

Zarr-vectors

Scalable data formats for points,
lines, graphs, meshes, & more

Specification by Forrest Collman

Spring 2026

Allen Institute for Brain Sciences

Multiscale X-Ray Imaging Group, UCL Hawkes Institute



Summary of Zarr-vectors (v0.2) approach

Chunked data formats are necessary for handling and processing of large data volumes. Non-array ('vector') based formats are complex and have been stored as large single files. These structures are often derivatives of array data held as OME-Zarr, and data volumes are growing rapidly as recent datasets are processed more efficiently.

Zarr-vectors (ZVR) is a draft specification for vector-based data. In particular, for the storage, handling and blocking of points, lines, spatial paths (streamlines), timepoint paths, graph networks, and meshes.

This specification is deliberately broad to include the wide range of communities that use complex or non-array based data.



Background

Existing chunked or
hierarchical data formats

Background

Current OME-Zarr v3 Specification

Zarr is a **chunked-based format for large numerical arrays** across multiple resolution levels (OME-). OME-Zarr v3 specification introducing sharding (via Zarr v3) to reduce overall file count and improve I/O reads and compression ratios.

Array shapes (5D): [D1, D2, X, Y, Z] often [time, channel, x,y,z]

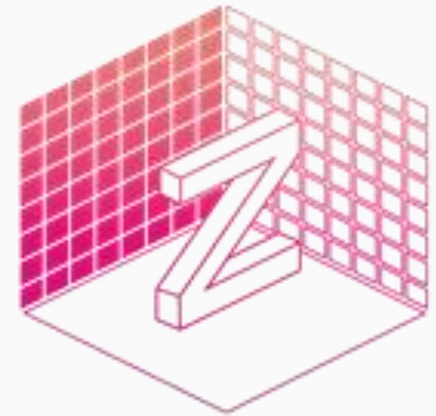
Compression: lossless (TYPE)

Metadata: multiscales file (TYPE, FIELDS)

Pyramid type: Octree (bin 2,2,2)

Array data can only be coarsened by binning – and reduced chunkwise.

A vector based approach



Zarr

OME-Zarr v0.5 (Zarr v3)

Directory structure

```
image.zarr/
├─ zarr.json          # group node; attributes.ome.multiscales defines everything
├─ 0/                 # resolution level 0 – a single Zarr array
│   ├─ zarr.json      # array metadata (shape, chunks, dtype, codecs, sharding)
│   └─ c/0/0/0/0      # chunk files (key encoding configurable)
├─ 1/                 # resolution level 1 – another single Zarr array
│   ├─ zarr.json
│   └─ ...
├─ labels/            # optional: segmentation masks
│   ├─ zarr.json      # lists label names
│   └─ segmentation/
│       ├─ zarr.json  # multiscales again, integer arrays
│       ├─ 0/
│       └─ 1/
```

Background

NIfTI-Zarr data specification

NIfTI-Zarr is a specification for including registration information into OME-Zarr datasets by **adding a standard NIfTI header** to the directory root.

Github repo: <https://github.com/neurosciences/nifti-zarr>

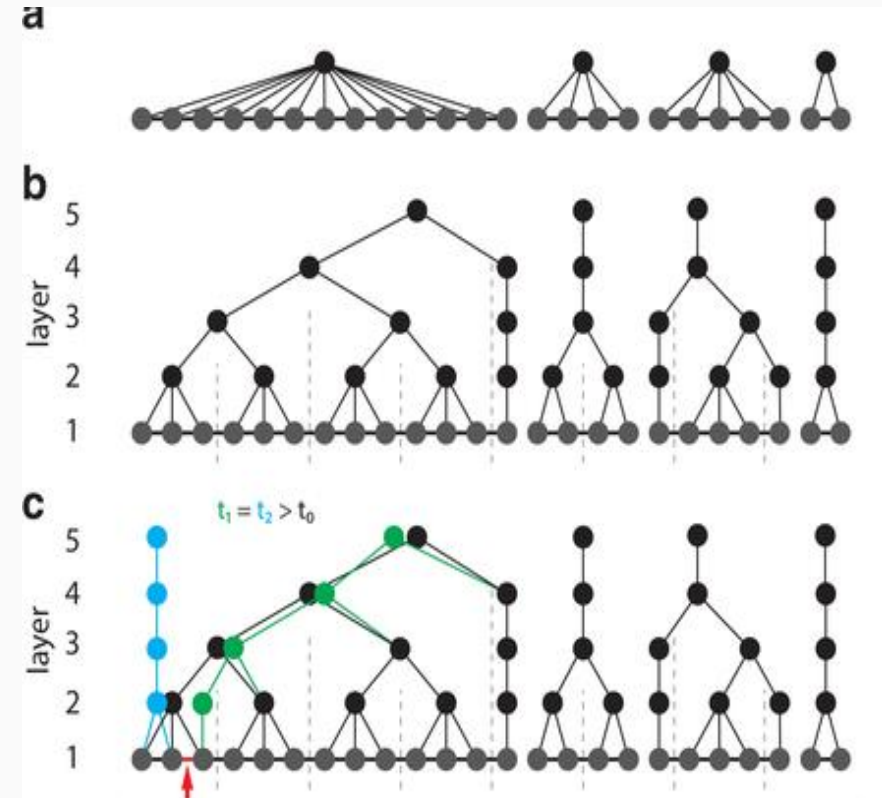
Background

ChunkGraph data structure

Produced by the Allen Institute for data efficient proof-reading of axonal segmentations of the FlyWire drosophila connectome

(DOI: 10.1038/s41592-021-01330-0, Fig.3A-C)

Structure: **Progressive coarsening of graphs** by capping number of nodes (i.e. one) allowed in a given chunk. Enforces an even spatial distribution of nodes at higher levels (may suppress relatively more complex regions but balances coverage). Uses much smaller chunk sizes than underlying zarr image dataset.



Background

Neuroglancer Precomputed

A data structure for the fast and scalable representation of some types of annotation data in neuroglancer. It is really **three separate sub-formats** under one directory, each designed for a different data type (volumes, meshes, and skeletons).

Data Type	Approach
Volume	Identical to OME-Zarr
Mesh	Per-object surface representations tied to segmentation labels. Each segment ID gets a manifest listing its mesh fragments, and each fragment is a binary vertex positions (xyz) followed by triangle indices (triplets). Uses an octree for resolution levels.
Skeleton	Per-object graph structures; each segment ID gets a single binary file with vertex count, vertex positions (xyz), edge count, edges (src,trg), then any per-vertex attributes (no edge weights). Not changed at resolution levels, just exists as one file.

