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 **??**.

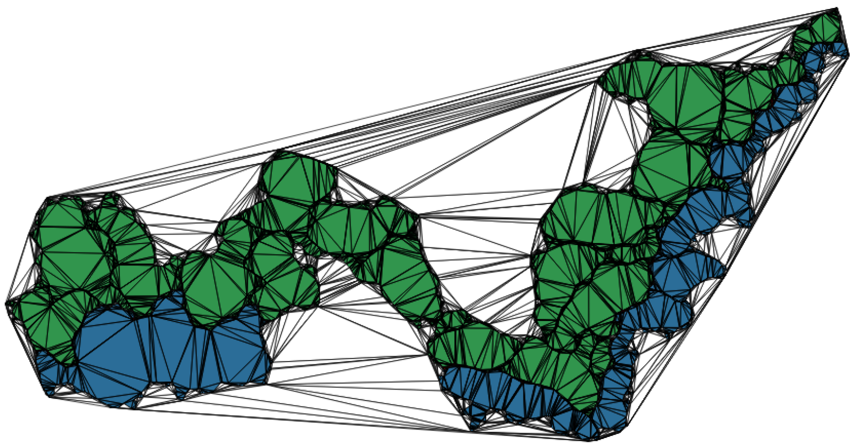


Figure 1.1.: One nice figure

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.

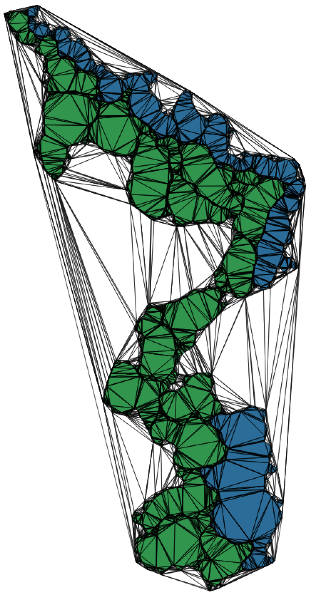
1.3.1. Figures in PDF are possible and even encouraged!

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

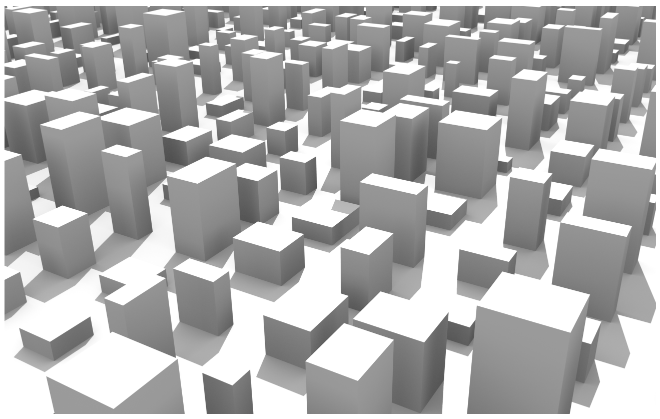
You include a PDF the same way as you do for a PNG, see Figure 1.3,

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.

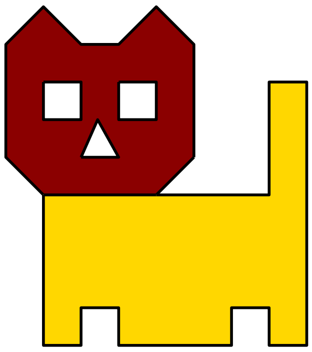


(a)

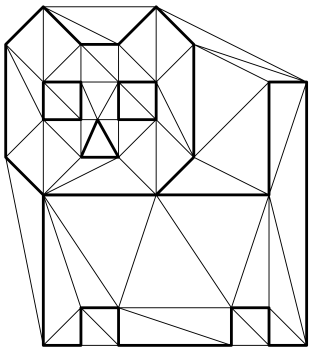


(b)

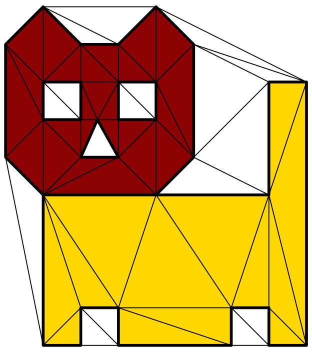
Figure 1.2.: Two figures side-by-side. (a) A triangulation of 2 polygons. (b) Something not related at all.



(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.

1.5. Footnotes

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

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 VD(S).

