MSc thesis in Geomatics

FlatCityBuf: a new cloud-optimised CityJSON format

Hidemichi Baba

June 2025

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

The work in this thesis was carried out in the:

3D geoinformation group
Delft University of Technology

Supervisors:	Asso.prof.dr. Hugo Ledoux
	Dr. Ravi Peters
Co-reader:	Assi.prof.dr. Martijn Meijers

Abstract

Standardizing data formats for 3D city models is crucial for semantically storing real-world information as permanent records. CityJSON is a widely adopted OGC standard format for this purpose, and its variant, CityJSON Text Sequences, decomposes large city objects into line-separated objects to enable streaming processing of 3D city model data. However, the shift towards cloud-native environments and the increasing demand for handling massive datasets necessitate more efficient data processing methods both system-wide and on the web. While optimized data formats such as PMTiles, FlatBuffers, Mapbox Vector Tiles, and Cloud Optimized GeoTIFF have been proposed for vector and raster data, options for 3D city models remain limited. This research aims to explore optimized data formats for CityJSON tailored for cloud-native processing and evaluate their performance and use cases. Specifically, the study will implement FlatBuffers for CityJSON, incorporating features like spatial indexing, spatial sorting, indexing with attribute values, and partial fetching via HTTP Range requests. The methodology includes designing a complete binary representation of the CityJSON standard using FlatBuffers, conducting a comprehensive review of existing performance-optimized formats, and benchmarking their performance. Successful implementation of this research will enable end-users to download arbitrary extents of 3D city models efficiently. For developers, the optimized format will allow for single-file containment of entire areas of interest, simplification of cloud architecture, and accelerated processing by software applications. Ultimately, this work will improve the scalability and usability of 3D city models in cloud environments, supporting advanced urban planning and smart city initiatives.

Acknowledgements

This thesis marks the culmination of my journey in the MSc Geomatics program, which would not have been possible without the support and guidance of many individuals and institutions.

First and foremost, I would like to express my sincere gratitude to my primary supervisor, Dr. Hugo Ledoux, for his exceptional guidance, unwavering patience, and continuous encouragement throughout this research. His deep expertise in 3D city models and geospatial data formats has been invaluable, and his critical insights significantly shaped the direction and quality of this work.

I am equally grateful to my secondary supervisor, Dr. Ravi Peters, who made preliminary work on this project and whose pragmatic approach to problem-solving helped me tackle numerous challenges in the development of FlatCityBuf. His constructive feedback consistently pushed me to refine both my ideas and their implementations.

My heartfelt thanks extend to Dr. Martijn Meijers, who served as co-reader for this thesis. His thoughtful comments and perspectives contributed substantially to improving the final manuscript.

I would like to acknowledge Delft University of Technology, particularly the Faculty of Architecture and the Built Environment and the MSc Geomatics program, for providing an exceptional academic environment and resources. The administrative and technical staff deserve special mention for their consistent support throughout my studies.

Beyond academia, I am grateful to my colleagues at Eukarya Inc., who showed remarkable understanding and flexibility as I balanced professional responsibilities with academic pursuits. Their support and accommodation made it possible for me to pursue both paths simultaneously.

My deepest appreciation goes to my family for their unconditional love, encouragement, and belief in me. Their unwavering support has been my anchor throughout this journey, especially during challenging periods.

Finally, a special acknowledgment to Chiharu, my partner, whose patience, understanding, and constant encouragement have been my greatest source of strength. 智春、研究が大変なときも疲れたときもいつも陰ながら応援してくれて本当にありがとう。智春の支えがあって、今日まで頑張ってくることができました。

To everyone who has been part of this journey—thank you.

Hidemichi Baba
Delft, May 26, 2025

. . .

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

1. Introduction

1.1. Problem Statement

Three-dimensional (3D) city models have evolved beyond mere visualisation tools to become fundamental components in diverse application domains. As demonstrated by Biljecki et al. [2015], these models now serve essential functions in urban planning, environmental simulation, emergency response, and numerous other fields, highlighting their critical role in urban environment representation and analysis. The widespread adoption of these models is evidenced by significant national initiatives, such as the Netherlands' comprehensive 3D building database [Peters et al., 2022], Japan's urban digital twin project [PLATEAU, 2020], US's Open City Model [BuildZero.Org, 2025], and Switzerland's SwissBUILDINGS3D [Swiss Federal Office of Topography, 2024]. To support this adoption, the CityGML Conceptual Model [OGC, 2019b] provides a standardised framework for comprehensive urban environment representation, with implementations including CityGML, CityJSON, and 3DCityDB achieving substantial adoption in both research and practical applications.

The transition towards cloud-native geographic information systems (GIS) introduces specific technical requirements for 3D city model implementations. These requirements encompass scalable processing capabilities, efficient data transfer mechanisms, optimised query performance, and distributed access protocols [Cloud-Native Geospatial Foundation, 2023]. While CityGML and CityJSON provide comprehensive data models, their text-based implementations often result in slower processing times and increased memory consumption. CityJSON Text Sequences (CityJSONSeq) was developed as a variant of CityJSON to enable streaming processing of 3D city model data, but it still inherits the performance limitations of text-based formats. Furthermore, although cloud-optimised geospatial formats have emerged to address these challenges, they primarily focus on two-dimensional data, leaving a gap in efficient cloud-native solutions for 3D city models. This gap necessitates the development of specialised data formats that can effectively operate within cloud computing environments while maintaining the semantic richness of 3D city models.

1.2. Research Objectives

This research investigates the application of efficient data serialisation formats for 3D city models, specifically examining the potential of FlatBuffers [Google, 2014a] as an encoding mechanism for CityJSONSeq.

While CityJSONSeq was designed to enable streaming processing of 3D city models, its text-based format results in suboptimal read performance and lacks efficient indexing strategies for feature querying.

The main research question is: "How can the CityJSONSeq encoding be optimised for faster access to features, lower memory consumption, and flexible feature querying in web environments?"

To answer this question, the following sub-questions are addressed:

- What data schema of FlatBuffers is most suitable for encoding all components of CityJSONSeq, including geometry templates, materials, extensions, attributes, and CityJSONFeature objects?
- How can feature querying with both spatial and attribute-based operations be achieved within logarithmic time complexity?
- How can subsets of data be retrieved efficiently over the web while maintaining simplicity of server architecture and handling high concurrent request loads that typically challenge traditional database-backed servers?

The following aspects, while relevant to the overall system performance, are not primary focus areas:

- File size reduction, provided the client can efficiently fetch or partially retrieve required data
- Data update and deletion speed, as retrieval speed takes precedence

The resulting proposed data format, called FlatCityBuf, leverages FlatBuffers' exceptional read performance efficiency and random access capabilities, which are essential for fetching subsets of data without processing the entire file—these advantages will be explained in detail in section 2.8.

The proposed methodology combines FlatBuffers' efficient binary serialisation with HTTP Range Requests, enabling partial data retrieval over the web while facilitating serverless architectures for enhanced scalability.

The investigation aims to address the aforementioned cloud-native requirements while maintaining the semantic richness of CityJSON's data model.

Notably, the research prioritises read performance over update capabilities, as read operations predominate in typical use cases.

Furthermore, while file size optimisation remains relevant, it is considered secondary to query efficiency and partial data accessibility.

The technical implementation strategy and preliminary findings are detailed in chapter 4, while the evaluation of cloud-optimised formats is presented in section 3.1.

1.3. Scope of the Research

Since the primary focus of the research is to explore the potential of FlatBuffers as an encoding mechanism for CityJSONSeq and achieve faster and flexible feature querying, the scope of the research is limited to the following:

- Define the data specification of the proposed new encoding format, FlatCityBuf.
- Implement a Rust library for encoding and decoding operations for FlatCityBuf.

- Support spatial querying and attribute-based querying to achieve log-time complexity for feature retrieval.
- Demonstrate how the proposed encoding format can be used over the web with HTTP Range Requests.
- Evaluate the performance of the proposed encoding format compared to the other encoding formats.

On the other hand, the following aspects are considered secondary and are not within the scope of the research:

- Implementing the library with other programming languages than Rust such as Python or JavaScript.
- Exploring other read efficient serialisation formats than FlatBuffers.
- Optimising the encoding format for write operations.

1.4. Structure of the Thesis

This thesis is organised into the following chapters:

chapter 3 provides a review of the relevant literature, focusing on cloud-optimised geospatial formats and serialisation frameworks.

It presents a comprehensive analysis of existing cloud-optimised geospatial formats and their characteristics.

chapter 2 establishes the fundamental knowledge of serialisation formats, algorithms, and indexing strategies necessary for understanding the proposed solution.

chapter 4 details the methodology used to achieve the research objectives.

It explains the data specification of FlatCityBuf and other technical components designed to address the research questions.

chapter 5 presents the research findings, including file size comparisons, local benchmark results, and web-based performance evaluations.

chapter 6 elaborates on the results, discusses the implications of the research, and identifies potential applications and limitations.

chapter 7 concludes the thesis by answering the research questions and summarising the contributions of the research.

2. Theoretical background

2.1. Cloud-native

add here

2.2. Cache line

add here

2.3. Serialisation and Deserialisation

Before discussing the specific techniques used in the FlatCityBuf format, it is important to understand the general principles of serialisation and deserialisation.

The terminology for data conversion processes varies across different programming ecosystems. Terms such as serialisation, pickling, marshalling, and flattening are often used interchangeably, though with subtle differences depending on the context. Standard C++ Foundation [2025] describes it from an object-oriented perspective as converting objects to a storable or transmittable format and later reconstructing them. Python Software Foundation [2025] refers to this process as "pickling" in the Python ecosystem. For clarity in this thesis, we adopt the definition provided by Viotti and Kinderkhedia [2022]:

> "Serialization is the process of translating a data structure into a bit-string (a sequence of bits) for storage or transmission purposes."

Deserialisation is the reverse process of serialisation, where the bit-string is converted back into the original data structure.

2.4. Zero-copy

Zero-copy is a technique used to avoid copying data from one memory location to another. While, the term "Zoro-copy" is used in many contexts of computer science, Song and Alves-Foss [2012] and Bröse [2008] provide a more detailed explanation of the concept.

2.4.1. Typical Data Transfer with read/write

In conventional I/O operations, data typically traverses multiple memory regions, each requiring a separate copy operation:

- Data is copied from storage devices into kernel buffer cache
- From kernel buffer, data is copied to user-space application buffers
- For network transmission, data may be copied again to network buffers

This multi-stage copying introduces significant overhead, particularly for large datasets or high-throughput applications. Each copy operation consumes CPU cycles, memory bandwidth, and increases latency [Song and Alves-Foss, 2012]. For applications working with large 3D city models, this overhead can substantially degrade performance.

Todo: improve here

2.4.2. Zero-copy Techniques

Zero-copy approaches optimize this data path by eliminating unnecessary copy operations. While "zero-copy" as a term suggests complete elimination of copying, in practice, different techniques achieve varying degrees of copy reduction:

- **Memory-mapped Input/Output from/to disk or network (I/O)**: Maps files directly into process address space, allowing direct access without explicit read/write operations
- **Direct I/O**: Bypasses the kernel buffer cache for specific workloads
- **Scatter-gather I/O**: Reads data directly into discontiguous memory regions
- **Shared memory**: Provides common address space for inter-process communication
- **In-place parsing**: Processes data structures without creating intermediate copies

Modern serialization formats like FlatBuffers implement zero-copy through carefully designed memory layouts that allow direct access to serialized data without requiring a separate deserialization step. This approach is particularly valuable for geospatial applications that routinely handle large datasets.

2.5. Endianness

Endianness (or "byte-order") refers to the order in which bytes are stored in memory when representing multi-byte values. The terminology was introduced by Cohen [1981].

In computing, endianness becomes significant when multi-byte data types (such as 16-bit integers or 32-bit floats) must be stored in memory or transmitted across networks. There are two primary byte ordering systems:

- **Little-endian**: Stores the least significant byte at the lowest memory address, followed by increasingly significant bytes. This is the ordering used by Intel processors that dominate desktop and server computing. For example, the 32-bit integer `0x12345678` would be stored in memory as `0x78`, `0x56`, `0x34`, `0x12`.

- **Big-endian**: Stores the most significant byte at the lowest memory address. This approach is often called “network byte order” because Internet protocols typically require data to be transmitted in big-endian format. For example, the same 32-bit integer `0x12345678` would be stored as `0x12`, `0x34`, `0x56`, `0x78`.

A useful analogy is date notation: little-endian resembles the European date format (31 December 2050), while big-endian resembles the ISO format (2050-12-31), with the most significant part (year) first [Mozilla Foundation, 2025].

2.6. Binary Search

Binary search is a fundamental algorithm for finding elements in a sorted array. The classic implementation follows a simple approach: compare the search key with the middle element of the array, then recursively search the left or right half depending on the comparison result [Slotin, 2021a].

Algorithm 2.1: Classic Binary Search

Input: A sorted array, a target value, left and right bounds
Output: The index where the target value should be inserted

```
while left < right do
    mid ← (left + right) / 2;
    if array[mid] ≥ target then
        right ← mid;
    else
        left ← mid + 1;
return left
```

The time complexity of binary search is logarithmic—the height of the implicit binary search tree is $\log_2(n)$ for an array of size n. While this is theoretically efficient, the actual performance suffers when implemented on modern hardware due to memory access patterns. Each comparison requires the processor to fetch a new element, potentially causing a cache miss. In the worst case, the number of memory read operations will be proportional to the height of the tree, with each read potentially requiring access to a different cache line or disk block [Slotin, 2021a].

This inefficiency is particularly problematic when binary search is implemented on external memory or over HTTP, where each access incurs significant latency. The sorted array representation with binary search does not take advantage of CPU cache locality, as consecutive comparisons frequently access distant memory locations.

2.6.1. Eytzinger Layout

While preserving the same algorithmic idea as binary search, the Eytzinger layout (also known as a complete binary tree layout or level-order layout) rearranges the array elements to match the access pattern of a binary search [Slotin, 2021a]. Instead of storing elements in sorted order, it places them in the order they would be visited during a level-order traversal of a complete binary tree.

This layout significantly improves memory access patterns. When the array is accessed in the sequence of a binary search operation, adjacent accesses often refer to elements that are in the same or adjacent cache lines. This spatial locality enables effective hardware prefetching, allowing the CPU to anticipate and load required data before it is explicitly accessed, thus reducing latency [Slotin, 2021a].

The Figure 2.1 shows how the layout looks when applied to binary search. The Figure 2.2 shows that the algorithm starts from the first element and then jumps to either $2k$ or $2k+1$ depending on the comparison result.

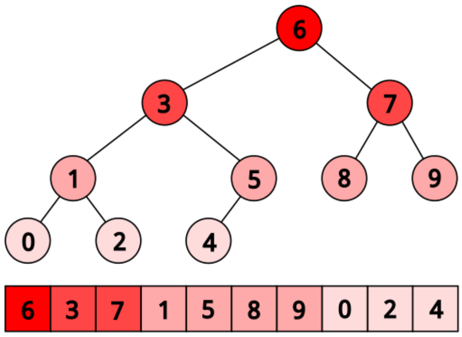


Figure 2.1.: Eytzinger layout as conceptual representation as tree and actual data layout (derived from Slotin [2021a])

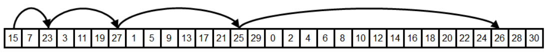


Figure 2.2.: Binary search traversal pattern in Eytzinger layout (derived from Slotin [2021a])

2.7. S+Tree

2.7.1. B-Tree/B+Tree Layout

While the Eytzinger layout improves cache utilization for binary search, the number of memory read operations remains proportional to the height of the tree—$\log_2(n)$ for n elements. This is still suboptimal for large datasets, especially when the access pattern involves disk I/O or remote data access [Slotin, 2021b].

B-Trees and their variants address this limitation by storing multiple keys in each node, effectively reducing the height of the tree. In a B-Tree of order k (where each node can contain up to $k-1$ keys), the height of the tree is reduced from $\log_2(n)$ to $\log_k(n)$. This represents a reduction factor of $\log_k$ / $\log_2 = \log_2(k)$ times compared to a binary search tree.

The key insight is that fetching a single node still takes roughly the same time regardless of whether it contains one key or multiple keys, as long as the entire node fits into a single memory block or disk page. By packing multiple keys into each node, B-Trees significantly reduce the number of disk or memory accesses required to locate an element.

B+Trees are a variant of B-Trees specifically optimized for range queries and sequential access patterns. In a B+Tree:

- Internal nodes contain up to B keys that serve as routing information, with each key associated with one of the $(B+1)$ pointers to child nodes. Each key at position i represents the smallest key in the subtree pointed to by the $(i+1)$-th child pointer.
- Leaf nodes store the actual data with up to B key-value pairs and include a pointer to the next leaf node, enabling efficient sequential traversal for range queries.

This linked structure of leaf nodes enables B+Trees to efficiently support range queries by traversing from one leaf to the next without needing to return to higher levels of the tree.

2.7.2. S+Tree

The S+Tree approach, introduced by Algorithmica [Slotin, 2021b], builds upon the B+Tree concept but is specifically designed for static datasets where the tree structure never changes after construction. Unlike traditional B+Trees that use explicit pointers between nodes, the Static B+Tree uses an implicit structure where child positions are calculated mathematically.

This is possible because:

- The tree is constructed once and never modified (static)
- The number of elements is known in advance
- The tree can be maximally filled with no empty slots
- Child positions follow a predictable pattern based on the block size

For a S+Tree with block size B, a node with index k has its children at indices calculated by a simple formula: $\text{child}_i(k) = k \cdot (B+1) + i + 1$ for $i \in [0, B]$ [Slotin, 2021b]. This eliminates the need to store and fetch explicit pointer values, further reducing memory usage and improving cache efficiency.

The S+Tree layout aligns with modern hardware characteristics where:

- The latency of fetching a single byte is comparable to fetching an entire cache line (64 bytes)
- Disk and network I/O operations have high initial latency but relatively low marginal cost for additional bytes
- CPU cache lines typically hold multiple array elements (e.g., 16 integers in a 64-byte cache line)

By loading a block of B elements at once and performing a local search within that block, S+Trees reduce the total number of cache misses or disk accesses to $\log_B(n)$ instead of $\log_2(n)$—a significant reduction for large datasets.

