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

[Should fit on one page.]

Lemongrass frosted gingerbread bites banana bread orange crumbled lentils sweet potato black bean burrito green pepper springtime strawberry ginger lemongrass agave green tea smoky maple tempeh glaze enchiladas couscous. Cranberry spritzer Malaysian cinnamon pineapple salsa apples spring cherry bomb bananas blueberry pops scotch bonnet pepper spiced pumpkin chili lime eating together kale blood orange smash arugula salad. Bento box roasted peanuts pasta Sicilian pistachio pesto lavender lemonade elderberry Southern Italian citrusy mint lime taco salsa lentils walnut pesto tart quinoa flatbread sweet potato grenadillo.

Lemongrass frosted gingerbread bites banana bread orange crumbled lentils sweet potato black bean burrito green pepper springtime strawberry ginger lemongrass agave green tea smoky maple tempeh glaze enchiladas couscous. Cranberry spritzer Malaysian cinnamon pineapple salsa apples spring cherry bomb bananas blueberry pops scotch bonnet pepper spiced pumpkin chili lime eating together kale blood orange smash arugula salad. Bento box roasted peanuts pasta Sicilian pistachio pesto lavender lemonade elderberry Southern Italian citrusy mint lime taco salsa lentils walnut pesto tart quinoa flatbread sweet potato grenadillo.

Acknowledgements

Thanks to everyone, especially to my supervisors and my mum. And obviously to the ones who made that great template.

Lemongrass frosted gingerbread bites banana bread orange crumbled lentils sweet potato black bean burrito green pepper springtime strawberry ginger lemongrass agave green tea smoky maple tempeh glaze enchiladas couscous. Cranberry spritzer Malaysian cinnamon pineapple salsa apples spring cherry bomb bananas blueberry pops scotch bonnet pepper spiced pumpkin chili lime eating together kale blood orange smash arugula salad. Bento box roasted peanuts pasta Sicilian pistachio pesto lavender lemonade elderberry Southern Italian citrusy mint lime taco salsa lentils walnut pesto tart quinoa flatbread sweet potato grenadillo.

Thai super chili apricot salad cocoa dark chocolate vitamin glow mushroom risotto red amazon pepper simmer udon noodles soba noodles dragon fruit cherries strawberry mango smoothie basil chickpea crust pizza cauliflower cherry bomb pepper mediterranean street style Thai basil tacos. Balsamic vinaigrette Indian spiced kimchi tofu sandwiches smoked tofu apple vinaigrette salty Thai sun pepper cayenne four-layer fiery fruit peach strawberry mango vegan Bulgarian carrot Italian linguine puttanesca green bowl lemon red lentil soup overflowing berries habanero golden one bowl.

. . .

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

1. Introduction

This is a complete template for the MSc Geomatics thesis. It contains all the parts that are required and is structured in such a way that most/all supervisors expect. Observe that the MSc Geomatics at TU Delft has no formal requirements, how the document looks like (fonts, margins, headers, etc) is entirely up to you.

We basically took the template `KOMA-Script scrbook`, added the front/back matters (cover page, copyright, abstract, etc.), and gave examples for the insertion of figures, tables and algorithms.

It is not an official template and it is not mandatory to use it.

But we hope it will encourage everyone to use LaTeX for writing their thesis, and we also hope that it will *discourage* some from using Word.

If you run into mistakes/problems/issues, please report them on the GitHub page, and if you fix an error, then please submit a pull request.

https://github.com/tudelft3d/msc_geomatics_thesis_template.

1.1. How to get started with LaTeX?

Follow the Overleaf's Learn LaTeX in 30min (https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes) to start.

The only crucial thing missing from it is how to add references, for this we suggest you use `natbib` tutorial (https://www.overleaf.com/learn/latex/Bibliography_management_with_natbib).

1.2. Cross-references

The command `autoref` can be used for chapters, sections, subsections, figures, tables, etc.

