FlatCityBuf: a new cloud-optimised CityJSON format

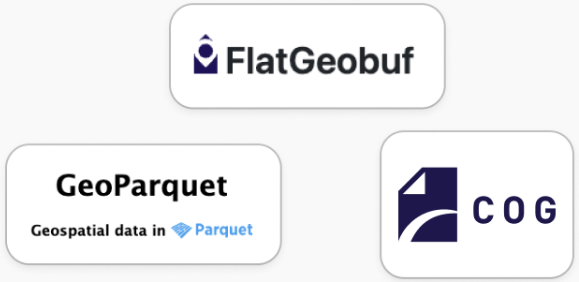
Hidemichi Baba, Hugo Ledoux, and Ravi Peters

TU Delft 3D geoinformation group, Netherlands

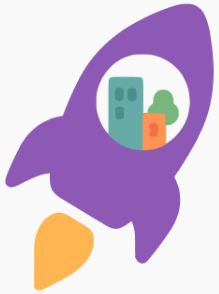
Cloud Native Computing Symposium 2025

Goal

Cloud optimised

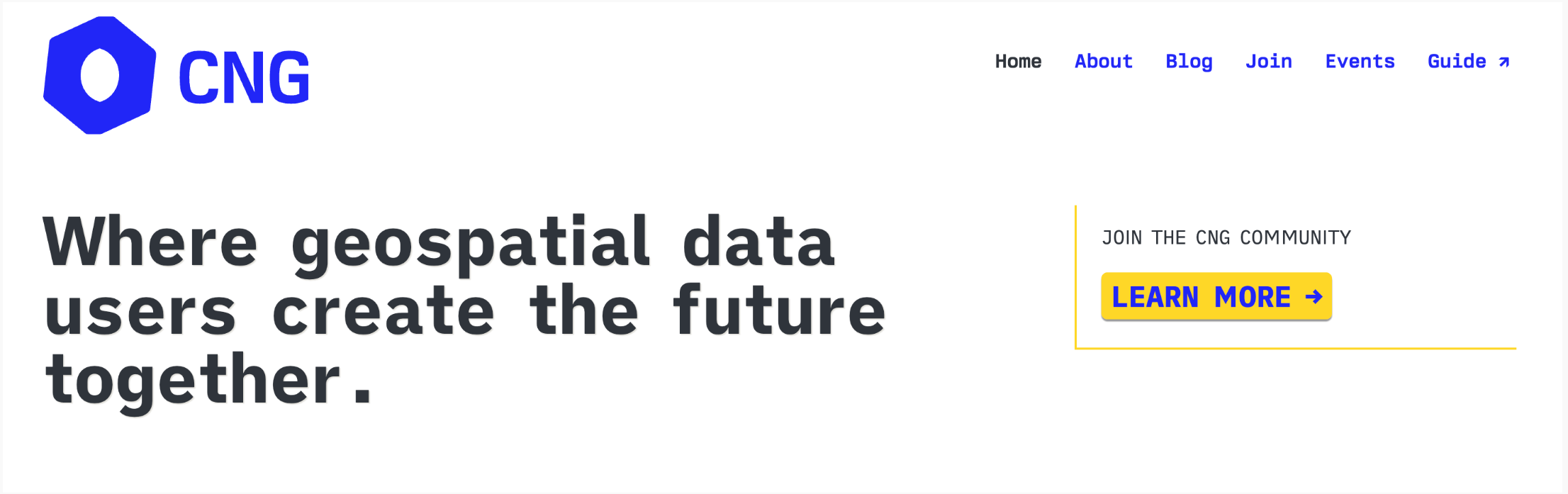


=



FlatCityBuf

There is an conference



(*The Cloud-Native Geospatial Forum, 2025*)

Geonovum Symposium as well

(*Geonovum Symposium, 2025*)

Traditional vs Cloud-Optimised Geospatial Formats

Traditional Formats

Download → Parse → Use

Examples: Shapefile, GeoJSON, GeoTIFF

Cloud-Optimised Formats

Query → Stream → Use

Examples: COG, FlatGeobuf, GeoParquet

A 2D Revolution: Cloud-Optimised Formats

A paradigm shift in geospatial data delivery

- **Cloud Optimised GeoTIFF (COG)**
- **FlatGeobuf**
- **GeoParquet**
- **PMTiles**

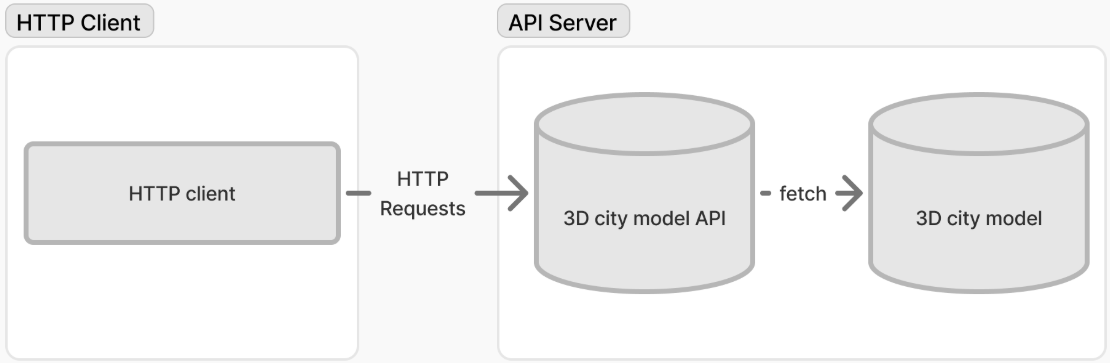
Key Innovation:

- **Object storage instead of databases**
- **Web-optimised delivery**
- **No backend infrastructure**