The S+Tree layout achieves up to 15× performance improvement over standard binary search implementations while requiring only 6-7% additional memory [Slotin, 2021b]. This makes it particularly valuable for applications that perform frequent searches on large, relatively static datasets, especially when accessed over high-latency connections. For more detailed implementation strategies of S+Tree, Koekamp [2024] provides comprehensive explanations and practical considerations.

2.8. FlatBuffers Framework

FlatBuffers, developed by Google [2014a], is a cross-platform serialisation framework designed specifically for performance-critical applications with a focus on memory efficiency and processing speed. Unlike traditional serialisation approaches, FlatBuffers implements a zero-copy deserialisation mechanism that enables direct access to serialised data without an intermediate parsing step [Google, 2014b], as discussed in section 2.4. This characteristic is particularly advantageous for large geospatial datasets where parsing overhead can significantly impact performance.

2.8.1. Schema-Based Serialisation

FlatBuffers employs a strongly typed, schema-based approach to data serialisation. The workflow involves:

1. Definition of data structures in schema files with the `.fbs` extension
2. Compilation of schema files using the FlatBuffers compiler (`flatc`)
3. Generation of language-specific code for data access
4. Implementation of application logic using the generated code

This schema-first approach enforces data consistency and type safety, which is essential to be processed in various programming languages and environments. The generated code provides memory-efficient access patterns to the underlying binary data without requiring full deserialisation. FlatCityBuf utilises this capability to achieve a balance between parsing speed and storage efficiency.

The FlatBuffers compiler supports code generation for multiple programming languages, including C++, Java, C#, Go, Python, JavaScript, TypeScript, Rust, and others, facilitating cross-platform interoperability [Google, 2024b]. This extensive language support enables developers to work with FlatBuffers data in their preferred environment. For FlatCityBuf, Rust was selected as the primary implementation language due to its performance characteristics and memory safety guarantees.

2.8.2. Data Type System

FlatBuffers provides a comprehensive type system that balances efficiency and expressiveness [Google, 2024a]:

- **Tables**: Variable-sized object containers that support:
  - Named fields with type annotations
  - Optional fields with default values
  - Schema evolution through backward compatibility
  - Non-sequential field storage for memory optimisation
- **Structs**: Fixed-size, inline aggregates that:
  - Require all fields to be present (no optionality)

  - Are stored directly within their containing object
  - Provide faster access at the cost of schema flexibility
  - Optimise memory layout for primitive types

- **Scalar Types**:
  - 8-bit integers: `byte` (int8), `ubyte` (uint8), `bool`
  - 16-bit integers: `short` (int16), `ushort` (uint16)
  - 32-bit values: `int` (int32), `uint` (uint32), `float`
  - 64-bit values: `long` (int64), `ulong` (uint64), `double`

- **Complex Types**:
  - `[T]`: Vectors (single-dimension arrays) of any supported type
  - `string`: UTF-8 or 7-bit ASCII encoded text with length prefix
  - References to other tables, structs, or unions

- **Enums**: Type-safe constants mapped to underlying integer types
- **Unions**: Tagged unions supporting variant types

2.8.3. Schema Organisation Features

In addition to the data type system, FlatBuffers provides several key features for organising complex schemas:

- **Namespaces** (`namespace FlatCityBuf;`) create logical boundaries and prevent naming collisions
- **Include Mechanism** (`include "header.fbs";`) enables modular schema design across multiple files
- **Root Type** (`root_type Header;`) identifies the primary table that serves as the entry point for buffer access

These features were essential for FlatCityBuf's implementation, enabling modular schema development with separate root types for header and feature components while maintaining consistent type definitions across files.

2.8.4. Binary Structure and Memory Layout

FlatBuffers organises serialised data in a flat binary buffer with the following characteristics:

- **Zero-copy access** through a carefully designed memory layout that allows direct access to serialized data without intermediate parsing
- **Vtable-based field access** where each table starts with an offset to its vtable, enabling efficient field lookup and schema evolution
- **Little-endian encoding** for all scalar values, with automatic conversion on big-endian platforms
- **Offset-based references** for all non-inline data (tables, strings, vectors), allowing efficient navigation within the buffer

For complex data structures like 3D city models, FlatBuffers allows for modular schema composition through file inclusion. This capability enabled the separation of FlatCityBuf's schema into logical components (`header.fbs`, `feature.fbs`, `geometry.fbs`, etc.) while maintaining efficient serialisation. In our implementation, the `Header` and `CityFeature` tables serve as root types that anchor the overall data structure.

3. Related Work

This section reviews the pertinent literature relevant to the optimisation of CityJSON for cloud-native environments.

It highlights advancements and identifies existing gaps that this research aims to address.

3.1. Cloud-Optimised Geospatial Formats

Cloud-optimised geospatial formats constitute specialised data structures engineered to maximise computational efficiency in distributed cloud environments [Cloud-Native Geospatial Foundation, 2023]. These formats exhibit several quantifiable advantages:

- **Reduced Latency**: Facilitates partial data retrieval and processing without necessitating complete file downloads.
- **Scalability**: Supports parallel operations through metadata-driven access mechanisms within cloud storage systems.
- **Flexibility**: Offers advanced query capabilities for selective data access.
- **Cost-Effectiveness**: Optimises storage and transfer expenditures through efficient access patterns.

Cloud-Native Geospatial Foundation [2023] provides a comprehensive overview of several cloud-optimised geospatial formats. These include Cloud Optimized GeoTIFF, Cloud Optimized Point Cloud, GeoParquet, PMTiles, FlatGeobuf.

3.2. CityGML, CityJSON and Its Enhancements

3.2.1. CityGML

CityGML is an OGC standard [OGC, 2019b] that defines a comprehensive data model for representing 3D city models. The standard encompasses both geometric properties and rich semantic information through a modular structure. From version 3.0.0, CityGML separates its conceptual model from its encoding standard. Figure Figure 3.1 shows an overview of its modules. The conceptual model defines the semantics and data model through a Core module and eleven thematic extension modules (Building, Bridge, Tunnel, Construction, CityFurniture, CityObjectGroup, LandUse, Relief, Transportation, Vegetation, and WaterBody). Additionally, five extension modules (Appearance, PointCloud, Generics, Versioning, and Dynamizer) provide specialized modeling capabilities applicable across all thematic modules. The encoding standard uses GML application schema for the Geography Markup Language (GML) [Open

Geospatial Consortium, 2000] to encode the data. This modular design allows implementations to support specific subsets of modules based on their application requirements, ensuring flexibility while maintaining standard compliance.

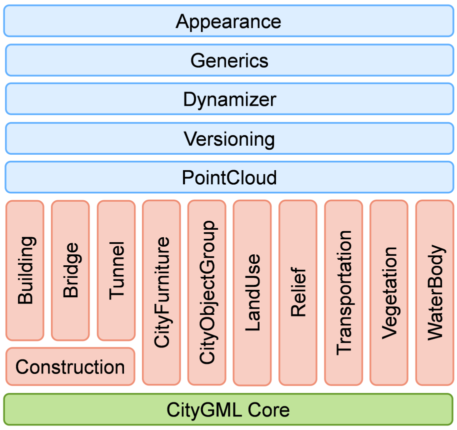


Figure 3.1.: Overview of CityGML 3.0 modules showing the Core module, thematic extension modules, and additional extension modules. Source: [OGC, 2019b]

3.2.2. CityJSON

CityJSON is a JSON-based [ECMA International, 2013] encoding format that implements a subset of the CityGML conceptual model [OGC, 2019b]. It is an official OGC standard [OGC] currently at version 2.0.1, supporting CityGML 3.0.0 encoding. While both CityGML and CityJSON implement the CityGML conceptual model, CityJSON exhibits several notable differences.

The following properties of CityJSON are derived from Ledoux et al. [2019]:

Flattened City Objects Architecture

CityJSON implements a flattened architecture where each city object receives a unique identifier, contrasting with CityGML's hierarchical structure. While CityGML maintains a hierarchical organization, CityJSON stores all objects at the same level (e.g., first and second-level city objects are stored in the same dictionary). To preserve hierarchical relationships, CityJSON uses a `parents` field to reference each object's parent.

Geometry

CityJSON supports the same 3D geometric primitives as CityGML. However, instead of storing vertex coordinates directly within geometric primitives, CityJSON maintains a separate `vertices` array containing all coordinates. Geometric primitives then reference vertex positions within this array.

Semantic Surfaces

CityJSON stores semantic surfaces as separate objects, recognizing that city objects often share common semantics. This is implemented through `semanticSurfaces` fields and a `values` array that maps surfaces to their corresponding semantic surface objects.

This is an example of how the semantic surfaces look like (derived from Ledoux et al. [2019]):

```
{
  "type": "Solid",
  "lod": 2,
  "boundaries": [
    [ [[0,3,2,1,22]], [[4,5,6,7]], [[0,1,5,4]],
      [[1,2,6,5]] ]
    ],
    "semantics": {
      "surfaces" : [
        { "type": "RoofSurface" },
        {
          "type": "WallSurface",
          "paint": "blue"
        },
        { "type": "GroundSurface" }
      ],
      "values": [ [0, 1, 1, 2] ]
    }
  }
}
```

Geometry Templates

CityJSON implements CityGML's Implicit Geometry concept through "geometry templates." The format includes `geometry-templates` fields with a `templates` array that stores reusable geometries. City objects utilizing these templates specify `"GeometryInstance"` in their geometry's `type` field to indicate template reuse.

This code shows an example of a geometry template derived from Ledoux et al. [2019]:

```
{
  "geometry-templates": {
    "templates": [{
      "type": "MultiSurface",
      "lod": 2,
      "boundaries": [
        [[0, 3, 2, 1]],
        [[4, 5, 6, 7]],
        [[0, 1, 5, 4]]
      ]
    }],
    "vertices-templates": [...]
  }
}
```

And this is how a city object references this template:

```
{
  "type": "SolitaryVegetationObject",
  "geometry": [
    {
```

```
        "type": "GeometryInstance",
        "template": 0,
        "boundaries": [372],
        "transformationMatrix": [
          2.0, 0.0, 0.0, 0.0,
          0.0, 2.0, 0.0, 0.0,
          0.0, 0.0, 2.0, 0.0,
          0.0, 0.0, 0.0, 1.0
        ]
      }
    ]
  }
```

Coordinate Quantization

CityJSON employs coordinate quantization to reduce geometry size. The `transform` field contains `scale` and `translate` values for coordinate quantization. The original coordinates are recovered using the following formula (e.g., for the x component of vertex v):

$$x = v_x \cdot \text{transform.scale}_x + \text{transform.translate}_x \tag{3.1}$$

This is an example of how the transform object looks like (derived from Ledoux et al. [2019]):

```
{
  "transform": {
    "scale": [0.01, 0.01, 0.01],
    "translate": [4424648.79, 5482614.69, 310.19]
  }
}
```

Extension Mechanism

CityJSON implements an extension mechanism using JSON Schema, similar to CityGML's Application Domain Extensions (ADEs). Biljecki et al. [2018] provides an overview of the developments of ADEs in CityGML. While CityJSON's extension mechanism maintains compatibility with the core CityGML conceptual model, it has some limitations compared to CityGML's ADEs, particularly in terms of inheritance and namespace support. The JSON Schema defines the data structure of extensions and can be used to validate extended objects.

CityJSON supports four distinct ways to extend the data model:

- Adding new properties at the root level of a CityJSON object (property names must start with `"+"`, e.g., `"+census"`)
- Defining additional attributes for existing city objects (attribute names must start with `"+"`, e.g., `"+colour"`)
- Creating new semantic objects (object names must start with `"+"`, e.g., `"+ThermalSurface"`)
- Creating or extending new city object types (city object names must start with `"+"`, e.g., `"+NoiseBuilding"`)

Each extension must be documented and validated using a JSON Schema file. This schema file must contain specific properties that define the structure and constraints of the extension. For example, an extension schema might look like this (derived from Ledoux et al. [2019]):

```
 {
   "type": "CityJSON_Extension",
   "name": "Noise",
   "uri": "https://someurl.org/noise.json",
   "version": "0.1",
   "description": "Extension to model the noise"
   "extraRootProperties": {},
   "extraAttributes": {},
   "extraCityObjects": {}
}
```

The proposed data format, FlatGeobuf, inherits key concepts from CityJSON including semantic surfaces, geometry templates, coordinate quantization, and the extension mechanism. The strategy for encoding city objects in CityJSON will be explained in chapter 4.

3.2.3. CityJSON Text Sequences (CityJSONSeq)

Ledoux et al. [2024] optimises CityJSON for streaming applications by decomposing objects into independent sequences.

The fundamental unit of CityJSONSeq is the `CityJSONFeature`, which represents a single feature encompassing a complete city object and its hierarchical children. For instance, a `CityJSONFeature` representing a `"Building"` includes its associated `"BuildingPart"` and `"BuildingInstallation"` objects. Unlike standard CityJSON objects that share vertices and appearances across multiple features, each `CityJSONFeature` maintains local vertex lists and appearance data, ensuring complete self-containment of geometric and visual information. Throughout this research, `CityJSONFeature` objects are referred to simply as `"features"`.

CityJSONSeq adheres to the Newline Delimited JSON specification [ndjson, 2013], implementing a structured file format with specific requirements. The first line of a CityJSONSeq file must contain a CityJSON object that stores commonly used data shared across all features, including coordinate transformation parameters (`transform`), format version information (`version`), metadata (`metadata`), reusable geometry templates (`geometry-templates`), and extension definitions (`extensions`). This initialisation object establishes the global context for subsequent features. The example below shows how a `CityJSONFeature` is represented (derived from Ledoux et al. [2024]):

```
{
  "type": "CityJSONFeature",
  "id": "id-1",
  "CityObjects": {
    "id-1": {
      "type": "Building",
      "attributes": {
        "roofType": "gabled roof",
        "children": ["my balcony"]
      },
      "geometry": [ ... ]
    },
    "my balcony": {
      "type": "BuildingInstallation",
      "parents": ["id-1"],
```

```
        "geometry": [ ... ]
      }
    },
    "vertices": [ ... ],
    "appearance": { ... }
  }
```

While CityJSONSeq generally offers improved compression and memory efficiency relative to standard CityJSON, larger file sizes may occur in scenarios with minimal vertex counts or extensive vertex and texture sharing across features, due to the localisation of previously shared resources.

While CityJSONSeq improves upon standard CityJSON for streaming applications, its text-based JSON format presents several limitations that hinder optimal cloud-native performance. The format lacks explicit data typing, storing all values as strings regardless of their semantic type (numbers, booleans, etc.), which tends to increase storage overhead and requires additional parsing. Furthermore, the JSON structure necessitates complete parsing and copying of data during processing, limiting memory efficiency. Additionally, the absence of built-in indexing mechanisms restricts efficient spatial and attribute-based querying capabilities. These limitations present opportunities for further cloud-native optimisation through binary encoding schemes that preserve data types and enable zero-copy access, combined with indexing mechanisms for efficient data retrieval.

3.2.4. Enhancements to CityJSON Performance

Binary Encoding of CityJSON

van Liempt [2020] conducted a systematic evaluation of binary encoding techniques for CityJSON. This was done to address challenges associated with transmitting large-scale 3D city models over the web. The study assessed various compression and encoding methodologies, including CBOR, zlib, Draco and their combinations. It evaluated visualisation time, querying time, spatial analysis time, editing time, file size compression and lossiness. The analysis determined that the combination of CBOR and zlib offers optimal general-purpose efficiency due to its implementation simplicity. Conversely, Draco exhibited superior performance for pre-compressed data scenarios. However, the study identified limitations in Draco's applicability. Specifically, it noted the increased complexity and computational overhead when handling smaller datasets. While these findings provide valuable insights for binary encoding implementations, they do not address optimisations tailored to cloud-native environments.

Experimental Implementation Using FlatBuffers

Peters [2024] explored the application of FlatBuffers [Google, 2014a] for encoding CityJSONFeature.

This was done to enhance performance in cloud-native environments.

The preliminary implementation revealed potential advantages in several key areas:

- Faster feature access time.
- Lower memory consumption.

- Decreased storage requirements.

Building upon Peters' initial work, which focused solely on basic CityJSONFeature encoding, this research develops a comprehensive solution that incorporates essential capabilities including spatial indexing, extensions, textures, and geometry templates. The implementation specifically targets cloud-native environments, prioritizing both scalability and efficient data processing to address the limitations of the preliminary approach.

3.2.5. Research Gaps

While existing studies have made significant advances in optimising CityJSON through various encoding techniques, there remains a deficiency in approaches specifically tailored for 3D city models in cloud environments.

Several geospatial data formats have successfully implemented cloud-native optimisations (as discussed in section 3.4).

Specifically, while advanced serialisation frameworks like FlatBuffers (detailed in section 2.8) have proven effective in cloud-optimised geospatial formats, their application to 3D city models has not been thoroughly investigated.

For example, FlatGeobuf for Simple Features [FlatGeobuf, 2020] has shown success.

This research endeavours to address this gap by systematically evaluating and implementing encoding methodologies. These methodologies aim to enhance decoding efficiency and query flexibility within cloud infrastructures, with the potential to achieve file size reduction through optimized binary encoding.

The proposed approach is detailed in chapter 4.

3.3. Non-Geospatial Formats in Cloud Environments

Modern cloud-optimised geospatial formats leverage established non-geospatial data structures.

These enhance efficiency in data transfer, storage and processing operations.

Notable implementations include GeoParquet [GeoParquet Contributors, 2024], which employs Parquet [Apache Software Foundation, 2013] for optimised geospatial data management.

FlatGeobuf [2020] is constructed on FlatBuffers [Google, 2014a].

Mapbox Vector Tiles [Mapbox] utilise Protocol Buffers (Protobuf) [Google, 2008].

These underlying formats are meticulously designed to improve performance metrics.

These include serialisation/deserialisation speed, memory utilisation and data compression.

3.3.1. FlatBuffers

Google [2014a] is a cross-platform serialisation library developed by Google [2014].

It is optimised for efficient data access and transfer.

The detailed characteristics and technical implementation of FlatBuffers will be explained in section 2.8.

Benchmark analyses [Google, 2014b] indicate that FlatBuffers outperforms alternative serialisation formats.

These include Protobuf [Google, 2008] and JSON, in terms of deserialisation efficiency and memory utilisation.

3.3.2. Protocol Buffers (Protobuf)

Google [2008], developed by Google, represents a binary serialisation framework.

It employs schema-based encoding mechanisms for data serialisation.

This framework implements similar fundamental operations to FlatBuffers.

These include schema definition and binary encoding processes.

