FlatCityBuf: A new cloud-optimised CityJSON format

Hidemichi Baba[1], Hugo Ledoux[1], Ravi Peters[2]

[1] Delft University of Technology, the Netherlands—[h.b.baba, h.ledoux]@tudelft.nl
[2] 3DGI, the Netherlands—ravi.peters@3dgi.nl

Keywords: Cloud-optimised geospatial data, CityJSON, FlatBuffers, 3D city models, spatial indexing, CityGML

Abstract

With the increasing availability of large-scale 3D city models, efficient data storage and transmission formats are essential. While the geospatial community has developed *cloud-optimised formats* for 2D datasets (binary files that can be efficiently indexed and accessed through HTTP Range requests), 3D city models with complex geometries, attributes, textures, and semantic surfaces still rely on text-based files using the CityGML standard (CityJSON and XML files). In this paper, we present *FlatCityBuf*, a new compact binary encoding format for 3D city models based on FlatBuffers and CityJSON. Our approach leverages the benefits of FlatBuffers, including cross-platform support, zero-copy data access, and efficient deserialisation, while adhering to the CityGML data model. The addition of spatial and attribute indices enables efficient queries to retrieve partial data. We evaluate the read performance and compression ratios of FlatCityBuf against CityJSONSeq using real-world 3D city models and demonstrate its advantages over existing formats. The results highlight FlatCityBuf's efficient storage and transfer of 3D city model data, achieving for real-world datasets 10–30% compression compared to the already compact CityJSON format; for deserialisation it is 9–250× faster and uses 2–6× less memory. The schemas and accompanying software for conversion to/from CityJSON are publicly available at `https://github.com/cityjson/flatcitybuf` under a permissive license.

1. Introduction

Three-dimensional city models have recently evolved from visualisation tools to fundamental components in urban planning, environmental simulation, and emergency response (Biljecki et al., 2015). National initiatives such as the Netherlands' 3DBAG (Peters et al., 2022), Japan's PLATEAU (PLATEAU, 2020), and Switzerland's SwissBUILDINGS3D (Swiss Federal Office of Topography, 2024) are examples containing millions of buildings. These are usually disseminated in one formats of the CityGML conceptual model (OGC, 2021), that is CityJSON (Ledoux et al., 2019) or CityGML-XML (OGC, 2021).

At the same time, to combat the growth in file size, the (2D) geospatial industry has shifted from desktop-based to *cloud-optimised* formats. These new formats allow us to download on-the-fly subsets of large datasets, increase accessibility and multi-user scalability, and are cross-platform (Cloud-Native Geospatial Foundation, 2023). However, this transition introduces technical challenges including network latency, bandwidth limitations, and the need to serve concurrent users efficiently (Alesheikh et al., 2002).

Traditional 3D city model formats face severe performance limitations in cloud environments. CityGML-XML's text-based format exhibits performance constraints, and its counterpart CityJSON, while being generally 7X smaller, also struggles (Van Liempt, 2020). While the format *CityJSON Text Sequences* (CityJSONSeq) (Ledoux et al., 2024) was developed to enable streaming processing of features, it retains the performance limitations inherent to text-based encoding.

While cloud-optimised geospatial formats have emerged for 2D data (Cloud-Native Geospatial Foundation, 2023), efficient cloud-optimised solutions for 3D city models remain limited. This gap necessitates specialised data formats that operate effectively in cloud computing environments while maintaining semantic richness.

This paper investigates *FlatBuffers* (Google, 2014b) as an encoding mechanism for CityJSONSeq. Our proposed data format, *FlatCityBuf*, combines FlatBuffers' binary serialisation with HTTP Range Requests, enabling partial data retrieval and serverless architectures. As futher defined in Section 3, it incorporates spatial- and attribute- indexing to achieve logarithmic query complexity. It achieves this while being compacter than existing formats and while being more performant in a cloud environment. All the features of CityJSON are preserved: textures, geomatry templates, metadata, etc.

2. Related work

2.1 CityJSON and CityJSONSeq

CityJSON is a JSON-based encoding format, standardised by the OGC, implementing a subset of the CityGML conceptual model (OGC, 2021) As defined in Ledoux et al. (2019), unlike CityGML's hierarchical structure, it employs a flattened architecture and different mechanisms to simplify the structure and create smaller file sizes.

CityJSONSeq modifies CityJSON for streaming applications by decomposing objects into independent `CityJSONFeature` sequences (Ledoux et al., 2024). Each feature maintains local vertex lists and appearance data, ensuring self-containment while adhering to the *Newline Delimited JSON specification*[1]. However, CityJSONSeq's text-based format exhibits several limitations in a cloud environement: lack of explicit data typing, complete data parsing requirements during processing (not possible to obtain a subset without parsing the whole file), and absence of built-in indexing mechanisms for efficient spatial and attribute-based querying.

[1] `https://github.com/ndjson/ndjson-spec/`

2.2 Cloud-Optimised geospatial formats

Cloud-optimised geospatial formats enable efficient on-demand access to geospatial data through reduced latency, scalability, flexibility, and cost-effectiveness (Cloud-Native Geospatial Foundation, 2023). Contemporary implementations include FlatGeobuf, GeoParquet, PMTiles, and Mapbox Vector Tiles, each targeting specific use cases and performance characteristics.