Server Architecture: Traditional vs Cloud Optimised

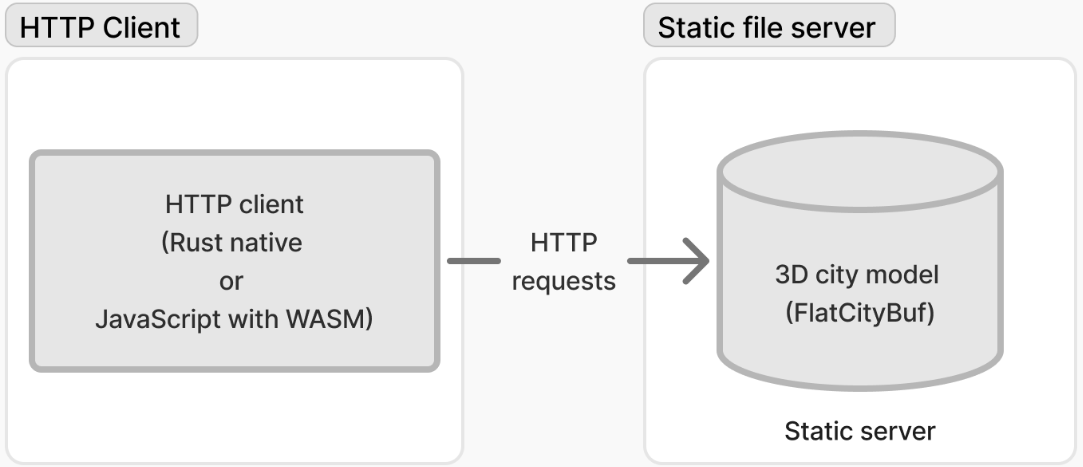
Traditional Server Architecture

Complex, less scalable and expensive



FlatCityBuf's Server Architecture

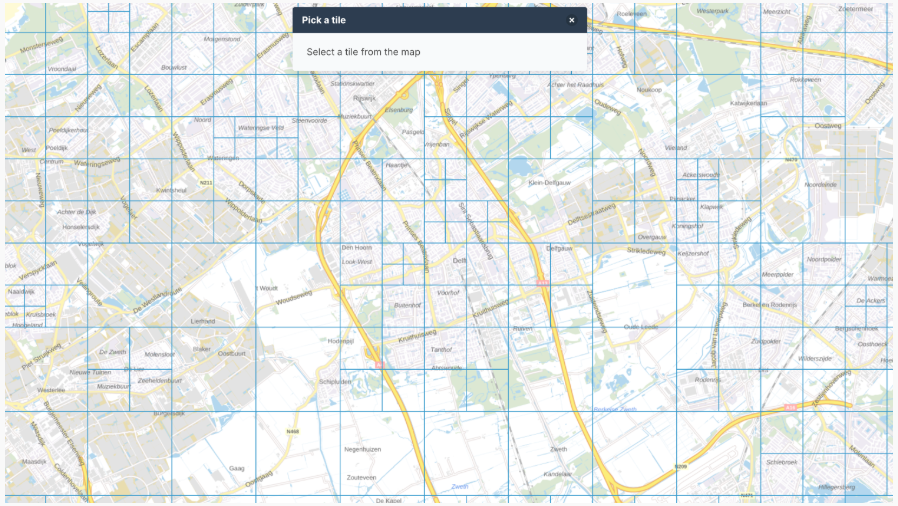
Simple, scalable and cost-effective



The Gap we fill: 3D City Models Still Behind

Current 3D Distribution

Current Issues: Tiled, Text format



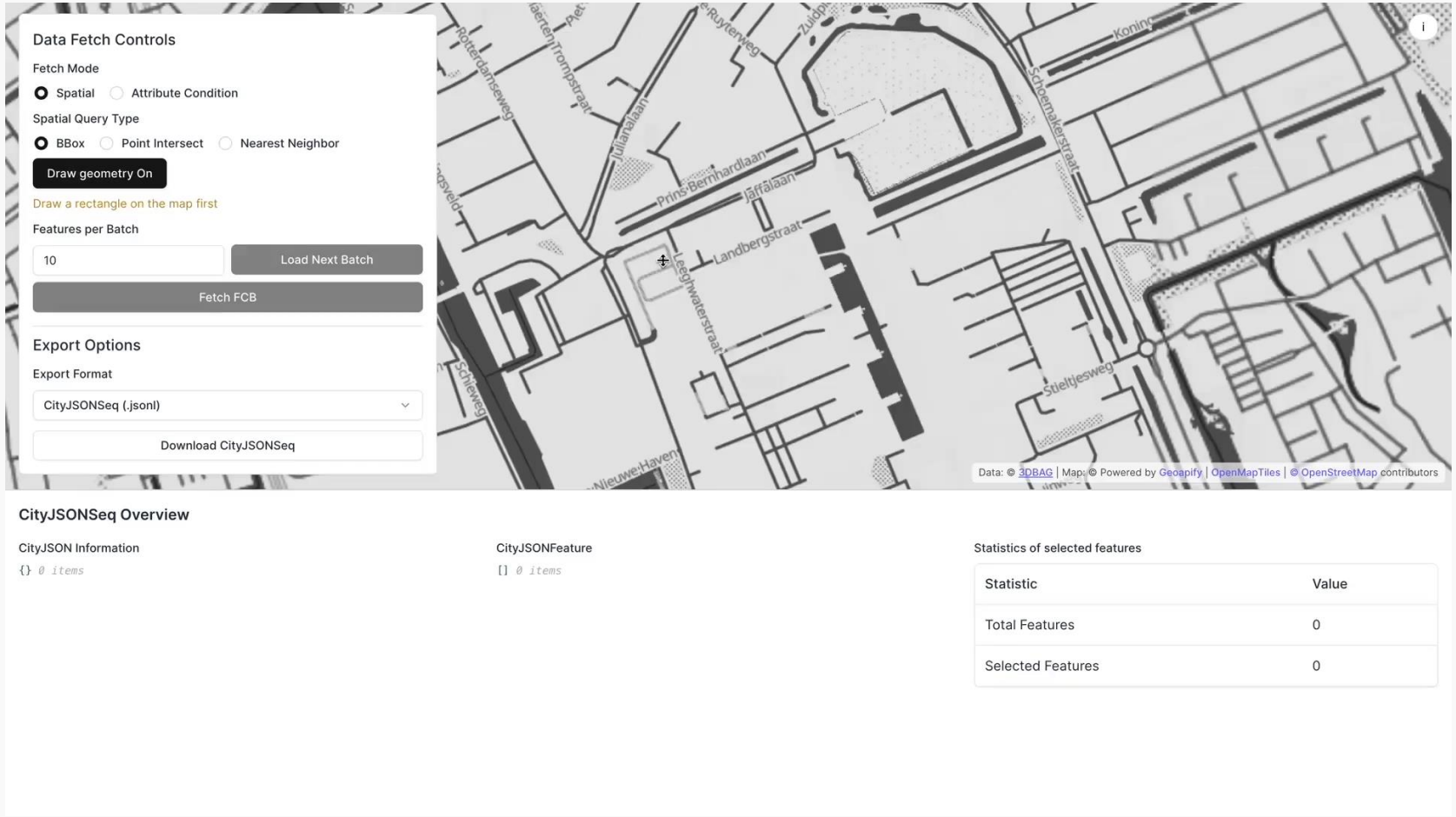
The Problem

No cloud-optimised format for 3D city models

- CityGML: Text-based, no streaming
- CityJSON(Seq): Streaming, but no spatial index
- 3DCityDB: Not scallable

FlatCityBuf: A Cloud-Optimised CityJSON Format

2-15x faster performance with cloud-native streaming



FlatBuffers: General Idea

FlatBuffers

developed by Google.

- No parsing needed (Zero-copy)
- Low memory consumption
- Strictly typed (Schema driven)

JSON

Compared with FlatBuffers, it's:

- Text-based
- Needs to parse
- More code to access data

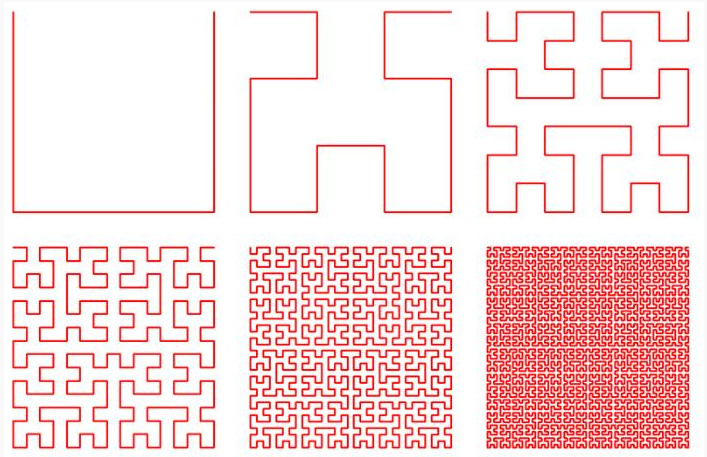
File Structure Design

The file consists of:

Spatial Indexing: Packed Hilbert R-tree

Construction

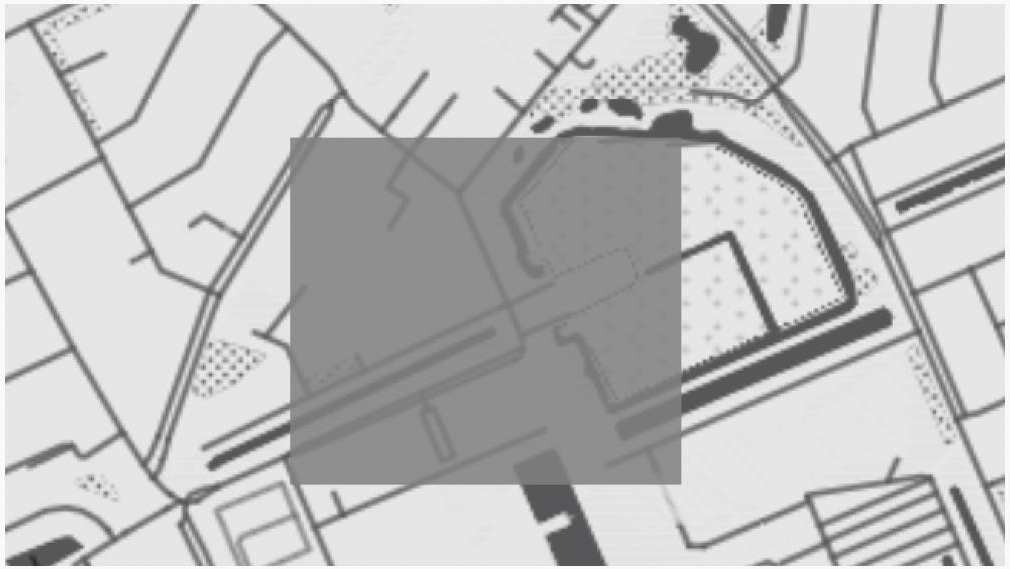
1. Calculate BBox of city features
2. Map to Hilbert curve
3. Sort by spatial locality
4. Build R-tree bottom-up
5. Pack into linear array



(m Williams, 2022)

Query Support

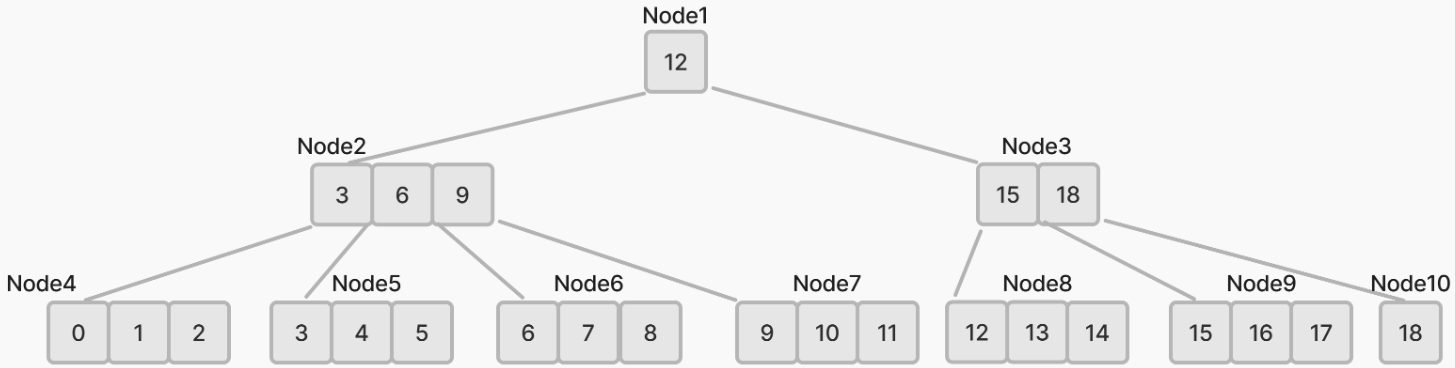
- **Bounding box queries** in O(log n)
- **Point/Nearest neighbour** search

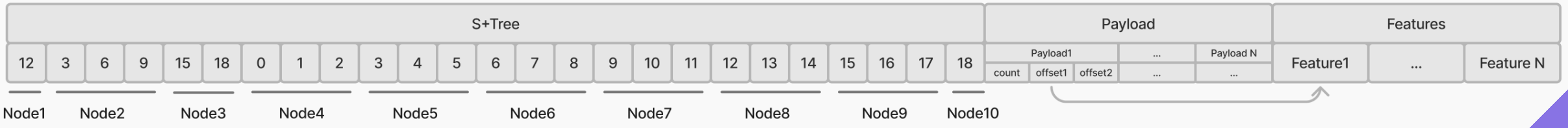


Attribute Indexing: Static B+Tree

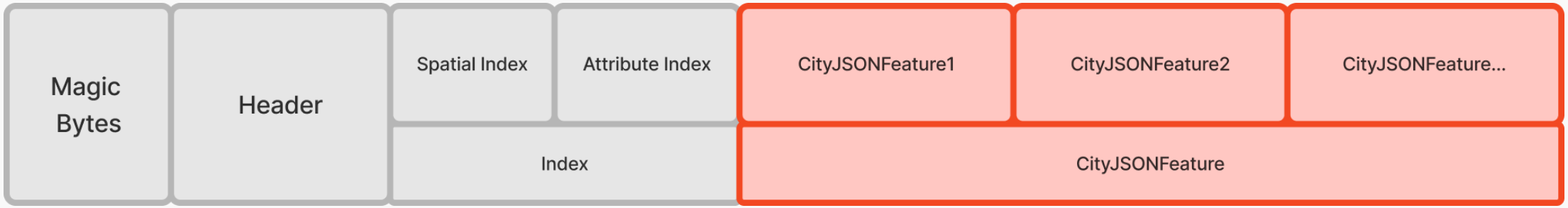
Structure

1. Sort features **2.** Build B+Tree **3.** Store keys and pointers **4.** Pack into linear array





Feature Encoding: Structure



- **Feature encoding preserves CityJSON structure with FlatBuffers efficiency**
  - **CityFeature**: city objects array, vertices, appearance data
  - **CityObject**: city object type, geometry, attributes, semantics, etc
  - **Flattened arrays**: Parallel structure for nested geometries

HTTP Range Requests: Selective Data Access

Traditional Approach

```
GET /data.json
→ Download entire file
→ Parse JSON
→ Filter features
```

Problem: Downloads whole file, even if only a few features are needed

FlatCityBuf Approach

Solution: Only fetch required bytes

Real-World Scale: Netherlands Dataset

3DBAG's 10 million buildings are encoded in

70.4 GB (single file)

Data remains accessible for efficient subsetting despite the large file size.