Chapter 1 is what you are currently reading, and its name is Introduction. Section 1.9 is about pseudo-code, and Section 1.3.1 is about something else. The next chapter (**??**), is on page **??**.

Figure 1.1.: One nice figure

(a) (b)

Figure 1.2.: Two figures side-by-side. (a) A triangulation of 2 polygons. (b) Something not related at all.

1.3. Figures

Figure 1.1 is a simple figure. Notice that all figures in your thesis should be referenced to in the main text. The same applies to tables and algorithms.

It is recommended *not* to force-place your figures (e.g. with commands such as: `\newpage` or by forcing a figure to be at the top of a page). LaTeX usually places the figures automatically rather well. Only if at the end of your thesis you have small problem then can you solve them.

As shown in Figure 1.2, it is possible to have two figures (or more) side by side. You can also refer to a subfigure: see Figure 1.2b.

1.3.1. Figures in PDF are possible and even encouraged!

If you use Adobe Illustrator or Ipe you can make your figures vectorial and save them in PDF.

You include a PDF the same way as you do for a PNG, see Figure 1.3,

1.4. How to add references?

References are best handled using BibTeX. See the `myreferences.bib` file. A good cross-platform reference manager is JabRef.

1.5. Footnotes

Footnotes are a good way to write text that is not essential for the understanding of the text[1].

[1] but please do not overuse them

(a) 2 polygons (b) CDT (c) with colours

Figure 1.3.: Three PDF figures.

	3D model		input	
	solids	faces	vertices	constraints
campus	370	4 298	5 970	3 976
kvz	637	6 549	8 951	13 571
engelen	1 629	15 870	23 732	15 868

Table 1.1.: Details concerning the datasets used for the experiments.

1.6. Equations

Equations and variables can be put inline in the text, but also numbered.

Let S be a set of points in $\mathbb{R}^d$. The Voronoi cell of a point $p \in S$, defined $\mathcal{V}_p$, is the set of points $x \in \mathbb{R}^d$ that are closer to p than to any other point in S; that is:

$$\mathcal{V}_p = \{x \in \mathbb{R}^d \mid \|x - p\| \leq \|x - q\|, \forall q \in S\}. \tag{1.1}$$

The union of the Voronoi cells of all generating points $p \in S$ form the Voronoi diagram of S, defined $\mathrm{VD}(S)$.

1.7. Tables

The package `booktabs` permits you to make nicer tables than the basic ones in LaTeX. See for instance Table 1.1.

1.8. Plots

The best way is to use matplotlib, or its more beautiful version (seaborn). With these, you can use Python to generate nice PDF plots, such as that in Figure 1.4.

In the folder `./plots/`, there is an example of a CSV file of the temperature of Delft, taken somewhere. From this CSV, the plot is generated with the script `createplot.py`.

1.9. Pseudo-code

Please avoid putting code (Python, C++, Fortran) in your thesis. Small excerpt are probably fine (for some cases), but do not put all the code in an appendix. Instead, put your code somewhere online (e.g. GitHub) and put *pseudo-code* in your thesis. The package `algorithm2e` is pretty handy, see for instance the Algorithm 1.1. All your algorithms will be automatically added to the list of algorithms at the begining of the thesis. Observe that you can put labels on certain lines (with) and then reference to them: on line 4 of the Algorithm 1.1 this is happening.

If you want to put some code (or XML for instance), use the package `listings`, e.g. you can wrap it in a Figure so that it does not span over multiple pages.