Background

Neuroglancer Precomputed structure

```
volume.precomputed/
├─ info                # single JSON file describing everything
├─ 4_4_40/             # scale level (named by resolution e.g. 4nm × 4nm × 40nm)
│   └─ 0-512_0-512_0-256 # chunk files (named by voxel range)
├─ 8_8_40/             # coarser scale
│   └─ ...
├─ mesh/               # optional: surface meshes for segmentation objects
│   └─ info            # mesh-specific metadata
│   └─ <segment-id>:0  # per-object manifest (JSON listing fragments)
│       └─ <fragment-file> # binary mesh data (vertices + triangles)
└─ skeletons/          # optional: skeleton representations
    └─ info            # skeleton-specific metadata
        └─ <segment-id> # binary skeleton data per object
```

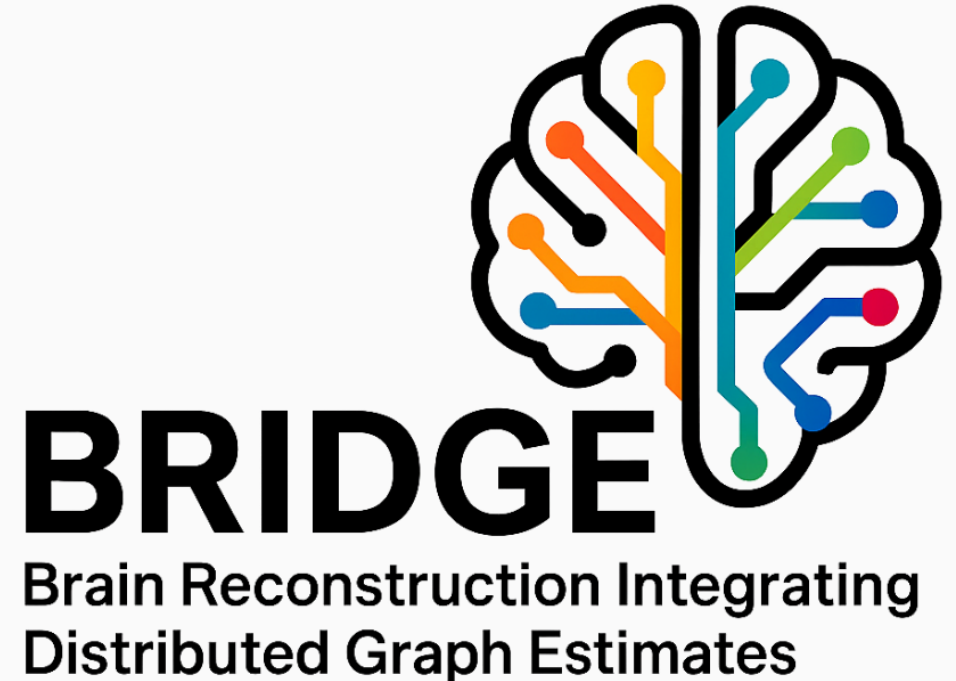
Background

BRIDGE graphs data structure

A chunked approach to hierarchical linked paths across **chunked distributed graph networks**. Aimed at representing paths extracted from distributed graphs as alternative to tractography.

Designed for building synchrotron streamlines as a method rather than as a widespread data storage formatting. Uses the same chunk size as underlying OME-Zarr image data.

Layers: Uses simplified rather than coarsened representations of paths, binning chunk regions to mirror OME-Zarr.



Background Input Data Structures

Existing structures for
holding various elements

Nodes, points, point clouds

Existing node dataset formats (csv, json, binary, ...). Common binary approach is with LAS: a **binary single-file with fixed header** (CRS, point count, scale/offset, bounding box), then flat array of point records. Each record is a fixed-width struct with attributes (non-ragged).

file.las

```
|— Header (375 bytes)      # version, point count, scale, offset, CRS,
    bbox
|— VLRs                   # variable length records (CRS detail, extras)
└— Point Records         # flat array, fixed stride per point
    |— point 0: [X,Y,Z, intensity, return, classification, GPS_time, R,G,B]
    |— point 1: [X,Y,Z, intensity, return, classification, GPS_time, R,G,B]
    └— ...
```

Streamlines

Commonly held as **lists of points** (.trk) or sometimes as linked arrays (dipy processing). TRK is a binary single-file with 1000-byte header (NIfTI plus scalar/property counts), then interleaved streamlines. Each streamline is a point count integer followed by that many XYZ floats (and with inline scalars if declared).

file.trk

```
|— Header (1000 bytes)          # vox_to_ras, voxel_size, dim, n_scalars,
  n_properties
└— Streamline Data              # sequential, interleaved
    |— streamline 0: [n_points, (x,y,z,s0,s1)×n, prop0, prop1]
    |— streamline 1: [n_points, (x,y,z,s0,s1)×n, prop0, prop1]
    └— ...
```

Streamlines continued: The TRX streamline format

A slightly **more comprehensive format** compared to .trk built for grouped and tract-based analysis approaches.

Allows more complex point and streamline attribute data to be accessed effectively.

It is still single resolution.

```
file.trx/
├─ header.json          # VOXEL_TO_RASMM, DIMENSIONS, NB_STREAMLINES,
  NB_VERTICES
├─ positions.3.float32  # NB_VERTICES × 3, all streamline points concatenated
├─ offsets.uint64       # NB_STREAMLINES: start index of each streamline
├─ dpv/                # data per vertex
│   └─ fa.1.float32     # NB_VERTICES × 1
│   └─ color.3.uint8    # NB_VERTICES × 3
├─ dps/                # data per streamline
│   └─ algo.1.uint8     # NB_STREAMLINES × 1
│   └─ clusters_QB.1.uint16 # NB_STREAMLINES × 1
├─ groups/
│   └─ AF_L.uint32      # indices of streamlines in arcuate fasciculus left
│   └─ CST_R.uint32     # indices of streamlines in corticospinal tract right
└─ dpg/                # data per group
    └─ mean_fa.1.float32 # n_groups × 1
```

Skeletons

Often just **held as tables** (.csv) – stanadarised table formats exist like SWC
exist to improve consistency for I/O but still as single files

SWC format:

file.swc

#	ID	type	X	Y	Z	radius	parent	
1	1		0.0	0.0	0.0	5.0	-1	# soma (root)
2	3		1.2	0.3	0.1	2.0	1	# dendrite off soma
3	3		2.4	0.6	0.2	1.5	2	# dendrite continues
4	3		3.1	1.2	0.3	1.0	3	# dendrite continues
5	3		2.8	0.1	0.8	1.2	3	# branch point
6	2		-1.0	0.2	0.1	1.8	1	# axon off soma
7	2		-2.1	0.4	0.2	1.5	6	# axon continues

Graphs

Graphs are **structured as links between vertices** (src,trg pairs). Edge and node attributes added separately and unrelated to paths. Directed or undirected. Generally as one large per-object table.

In XML format, nodes and edges are held as elements, with arbitrary attributes via `<data>` keys.

```
file.graphml
<graphml>
  <key id="w" for="edge" attr.name="weight" attr.type="float"/>
  <key id="r" for="node" attr.name="radius" attr.type="float"/>
  <graph edgedefault="undirected">
    <node id="n0"><data key="r">2.5</data></node>
    <node id="n1"><data key="r">1.8</data></node>
    <node id="n2"><data key="r">3.0</data></node>
    <edge source="n0" target="n1"><data key="w">0.5</data></edge>
    <edge source="n1" target="n2"><data key="w">0.8</data></edge>
    <edge source="n0" target="n2"><data key="w">0.3</data></edge>
  </graph>
</graphml>
```


Meshes

Existing file formats (mainly .STL and .OBJ) represent tessellated meshed objects in 3D space. A mesh is just two lists: **vertices** (xyz) and **faces** (which vertices connect to form surfaces as triangles or tetrahedrons). Normals, colours, textures are attributes layered on top.