(Indexed all attributes; without index: 63.9GB. CityJSONSeq: 65.2GB - slightly smaller.)

Performance Results: Local environment

Read time 8.6-256.8x faster and 50-80% lower memory usage on local environemnt

	Processing Time			Memory Consumption		
Dataset	cjseq[a]	FlatCityBuf	Ratio[b]	cjseq[a]	FlatCityBuf	Ratio[b]
3DBAG	56.3 ms	6.6 ms	8.6×	23.9 MB	5.1 MB	4.7×
3DBV	3.99 s	122.5 ms	32.6×	283.8 MB	63.2 MB	4.5×
Helsinki	4.05 s	132.2 ms	30.6×	15.3 MB	5.2 MB	2.9×
Ingolstadt	37.2 ms	0.5 ms	75.8×	30.1 MB	6.9 MB	4.4×
Montréal	50.3 ms	0.6 ms	81.6×	36.3 MB	5.7 MB	6.4×
NYC	887.6 ms	42.9 ms	20.7×	20.6 MB	5.0 MB	4.1×
Rotterdam	22.2 ms	1.3 ms	17.6×	9.2 MB	4.4 MB	2.1×
Vienna	45.9 ms	1.9 ms	24.0×	14.6 MB	5.2 MB	2.8×
Zürich	1.88 s	151.9 ms	12.4×	31.3 MB	5.1 MB	6.2×
PLATEAU (Building)	861.4 ms	32.5 ms	26.5×	220.9 MB	64.4 MB	3.4×
PLATEAU (Bridge)	83.9 ms	0.3 ms	256.8×	75.0 MB	12.0 MB	6.3×
PLATEAU (Railway)	37.9 ms	2.0 ms	18.5×	19.0 MB	5.1 MB	3.8×
PLATEAU (Transport)	244.0 ms	13.3 ms	18.4×	76.7 MB	20.2 MB	3.8×
PLATEAU (Tunnels)	47.9 ms	1.9 ms	24.9×	70.6 MB	12.6 MB	5.6×
PLATEAU (Vegetation)	852.3 ms	32.9 ms	25.9×	189.8 MB	56.9 MB	3.3×

[a] CityJSONSeq
[b] Ratio = CityJSONSeq metric / FlatCityBuf metric (higher values indicate better FlatCityBuf performance)

Performance Results (HTTP): vs 3DBAG API (RESTful API with DBMS)

ID Query Performance

Feature ID	Location	FCB (ms)	API (ms)	Ratio
NL.IMBAG.Pand.0503100000032914	TU Delft BK building	935.8	2412.5	2.6×
NL.IMBAG.Pand.0363100012185598	Amsterdam Central Station	858.0	2106.7	2.5×
NL.IMBAG.Pand.0014100010938997	Groningen Station	821.7	2254.8	2.7×
NL.IMBAG.Pand.0772100000295227	Eindhoven Station	1378.7	2013.4	1.5×
NL.IMBAG.Pand.0153100000261851	Enschede Station	1070.4	2058.8	1.9×
Average		**1012.9**	**2169.2**	**2.1×**

FCB = FlatCityBuf
API = 3DBAG API
Ratio = 3DBAG API / FlatCityBuf (higher values indicate better FlatCityBuf performance)

2× faster

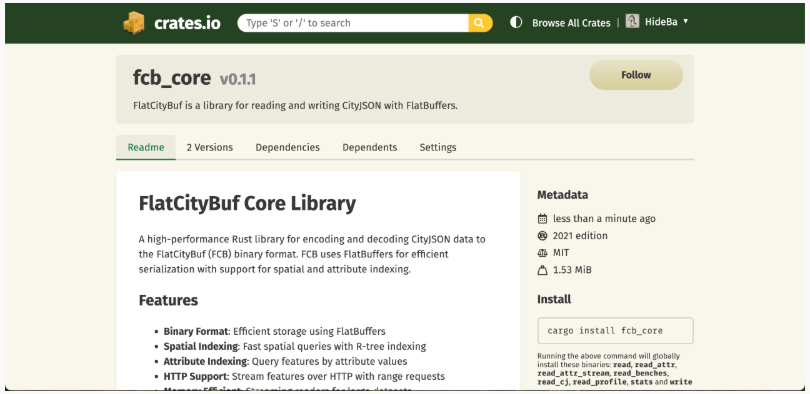
Bounding Box Query

Query Type	FlatCityBuf (ms)	3DBAG API (ms)	Ratio
Bounding box (2km × 2km)	492.6	7420.3	15.1×

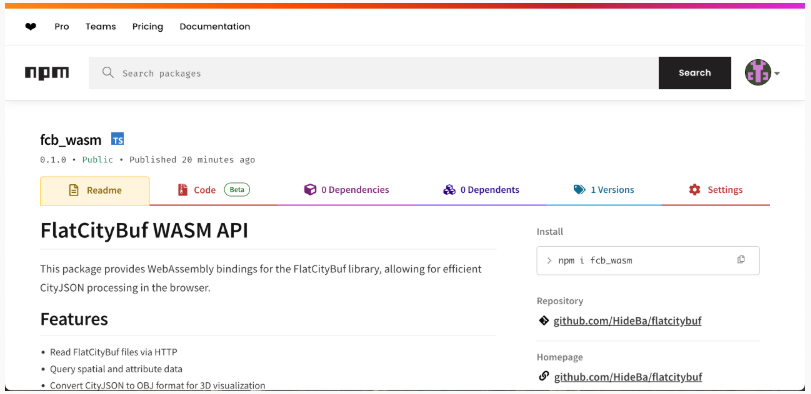
Ratio = 3DBAG API / FlatCityBuf (higher values indicate better FlatCityBuf performance)

15× faster

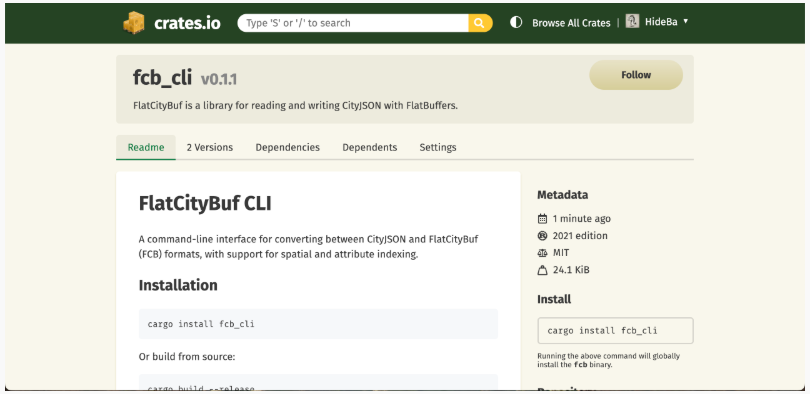
Outcomes: Software and Libraries



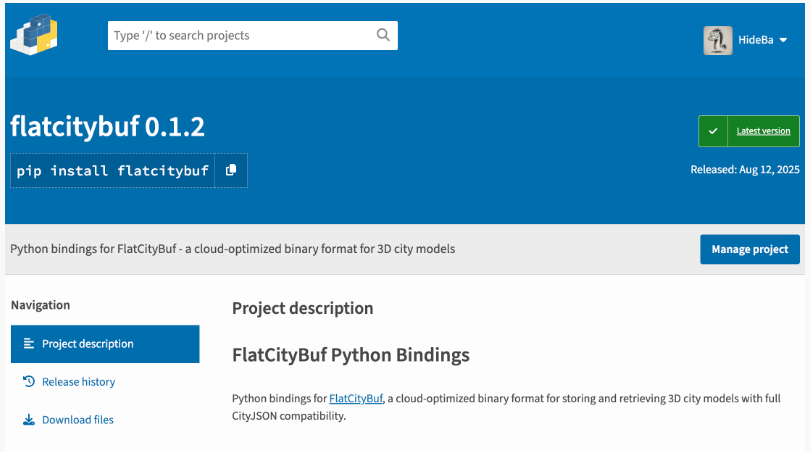
Core library (Rust) crates.io



WASM bindings for TypeScript npm



CLI tool (Rust) crates.io



Python bindings pypi

Discussion: Use Cases and Applications

Flexible Data Download

Query-based subset retrieval with multi-format export (CityJSON, OBJ, glTF)