FlatGeobuf implements the Simple Features specification (OGC, 2011) using FlatBuffers serialisation with a packed Hilbert R-tree spatial index (Williams, 2022). This architecture enables efficient serialisation, deserialisation, and selective geographic region retrieval through HTTP Range Requests without requiring complete dataset loading. The format demonstrates excellent deserialisation performance, memory utilisation, and spatial indexing capabilities, making it particularly suitable for web-based applications requiring partial data access.

While these formats have proven successful for 2D geospatial applications, efficient cloud-native solutions specifically designed for 3D city models remain limited, creating opportunities for specialised formats that maintain semantic richness while optimising cloud performance.

2.3 FlatBuffers framework

FlatBuffers is a cross-platform serialisation framework designed for performance-critical applications, implementing zero-copy deserialisation that enables direct access to serialised data without intermediate parsing (Google, 2014b). This characteristic provides significant advantages for large geospatial datasets where parsing overhead impacts performance substantially.

The framework employs schema-based serialisation with strongly typed data structures defined in `.fbs` files, compiled using `flatc` to generate language-specific access code. FlatBuffers supports comprehensive data types including tables (variable-sized objects with optional fields), structs (fixed-size inline aggregates), scalar types, and complex types such as vectors and strings. The binary structure organises data with vtable-based field access, little-endian encoding, and offset-based references, enabling efficient navigation within buffers.

Key technical advantages include memory efficiency through zero-copy access, schema evolution support via backward compatibility, and extensive cross-platform language support. Benchmark analyses indicate FlatBuffers outperforms alternative serialisation formats including Protocol Buffers and JSON in deserialisation efficiency and memory utilisation (Google, 2014a).

3. FlatCityBuf

FlatCityBuf addresses the limitations of text-based 3D city model formats through binary encoding, spatial indexing, and cloud-optimised access patterns. The format maintains semantic compatibility with CityJSON while enabling efficient partial data retrieval through HTTP Range Requests.

Its schema and accompanying software for conversion is available at `https://github.com/cityjson/flatcitybuf`.

3.1 File structure

FlatCityBuf implements a structured binary encoding with five sequentially arranged components (see Figure 1): **(1) Magic bytes**: 8-byte identifier for format validation and version compatibility; **(2) Header section**: Contains metadata, CityJSON properties, coordinate transformations, and indexing metadata encoded as size-prefixed FlatBuffers; **(3) Spatial index**: Packed Hilbert R-tree enabling efficient geospatial queries through 2D spatial indexing; **(4) Attribute index**: Static B+tree structures for accelerated attribute-based filtering with logarithmic complexity; **(5) Features section**: Individual CityJSONFeatures encoded as FlatBuffers tables with zero-copy access.

This sequential structure enables incremental file access critical for cloud applications where minimising data transfer is essential. All components use little-endian encoding with size-prefixed FlatBuffers records to facilitate precise HTTP Range Requests.

The magic bytes (`FCB01000`) provide immediate file type identification and version compatibility checking, following the approach established by cloud-optimised formats like FlatGeoBuf. This signature design enables applications to validate file type and version compatibility without parsing the entire header content.

FlatCityBuf leverages FlatBuffers' zero-copy deserialisation to eliminate parsing overhead common in JSON-based formats, using schema-based serialisation with size-prefixed records for efficient memory access. The approach prioritises read performance over update capabilities while maintaining full CityJSON compatibility .

3.2 Header Section

The header section encapsulates essential metadata for file interpretation, implemented as a size-prefixed FlatBuffers-serialised table. Following the approach of CityJSONSeq, the header maintains compatibility with CityJSON's global properties while adding FlatCityBuf specific extensions for optimised retrieval.

Core metadata components. The header preserves essential CityJSON metadata including version identifiers, coordinate reference system information, transformation parameters for coordinate quantisation, and geographical extent definitions. Additional fields support appearance information (materials and textures), geometry templates for reusable structures, and CityJSON extension mechanisms through embedded schema definitions. Key technical components include: **(1) Coordinate transformation**: Scale and translation vectors enabling efficient vertex coordinate storage through quantisation, maintaining precision while reducing storage requirements; **(2) Appearance model**: Global material and texture definitions with UV coordinate mappings, supporting visual representation properties defined in the CityJSON specifications; **(3) Geometry templates**: Reusable geometry definitions enabling storage efficiency for datasets with repetitive structures such as standardised building designs or street furniture; **(4) Extension support**: Embedded JSON schema definitions for CityJSON extensions, creating self-contained files without external dependencies.

Attribute Schema and indexing metadata. The header includes both attribute schema definitions and indexing-specific

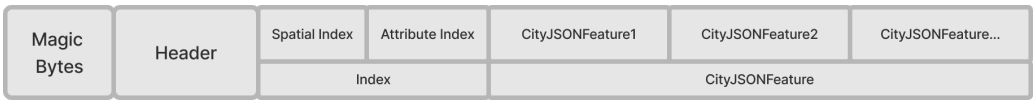


Figure 1. FlatCityBuf file structure showing sequential component layout optimised for HTTP Range Requests

metadata essential for data interpretation and efficient querying: **(1) Attribute schema**: Column definitions specifying data types, nullability, and validation constraints for feature attributes, enabling interpretation of binary-encoded attribute value; **(2) Spatial index parameters**: Node size configuration and feature count information for spatial index interpretation; **(3) Attribute index metadata**: Branching factors, byte lengths, and unique value counts for each indexed attribute, enabling efficient B+tree traversal.