Raw mesh data structure:

Vertices:

```
v0 = [0.0, 0.0, 0.0]
v1 = [1.0, 0.0, 0.0]
v2 = [0.5, 1.0, 0.0]
v3 = [0.5, 0.5, 1.0]
```

Faces (triangles):

```
f0 = [v0, v1, v2]
f1 = [v0, v1, v3]
f2 = [v0, v2, v3]
f3 = [v1, v2, v3]
```

Same example as OBJ:

```
v 0.0 0.0 0.0
v 1.0 0.0 0.0
v 0.5 1.0 0.0
v 0.5 0.5 1.0
f 1 2 3          # (1-indexed)
f 1 2 4
f 1 3 4
f 2 3 4
```

Draco compression

Draco is Google's mesh compression format* which compresses edges and attributes separately:

Connectivity compression (EdgeBreaker): The face list is a sequence of triangles, each referencing three vertex indices. As mesh triangles are not random, they share edges in predictable patterns so a lot of data can be omitted.

Quantisation: float coordinates are mapped to integers within a bounding box using a configurable number of bits (small loss of precision, be cautious).

Prediction: rather than storing each position, Draco predicts vertex's position from neighbours and only the residual is stored (small so compress well).

**This is included as an option in the Zarr-vectors spec.*



DRACO
3D DATA COMPRESSION

Zarr-vectors

Principals

Guiding aims and
requirements of the spec.

Aims of use of Zarr-vector data

Aims for use of final data:

- Broad range of data types for many communities to single scalable format
- Handle and visualise data within data-efficient structures
- Use multiple smaller files or sharded data (can cross drives)

Formats to include in Zarr-vectors:

- Point clouds (2D, 3D, N-dimensional)
- Meshes (triangular, quad, tetrahedral, etc.)
- Skeletons and graphs
- Streamlines and polylines
- Tracks through time
- Spatial transcriptomics (cells and detection points)
- Arbitrary N-dimensional vector data
- Spatial indexing and multi-resolution
- Distributed read/write operations
- Rich metadata support

General Principals for data structure

Hierarchy of structure:

Time, resolution, N channels, structural, spatial, attributes

Hierarchy of structural complexity:

Unit elements: points, planes

Basic elements: lines (+ vector), polygons – basic items dependant on units

Composite elements: paths (polyline), graphs, surface - directed or undirected linked basic components

Complex: mesh, multi-resolution objects – many composite elements or substructures

General Principals for data handling

Data should be interoperable and mirror underlying OME-Zarr array data.

Data size must vary with available memory and FOV.

Data must remain connected across chunk boundaries.

General Principals for layers

The whole structure(s) must be represented in a single chunk.

Solution: Replace array binning with coarsening

- Points (point cloud spatial grouping – metanodes)
- Lines/streamline (coarsening / remove intermediate points)
- Graph (topological coarsening)
- Meshes (total element reduction)

Zarr-vectors

Main Challenges

Challenges and scalability
bottlenecks for storage,
filtering, or handling

Linking across chunks

It will be challenging to coherently link

Building a balanced format:

Analysis vs I/O efficiency trade-offs

For some cases structure will be easy to build for either fast I/O or for more effective handling. Find a balanced case that is acceptable to both sides.

Nomenclature and terms of reference

Definitions of data types
used in the Zarr-vectors

Vertices

Vertices are the atomic spatial units — individual points in N-dimensional space. A vertex is a single coordinate like (x=142.3, y=87.1, z=204.6). Every piece of data in Zarr-vectors ultimately resolves to vertices.

A point cloud is just vertices. A streamline is an ordered sequence of vertices. A mesh triangle is three vertices plus a connectivity record.

Objects

Objects are the meaningful geometric entities your science cares about: one neuron, one streamline, one mesh surface, one cell. An object is composed of one or more vertices.

A streamline with 500 points is one object made of 500 vertices. A neuron skeleton spanning 30 spatial chunks is one object whose vertices are scattered across those chunks.

The object is what has biological or physical meaning.

Groups

Groups are collections of objects that share some semantic label.

I.e. all streamlines belonging to the arcuate fasciculus, all cells classified as GABAergic, all mesh fragments belonging to the cortex.

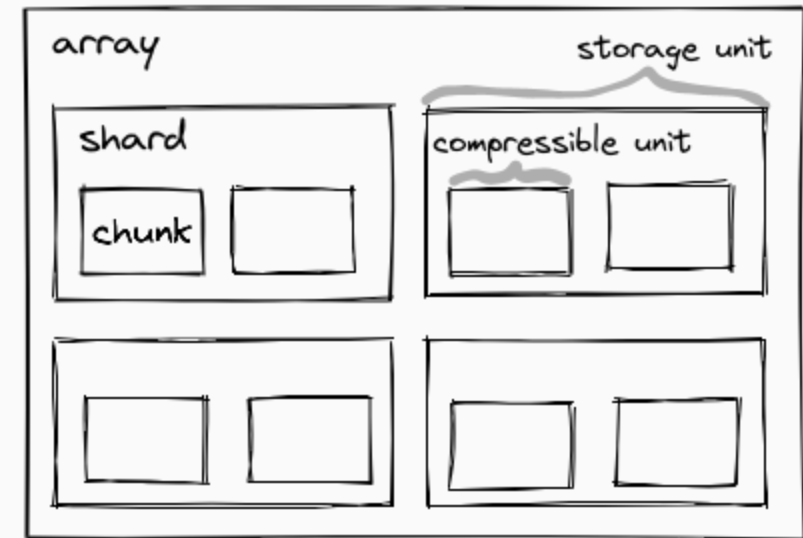
A group doesn't own vertices directly. it owns objects, which in turn have their own vertices.

Sharding

Sharding is the **concept of including many array chunks into a single storage unit**. This is to reduce the number of files in a Zarr (v3) without having to change the chunk size.

Each chunk is compressed separately, then sorted in a shard (so accessing one chunk doesn't need to decompress other chunks in the shard).

The aim of this is to simplify file storage and improve upload and download times (often impacted by file count) without changing what the Zarr is doing.