Data Processing Applications

Optimised for I/O intensive pipelines like 3DBAG generation

Limitations & Trade-offs

Immutable format

Limited query flexibility

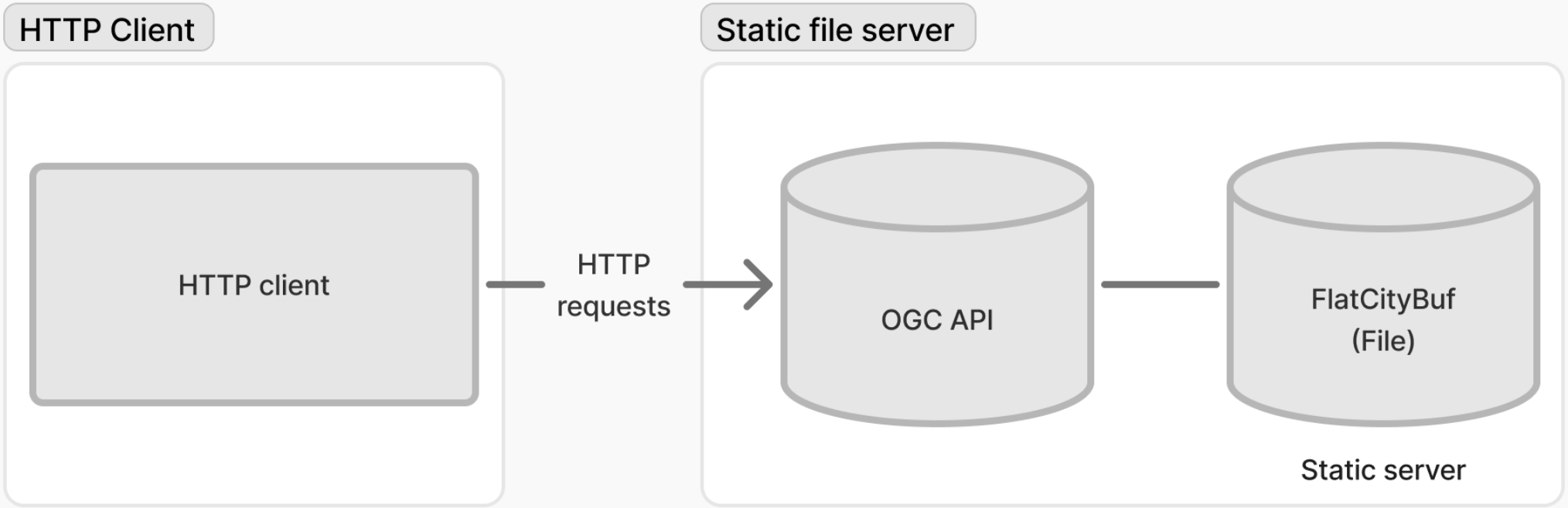
Complex client implementation (less interoperable)

Best for read-heavy, web-based applications

Experimental work: 3DBAG API with FlatCityBuf

No DBMS, just static file hosting and API

```
GET /collections/pand/items?bbox=minX, minY, maxX, maxY&limit=10
```



3DBAG API with FlatCityBuf: Key Features

Demo API is available at: https://flatcitybuf-api-264879243442.europe-west4.run.app/

Fetch features using queries such as:

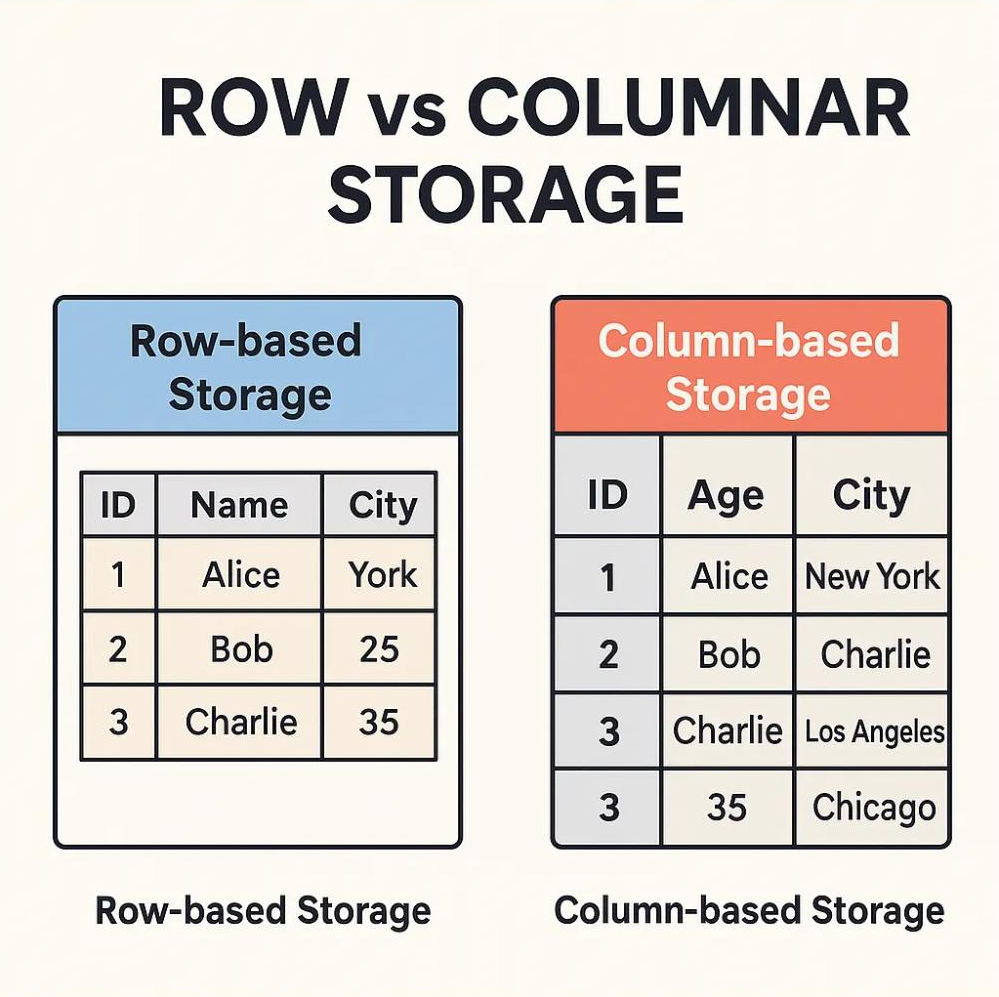
- **Bounding box query:**
  Example: `bbox=minX, minY, maxX, maxY`
- **ID query:**
  Example: `identificatie=NL.IMBAG.Pand.0153100000209948`
- **Attribute query:**
  Example: `construction_year > 1990 AND height > 100`

Available response formats:

- JSON (features and links to next page)
- CityJSON
- CityJSON TextSequences
- Wavefront OBJ

CityParquet (ongoing work)

Why Parquet for 3D City Models?



Key Features

- Columnar storage
- Compression
- Pushdown predicates

(Shetty, 2025)

Why not GeoParquet?

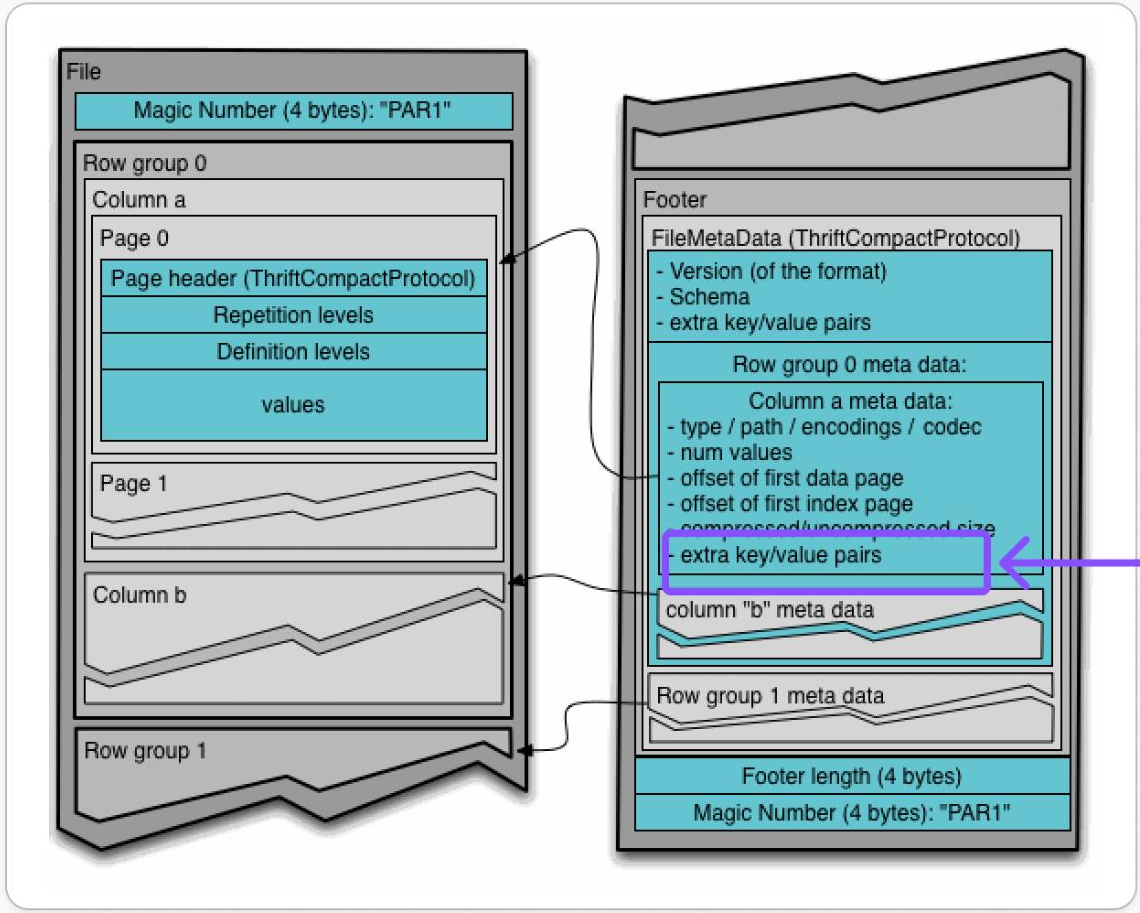
GeoParquet is for Simple Features

3D City Models cannot be represented as Simple Features

CityParquet: How to encode CityJSON into Parquet?

