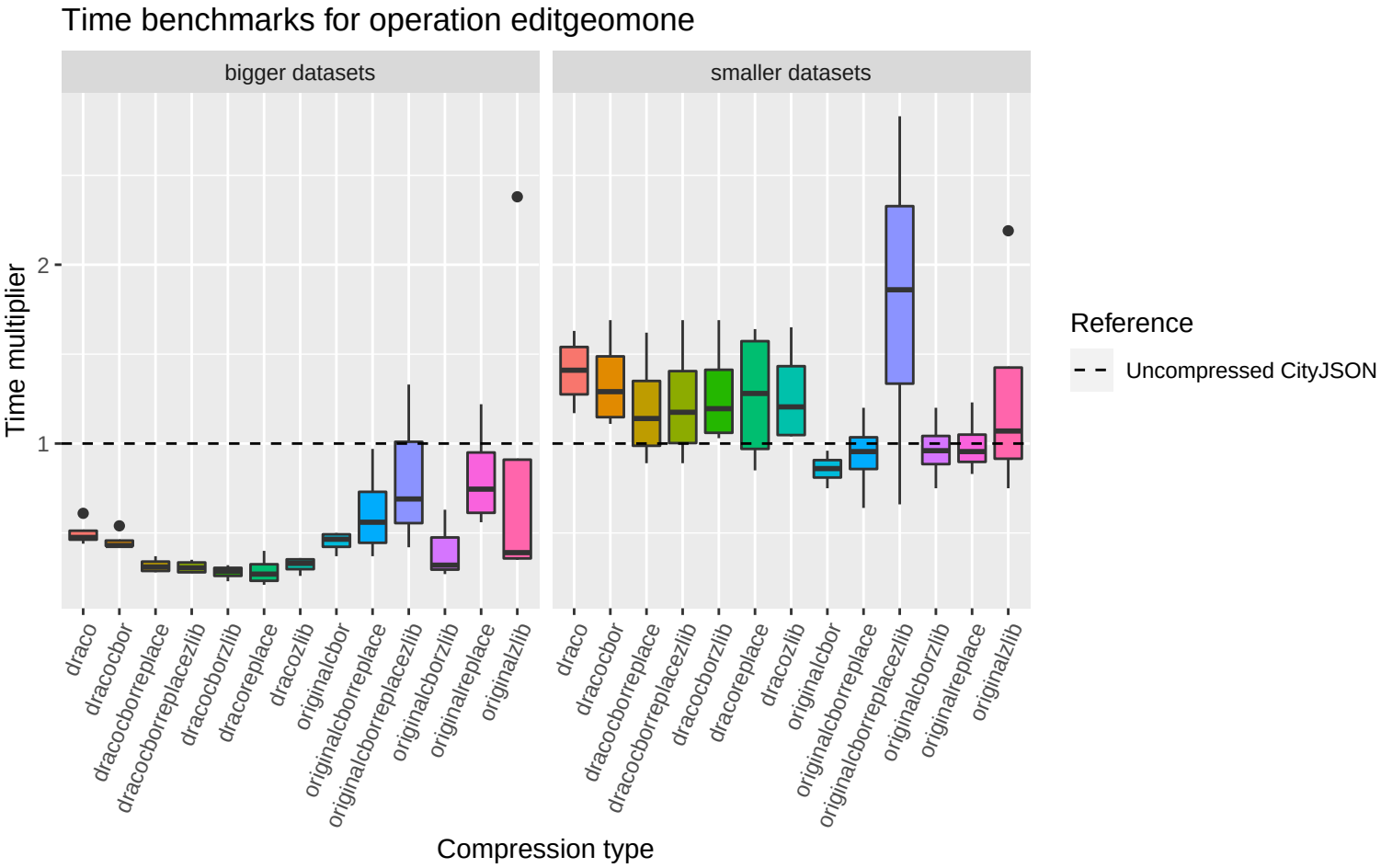

Figure 6.11: Boxplots of time performance of compressed datasets on editing geometry of one feature compared to uncompressed CityJSON.

Only original-cbor performs better than the baseline with all smaller datasets. The other compression types are not suitable to use in that case. Using Draco generally gives worse results, which is because all vertices of the Draco geometry need to be traversed and checked on the City Object ID that they belong to. As opposed to original geometries, where just one City Object can be taken from the JSON to edit.

Original-zlib has outliers for both bigger and smaller, which is the Zürich dataset that takes long to encode with this compression type (see Section 6.1).

With the bigger datasets it can be a good idea to use Draco, as compression types using it perform well. This seems surprising when looking at the encoding times graph, but it shows the performance in Python where all City Objects have to be traversed, converted to OBJ, and only then compressed by Draco. But for this use case the vertices of the Three.js mesh that the Draco geometry initially was decoded to can be edited and encoded again by Draco, which turns out to be faster.

Original-cbor-zlib is best if Draco is not used, and original-zlib and original-cbor are suitable choices as well. Despite attributes not having to be compressed, using a compression type with replace (and

original geometries) is not a good choice because of their relatively high file sizes and decoding times. When using Draco this is not the case because the mean file sizes with Draco and replace combined have a smaller difference in file size with other Draco compression types.