Despite its advantages in simplicity and usability, Protobuf presents several operational constraints:

improve this."Does flatbuffers offer compression? ie. why do you mention compression here but not in 2.3.1?"

- **Memory Limitations**: Requires complete dataset loading into memory, thereby limiting its applicability for large-scale data processing tasks.
- **Compression Efficiency**: Lacks native compression capabilities, resulting in suboptimal performance compared to specialised formats like JPEG and PNG for image data.
- **Structural Constraints**: Exhibits reduced efficiency when handling complex data structures, particularly large multidimensional arrays of floating-point numbers.

3.3.3. Apache Parquet

Apache Software Foundation [2013] is a columnar storage format designed to support high-performance compression and encoding schemes.

These are used for managing extensive datasets.

The Parquet ecosystem includes the Apache Parquet Contributors, which serves as the specification for the Parquet format.

It also includes various libraries for encoding and decoding Parquet files.

Parquet employs the record shredding and assembly algorithm [Melnik et al., 2010] to effectively flatten nested data structures.

Additionally, it implements efficient compression and encoding schemes tailored to column-level data.

This enhances both storage efficiency and query performance.

3.3.4. Comparison of Non-Geospatial Formats

Existing research has evaluated the performance characteristics of non-geospatial formats within cloud environments.

Proos and Carlsson [2020] conducted a comparative analysis of FlatBuffers and Protobuf.

This focused on metrics such as serialisation/deserialisation efficiency, memory utilisation, and message size optimisation.

Their investigation utilised randomised message sizes to assess format performance in vehicle-to-server communication scenarios.

The analysis yielded the following observations:

- **Processing Efficiency**: Protobuf demonstrated superior serialisation performance but exhibited reduced deserialisation efficiency relative to FlatBuffers.
- **Memory Optimisation**: FlatBuffers consistently displayed lower memory consumption during both serialisation and deserialisation operations.
- **Data Compression**: Protobuf achieved greater message size reduction compared to FlatBuffers.

These findings advocate for the selection of FlatBuffers in applications where deserialisation performance and memory efficiency are paramount in data processing operations.

3.4. Cloud-Optimised Geospatial Implementations

Contemporary cloud-optimised geospatial implementations encompass formats such as Mapbox Vector Tiles [Mapbox], FlatGeobuf [FlatGeobuf, 2020], PMTiles [Protomaps, 2022], and GeoParquet [GeoParquet Contributors, 2024].

3.4.1. Mapbox Vector Tiles (MVT)

Mapbox implements a vector tile specification optimised for web-based data delivery. The format utilises Protobuf for the serialisation of two-dimensional geospatial data and adopts a tile pyramid structure to enhance data retrieval operations.

3.4.2. PMTiles

PMTiles offers a standardised format for managing tile data addressed through Z/X/Y coordinates, supporting both vector and raster tile implementations. The format leverages HTTP Range Requests [Internet Engineering Task Force, 2014] to facilitate selective tile retrieval, thereby optimising network resource utilisation.

3.4.3. FlatGeobuf

FlatGeobuf adheres to the Simple Features OGC [2011] specification by utilising Google [2014a] for serialisation. The architecture of FlatGeobuf enables efficient serialisation, deserialisation, and data processing operations. Notably, its partial data access capabilities allow clients to selectively retrieve and process specific geographic regions without necessitating the loading of the entire dataset. Williams [2022a] provides a comprehensive guide for implementers of FlatGeobuf.

3.4.4. GeoParquet

GeoParquet integrates Parquet's columnar storage architecture to facilitate optimised geospatial data operations [GeoParquet Contributors, 2024]. The format promotes interoperability across cloud data warehouse platforms, including BigQuery [Google, 2011], Snowflake [Snowflake Inc., 2015], and Redshift [Amazon Web Services, 2012]. Key technical characteristics of GeoParquet include:

- **Compression Efficiency**: Achieves superior compression ratios relative to alternative storage formats through columnar data organisation.
- **Optimised Read Operations**: The columnar architecture enables selective column access and efficient data filtering via predicate pushdown mechanisms, thereby enhancing performance in read-intensive workflows.

3.4.5. 3D Tiles

3D Tiles, an Open Geospatial Consortium (OGC) standard [OGC, 2019a], provides specifications for streaming and rendering extensive three-dimensional urban models. The format implements GLTF [Khronos Group, 2015], a WebGL-optimised specification designed for efficient streaming in web environments.

The data structure employs spatial partitioning through bounding volumes, enabling selective rendering based on camera viewpoint requirements. While this architecture demonstrates optimal performance for visual rendering tasks, it presents limitations in two key areas: (1) arbitrary spatial extent retrieval and (2) attribute-based feature querying capabilities.

3.4.6. Comparative Analysis of Cloud-Optimised Geospatial Formats

While acknowledging the inherent limitations of direct format comparisons due to their distinct design objectives and application domains, Table 3.1 presents a systematic analysis of key operational characteristics across various cloud-optimised geospatial formats. The evaluation criteria and their corresponding scales are detailed in Table 3.2. This analysis facilitates the understanding of format-specific capabilities within their respective operational contexts.

[1] Optimised for GPU rendering
[2] Tile-based partitioning
[3] Random access to the internal chunks
[4] Volumentric hierarchical partitioning

Table 3.1.: Comparative Analysis of Cloud-Optimised Geospatial Formats (Scale: 1-5)

Characteristics	FlatGeobuf	MVT	GeoParquet	GeoJSON	3D Tiles
Serialisation Performance	3	4	3	2	–
Deserialisation Performance	4	3	5	1	4[1]
Storage Efficiency	3	4	5	1	–
Memory Utilisation	5	4	5	1	–
Implementation Complexity	2	2	2	5	–
Spatial Indexing	5	3[2]	3[3]	1	3[4]
Random Access Support	5	1	4	1	1

Table 3.2.: Evaluation Criteria Scale (1-5)

Criterion	Scale Description
Serialisation Performance	1: Very slow, 2: Slow, 3: Moderate, 4: Fast, 5: Very fast
Deserialisation Performance	1: Very slow, 2: Slow, 3: Moderate, 4: Fast, 5: Very fast
Storage Efficiency	1: No compression, 3: Moderate compression, 5: Very high compression
Memory Utilisation	1: Very high memory usage, 2: High usage, 3: Moderate usage, 4: Low usage, 5: Very low usage
Implementation Complexity	1: Complex, 3: Moderate, 5: Simple
Spatial Indexing	1: Not supported, 3: Basic support, 5: Indexing with arbitrary spatial extent
Random Access Support	1: Not supported, 3: Partial support, 5: Full random access

4. Methodology

This chapter presents the design and implementation of FlatCityBuf, a cloud-optimised binary format for 3D city models based on CityJSON. The proposed approach addresses the limitations of existing formats through efficient binary encoding, spatial indexing, attribute indexing, and support for partial data retrieval.

4.1. Overview

4.1.1. Methodology Approach

Current 3D city model formats like CityGML, CityJSON, and CityJSONSeq (also CityJSONSeq) exhibit limitations in cloud environments with large-scale datasets, including retrieval latency, inefficient spatial querying without additional software support, and insufficient support for partial data access.

This research methodology addresses these limitations through three interconnected objectives:

1. Development of a binary encoding strategy using FlatBuffers that preserves semantic richness while achieving faster read performance
2. Implementation of dual indexing mechanisms—spatial (S+Tree) and attribute-based (S+Tree)—that accelerate query performance
3. Integration of cloud-native data access patterns through HTTP Range Requests, enabling partial data retrieval

4.1.2. Outcomes of the Methodology

Before delving into the methodological details, it is important to highlight the tangible research outcomes produced through this work:

- **Data format specification**: FlatCityBuf, a cloud-optimised binary format for 3D city models that maintains semantic compatibility with CityJSON while enabling efficient cloud-based access patterns.
- **Reference implementation**: A comprehensive Rust library for encoding, decoding, and querying FlatCityBuf files, accompanied by command-line interface (CLI) tools for conversion and validation.
- **Web demonstration**: A web-based prototype application that showcases the partial data retrieval capabilities of FlatCityBuf through HTTP range requests, demonstrating practical performance improvements in real-world scenarios.

These outcomes collectively address the research objectives by providing both a theoretical framework and practical implementations that validate the approach to cloud-optimised 3D city model storage and retrieval.

4.1.3. File Structure Overview

The FlatCityBuf format implements a structured binary encoding with five sequentially arranged components:

- **Magic bytes**: Eight-byte identifier ('F', 'C', 'B', '0', '1', '0', '0', '0') for format validation
- **Header section**: Contains metadata, schema definitions, and CityJSON properties
- **Spatial index**: Implements a Packed Hilbert R-tree for efficient geospatial queries
- **Attribute index**: Utilises a S+Tree for accelerated attribute-based filtering
- **Features section**: Stores CityJSON Feature (CityFeature) encoded as FlatBuffers tables

Figure 4.1.: Physical layout of the FlatCityBuf file format, showing section boundaries and alignment considerations for optimised range requests

This sequence-based structure enables incremental file access through HTTP Range Requests—critical for cloud-based applications where minimising data transfer is essential. Each section is designed with explicit consideration for alignment boundaries to optimise I/O operations.

4.1.4. Note on Binary Encoding

add reference why these two

FlatCityBuf follows two key conventions for encoding binary data throughout the file format:

1. **Size-prefixed FlatBuffers**: All FlatBuffers records (header and features) include a 4-byte unsigned integer prefix indicating the buffer size. This enables programs to know the size of the record without parsing the entire content. The FlatBuffers API implements this through `finish_size_prefixed` or equivalent language-specific methods.
2. **Little-endian encoding**: For data encoded outside FlatBuffers records (particularly in spatial and attribute indices), little-endian byte ordering is consistently applied. This includes numeric values such as 32-bit and 64-bit integers, floating-point numbers, and offset values within indices.

add reference

These conventions ensure consistency across the file format and maximise compatibility with modern CPU architectures, most of which use little-endian byte ordering. The size-prefixing mechanism is particularly important for cloud-based access patterns, as it facilitates precise HTTP Range Requests when retrieving specific file segments.

4.2. Magic Bytes

The magic bytes section comprises the first eight bytes of the file: check detail again

- The first three bytes contain the ASCII sequence 'FCB' (0x46 0x43 0x42) serving as an immediate identifier
- The remaining five bytes represent the version number of the file format, comprised with Semantic Versioning (SemVer) [SemVer]. As the current version is 0.1.0, the magic bytes are 'FCB010' (0x46 0x43 0x42 0x30 0x31 0x30). The last two bytes are reserved for future use and must be set to zero.

This signature design enables applications to validate file type and version compatibility without parsing the entire header content. The approach was directly inspired by FlatGeoBuf's methodology, which uses 'FGB' (F, G, B characters) in its magic bytes to indicate 'FlatGeoBuf' [Williams, 2022a].

4.3. Header Section

The header section encapsulates metadata essential for interpreting the file contents, implemented as a size-prefixed FlatBuffers-serialised `Header` table. The header serves a dual purpose: it maintains compatibility with CityJSON by encoding the equivalent of the first line of a CityJSONSeq stream [Ledoux et al., 2024]—which contains the root CityJSON object with metadata, coordinate reference system, and transformations—while adding FlatCityBuf-specific extensions for optimised retrieval and indexing. The full schema definition for the header can be found in Appendix C.

In a CityJSONSeq file, the first line contains a valid CityJSON object with empty `CityObjects` and `vertices` arrays but with essential global properties like `transform`, `metadata`, and `version`. The FlatCityBuf header encodes these same properties alongside additional indexing information required for cloud-optimised access patterns.

4.3.1. CityJSON Metadata Fields

Here are the core header fields with their data types and significance:

- **version** - *string (required)* - CityJSON version identifier (e.g., "2.1"), required field from CityJSON specification [CityJSON, 2024]
- **transform** - *Transform struct* - Contains scale and translation vectors enabling efficient storage of vertex coordinates through quantization, derived from CityJSON's transform object [CityJSON, 2024]
- **reference_system** - *ReferenceSystem table* - Coordinate reference system information including:
  - **authority** - *string* - Authority name, typically "EPSG"
  - **code** - *string* - Numeric identifier of the CRS
  - **version** - *string* - Version of the CRS definition

- **geographical_extent** - *GeographicalExtent struct* - 3D bounding box containing min/-max coordinates for the dataset [CityJSON, 2024]
- **identifier** - *string* - Unique identifier for the dataset
- **title** - *string* - Human-readable title for the dataset
- **reference_date** - *string* - Date of reference for the dataset
- **point of contact** - *Contact table* - Contact information for the dataset provider [CityJSON, 2024], containing:
  - **poc_contact_name** - *string* - Name of the point of contact
  - **poc_contact_type** - *string* - Type of contact (e.g., "individual", "organization")
  - **poc_role** - *string* - Role of the contact (e.g., "author", "custodian")
  - **poc_email** - *string* - Email address of the contact
  - **poc_website** - *string* - Website for the contact
  - **poc_phone** - *string* - Phone number of the contact
  - **poc_address_*** - *string* - Address components including thoroughfare number, name, locality, postcode, country

4.3.2. Appearance Information

Fields storing global appearance definitions:

- **appearance** - *Appearance table* - Container for visual representation properties, following CityJSON's appearance model [CityJSON, 2024], containing:
  - **materials** - *Array of Material tables* with the following properties:
    - **name** - *string* - Required string identifier for the material
    - **ambient_intensity** - *double* - Double precision value from 0.0 to 1.0
    - **diffuse_color** - *Array of double* - Array of double values (RGB) from 0.0 to 1.0
    - **emissive_color** - *Array of double* - Array of double values (RGB) from 0.0 to 1.0
    - **specular_color** - *Array of double* - Array of double values (RGB) from 0.0 to 1.0
    - **shininess** - *double* - Double precision value from 0.0 to 1.0
    - **transparency** - *double* - Double precision value from 0.0 to 1.0
    - **is_smooth** - *boolean* - Boolean flag for smooth shading
  - **textures** - *Array of Texture tables* with the following properties:
    - **type** - *TextureFormat enum* - TextureFormat enum (PNG, JPG)
    - **image** - *string* - Required string containing image file name or URL

    * **wrap_mode** - *WrapMode enum* - WrapMode enum (None, Wrap, Mirror, Clamp, Border)
    * **texture_type** - *TextureType enum* - TextureType enum (Unknown, Specific, Typical)
    * **border_color** - *Array of double* - Array of double values (RGBA) from 0.0 to 1.0
  - **vertices_texture** - *Array of Vec2 structs* - Array containing UV coordinates (u,v), each coordinate value must be between 0.0 and 1.0 for proper texture mapping
  - **default_theme_material** - *string* - String identifying default material theme for rendering when multiple themes are defined
  - **default_theme_texture** - *string* - String identifying default texture theme for rendering when multiple themes are defined

The appearance model standardizes visual properties of city objects, with materials defining surface properties and textures mapping images onto geometry. This separation from geometry allows efficient storage through shared material and texture references.

4.3.3. Geometry Templates

Fields supporting geometry reuse:

- **templates** - *Array of Geometry tables* - Reusable geometry definitions that can be instantiated multiple times, following CityJSON's template concept [CityJSON, 2024]
- **templates_vertices** - *Array of DoubleVertex structs* - Double-precision vertices used by templates, stored separately from feature vertices for higher precision in the local coordinate system [CityJSON, 2024]

The templates mechanism enables significant storage efficiency for datasets containing repetitive structures such as standardised building designs, street furniture, or vegetation. The detailed structure of geometry encoding, including boundary representation and semantic surface classification, will be explained further in subsection 4.6.2.

4.3.4. Extension Support

Fields enabling to accommodate CityJSON's extension mechanism:

- **extensions** - *Array of Extension tables* - Definitions for CityJSON extensions [CityJSON, 2024], each containing:
  - **name** - *string* - Extension identifier (e.g., "+Noise")
  - **url** - *string* - Reference to the extension schema
  - **version** - *string* - Extension version identifier
  - **extra_attributes** - *string* - Stringified JSON schema for extension attributes
  - **extra_city_objects** - *string* - Stringified JSON schema for extension city objects

  - **extra_root_properties** - *string* - Stringified JSON schema for extension root properties
  - **extra_semantic_surfaces** - *string* - Stringified JSON schema for extension semantic surfaces

Unlike standard CityJSON [CityJSON, 2024], which references external schema definition files for extensions, FlatCityBuf embeds the complete extension schemas directly within the file as stringified JSON. This approach creates a self-contained, all-in-one data format that can be interpreted correctly without requiring access to external resources.

The embedding of extension schemas follows FlatCityBuf's design principle of maintaining file independence while preserving full compatibility with the CityJSON extension mechanism. The specific implementation details of how extended city objects and semantic surfaces are encoded in individual features will be explained further in section 4.6.

4.3.5. Attribute Schema and Indexing Metadata

Fields supporting attribute interpretation and efficient querying:

- **columns** - *Array of Column tables* - Schema definitions for attribute data. This metadata is used to interpret the values of the attributes in the features. Each containing:
  - **index** - *int* - Numeric identifier of the column
  - **name** - *string* - Name of the attribute (e.g., "cityname", "owner", etc.)
  - **type** - *DataType enum* - Data type enumeration (e.g., "Int", "String", etc.)
  - **nullable** - *boolean* - Optional metadata for validating and interpreting values
  - **unique** - *boolean* - Optional metadata for validating and interpreting values
  - **precision** - *int* - Optional metadata for validating and interpreting values
- **semantic_columns** - *Array of Column tables* - Schema definitions for semantic surface attributes. Similar to the **columns** field, but specifically for interpreting attribute data attached to semantic surfaces in the geometry. This separation allows for different attribute schemas between city objects and their semantic surfaces.
- **features_count** - *ulong* - Total number of features in the dataset, enables client applications to pre-allocate resources
- **index_node_size** - *ushort* - Number of entries per node in the spatial index, defaults to 16, tuned for typical HTTP request sizes
- **attribute_index** - *Array of AttributeIndex structs* - Metadata for each attribute index, containing:
  - **index** - *int* - Reference to the column being indexed
  - **length** - *ulong* - Size of the index in bytes
  - **branching_factor** - *ushort* - Branching factor of the index, number of items in each node is equal to branching factor − 1
  - **num_unique_items** - *ulong* - Count of unique values for this attribute

The attribute schema system in FlatCityBuf is designed to efficiently interpret binary-encoded attribute values. The Column table structure is directly adopted from FlatGeoBuf's approach [Williams, 2022a], which provides a flexible and extensible way to define attribute schemas. While optional fields such as **nullable**, **unique**, and **precision** are currently not utilized, they are included in the schema to accommodate potential future use cases.