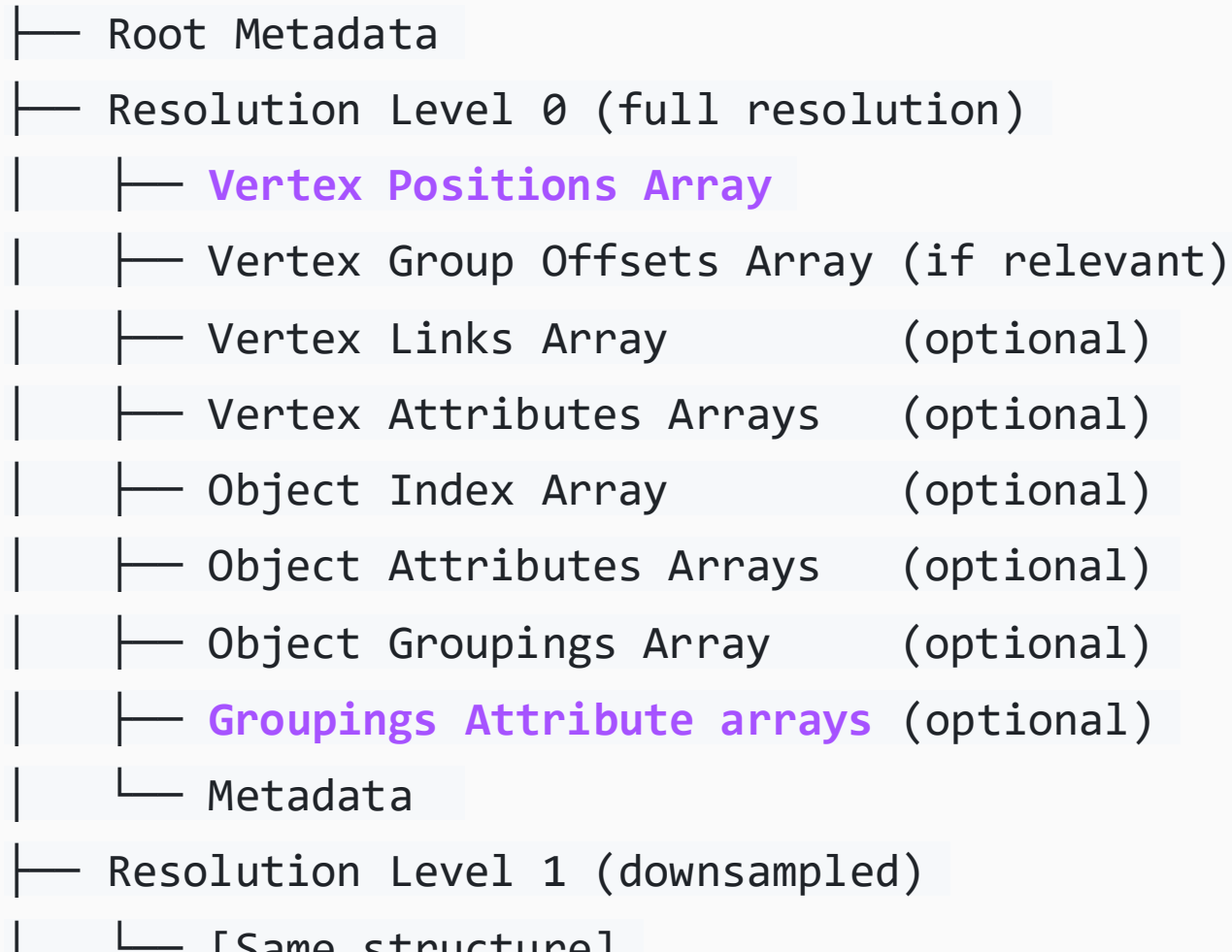
Structure & Metadata

The overall folder directory
and it's metadata

Overall Zarr-vectors file structure (from spec.)

As .zarrvectors or .zvr

Zarr Store Root



Whole structure metadata

Overall metadata (like OME-Zarr 'multiscales') or root metadata:

Defines axes as a list of {name, type, unit} objects

(e.g. {"name": "z", "type": "space", "unit": "micrometer"})

Can expand to N dimensions (unlike OME-Zarr which is restricted to 2 to 5 dimensions)

Also include NIfTI header for registration between Zarr-vectors datasets (like NIfTI-Zarr).

Per-layer metadata

Similar to .zattrs per level

Things to add:

How does Zarr-vector differ from OME-Zarr?

OME-Zarr has a single array (or a set of files corresponding to many chunks representing a single very large array) per resolution level.

Zarr-vectors uses many arrays to describe different information needed to handle and reconstruct vector-based objects across chunks.

These new arrays are:

- Vertices/
- Vertex_group_offsets/
- Links/
- Attributes/
- Object_index/
- Groupings/
- Grouping_attributes/

vertices/

This is the **only required array** and describes vertex **positions in space** (minimum of x or x,y,z position of a point in space). If empty, the whole structure is empty.

Each file describes the vertices that fall under the file it is responsible for.

N-dimensional Zarr array where the leading dimensions are the spatial index (the chunk grid) and the trailing dimension is ragged (each chunk holds a variable-length of vertex coordinates).

vertex_group_offset/

Indexes vertices within a single chunk (i.e. "find vertex id=3 in chunk (0,0,0)"). This array **enables within-chunk random access**.

Each spatial chunk has a $K \times 2$ array, where K is the number of vertex groups in that chunk held as:

[vertex_byte_offset, link_byte_offset]

which are the position where vertex group k begins in the vertices/ chunk and the position where its links begin in the links/ file.

To find where group k ends, you look at row $k+1$ (or the end of the chunk for the last group).

attributes/

Attributes are **per-vertex measurements or labels** (anything more than coordinates in space). Attributes array indexing aligns with vertices to match points with data.

Each attribute gets its own sub-array and can have N different attributes:
(e.g. `attributes/radius/`, `attributes/gene_expression/`, ...)

This is mirrored from the `.trx` format (channel dimension)

links/

Encodes **how vertices connect to each other** within a chunk. The meaning of "link" depends on the geometry type*:

Geometry	Definition in links/
Streamlines (polylines)	N/A, connected implicitly in vertex group
Skeletons (trees)	Links are implicit parent relationships. Only branching nodes are stored here (reduces data size)
Graphs	Links are an explicit edge list, $M \times 2$ per chunk (src, trg)
Mesh	Links are face definitions: tuples of vertex indices forming a face ($L=3$ for triangles, $L=4$ for tetrahedra). standard face-index-list as in OBJ/PLY

**This array is optional. Point clouds have no connectivity so this would be empty.*

object_index/

The object index **connects between the object level and vertices in files** (i.e. "given object ID 42, which vertex groups in which chunks contain its data?") The array maps each object ID to a ragged list of tuples as:

`(chunk_coordinates, vertex_group_index)`

Example: A streamline passes through three spatial chunks,

Object 42's entry might be `[(0,0,0,2), (0,0,1,0), (0,1,1,3)]`

So, vertex group 2 in chunk `(0,0,0)`, vertex group 0 in chunk `(0,0,1)`, ...

For ordered geometries, reconstruct by concatenating vertex groups in the order.

When everything fits in a single spatial chunk, object ID equals vertex group index and this entire array is omitted (TRX-compatible for small datasets).

object_attributes/

Per-object metadata for properties of the whole single object rather than individual vertices. Each attribute is an $O \times C$ array (O objects, C channels).

Examples: streamline source and sink regions (termination, $O \times 2$), mesh material ID, cell type classification, neuron compartment label.

These are dense arrays indexed by object ID, not spatially chunked.

You read: `object_attributes/termination/` and get one row per object in the store.

`object_attributes/edge_weights/` would be an array matching links and `cross_chunk_links` which contain edge attributes (or use `links_attributes`)

groups/

Defines the groups of objects. This is a $G \times$ ragged array where each row is a list of object IDs belonging to that group.

(i.e. Group 0 might be `[0, 1, 5, 12, 47]` (objects of corticospinal tract), and so on.

Groups can overlap and the same object can appear in multiple groups. This is useful for hierarchical classification (a cell belongs to both "glutamatergic" and the broader "neuron" super-group).

group_attributes/

Per-group metadata (properties of the collection, not the individual objects). Each attribute is a different G×C array (sub-array like in vertex or object attributes).

Example: A streamline tract group may have:

```
group/ -> group_ID -> [objects] (streamlines)
```

```
group_attributes/tract_names/ -> (group_ID, "CST")
```

then:

```
objects/ -> object ID -> [vertices] (points)
```

```
+ objects_attributes/length -> object_ID -> len([vertices])
```

then:

```
Vertices/ -> vertex_ID -> (i, [x,y,z])
```

+

gr

cross_chunk_links/

This array contains **all links between vertices of different chunks**.

The array stores these boundary-crossing connections as pairs of:

`(chunk_A_coords + vertex_offset, chunk_B_coords + vertex_offset)`.