Table 6.15: Mean performance with editgeomone on bigger datasets

Compression type	mean
dracocborzlib	0.2800000
dracoreplace	0.2875000
dracocborreplacezlib	0.3100000
dracocborreplace	0.3175000
dracozlib	0.3200000
originalcborzlib	0.4066667
originalcbor	0.4500000
dracocbor	0.4550000
draco	0.5000000
originalcborreplace	0.6150000
originalcborreplacezlib	0.8133333
originalreplace	0.8175000
originalzlib	0.8775000

Table 6.16: Mean performance with editgeomone on smaller datasets

Compression type	mean
originalcbor	0.8575
originalcborreplace	0.9375
originalcborzlib	0.9675
originalreplace	0.9925
dracocborreplace	1.1975
dracocborreplacezlib	1.2325
dracoreplace	1.2625
originalzlib	1.2700
dracozlib	1.2750
dracocborzlib	1.2775
dracocbor	1.3450
draco	1.4050
originalcborreplacezlib	1.8025

6.8.2 Editing geometry all features

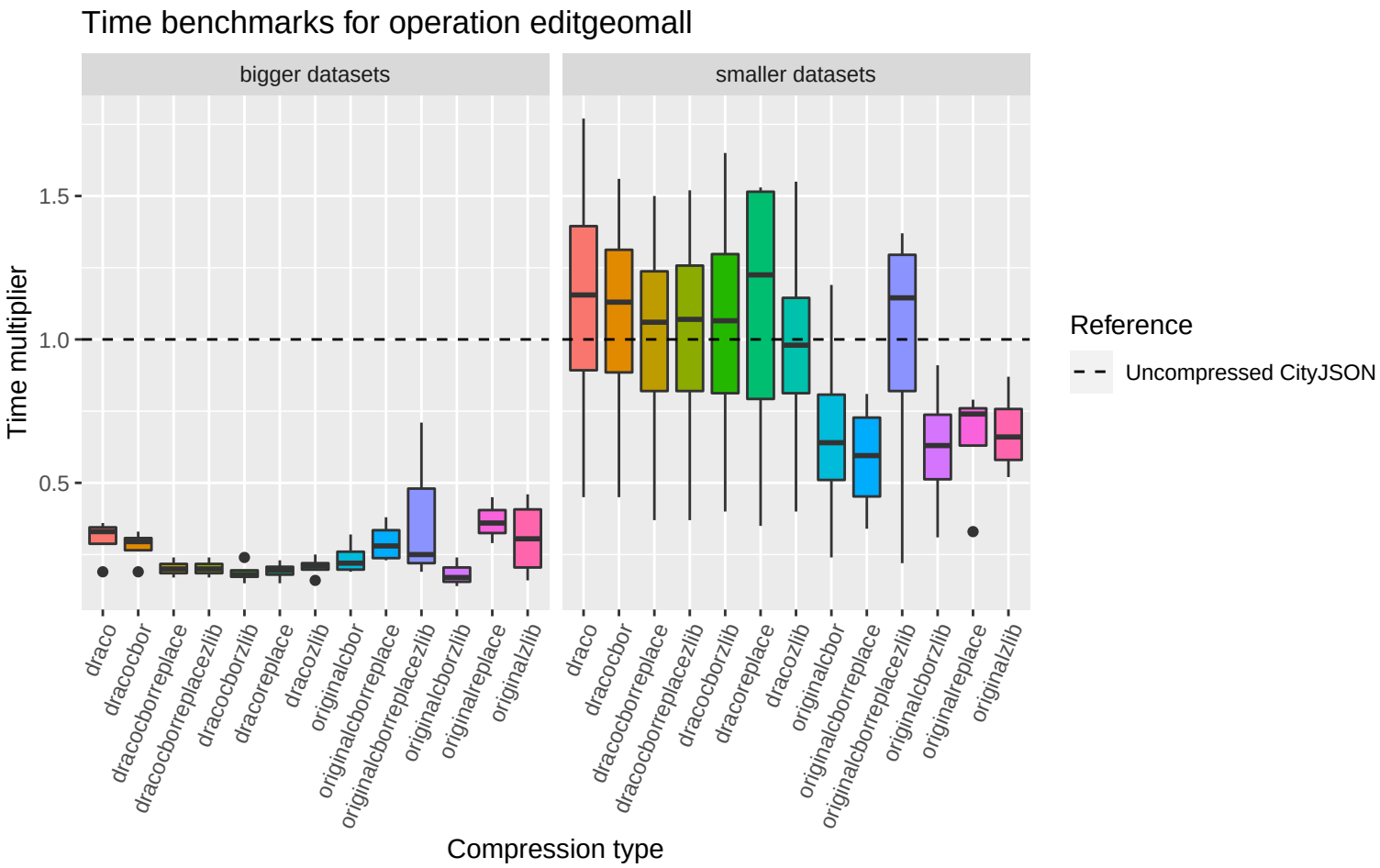

Figure 6.12: Boxplots of time performance of compressed datasets on editing geometry of all features compared to uncompressed CityJSON.

For the smaller datasets the results are similar to editing one geometry (Subsection 6.8.1). The largest difference is that compression types without Draco generally perform comparatively better, and they are almost always better than uncompressed CityJSON, besieds original-cbor-replace-zlib. Compressing smaller datasets with Draco takes longer (Section **??**), which is the most clear reason for this.

The comparative performance with bigger datasets is similar to the results of editing one feature's geometry as well, but the difference between Draco and non-Draco compression types is smaller. This is surprising because now all City Objects have to be traversed for the latter types, but with Draco all vertices already had to be iterated over, even when just wanting to edit one feature. Still using Draco has the best performance, except when compared to original-cbor-zlib which has the lowest average.