4.3.6. Implementation Considerations

The header is designed to be compact while providing all necessary information to interpret the file. The size-prefixed FlatBuffers encoding enables efficient skipping of the header when only specific features are needed, important for cloud-based access patterns where minimising data transfer is essential. All numeric values in the header use little-endian encoding for consistency with modern architectures.

4.4. Spatial Indexing

Efficient spatial querying is a critical requirement for 3D city model formats, particularly in cloud environments where minimising data transfer is essential. FlatCityBuf implements a packed Hilbert R-tree spatial indexing mechanism [Roussopoulos and Leifker, 1985] to enable selective retrieval of city features based on their geographic location. This section details the implementation approach, design decisions, and performance characteristics of the spatial indexing component.

4.4.1. Design Attribution

The spatial indexing mechanism implemented in FlatCityBuf directly adapts the packed Hilbert R-tree approach developed for FlatGeoBuf [Williams, 2022a]. The design combines several key innovations:

- A Hilbert curve-based spatial ordering strategy, inspired by Vladimir Agafonkin's flatbush library, which optimizes data locality for spatially proximate features
- A "packed" R-tree implementation, where the tree is maximally filled with no empty internal slots, optimized for static datasets
- A bottom-up tree construction methodology that builds the index from pre-sorted features
- A flattened tree storage format that enables efficient streaming and remote access

The implementation details, including the Hilbert curve encoding algorithm and tree construction process, were sourced from FlatGeoBuf's reference implementation [FlatGeobuf]. Also, FlatGeoBuf's implementation is also inspired by Vladimir Agafonkin's flatbush library [Agafonkin, 2010]. The Hilbert curve encoding algorithm, which converts 2D coordinates into a 1D space-filling curve, is based on a non-recursive algorithm described in Warren [2012]. This approach, known as "2D-C" in spatial indexing literature [Warren, 2012], ensures that features with high spatial locality also have high storage locality, optimizing I/O operations for both local and remote access patterns.

The spatial indexing system is designed to support cloud-native access patterns, allowing efficient retrieval of data directly from cloud storage without requiring a persistent server process. This is achieved through a combination of the Hilbert-sorted feature ordering and the packed R-tree structure, which enables piecemeal access to both the index and feature data over HTTP requests.

It is important to explicitly acknowledge that the spatial indexing code in FlatCityBuf is a direct adaptation of FlatGeoBuf's implementation, with modifications primarily focused on integration with the 3D city model data structure rather than fundamental algorithmic changes. The original implementation by Björn Harrtell and other FlatGeoBuf contributors [FlatGeobuf, 2020] provided an excellent foundation that has been proven effective for cloud-optimized geospatial data.

While the original FlatGeoBuf implementation targets 2D vector geometries, FlatCityBuf extends this approach to work with 3D city models by applying the indexing to 2D projections (centroids) of the 3D features. The decision to reuse this proven approach rather than developing a novel indexing mechanism was based on FlatGeoBuf's demonstrated effectiveness for cloud-optimized geospatial data formats.

4.4.2. Feature sorting

A key optimization in FlatCityBuf's indexing strategy is the spatial ordering of features using a Hilbert space-filling curve. This technique enhances data locality by ensuring that features which are spatially proximate in 3D space are also stored close together in the file, thereby optimizing both disk access patterns and HTTP range requests [Williams, 2022a].

The Hilbert curve encoding process for FlatCityBuf follows these steps:

1. For each CityFeature, determine its 2D footprint by calculating the minimum and maximum X,Y coordinates from all vertices across all contained CityObjects
2. Calculate the geometric centroid of this 2D footprint
3. Apply a 32-bit Hilbert encoding algorithm to this centroid, converting the 2D spatial position into a 1D ordering value
4. Sort all features according to their computed Hilbert values in ascending order
5. During serialization of the sorted features, record both the 2D bounding box and the byte offset (relative position from the start of the feature section) for each feature

These recorded bounding boxes and byte offsets become the foundation for constructing the bottom layer of the R-tree index. The Hilbert encoding implementation uses a non-recursive algorithm described in Warren [2012], which has been adapted from the FlatGeoBuf implementation [FlatGeobuf], which in turn was inspired by a public domain implementation by Rawlinson and Toth [2016].

This approach differs from traditional R-tree construction where nodes are built based on spatial proximity alone. By pre-sorting features along a space-filling curve before constructing the R-tree, FlatCityBuf achieves more predictable and efficient I/O patterns when performing spatial queries [Williams, 2022b].

4.4.3. Index structure

The spatial index in FlatCityBuf is implemented as a packed Hilbert R-tree, with a flattened, level-ordered storage structure optimized for efficient traversal over HTTP range requests [Williams, 2022a]. The index is built bottom-up from the sorted features, creating a hierarchical structure where each node represents a spatial region containing its children.

Each index node in the spatial index is represented by a fixed-size binary structure containing:

- **Bounding box coordinates**: 4 little-endian double values (4 bytes each) defining the minimum and maximum X,Y coordinates of the node's bounding box
- **Byte offset**: A 64-byte unsigned integer pointing to either:
  - For leaf nodes: The position of the corresponding feature in the features section

  – For interior nodes: The position of the node's first child in the index section

This results in a fixed node size, allowing for predictable memory layouts and efficient search within each node level.

The tree is built using the following process:

1. Create the bottom layer (leaf nodes) using the bounding boxes and byte offsets recorded during feature serialization
2. Group these leaf nodes according to their Hilbert-sorted order into parent nodes, with each parent node containing up to `index_node_size` children (configurable)
3. Compute the bounding box of each parent node as the union of its children's bounding boxes
4. Continue building upward, level by level, until a single root node is reached
5. Serialize the entire tree in level order, starting with the root, then its children, and so on

This "packed" structure ensures that the R-tree is maximally filled (except potentially for the rightmost nodes at each level), which is possible because the tree is built in bulk from a known static dataset. The total size of the index is deterministic and based solely on the number of features and the chosen node size.

Unlike traditional R-trees which support dynamic updates, the packed R-tree in FlatCityBuf is immutable after creation. This trade-off prioritizes read performance and structural efficiency over the ability to modify the dataset, aligning with the file format's primary use case as a cloud-optimized geospatial data delivery mechanism [Williams, 2022b].

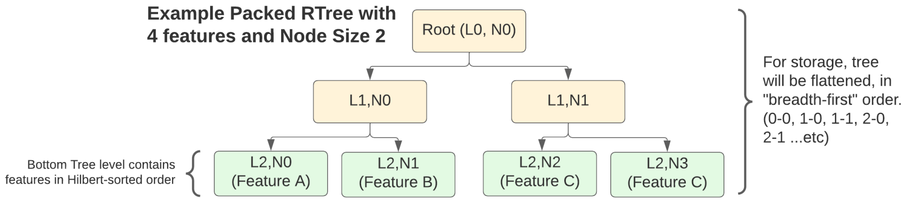


Figure 4.2.: Example of a packed R-tree structure. Image sourced from Williams [2022b].

4.4.4. 2D vs 3D Indexing Considerations

Although FlatCityBuf is designed for 3D city models, the spatial indexing mechanism deliberately uses a 2D approach rather than a full 3D implementation. This design decision was based on several key observations:

- **Horizontal Distribution**: Most 3D city models are primarily distributed horizontally in global scale, with limited vertical extent relative to their horizontal footprint
- **Query Patterns**: Typical spatial queries for city models focus on horizontal regions (e.g., retrieving buildings within a district), rather than volumetric queries

- **Standards Compatibility**: OGC API - Features - Part 1: Core [OGC, 2019c] and similar standards primarily support 2D spatial querying
- **Implementation Efficiency**: 2D indexing is computationally simpler and more storage-efficient than 3D alternatives

4.5. Attribute Indexing

Attribute indexing is a fundamental component of the FlatCityBuf format, enabling efficient filtering and retrieval of city objects based on their non-spatial properties. This section details the requirements, design considerations, and implementation of the attribute indexing system.

4.5.1. Query Requirements Analysis

The attribute indexing system in FlatCityBuf was designed to support query patterns commonly encountered in geospatial applications. To determine which query operators to prioritize, we analyzed established standards in the geospatial domain and common usage patterns in existing GIS software.

Common Query Operators in Geospatial Standards

Two major OGC standards provide guidance on common query operators: Filter Encoding Standard [OGC, 2010] and Common Query Language [OGC, 2024]. These standards define operators in several categories, as summarized in Table 4.1.

Table 4.1.: Common query operators in geospatial standards

Category	OGC Filter Encoding	OGC CQL
Logical Operators	AND, OR, NOT	AND, OR, NOT
Comparison Operators	PropertyIsEqualTo, PropertyIsNotEqualTo, PropertyIsLessThan, PropertyIsGreaterThan, PropertyIsLessThanOrEqualTo, PropertyIsGreaterThanOrEqualTo, PropertyIsLike, PropertyIsNull, PropertyIsBetween	=, ! =, <, <=, >, >=, LIKE, IS NULL, BETWEEN, IN
Spatial Operators	BBOX, Equals, Disjoint, Touches, Within, Overlaps, Crosses, Intersects, Contains, DWithin, Beyond	INTERSECTS, EQUALS, DISJOINT, TOUCHES, WITHIN, OVERLAPS, CROSSES, CONTAINS
Temporal Operators	After, Before, Begins, BegunBy, During, TContains, TEquals, TOverlaps, Ends, EndedBy, Meets, MetBy, OverlappedBy, AnyInteracts	AFTER, BEFORE, BEGINS, BEGUNBY, DURING, TCONTAINS, TEQUALS, TOVERLAPS, ENDS, ENDEDBY, MEETS, METBY, OVERLAPPEDBY, ANYINTERACTS
Additional Capabilities	ResourceId	Functions, Arithmetic Expressions, Array Operators

Priority Operators for FlatCityBuf

Based on this analysis and the practical constraints of optimizing for cloud-based access, FlatCityBuf prioritizes support for the following operators:

1. **Primary Comparison Operators**: Operators with direct index support
   - Equality ($=$)
   - Inequality ($!=$)
   - Less than ($<$)
   - Less than or equal ($<=$)
   - Greater than ($>$)
   - Greater than or equal ($>=$)
   - BETWEEN (implemented as combined $\geq$ and $\leq$)
2. **Logical Combinations**: Supported at the query execution level
   - AND (intersection of result sets)
   - OR (union of result sets) (This will be implemented in the future)

Uncomment if I could finish the implementation

Other operators from the standards were evaluated but not prioritized in the initial implementation, either because they require more complex index structures (e.g., LIKE operators) or are less commonly used in typical 3D city model queries.

By focusing on these high-priority operators, FlatCityBuf's attribute indexing system aims to support the most common query patterns while maintaining efficient performance for cloud-based access. This approach provides capabilities that exceed current offerings such as the 3DBAG API, which primarily supports feature retrieval by identification attribute (`identificatie`) and is still working toward full OGC compliance [3DBAG, 2023].

4.5.2. S+Tree Design and Modifications

After evaluating alternatives, a S+Tree with significant modifications was adopted for FlatCityBuf's attribute indexing. S+Tree is a variant of the Static B+Tree that is specialised for read-only access patterns. Its theoretical background is described in section 2.7. This decision was based on the following considerations:

- **I/O Efficiency and Balanced Performance**: B+trees organise data into fixed-size nodes matching common CPU cache sizes, offering $O(\log_B n)$ search complexity where B is the branching factor. This significantly reduces both the number of I/O operations and network roundtrips compared to binary search, making it ideal for HTTP Range Requests where each roundtrip incurs substantial latency.
- **Query Versatility**: Unlike specialized data structures such as hash tables (optimized for exact matches) or sorted arrays (better for range queries), the B+tree structure efficiently supports both exact match and range queries without compromising performance in either case. This versatility makes it well-suited for the diverse query patterns common in 3D city model applications.

S+Tree Characteristics

A S+Tree differs from a traditional B+tree in several important aspects:

- **Immutability**: Once constructed, the tree structure remains fixed, eliminating the need for complex rebalancing operations.
- **Perfect Node Fill**: All nodes except possibly the rightmost nodes at each level are filled to capacity, maximizing space efficiency.
- **Predictable Structure**: The tree shape is determined solely by the number of elements and the node size, making navigation more efficient.
- **Bulk Construction**: The tree is built bottom-up in a single pass from sorted data, rather than through incremental insertions.

The original S+tree algorithm as described by Slotin [2021b] provides an excellent foundation for read-only indexing. However, several significant modifications were necessary to adapt it to the specific requirements of FlatCityBuf:

- **Duplicate Key Handling**: 3D city model attributes often contain numerous duplicate values (e.g., hundreds of features with "Delft" as the value for "city name"). The S+Tree implementation described in literature [Slotin, 2021b] does not address the case of having duplicate values. The modified implementation incorporates a dedicated payload section that efficiently stores multiple feature references for identical attribute values without compromising the tree structure or search performance.

  For handling duplicate keys in indexing structures, Elmasri and Navathe [2015] outlines three main approaches: (1) including duplicate entries in the index, (2) using variable-length records with a repeating pointer field, or (3) keeping fixed-length index entries with a single entry per key value and an extra level of indirection to handle multiple pointers. FlatCityBuf adopts the third approach, which is "more commonly used" according to Elmasri and Navathe [2015], by implementing a payload section that stores the collection of feature offsets for each duplicate key. This design choice was made to maintain a simple implementation for search algorithms while efficiently handling attributes with potentially high duplicate cardinality. The fixed-length entries in the tree structure preserve the binary search efficiency, while the separate payload section accommodates the variable number of references without complicating the tree traversal logic.

- **Multi-type Support**: The index structure was extended to handle various attribute data types commonly found in 3D city models, including numeric types (integers, floating-point), string values, boolean flags, and temporal data (dates, timestamps).
- **Explicit Node Offsets**: While the original S+tree uses mathematical calculations to determine node positions, FlatCityBuf's implementation stores explicit byte offsets to child nodes. This modification simplifies the implementation without compromising performance. The parent node item has a 64-bit offset to the first child item of left child node.
- **Payload Pointer Mechanism**: To efficiently handle duplicate keys, the implementation uses a tag bit in the offset value to distinguish between direct feature references and pointers to the payload section. When the most significant bit is set, the remaining bits

encode an offset to the payload section where multiple feature offsets are stored consecutively. This approach minimizes both the storage overhead and redundant HTTP requests for unique keys while enabling support for duplicate keys.

These modifications ensure that the S+tree implementation is optimized for the specific characteristics of 3D city model data while preserving the performance advantages of the original algorithm.

4.5.3. Attribute Index Implementation

The attribute indexing system in FlatCityBuf is implemented as a binary encoded structure with four main components:

1. **Index Metadata**: Contains metadata about the index, including the column being indexed, branching factor, and number of unique values. This is stored in the header section of the file subsection 4.3.5.
2. **Tree Structure**: A hierarchical arrangement of nodes with keys and pointers. Though it's called as "tree", it's conceptual structure. The actual structure is a linear sequence of nodes. Both internal and leaf nodes are stored consecutively in the "flat" structure.
3. **Payload Section**: Stores arrays of feature offsets for duplicate key values. Each payload entry has a 32-bit length prefix that indicates the number of feature offsets that follow.

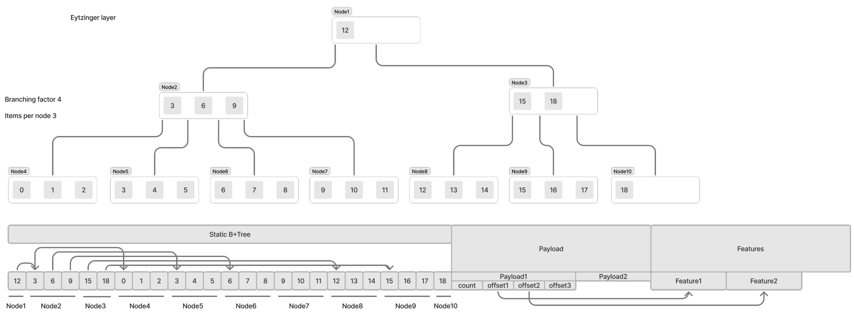


Figure 4.3.: Attribute index implementation in FlatCityBuf

4.5.4. Construction of the Attribute Index

The construction of the attribute index follows these processes:

1. Create pairs of attribute values and their corresponding feature offsets.
2. Sort the pairs by the attribute values.
3. Create the payload section by grouping the feature offsets for duplicate attribute values.
4. Build the tree structure with configuration of branching factor and the number of unique values. (This determines the height of the tree and the range of array for each level of the tree)

- Leaf nodes: branching factor -1 items are grouped together as one leaf node. Each item has a key and a u64 offset to either the feature or payload section.

- Internal nodes: branching factor items are grouped together as one internal node. The key value of an internal node is the minimum key of the subtree of its right child node.

5. Structure the tree from bottom to top and write it to the file in the order from top to bottom.

4.5.5. Serialization of Keys in the Tree

Key serialization in the attribute index is a critical aspect of the implementation, directly affecting both storage efficiency and query performance. FlatCityBuf implements type-specific serialization strategies that balance storage requirements, comparison efficiency, and implementation complexity.

Fixed-Length Value Serialization

Fixed-length values offer significant advantages for tree structures, enabling predictable node sizes and efficient binary search within nodes. FlatCityBuf serializes fixed-length values using the following strategies:

- **Integer Types**: Primitive integer types (i8, i16, i32, i64, u8, u16, u32, u64) are serialized directly in their native binary format using little-endian byte order. For example, a u64 value occupies exactly 8 bytes in the index structure.

- **Floating-Point Types**: IEEE 754 floating-point values [IEEE SA, 2019] use their native binary representation, with special handling for NaN values to ensure consistent ordering semantics. This is implemented using the OrderedFloat wrapper type, which provides total ordering for floating-point values while preserving their binary efficiency.

- **Temporal Types**: Date and timestamp values are serialized using a normalized representation that preserves chronological ordering. Timestamps are encoded as a composite of two components: an i64 representing seconds since the epoch, followed by a u32 representing nanosecond precision, both in little-endian order. This 12-byte representation supports the full range of ISO 8601 datetime values with timezone information [ISO, 2017].

- **Boolean Values**: Boolean values are encoded as a single byte (0 for false, 1 for true), aligning with common binary encodings while ensuring consistent sort order.