The attribute schema system, directly adopted from FlatGeoBuf's approach, provides a flexible framework for defining attribute schemas while maintaining compatibility with 3D city model requirements. The indexing metadata enables client applications to determine optimal query strategies and navigate the spatial and attribute indices efficiently without parsing the entire dataset. The spatial index uses a configurable node size (defaulting to 16 entries) optimised for typical HTTP request patterns, while attribute indices store B+tree structural information including branching factors and unique value counts for query optimisation.

Binary Encoding conventions. FlatCityBuf follows two key conventions for consistent binary data encoding: **(1) Size-prefixed FlatBuffers**: All FlatBuffers records include a 4-byte prefix indicating buffer size, enabling precise HTTP Range Requests without parsing entire content; **(2) Little-endian encoding**: Consistent byte ordering for all numeric values outside FlatBuffers records, matching modern CPU architectures and FlatBuffers conventions.

These conventions ensure format consistency and maximise compatibility with modern architectures while facilitating efficient cloud-based access patterns.

3.3 Spatial Index Section

Efficient spatial querying is critical for cloud-based 3D city model access where minimising data transfer is essential. FlatCityBuf implements spatial indexing through a packed Hilbert R-tree mechanism, directly adapting the proven approach from FlatGeoBuf (FlatGeobuf, 2020a).

Packed Hilbert R-tree Implementation. The spatial indexing mechanism reuses FlatGeoBuf's packed Hilbert R-tree implementation with minimal modifications for 3D city model integration. Key characteristics include: **(1) Hilbert curve ordering**: Features are spatially sorted using a Hilbert space-filling curve to optimise data locality, ensuring spatially proximate features are stored adjacently in the file; **(2) Packed tree structure**: The R-tree is maximally filled with no empty internal slots, optimised for static datasets through bottom-up construction; **(3) Flattened storage**: Level-ordered tree serialisation enables efficient streaming and remote access via HTTP Range Requests; **(4) Fixed-size nodes**: Each node contains bounding box coordinates (4 double values) and byte offset (64-bit unsigned integer), enabling predictable memory layouts.

The implementation directly incorporates FlatGeoBuf's reference implementation (FlatGeobuf, 2020b), which itself draws from Vladimir Agafonkin's flatbush library (Agafonkin, 2010). The Hilbert curve encoding algorithm follows the non-recursive approach described by Warren (Warren, 2012), ensuring efficient 2D coordinate mapping to 1D ordering values.

2D Spatial Indexing for 3D City Models. Although designed for 3D city models, FlatCityBuf deliberately employs 2D spatial indexing rather than full 3D implementation. This design decision reflects the horizontal distribution characteristics of most 3D city models, which are primarily distributed horizontally with limited vertical extent relative to their horizontal footprint. Additionally, typical spatial queries focus on horizontal regions (e.g., buildings within districts) rather than volumetric queries, aligning with how OGC API standards (OGC, 2019a) primarily support 2D spatial querying.

The indexing process extracts 2D bounding boxes from city features by calculating minimum and maximum X,Y coordinates across all vertices. Feature centroids are encoded using 32-bit Hilbert values for sorting, while complete 2D bounding boxes are stored for spatial intersection tests.

The packed R-tree structure enables efficient cloud-native access through incremental traversal via HTTP Range Requests, predictable byte layouts for remote access, and Hilbert-sorted feature ordering that allows to batched HTTP requests.

3.4 Attribute index section

Efficient attribute-based querying is essential for 3D city model applications requiring feature filtering based on non-spatial properties. FlatCityBuf implements attribute indexing through Static B+trees, enabling logarithmic query complexity for attribute-based operations.

FlatCityBuf adopts a Static B+tree (S+tree) (Slotin, 2021) with significant modifications. The design prioritises read-only access patterns common in cloud-based applications while supporting diverse query operators defined in OGC Filter Encoding (OGC, 2010) and Common Query Language (OGC, 2024) standards.

Similar to the spatial index, the attribute index uses an immutable structure that remains fixed once constructed, eliminating rebalancing operations and optimising for static datasets. All nodes except rightmost nodes at each level are maximally filled to ensure space efficiency, with trees built bottom-up from sorted data in a single pass rather than through incremental insertions. The predictable layout means tree structure is determined solely by element count and node size, enabling efficient navigation patterns essential for HTTP Range Request access.

The implementation supports common comparison operators and logical combinations addressing typical 3D city model query patterns. Supported operators include equality (`building_type = "residential"`), inequality (`city_name != "Amsterdam"`), comparison operators (`height > 25`), range queries (`floor_count BETWEEN 3 AND 8`), and logical combinations (`height > 15 AND building_type = "office"`). These roughly follow OGC Filter Encoding standards (OGC, 2010) while maintaining efficient cloud-based performance.