CityParquet: CityJSON's metadata

Encode as JSON object in the Parquet's key value metadata



- Geographical extent
- Reference system
- Version
- Transform
- Points of contact
- Attribute names

Image derived from (Parquet, 2025)

CityParquet: One record per CityObject

CityParquet encodes CityJSON's CityObject into a single table with nested columns.

```
select * from test.city_objects;
```

delft

2,231 rows returned in 135ms

	ig	# b3_nodata_radius_ahn3	# b3_rmse_lod12	# b3_puntdichtheid_ahn4	T eindgeldigheid	# b3_opp_buitenmuur	# b3_nodata_fractie_ahn4	# b3_h_dak_min	T tijdstipnietbaglv	T status	# b3_volume_lod13	# b3
1		NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	
2		0.2547723054885864	0.03385167941451073	39.73333358764648	NULL	25.71	0	3.0999999046325684	NULL	Pand in gebruik	19.593151092529297	
3		0.29476311802864075	0.034421537071466446	26.829267501831055	NULL	28.81	0	2.190000057220459	NULL	Pand in gebruik	27.841793060302734	
4		NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	
5		NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	
6		0.359510600566864	3.2317864894866943	46.19647216796875	NULL	165.03	0	3.890000104904175	NULL	Pand in gebruik	657.8263549804688	
7		0.33875787258148193	0.03871022164821625	36.720001220703125	NULL	37.41	0	2.549999952316284	NULL	Pand in gebruik	152.0753631591797	
8		NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	
9		NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	
10		0.32769688963890076	0.3889843821525574	26.36103057861328	NULL	192.86	0	6.389999866485596	NULL	Pand in gebruik	883.1010131835938	
11		NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	
12		0.3302719295024872	2.901437520980835	26.25742530822754	NULL	95.47	0.025723472237586975	3.4100000858306885	NULL	Pand in gebruik	813.2996215820312	
13		0.39010798931121826	2.6202948093414307	40.551456451416016	NULL	156.65	0	2.5399999618530273	NULL	Pand in gebruik	699.4473266601562	
14		NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	

Status & Next

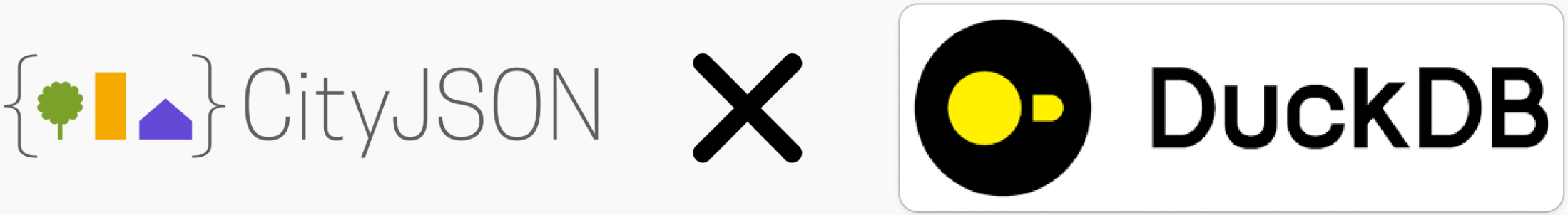
- Done: Schema, Encoder, Decoder, CLI, Metadata, Tests
- Enhanced geometry encoding
- Benchmarking and file size comparison

Opinion wanted

- How do you use GeoParquet? If CityParquet is released, would you use it?
- What CityParquet schema seems to be useful for you?

DuckDB extension for CityJSON (ongoing work)

DuckDB is your best friend to read/write Cloud-Optimised Geospatial Formats.
Why not use it for CityJSON?



Thank you

Questions?

Resources:

- Demo: fcb-web-prototype.netlify.app
- Code:

  https://github.com/cityjson/flatcitybuf

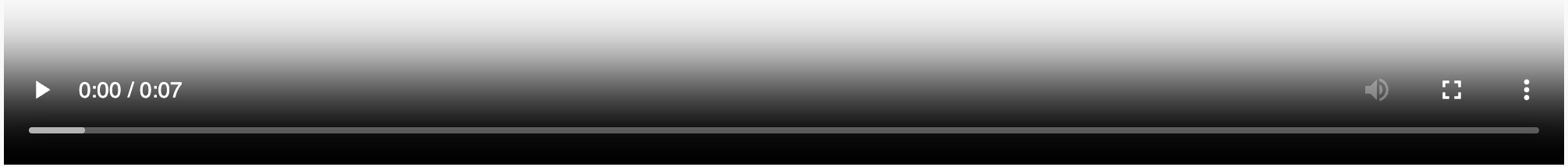
Contact:

h.b.baba@tudelft.nl

Appendix

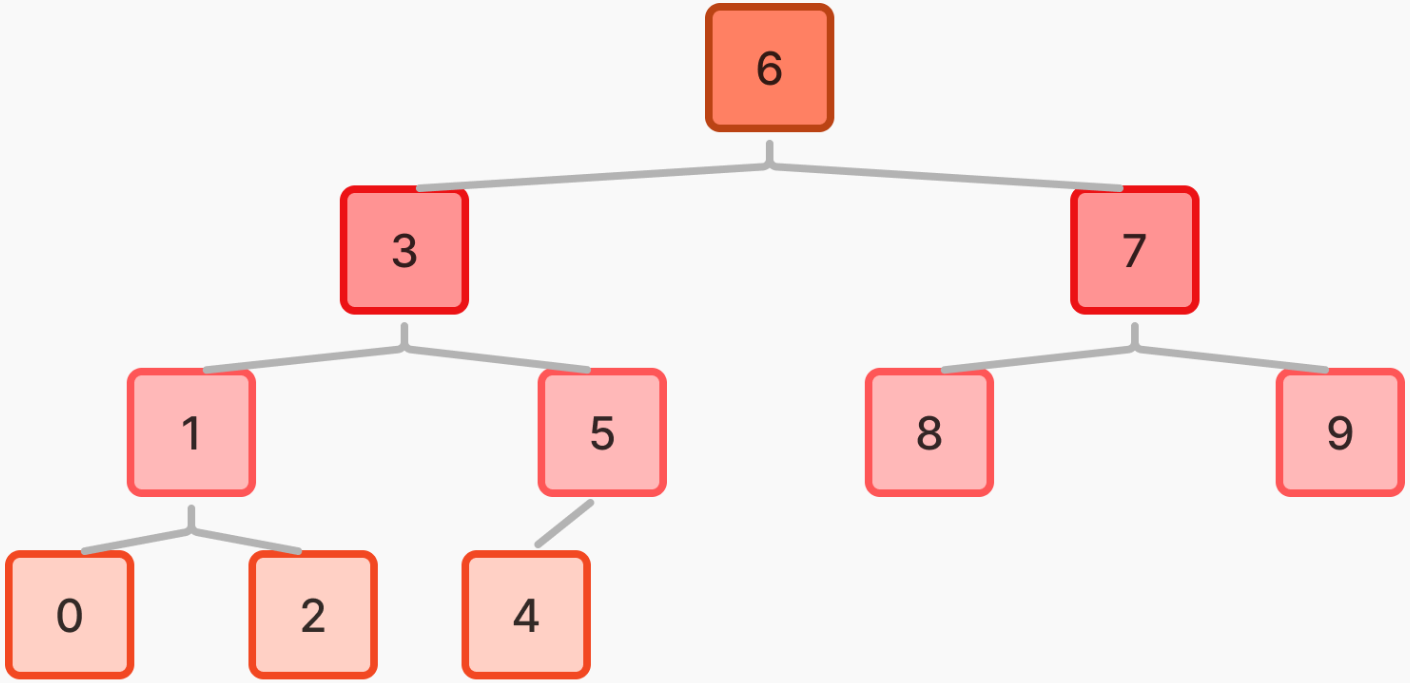
Appendix (related work)

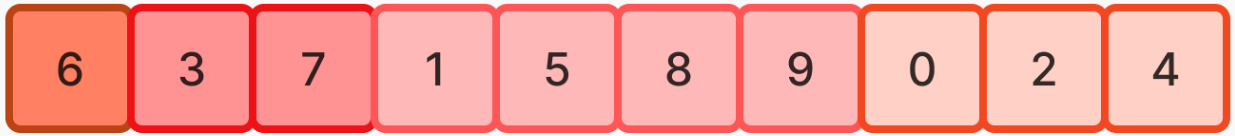
Example of Cloud-Optimised Geospatial Formats (2D): FlatGeobuf



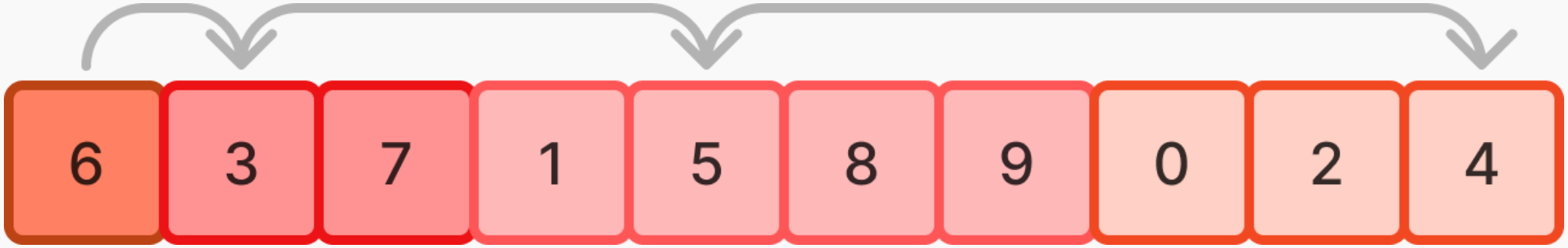
Appendix (Theoretical)

Appendix (Theoretical): Eytzinger Layout (1/2)





Appendix (Theoretical): Eytzinger Layout (2/2)



Appendix (Theoretical): Column-oriented and Row-oriented storage

Example data

id	city	country
1	Tokyo	Japan
2	London	United Kingdom
3	Amsterdam	Netherlands

Row-oriented storage

```
1, Tokyo, Japan , 2, London, UK, 3, Amsterdam, Netherlands
```