Table 6.17: Mean performance with editgeomall on bigger datasets

Compression type	mean
originalcborzlib	0.1833333
dracocborzlib	0.1875000
dracoreplace	0.1925000
dracocborreplace	0.2025000
dracocborreplacezlib	0.2025000
dracozlib	0.2075000
originalcbor	0.2375000
dracocbor	0.2775000
originalcborreplace	0.2925000
draco	0.3025000
originalzlib	0.3075000
originalreplace	0.3666667
originalcborreplacezlib	0.3833333

Table 6.18: Mean performance with editgeomall on smaller datasets

Compression type	mean
originalcborreplace	0.5850
originalcborzlib	0.6200
originalreplace	0.6500
originalcbor	0.6775
originalzlib	0.6775
originalcborreplacezlib	0.9700
dracozlib	0.9775
dracocborreplace	0.9975
dracocborreplacezlib	1.0075
dracocborzlib	1.0450
dracocbor	1.0675
dracoreplace	1.0825
draco	1.1325

6.9 Conclusion and recommendations

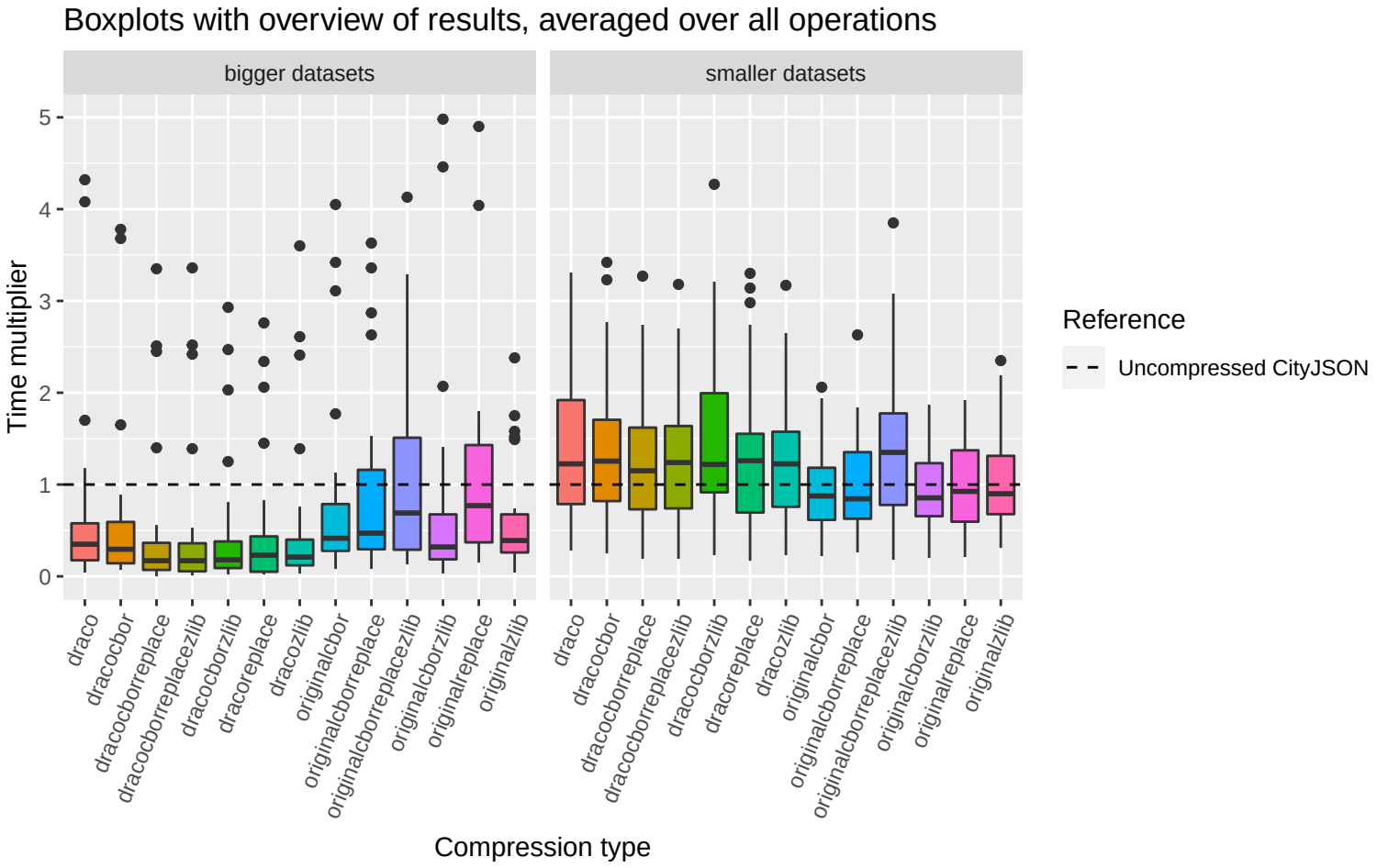

Figure 6.13: Overview of the performance of all compression types by dataset type. The results are averaged over every operation.

On figure 6.13, the difference between operations is removed here such that appropriate compression types for general purpose can be found. It is possible to pick a best one based on specific cases—for a dataset type and operation combination that is known beforehand. You can see in the presented charts which compression type you should then choose for the best performance. But sometimes a software developer would want to write flexible software that works well with a larger variety of use cases, rather than creating it for visualisation or editing only. Additionally, implementating multiple compression types for CityJSON will make it more complex as a file type, because all software that uses it would need to be able to handle all the compressed versions.

For these reasons this overview is given, such that compression types that perform well in most cases can be identified. The variation between operations can be big however—querying or buffering one object for example always took much longer with compressed datasets. This skews the results and the mean is the most important, not the size of the box plot. The plot's y-scale has been limited to a time multiplier of 5 as there are outliers higher than that, which makes the graph unreadable.

When using smaller datasets the benefits of using compression are always less than with larger ones, and often it is even a bad idea to use it with a general purpose. Especially using Draco is generally not beneficial. Original-cbor, original-cbor-replace, original-zlib, and original-cbor-zlib

are on average still faster than uncompressed CityJSON, but not much. There are however use cases where the compression of smaller datasets does yield good results as has been shown in this chapter.

But compression was already expected to work better with bigger datasets, and this is indeed the case—the time multiplier is lower for every compression type. In this case using Draco is worth it if high performance is important, but not necessarily if only Draco is used—in that case, original-cbor-zlib has a lower mean value. Adding further compression besides Draco gives even better results. Draco-cbor-replace and draco-cbor-replace-zlib are the best ones, but draco-cbor-zlib, draco-replace, and draco-zlib also do well. Disadvantages are however that Draco is harder to use, and the replace method as well.