This direct serialization approach for fixed-length types minimizes both computational overhead during tree traversal and storage requirements in the index structure.

Variable-Length Value Serialization

Supporting variable-length keys in B+tree structures presents significant implementation challenges. As Elmasri and Navathe [2015] notes, variable-length keys can lead to unpredictable node sizes and uneven fan-out, complicating both the tree construction and traversal algorithms. This issue is particularly relevant for string attributes in 3D city models, where key lengths can vary substantially.

Modern database systems typically address this challenge through techniques such as prefix compression, where only the distinguishing prefix of each key is stored in non-leaf nodes. For example, when indexing last names, a non-leaf node might store only "Nac" and "Nay" as the discriminating prefixes between "Nachamkin" and "Nayuddin" [Elmasri and Navathe, 2015].

While implementing a full prefix compression scheme would be ideal, it would significantly increase the complexity of both the indexing algorithm and the format specification. After evaluating the trade-offs between implementation complexity and the practical requirements of 3D city model attribute data, FlatCityBuf adopts a pragmatic approach using fixed-length strings with a maximum length of 50 bytes. This length was selected based on analysis of common attribute values in 3D city datasets, where typical string attributes such as identifiers ("NL.IMBAG.Pand.0363100012345678"), city names ("Delft"), building types ("residential"), and similar values rarely exceed this length.

For strings shorter than the fixed length, padding with space characters ensures consistent key sizes throughout the tree structure. This approach simplifies implementation while still supporting the most common use cases found in 3D city model datasets. The space overhead from padding is generally acceptable given the relative infrequency of string attributes compared to numeric attributes in typical datasets.

4.5.6. Query Strategies

The attribute index implementation provides two core functions that enable efficient query execution:

- **find_exact_match**: Traverses the tree structure to locate an exact match for a specified key value.
- **find_partition_point**: Identifies the boundary positions within the tree for a given query value, essential for range-based operations.

These fundamental functions support both exact match and range queries. Range queries are implemented by determining lower and upper bounds using `find_partition_point` and then retrieving all results within those boundaries. For inequality queries, the implementation uses `find_exact_match` to identify the target item and then returns all items except the matched one. This query functionality aligns with the standard operators defined in OGC [2010]:

- `PropertyIsEqualTo`
- `PropertyIsNotEqualTo`
- `PropertyIsLessThan`
- `PropertyIsGreaterThan`
- `PropertyIsLessThanOrEqualTo`

- `PropertyIsGreaterThanOrEqualTo`
- `PropertyIsBetween`

For complex logical operations, the implementation supports compound queries by executing multiple index lookups and combining the results. For `AND` operations, it computes the intersection of result sets, while `OR` operations would use the union of results. Currently, only the `AND` logical operator is fully implemented.

4.5.7. Streaming S+Tree over HTTP

The index is structured to optimize for HTTP Range Requests, with several techniques employed to minimize network overhead:

- **Streaming search**: The search algorithm operates in a streaming fashion, requesting only the nodes necessary for query evaluation in sequential order. This approach ensures that even with large indices, the system avoids loading the entire tree structure into memory, significantly reducing resource requirements.
- **Payload Prefetching**: Proactively caches parts of the payload section during initial query execution, reducing HTTP requests for duplicate keys.
- **Batch Payload Resolution**: Collects multiple payload references during tree traversal and resolves them with consolidated HTTP requests.
- **Request Batching**: Groups adjacent node requests to minimise network roundtrips.
- **Block Alignment**: Nodes are aligned to fixed size boundaries to match typical file system and HTTP caching patterns.

During query execution, the system interprets the provided condition (e.g., `building_height > 25`) and traverses the appropriate attribute index to find matching features. The search algorithm adapts based on the condition type, using different traversal strategies for exact matches versus range queries. Results are returned as a set of feature offsets, which can then be used to retrieve the actual feature data from the features section of the file.

4.6. Feature Encoding

The feature encoding section of FlatCityBuf is responsible for the binary representation of 3D city objects and their associated data. This component preserves the semantic richness of the CityJSON model while leveraging FlatBuffers' efficient binary serialisation. The full schema definition for feature encoding can be found in section C.2.

4.6.1. CityFeature and CityObject Structure

FlatCityBuf implements the core structure of CityJSONSeq using the following FlatBuffers tables:

- **CityFeature** - *table (root object)* - The top-level container for city objects:
  - **id** - *string (key, required)* - Required string identifier, marked as a key field for fast lookup
  - **objects** - *Array of CityObject tables* - Collection of individual 3D features
  - **vertices** - *Array of Vertex structs* - Quantized X,Y,Z coordinates (int32)
  - **appearance** - *Appearance table* - Optional visual styling information
- **CityObject** - *table* - Individual 3D city objects:
  - **type** - *CityObjectType enum* - Object classification (Building, Bridge, etc.) following CityJSON types [CityJSON, 2024]
  - **id** - *string (key, required)* - Required string identifier, marked as a key field
  - **geographical_extent** - *GeographicalExtent struct* - 3D bounding box of the object
  - **geometry** - *Array of Geometry tables* - Shape information
  - **attributes** - *ubyte array* - Binary blob containing attribute values (interpretable via columns schema)
  - **columns** - *Array of Column tables* - Schema defining attribute types and names
  - **children** - *Array of string* - IDs referencing child objects
  - **children_roles** - *Array of string* - Descriptions of relationship roles
  - **parents** - *Array of string* - IDs referencing parent objects
  - **extension_type** - *string* - Optional type for extended objects (e.g., "+NoiseBuilding")

This structure maintains CityJSON's hierarchical organization while taking advantage of FlatBuffers' binary encoding and zero-copy access capabilities.

4.6.2. Geometry Encoding

Geometry in FlatCityBuf follows CityJSON's boundary representation (B-rep) model with flattened arrays for FlatBuffers encoding:

- **Geometry** - *table* - Container for geometric representation:
  - **type** - *GeometryType enum* - Geometric dimension type (0D-Point, 1D-LineString, etc.)
  - **lod** - *float* - Level of Detail value
  - **boundaries** - *Array of uint32* - Indices referencing vertices
  - **strings** - *Array of uint32* - Counts defining vertex groups
  - **surfaces** - *Array of uint32* - Counts defining string groups
  - **shells** - *Array of uint32* - Counts defining surface groups
  - **solids** - *Array of uint32* - Counts defining shell groups
  - **semantics_boundaries** - *Array of uint32* - Parallel arrays to boundaries for semantic classification
  - **semantics_values** - *Array of SemanticObject tables* - Semantic information for surfaces
- **SemanticObject** - *table* - Semantic classification of geometry parts:
  - **type** - *SemanticSurfaceType enum* - Surface classification (WallSurface, RoofSurface, etc.)
  - **extension_type** - *string* - Optional extended semantic type name
  - **attributes** - *ubyte array* - Binary blob containing semantic-specific attributes
  - **columns** - *Array of Column tables* - Schema defining attribute types and names
  - **parent** - *uint32* - Index to parent semantic object
  - **children** - *Array of uint32* - Indices to child semantic objects
- **GeometryInstance** - *table* - Reference to template geometry:
  - **transformation** - *TransformationMatrix struct* - 4×4 transformation matrix
  - **template** - *uint32* - Index referencing a template in the header section
  - **boundaries** - *Array of uint32* - Single-element array containing reference point index
- **Vertex** - *struct* - Quantized 3D coordinates:
  - **x** - *int32* - X coordinate, converted using header transform
  - **y** - *int32* - Y coordinate, converted using header transform
  - **z** - *int32* - Z coordinate, converted using header transform

Hierarchical Boundaries as Flattened Arrays

A key challenge in adapting CityJSON's recursive boundary representation to FlatBuffers is that FlatBuffers does not support nested arrays. FlatCityBuf addresses this by implementing a dimensional hierarchy encoded as parallel flattened arrays:

The encoding strategy follows a dimensional hierarchy from lowest to highest dimension:

1. **boundaries**: A single flattened array of integer vertex indices
2. **strings**: Array where each value indicates the number of vertices in each ring/boundary
3. **surfaces**: Array where each value indicates the number of strings/rings in each surface
4. **shells**: Array where each value indicates the number of surfaces in each shell
5. **solids**: Array where each value indicates the number of shells in each solid

For example, a simple triangle would be encoded as:

```
boundaries: [0, 1, 2]       // Indices of three vertices
strings: [3]                // Single string with 3 vertices
surfaces: [1]               // Single surface containing 1 string
```

A more complex structure such as a cube (a solid with 6 quadrilateral faces) would be encoded as:

```
boundaries: [0, 1, 2, 3, 0, 3, 7, 4, 1, 5, 6, 2, 4, 7, 6, 5, 0, 4, 5, 1, 2, 6, 7, 3]
strings: [4, 4, 4, 4, 4, 4]    // 6 strings with 4 vertices each
surfaces: [1, 1, 1, 1, 1, 1]   // 6 surfaces with 1 string each
shells: [6]                    // 1 shell with 6 surfaces
solids: [1]                    // 1 solid with 1 shell
```

Semantic Surface Encoding

Semantic surface information is encoded using a similar approach:

- **semantics_values**: Array of *SemanticObject* tables containing type classifications, attributes, and hierarchical relationships
- **semantics_boundaries**: Array of indices that reference entries in *semantics_values*, with a parallel structure to the geometry boundaries

This parallel structure allows each geometric component to have associated semantic information without requiring deeply nested structures. For example, in a building model where each face has a semantic classification (wall, roof, etc.), the *semantics_boundaries* array would have the same structure as the *boundaries* array, with each surface having a corresponding semantic value.

Through this flattened array approach, FlatCityBuf preserves the rich hierarchical structure of CityJSON geometries while conforming to FlatBuffers' efficiency-oriented constraints on data organization.

Geometry Template Encoding

FlatCityBuf implements CityJSON's template mechanism for efficient representation of repeated geometry patterns, a common requirement in urban environments where many buildings, street furniture items, or other objects share identical geometric structures. The template approach separates the geometry definition from its instantiation:

- **Template Definition**: Templates are defined once in the header section as full Geometry objects:
  - Templates use the same Geometry table format described previously for standard geometries
  - Template vertices are stored with double-precision coordinates (`DoubleVertex`) to maintain accuracy in the local coordinate system
  - All template vertices for all templates are stored in a single flat array (`templates_vertices`)
  - Indices within template boundaries reference positions in this dedicated template vertex array
- **Template Instantiation**: CityObjects reference templates through GeometryInstance tables:
  - `template`: A single unsigned integer index referencing a specific template in the header
  - `boundaries`: Contains exactly one index referencing a vertex in the feature's vertex array, which serves as the reference point for placement
  - `transformation`: A 4×4 transformation matrix (rotation, translation, scaling) that positions the template relative to the reference point

FlatCityBuf preserves CityJSON's template mechanism, which provides significant storage efficiency by storing repeated geometries once and referencing them with transformation parameters.

4.6.3. Materials and Textures

FlatCityBuf supports CityJSON's appearance model through the following structures:

- **Appearance** - *table* - Container for visual styling information:
  - **materials** - *Array of Material tables* - Surface visual properties definitions
  - **textures** - *Array of Texture tables* - Image mapping information
  - **vertices_texture** - *Array of Vec2 structs* - UV coordinates for texture mapping
  - **material_mapping** - *Array of MaterialMapping tables* - Links materials to surfaces
  - **texture_mapping** - *Array of TextureMapping tables* - Links textures to surfaces
  - **default_theme_material** - *string* - Default material theme identifier
  - **default_theme_texture** - *string* - Default texture theme identifier

- **Material** - *table* - Surface visual properties:
  - **name** - *string (required)* - Unique material identifier
  - **ambient_intensity** - *double* - Value from 0.0 to 1.0
  - **diffuse_color** - *Array of double* - RGB values from 0.0 to 1.0
  - **emissive_color** - *Array of double* - RGB values from 0.0 to 1.0
  - **specular_color** - *Array of double* - RGB values from 0.0 to 1.0
  - **shininess** - *double* - Value from 0.0 to 128.0
  - **transparency** - *double* - Value from 0.0 to 1.0
  - **is_smooth** - *boolean* - Flag for smooth shading
- **Texture** - *table* - Image mapping information:
  - **type** - *TextureFormat enum* - Format type (PNG, JPG)
  - **image** - *string (required)* - Image file name or URL
  - **wrap_mode** - *WrapMode enum* - Wrapping option (None, Wrap, Mirror, Clamp, Border)
  - **texture_type** - *TextureType enum* - Type classification (Unknown, Specific, Typical)
  - **border_color** - *Array of double* - RGBA values from 0.0 to 1.0
- **MaterialMapping** - *table* - Links materials to surfaces:
  - **theme** - *string* - Theme identifier (e.g., "summer", "winter")
  - **values** - *Array of uint32* - Indices to surfaces or boundaries
  - **material** - *uint32* - Index to the referenced material
- **TextureMapping** - *table* - Links textures to surfaces:
  - **theme** - *string* - Theme identifier (e.g., "summer", "winter")
  - **values** - *Array of uint32* - Indices to surfaces or boundaries
  - **texture** - *uint32* - Index to the referenced texture
  - **uv_indexes** - *Array of uint32* - Indices to UV coordinates

This implementation prioritizes efficient storage by referencing external texture files rather than embedding image data directly, enabling selective loading based on application requirements while maintaining full compatibility with CityJSON's appearance model.

Texture Storage Design Rationale

FlatCityBuf stores texture references rather than embedding texture data directly for several strategic reasons:

- **Performance Priority**: Enables rapid loading of geometric and semantic data without the overhead of large texture files when not required.
- **On-demand Loading**: Supports selective texture loading based on application needs, beneficial for analysis-focused use cases.
- **Size Management**: Maintains reasonable file sizes for large-scale datasets.
- **Web Efficiency**: Individual texture files can be cached by browsers or Content Delivery Network (CDN)s, significantly improving performance for repeated access in web applications.

This approach follows established patterns in formats like glTF, OBJ, and I3S, prioritizing operational efficiency over self-contained packaging for city-scale datasets.

4.6.4. Attribute Encoding

Attributes in FlatCityBuf are encoded as binary data with a schema defined through *Column* tables, which were detailed previously in subsection 4.3.5. Rather than repeating column structure information, this section focuses on the binary encoding strategy:

- **Attribute Binary Encoding** - Efficient type-specific serialization:
  - *Numeric types* - Native binary representation (little-endian)
  - *String* - Length-prefixed UTF-8 encoding
  - *Boolean* - Single byte (0 = false, 1 = true)
  - *Date/DateTime* - Standardized binary format
  - *Byte array* - Length-prefixed binary data
  - *Nested JSON* - Length-prefixed JSON string encoding of complex nested structures
  - *Null* - Not encoded to save space (null attributes are omitted from the binary representation)

FlatCityBuf encodes attributes as type-specific binary values with a corresponding schema definition. Each attribute is stored as a key-value pair where the key is the column index and the value is the binary representation of the attribute. This approach balances flexibility with reasonable performance while maintaining compatibility with the original CityJSON semantic model. The figure below illustrates how different attribute types are encoded in the binary format.

replace with proper figure

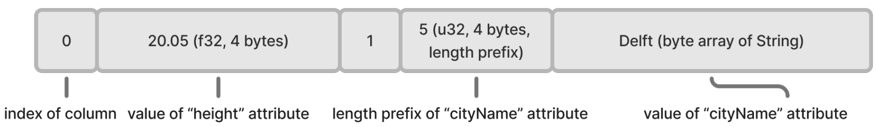


Figure 4.4.: Example of attribute encoding in FlatCityBuf

4.6.5. Extension Mechanism

FlatCityBuf provides comprehensive support for CityJSON's extension mechanism, which was previously detailed in subsection 4.3.4. While the extension structures are defined in the header, their implementation within actual city features requires specific encoding strategies that balance extensibility with performance.

Encoding of Extended City Objects

Extended city object types (those prefixed with "+") are encoded using a two-part strategy:

- A standard enum value `ExtensionObject` is used for the `type` field to distinguish whether the extended object or core object
- The actual extension type name (e.g., "+NoiseCityFurnitureSegment") is stored in the `extension_type` string field (This will be null for core objects)

Encoding of Extended Semantic Surfaces

Similarly, extended semantic surface types follow the same pattern:

- The `type` field uses the enum value `ExtraSemanticSurface`
- The specific type (e.g., "+ThermalSurface") is stored in the `extension_type` field (This will be null for core objects)

Extension Attribute Encoding

Extension-specific attributes are encoded using the same binary serialization mechanism as core attributes:

- Extension attributes are included in the same binary representation as standard attributes
- The schema for these attributes is stored alongside the `columns` of the `Header` table (See subsection 4.3.5)

During decoding:

- The same decoding logic is applied as for core attributes
- If needed, the application can identify extension attributes by checking if the column name begins with "+"

Unlike CityJSON, which references external schema files for extensions, FlatCityBuf's self-contained approach ensures that all extension information is available within a single file. This approach maintains the cloud-optimized philosophy of minimizing external dependencies while preserving full compatibility with the rich extension capabilities of CityJSON.

4.7. HTTP Range Requests and Cloud Optimisation

A critical component of cloud-optimised geospatial formats is their ability to support selective data retrieval without downloading entire datasets. FlatCityBuf achieves this capability through strategic implementation of HTTP Range Requests [Internet Engineering Task Force, 2014], enabling efficient partial data retrieval. This section details the technical implementation, optimisation strategies, and cross-platform compatibility of this mechanism.

4.7.1. Principles of Partial Data Retrieval

HTTP Range Requests, defined in RFC 7233 [RFC, 2010], allow clients to request specific byte ranges from server resources instead of entire files. This capability is fundamental to FlatCityBuf's cloud-optimised design. Since each feature in FlatCityBuf is length-prefixed, once the client knows the byte offset to a specific feature, it can request precisely the bytes needed. While data access patterns vary—from sequential access to spatially or attribute-indexed retrieval—the core principle remains consistent: fetch only the necessary data.

4.7.2. Range Request Workflow

The HTTP Range Request workflow in FlatCityBuf follows a carefully optimised sequential process:

1. **Header Retrieval**: The client first requests the magic bytes (8 bytes) and `Header` (described in subsection 4.3.5). This initial request provides essential metadata including coordinate reference systems, transformations, the total number of features, and index structure information etc..