This approach is as an alternative to boundary deduplication (where you'd place a copy of the border vertex in both chunks and rely on coordinate matching to infer the connection). Some datasets use boundary deduplication instead, and pure point clouds don't need it at all.

Handling parametric elements

Parametric elements like **lines and planes can be represented as expressions:**

Planes:

$$Ax + By + Cz + D = 0$$

Lines (a bit more complex):

$$P(t) = P_0 + t \cdot d$$

Where $P_0 = (x_0, y_0, z_0)$ is a point on the line, $d = (d_x, d_y, d_z)$ is the direction vector, and t is a scalar parameter.

These expressions don't easily fit into the current spec: so suggested solution is to place a point at the origin of the chunk – and assign parametric elements to it as an attribute. Then duplicate this into each relevant chunk (slow I/O, fast read).

Or: make a parametric/ array (no layers) then include them all there as they don't discretise. (fast I/O and handling but risks complex data irreducible at higher levels – exceeds RAM :/)

Parametric/ (proposed solution)

An array outside the levels structure (root/parametric/) to contain geometric **objects represented by mathematical expressions** like planes and indefinite lines.

These may be geometry objects that we don't want to vary by resolution (rendering matter not for data storage) like anatomical planes or bounding boxes. Or object that aren't able to be discretised

Planes to be used for filtering of streamlines (intersections with polylines/mesh).

Lines to be used for annotation (custom axes, normals).

Spheres for annotation or region placement (more precise than mask in low-res case)

Parametric directory structure

```
dataset.zarr/
├── .zattrs                                # root metadata (SID, CRS, etc.)
├── resolution_0/
│   └── [standard vertex-based arrays]
├── resolution_1/
│   └── ...
├── parametric/
│   ├── .zattrs                            # geometry_types: ["plane", "line", ...], CRS ref
│   ├── objects/
│   │   ├── .zarray                        # 0 × ragged: type enum + coefficients per object
│   │   ├── object_attributes/
│   │   │   ├── name/                     # 0xC: "midsagittal", "coronal_slice_42"
│   │   │   ├── bounds/                   # 0x6: optional finite extent [xmin,xmax,ymin,...]
│   │   │   └── display/                   # 0xC: color, opacity, etc.
│   │   └── groupings/                    # G × ragged: group parametric objects together
│   │       ├── .zarray                    # e.g. [[0,1], [2,3]] – anatomical planes, clipping planes
│   │       └── groupings_attributes/
│   │           └── group_name/            # "reference_planes", "clipping_set_1"
```

<- Held outside of resolution levels – these objects are not reduced by chunking

<- Creates a copy of per resolution structure – but **excludes vertices** semantic layer, so treats parametric elements directly as objects.

<- I.e. a bounding box would be a group of parametric objects.

Zarr-vectors

Basic elements

The simplest data
represented by Zarr-vectors

A single point

A single points of spatial data with linked to vertex attribute information. The **most basic unit** described by Zarr-vectors

The vertices chunk 1.2.0 contains one vertex group with a single 3D coordinate. `vertex_group_offsets` has one row `[0, -]` (no links). The `object_index` maps this object ID to `[(1,2,0, 0)]` — chunk (1,2,0), vertex group 0. No `links` array needed.

resolution_0/

└─ vertices/1.2.0	→ [[x, y, z]]	# 1 vertex group, 1 vertex
└─ vertex_group_offsets/1.2.0	→ [[0, -]]	# 1 row, no links
└─ object_index/0	→ [(1,2,0, 0)]	# object 0 → chunk (1,2,0), vg 0

Point clouds

Array of many point vertices across multiple files. **Layers by grouping of points together to meta-points** (cloud coarsening – Knn, K=8). An example of many measured points across chunks with intensity and colour:

```
resolution_0/  
├─ vertices/  
|   ├─ 0.0.0   → [ [x,y,z], [x,y,z], ... ]   # all points in chunk, one vertex group  
|   ├─ 0.0.1   → [ [x,y,z], ... ]  
|   └─ ...  
├─ attributes/  
|   ├─ intensity/  
|   |   ├─ 0.0.0 → [ i0, i1, i2, ... ]       # aligned 1:1 with vertices  
|   |   └─ ...  
|   └─ color/  
|       ├─ 0.0.0 → [ [r,g,b], [r,g,b], ... ]   # channel dim = 3  
|       └─ ...
```

Lines (within a chunk)

A **line (A→B) within a single chunk** (both endpoints in chunk (0,1,0))

The vertices chunk 0.1.0 contains one vertex group with two consecutive vertices (point A, point B). The object_index maps the line's object ID to a single entry [(0,1,0, 0)]. Links are implicit sequential (vertex 0 connects to vertex 1), so with links_convention: "implicit_sequential" the links array is omitted entirely.

resolution_0/

└─ vertices/0.1.0 → [[Ax,Ay,Az], [Bx,By,Bz]] # 1 vertex group, 2 vertices

└─ vertex_group_offsets/0.1.0 → [[0, -]] # 1 row

└─ object_index/0 → [(0,1,0, 0)] # single chunk reference

Lines (across chunks)

A **line (A→B) crossing a chunk boundary** (A in chunk $(0, 1, 0)$, B in chunk $(0, 2, 0)$)

Each chunk holds one vertex group containing its endpoint. The `object_index` stores two entries in order: $[(0, 1, 0, 0), (0, 2, 0, 0)]$, defining the reconstruction sequence. Connectivity between the chunks is handled either by boundary deduplication (A is placed exactly on the border and duplicated in both chunks) or by an explicit entry in `cross_chunk_links` referencing the last vertex of the first chunk and the first vertex of the second.

resolution_0/

└─ vertices/0.1.0	→ [[Ax,Ay,Az]]	# vertex group 0: point A
└─ vertices/0.2.0	→ [[Bx,By,Bz]]	# vertex group 0: point B
└─ vertex_group_offsets/0.1.0	→ [[0, -]]	
└─ vertex_group_offsets/0.2.0	→ [[0, -]]	
└─ object_index/0	→ [(0,1,0, 0), (0,2,0, 0)]	# ordered: A's chunk then B's chunk
└─ cross_chunk_links/0	→ [(0,1,0,0) → (0,2,0,0)]	# OR boundary deduplication instead

Lines (parametric)

A line through space, represented by an expression: $P(t) = P_0 + t \cdot d$

```
dataset.zarr/  
└─ parametric/  
  └─ objects/  
    └─ .zarray      → [ [1, 10, 20, 30, 0, 0, 1] ]  
    └─              #   type=1 (line), P0=(10,20,30), d=(0,0,1)  
    └─              #   → P(t) = (10, 20, 30+t)  
  └─ object_attributes/  
    └─ name/        → [ "z_axis_probe" ]
```

Planes (parametric)

A plane as an algebraic expression between x, y, z with the general form:

$$Ax + By + Cz + D = 0.$$

May have limits as expression (f(x,y,z) or x>n) in object attributes.

```
dataset.zarr/  
└─ parametric/  
  └─ objects/  
    └─ .zarray          → [ [0, 0.0, 1.0, 0.0, -50.0] ]  
    └─                  #   type=0 (plane), A=0, B=1, C=0, D=-50  
    └─                  #   → 0x + 1y + 0z - 50 = 0 → y = 50  
  └─ object_attributes/  
    └─ name/            → [ "coronal_plane" ]
```

Vectors

Vectors are held as Zarr / OME-Zarr arrays. However, it may be useful to represent as lines/points with vector/magnitude attributes to points in space (i.e. a DVC strain vector field). **Useful for non-uniform fields** (or points assigned orientation or movement in response to force).

resolution_0/

```
└─ vertices/1.2.0          → [ [x, y, z] ]    # 1 vertex group, 1 vertex
└─ vertex_group_offsets/1.2.0 → [ [0, -] ]      # 1 row, no links
└─ object_index/0          → [ (1,2,0, 0) ]    # object 0 → chunk (1,2,0), vg
└─ vertex_attributes/vector → [ ((1,2,0,0), [dx, dy, dz] ] # orientation vec
└─ vertex_attributes/magnitide → [ ((1,2,0,0), magnitude ] # vec magnitude
```

Could be **used to represent warps** (non-array (feature point) dense displacement field)

Composite elements

Elements formed of more
than one basic element

Surfaces

Plane bounded by at least three distinct line objects.