If not using Draco, original-cbor, original-zlib, and original-cbor-zlib are good all-round choices. They have the additional advantage of being easy to implement as they do not use Draco or replace, and CBOR has the potential to be queried directly (as explained in Section 5.6.2) which could improve the results even further.

Concludingly, choosing one best compression type is hard. Because not one clearly superior option can be chosen, I think that it is best to find a balance between good performance and ease of implementation. In addition to that, it is most important that it works well with bigger datasets, because these have the largest performance issues and benefit more from compression. Not only relatively, but especially in absolute terms. It has to be kept in mind that the baseline for the performance of smaller datasets is already small. A time multiplier of 0.5 looks nice, but in the end you are often talking about milliseconds saved. For bigger datasets, this can be seconds to even minutes.

Considering this, I think that original-cbor, original-zlib, and original-cbor-zlib are all-round the best compression types. They improve the general performance as can be seen from their mean values in Figure 6.13. At the same time they are easy to implement, as both CBOR and zlib only need one line of code for encoding of decoding (see Section 5.4.1). Additionally they are widely supported and have libraries in many programming languages [CBOR, 2020; zlib, 2020].

task	method	delft	denhaag	hdb	montreal	newyork	RotterdamLoD1Lod2	tudnl3d	Zurich
bufferall	draco	0.82	0.68	0.09	1.04	0.11	1.29	0.04	0.11
bufferall	dracocbor	0.92	0.7	0.07	1	0.07	1.44	0.14	0.14
bufferall	dracocborreplace	0.96	0.66	0.12	1.01	0	1.28	0.03	0.09
bufferall	dracocborreplacezlib	0.83	0.64	0.05	1.12	0.01	1.51	0.03	0.04
bufferall	dracocborzlib	0.9	0.72	0.03	1.07	0.03	2.2	0.02	0.03
bufferall	dracoreplace	1.26	0.68	0.05	0.98	0.02	1.33	0.02	0.02
bufferall	dracozlib	0.87	0.64	0.04	1.13	0.14	1.4	0.03	0.05
bufferall	originalcbor	0.62	0.36	0.33	0.96	NA	0.87	NA	NA
bufferall	originalcborreplace	0.75	0.5	0.25	0.86	NA	0.59	NA	NA
bufferall	originalcborreplacezlib	0.7	0.47	0.29	1.14	NA	1.55	NA	NA
bufferall	originalcborzlib	0.68	0.41	0.28	0.86	NA	0.74	NA	NA
bufferall	originalreplace	0.83	0.49	0.37	1.91	NA	0.53	NA	NA
bufferall	originalzlib	0.66	0.31	0.25	0.89	NA	0.87	NA	NA
bufferone	draco	2.34	2.68	9.81	2.94	21.08	3.31	7.55	30.19
bufferone	dracocbor	2.34	2.77	8.76	3.42	18.69	3.23	7.4	27.6
bufferone	dracocborreplace	2.74	2.49	6.11	2.23	12.07	3.27	6.91	18.28
bufferone	dracocborreplacezlib	2.41	2.4	5.91	2.7	11.82	3.18	6.8	18.26
bufferone	dracocborzlib	4.27	2.61	5.88	2.95	11.62	3.21	5.8	15.89
bufferone	dracoreplace	2.74	3.3	6.66	2.39	8.14	2.98	5.01	12.91
bufferone	dracozlib	2.43	2.41	6.57	2.65	13.8	3.17	6.25	19.83
bufferone	originalcbor	1.31	1.56	7.61	1.94	14.21	2.06	8.15	24.18
bufferone	originalcborreplace	1.32	1.59	6.54	1.52	10.9	1.65	9.04	21.57
bufferone	originalcborreplacezlib	1.37	1.6	6.85	1.55	10.44	1.88	8.8	20.72
bufferone	originalcborzlib	1.24	1.37	4.46	1.87	6.13	1.66	4.98	11.69
bufferone	originalreplace	1.7	1.68	11.22	1.92	16.94	1.81	14.25	35.74
bufferone	originalzlib	1.19	1.3	16.62	1.59	7.56	1.99	5.86	13.31
editattrall	draco	1.16	0.88	0.36	0.62	1.18	1.47	0.29	0.41
editattrall	dracocbor	1.37	0.94	0.32	0.77	0.68	1.3	0.25	0.29
editattrall	dracocborreplace	1.15	0.67	0.13	0.78	0.16	1.29	0.14	0.06
editattrall	dracocborreplacezlib	1.39	0.71	0.14	0.59	0.12	1.62	0.14	0.13
editattrall	dracocborzlib	1.4	0.97	0.16	0.92	0.35	1.61	0.13	0.12
editattrall	dracoreplace	1.14	0.73	0.32	0.53	0.69	1.41	0.25	0.4
editattrall	dracozlib	1.32	0.88	0.2	0.72	0.44	1.43	0.21	0.36

(continued)