Column-oriented storage

```
1, 2, 3, Tokyo, London, Amsterdam, Japan, UK, Netherlands
```

Appendix (Theoretical): Endianness

Little Endian

- Least significant byte is stored first
- Example: 0x12345678
- Stored as: 0x78 0x56 0x34 0x12
- e.g. (31 December 2050) in calendar date format

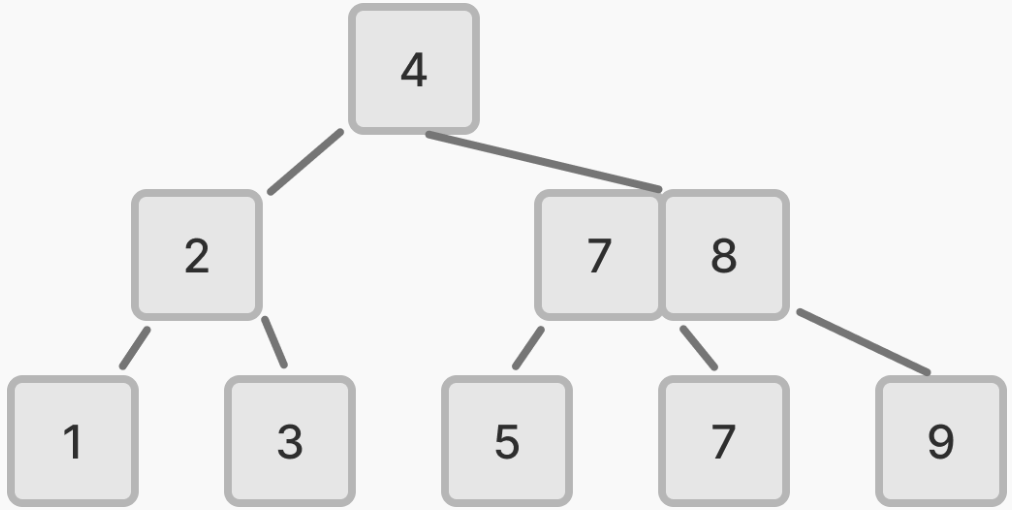
Big Endian

- Most significant byte is stored first
- Example: 0x12345678
- Stored as: 0x12 0x34 0x56 0x78
- e.g. (2025-12-31) in calendar date format

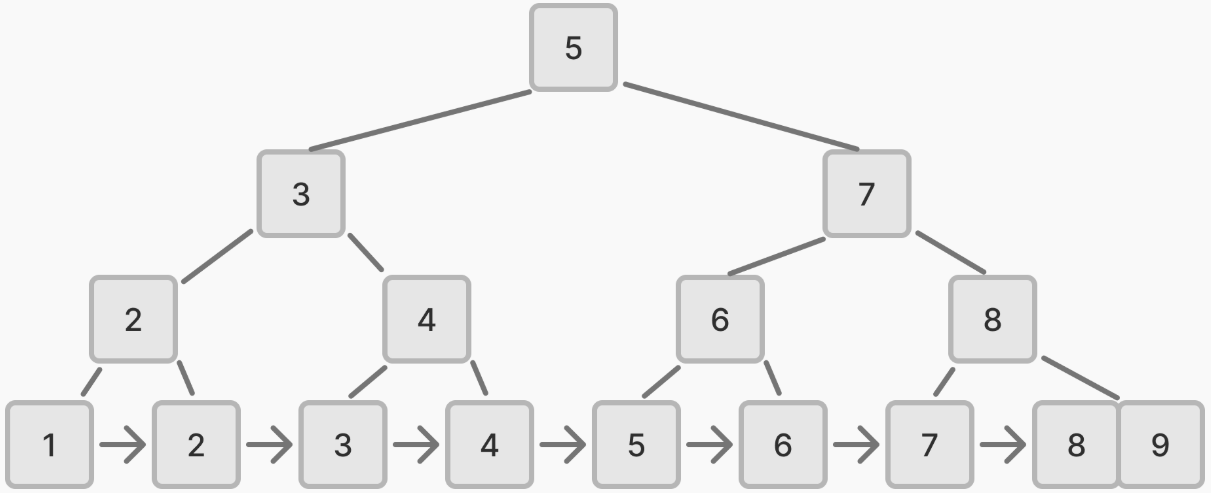
B-Tree/B+Tree

B-Tree and its variants, B+Tree are self-balancing binary search trees. With block size B, trees achieve log_B(n) instead of log_2(n) memory accesses.

B-Tree



B+Tree



Appendix (Theoretical): WebAssembly

WebAssembly is a binary instruction format that enables high-performance execution of code in web browsers. It allows languages like C, C++, and Rust to run at near-native speed on the web.

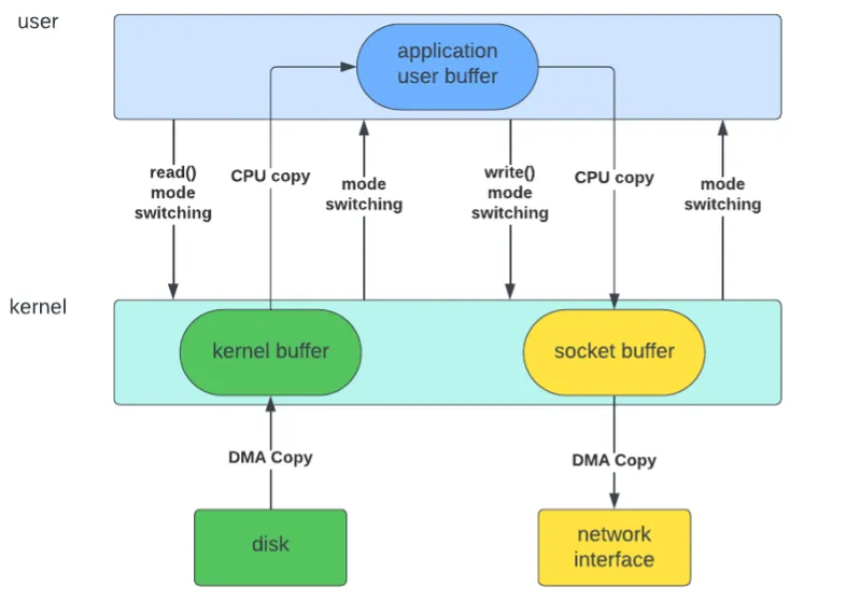
Appendix (Theoretical): FlatBuffers

```
vtable (AnnotatedBinary.Bar):
+0x00A0 | 08 00       | uint16_t   | 0x0008 (8)    | size of this vtable
+0x00A2 | 13 00       | uint16_t   | 0x0013 (19)   | size of referring table
+0x00A4 | 08 00       | VOffset16  | 0x0008 (8)    | offset to field ‘a‘ (id: 0)
+0x00A6 | 04 00       | VOffset16  | 0x0004 (4)    | offset to field ‘b‘ (id: 1)
```

FlatBuffers: Zero-copy

Multiple Copies of the Same Data

When data is processed on a machine, it is often copied multiple times.



Zero-copy

Zero-copy avoids data copying between memory locations, reducing I/O overhead. Formats like FlatBuffers enable direct access to serialised data without deserialisation.

Though the term "zero" is often used, it's not necessarily zero. It implies the data is copied much less than other approaches.

Appendix: Scope of the Research

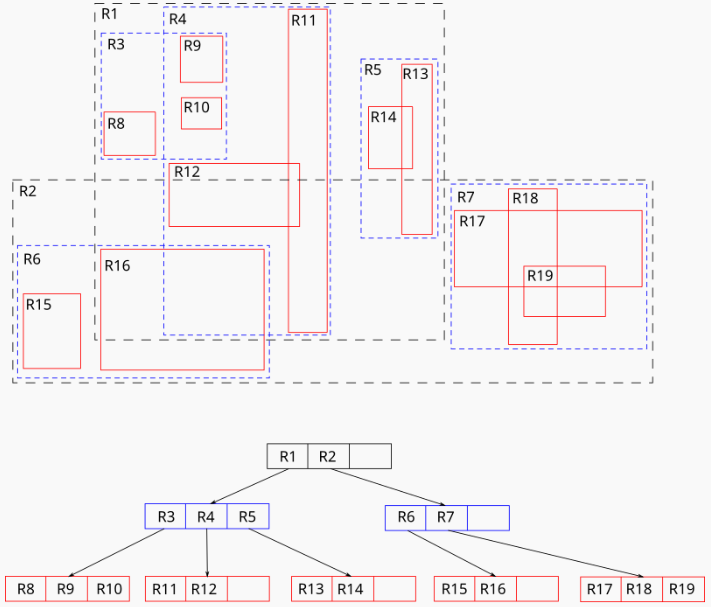
- **In scope:**
  - FlatCityBuf format design and implementation (Rust)
  - Spatial and attribute indexing
  - HTTP Range Request data retrieval
  - Performance evaluation vs CityJSONSeq
  - Web-based demo
- **Out of scope:**
  - Implementing libraries in other programming languages
  - Exploring alternative serialisation frameworks like Parquet or Protocol Buffers
  - Optimising for write operations

Appendix: Methodology

Spatial Indexing (1/4): General Idea

R-tree

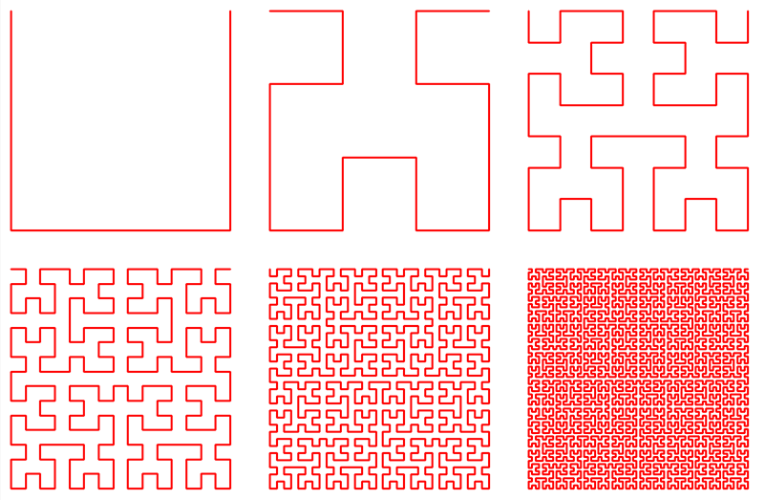
R-tree is a spatial index structure for 2D and 3D data.