1.7. Tables

The package `booktabs` permits you to make nicer tables than the basic ones in LaTeX. See for instance Table C.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]but please do not overuse them

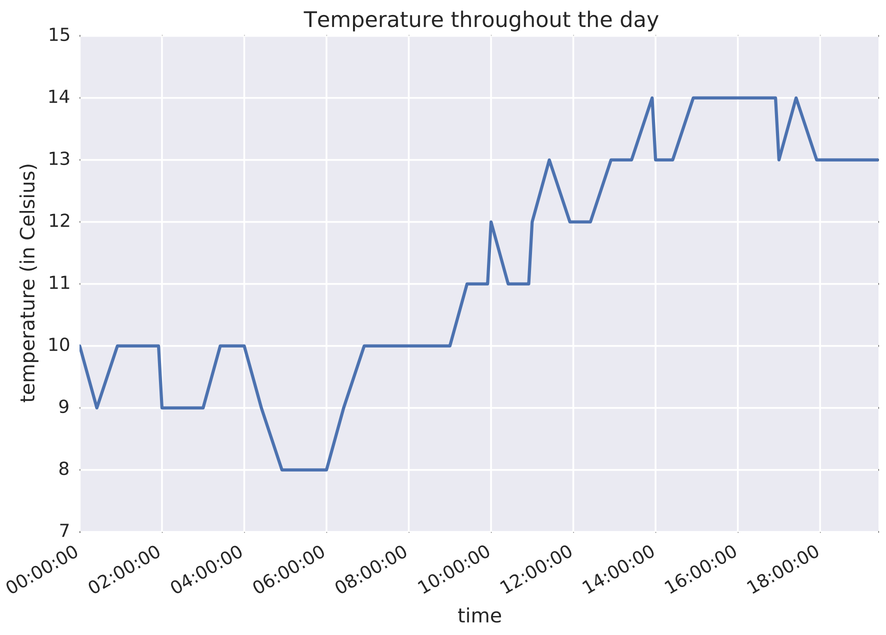


Figure 1.4.: A super plot

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.

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!**).