May have node objects as vertices.

resolution_0/

└─ vertices/0.0.0 → [[x₀,y₀,z₀], [x₁,y₁,z₁], [x₂,y₂,z₂]] # 3 vertices forming one triangle, 1 vertex group
containing all 3

└─ vertex_group_offsets/0.0.0 → [[0, 0]] # 1 row: vertex group 0 starts at byte 0, link_offset 0 → face definition starts at
byte 0 in links

└─ links/0.0.0 → [[0, 1, 2]] # 1 face: triangle connecting vertex 0 → 1 → 2, L=3 (triangular face)

└─ object_index/0 → [(0,0,0, 0)] # object 0 → chunk (0,0,0), vertex group 0

└─ object_attributes/

 └─ surface_normal/ → [[nx, ny, nz]] # 0x3: outward normal of the triangle

Streamline (directed polyline path)

The minimal model of a streamline (implicitly directed path) as an object list of vertices.

```
resolution_0/
├─ vertices/0.0.0      → [ [x0,y0,z0], [x1,y1,z1], [x2,y2,z2], [x3,y3,z3], [x4,y4,z4] ]
|                        # 1 vertex group, 8 vertices in sequence, implicitly: 0→1→2→3→4→5
|
├─ vertex_group_offsets/0.0.0 → [ [0, -] ]      # 1 vertex group starting at byte 0, no link offset (links omitted)
├─ object_index/0         → [ (0,0,0, 0) ] # streamline 0 → single chunk, vertex group 0
├─ object_attributes/
|   └─ termination/       → [ [region_A, region_B] ]      # 0x2: source and sink regions|
├─ attributes/
|   └─ fa/0.0.0           → [ fa0, fa1, fa2, fa3, fa4, fa5 ] # fractional anisotropy per vertex, aligned 1:1
```

Streamline (across chunks)

The minimal model of a streamline passing between two different spatial chunks:

```
resolution_0/
├─ vertices/0.0.0          → [ [x0,y0,z0], [x1,y1,z1], [x2,y2,z2] ]    # vertex group 0: first 3 points
├─ vertices/0.0.1          → [ [x3,y3,z3], [x4,y4,z4], [x5,y5,z5] ]    # vertex group 0: last 3 points
├─ vertex_group_offsets/0.0.0 → [ [0, -] ]
├─ vertex_group_offsets/0.0.1 → [ [0, -] ]
├─ object_index/0          → [ (0,0,0, 0), (0,0,1, 0) ]    # ordered: concatenate to reconstruct 0→1→2→3→4→5
├─ cross_chunk_links/0     → [ (0,0,0, 2) → (0,0,1, 0) ]    # vertex 2 in chunk (0,0,0) connects to vertex 0 in chunk (0,0,1)
├─ object_attributes/
|   └─ termination/        → [ [region_A, region_B] ]
|
├─ attributes/
|   └─ fa/0.0.0            → [ fa0, fa1, fa2 ]
|   └─ fa/0.0.1            → [ fa3, fa4, fa5 ]
```

Tree

A tree is an implicitly directed polyline with special considerations at branching points.

resolution_0/

```
└─ vertices/0.0.0 → [ [x0,y0,z0], ... ] # n0 (root) and n1, ... n6 in a vertex group
|
| # in depth-first order: n0→n1→n3→n6→n4→n2→n5
└─ links/0.0.0 → [[2, 0], [4, 1], [5, 2]] # n5's parent is n2 (not n4, breaks sequence)
|
| # only 3 branch links stored
|
| # sequential links are implicit:
|
| # n1←n0 (1←0) ✓ sequential
|
| # n3←n1 (3←2) ✓ sequential
└─ vertex_group_offsets/0.0.0 → [ [0, 0] ]
└─ object_index/0 → [ (0,0,0, 0) ]
```

Tree described:

```
n0 (root)
      / \
     n1  n2
    / \   \
   n3 n4  n5
    |
   n6
```

Graph (Spatial graph network)

A **collection of vertices with internal and between chunk links**. Where the edge list is held between links and cross_chunk_links and nodes are vertices. TBD – how edge properties are held

resolution_0/

```
└─ vertices/0.0.0      → [ [x0,y0,z0], [x1,y1,z1] ]      # vertex 0 = n0, 1 = n1
└─ vertices/0.0.1      → [ [x2,y2,z2], [x3,y3,z3] ]      # vertex 0 = n2, 1 = n3
└─ links/0.0.0          → [ [0, 1] ]          # n0–n1 (local indices within chunk) <- internal edges
└─ links/0.0.1          → [ [0, 1] ]          # n2–n3 (local indices within chunk)
└─ vertex_group_offsets/0.0.0 → [ [0, 0] ]
└─ vertex_group_offsets/0.0.1 → [ [0, 0] ]
|
└─ object_index/0       → [ (0,0,0, 0), (0,0,1, 0) ]
|
└─ cross_chunk_links/0   → [ [(0,0,0, 0), (0,0,1, 0)], [(0,0,0, 0), (0,0,1, 1)], [(0,0,0, 1), (0,0,1, 0)] ] # n0–n2, n0–n3, n1–n2
```

Edge attributes and weights

Zarr-vectors is a vertex-based data structure so we need to add a `link_attributes` array.

Existing structures hold edge weights alongside links – how is handling going to be different here?

Impacts on pathfinding efficiency and handling in chunked graphs?

Complex elements

Elements formed of many
composite elements

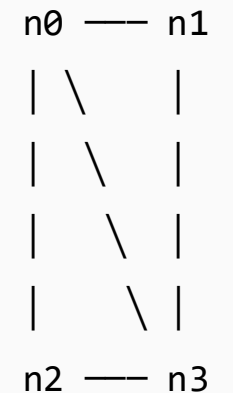
Mesh

The most complex object represented by Zarr-vectors constructed from graph and surface structures.

resolution_0/

```
└─ vertices/0.0.0      → [ [x0,y0,z0], ... ]      # n0 - n3 in a vertex group
└─ links/0.0.0         → [ [0, 1, 3], [0, 3, 2] ] # face 0: n0-n1-n3, ...
|                                     # 2 faces, L=3 (triangles)
|
└─ vertex_group_offsets/0.0.0 → [ [0, 0] ]
└─ object_index/0       → [ (0,0,0, 0) ]
└─ attributes/
    └─ normal/0.0.0     → [ [nx0,ny0,nz0], ... ] # per-vertex normals
```

Mesh described:



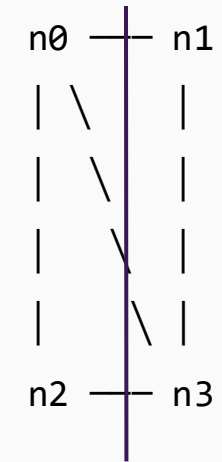
Mesh (across chunks)

The worst case for meshes if both faces span the chunk boundary

resolution_0/

```
└─ vertices/0.0.0      → [ [x0,y0,z0], [x1,y1,z1] ]      # n0, n1
└─ vertices/0.0.1      → [ [x2,y2,z2], [x3,y3,z3] ]      # n2, n3
└─ links/0.0.0         → [ ]                                     # if any local faces
└─ links/0.0.1         → [ ]                                     # if any local faces
|
└─ vertex_group_offsets/0.0.0 → [ [0, 0] ]
└─ vertex_group_offsets/0.0.1 → [ [0, 0] ]
└─ object_index/0       → [ (0,0,0, 0), (0,0,1, 0) ]           # mesh spans both chunks
└─ cross_chunk_links/0  → [ [(0,0,0,0),(0,0,0,1),(0,0,1,1)],      # face 0: n0–n1–n3
|                          [(0,0,0,0),(0,0,1,1),(0,0,1,0)] ]    # face 1: n0–n3–n2
└─ attributes/
|   └─ normal/0.0.0     → [ [nx0,ny0,nz0], [nx1,ny1,nz1] ]
|   └─ normal/0.0.1     → [ [nx2,ny2,nz2], [nx3,ny3,nz3] ]
```

Mesh described:



Computing Higher Levels

Approaches to reducing
different structures sizes to
fit into available memory

Principals:

How to reduce vector-based elements

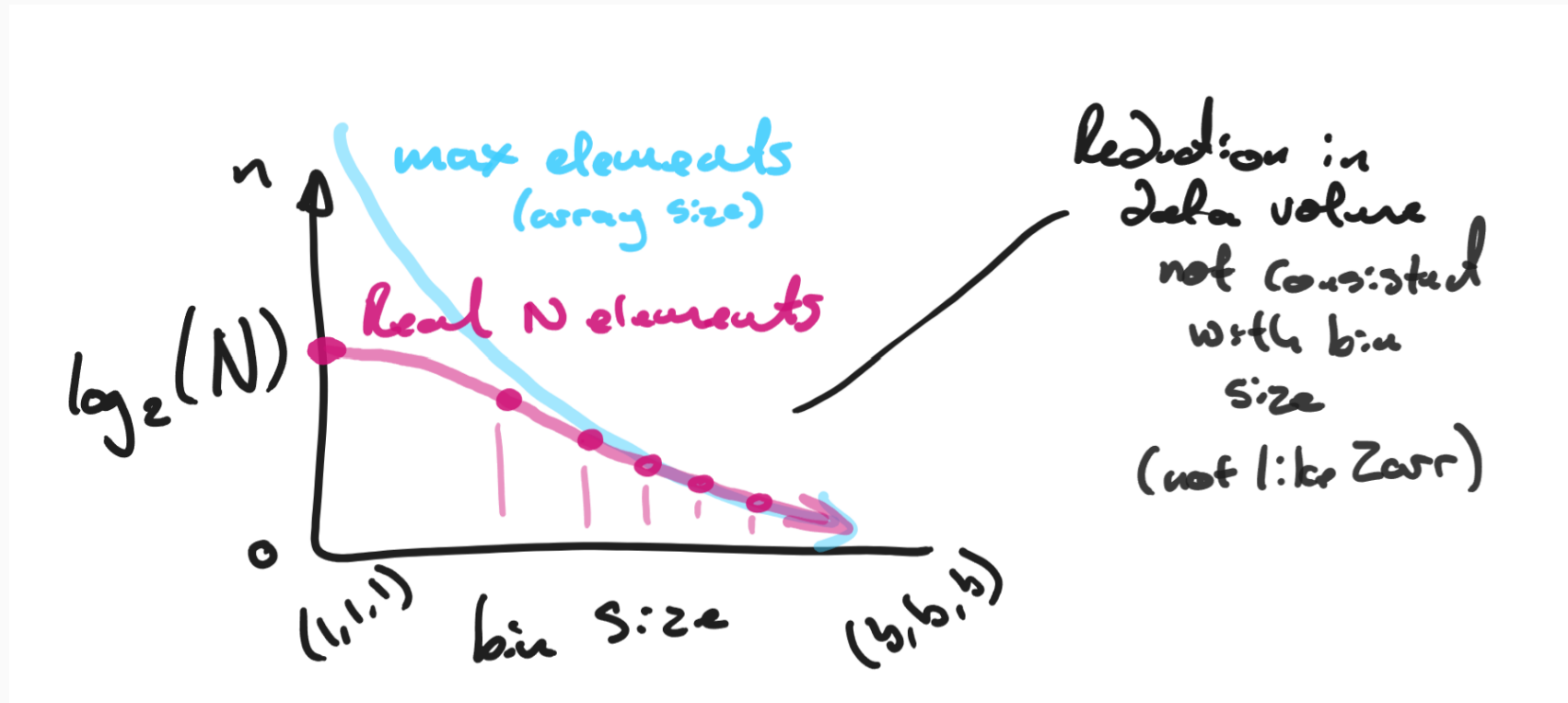
For visualisation we want to have as dense a vector-object (most number of elements that fit in memory) in a given spatial area.

We want to follow the octree structure with **progressive spatial coarsening of vertex elements** (mirroring ChunkGraph) so each level reduces global element count by a factor of 2, 4, or 8 per level. The links which cross these new meta nodes (within or between chunks) are added as a single link in the new level.

Vertices at higher-levels act like objects and reference child vertices vectors (as vertex attributes). So, objects (i.e. a streamline) keep the same structure, just containing the coarsened vertices.

Mesh surfaces are a little more complex to reduce – see later slides.

Element count vs bin size



With non-uniform vertex distribution, a sparse point cloud won't start to be reduced until higher levels. So the important factor is a consistent reduction in mean data size per chunk (N vertex), compared to the previous layer.

Point clouds (all vertices)

Find meta nodes by progressive binning until elements reduced by factor of N (similar to ChunkGraph).

New vertices (meta-points) are given the mean spatial position of their vertices in the prior level.

Computational approach:

- Find $N(0)$ vertices at lvl 0
- For bin size = $(1,1,1)$ count pixels where a vertex exists, if $N(1)$ vertices is $(N(0)/K)$, then save new level
- If not, try bin size = $(2,2,2)$ and so on.

Generally: Increment bin_size by 1 (or n , or power m) and save new level if $N(i) < N(i-1)/K$, otherwise increment again. Prevents wasted data at low levels and gives control at higher levels.