Modifications for FlatCityBuf. Several modifications adapt the S+tree algorithm (Slotin, 2021) for FlatCityBuf requirements: **(1) Duplicate key handling**: City model attributes often contain numerous duplicate values (e.g., hundreds of features with identical city names). The implementation incorporates a dedicated payload section storing multiple feature references for identical attribute values without compromising tree structure or search performance; **(2) Multi-type support**: Extended to handle various attribute data types including numeric types (integers, floating-point), string values, boolean flags, and temporal data; **(3) Explicit node offsets**: Stores explicit byte offsets to child nodes rather than mathematical calculations, simplifying implementation while maintaining performance; **(4) Payload pointer mechanism**: Uses tag bits in offset values to distinguish between direct feature references and pointers to payload sections containing multiple feature offsets.

These modifications ensure optimal performance while preserving S+tree algorithmic advantages.

Key serialisation strategy. Key serialisation balances storage efficiency, comparison performance, and implementation complexity through type-specific strategies: **(1) Fixed-length values**: Integer and floating-point types use native binary representation with little-endian encoding. Temporal values use composite encoding (i64 seconds + u32 nanoseconds) supporting full ISO 8601 (ISO, 2017) datetime ranges; **(2) Variable-length strings**: Employ fixed-length encoding with 50-byte maximum length based on analysis of common 3D city model string attributes. Shorter strings are space-padded to ensure consistent key sizes throughout tree structures; **(3) Boolean values**: Single-byte encoding (0/1) maintaining sort order, though boolean attributes rarely provide effective indexing due to limited discriminative power.

This approach prioritises implementation simplicity while supporting the most common attribute types found in 3D city model datasets.

Index construction and query execution. Index construction follows a systematic process: **(1)** Create attribute value and feature offset pairs; **(2)** Sort pairs by attribute values; **(3)** Generate payload section grouping feature offsets for duplicate values; **(4)** Build tree structure bottom-up using configured branching factor; **(5)** Serialise tree in top-down order with explicit byte offsets.

Query execution leverages two core functions: `find_exact_match` for precise value location and `find_partition_point` for boundary identification in range operations. Range queries combine lower and upper bound determination with result set retrieval, while inequality queries use exact matching with result set negation.

Figure 2 illustrates the complete S+tree structure implementation in FlatCityBuf, showing the hierarchical arrangement of tree nodes, the payload section for handling duplicate keys, and the connections to feature references.

3.5 Feature Section

The feature section stores individual city features as FlatBuffers-encoded binary structures, preserving CityJSON's semantic richness while enabling zero-copy access. This component represents the core contribution of mapping CityJSON data structures to efficient binary encoding.

CityJSONFeature to FlatBuffers mapping. FlatCityBuf implements CityJSONSeq's core structure through FlatBuffers tables that maintain semantic compatibility while optimising for binary access: **(1) CityFeature**: Top-level container with identifier, collection of CityObjects, quantised vertices, and optional appearance information; **(2) CityObject**: Individual city objects containing type classification, geographical extent, geometry arrays, binary-encoded attributes, and hierarchical relationships; **(3) Geometry**: Boundary representation following CityJSON's geometric model with flattened arrays for FlatBuffers compatibility; **(4) SemanticObject**: Surface classification system supporting both standard CityJSON semantic types and extension mechanisms.

This mapping preserves all CityJSON capabilities including coordinate quantisation, geometry templates, semantic surfaces, and extension support while enabling direct binary access without parsing overhead.

Hierarchical boundary encoding. A key technical challenge involves adapting CityJSON's recursive boundary representation to FlatBuffers' flat array structure. FlatCityBuf addresses this through dimensional hierarchy encoding using parallel flattened arrays: **boundaries**: Single array of vertex indices referencing the feature's vertex list; **strings**: Array indicating vertex counts per ring/boundary; **surfaces**: Array indicating string counts per surface; **shells**: Array indicating surface counts per shell; **solids**: Array indicating shell counts per solid.

For example, a simple triangle encodes as:

```
boundaries: [0,1,2]
strings: [3]
surfaces: [1]
```

Complex geometries like buildings use the full dimensional hierarchy. This approach maintains CityJSON's geometric expressiveness while enabling efficient binary storage and access.

Semantic surfaces follow the same hierarchical boundary encoding approach, using parallel arrays to classify geometric components with semantic meaning (WallSurface, RoofSurface, etc.). This maintains consistency with the geometric boundary structure while preserving CityJSON's semantic richness.

Attribute encoding strategy. Attributes in FlatCityBuf are encoded as binary data with schema definitions provided through Column tables in the header (see Figure 3) The encoding strategy focuses on efficient type-specific serialisation, and we use length-prefixed JSON string encoding for nested JSON objects.

Each attribute is stored as a key-value pair where the key represents the column index and the value contains the binary-encoded attribute data. Semantic surface attributes follow the same encoding strategy as city object attributes, using identical type-specific serialisation for consistency across the format.

Extension mechanism. FlatCityBuf supports CityJSON's Extension mechanism for data model customisation. Extended city object types (prefixed with `"+"`) use a two-part encoding: standard enum values (`ExtensionObject`, `ExtraSemanticSurface`) combined with extension type strings (e.g., `"+NoiseCityFurnitureSegment"`). Extension attributes follow the same binary serialisation as core attributes.