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

```
1 while $\tau_r$ not found do
2     for $i \leftarrow 0$ to $3$ do
3         $\sigma_i \leftarrow$ get face opposite vertex $i$ in $\tau$;
4         if $Orient(\sigma_i, p) < 0$ then
5             $\tau \leftarrow$ get neighbouring tetrahedron of $\tau$ incident to $\sigma_i$;
6             break;
7     if $i = 3$ then
          // all the faces of $\tau$ have been tested
8         return $\tau_r = \tau$
```

```
<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`.

1.11. 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.

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 [?].

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

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

height of the tree, with each read potentially requiring access to a different cache line or disk block [?].

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 [?]. 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 [?].

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. Static B+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 [?].

The Static B+Tree (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 [**?**].

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 Static B+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 $i \in [0, B]$ [**?**]. 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 [**?**]. 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 **?**, 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 [**?**], 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 [**?**]. 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 [**?**]:

- **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 **cjseq!** (**cjseq!**)) 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 (Packed Hilbert R-tree) and attribute-based (Static B+tree)—that accelerate query performance
3. Integration of cloud-native data access patterns through HTTP Range Requests, enabling partial data retrieval

4.1.2. File Structure Overview

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

- **Magic bytes**: Eight-byte identifier ('FCB
  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 Static B+tree for accelerated attribute-based filtering
- **Features section**: Stores city objects encoded as FlatBuffers tables

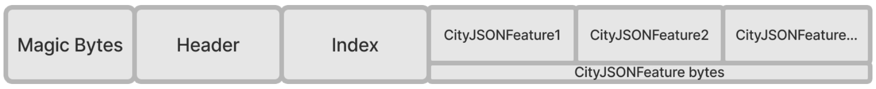


Figure 4.1.: Physical layout of the FlatCityBuf file format, showing section boundaries and alignment considerations for optimised range requests

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.

add reference why these two

4.1.3. Note on Binary Encoding

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 four bytes contain the ASCII sequence 'FCB 0' (0x46 0x43 0x42 0x00) serving as an immediate identifier
- The remaining four bytes follow a pattern similar to FlatGeoBuf [?], with byte 5-6 containing major version (currently 01), followed by ASCII 'FB' (0x46 0x42), and byte 8 containing patch version (currently 00)

This signature design enables applications to validate file type and version compatibility without parsing the entire header content.

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 [?]—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 [?]
- **transform** - *Transform struct* - Contains scale and translation vectors enabling efficient storage of vertex coordinates through quantization, derived from CityJSON's transform object [?]
- **reference_system** - *ReferenceSystem table* - Coordinate reference system information including:
  - *authority* - Authority name, typically "EPSG"
  - *code* - Numeric identifier of the CRS
  - *version* - Version of the CRS definition
- **geographical_extent** - *GeographicalExtent struct* - 3D bounding box containing min/max coordinates for the dataset [?]

- **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 information for the dataset provider [**?**]:
  - *poc_contact_name* - Name of the point of contact
  - *poc_contact_type* - Type of contact (e.g., "individual", "organization")
  - *poc_role* - Role of the contact (e.g., "author", "custodian")
  - *poc_email* - Email address of the contact
  - *poc_website* - Website for the contact
  - *poc_phone* - Phone number of the contact
  - *poc_address_** - 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 [**?**], containing:
  - *materials* - Array of Material tables with the following properties:
    - *name* - Required string identifier for the material
    - *ambient_intensity* - Double precision value from 0.0 to 1.0
    - *diffuse_color* - Array of double values (RGB) from 0.0 to 1.0
    - *emissive_color* - Array of double values (RGB) from 0.0 to 1.0
    - *specular_color* - Array of double values (RGB) from 0.0 to 1.0
    - *shininess* - Double precision value from 0.0 to 1.0
    - *transparency* - Double precision value from 0.0 to 1.0
    - *is_smooth* - Boolean flag for smooth shading
  - *textures* - Array of Texture tables with the following properties:
    - *type* - TextureFormat enum (PNG, JPG)
    - *image* - Required string containing image file name or URL
    - *wrap_mode* - WrapMode enum (None, Wrap, Mirror, Clamp, Border)
    - *texture_type* - TextureType enum (Unknown, Specific, Typical)
    - *border_color* - Array of double values (RGBA) from 0.0 to 1.0
  - *vertices_texture* - Array of Vec2 structs containing UV coordinates (u,v), each coordinate value must be between 0.0 and 1.0 for proper texture mapping

- *default_theme_material* and *default_theme_texture* - Strings identifying default themes for rendering when multiple themes are defined

The appearance model provides a standardized way to define visual properties of city objects, supporting multiple rendering engines and visualization tools. Materials define surface properties while textures allow mapping of image data onto geometry surfaces. The separation of appearance definitions from geometry enables efficient storage by allowing multiple surfaces to reference the same materials or textures.

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 [**?**]
- **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 [**?**]

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.

4.3.4. Extension Support

Fields enabling schema extensibility:

- **extensions** - *Array of Extension tables* - Definitions for CityJSON extensions [**?**], each containing:
  - *name* - Extension identifier (e.g., "+Noise")
  - *url* - Reference to the extension schema
  - *version* - Extension version identifier
  - *extra_attributes*, *extra_city_objects*, *extra_root_properties*, *extra_semantic_surfaces* - Stringified JSON schemas for extension components

Unlike standard 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* - Numeric identifier of the column
  - *name* - Name of the attribute (e.g., ”cityname”, ”owner”, etc.)
  - *type* - Data type enumeration (e.g., ”Int”, ”String”, etc.)
  - *nullable*, *unique*, *precision* - Optional metadata for validating and interpreting values
- **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* - Reference to the column being indexed
  - *length* - Size of the index in bytes
  - *branching_factor* - Branching factor of the index, number of items in each node is equal to branching factor $- 1$
  - *num_unique_items* - Count of unique values for this attribute

add more details about here

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 [**?**] 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 [**?**]. Both the conceptual design and implementation details were sourced from FlatGeoBuf's approach as documented in **?** and . This includes the Hilbert curve ordering strategy, node structure, tree construction methodology, and query algorithms.

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. Spatial Indexing Requirements

The spatial indexing mechanism in FlatCityBuf was designed to address several key requirements:

make more concise

- **Selective Data Access**: Allow retrieval of city features within a specified bounding box without downloading the entire dataset
- **HTTP Range Request Compatibility**: Support efficient operation over HTTP using range requests
- **Compact Representation**: Minimise storage overhead while maintaining query efficiency
- **Cloud Optimisation**: Reduce network traffic and improve query responsiveness in cloud environments

These requirements led to the adoption of a packed R-tree structure, specifically optimised for static datasets accessed through block-oriented I/O patterns.

4.4.3. Packed Hilbert R-tree Implementation

The spatial index in FlatCityBuf implements a packed R-tree structure with Hilbert curve ordering, following the implementation pattern established by FlatGeoBuf [**?**]. This approach offers several advantages over traditional R-tree variants:

- **Static Structure**: The tree is built once and remains immutable, eliminating the need for complex insertion and rebalancing algorithms
- **Perfect Packing**: Nodes are filled to capacity (except possibly the rightmost nodes at each level), maximising space efficiency
- **Hilbert Ordering**: Features are sorted using a Hilbert space-filling curve to improve spatial locality [?]
- **Block-Aligned Design**: Node sizes align with common block sizes for efficient I/O operations

The index is organised as a sequence of nodes, each containing:

- **Minimum X,Y**: The minimum coordinates of the node's bounding box (8 bytes each)
- **Maximum X,Y**: The maximum coordinates of the node's bounding box (8 bytes each)