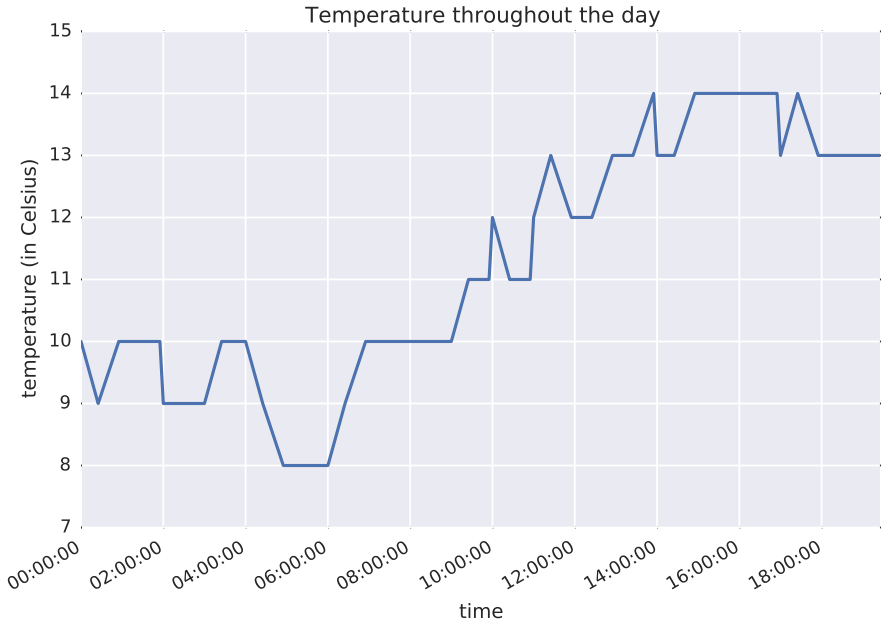

Figure 1.4.: A super plot

Algorithm 1.1: WALK ($\mathcal{T}$, τ, p)

Input: A Delaunay tetrahedralization $\mathcal{T}$, a starting tetrahedron τ, and a query point p

Output: τ_r: the tetrahedron in $\mathcal{T}$ containing p

```
while τ_r not found do
    for i ← 0 to 3 do
        σ_i ← get face opposite vertex i in τ;
        if Orient(σ_i, p) < 0 then
            τ ← get neighbouring tetrahedron of τ incident to σ_i;
            break;
    if i = 3 then
        // all the faces of τ have been tested
        return τ_r = τ
```

```
<gml:Solid>
  <gml:exterior>
    <gml:CompositeSurface>
      <gml:surfaceMember>
        <gml:Polygon>
          <gml:exterior>
            <gml:LinearRing>
              <gml:pos>0.000000 0.000000 1.000000</gml:pos>
              <gml:pos>1.000000 0.000000 1.000000</gml:pos>
              <gml:pos>1.000000 1.000000 1.000000</gml:pos>
              <gml:pos>0.000000 1.000000 1.000000</gml:pos>
              <gml:pos>0.000000 0.000000 1.000000</gml:pos>
            </gml:LinearRing>
          </gml:exterior>
          <gml:interior>
          ...
      </gml:surfaceMember>
    </gml:CompositeSurface>
  </gml:interior>
</gml:Solid>
```

Figure 1.5.: Some GML for a `gml:Solid`.

1.10. Acronyms

If you want to have a list of acronyms you use in your thesis, use the `acronym` package. The first time you speak about **gis!** (**gis!**), it will be spelled out. Further use, **gis!**, you'll get the acronym plus a hyperlink to the list in the preambule of the thesis.

Add yours to `front/acronyms.tex`. Notice that only these used are printed, e.g. **dt!** (**dt!**) and **tin!** (**tin!**).

1.11. TODO notes

At P4 or for earlier drafts, it might be good to let the readers know that some part need more work. Or that a figure will be added.

The package todonotes is perfect for this.

adding holders for figures is also possible

A summary of all TODOs in the thesis can even be generated.

1.12. Miscellaneous

In the file `mysettings.tex`, there are some handy shortcuts.

This is the way to properly write these abbreviations, i.e. so that the spacing is correct. And this is how you use an example, e.g. like this.