It should be noticed that unlike CityJSON's external schema references, FlatCityBuf embeds all extension information within

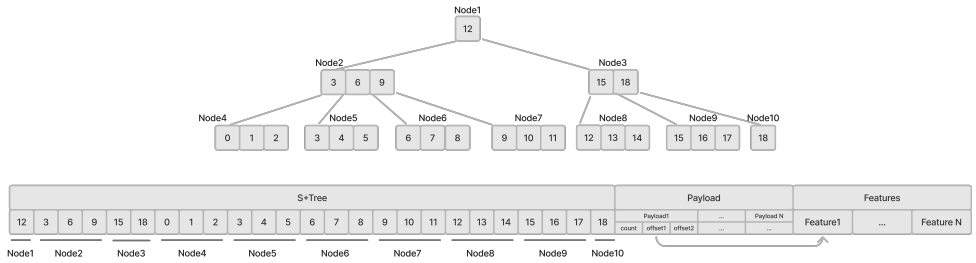


Figure 2. Static B+tree structure implementation in FlatCityBuf showing tree nodes, payload section, and feature references

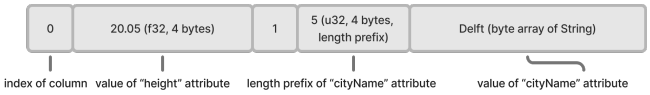


Figure 3. Example of attribute encoding in FlatCityBuf showing type-specific binary serialisation

the file, maintaining cloud-optimised self-containment while preserving full extension compatibility.

3.6 HTTP Range request implementation

HTTP Range Requests enable FlatCityBuf's cloud-optimised data access by allowing selective retrieval of file segments without downloading entire datasets. This capability is fundamental to serverless architectures and efficient bandwidth utilisation.

Partial data retrieval principles. HTTP Range Requests, defined in RFC 7233 (Internet Engineering Task Force), 2014), allow clients to request specific byte ranges from server resources. FlatCityBuf leverages this capability through strategic file organisation and size-prefixed record design. Since each feature uses length-prefixed encoding, clients can determine exact byte boundaries and request precisely the required data segments.

The approach supports diverse access patterns including sequential reading, spatial index traversal, and attribute-based filtering while maintaining consistent performance characteristics regardless of query complexity.

Range Request workflow. FlatCityBuf implements a systematic workflow that minimises both HTTP request frequency and total data transfer volume while supporting complex spatial and attribute query combinations: **(1) Header retrieval**: Initial request fetches magic bytes and header section, providing essential metadata including coordinate systems, transformations, feature counts, and index structure information; **(2) Index navigation**: Based on query parameters, clients selectively traverse relevant index structures (spatial R-tree for bounding box queries, attribute B+trees for property filters) using targeted range requests; **(3) Feature resolution**: Using byte offsets from index traversal, clients make targeted requests for specific features with sizes determined by consecutive offset differences; **(4) Progressive processing**: Features are processed incrementally as they arrive, enabling responsive user interfaces before complete data retrieval.

Network optimisation techniques. Several techniques, primarily derived from FlatGeoBuf's approach (FlatGeobuf, 2020a), minimise network latency overhead: **(1) Request batching**: Multiple feature requests are consolidated into larger HTTP requests, reducing round-trip frequency while maintaining precise data boundaries; **(2) Payload prefetching**: Attribute index payload sections are proactively downloaded when indexes are accessed, reducing latency for subsequent operations; **(3) Streaming index traversal**: Index structures load only necessary nodes during tree navigation rather than complete index structures, supporting efficient remote traversal; **(4) Buffered HTTP client**: Implementation uses cached range clients that avoid redundant requests for overlapping byte ranges.

4. Experiments with real-world data

This section presents comprehensive evaluations of FlatCityBuf performance across multiple dimensions including implementation outcomes, storage efficiency, local processing benchmarks, and web-based performance characteristics.

FlatCityBuf implementation produced several tangible research outcomes demonstrating practical applicability:

- **Reference implementation**: Comprehensive Rust library for encoding/decoding (from/to CityJSONSeq), and querying FlatCityBuf files with command-line interface tools for conversion and validation;
- **Cross-platform support**: Native Rust library and WebAssembly (W3C, 2019) module enabling deployment across server-side applications, desktop GIS tools, and browser-based environments;
- **Web demo**: Functional prototype accessible at `https://flatcitybuf-prototype.hideba.me/` operating on a 3.4GB dataset covering 20kmX20km of the Netherlands, demonstrating spatial and attribute query capabilities through HTTP Range Requests;
- **Cloud integration**: Seamless integration with object storage services (AWS S3, Google Cloud Storage, Azure Blob Storage) supporting serverless architectures without specialised server processing

The web prototype demonstrates practical performance improvements by enabling responsive interaction with large 3D city models without downloading entire datasets, providing spatial querying, attribute filtering, and data export capabilities.

4.1 File size analysis

We analysed the file size of diverse real-world datasets, we used the same datasets as those from Ledoux et al. (2024), and supplemented them with PLATEAU datasets (PLATEAU, 2020); see Table 1.

Results demonstrate that FlatCityBuf achieves superior compression for several datasets (Helsinki, Ingolstadt, NYC, Zürich) with compression factors of 16-24%. Conversely, PLATEAU

Table 1. File size and deserialisation (time and memory) for several openly available real-world datasets.