- **Offset**: The byte offset to either a child node or a feature in the features section (8 bytes)

This results in a fixed node size, allowing for predictable memory layout and efficient search within nodes.

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 and similar standards primarily support 2D spatial querying, making 2D indexing more broadly compatible

add citation

- **Implementation Efficiency**: 2D indexing is computationally simpler and more storage-efficient than 3D alternatives

For implementation, the centroid of each CityFeature's vertices is calculated using only X and Y coordinates, and this 2D centroid is used for Hilbert encoding and spatial indexing. This approach provides a reasonable balance between query performance and implementation complexity for most urban modelling use cases.

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 was designed to support specific query patterns commonly used in geospatial applications. Based on an analysis of typical use cases, the following query types were identified as essential:

- **Exact Match**: Queries that seek records matching a specific attribute value (e.g., `building_type = "residential"`)
- **Range Queries**: Queries that select records with attribute values falling within a specified range (e.g., `height >= 10 AND height <= 20`)
- **Compound Conditions**: Multiple conditions combined with logical operators (e.g., `building_type = "residential" AND height > 15`)

While the SQL standard [?] defines a comprehensive set of predicates and operators for database querying, implementing the full spectrum of these capabilities is beyond the scope of this research. FlatCityBuf deliberately focuses on a subset of operators that provide the greatest utility for typical 3D city model queries while maintaining efficient implementation over HTTP.

Notably absent are pattern-matching operations such as the SQL `LIKE` operator (e.g., `city LIKE "Delf%"`), which would require specialized text indexing structures. Similarly, functions like `CONTAINS`, `BETWEEN`, aggregate functions (`COUNT`, `SUM`, etc.), and advanced text search capabilities were deemed lower priority compared to the core comparison operators. These advanced query types can be implemented at the application layer after retrieving the relevant data.

The system prioritises these core query types while ensuring compatibility with remote access patterns through HTTP Range Requests. This focused approach aligns with FlatCityBuf's primary goal as an efficient storage and retrieval format rather than a comprehensive query processing system.

4.5.2. Static B+tree Design

After evaluating alternatives, a Static B+tree (S+tree) approach was adopted. This decision was based on the following considerations:

- **Block-Oriented Access**: B+trees organise data into fixed-size nodes (typically matching file system block sizes), minimising the number of I/O operations required.
- **HTTP Range Request Efficiency**: By aligning nodes with typical block sizes (4KB), the structure optimises for HTTP Range Requests, reducing the number of network roundtrips.

- **Balanced Performance**: The structure offers $O(\log_B n)$ search complexity where B is the branching factor, significantly reducing the number of node accesses compared to a binary search tree.
- **Range Query Support**: The leaf-level organisation of B+trees facilitates efficient range queries through sequential access.

The implementation uses a static variant of the B+tree, built once and used for read-only operations. This design choice aligns with the immutable nature of the FlatCityBuf format and enables additional optimisations not possible with dynamic tree structures.

4.5.3. Implicit Layout and Memory Efficiency

The Static B+tree implementation uses an Eytzinger layout for node placement, offering several benefits:

- **Implicit Structure**: No explicit pointers are stored between nodes, reducing storage overhead.
- **Cache Locality**: Nodes at the same level are stored contiguously, improving cache efficiency.
- **Minimal Metadata**: The tree structure is determined mathematically based on the branching factor and number of entries.

This approach produces a compact, cache-friendly structure that requires minimal metadata while maintaining excellent query performance. Internally, the layout is constructed using the following algorithm:

1. Calculate level bounds based on the branching factor and total number of entries
2. Copy leaf nodes from the sorted array of entries
3. Build internal nodes bottom-up, selecting appropriate separator keys
4. Store nodes contiguously in level order (root first, then each subsequent level)

4.5.4. Type-Specific Serialisation

The attribute index supports various data types common in 3D city models, including:

- **Numeric Types**: Integers (i8, i16, i32, i64, u8, u16, u32, u64) and floating-point values (f32, f64)
- **Temporal Types**: Dates and timestamps with timezone information
- **String Types**: Fixed-width strings with prefix encoding
- **Boolean Values**: Represented as single bytes

Each type implements a specialised serialisation strategy that preserves ordering semantics while optimising storage efficiency. For floating-point values, the implementation uses 'OrderedFloat' to handle NaN values correctly. Strings utilise a fixed-width prefix encoding that balances storage requirements with efficient comparison operations.

4.5.5. Duplicate Key Handling

A significant optimisation in the attribute index is the handling of duplicate keys:

- **Primary Index Structure**: Contains only unique keys, with pointers to either direct feature offsets or to a payload section
- **Payload Section**: Stores lists of offsets for duplicate key values
- **Tag Bit**: The most significant bit of the offset value indicates whether it points directly to a feature or to the payload section

This approach maintains the efficiency of the tree structure while properly handling attributes with many duplicate values. For example, building type attributes often have many identical values (e.g., hundreds of "residential" buildings), which are all efficiently indexed through a single payload reference.

4.5.6. HTTP Optimisation Techniques

The attribute index implements several techniques specifically designed to optimise performance over HTTP:

- **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.

Internal testing demonstrated that these optimisations reduce HTTP requests by up to 90% for typical queries compared to naïve implementations, significantly improving overall query performance for web-based applications.

4.5.7. Multi-Index Query Execution

The attribute indexing system supports complex queries across multiple attributes through a coordinated query execution strategy:

1. Each condition in the query is evaluated against the appropriate attribute index
2. Results from individual conditions are represented as sets of feature offsets
3. For conjunctive queries (AND logic), set intersection determines the final result
4. For disjunctive queries (OR logic), set union combines the results

The implementation provides specialised index structures for different access patterns:

- **MemoryIndex**: For in-memory operations with the entire index loaded

- **StreamIndex**: For file-based access using standard Read+Seek operations
- **HttpIndex**: For remote access using HTTP Range Requests

Each implementation provides semantically equivalent operations while optimising for its specific access pattern, ensuring consistent results regardless of the access method.

4.5.8. Performance Characteristics

The Static B+tree implementation demonstrates several key performance characteristics:

- **Query Complexity**: $O(\log_B n)$ node accesses for both exact match and range queries, where B is the branching factor (typically 16)
- **Storage Efficiency**: Approximately 8 bytes per indexed entry plus payload overhead for duplicate keys
- **Construction Time**: $O(n \log n)$ for sorting plus $O(n)$ for tree construction
- **HTTP Efficiency**: Typically 3-5 HTTP requests per query for moderate-sized datasets (millions of entries)

Benchmarks conducted with various datasets demonstrated query performance 10-20 times faster than traditional approaches when accessed over HTTP, particularly for datasets with many duplicate attribute values.

The combination of block-oriented design, implicit layout, and HTTP optimisations results in an attribute indexing system that efficiently supports the required query patterns while minimising both storage requirements and network overhead.

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. This section details the encoding strategies for city objects, geometry, attributes, appearances, and extensions.

4.6.1. CityFeature and CityObject Structure

FlatCityBuf organises data following CityJSON's hierarchical model, but with optimisations for binary serialisation:

- **CityFeature**: The top-level container encoded as a FlatBuffers table, repre´senting a collection of city objects that share common vertices and other properties.
- **CityObject**: Individual 3D features (e.g., buildings, bridges) that compose a city model, with their own geometry, attributes, and semantic information.

The schema establishes a one-to-many relationship between CityFeatures and CityObjects, allowing efficient vertex sharing while maintaining semantic distinctions between different city objects. This structure is implemented through the following FlatBuffers schema:

```
table CityFeature {
  id: string (key, required);
  objects: [CityObject];
  vertices: [Vertex];
  appearance: Appearance;
}

table CityObject {
  type: CityObjectType;
  id: string (key, required);
  geographical_extent: GeographicalExtent;
  geometry: [Geometry];
  attributes: [ubyte];
  columns: [Column];
  children: [string];
  children_roles: [string];
  parents: [string];
}
```

This structure enables efficient access to individual city objects without loading the entire dataset, while maintaining the relationships and hierarchies defined in the CityJSON model.

4.6.2. Geometry Encoding

Geometry encoding is one of the most critical aspects of FlatCityBuf, as it represents the 3D shapes that constitute city models. The approach follows CityJSON's boundary representation (B-rep) model with optimisations for binary storage:

Boundary Representation

FlatCityBuf encodes geometry boundaries using a hierarchical indexing approach that aligns with CityJSON's dimensional hierarchy:

- **Indices/Boundaries**: A flattened array of vertex indices that reference vertices in the containing CityFeature's vertex array.
- **Strings**: Arrays defining the number of vertices in each boundary ring (typically 3 or 4 vertices per string).
- **Surfaces**: Arrays indicating the number of rings (strings) that compose each surface.
- **Shells**: Arrays specifying the number of surfaces in each shell.
- **Solids**: Arrays denoting the number of shells in each solid.

This hierarchical structure efficiently represents multi-dimensional geometry with minimal redundancy. For example, a simple triangle surface would be encoded as:

```
boundaries: [0, 1, 2]  // Vertex indices
strings: [3]           // 3 vertices in the string
surfaces: [1]          // 1 string in the surface
```

While a cube (a solid with 6 quadrilateral surfaces) would be represented as:

```
boundaries: [0, 1, 2, 3, 0, 3, 7, 4, ...]  // 24 vertex indices
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 Classification

Semantic information is essential for distinguishing different functional parts of city objects (e.g., walls, roofs, doors). FlatCityBuf encodes these semantics through:

- **SemanticObjects**: Tables containing type classification and attributes for each semantic unit.
- **Semantic Arrays**: Parallel arrays to the geometry hierarchy that reference semantic objects.

The schema defines semantic objects as:

```
table SemanticObject {
  type: SemanticSurfaceType;
  extension_type: string;
  attributes: [ubyte];
  columns: [Column];
  parent: uint;
  children: [uint];
}
```

This approach preserves the semantically rich information of CityJSON while optimising for binary storage and retrieval efficiency. Surface types are encoded as enumerated values (e.g., `WallSurface`, `RoofSurface`) for compact representation.

Geometry Templates

For city models containing repeated geometries (e.g., identical building types), FlatCityBuf supports CityJSON's template mechanism with optimisations for binary representation:

- **Template Definition**: Geometry templates are stored once in the header section with double-precision coordinates to maintain precision:

  ```
  table Header {
    // ...
    templates: [Geometry];
    templates_vertices: [DoubleVertex];
    // ...
  }
  ```

- **Template Instance**: CityObjects reference templates using a compact representation:

  ```
  table GeometryInstance {
    transformation: TransformationMatrix;
    template: uint;
    boundaries: [uint];
  }
  ```

Each instance consists of:

- A reference to the template geometry
- A single vertex index serving as the reference point
- A 4×4 transformation matrix for positioning, rotating, and scaling

This approach significantly reduces file size for datasets with repeated structures, commonly found in planned urban developments with standardised building designs.

Appearances: Materials and Textures

FlatCityBuf supports CityJSON's appearance model, including materials and textures, through a reference-based approach:

- **Materials**: Defined as FlatBuffers tables with properties for colour, transparency, and shininess.
- **Textures**: Represented as metadata with URI references to external texture files.