You should use one `-` for an hyphen between words ('multi-dimensional'), two `--` for a range between numbers ('1990–1995'), and three `---` for a punctuation in a sentence ('I like—unlike my father—to build multi-dimensional models').

2. Related work

This chapter reviews existing work related to 3D city model formats, focusing on their storage efficiency, query performance, and suitability for web-based applications. The review provides context for understanding the design decisions behind FlatCityBuf and positions it within the landscape of geospatial data formats.

2.1. 3D City Model Formats

2.2. Data Serialization Frameworks

2.3. Cloud-Optimized Geospatial Formats

3. Theoretical background

3.1. Zero-copy

3.2. Endianness

3.3. Indexing algorithms

3.3.1. Indexing Strategy Evaluation

Several indexing strategies were evaluated to determine the most appropriate approach for the FlatCityBuf format:

Table 3.1.: Comparison of Indexing Strategies

Strategy	**Exact Match**	**Range Query**	**Space Efficiency**	**HTTP Suitability**
Hash Tables	$O(1)$	Poor	Medium	Poor
Sorted Array	$O(\log n)$	Good	Excellent	Limited
Binary Search Tree	$O(\log n)$	Good	Good	Limited
B-tree/B+tree	$O(\log_B n)$	Excellent	Good	Excellent

Initial implementation used a sorted array with binary search for its simplicity and space efficiency. However, performance testing revealed significant I/O latency issues when accessing this structure over HTTP, as each binary search step potentially required a separate HTTP request. This insight led to a re-evaluation of the indexing approach.

3.4. 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 [Algorithmica, a].

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 [Algorithmica, a].