task	method	delft	denhaag	hdb	montreal	newyork	RotterdamLoD1Lod2	tudnl3d	Zurich
editattrall	originalcbor	1.28	0.9	0.39	0.55	0.95	0.98	0.57	0.54
editattrall	originalcborreplace	2.63	0.66	0.45	0.73	1.49	1.39	1.25	NA
editattrall	originalcborreplacezlib	2.31	1.51	0.74	0.45	1.18	2.82	1.51	1.64
editattrall	originalcborzlib	1.48	0.69	0.15	0.46	0.34	0.96	0.29	0.33
editattrall	originalreplace	1.57	0.78	0.77	0.49	1.75	1.36	1.46	1.09
editattrall	originalzlib	1.35	0.68	0.19	0.98	0.42	0.97	0.49	0.26
editattrone	draco	0.99	0.79	0.26	0.36	0.23	0.63	0.1	0.17
editattrone	dracocbor	1.28	0.83	0.23	0.42	0.15	0.6	0.08	0.12
editattrone	dracocborreplace	1.09	0.57	0.08	0.43	0.03	0.53	0.05	0.02
editattrone	dracocborreplacezlib	1.16	0.66	0.1	0.28	0.03	0.75	0.05	0.03
editattrone	dracocborzlib	1.26	1.17	0.14	0.42	0.19	0.64	0.06	0.08
editattrone	dracoreplace	0.7	0.4	0.2	0.19	0.02	0.29	0.03	0.02
editattrone	dracozlib	1.37	0.75	0.14	0.42	0.1	0.64	0.07	0.12
editattrone	originalcbor	0.82	0.64	0.29	0.3	0.23	0.45	0.22	0.37
editattrone	originalcborreplace	1.41	0.57	0.3	0.45	0.28	0.65	0.42	0.29
editattrone	originalcborreplacezlib	3.85	1.09	0.5	0.27	0.2	1.33	0.67	0.25
editattrone	originalcborzlib	1.74	1.18	0.2	0.38	0.17	0.51	0.27	NA
editattrone	originalreplace	1.41	0.61	0.52	0.29	0.34	0.55	0.46	0.17
editattrone	originalzlib	1.48	0.67	0.26	0.81	0.29	1.08	0.34	NA
editgeomall	draco	1.77	1.04	0.34	0.45	0.32	1.27	0.19	0.36
editgeomall	dracocbor	1.56	1.03	0.3	0.45	0.29	1.23	0.19	0.33
editgeomall	dracocborreplace	1.5	0.97	0.21	0.37	0.19	1.15	0.17	0.24
editgeomall	dracocborreplacezlib	1.52	0.97	0.21	0.37	0.19	1.17	0.17	0.24
editgeomall	dracocborzlib	1.65	0.95	0.18	0.4	0.18	1.18	0.15	0.24
editgeomall	dracoreplace	1.51	0.94	0.23	0.35	0.15	1.53	0.19	0.2
editgeomall	dracozlib	1.55	0.95	0.21	0.4	0.21	1.01	0.16	0.25
editgeomall	originalcbor	1.19	0.68	0.24	0.24	0.2	0.6	0.19	0.32
editgeomall	originalcborreplace	0.7	0.49	0.24	0.34	0.23	0.81	0.38	0.32
editgeomall	originalcborreplacezlib	1.37	1.02	0.25	0.22	0.19	1.27	0.71	NA
editgeomall	originalcborzlib	0.91	0.58	0.17	0.31	0.14	0.68	0.24	NA
editgeomall	originalreplace	0.75	0.73	0.36	0.33	0.29	0.79	0.45	NA
editgeomall	originalzlib	0.87	0.6	0.22	0.52	0.16	0.72	0.39	0.46
editgeomone	draco	1.63	1.31	0.47	1.17	0.61	1.51	0.44	0.48

(continued)

task	method	delft	denhaag	hdb	montreal	newyork	RotterdamLoD1Lod2	tudnl3d	Zurich
editgeomone	dracocbor	1.69	1.16	0.42	1.11	0.54	1.42	0.43	0.43
editgeomone	dracocborreplace	1.62	1.02	0.28	0.89	0.33	1.26	0.37	0.29
editgeomone	dracocborreplacezlib	1.69	1.04	0.28	0.89	0.33	1.31	0.35	0.28
editgeomone	dracocborzlib	1.69	1.07	0.23	1.03	0.3	1.32	0.32	0.27
editgeomone	dracoreplace	1.64	1.01	0.3	0.85	0.24	1.55	0.4	0.21
editgeomone	dracozlib	1.65	1.05	0.26	1.04	0.36	1.36	0.35	0.31
editgeomone	originalcbor	0.96	0.89	0.37	0.75	0.44	0.83	0.49	0.5
editgeomone	originalcborreplace	0.93	0.64	0.37	0.98	0.47	1.2	0.97	0.65
editgeomone	originalcborreplacezlib	2.16	1.56	0.69	0.66	0.42	2.83	1.33	NA
editgeomone	originalcborzlib	1.2	0.75	0.27	0.93	0.32	0.99	0.63	NA
editgeomone	originalreplace	0.99	0.83	0.56	0.92	0.86	1.23	1.22	0.63
editgeomone	originalzlib	1.17	0.75	0.36	2.19	0.35	0.97	2.38	0.42
queryall	draco	0.78	0.44	0.09	0.28	0.17	0.52	0.19	NA
queryall	dracocbor	0.79	0.43	0.07	0.25	0.14	0.48	0.18	NA
queryall	dracocborreplace	0.75	0.36	0.04	0.19	0.06	0.42	0.13	NA
queryall	dracocborreplacezlib	0.76	0.37	0.04	0.19	0.06	0.42	0.12	NA
queryall	dracocborzlib	0.78	0.37	0.03	0.23	0.04	0.43	0.1	NA
queryall	dracoreplace	0.74	0.35	0.05	0.17	0.02	0.38	0.08	NA
queryall	dracozlib	0.76	0.38	0.04	0.23	0.07	0.45	0.12	NA
queryall	originalcbor	0.64	0.44	0.08	0.22	0.16	0.38	0.33	NA
queryall	originalcborreplace	0.78	0.31	0.08	0.26	0.15	0.55	0.51	NA
queryall	originalcborreplacezlib	1.01	0.52	0.15	0.18	0.13	0.8	0.56	NA
queryall	originalcborzlib	0.95	0.3	0.03	0.2	0.06	0.35	0.17	NA
queryall	originalreplace	0.93	0.37	0.15	0.21	0.22	0.53	0.77	NA
queryall	originalzlib	0.65	0.31	0.04	0.34	0.07	0.38	0.29	NA
queryone	draco	2.04	1.56	4.32	2.15	6.17	2.44	1.7	4.08
queryone	dracocbor	1.97	1.58	3.78	1.98	5.44	2.33	1.65	3.68
queryone	dracocborreplace	1.86	1.37	2.51	1.62	3.35	2.07	1.4	2.45
queryone	dracocborreplacezlib	1.88	1.4	2.52	1.58	3.36	2.05	1.39	2.42
queryone	dracocborzlib	1.96	1.42	2.03	1.82	2.93	2.14	1.25	2.47
queryone	dracoreplace	1.9	1.36	2.76	1.54	2.34	1.99	1.45	2.06
queryone	dracozlib	1.93	1.41	2.41	1.83	3.6	2.19	1.39	2.61
queryone	originalcbor	1.18	1.26	3.11	1.38	4.05	1.36	1.77	3.42