For textures, FlatCityBuf optimises for selective loading by storing only texture metadata and references rather than embedding texture data directly:

```
table Texture {
  type: TextureType;
  image: string;
  wrap_mode: WrapMode;
  texture_type: TextureSemanticType;
  border_color: Color;
}
```

This design choice prioritises:

- Efficient storage and retrieval of geometric data
- Support for asynchronous texture loading
- Compatibility with web-based caching mechanisms
- Selective loading based on application requirements

Material and texture assignments are managed through mapping tables that associate specific surfaces with appearance definitions.

4.6.3. Attribute Encoding

Attributes store non-geometric properties of city objects (e.g., year of construction, owner, function). FlatCityBuf implements an efficient binary representation:

- **Schema Declaration**: The header section contains column definitions describing the name, type, and other metadata for each attribute:

  ```
  table Column {
    index: ushort;
    name: string (required);
    type: ColumnType;
  }
  ```

- **Binary Storage**: Attributes are serialised as a binary blob using type-specific encoding:
  - Numeric types (integers, floats): Native binary representation
  - Strings: Length-prefixed UTF-8 encoding

  - Booleans: Single-byte representation
  - Dates/Times: Standardised binary format

- **Variable-Length Fields**: For variable-length types like strings, a length prefix is stored before the data to enable efficient traversal.

This approach balances flexibility with efficiency, allowing FlatCityBuf to represent the diverse attribute sets found in city models while maintaining high performance. By moving type information to the schema level rather than self-describing each value, this encoding achieves significant storage reductions compared to JSON-based formats.

4.6.4. Extension Mechanism

FlatCityBuf provides comprehensive support for CityJSON's extension mechanism, allowing customisation beyond the core schema. Unlike CityJSON's approach of referencing external schema files, FlatCityBuf implements a self-contained extension model:

- **Extension Definition**: Extensions are defined in the header as FlatBuffers tables:

```
table Extension {
  name: string;
  description: string;
  url: string;
  version: string;
  version_cityjson: string;
  extra_attributes: string;
  extra_city_objects: string;
  extra_root_properties: string;
  extra_semantic_surfaces: string;
}
```

- **Extended CityObjects**: Custom types are encoded using a special enum value with an explicit extension type string:

```
enum CityObjectType:ubyte {
  // ... standard types ...
  ExtensionObject
}

table CityObject {
  type: CityObjectType;
  extension_type: string; // e.g. "+NoiseCityFurnitureSegment"
  // ...
}
```

- **Extended Semantics**: Similar to CityObjects, custom semantic surface types use a special enum value with an extension type string.

- **Attribute Integration**: Extension attributes are encoded in the same binary attribute array as core attributes, simplifying implementation.

This approach offers several advantages:

- Self-contained files that don't require external schema references
- Efficient representation of extended types using a hybrid enum-string approach
- Consistent attribute handling for both core and extension properties
- Preservation of extension semantics despite binary encoding

The extension mechanism enables FlatCityBuf to support domain-specific additions (e.g., noise simulation data, energy modelling parameters) without compromising the efficiency of the core format.

4.6.5. Performance Considerations

The feature encoding strategy in FlatCityBuf was designed with several performance considerations:

- **Memory Locality**: Related data (e.g., vertices, boundaries) are stored contiguously to improve cache efficiency.
- **Size Reduction**: Binary encoding reduces storage requirements by 50-70% compared to JSON representations.
- **Random Access**: FlatBuffers' table structure enables direct access to specific components without parsing entire objects.
- **Zero-Copy Access**: The binary format allows direct memory mapping without intermediate parsing steps.
- **Minimal Redundancy**: Shared vertices and template mechanisms reduce duplication.

These optimisations contribute to the overall performance of FlatCityBuf, particularly for large datasets accessed over network connections where minimising data transfer and parsing overhead is critical.

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, enabling efficient web-based access to 3D city models. 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, allow clients to request specific byte ranges from server resources instead of entire files. FlatCityBuf's structure is specifically designed to leverage this capability, with particular attention to:

- **Aligned Section Boundaries**: File sections are aligned to facilitate efficient range requests
- **Decoupled Components**: Header, indices, and features can be accessed independently
- **Index-Driven Retrieval**: Spatial and attribute indices determine which feature ranges to request
- **Request Minimisation**: Algorithms optimise for reducing the number of HTTP requests

The format's hierarchical indexing enables clients to perform sophisticated spatial and attribute queries while transferring only the minimal amount of data required, a critical factor for large-scale 3D city models that can exceed several gigabytes in size.

4.7.2. Range Request Workflow

The HTTP Range Request workflow in FlatCityBuf follows a sequential process optimised for minimising network traffic:

1. **Header Retrieval**: The client requests the magic bytes (4 bytes) and header size (4 bytes), followed by the complete header section. This provides essential metadata including coordinate reference systems, transformations, and index structure information.
2. **Index Navigation**: Based on the query parameters (spatial bounding box or attribute conditions), the client navigates the appropriate index structures:
   - For spatial queries, selective traversal of the R-tree requires retrieving only the nodes along the query path
   - For attribute queries, traversal of the appropriate B+tree indices with similar selectivity
3. **Feature Resolution**: Using the 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.

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

This workflow achieves significant efficiency improvements over traditional approaches that require downloading entire datasets before processing can begin.