(*Wikipedia, 2025*)

Space-filling Curves

Space-filling curves such as Hilbert curve map multi-dimensional data to one dimension while preserving spatial locality.



(*m Williams, 2022*)

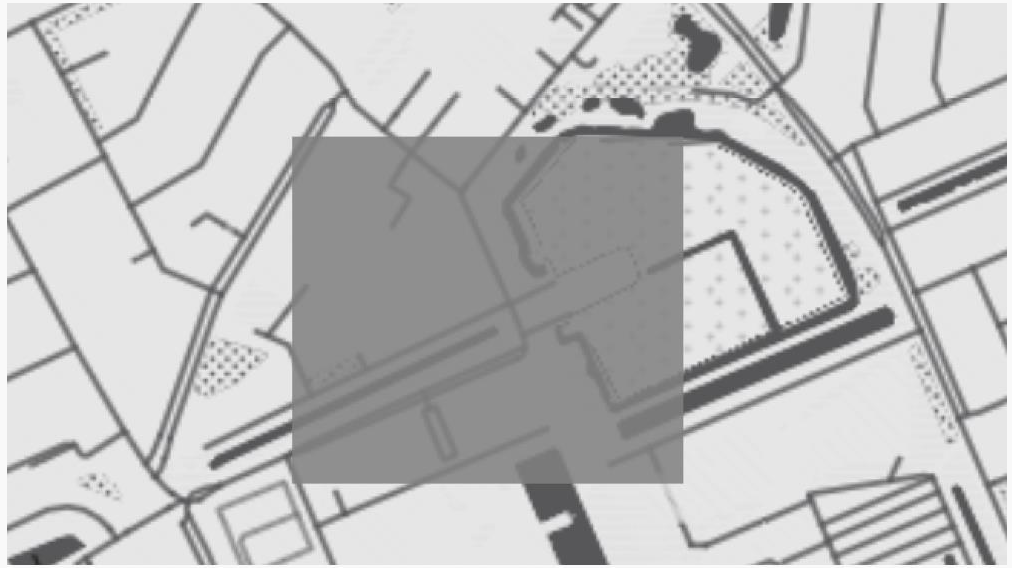
Spatial Indexing (3/4): Construction Steps

1. **Calculate bounding boxes**: Compute BBox for each feature
2. **Hilbert curve mapping**: Map BBox centres to Hilbert curve positions
3. **Sort by Hilbert value**: Order features to maintain spatial locality
4. **Build R-tree bottom-up**: Group features into leaf nodes, build parent nodes
5. **Pack into linear array**: Serialise tree into contiguous memory layout

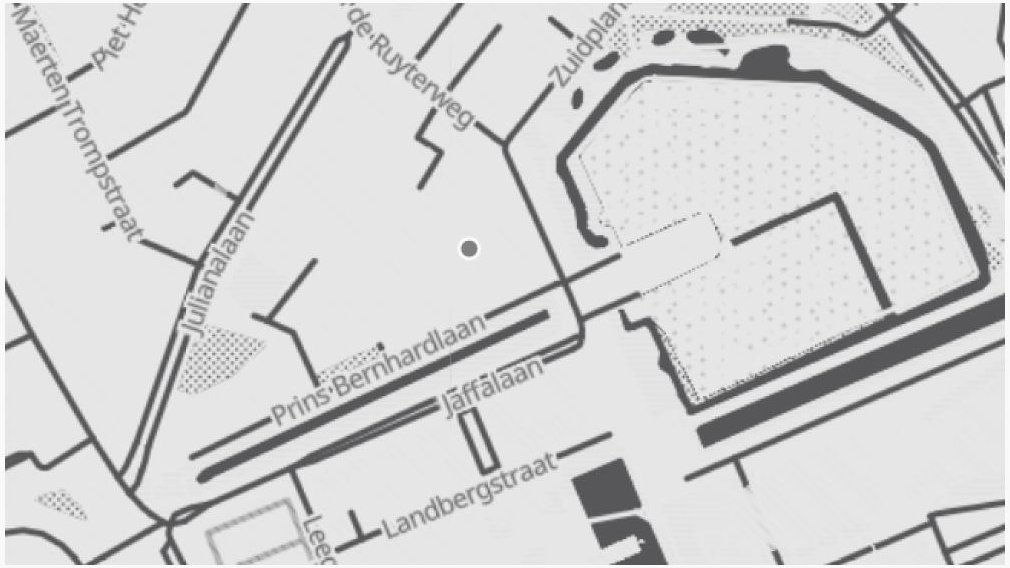
Spatial Indexing (4/4): Supported Queries

Spatial indexing in a streaming manner

Bounding Box

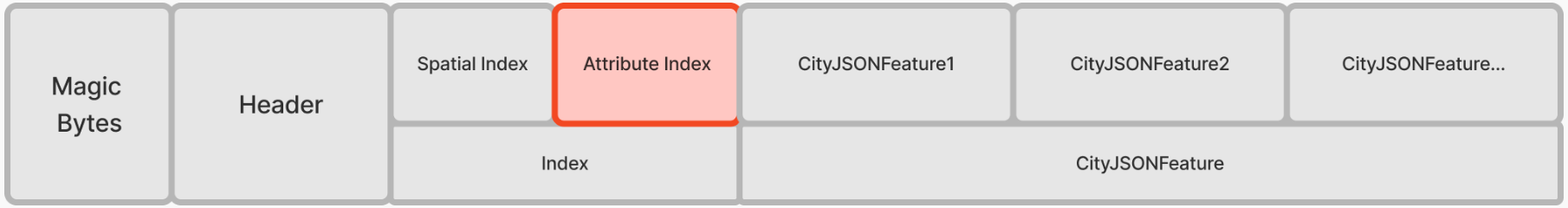


Point/Nearest Neighbour

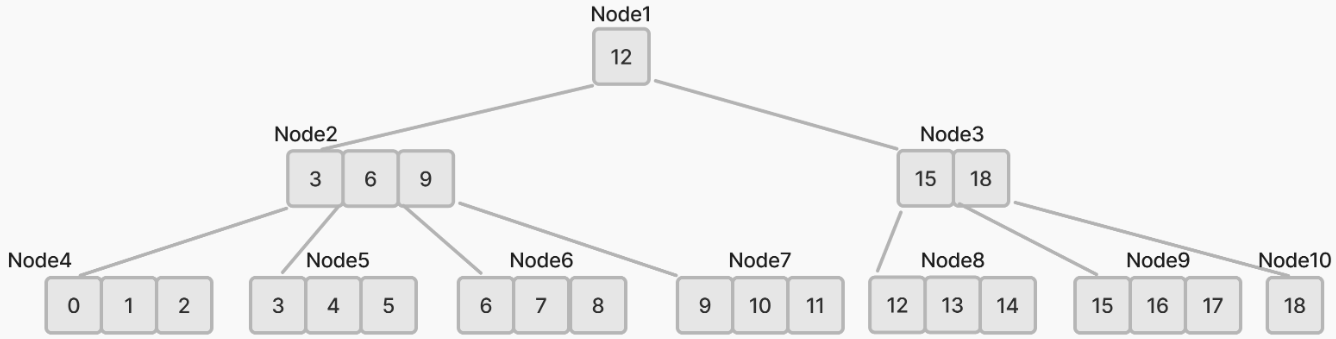


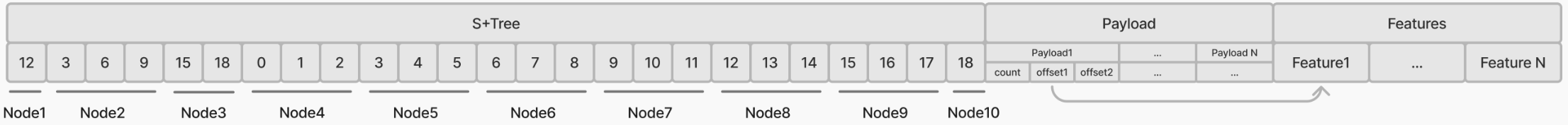
Attribute Indexing (1/3): Structure

Attribute Index in the file



Static B+Tree





Attribute Indexing (2/3): Construction Steps

1. **Sort features by attribute**: Order features by the indexed attribute value (e.g. `1.apple`, `2.banana`, `3.cherry`)
2. **Build B+Tree bottom-up**: Create leaf nodes with sorted features, build internal nodes
3. **Store keys and pointers**: Internal nodes store keys and child pointers, leaves store actual data
4. **Pack into linear array**: Serialise tree structure for efficient disk storage
5. **Create index metadata**: Store root offset, node size, and branching factor

Attribute Indexing (3/3): Supported Queries

Attribute indexing is also in a streaming manner!

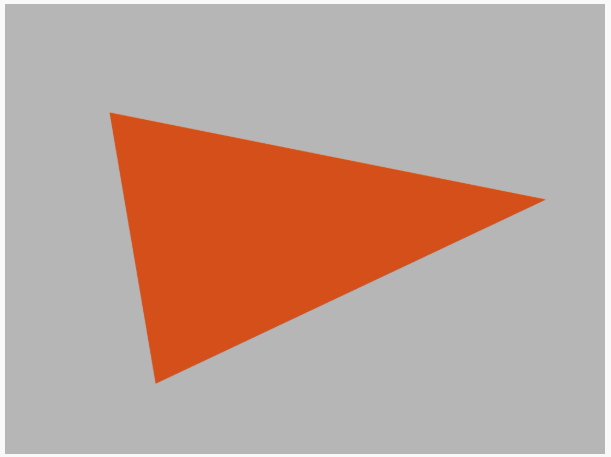
- **Exact Match Queries**
  - Find features with specific attribute values
  - Example: `city = "Tokyo"`
- **Range Queries**
  - Find features within attribute value ranges (`<` , `<=` , `>` , `>=`)
  - Example: `construction_year BETWEEN 1990 AND 2000`
- **Logical Combinations**
  - Find features that satisfy multiple conditions
  - Example: `(construction_year > 1990) AND (height > 100)`

Feature Encoding (2/3): Geometry Encoding

FlatBuffers does not support nested arrays. We use flattened arrays to represent nested geometries.