(continued)

task	method	delft	denhaag	hdb	montreal	newyork	RotterdamLoD1Lod2	tudnl3d	Zurich
queryone	originalcborreplace	1.39	0.88	2.87	1.53	3.63	1.84	2.63	3.36
queryone	originalcborreplacezlib	2.03	1.77	5.62	1.13	3.29	1.79	2.55	4.13
queryone	originalcborzlib	1.29	0.85	1.2	1.23	1.41	1.24	0.91	2.07
queryone	originalreplace	1.28	1.03	4.9	1.31	5.89	1.92	4.04	5.51
queryone	originalzlib	1.21	0.88	1.49	2.35	1.75	1.36	1.52	1.58
visualise	draco	1.91	1.95	0.98	1.18	1.03	1.98	0.42	0.98
visualise	dracocbor	2.04	1.36	0.89	1.09	0.77	1.75	0.61	0.67
visualise	dracocborreplace	2.07	1.25	0.56	1.42	0.56	2.14	0.36	0.38
visualise	dracocborreplacezlib	2.06	1.37	0.53	1.42	0.42	2.51	0.37	0.5
visualise	dracocborzlib	2.1	2.16	0.69	0.94	0.81	3.18	0.41	0.48
visualise	dracoreplace	1.56	1.26	0.83	1.36	0.53	3.14	0.44	0.43
visualise	dracozlib	2.41	1.38	0.54	1.08	0.58	1.45	0.32	0.76
visualise	originalcbor	0.89	1.16	0.77	0.81	0.78	0.88	0.81	1.13
visualise	originalcborreplace	0.99	0.83	0.72	1	0.86	1.34	1.53	1.07
visualise	originalcborreplacezlib	1.02	1.6	1.23	0.71	0.65	3.08	1.62	1.88
visualise	originalcborzlib	1.75	0.85	0.38	0.76	0.37	0.84	0.71	0.64
visualise	originalreplace	1.6	1.01	1.43	0.79	1.21	1.3	1.63	1.8
visualise	originalzlib	1.56	0.88	0.45	1.65	0.63	0.91	0.74	0.72

Table 6.19: Table of full results, showing performance time multipliers. The minimum values per operation, compresstion type, and dataset are coloured green, the maximum values are in red.

7 Conclusion

7.1 Performed work and summary of achieved results

To do: finish this part. These things will change for P5 so I have not written a conclusion for it yet. It will answer the research question. For now, see Section 6.9

7.2 CityJSON on the web and compression

Here I freely discuss how I feel about the position of CityJSON for use on the web and what I think are the best options for actual implementation after this thesis. It is lacking references to other parts of the thesis.

I believe that it is best to use I3S or b3dm on the web when high visualisation performance is required. This is because of them being created with this specific purpose in mind. They already have tiling implemented. The large advantage that CityJSON has over these two is its ease of use. Therefore, I think this advantage should be preserved as much as possible and I3S and b3dm can be seen as complementary file formats rather than competitors.

CityJSON facilitates the creation of 3D models in a user-friendly way. This is (at least currently) easier than creating I3S and b3dm from scratch, since for instance cjio and 3dfier can be utilised to do create CityJSON datasets. CityJSON datasets can be inspected easily and the structure is more straightforward. Doing analysis with them is easier as well because of its simpler structure and geometries can be accessed and read easily, as opposed to having to work with the—often binary—glTF.

I3S and b3dm lack an extensive data model and need to be converted from other file types, which would then also have to be created first. This could be done from CityJSON as well when converters for it are written.

What I thus see as the ideal 3D model workflow is creating CityJSON datasets at first. It can be used to easily inspect the data, check its quality, and do analysis if wanted. Then if one is to publish the data for viewing on the web, it could be converted to one of the other formats.

That is if high performance is needed. CityJSON can be visualised on the web as well in a relatively simple manner as well, through converting the geometries to a three.js mesh. It definitely can be useful for web visualisation as well, let alone other web use such as dissemination.

CityJSON can thus perfectly exist together with I3S and b3dm as it has its own advantages. The focus should be kept on keeping it easy to use for regular users. Compression can make this more complicated, at least for developers, as there are extra steps to be taken to read or store a dataset. However, the use of massive datasets on the web can still be troublesome, which data compression could relieve. I therefore think that it is still feasible to implement compression functionalities for it, but with the strict requirement that it needs to be easy to use.

For me, the inclusion of Draco compression is out of the question. According to my experience it is not easy to work with, at least not when using it in liaison with 3D city models. This is because

Draco by default compresses all objects of an OBJ into one big object. It is possible to keep the separation between objects, but the JavaScript decoder is not able to handle this well out of the box and it will result in a less good compression. To make it somewhat work, I had to dive into the source code and make alterations which was time consuming. In addition to that, offline users would have to compile the Draco library on their computer in order to be able to read and write datasets that include Draco geometries, which is troublesome as well. And Draco is harder to use with any operation other than visualisation.