	File size			Deserialisation time			Deserialisation memory		
Dataset	**CityJSONSeq**	**FlatCityBuf**	**compress**	**CityJSONSeq**	**FlatCityBuf**	**ratio**	**CityJSONSeq**	**FlatCityBuf**	**ratio**
3DBAG	6MB	6MB	-6%	56.3ms	6.6ms	8.6×	23.9MB	5.1MB	4.7×
3DBV	317MB	281MB	12%	3.99s	122.5ms	32.6×	283.8MB	63.2MB	4.5×
Helsinki	412MB	345MB	16%	4.05s	132.2ms	30.6×	15.3MB	5.2MB	2.9×
Ingolstadt	4MB	3MB	19%	37.2ms	0.5ms	75.8×	30.1MB	6.9MB	4.4×
Montréal	5MB	5MB	-4%	50.3ms	0.6ms	81.6×	36.3MB	5.7MB	6.4×
NYC	95MB	76MB	20%	887.6ms	42.9ms	20.7×	20.6MB	5.0MB	4.1×
Rotterdam	3MB	3MB	-4%	22.2ms	1.3ms	17.6×	9.2MB	4.4MB	2.1×
Vienna	5MB	4MB	14%	45.9ms	1.9ms	24.0×	14.6MB	5.2MB	2.8×
Zürich	247MB	189MB	24%	1.88s	151.9ms	12.4×	31.3MB	5.1MB	6.2×
Building	77MB	79MB	-3%	861.4ms	32.5ms	26.5×	220.9MB	64.4MB	3.4×
Bridge	5MB	5MB	-9%	83.9ms	0.3ms	256.8×	75.0MB	12.0MB	6.3×
Railway	4MB	4MB	-2%	37.9ms	2.0ms	18.5×	19.0MB	5.1MB	3.8×
Transport	26MB	26MB	-1%	244.0ms	13.3ms	18.4×	76.7MB	20.2MB	3.8×
Tunnels	5MB	5MB	4%	47.9ms	1.9ms	24.9×	70.6MB	12.6MB	5.6×
Vegetation	2MB	2MB	-31%	852.3ms	32.9ms	25.9×	189.8MB	56.9MB	3.3×

Figure 4. Our web prototype demonstrating FlatCityBuf's spatial and attribute query capabilities on a 3.4GB dataset, accessible at `https://flatcitybuf-prototype.hideba.me/`.

datasets exhibit mixed results, with vegetation data showing increased file sizes. Analysis reveals that compression efficiency depends on multiple interconnected factors rather than format limitations.

We ran controlled experiments and identified four key factors affecting compression performance: **(1) Level of Detail (LoD)**: Compression remains consistent at 25% across LoD0 to LoD2.2, indicating format design determines efficiency regardless of geometric detail level; **(2) Number of attributes**: Compression improves dramatically from 5% (10 attributes) to 44% (1000 attributes) due to FlatCityBuf's schema-based approach eliminating redundant key storage (which are very common in practice); **(3) Geometric complexity**: More complex geometries achieve better compression (26% vs 15%) as FlatCityBuf's fixed-size encoding becomes advantageous over text-based representation with larger boundary fields; **(4) Coordinate scale**: Large coordinate values favour FlatCityBuf (18% compression) while small values favour CityJSONSeq (-29%), explaining superior performance for metric coordinate systems versus geographic coordinates.

These findings explain dataset-specific performance variations, with FlatCityBuf demonstrating optimal efficiency for attribute-rich datasets using large-scale coordinate systems with complex geometries.

4.2 Deserialisation performance

Table 1 shows the performance results of a local benchmark that reads all features from each dataset and accesses their geometry type field. This processing task requires deserialisation for both formats, but FlatCityBuf benefits from zero-copy deserialisation where data can be accessed directly without parsing overhead. The table presents both processing time and memory consumption for this representative read operation.

FlatCityBuf demonstrates substantial performance improvements across all datasets, achieving processing time speedups ranging from 8.6× (3DBAG) to 256.8× (PLATEAU Bridge). Memory consumption reductions range from 2.1× to 7.6×, with particularly notable improvements for larger datasets where parsing overhead becomes significant.

The results highlight FlatBuffers' zero-copy deserialisation advantages, where smaller datasets like PLATEAU Bridge show the most dramatic improvements due to proportionally higher parsing overhead in traditional JSON processing.

4.3 Web performance evaluation

Web-based performance evaluation compares FlatCityBuf's HTTP Range Request approach against the 3DBAG API (3DBAG, 2023) for realistic cloud-based access patterns. Testing used the complete Netherlands 3DBAG dataset, comprising 65.2GB in CityJSONSeq format, 63.89GB in FlatCityBuf without indexing, and 70.8GB in FlatCityBuf with full spatial and attribute indexing. Performance evaluation utilized the indexed 70.8GB FlatCityBuf file with spatial and identifier-based queries across 100 samples with 10 warmup iterations.

Results demonstrate substantial performance advantages for both query types:

- **Feature ID queries**: Testing across 5 landmark buildings (TU Delft, Amsterdam Central Station, Groningen Station, Eindhoven Station, Enschede Station) shows FlatCityBuf averaging 1012.9ms versus 3DBAG API's 2169.2ms, achieving 2.1× speedup through attribute indexing and logarithmic lookup complexity
- **Spatial bounding box queries**: 2km × 2km area around TU Delft campus demonstrates FlatCityBuf's 492.6ms versus 3DBAG API's 7420.3ms, achieving 15.1× speedup through Hilbert curve spatial sorting and batched HTTP Range Request operations

While acknowledging architectural differences between static file access and dynamic API services, the evaluation demonstrates FlatCityBuf's practical advantages for cloud-native 3D

city model applications. The 15.1× improvement for spatial queries particularly highlights the benefits of client-side spatial indexing combined with range request optimization.