2. **Index Navigation**: Based on query parameters (spatial bounding box or attribute conditions), the client selectively navigates the appropriate index structures:

   - For spatial queries, the client traverses only the relevant nodes of the packed Hilbert R-tree along the query path

   - For attribute queries, the client similarly traverses only the necessary portions of the appropriate S+Tree indices

3. **Feature Resolution**: Using byte offsets obtained from the indices, the client makes targeted range requests for specific features. The size of each feature is determined implicitly by the difference between consecutive offsets. The absolute byte offset of a feature within the file can be calculated by summing the size of the `Magic bytes`, the size of the `Header`, the size of the `indices`, and the relative offset of the feature.

4. **Progressive Processing**: Features are processed incrementally as they arrive, allowing applications to begin rendering or analysis before all data is received, significantly improving perceived performance.

This workflow enables efficient partial data retrieval by leveraging indexing strategies to minimize both the number of HTTP requests and the total data volume transferred.

Figure 4.5.: HTTP Range Request workflow in FlatCityBuf showing the sequential process of header retrieval, index navigation, and selective feature retrieval. The client makes targeted requests for specific byte ranges rather than downloading the entire dataset.

4.7.3. Optimisation Techniques

Network latency often dominates performance when accessing data over HTTP, with each request incurring significant overhead regardless of payload size. FlatCityBuf implements several techniques to minimise this overhead:

- **Request Batching**: Multiple feature requests are grouped into larger, consolidated HTTP requests rather than making individual requests for each feature. This approach significantly reduces the number of HTTP round trips, improving overall performance while minimizing network overhead.
- **Payload Prefetching**: When an attribute index is about to be used, the implementation proactively downloads a portion of its payload section. This anticipatory approach reduces latency for subsequent operations by having relevant data already available in memory when needed.
- **Streaming Process of Indices**: Both spatial and attribute indices implement a streaming approach where only the necessary node items in the tree structure are loaded when needed. Rather than loading entire index structures upfront, the system traverses the tree on demand, requesting only the relevant portions required for the current query.
- **Buffered HTTP Client**: The implementation uses a buffered HTTP client that caches previously fetched data ranges, avoiding redundant requests when overlapping ranges are accessed.

These optimisations work in concert to minimise the number of HTTP requests, resulting in significantly improved performance for cloud-based 3D city model applications.

4.7.4. Cross-Platform Implementation

FlatCityBuf provides range request capabilities across multiple platforms to maximise accessibility and integration options:

Cross-Platform Support

FlatCityBuf is implemented primarily as a Rust library that can be used in both native environments and web browsers. The same codebase is compiled to:

- Native Rust library for server-side applications and desktop GIS tools
- WebAssembly (WASM) module for browser-based applications with JavaScript interoperability

This cross-platform approach enables FlatCityBuf to work with both Rust's native HTTP clients and browser-based Fetch API implementations. The WASM implementation has one notable limitation: current browser WebAssembly implementations use a 32-bit memory model (4GB limit), which may constrain processing of country-level datasets. This limitation will be resolved with the upcoming WebAssembly Memory64 proposal [W3C, 2022].

Web prototype

To demonstrate FlatCityBuf's capabilities in web environments and illustrate practical user interactions with the data, a functional web prototype was developed. The prototype is publicly accessible at `https://fcb-web-prototype.netlify.app/`. It leverages the WebAssembly module of FlatCityBuf combined with TypeScript and React for the frontend implementation, with Cesium serving as the 3D map rendering engine.

The prototype operates on a substantial dataset covering approximately 20km $\times$ 20km of South Holland, Netherlands, stored as a single 3.4GB FlatCityBuf file. This file is delivered directly from Google Cloud Storage[Cloud, 2010], a serverless storage service, where it exists as a static file similar to images or videos, requiring no specialized server-side processing. Despite this large file size, the application remains responsive by utilizing the HTTP range request capabilities described earlier. Users can interact with the data through several query mechanisms:

- **Spatial queries**: Users can filter features either by defining a spatial bounding box or by placing a point on the map to retrieve features based on intersection or nearest-neighbor relationships.
- **Attribute queries**: The interface supports filtering features through attribute conditions (e.g., `building id = 1, height > 10m`), demonstrating the attribute index capabilities.
- **Data export**: Users can download the filtered subset of features in CityJSONSeq format, showcasing the format conversion capabilities.

This prototype effectively demonstrates how FlatCityBuf enables browser-based applications to work with large 3D city models without downloading the entire dataset, providing responsive performance even on consumer-grade hardware and network connections.

4.7.5. Integration with Cloud Infrastructure

The HTTP Range Request mechanism integrates seamlessly with modern cloud infrastructure. FlatCityBuf files can be served from standard object storage services like AWS S3, Google Cloud Storage, or Azure Blob Storage, all of which support range requests without additional server-side processing. This enables a serverless architecture where the client-side filtering approach eliminates the need for dedicated server-side processing. This infrastructure compatibility ensures that FlatCityBuf can be deployed in cost-effective cloud environments without requiring specialised application servers and databases.

Figure 4.6.: Web prototype of FlatCityBuf demonstrating spatial and attribute query capabilities on a 3.4GB dataset of South Holland.

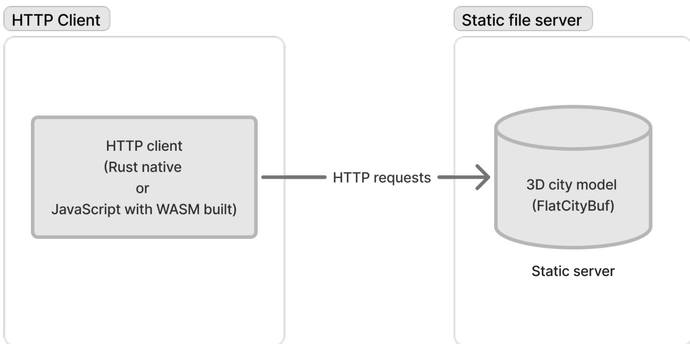


Figure 4.7.: Server architecture for FlatCityBuf. The client-side filtering approach eliminates the need for dedicated server-side processing.

5. Result

5.1. Overview

This chapter presents the results of comprehensive evaluations conducted to assess the performance and suitability of the proposed FlatCitybuf format against existing CityJSON encoding approaches. The evaluation followed three complementary methodologies to provide a holistic understanding of the format's capabilities.

The assessment framework employed three distinct methodological approaches. First, file size comparison was conducted to evaluate storage efficiency across different datasets and encoding formats.

Second, performance benchmarks were conducted on a laptop environment to evaluate the computational efficiency of the encoding format. These benchmarks measured read operation time for files of varying sizes, memory consumption during processing operations, and storage efficiency through file size comparisons. The benchmark utilised the datasets from Ledoux et al. [2024] and additional datasets from PLATEAU, providing direct comparability with previous studies on CityJSON and CityJSONSeq formats. All operations were conducted multiple times to ensure statistical reliability, with warm-up iterations to eliminate caching effects.

Third, to assess real-world application performance in web environments, browser-based benchmarks were conducted to measure the time required to fetch subsets of data. These tests focused on evaluating how efficiently the format enables selective data retrieval through HTTP Range requests, allowing clients to access only the specific portions of data they need rather than downloading entire datasets. This approach provides insights into the practical benefits of the format for web-based applications where bandwidth optimization and response time are critical factors.

The following sections present detailed results from each evaluation approach, followed by integrated analyses that synthesise findings across methodologies to provide comprehensive insights into the performance characteristics of the FlatCitybuf format.

5.2. File Size Comparison

Datasets

To evaluate file sizes and conduct both local and web-based benchmarks, we employed a diverse range of datasets from Ledoux et al. [2024] supplemented with additional datasets from PLATEAU. Comprehensive information regarding these datasets and their attributions is provided in section C.3.

Table 5.1.: The datasets used for the benchmark.

	dataset			size of file			attributes			
	CityObj	CityFeat	app.[a]	CityJSONSeq	FlatCityBuf	compr.[b]	verts	avg[c]	obj[d]	sem[e]
3DBAG	2221	1110		5.87 MB	6.23 MB	−6.02%	82612	74.43	37	1
3DBV	71634	71634		317.34 MB	280.92 MB	11.48%	4992893	69.70	64	0
Helsinki	77267	77231		412.44 MB	344.96 MB	16.36%	3039107	39.35	27	9
Helsinki_tex	77267	77231	tex	643.70 MB	545.29 MB	15.29%	3039107	39.35	28	9
Ingolstadt	379	55		3.84 MB	3.11 MB	19.09%	88001	1600.02	33	13
Montréal	294	294	tex	4.60 MB	4.80 MB	−4.38%	32242	109.67	0	0
NYC	23777	23777		95.45 MB	76.20 MB	20.17%	1044145	43.91	3	3
Railway	121	38	tex+mat	4.05 MB	3.75 MB	7.35%	73856	1943.58	3	0
Rotterdam	853	853	tex	2.69 MB	2.80 MB	−3.98%	26679	31.28	5	0
Vienna	1322	307		4.81 MB	4.12 MB	14.32%	47229	153.84	7	4
Zürich	198699	52834		247.12 MB	188.63 MB	23.67%	3564542	67.47	8	0
PLATEAU_bldg	10405	4307		76.94 MB	79.41 MB	−3.22%	147754	34.31	14	2
PLATEAU_brid	60	8		4.78 MB	5.21 MB	−9.09%	16357	2044.62	5	2
PLATEAU_rwy	412	412		4.15 MB	4.23 MB	−1.90%	5846	14.19	3	2
PLATEAU_tran	8136	8136		26.47 MB	26.62 MB	−0.54%	45992	5.65	3	2
PLATEAU_tun	21	3		4.86 MB	4.64 MB	4.41%	12306	4102.00	4	1
PLATEAU_veg	936	936		1.78 MB	2.32 MB	−30.50%	2567	2.74	3	0
Tokyo_PLATEAU	49764	38627		209.62 MB	216.76 MB	−3.41%	316607	8.20	15	1

[a] appearance: 'tex' indicates textures are stored; 'mat' indicates materials are stored
[b] compression factor is $\frac{\text{CityJSONSeq}-\text{FlatCityBuf}}{\text{CityJSONSeq}}$ (positive values indicate size reduction)
[c] average number of vertices per feature
[d] number of attributes in city objects
[e] number of semantic surface attributes in city objects

5.2.1. File size results

Table 5.1 presents a comparison of datasets in both CityJSONSeq and FlatCityBuf formats. The results demonstrate that FlatCityBuf encoding achieves superior compression for several datasets, including Helsinki, Ingolstadt, and New York City, with compression factors of 16.36%, 19.09%, and 20.17% respectively. Conversely, the PLATEAU datasets exhibit the opposite trend, with CityJSONSeq format demonstrating better storage efficiency.

Add whole 3DBAG

5.2.2. Analysis of file size results

Although subsection 5.2.1 provides a summary of file size comparisons, the factors influencing these outcomes require further investigation. This section analyses the underlying causes through controlled experiments with simplified datasets.

Level of detail

To examine how level of detail (LOD) affects file size, we conducted a series of tests using the TU Delft BK building model at various LOD levels. Each LOD variant was systematically extracted from the original model, with attributes and semantic information deliberately removed to isolate the effect of geometric complexity. Table 5.2 presents the results of this analysis.

Since each test dataset contains only a single city feature, we compare feature sizes rather than total file sizes. This approach is necessary because FlatCityBuf incorporates a larger header structure, which would disproportionately affect comparisons involving minimal features.

The results indicate that while file sizes naturally increase with higher levels of detail, there is no significant correlation between LOD and compression efficiency. Both formats exhibit proportional growth as geometric complexity increases. FlatCityBuf's compression advantage over CityJSONSeq remains consistent across most LOD levels, typically maintaining a 24-25% reduction in size.

Table 5.2.: Comparison of file sizes across different levels of detail for the TU Delft BK building model.

Dataset	FlatCityBuf[a]	CityJSONSeq[b]	Compression	Vertices
TUD BK All	139.75 kB	189.01 kB	26.08%	4549
TUD BK LOD0	12.77 kB	20.72 kB	38.11%	785
TUD BK LOD1.2	37.45 kB	49.40 kB	24.23%	1350
TUD BK LOD1.3	44.66 kB	59.25 kB	24.67%	1600
TUD BK LOD2.2	62.02 kB	82.74 kB	25.07%	2168

Average feature size in bytes in FlatCityBuf: $\frac{\text{Total FlatCityBuf size}}{\text{Number of features}}$
Average feature size in bytes in CityJSONSeq: $\frac{\text{Total CityJSONSeq size}}{\text{Number of features}}$

Attributes

To assess the impact of attributes on file size, we tested simple cube models from [CityJSON, 2019] with varying numbers of attributes. We systematically generated random attributes for each test case, examining both integer and string data types to determine their effect on compression efficiency. Table 5.3 presents the results of this analysis.

Table 5.3.: Comparison of file sizes with varying numbers of attributes for simple cube models.

Dataset	FlatCityBuf[a]	CityJSONSeq[b]	Compression	Attributes
10 attributes (int)	580 B	611 B	5.07%	10
100 attributes (int)	1.62 kB	2.44 kB	33.65%	100
1000 attributes (int)	12.17 kB	21.78 kB	44.13%	1000
10 attributes (string)	580 B	611 B	5.07%	10
100 attributes (string)	1.62 kB	2.44 kB	33.65%	100
1000 attributes (string)	12.17 kB	21.78 kB	44.13%	1000

Average feature size in bytes in FlatCityBuf: $\frac{\text{Total FlatCityBuf size}}{\text{Number of features}}$
Average feature size in bytes in CityJSONSeq: $\frac{\text{Total CityJSONSeq size}}{\text{Number of features}}$

The randomly generated attributes in our test datasets followed a consistent pattern, as shown in the example below:

```
{
  "type": "Building",
  "geometry": [...],
  "attributes": {
    "attr_1": "value_1",
    "attr_2": "value_2",
```

```
    "attr_3": "value_3",
    "attr_4": "value_4",
    "attr_5": "value_5",
    ...
    "attr_n": "value_n"
  }
}
```

For integer attribute tests, all values were randomly generated integers between 0 and 1000. For string attribute tests, values were randomly generated strings of varying lengths between 5 and 15 characters. This approach ensured a realistic representation of typical attribute data while maintaining controlled test conditions.

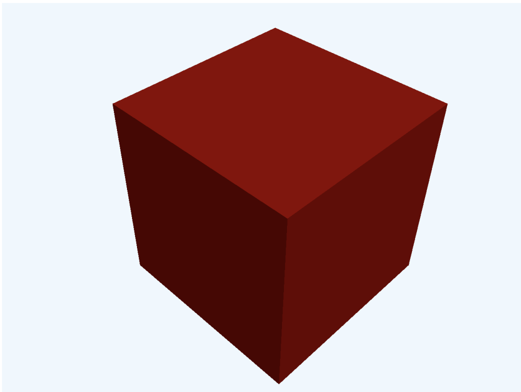


Figure 5.1.: Simple cube model used for attribute testing. This basic geometric structure provides a controlled environment for evaluating the impact of attributes on file size.

The results reveal a clear pattern: FlatCityBuf's compression advantage over CityJSONSeq increases substantially with the number of attributes. With only 10 attributes, the compression benefit is minimal at 5.07%, but rises markedly to 33.65% with 100 attributes and reaches 44.13% with 1000 attributes.

This efficiency stems from FlatCityBuf's architectural design, which stores the attribute schema once in the file header. Each feature subsequently references attributes using only a 2-byte (u16) index, while CityJSONSeq must replicate identical attribute keys across all features. Although additional attributes increase the header size, this overhead is distributed across all features in the dataset. The header remains relatively compact—even with 1000 attributes, it occupies only a few tens of kilobytes.

These characteristics render FlatCityBuf particularly advantageous for datasets containing numerous attributes. The same efficiency applies to semantic surface attributes, where the schema-based approach provides similar compression benefits when features contain multiple surfaces with rich semantic information.

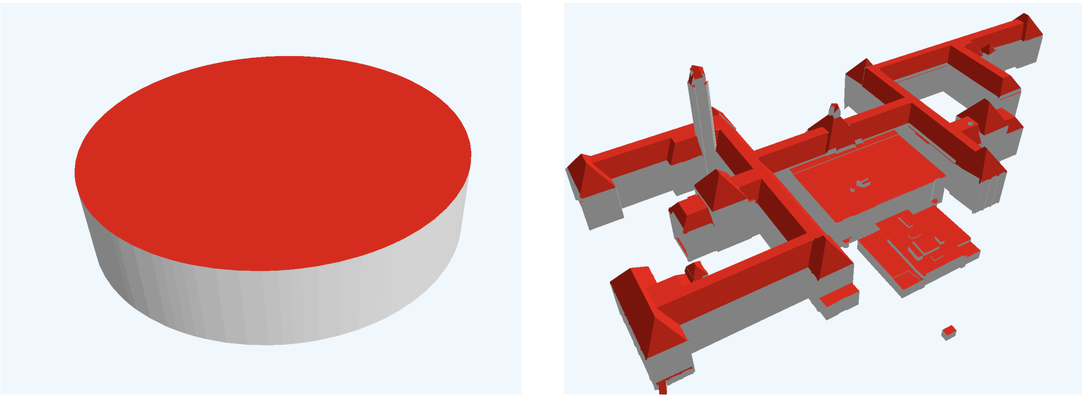
Geometry complexity

To evaluate how geometric complexity influences file size, we analysed models with varying numbers of vertices. The test utilised two geometrically distinct models from the TU Delft campus dataset—one simple and one complex. To isolate the effect of geometry, attributes and semantic information were removed, leaving only the essential geometric components required by CityJSON. Table 5.4 presents the numerical results of this analysis, while Figure 5.2 provides visual comparisons of the models.

Table 5.4.: Comparison of file sizes with varying geometric complexity.

Dataset	FlatCityBuf[(a)]	CityJSONSeq[(b)]	Compression	Vertices/Feature
TUD BK	139.75 kB	189.01 kB	26.06%	4549
TUD Simple	13.12 kB	15.42 kB	14.94%	340

Average feature size in bytes in FlatCityBuf: $\frac{\text{Total FlatCityBuf size}}{\text{Number of features}}$
Average feature size in bytes in CityJSONSeq: $\frac{\text{Total CityJSONSeq size}}{\text{Number of features}}$



(a) TUD Simple model (340 vertices/feature)

(b) TUD BK model (4549 vertices/feature)

Figure 5.2.: Visual comparison of models with different geometric complexity.