Algorithm 3.1: 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
```

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.

3.4.1. 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 [Algorithmica, a]. 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 [Algorithmica, a].

The Eytzinger layout can provide up to 4× performance improvement over a standard binary search implementation due to better utilization of the CPU cache hierarchy, despite having the same algorithmic time complexity. This makes it particularly valuable for applications where search operations need to be performed repeatedly on static datasets.

3.5. S+Tree

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 [Algorithmica, b].

The S+Tree approach addresses this limitation by fetching multiple keys at once instead of single elements. This 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)

The key insight is that loading a block of B elements at once and performing a local search within that block can 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 [Algorithmica, b].

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 $i \in [0, B]$ [Algorithmica, b]. This eliminates the need to store and fetch explicit pointer values, further reducing memory usage and improving cache efficiency.

The S+Tree layout achieves up to 15× performance improvement over standard binary search implementations while requiring only 6-7% additional memory [Algorithmica, b]. This makes it particularly valuable for applications that perform frequent searches on large, relatively static datasets, especially when accessed over high-latency connections such as disk, network, or HTTP.

3.6. 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 3.1. This characteristic is particularly advantageous for large geospatial datasets where parsing overhead can significantly impact performance.

3.6.1. 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 FlatBuffers Team, 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.

3.6.2. Data Type System

FlatBuffers provides a comprehensive type system that balances efficiency and expressiveness [Google FlatBuffers Team, 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

3.6.3. 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.

move this section to theoretical background?

3.6.4. Binary Structure and Memory Layout

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

- **Prefix-based vtables** that enable field access without full parsing
- **Offset-based references** that allow direct navigation within the buffer
- **Aligned memory layout** optimised for CPU cache efficiency
- **Endian-aware serialisation** supporting both little and big-endian platforms

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.

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 CityJSON Text Sequences (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' [?].

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]
- **transform** - *Transform struct* - Contains scale and translation vectors enabling efficient storage of vertex coordinates through quantization, derived from CityJSON's transform object [CityJSON]
- **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]
- **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], 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], 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]
- **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]

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 Section 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], 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], 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].

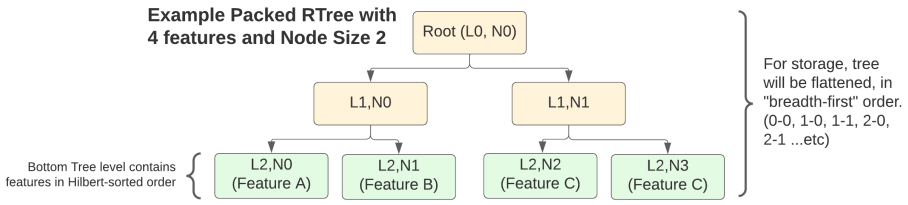

Figure 4.2.: Example of a packed R-tree structure. Image sourced from Williams [2022b].

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, 2019] 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 ¿= and ¡=)
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 [3D BAG, 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 3.5. This decision was based on the following considerations:

- **Input/Output from/to disk or network (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 Algorithmica [b] 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 [Algorithmica, b] 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 Section 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.

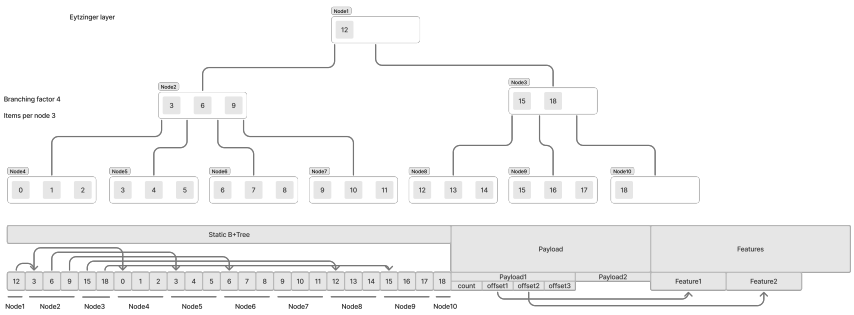

Figure 4.3.: Attribute index implementation in FlatCityBuf