The web prototype validates these performance characteristics by providing responsive interaction with multi-gigabyte datasets through standard web browsers, demonstrating the format's practical applicability for modern web-based GIS applications.

5. Discussion

5.1 Use cases of FlatCityBuf

FlatCityBuf demonstrates particular advantages in scenarios requiring flexible data access and high-performance data processing. The format enables precise data retrieval through selective downloading, addressing limitations of existing services that constrain users to predefined tiles. The web prototype demonstrates users can download filtered datasets based on specific criteria (e.g., buildings exceeding 100m in height if it is stored as an attribute) through attribute indexing mechanisms.

For large-scale data processing pipelines, FlatCityBuf's superior read performance addresses I/O bottlenecks that typically constrain analytical workflows. The 3DBAG generation pipeline exemplifies this application, where multiple processing stages requiring CityJSONSeq file access would benefit substantially from improved I/O efficiency. Additionally, the format simplifies analytical workflows by encapsulating data in unified files accessible across platforms, eliminating the complexity of managing chunked datasets across multiple files while maintaining high performance for both selective queries and full dataset processing.

5.2 Impact on Server Architecture

FlatCityBuf enables fundamental simplification of server architectures for 3D city model delivery. Traditional approaches typically employ both application and database servers. For example, 3DCityDB (Yao et al., 2018) utilises PostgreSQL or Oracle as the database server with PostgREST API providing data access through its toolchain. Similarly, the 3DBAG API[2] uses PostgreSQL as its database server and Flask (Python web framework) as the application server.

In contrast, FlatCityBuf operates as a static file requiring only basic HTTP servers such as *nginx*[3], aligning with modern cloud service offerings like *AWS S3*[4] and *Google Cloud Storage*[5].

This architectural shift provides substantial scalability advantages. Traditional RDBMS-based systems encounter scaling limitations requiring sharding, replication, or resource expansion, all demanding additional computational resources. FlatCityBuf circumvents these challenges by leveraging cloud providers' inherent scalability and high availability infrastructure, enabling unrestricted access without rate-limiting mechanisms typically necessary with traditional architectures.

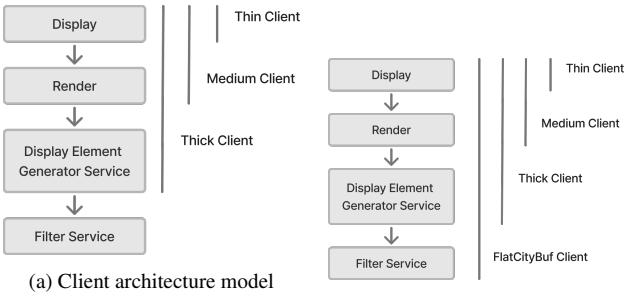
Cost-effectiveness represents another significant advantage. While precise comparisons vary by use case, hosting static files through cloud providers is substantially more economical than maintaining dedicated database and application servers. For example, Google Cloud Storage costs $0.020 USD per GB per month in the Netherlands (europe-west4) region[6]. In contrast, computing services such as Compute Engine cost $0.25 USD per vCPU hour for on-demand instances in the same region (4 vCPU, 16 GiB memory, 375 GiB SSD)[7]. This demonstrates the fundamental cost advantage of storage-based over compute-intensive architectures.

5.3 Limitations

Despite advantages in simplicity, scalability, and cost-effectiveness, FlatCityBuf presents notable limitations. Query flexibility remains more constrained than specialised spatial database applications. Traditional approaches employing RDBMS with spatial indexing provide more comprehensive query functionality. For instance, 3DCityDB enables filtering by LoD, `CityObject` type (Yao et al., 2018), and various other parameters, whereas FlatCityBuf primarily supports spatial- attribute-based filtering. Similarly, regarding spatial functions, 3DCityDB can utilise the extensive spatial capabilities of PostGIS (PostGIS, 2001), while FlatCityBuf currently only implements bounding box queries, nearest neighbour queries, and point intersection queries.

Furthermore, client-side application complexity increases as FlatCityBuf shifts computational responsibility from server to client. This shift follows the client-server architecture spectrum described by Alesheikh et al. (2002), who categorised systems ranging from "Thin Client" (where clients primarily handle display) to "Thick Client" (where clients perform most processing tasks). FlatCityBuf represents an extreme case of the "Thick Client" architecture, where clients assume responsibility for filtering services in addition to other processing tasks (Figure 5).

This architectural choice impacts interoperability. OGC API (OGC, 2019b) and equivalent Web API services adhere to standardised designs that enable universal client access—whether through command-line interfaces, web browsers, or mobile applications. While FlatCityBuf supports cross-platform deployment, it requires language-specific or platform-specific library implementations, potentially limiting accessibility compared to standard web APIs.



(a) Client architecture model modified from Alesheikh et al. (2002)

(b) FlatCityBuf architecture with Alesheikh et al. (2002)'s model

Figure 5. Comparison of client complexity with Alesheikh et al. (2002)'s model and FlatCityBuf's architecture.