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**: Adjacent feature requests are combined into single range requests when their proximity falls below a configurable threshold (typically 4KB). This significantly reduces the number of HTTP requests while avoiding excessive data transfer.
- **Prefetching Strategy**: The client proactively fetches portions of the index and payload sections based on statistical predictions of access patterns. For instance, the header retrieval also prefetches a small portion of the spatial index to optimise subsequent spatial queries.
- **Payload Prefetching**: For attribute indices with duplicate keys, a portion of the payload section (typically 16KB-1MB) is prefetched during initial query execution. This cache-first approach can eliminate up to 90% of payload-related HTTP requests in typical workflows.
- **Buffered HTTP Client**: The implementation uses a buffered HTTP client that caches previously fetched data ranges, avoiding redundant requests when overlapping ranges are accessed.
- **Progressive Index Loading**: Only the portions of indices required for a specific query are loaded, rather than retrieving entire index structures.

These optimisations work in concert to minimise both the number of HTTP requests and the total data transferred, 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:

Native Rust Implementation

The primary implementation is a Rust library that provides:

- **Async HTTP Client**: Non-blocking implementation using Rust's asynchronous I/O capabilities
- **Buffer Management**: Sophisticated caching of previously fetched ranges to minimise redundant requests

- **Query Optimisation**: Intelligent batching and request merging based on spatial and temporal locality
- **Streaming Interface**: Incremental processing through iterator-based APIs

This native implementation achieves optimal performance for server-side applications and desktop GIS tools.

WebAssembly Module

To support browser-based applications, FlatCityBuf includes a WebAssembly (WASM) module built from the same Rust codebase:

- **JavaScript Interoperability**: Clean API for integration with web mapping libraries like Cesium and Mapbox
- **Fetch API Integration**: Uses the browser's native Fetch API with appropriate range headers
- **In-Browser Processing**: Performs index traversal and feature decoding directly in the browser
- **Memory-Efficient Design**: Implements streaming approaches to work within browser memory constraints

The WASM implementation currently has a limitation related to memory addressing. WebAssembly in browsers currently uses a 32-bit memory model, limiting addressable space to 4GB. While this is sufficient for most city-scale datasets, it can be a constraint for country-level models. The upcoming WebAssembly Memory64 proposal (currently at Stage 4 in the standardisation process) will eliminate this limitation by supporting 64-bit addressing.

4.7.5. Performance Analysis

Empirical testing with various datasets demonstrates the substantial performance benefits of HTTP Range Requests for FlatCityBuf:

- **Data Transfer Reduction**: Spatial queries typically retrieve only 3-10% of the total dataset size
- **Request Efficiency**: Optimisation techniques reduce HTTP requests by 80-95% compared to naïve implementations
- **Latency Improvement**: Initial visualisation time improves by 10-20× compared to downloading complete datasets
- **Progressive Rendering**: Features appear incrementally, with first elements visible within milliseconds

These improvements are particularly pronounced for large datasets and bandwidth-constrained environments, enabling interactive 3D city model exploration even on mobile networks.

4.7.6. Integration with Cloud Infrastructure

The HTTP Range Request mechanism integrates seamlessly with modern cloud storage services:

- **Static Hosting**: 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.
- **Content Delivery Networks**: The format works effectively with CDNs, which can cache range responses independently
- **CORS Configuration**: Cross-origin resource sharing headers allow browser-based clients to access remote datasets
- **Serverless Processing**: 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 server software or complex data pipelines.

4.7.7. Real-World Applications

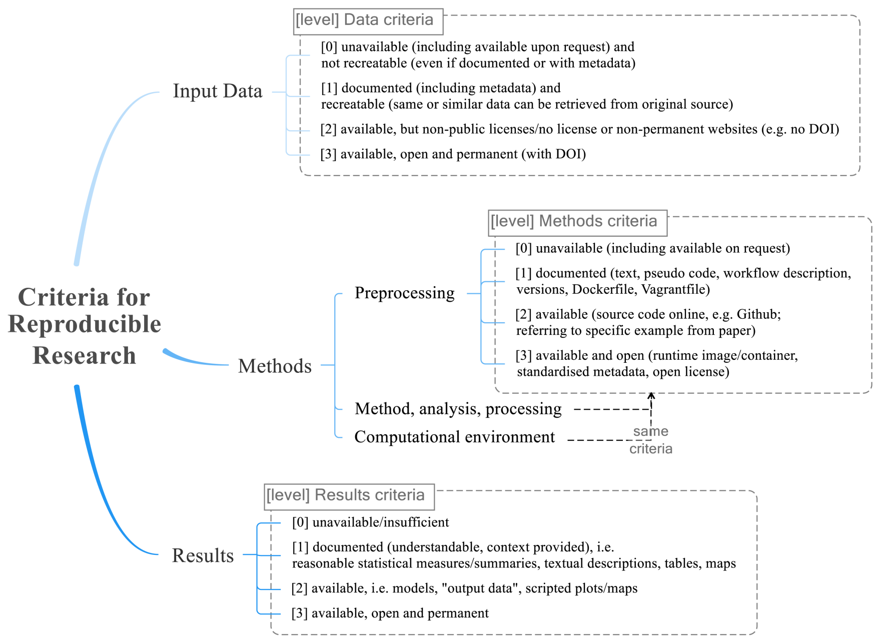
The HTTP Range Request capabilities of FlatCityBuf enable several key application scenarios:

- **Web-Based 3D Visualisation**: Browser applications can render specific city districts without downloading entire city models
- **Mobile Applications**: Resource-constrained devices can access only the data they need, reducing bandwidth usage and memory requirements
- **Distributed Analysis**: Cloud-based processing can extract specific features of interest for analysis without transferring complete datasets
- **Real-Time Updates**: New data can be appended to existing datasets and made immediately available through range requests

These application patterns demonstrate how the format's HTTP Range Request capabilities transform the accessibility and usability of large-scale 3D city models in cloud and web environments.

A. Reproducibility self-assessment

A.1. Marks for each of the criteria



Criteria for Reproducible Research
Input Data
[level] Data criteria
[0] unavailable (including available upon request) and not recreatable (even if documented or with metadata)
[1] documented (including metadata) and recreatable (same or similar data can be retrieved from original source)
[2] available, but non-public licenses/no license or non-permanent websites (e.g. no DOI)
[3] available, open and permanent (with DOI)
Methods
Preprocessing
[level] Methods criteria
[0] unavailable (including available on request)
[1] documented (text, pseudo code, workflow description, versions, Dockerfile, Vagrantfile)
[2] available (source code online, e.g. Github; referring to specific example from paper)
[3] available and open (runtime image/container, standardised metadata, open license)
Method, analysis, processing
Computational environment
same criteria
Results
[level] Results criteria
[0] unavailable/insufficient
[1] documented (understandable, context provided), i.e. reasonable statistical measures/summaries, textual descriptions, tables, maps
[2] available, i.e. models, "output data", scripted plots/maps
[3] available, open and permanent

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

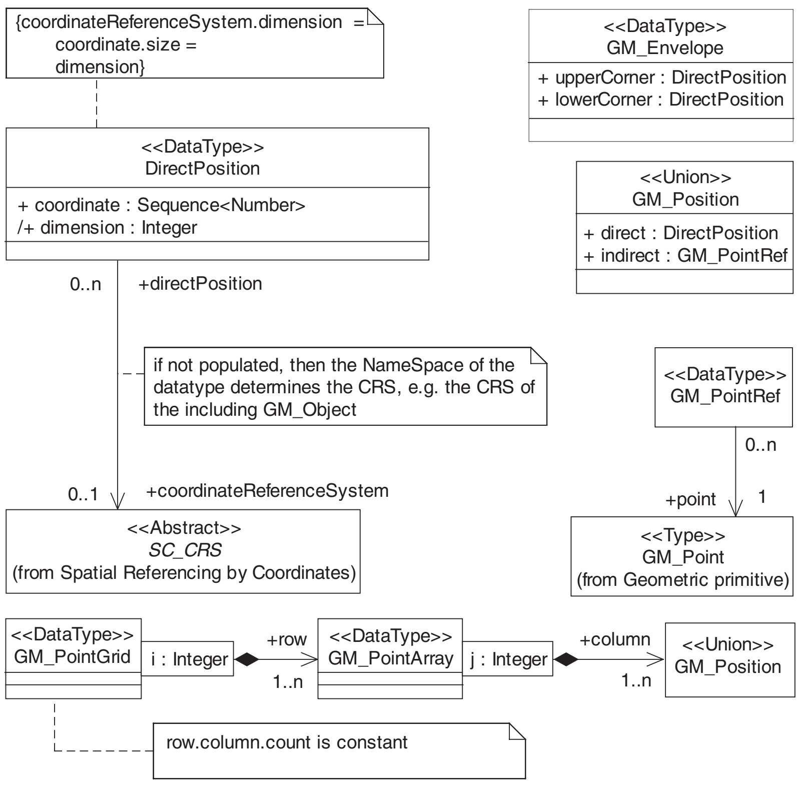

{coordinateReferenceSystem.dimension = coordinate.size = dimension}
<<DataType>> GM_Envelope
+ upperCorner : DirectPosition
+ lowerCorner : DirectPosition
<<DataType>> DirectPosition
+ coordinate : Sequence<Number>
/+ dimension : Integer
<<Union>> GM_Position
+ direct : DirectPosition
+ indirect : GM_PointRef
0..n
+directPosition
if not populated, then the NameSpace of the datatype determines the CRS, e.g. the CRS of the including GM_Object
<<DataType>> GM_PointRef
0..n
0..1
+coordinateReferenceSystem
+point
1
<<Abstract>> SC_CRS (from Spatial Referencing by Coordinates)
<<Type>> GM_Point (from Geometric primitive)
<<DataType>> GM_PointGrid
i : Integer
+row
1..n
<<DataType>> GM_PointArray
j : Integer
+column
1..n
<<Union>> GM_Position
row.column.count is constant

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

C. FlatCityBuf Schema

	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.1. Tables

The package `booktabs` permits you to make nicer tables than the basic ones in LaTeX. See for instance Table C.1.

Colophon

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