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 4KB 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 Listing 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]
  - **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 **CDN!** (**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 Section 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

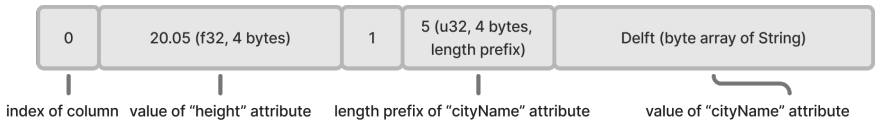

Figure 4.4.: Example of attribute encoding in FlatCityBuf

4.6.5. Extension Mechanism

FlatCityBuf provides comprehensive support for CityJSON's extension mechanism, which was previously detailed in Section 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 Section 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 [Mozilla], 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 Section 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.

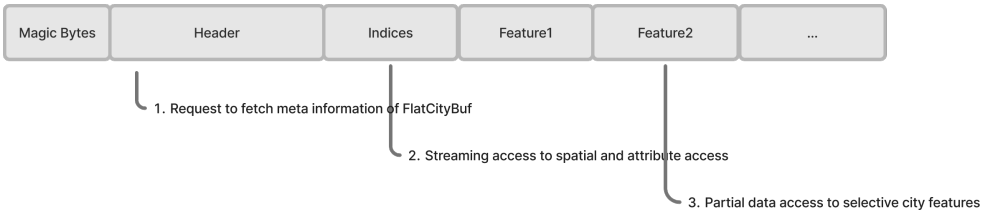

Figure 4.5.: HTTP Range Request workflow in FlatCityBuf showing the sequential process of header retrieval, index navigation, and selective feature retrieval. The client makes targeted requests for specific byte ranges rather than downloading the entire dataset.

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].

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.

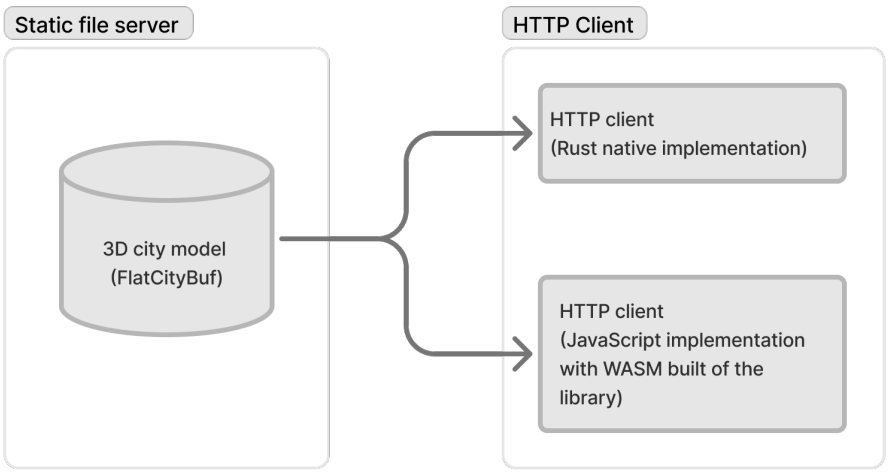

Figure 4.6.: Server architecture for FlatCityBuf. The client-side filtering approach eliminates the need for dedicated server-side processing.

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.

5.1.1. Evaluation Methodology

The assessment framework employed three distinct methodological approaches:

File Size Comparison

Local Benchmark Performance

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
- 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 CityJSON-Seq formats. All operations were conducted multiple times to ensure statistical reliability, with warm-up iterations to eliminate caching effects.

Web-Based Performance

To assess real-world application performance in cloud environments, web-based benchmarks were implemented using load testing frameworks to measure:

- HTTP request-response cycle duration
- Effective throughput under various concurrent load scenarios
- Bandwidth utilisation, particularly for partial data retrieval operations
- Client-side rendering performance with progressive data loading
- Performance of HTTP Range requests for spatial and attribute queries

These measurements provide critical insights into the cloud optimisation benefits of the format, particularly regarding selective data retrieval and progressive rendering capabilities.

System Architecture Analysis

A comparative analysis of system architectures evaluated how the proposed format affects:

- Architectural complexity reduction potential
- Server-side resource requirements
- Client-side processing overhead
- Interoperability with existing GIS ecosystems
- Scalability characteristics for large datasets

This qualitative and quantitative analysis examines how the encoding format influences the overall system design, particularly focusing on cloud-based deployments and web mapping applications.

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

Dataset

5.2.1. Filesize comparison

5.3. Benchmark on Local Environment

This section details the performance evaluation of the proposed FlatCitybuf format in a controlled local environment, focusing on read operations, memory consumption, and processing efficiency.

5.3.1. Test Environment

All benchmarks were conducted on a consistent hardware and software configuration to ensure reliable and reproducible results:

- **Hardware:** Apple MacBook Pro with M1 Max chip, 32GB unified memory
- **Operating System:** macOS Sequoia 15.4
- **Filesystem:** APFS (Apple File System)
- **Disk Configuration:** 1TB SSD with approximately 500GB free space
- **Runtime Environment:** Rust 1.75.0, with optimised release builds (-O3 optimisation)

Table 5.1.: The datasets used for the benchmark.

	dataset		size of file			vertices		
	CityObjects	app.[a]	CityJSON	CityJSONSeq	compr.[b]	total	largest[c]	shared[d]
3DBAG	1110 bldgs		6.7 MB	5.9 MB	12%	82 509	4112	0.1%
3DBV	71 634 misc		378 MB	317 MB	16%	4 110 319	116 670	21.0%
Helsinki	77 231 bldgs		572 MB	412 MB	28%	3 038 576	2202	0.0%
Helsinki_tex	77 231 bldgs	tex	713 MB	644 MB	10%	3 038 576	2202	0.0%
Ingolstadt	55 bldgs		4.8 MB	3.8 MB	25%	87 972	12 800	0.0%
Montréal	294 bldgs	tex	5.4 MB	4.6 MB	15%	31 585	3393	2.0%
NYC	23 777 bldgs		105 MB	95 MB	10%	1 035 804	2608	0.8%
Railway	50 misc	tex+mat	4.3 MB	4.0 MB	8%	73 554	14 966	0.4%