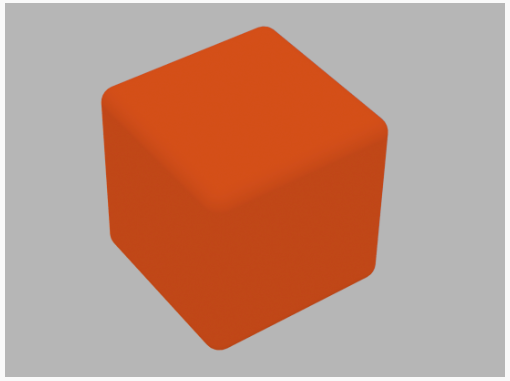
Example (Triangle)

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



Example (Cube)

```
boundaries : [0 , 1 , 2 , 3 , 0 , 3 , 7 , 4 ...]
strings : [4 , 4 , 4 , 4 , 4 , 4]
surfaces : [1 , 1 , 1 , 1 , 1 , 1]
shells : [6]
solids : [1]
```

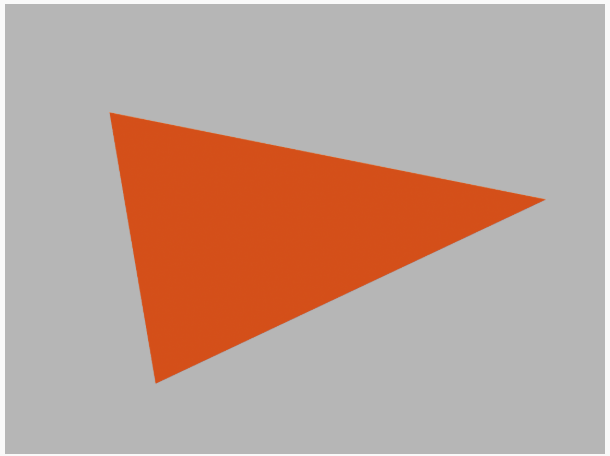


2.6 Feature Encoding (2/3): Geometry Encoding

FlatBuffers does not support nested arrays. We use flattened arrays to represent nested geometries.

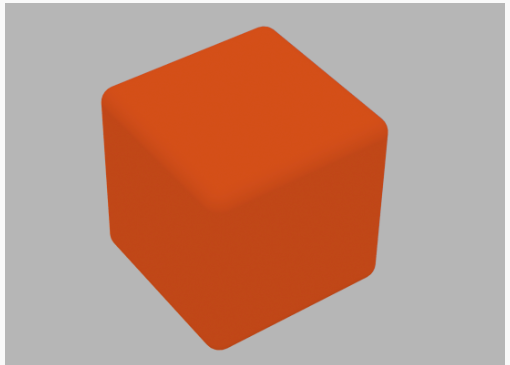
Example (Triangle)

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



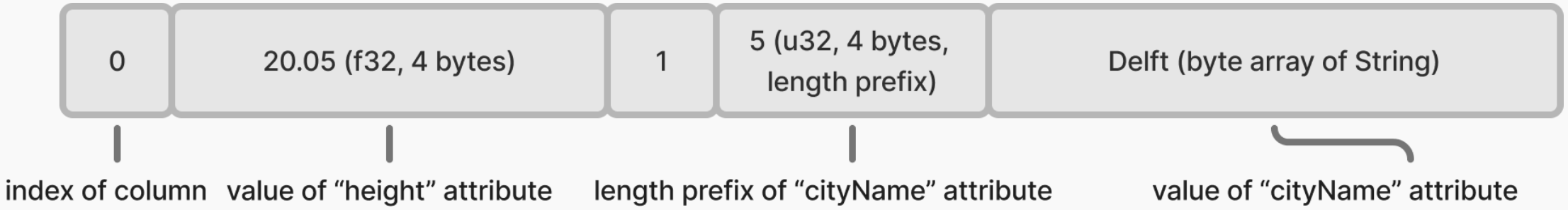
Example (Cube)

```
boundaries : [0 , 1 , 2 , 3 , 0 , 3 , 7 , 4 ...]
strings : [4 , 4 , 4 , 4 , 4 , 4]
surfaces : [1 , 1 , 1 , 1 , 1 , 1]
shells : [6]
solids : [1]
```



Feature Encoding (3/3): Attribute Encoding

Attributes are encoded with their own binary representation. (in little endian)



Appendix A: Results & Evaluation

Appendix A: File Size Comparison (Level of Detail)

Dataset	FlatCityBuf[a]	CityJSONSeq[b]	Compression	Vertices
TUD BK All	139.75 kB	189.01 kB	26.08%	4549
TUD BK LoD0	12.77 kB	20.72 kB	38.11%	785
TUD BK LoD1.2	37.45 kB	49.40 kB	24.23%	1350
TUD BK LoD1.3	44.66 kB	59.25 kB	24.67%	1600
TUD BK LoD2.2	62.02 kB	82.74 kB	25.07%	2168

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

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

Appendix A: File Size Comparison (Attribute Quantity)

Data

```
{
   "type ":  "Building",
   "geometry ":  [  .  .  .  ] ,
   "attributes ":  {
   "attr_1":  "value_1",
   "attr_2":  "value_2",
   "attr_3":  "value_3",
   "attr_4":  "value_4",
   "attr_5":  "value_5",
   ...
   "attr_n":  " value_n"
   }
}
```

Result

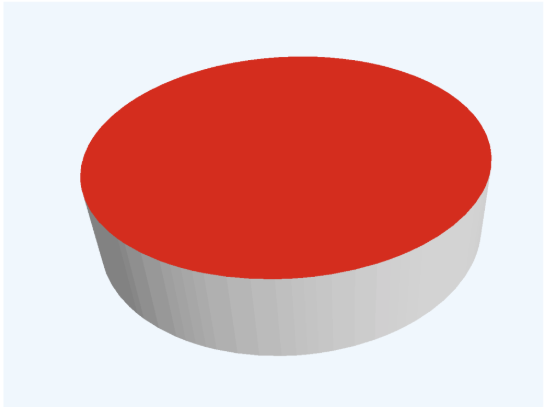
Dataset	FlatCityBuf[a]	CityJSONSeq[b]	Compression
10 attributes (int)	580 B	611 B	5.07%
100 attributes (int)	1.62 kB	2.44 kB	33.65%
1000 attributes (int)	12.17 kB	21.78 kB	44.13%
10 attributes (string)	580 B	611 B	5.07%
100 attributes (string)	1.62 kB	2.44 kB	33.65%
1000 attributes (string)	12.17 kB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

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

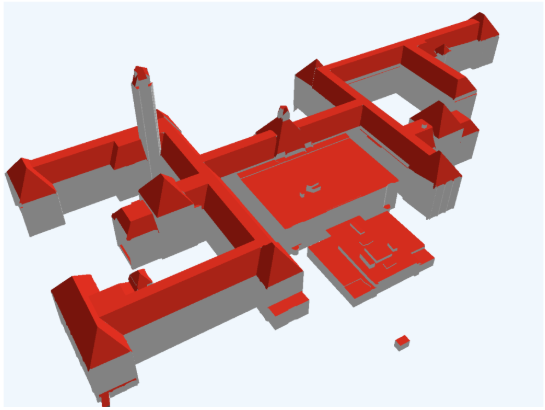
Average feature size in bytes in CityJSONSeq: $\frac{\text{Total CityJSONSeq size}}{\text{Number of features}}$

Appendix A: File Size Comparison (Geometric Complexity)

Data



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



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

Result

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

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

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

Appendix A: File Size Comparison (Coordinate Scale)

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

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

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

--

Appendix A: Read Performance (vs CityJSONSeq)

Appendix A: Read Performance (vs CBOR)

	Processing Time			Memory Consumption		
Dataset	**CBOR**	**FlatCityBuf**	**Ratio[a]**	**CBOR**	**FlatCityBuf**	**Ratio[a]**
3DBAG	74.0 ms	6.6 ms	11.2×	194.1 MB	5.1 MB	38.1×
3DBV	6.34 s	122.5 ms	51.8×	4.96 GB	63.2 MB	80.3×
Helsinki	7.97 s	132.2 ms	60.3×	5.14 GB	5.2 MB	1011.2×
Ingolstadt	46.9 ms	0.5 ms	95.7×	187.5 MB	6.9 MB	27.3×
Montréal	58.4 ms	0.6 ms	94.7×	257.1 MB	5.7 MB	45.3×
NYC	1.33 s	42.9 ms	31.0×	1.65 GB	5.0 MB	337.4×
Rotterdam	30.8 ms	1.3 ms	24.4×	140.0 MB	4.4 MB	31.9×
Vienna	58.8 ms	1.9 ms	30.7×	179.8 MB	5.2 MB	34.7×
Zürich	3.53 s	151.9 ms	23.3×	4.51 GB	5.1 MB	913.2×
PLATEAU (Building)	1.06 s	32.5 ms	32.4×	1.83 GB	64.4 MB	28.4×
PLATEAU (Bridge)	63.6 ms	0.3 ms	194.7×	305.4 MB	12.0 MB	25.6×
PLATEAU (Railway)	46.3 ms	2.0 ms	22.7×	141.0 MB	5.1 MB	27.9×
PLATEAU (Transport)	316.1 ms	13.3 ms	23.8×	614.5 MB	20.2 MB	30.5×
PLATEAU (Tunnels)	147.7 ms	1.9 ms	76.7×	400.2 MB	12.6 MB	31.8×
PLATEAU (Vegetation)	997.8 ms	32.9 ms	30.3×	1.97 GB	56.9 MB	35.3×

[a] Ratio = CBOR metric / FlatCityBuf metric (higher values indicate better FlatCityBuf performance)