Furthermore, since the format contains immutable spatial and attribute indices, updating data necessitates rewriting the entire

[2] `https://api.3dbag.nl`
[3] `https://nginx.org/`
[4] `https://aws.amazon.com/s3/`
[5] `https://cloud.google.com/storage`

[6] Google Cloud Storage Pricing, `https://cloud.google.com/storage/pricing#europe`, accessed January 2025
[7] Google Cloud Compute Engine Pricing, `https://cloud.google.com/compute/all-pricing?hl=en`, accessed January 2025

file. This constraint renders FlatCityBuf less suitable for frequently updated datasets, positioning it optimally for data analysis and efficient download services rather than dynamic data management scenarios.

5.4 Future work

Several promising directions emerge for extending FlatCityBuf's capabilities and adoption. Expanding language support beyond Rust would enhance accessibility and ecosystem integration, particularly for languages with garbage collection mechanisms such as Python, JavaScript, and Java. Python support would enable seamless integration with geospatial analysis workflows, while JavaScript implementation would facilitate web-based visualisation without WebAssembly dependencies.

Acknowledgements

This research was carried out within the framework of the MultiRoofs project, co-funded by the European Union through the Interreg North-West Europe Programme.

References

3DBAG, 2023. 3DBAG API. https://3dbag.nl/api/ (accessed: 2025-01-20).

Agafonkin, V., 2010. Flatbush: A very fast static spatial index for 2D points and rectangles in javascript. https://github.com/mourner/flatbush (accessed: 2025-06-08).

Alesheikh, A. A., Helali, H., Behroz, H. A., 2002. Web GIS: technologies and its applications. *ISPRS Symposium on Geospatial Theory, Processing and Applications*, 1-9.

Biljecki, F., Stoter, J., Ledoux, H., Zlatanova, S., Çöltekin, A., 2015. Applications of 3D City Models: State of the Art Review. *ISPRS International Journal of Geo-Information*, 4(4), 2842.

Cloud-Native Geospatial Foundation, 2023. Cloud-Optimised Geospatial Formats Guide. https://guide.cloudnativegeo.org/ (accessed: 2025-06-08).

FlatGeobuf, 2020a. FlatGeobuf. https://flatgeobuf.org/ (accessed: 2024-12-17).

FlatGeobuf, 2020b. FlatGeobuf GitHub Repository. https://github.com/flatgeobuf/flatgeobuf/ (accessed: 2024-12-17).

Google, 2014a. C++ Benchmarks. https://flatbuffers.dev/flatbuffers_benchmarks.html (accessed: 2025-01-13).

Google, 2014b. FlatBuffers. https://flatbuffers.dev/ (accessed: 2024-12-17).

Internet Engineering Task Force), 2014. Hypertext Transfer Protocol (HTTP/1.1): Range Requests. Accessed: 2025-01-13.

ISO, 2017. Iso 8601 — date and time format. https://www.iso.org/iso-8601-date-and-time-format.html (accessed: 2025-06-08).

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

Ledoux, H., Stavropoulou, G., Dukai, B., 2024. Streaming cityjson datasets. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives*, 48, International Society for Photogrammetry and Remote Sensing, 57–63.

OGC, 2010. Filter Encoding Standard. Accessed: 2025-06-08.

OGC, 2011. Simple Features Access. https://www.ogc.org/publications/standard/sfa/ (accessed: 2025-06-08).

OGC, 2019a. OGC API - Features 1.0 - Part 1: Core.

OGC, 2019b. OGC API - Features Standard.

OGC, 2021. OGC City Geography Markup Language (CityGML) Part 1: Conceptual Model Standard. Open Geospatial Consortium inc. Document 20-010, version 3.0.0, available at https://docs.ogc.org/is/20-010/20-010.html.

OGC, 2024. Common Query Language (CQL2). Accessed: 2025-06-08.

Peters, R., Dukai, B., Vitalis, S., van Liempt, J., Stoter, J., 2022. Automated 3D reconstruction of LoD2 and LoD1 models for all 10 million buildings of the Netherlands. *Photogrammetric Engineering and Remote Sensing*, 88(3), 165–170.

PLATEAU, 2020. PLATEAU. https://www.mlit.go.jp/plateau/ (accessed: 2025-01-13).

PostGIS, 2001. PostGIS. Accessed: 2025-06-08.

Slotin, S., 2021. Static B-Trees. https://en.algorithmica.org/hpc/data-structures/s-tree/ (accessed: 2025-06-08).

Swiss Federal Office of Topography, 2024. SwissBUILDINGS3D 3.0 Beta. https://www.swisstopo.admin.ch/en/landscape-model-swissbuildings3d-3-0-beta (accessed: 2025-06-08).

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

W3C, 2019. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-1/ (accessed: 2025-06-08).

Warren, H. S., 2012. Hacker's Delight, Second Edition. https://www.oreilly.com/library/view/hackers-delight-second/9780133084993/ (accessed: 2025-06-08).

Williams, H., 2022. Kicking the Tires: Flatgeobuf. https://worace.works/2022/02/23/kicking-the-tires-flatgeobuf/.

Yao, Z., Nagel, C., Kunde, F., Hudra, G., Willkomm, P., Donaubauer, A., Adolphi, T., Kolbe, T. H., 2018. 3DCityDB — a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. *Open Geospatial Data, Software and Standards*, 3(2).