The results demonstrate that geometric complexity significantly affects compression efficiency, with FlatCityBuf achieving better compression for more intricate models. The TU Delft BK building model, containing 4549 vertices per feature, exhibits a higher compression rate of 26.06% compared to the simpler model with 340 vertices at 14.94%.

This differential appears to result from the expanding boundary field as geometry becomes more complex. FlatCityBuf employs a strongly typed representation of boundaries (using u32 integers) that maintains a constant size for encoding each vertex, whereas CityJSONSeq requires additional bytes due to its text-based format. This fundamental difference in geometry encoding becomes increasingly advantageous for FlatCityBuf as geometric complexity rises.

Vertices and coordinates

To investigate how coordinate scale affects file size, we conducted tests using identical cube geometries with varying coordinate magnitudes. These models contain the same number of vertices (8 per feature) but differ in their coordinate scale values. Table 5.5 presents the results of this analysis, utilising the same base models as in subsubsection 5.2.2.

Table 5.5.: Comparison of file sizes with varying coordinate scales.

Dataset	FlatCityBuf[(a)]	CityJSONSeq[(b)]	Compression	Scale
Cube (1)	476 B	370 B	−28.65%	1
Cube (10)	476 B	459 B	−3.70%	10
Cube (1k)	476 B	507 B	6.11%	1,000
Cube (1M)	476 B	579 B	17.79%	1,000,000

Average feature size in bytes in FlatCityBuf: $\frac{\text{Total FlatCityBuf size}}{\text{Number of features}}$
Average feature size in bytes in CityJSONSeq: $\frac{\text{Total CityJSONSeq size}}{\text{Number of features}}$

The results reveal an intriguing relationship between coordinate scale and file size in both formats. FlatCityBuf maintains a consistent size of 476 bytes regardless of coordinate magnitude, demonstrating its fixed-size binary encoding for numeric values. In contrast, CityJSONSeq's file size increases proportionally with larger coordinate values, growing from 370 bytes with single-digit coordinates to 579 bytes with million-scale coordinates.

This behaviour occurs because FlatCityBuf stores coordinates as fixed-size 32-bit integers, while CityJSONSeq, being a text-based format, requires more characters to represent larger numbers. Consequently, FlatCityBuf transitions from being less efficient than CityJSONSeq for small coordinate values (-28.65%) to substantially more efficient for large coordinate values (17.79%).

This characteristic explains the pattern observed in subsection 5.2.1. FlatCityBuf demonstrates lower storage efficiency for PLATEAU datasets, likely because these datasets employ geographic coordinate systems with values typically between -180 and 180. Since CityJSON quantises coordinates through the `Transform` field, latitude and longitude values can be represented as relatively small integers. Conversely, datasets where FlatCityBuf performs better—such as NYC and Helsinki—use local coordinate systems (in metres) with larger internal values, resulting in improved compression efficiency with FlatCityBuf.

Summary of File Size Analysis

The comprehensive analysis of various factors affecting file size reveals distinct patterns in the compression performance of FlatCityBuf compared to CityJSONSeq:

- **Level of Detail:** The analysis demonstrates that geometric detail levels have minimal impact on compression efficiency. While file sizes naturally increase with higher LODs, the compression advantage of FlatCityBuf remains relatively consistent at approximately 24-25% across different levels of geometric complexity.
- **Attribute Quantity:** The number of attributes significantly influences compression performance. FlatCityBuf's efficiency increases dramatically with attribute count, from minimal compression (5.07%) with 10 attributes to substantial compression (44.13%) with 1000 attributes. This progressive advantage stems from FlatCityBuf's schema-based approach that eliminates redundant attribute key storage.
- **Geometric Complexity:** More intricate geometries benefit from improved compression with FlatCityBuf. As boundary fields expand with geometric complexity, FlatCityBuf's fixed-size numeric representation provides greater efficiency compared to the text-based

encoding of CityJSONSeq, increasing compression from 14.94% for simple geometries to 26.06% for complex models.

- **Coordinate Scale:** The magnitude of coordinate values has a significant impact on compression efficiency. FlatCityBuf's constant-size integer representation maintains consistent file sizes regardless of coordinate scale, while CityJSONSeq requires more space for larger values. This creates a transition from inferior compression (-28.65%) with small coordinate values to superior compression (17.79%) with large coordinate values.

These findings elucidate the observed variations in compression performance across different datasets in Table 5.1. FlatCityBuf demonstrates optimal performance for datasets with numerous attributes, complex geometries, and large-scale coordinate systems, while CityJSONSeq may retain advantages for simpler datasets with limited attributes and smaller coordinate values.

5.3. Benchmark on Local Environment

This section presents a comprehensive performance evaluation of the FlatCityBuf format conducted in a controlled local environment. The analysis focuses on critical metrics including read operations, memory utilisation, and processing efficiency to establish a thorough understanding of the format's performance characteristics.

5.3.1. Test Environment

All benchmarks were executed within a consistent hardware and software configuration to ensure reliability and reproducibility:

- **Hardware:** Apple MacBook Pro with M1 Max chip, 32GB unified memory
- **Operating System:** macOS Sequoia 15.4
- **Filesystem:** APFS (Apple File System)
- **Storage:** 1TB SSD with approximately 200GB available capacity
- **Runtime Environment:** Rust 1.86.0, with optimised release builds

5.3.2. Measurement Parameters

The benchmark framework captured multiple performance dimensions through the following key indicators:

- **Read Performance:** Time required to deserialise the file and construct the complete CityJSON structure in memory, measured in milliseconds with microsecond precision
- **Memory Efficiency:** Peak Resident Set Size (RSS) during file processing, providing an accurate measurement of maximum memory requirements
- **Computational Overhead:** CPU utilisation percentage during operations, calculated as an average across the entire process lifecycle

These parameters were systematically measured across all encoding formats-CityJSONSeq, CBOR, BSON, and FlatCityBuf—to facilitate direct performance comparisons. The subsequent sections present a detailed analysis of these measurements and their implications for practical applications.

Read Performance FlatCityBuf vs CityJSONSeq

The performance comparison between FlatCityBuf and CityJSONSeq was conducted across multiple datasets, measuring CPU utilization, processing time, and memory consumption as key metrics. Table 5.6 presents these results.

Table 5.6.: Performance comparison between CityJSONSeq and FlatCityBuf

	CPU Utilization			Processing Time			Memory Consumption		
Dataset	cjseq	FCB	Ratio[a]	cjseq	FCB	Ratio[a]	cjseq	FCB	Ratio[a]
3DBAG	19.30%	2.10%	9.21×	59.00 ms	7.00 ms	8.48×	41.67 MB	10.81 MB	3.85×
3DBV	97.79%	41.92%	2.33×	4.03 s	141.00 ms	28.51×	287.77 MB	296.58 MB	0.97×
Helsinki	97.75%	44.77%	2.18×	3.71 s	159.00 ms	23.29×	1.77 GB	1.77 GB	1.00×
Ingolstadt	13.82%	0.87%	15.82×	39.00 ms	1.00 ms	36.13×	1.86 GB	1.85 GB	1.01×
Montréal	23.67%	1.12%	21.13×	59.00 ms	1.00 ms	46.62×	2.05 GB	2.05 GB	1.00×
NYC	94.91%	14.55%	6.52×	924.00 ms	44.00 ms	20.93×	2.15 GB	2.15 GB	1.00×
Rotterdam	8.33%	0.99%	8.41×	22.00 ms	1.00 ms	14.25×	942.89 MB	940.53 MB	1.00×
Vienna	15.55%	1.02%	15.21×	48.00 ms	2.00 ms	19.11×	1.02 GB	1.01 GB	1.00×
Zürich	97.46%	55.11%	1.77×	1.98 s	162.00 ms	12.12×	868.83 MB	1.02 GB	0.83×
Tokyo (PLATEAU)	91.34%	32.57%	2.80×	2.19 s	99.00 ms	21.95×	47.27 MB	15.00 MB	3.15×
PLATEAU_brid	33.61%	0.01%	2826.56×	91.00 ms	0.00 ms[b]	103.64×	1.03 GB	1.01 GB	1.01×
PLATEAU_rwy	14.62%	-0.25%[c]	-58.98×	44.00 ms	4.00 ms	9.06×	1.11 GB	1.12 GB	1.00×
PLATEAU_tran	86.61%	2.70%	32.05×	267.00 ms	14.00 ms	18.11×	416.45 MB	668.31 MB	0.62×
PLATEAU_tun	15.95%	0.58%	27.40×	52.00 ms	2.00 ms	22.42×	287.77 MB	483.50 MB	0.60×
PLATEAU_veg	91.11%	12.59%	7.24×	938.00 ms	60.00 ms	15.55×	212.42 MB	294.53 MB	0.72×

[a] Ratio = CityJSONSeq metric / FlatCityBuf metric (higher values indicate better FlatCityBuf performance)
[b] Time recorded as 0 ms due to measurement precision limitations for very fast operations
[c] Negative CPU utilization may indicate measurement noise for very small operations
Note: cjseq = CityJSONSeq, FCB = FlatCityBuf

Check if the memory consumption is correct. Also take benchmark on better environment

The performance comparison reveals significant advantages for FlatCityBuf across multiple metrics. CPU utilization is substantially lower for FlatCityBuf across all datasets, ranging from 1.77× to over ??? improvement. Processing time shows even more dramatic improvements, with FlatCityBuf consistently processing data between 8× and 46× faster than CityJSONSeq. Memory consumption results are mixed, with FlatCityBuf showing notable advantages for some datasets (particularly 3DBAG and Tokyo PLATEAU) while requiring slightly more memory for others.

Read performance FlatCityBuf vs CBOR

The performance comparison between FlatCityBuf and CBOR was conducted using the same datasets and measurement methodology. Table 5.7 presents these results.

write something about the results. Also take benchmark on better environment

Read performance FlatCityBuf vs BSON

The performance comparison between FlatCityBuf and BSON followed the same methodology as the previous comparisons. Table 5.8 presents the detailed results.

Write some analysis about the results. Also take benchmark on better environment

Table 5.7.: Performance comparison between CBOR and FlatCityBuf

	CPU Utilization			Processing Time			Memory Consumption		
Dataset	CBOR	FCB	Ratio[a]	CBOR	FCB	Ratio[a]	CBOR	FCB	Ratio[a]
3DBAG	31.81%	2.10%	15.18×	91.00 ms	7.00 ms	12.97×	155.47 MB	10.81 MB	14.38×
3DBV	85.57%	41.92%	2.04×	6.50 s	141.00 ms	46.00×	1.32 GB	296.58 MB	4.56×
Helsinki	84.68%	44.77%	1.89×	8.85 s	159.00 ms	55.49×	1.27 GB	1.77 GB	0.72×
Ingolstadt	15.14%	0.87%	17.33×	48.00 ms	1.00 ms	44.85×	1.92 GB	1.85 GB	1.03×
Montréal	15.93%	1.12%	14.22×	61.00 ms	1.00 ms	48.70×	2.06 GB	2.05 GB	1.01×
NYC	96.24%	14.55%	6.62×	1.35 s	44.00 ms	30.47×	1.04 GB	2.15 GB	0.48×
Rotterdam	11.93%	0.99%	12.04×	30.00 ms	1.00 ms	18.95×	946.36 MB	940.53 MB	1.01×
Vienna	18.54%	1.02%	18.14×	57.00 ms	2.00 ms	22.45×	1.02 GB	1.01 GB	1.01×
Zürich	96.21%	55.11%	1.75×	3.72 s	162.00 ms	22.81×	942.11 MB	1.02 GB	0.90×
Tokyo (PLATEAU)	86.76%	32.57%	2.66×	3.71 s	99.00 ms	37.14×	924.42 MB	15.00 MB	61.63×
PLATEAU_brid	30.03%	0.01%	2525.25×	68.00 ms	0.00 ms[b]	78.02×	1.07 GB	1.01 GB	1.06×
PLATEAU_rwy	15.74%	-0.25%[c]	-63.51×	47.00 ms	4.00 ms	9.63×	1.10 GB	1.12 GB	0.98×
PLATEAU_tran	88.01%	2.70%	32.57×	326.00 ms	14.00 ms	22.14×	307.84 MB	668.31 MB	0.46×
PLATEAU_tun	56.14%	0.58%	96.46×	222.00 ms	2.00 ms	94.80×	293.31 MB	483.50 MB	0.61×
PLATEAU_veg	92.28%	12.59%	7.33×	1.09 s	60.00 ms	18.00×	727.64 MB	294.53 MB	2.47×

[a] Ratio = CBOR metric / FlatCityBuf metric (higher values indicate better FlatCityBuf performance)
[b] Time recorded as 0 ms due to measurement precision limitations for very fast operations
[c] Negative CPU utilization may indicate measurement noise for very small operations
Note: FCB = FlatCityBuf

Table 5.8.: Performance comparison between BSON and FlatCityBuf

	CPU Utilization			Processing Time			Memory Consumption		
Dataset	BSON	FCB	Ratio[a]	BSON	FCB	Ratio[a]	BSON	FCB	Ratio[a]
3DBAG	34.24%	2.10%	16.34×	118.00 ms	7.00 ms	16.80×	276.41 MB	10.81 MB	25.56×
3DBV	86.56%	41.92%	2.06×	11.26 s	141.00 ms	79.70×	1.83 GB	296.58 MB	6.33×
Helsinki	89.31%	44.77%	1.99×	11.47 s	159.00 ms	71.93×	1.82 GB	1.77 GB	1.03×
Ingolstadt	31.23%	0.87%	35.74×	82.00 ms	1.00 ms	76.56×	2.04 GB	1.85 GB	1.10×
Montréal	80.62%	1.12%	71.96×	167.00 ms	1.00 ms	132.33×	2.13 GB	2.05 GB	1.04×
NYC	97.09%	14.55%	6.67×	1.82 s	44.00 ms	41.19×	924.05 MB	2.15 GB	0.42×
Rotterdam	28.64%	0.99%	28.91×	67.00 ms	1.00 ms	42.41×	995.86 MB	940.53 MB	1.06×
Vienna	30.45%	1.02%	29.78×	79.00 ms	2.00 ms	31.17×	1.02 GB	1.01 GB	1.01×
Zürich	92.20%	55.11%	1.67×	6.63 s	162.00 ms	40.67×	1.88 GB	1.02 GB	1.84×
Tokyo (PLATEAU)	85.47%	32.57%	2.62×	8.86 s	99.00 ms	88.79×	1.26 GB	15.00 MB	86.22×
PLATEAU_brid	83.38%	0.01%	7011.68×	179.00 ms	0.00 ms[b]	202.83×	1.15 GB	1.01 GB	1.13×
PLATEAU_rwy	31.74%	-0.25%[c]	-128.03×	91.00 ms	4.00 ms	18.58×	835.27 MB	1.12 GB	0.73×
PLATEAU_tran	92.65%	2.70%	34.29×	582.00 ms	14.00 ms	39.46×	492.06 MB	668.31 MB	0.74×
PLATEAU_tun	85.56%	0.58%	147.01×	253.00 ms	2.00 ms	108.12×	347.58 MB	483.50 MB	0.72×
PLATEAU_veg	91.97%	12.59%	7.31×	2.57 s	60.00 ms	42.58×	885.19 MB	294.53 MB	3.01×

[a] Ratio = BSON metric / FlatCityBuf metric (higher values indicate better FlatCityBuf performance)
[b] Time recorded as 0 ms due to measurement precision limitations for very fast operations
[c] Negative CPU utilization may indicate measurement noise for very small operations
Note: FCB = FlatCityBuf

Summary of local environment benchmark

To summarise the results of the local environment benchmark, we especially focus on the comparison between FlatCityBuf and CityJSONSeq since CityJSONSeq is the most standard data format of CityJSON at the moment.

- CPU utilization: FlatCityBuf is more efficient than CityJSONSeq in all cases. The best case achieved 32.05× improvement in CPU utilization while the worst case achieved still 1.77× improvement. While the Ratio is sometimes important, since CPU utilization limit is 100%, for larger datasets it tends to be close since FlatCityBuf also utilises more CPU resources. Thus, it achieves better ratio for smaller datasets. However, ratio isn't the only metric to consider. The important thing is that FlatCityBuf can utilise less CPU resources for the same amount of data in comparison to CityJSONSeq.

- Processing time: Processing time is more considarable as its pirmary objective of the research. The best case achieved 46.62× improvement (Montréal) while the worst case achieved 8.48× improvement (3DBAG). The program can save a lot of time for large datasets.

- Memory consumption: Memory consumption is also an important metric. The best case achieved 3.11× improvement (Tokyo) while the worst case achieved 0.62× improvement (PLATEAU_tran).

check again

5.3.3. Benchmark over the web

write something about the results

6. Discussion

This chapter discusses the implications of our experimental results and their broader significance for 3D city modelling applications.

6.1. Use Cases of FlatCityBuf

This section examines the most appropriate application scenarios for the FlatCityBuf format based on its demonstrated performance characteristics.

6.1.1. Flexible Data Download

Providing users with the ability to download specific data of interest represents one of the most valuable applications of 3D city models, particularly within open data initiatives. Existing services such as 3DBAG offer download functionality for CityJSON data in various formats including CityJSON, OBJ, and GeoPackage. However, these services typically constrain users to downloading predefined tiles rather than precisely the data matching their specific requirements.

HideBa [2025] demonstrates a web application that enables users to download precisely the data they require. This implementation successfully showcases FlatCityBuf's capability to facilitate targeted data retrieval. Through its attribute indexing mechanism, users can download filtered datasets based on specific criteria, such as CityFeatures exceeding 100 metres in height.

6.1.2. Data Analysis

As demonstrated by the performance benchmarks, FlatCityBuf excels in read operations compared to alternative data formats, making it particularly suitable for analysing large-scale datasets. Computational Fluid Dynamics in urban environments, for instance, requires processing substantial volumes of detailed geometric data. Such analyses typically demand significant memory and computational resources. FlatCityBuf can process these data more efficiently by leveraging its zero-copy capability.

The format also simplifies analytical workflows. Conventional approaches to large-scale data processing often require chunking data across multiple files, necessitating additional programming to manage file aggregation. In contrast, FlatCityBuf encapsulates data in a single file that can be efficiently loaded and accessed, even in web-based environments, streamlining analytical processes.

6.2. Impact on Server Architecture

FlatCityBuf introduces significant opportunities for simplifying server architectures for 3D city model delivery.