It is however a good idea to include the possibility for parsing to data to and from CBOR. To use it only a lightweight library is required, and it already exists for most languages. Only one simple line of code is necessary to encode or decode it from and to JSON. While not compressing the data size as far as other compression method combinations, it still always results in a smaller file size. On top of that, the results show good performance for it. It might also be possible to query it directly.

If data size is really important (for storage or just disseminating the data), the CBOR variant could be compressed with zlib which is still simple and shows good results. Only using zlib is possible too, but it can definitely not be queried directly, and it adds an extra step since you would have to decompress the zlib into JSON, and then parse the JSON. Whereas with CBOR you can just parse the CBOR immediately.

7.3 CityJSON + other efficient web use

- It is also possible to use 3DCityDB to serve data on the web instead of Flask.
- Streaming CityJSON and OGC API.
- Should tiling be implemented into CityJSON? I think not, for reasons that I mentioned in the previous section. It is better to invest in good conversion to B3DM and/or I3S.

7.4 Limitations

The choice of datasets could be improved, as I picked ones that were easy to find. It also seemed that the smaller datasets were more error prone in testing as the benchmarked times were smaller. This might have been the cause in results seeming unexplainable. The plan was to analyse the differences in performances between the selected datasets. This however made it too complex as there was too much data and too many differing results to explain. Some of them made few sense, so I had decided to average them into the two groups of smaller and larger datasets. For example... It would still be interesting to know about these differences because it was clear that it was not only the file size influencing the differences between performances of individual datasets.

Draco files are downloaded separately rather than loading them as embedded into compressed CityJSON files.

7.5 Future work (some could/will be investigated for P5)

- Encoding CityJSON geometries into Draco geometries can be improved by implementating direct encoding, rather than having to convert CityJSON to OBJ first.
- Investigate querying individual City Objects when using CBOR or the replace method.
- Compare CBOR to other binary JSON formats (BSON, MessagePack, UBJSON).

- Investigate how compression could work when using 3dcitydb to serve data.
- More thorough loss analysis
- Description of tools

A Datasets and characteristics

B Raw benchmarking results

C List of used tools

To do: explain used tools

D Reproducibility self-assessment

Bibliography

3D geoinformation research group at TU Delft. CityJSON/io: Python CLI to process and manipulate CityJSON files, 2019. URL https://github.com/cityjson/cjio.

Analytical Graphics, Inc. 3D Tiles Format Specification, 2018. URL https://github.com/AnalyticalGraphicsInc/3d-tiles/tree/master/specification#spatial-data-structures.

Analytical Graphics, Inc. Batched 3D Model, 2019. URL https://github.com/AnalyticalGraphicsInc/3d-tiles/blob/master/specification/TileFormats/Batched3DModel/README.md.

F. Biljecki, J. Stoter, H. Ledoux, S. Zlatanova, and A. Çöltekin. Applications of 3D city models: State of the art review. *ISPRS International Journal of Geo-Information*, 4(4):2842–2889, 2015.

F. Biljecki, H. Ledoux, and J. Stoter. Generating 3D city models without elevation data. *Computers, Environment and Urban Systems*, 64:1–18, July 2017. doi: 10.1016/j.compenvurbsys.2017.01.001.

F. Boersma. Online viewer for CityJSON files., 2019. URL https://github.com/fhb1990/CityJSON-viewer.

P. Bourke. RLE - Run Length Encoding, 1995. URL https://web.archive.org/web/20031020232320/http://astronomy.swin.edu.au/~pbourke/dataformats/rle/.

BSON. BSON homepage, 2019. URL http://bsonspec.org/.

Cable. Worldwide broadband speed league 2019, 2019. URL https://www.cable.co.uk/broadband/speed/worldwide-speed-league/.

CBOR. Concise Binary Object Representation (CBOR), 2013. URL https://tools.ietf.org/html/rfc7049.

CBOR. CBOR. RFC 7049 Concise Binary Object Representation, 2019. URL https://cbor.io/.

CBOR. Implementations, 2020. URL https://cbor.io/impls.html.

Cesium. Draco compressed meshes with glTF and 3D tiles, 2018. URL https://cesium.com/blog/2018/04/09/draco-compression/.

Cesium. 3D Tiles Overview, 2020. URL https://github.com/CesiumGS/3d-tiles/blob/master/3d-tiles-overview.pdf.

CityJSON. Cityjson specifications 1.0.1, 2019. URL https://www.cityjson.org/specs/1.0.1/.

CityJSON. A web viewer for CityJSON files, 2020. URL https://github.com/cityjson/ninja.

CityJSON. Datasets, 2020. URL https://www.cityjson.org/datasets/.

Dcoetzee. File:Huffman tree.svg, 2007. URL https://commons.wikimedia.org/wiki/File:Huffman_tree.svg.

M. Deering. Geometry compression. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 13–20, 1995.

S. Dehpour. Deep Difference and search of any Python object/data., 2020. URL https://github.com/seperman/deepdiff.

C. Dempsey. Web-based GIS, 2018. URL https://www.gislounge.com/web-based-gis/.

Esri. Esri Indexed 3d Scene Layer (I3S) and Scene Layer Package (*.slpk) Format Specification, 2019a. URL https://github.com/Esri/i3s-spec/blob/master/format/Indexed%203d%20Scene%20Layer%20Format%20Specification.md.

Esri. Scene Layers: Service and Package Standard, 2019b. URL https://github.com/Esri/i3s-spec.

A. Feldspar. An explanation of the deflate algorithm, 1997. URL http://www.zlib.net/feldspar.html.

R. T. Fielding. Representational State Transfer (REST), 2000. URL https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

Flask. Flask - web development, one drop at a time, 2020. URL https://flask.palletsprojects.com/en/1.1.x/.

F. Galligan. Draco Bitstream Specification, 2019. URL https://google.github.io/draco/spec/.

GeoJSON. GeoJSON. URL https://geojson.org/.