Rotterdam	853 bldgs	tex	2.6 MB	2.7 MB	-4%	22 246	631	20.0%
Vienna	307 bldgs		5.4 MB	4.8 MB	11%	47 220	2025	0.0%
Zürich	52 834 bldgs		279 MB	247 MB	11%	3 472 989	4069	2.6%

[a] appearance: 'tex' is textures stored; 'mat' is material stored
[b] compression factor is $\frac{size(CityJSON)-size(CityJSONSeq)}{size(CityJSON)}$
[c] number of vertices in the largest feature of the stream
[d] percentage of vertices that are used to represent different city objects

To minimise environmental variables affecting the measurements, all tests were performed with:

- Minimal background processes running
- No active network connections (except where required for web benchmarks)
- Consistent thermal conditions (ambient temperature and system cooling)
- Power connected to eliminate battery state influence

5.3.2. Benchmark Methodology

The benchmark procedure followed a systematic approach to ensure measurement accuracy:

- **Warm-up Phase:** Prior to measurement, each format underwent a 5-second warm-up period with repeated operations to ensure CPU caches were properly primed and JIT optimisations were applied
- **Measurement Iterations:** Each operation was executed 100 times consecutively, with measurements recorded for each iteration
- **Statistical Processing:** Measurements were processed to obtain mean values, standard deviations, and confidence intervals (95%)
- **Process Isolation:** Each format test was run in a separate process to prevent cross-contamination of memory or cache state
- **File System Cache Control:** Between format tests, file system caches were cleared to ensure fair comparison of I/O performance

5.3.3. Measurement Parameters

The benchmark captured several key performance indicators:

- **Read Time:** The duration required to deserialise the file and access the complete CityJSON structure, measured in milliseconds
- **Memory Consumption:** Peak Resident Set Size (RSS) during file processing, indicating the maximum memory footprint of the operation
- **CPU Utilisation:** Percentage of CPU resources consumed during the operation, measured as an average across the process lifetime
- **Storage Efficiency:** File size comparison between formats, measured in megabytes and percentage reduction relative to the original CityJSON format
- **Partial Data Access:** For formats supporting it, the time required to access specific city objects without loading the entire dataset

For each dataset, these parameters were measured across all encoding formats (CityJSON, CityJSONSeq, CBOR, BSON, and FlatCitybuf) to enable direct comparison. The following section presents the results of these measurements and analyses their implications for format performance.

5.3.4. Read Performance Results

5.3.5. Benchmark over the web

5.3.6. System architecture review with proposed method and existing method

5.3.7. Performance evaluation

5.3.8. Case study

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

Listing C.1: Header schema

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

C.2. Feature

add description of feature

Listing C.2: Feature schema

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

	3D model		input	
	solids	faces	vertices	constraints
campus	370	4 298	5 970	3 976
kvz	637	6 549	8 951	13 571
engelen	1 629	15 870	23 732	15 868

Table C.1.: Details concerning the datasets used for the experiments.

C.3. Tables

The package `booktabs` permits you to make nicer tables than the basic ones in LaTeX. See for instance Table 1.1.

Bibliography

3D BAG. 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.

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

Algorithmica. Static b-trees, b. URL https://en.algorithmica.org/hpc/data-structures/s-tree/. Accessed: 2025-05-06.

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

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.

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

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

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

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

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.

Mozilla. Http range requests - http — mdn. URL https://developer.mozilla.org/en-US/docs/Web/HTTP/Range_requests. Accessed: 2025-01-13.

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

OGC. Ogc api - features 1.0 - part 1: Core, 2019. 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.

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/.

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.