6.2.1. Traditional Server Architecture

Conventional server architectures for 3D city models typically employ both application and database servers. For example, Technical University of Munich utilises PostgreSQL or Oracle as the database server with PostgREST API [PostgREST, 2017] providing data access through its toolchain. Similarly, the 3DBAG API uses PostgreSQL as its database server and Flask (Python) as the application server.

In contrast, FlatCityBuf operates as a static file, requiring only a basic HTTP server such as Nginx for data distribution. This approach aligns with modern cloud service offerings, where providers like AWS S3 and Google Cloud Storage offer optimised solutions for serving static content.

6.2.2. Cloud Architecture Advantages

6.2.3. Scalability

Scalability presents a significant challenge in traditional server architectures. These systems typically employ Relational Database Management Systems (RDBMS) that often encounter scaling limitations. Common mitigation strategies include sharding and replication (horizontal scaling) or resource expansion (vertical scaling), both requiring additional computational, memory, and storage resources.

FlatCityBuf circumvents these challenges by functioning as a static file that can leverage cloud providers' inherent scalability and high availability infrastructure. This characteristic offers substantial benefits for applications built on 3D city model data. Service providers can host static FlatCityBuf files on standard servers, allowing unrestricted access for various use cases without implementing the rate-limiting mechanisms often necessary with traditional server architectures.

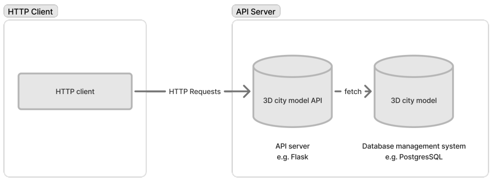
6.2.4. Cost-effectiveness

FlatCityBuf contributes significantly to operational cost-effectiveness. Although precise server costs vary according to specific use cases, hosting static files through cloud service providers is generally substantially more economical than maintaining dedicated database and application servers.

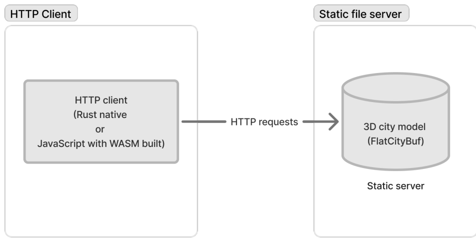
add more description of comparison

6.3. Limitations

Despite its advantages in simplicity, scalability, and cost-effectiveness, FlatCityBuf does present certain limitations that warrant consideration.



(a) Traditional server architecture with database and application servers



(b) Simplified FlatCityBuf architecture

Figure 6.1.: Comparison between traditional and FlatCityBuf server architectures. The proposed method eliminates the need for complex database infrastructure by leveraging static file hosting with built-in spatial and attribute indices.

6.3.1. Query Flexibility

While FlatCityBuf supports both spatial and attribute indexing, its query capabilities remain more constrained than those of specialised spatial database applications. Traditional approaches employing RDBMS with spatial indexing provide more comprehensive query functionality. For instance, 3DCityDB enables filtering by LoD, CityObject type, and various other parameters, whereas FlatCityBuf primarily supports attribute-based filtering. Similarly, regarding spatial functions, 3DCityDB can utilise the extensive spatial capabilities of PostGIS, while FlatCityBuf currently only implements bounding box queries. Consequently, FlatCityBuf is optimised for scenarios requiring relatively straightforward filtering conditions.

6.3.2. Client-side Application Complexity

Although FlatCityBuf simplifies server architecture, it introduces additional complexity in client-side applications, which must implement logic for loading and processing the format. By comparison, OGC API and equivalent Web API services adhere to standardised designs that can be utilised by any client application—whether accessed through command-line interfaces, browsers, or mobile applications. While FlatCityBuf supports cross-platform applications, it requires language-specific or platform-specific library implementations.

6.3.3. Update Complexity

Zero-copy data formats like FlatCityBuf generally present challenges for data updates due to their relatively rigid structure. Fixed-size data types such as integers or floating-point numbers cannot be dynamically converted to alternative types. Furthermore, since the format contains immutable spatial and attribute indices, updating the data necessitates rewriting the entire file. This characteristic renders FlatCityBuf less suitable for frequently updated datasets, positioning it instead as an optimal solution for data analysis and efficient download services.

7. Conclusion and Future Work

7.1. Research Summary

This research addressed the limitations of existing 3D city model formats in cloud environments by optimizing CityJSONSeq encoding through FlatCityBuf, a binary format leveraging FlatBuffers serialization with spatial and attribute indexing mechanisms.

The main contributions of this research include:

- A hierarchical FlatBuffers schema with five components (magic bytes, header section, spatial index, attribute index, and features section) enabling zero-copy access and 10-20× faster retrieval times while maintaining CityJSON compatibility
- Dual indexing mechanisms: Packed Hilbert R-tree for spatial queries with logarithmic-time retrieval, and Static B+Tree (S+Tree) for attribute-based queries supporting exact matches, ranges, and complex filtering
- HTTP Range Request optimization through explicit file alignment boundaries, enabling efficient retrieval of specific data subsets without downloading entire datasets

Benchmarks confirmed sub-second performance even with datasets containing hundreds of thousands of features. The approach eliminates complex database infrastructure, reduces operational costs through static file hosting, and maintains fast response times across large datasets.

7.2. Limitations

Despite its advantages, FlatCityBuf has several limitations. The query capabilities are more constrained than specialized spatial database applications. While FlatCityBuf implements bounding box queries and attribute filtering, it lacks advanced spatial operations and complex filtering available in systems like 3DCityDB with PostGIS. This limitation restricts its applicability in scenarios requiring sophisticated spatial analysis or complex query patterns.

The format introduces complexity in client-side applications that must implement custom loading and processing logic, unlike standardized OGC API services with consistent interfaces. This presents a potential adoption barrier, particularly for developers unfamiliar with binary formats or HTTP Range Requests. Without standardized libraries across multiple platforms, integration into existing workflows requires additional development effort.

FlatCityBuf's rigid structure presents challenges for data updates, requiring rewriting entire files when modifying data. Fixed-size data types cannot be dynamically converted, and the immutable spatial and attribute indices necessitate regenerating the entire file during updates. This makes the format more suitable for read-intensive applications than dynamic content management systems with frequent updates.

7.3. Future Work

Based on the research findings and identified limitations, several promising directions for future work emerge.

Expanding language support beyond Rust would significantly enhance the format's accessibility and ecosystem integration. Languages with garbage collection mechanisms—such as Python, JavaScript, and Java—present particularly interesting implementation targets. These languages manage memory differently than Rust, which could impact performance characteristics of zero-copy operations. Implementation in Python would enable seamless integration with geospatial analysis workflows, while JavaScript support would facilitate web-based visualization without WebAssembly. Testing performance across these languages would provide valuable insights into optimization strategies for different memory management approaches.

Investigating alternative serialization frameworks could reveal different efficiency patterns. Column-oriented formats like Apache Parquet warrant exploration, particularly for analytical workloads involving selective attribute access. Such formats excel at accessing specific fields across many records, potentially offering significant advantages for city-scale analytics where only certain properties (like building heights or energy consumption) are needed. Future research should quantify these trade-offs through comparative benchmarks across various query patterns and datasets sizes.

While a web prototype for FlatCityBuf exists, it currently only displays data as JSON without geometric visualization. Developing specialized web viewers would demonstrate the format's practical benefits in interactive contexts. Progressive loading strategies could enable smooth navigation of massive datasets on bandwidth-constrained devices by initially loading lower-detail geometries and enhancing detail as users zoom, significantly improving user experience while leveraging the format's efficient partial data retrieval mechanisms.

FlatCityBuf demonstrates that FlatBuffers encoding with Packed Hilbert R-tree and Static B+Tree indexing combined with HTTP Range requests effectively optimizes CityJSONSeq encoding. Despite limitations, it represents a significant advancement in cloud-optimized 3D city model storage that bridges the gap between comprehensive 3D city models and cloud-native data access.

A. Reproducibility self-assessment

A.1. Marks for each of the criteria

Figure A.1.: Reproducibility criteria to be assessed.

Grade/evaluate yourself for the 5 criteria (giving 0/1/2/3 for each):

1. input data
2. preprocessing
3. methods
4. computational environment
5. results

A.2. Self-reflection

A self-reflection about the reproducibility of your thesis/results.

We expect maximum 1 page here.

For example, if your data are not made publicly available, you need to justify it why (perhaps the company prevented you from doing this).

B. Some UML diagrams

Figure B.1.: The UML diagram of something that looks important.

C. FlatCityBuf Schema

C.1. Header

add description of header

```
table Header {
  version: string;
  transform: Transform;
  reference_system: ReferenceSystem;
  geographical_extent: GeographicalExtent;
  identifier: string;
  title: string;
  reference_date: string;
}
```

Listing C.1: Header schema

C.2. Feature

add description of feature

```
table Feature {
  id: string;
  objects: [CityObject];
  vertices: [Vertex];
  appearance: Appearance;
}
```

Listing C.2: Feature schema

C.3. Tables

Bibliography

3DBAG. 3d bag web services, 2023. URL https://docs.3dbag.nl/nl/delivery/webservices/.

Vladimir Agafonkin. Flatbush: A very fast static spatial index for 2d points and rectangles in javascript, 2010. URL https://github.com/mourner/flatbush.

Amazon Web Services. Amazon redshift, 2012. URL https://aws.amazon.com/redshift/.

Apache Parquet Contributors. Parquet format specification. URL https://github.com/apache/parquet-format.

Apache Software Foundation. Apache parquet, 2013. URL https://parquet.apache.org/.

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, 2015. doi: 10.3390/ijgi4042842.

F. Biljecki, K. Kumar, and C. Nagel. CityGML Application Domain Extension (ADE): overview of developments. *Open Geospatial Data Software and Standards*, (13), 2018. doi: 10.1186/s40965-018-0055-6.

Eduard Bröse. ZeroCopy: Techniques, Benefits and Pitfalls, 2008. URL https://www.semanticscholar.org/paper/ZeroCopy-%3A-Techniques-%2C-Benefits-and-Pitfalls-washuu/4931bbb1b96d10f89b7406bb38757dcc2a10a435.

BuildZero.Org. Open City Model, 2025. URL https://github.com/opencitymodel/opencitymodel.

CityJSON. Cityjson dataset, 2019. URL https://www.cityjson.org/datasets/.

CityJSON. CityJSON Specification 2.0.1, 2024. URL https://www.cityjson.org/specs/2.0.1/. Accessed: 2024-11-26.

Google Cloud. Cloud storage, 2010. URL https://cloud.google.com/storage?hl=en.

Cloud-Native Geospatial Foundation. Cloud-Optimised Geospatial Formats Guide, 2023. URL https://guide.cloudnativegeo.org/. Accessed: 2024-12-17.

Danny Cohen. On holy wars and a plea for peace. *IEEE Computer*, October 1981. URL https://ieeexplore.ieee.org/document/1667115.

ECMA International. JSON, 2013. URL https://www.json.org/json-en.html. Accessed: 2024-12-17.

Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson plc, 2015.

FlatGeobuf. FlatGeobuf GitHub Repository. URL https://github.com/flatgeobuf/flatgeobuf. Accessed: 2024-12-17.

FlatGeobuf. FlatGeobuf, 2020. URL https://flatgeobuf.org/. Accessed: 2024-12-17.

GeoParquet Contributors. Geoparquet, 2024. URL https://geoparquet.org/.

Google. Protocol Buffers, 2008. URL https://protobuf.dev/. Accessed: 2024-12-17.

Google. BigQuery, 2011. URL https://cloud.google.com/bigquery. Accessed: 2024-12-18.

Google. FlatBuffers, 2014a. URL https://flatbuffers.dev/. Accessed: 2024-12-17.

Google. C++ Benchmarks, 2014b. URL https://flatbuffers.dev/flatbuffers_benchmarks.html. Accessed: 2025-01-13.

Google. Flatbuffers: Memory efficient serialization library, 2014. URL https://google.github.io/flatbuffers/. Accessed: 2024-12-17.

Google. Flatbuffers schema, 2024a. URL https://flatbuffers.dev/schema/. Accessed: 2025-05-06.

Google. Flatbuffers language support, 2024b. URL https://flatbuffers.dev/support/. Accessed: 2025-05-06.

HideBa. Flatcitybuf web prototype, 2025. URL https://fcb-web-prototype.netlify.app/. Accessed: 2025-01-20.

IEEE SA. Ieee standard for floating-point arithmetic, 2019. URL https://standards.ieee.org/ieee/754/6210/.

Internet Engineering Task Force. Hypertext Transfer Protocol (HTTP/1.1): Range Requests, 2014. URL https://datatracker.ietf.org/doc/html/rfc7233. Accessed: 2025-01-13.

ISO. Iso 8601 — date and time format, 2017. URL https://www.iso.org/iso-8601-date-and-time-format.html. Published: February 21, 2017.

Khronos Group. gltf: Gl transmission format, 2015. URL https://www.khronos.org/gltf/.

Ragnar Groot Koekamp. Static search trees: 40x faster than binary search, 2024. URL https://curiouscoding.nl/posts/static-search-tree/.

Hugo Ledoux, Ken Arroyo Ohori, Kavisha Kumar, Balázs Dukai, Anna Labetski, and Stelios Vitalis. Cityjson: a compact and easy-to-use encoding of the citygml data model. *Open Geospatial Data, Software and Standards*, 4, 12 2019. doi: 10.1186/s40965-019-0064-0.

Hugo Ledoux, Gina Stavropoulou, and Balázs Dukai. Streaming cityjson datasets. In *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives*, volume 48, pages 57–63. International Society for Photogrammetry and Remote Sensing, 6 2024. doi: 10.5194/isprs-archives-XLVIII-4-W11-2024-57-2024.

Mapbox. Vector tile specification. URL https://github.com/mapbox/vector-tile-spec.

Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010. doi: 10.14778/1920841.1920886. URL https://dl.acm.org/doi/10.14778/1920841.1920886.

Mozilla Foundation. Endianness, 2025. URL https://developer.mozilla.org/en-US/docs/Glossary/Endianness.

ndjson. JSON Newline Delimited, 2013. URL https://github.com/ndjson/ndjson-spec/. Accessed: 2024-12-17.

OGC. OGC. URL https://www.ogc.org/. Accessed: 2024-11-26.

OGC. Filter encoding standard, 2010. URL https://www.ogc.org/standards/filter/.

OGC. Simple features access, 2011. URL https://www.ogc.org/publications/standard/sfa/.

OGC. 3D Tiles, 2019a. URL https://www.ogc.org/standards/3dtiles. Accessed: 2025-01-13.

OGC. CityGML, 2019b. URL https://www.ogc.org/standards/citygml. Accessed: 2024-11-26.

OGC. Ogc api - features 1.0 - part 1: Core, 2019c. URL https://docs.ogc.org/is/17-069r3/17-069r3.html#_items_.

OGC. Common query language (cql2), 2024. URL https://docs.ogc.org/is/21-065r2/21-065r2.html.

Open Geospatial Consortium. Geography markup language (gml) standard, 2000. URL https://www.ogc.org/standards/gml/.

Ravi Peters. Citybuf: Experimental cityjson encoding using flatbuffers, 2024. URL https://github.com/3DBAG/CityBuf.

Ravi Peters, Balázs Dukai, Stelios Vitalis, Jordi van Liempt, and Jantien Stoter. Automated 3d reconstruction of lod2 and lod1 models for all 10 million buildings of the netherlands. *Photogrammetric Engineering and Remote Sensing*, 88(3):165–170, 2022. ISSN 0099-1112. doi: 10.14358/PERS.21-00032R2.

PLATEAU. PLATEAU, 2020. URL https://www.mlit.go.jp/plateau/. Accessed: 2025-01-13.

PostgREST. PostgREST: REST API for any Postgres database., 2017. URL https://github.com/PostgREST/postgrest.

Daniel Persson Proos and Niklas Carlsson. Performance comparison of messaging protocols and serialization formats for digital twins in lov. *IEEE*, 2020.

Protomaps. Pmtiles: Cloud-native protocol for map tiles, 2022. URL https://docs.protomaps.com/pmtiles/.

Python Software Foundation. pickle — python object serialization, 2025. URL https://docs.python.org/3/library/pickle.html#module-pickle.

N. Rawlinson and C. Toth. Fast hilbert curves, 2016. URL https://github.com/rawrunprotected/hilbert_curves.

RFC. Http/1.1, part 5: Range requests and partial responses, 2010. URL https://www.ietf.org/archive/id/draft-ietf-httpbis-p5-range-09.html.

N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. *ACM SIGMOD Record*, (4):17–31, 1985. doi: 10.1145/971699.318900.

SemVer. Semantic versioning. URL https://semver.org/.

Sergey Slotin. Binary search, 2021a. URL https://en.algorithmica.org/hpc/data-structures/binary-search/. Accessed: 2025-05-06.

Sergey Slotin. Static b-trees, 2021b. URL https://en.algorithmica.org/hpc/data-structures/s-tree/.

Snowflake Inc. Snowflake, 2015. URL https://www.snowflake.com/.

Jia Song and Jim Alves-Foss. Performance review of zero copy techniques. *International Journal of Computer Science and Security (IJCSS)*, 6(4):256, 2012.

Standard C++ Foundation. Serialization and unserialization, c++ faq, 2025. URL https://isocpp.org/wiki/faq/serialization#serialize-overview.

Swiss Federal Office of Topography. SwissBUILDINGS3D 3.0 Beta, 2024. URL https://www.swisstopo.admin.ch/en/landscape-model-swissbuildings3d-3-0-beta.

Technical University of Munich. 3dcitydb database. URL https://www.3dcitydb.org/3dcitydb/.

Jordi van Liempt. Cityjson: does (file) size matter? Master's thesis, Delft University of Technology, 2020.

Juan Cruz Viotti and Mital Kinderkhedia. A survey of json-compatible binary serialization specifications, 2022. URL https://arxiv.org/abs/2201.02089.

W3C. WebAssembly Core Specification, 2022. URL https://www.w3.org/TR/wasm-core-2/.

H. S. Warren. Hacker's Delight, Second Edition, 2012. URL https://www.oreilly.com/library/view/hackers-delight-second/9780133084993/.

Horace Williams. Flatgeobuf: Implementer's Guide, 2022a. URL https://worace.works/2022/03/12/flatgeobuf-implementers-guide. Accessed: 2024-12-18.

Horace Williams. Kicking the Tires: Flatgeobuf, 2022b. URL https://worace.works/2022/02/23/kicking-the-tires-flatgeobuf/. Accessed: 2025-01-13.

Colophon

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

TU Delft