Google. mesh_prediction_scheme_parallelogram_encoder.h, 2018. URL https://github.com/google/draco/blob/master/src/draco/compression/attributes/prediction_schemes/mesh_prediction_scheme_parallelogram_encoder.h.

Google. mesh_sequential_encoder.h, 2019a. URL https://github.com/google/draco/blob/master/src/draco/compression/mesh/mesh_sequential_encoder.h.

Google. Draco 3D data compression, 2019b. URL https://google.github.io/draco/.

Google. Draco 3D data compression, 2019c. URL https://github.com/google/draco.

Google. Draco C++ decoder API, 2020a. URL https://github.com/google/draco/blob/master/src/draco/mesh/mesh.h.

Google. Draco JavaScript decoder API, 2020b. URL https://github.com/google/draco/blob/master/src/draco/javascript/emscripten/draco_web_decoder.idl.

Google. Mesh edgebreaker traversal predictive encoder, 2020c. URL https://github.com/google/draco/blob/master/src/draco/compression/mesh/mesh_edgebreaker_traversal_predictive_encoder.h.

HERE. harp.gl: 3d web map rendering engine, 2019. URL https://www.harp.gl/.

J. Hildebrand. Encode and decode CBOR documents, with both easy mode, streaming mode, and SAX-style evented mode., 2020. URL https://github.com/hildjj/node-cbor.

D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

Introducing JSON. Introducing JSON, 2019. URL https://www.json.org/json-en.html.

KeyCDN. What is latency and how to reduce it, 2018. URL https://www.keycdn.com/support/what-is-latency.

H. Ledoux. TU Delft campus dataset created with 3dfier. private communication, 2019.

H. Ledoux. CityJSON + RESTful access + streaming, 2020. URL https://github.com/hugoledoux/cityjson_ogcapi/blob/master/best-practice.md.

H. Ledoux, K. Ohori, K. Kumar, B. Dukai, A. Labetski, and S. Vitalis. CityJSON: A compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data, Software and Standards*, 4 (1):4, 2019.

N. Lohmann. JSON for Modern C++, 2020. URL https://github.com/nlohmann/json.

Mango. Web gis, 2017. URL https://mangomap.com/web-gis.

MessagePack. MessagePack, 2019. URL https://msgpack.org/.

Municipality of Rotterdam. Rotterdam multi-LoDs CityGML buildings, 2020. URL https://3d.bk.tudelft.nl/projects/geobim-benchmark/rotterdamlod12.html.

OGC API. Welcome To The OGC APIs, 2020. URL http://www.ogcapi.org/.

Open Geospatial Consortium. OGC API - Features, an overview, 2019a. URL https://github.com/opengeospatial/ogcapi-features/blob/master/overview.md.

Open Geospatial Consortium. OGC API - Features - Part 1: Core, 2019b. URL http://docs.opengeospatial.org/is/17-069r3/17-069r3.html#_items_.

Open Geospatial Consortium. OGC API - Features, 2019c. URL https://www.opengeospatial.org/standards/ogcapi-features.

Open Geospatial Consortium. OGC API - Features, 2020. URL https://github.com/opengeospatial/ogcapi-features.

M. Reddy. Object Files (.obj). URL http://www.martinreddy.net/gfx/3d/OBJ.spec.

J. Rossignac. 3D compression made simple: Edgebreaker with ZipandWrap on a corner-table. In *Proceedings International Conference on Shape Modeling and Applications*, pages 278–283. IEEE, 2001.

J. Rossignac, A. Safonova, and A. Szymczak. Edgebreaker on a corner table: A simple technique for representing and compressing triangulated surfaces. In *Hierarchical and geometrical methods in scientific visualization*, pages 41–50. Springer, 2003.

S. Sanfilippo. smaz, 2019a. URL https://github.com/antirez/smaz.

S. Sanfilippo. smaz, 2019b. URL https://github.com/antirez/smaz/blob/master/smaz.c.

K. Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.

Selenium. Selenium Projects, 2020. URL https://www.selenium.dev/projects/.

Socket.io. Socket.io 2.0 is here, 2020. URL https://socket.io/.

T. Suel. Delta compression techniques., 2019.

G. Taubin, W. P. Horn, F. Lazarus, and J. Rossignac. Geometry coding and vrml. *Proceedings of the IEEE*, 86(6):1228–1243, 1998.

Technische Universität München. 3dcitydb database, 2020. URL https://www.3dcitydb.org/3dcitydb/.

The Linux Information Project. Binary File Definition, 2006. URL http://www.linfo.org/binary_file.html.

Three.js. three.js, 2020a. URL https://threejs.org/.

Three.js. Mesh, 2020b. URL https://threejs.org/docs/#api/en/objects/Mesh.

Three.js. BufferGeometry, 2020c. URL https://threejs.org/docs/#api/en/core/BufferGeometry.

C. Touma and C. Gotsman. Triangle mesh compression. In *Proceedings of the Graphics Interface 1998 Conference, June 18-20, 1998, Vancouver, BC, Canada*, pages 26–34, June 1998. URL http://graphicsinterface.org/wp-content/uploads/gi1998-4.pdf.

UBJSON. Universal binary json specification: The universally compatible format specification for binary json., 2020. URL http://ubjson.org/.

WebGL. Getting Started, 2011. URL https://www.khronos.org/webgl/wiki/Getting_Started.

S. Yura. CBOR bindings for Python with (at the moment) simple streaming interface , 2016. URL https://github.com/funny-falcon/flunn.

I. Zderadicka. Comparison of json like serializations – json vs ubjson vs messagepack vs cbor, 2017. URL http://zderadicka.eu/comparison-of-json-like-serializations-json-vs-ubjson-vs-messagepack-vs-cbor/.

zlib. zlib - A Massively Spiffy Yet Delicately Unobtrusive Compression Library, 2020. URL https://zlib.net/.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class `scrbook`. The main font is Palatino.