Example of binning layers

K = 8 as default value to match OME-Zarr

Example with K = 8

Layer 0:	chunk = 10,	max 500 nodes/chunk,	N = 1,000,000	
Layer 1:	chunk = 20,	max 1 node/chunk,	N = 125,000	(8× fewer chunks)
Layer 2:	chunk = 40,	max 1 node/chunk,	N = 15,625	
Layer 3:	chunk = 80,	max 1 node/chunk,	N = 1,953	
Layer 4:	chunk = 160,	max 1 node/chunk,	N = 244	

Example with K = 2

Layer 0:	bin = none,	N = 1,000,000	→ stored
Layer 1:	bin = 50,	N = 500,000	→ stored ($N_0/2$ reached)
Layer 2:	bin = 90,	N = 250,000	→ stored ($N_1/2$ reached)
Layer 3:	bin = 150,	N = 125,000	→ stored
Layer 4:	bin = 300,	N = 62,500	→ stored

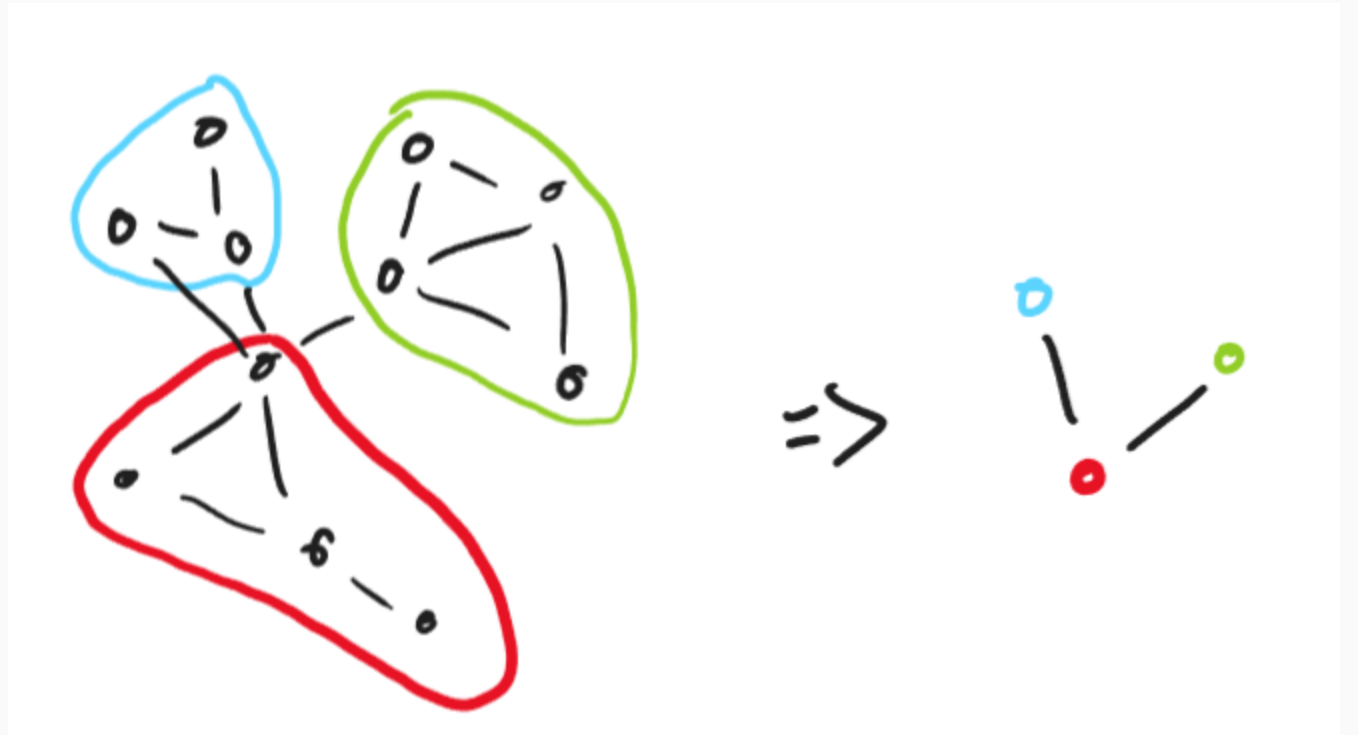
Streamlines (any polyline)

Implicitly binned by vertex binning so there is no specific coarsening strategy for streamlines, as the underlying vertices are already coarsened. At higher resolutions – polylines are vector of meta-vertices (implicitly identical to a linked array used by dipy)

Graphs

Implicitly binned by vertex binning.

If an edge existed between any node in two different groups of vertices – form an edge between the binned vertex.



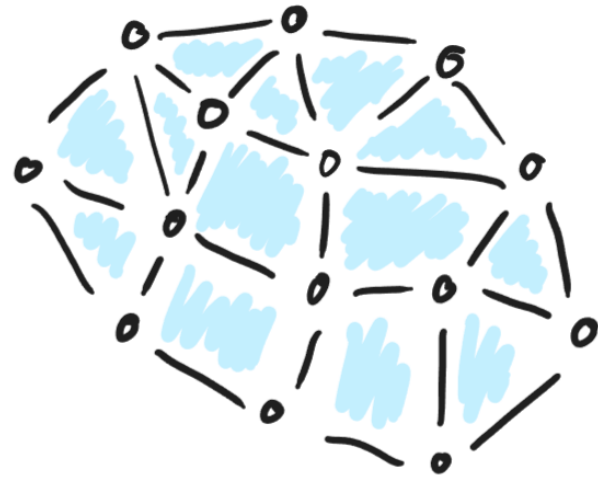
Graphs

data structure

A basic example of how a graph made of 8 nodes (over 3 chunks) get binned from level 0 to 1 by binning chunks by 2 to reduce total chunk count by factor of 8:

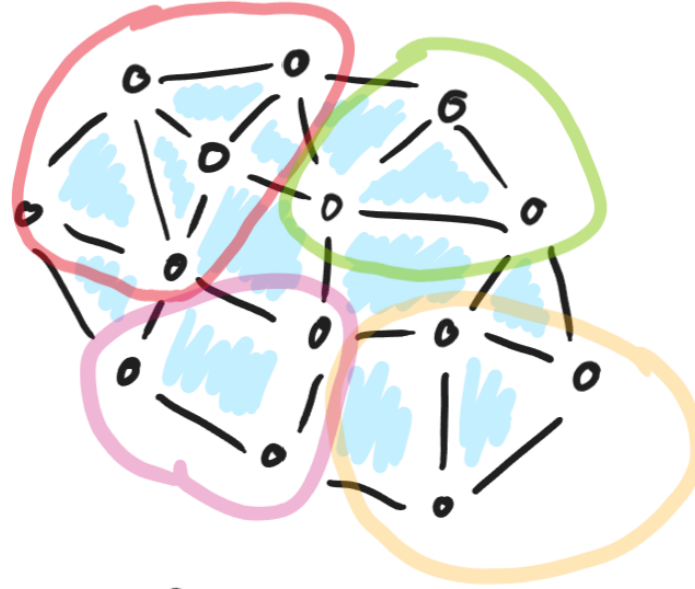
```
resolution_0/                                # full resolution
├─ vertices/0.0.0    → [n0, n1, n2, n3]
├─ vertices/1.0.0    → [n4, n5, n6, n7]
├─ vertices/0.1.0    → [n8]
├─ links/0.0.0       → [[0,1],[1,2],[2,3]]    # intra-chunk edges
├─ cross_chunk_links → [(0.0.0,3)→(1.0.0,0)]  # n3→n4
resolution_1/                                # coarsened
├─ vertices/0.0.0     → [m0, m1, m2]          # metanodes in larger chunk
├─ links/0.0.0        → [[0,1],[1,2]]         # m0→m1, m1→m2
├─ vertex_attributes/children/ → [ [n0,n1,n2,n3], # m0's children (level 0 refs)
                                   [n4,n5,n6,n7],   # m1's children
                                   [n8] ]           # m2's children
```

Meshes



full input mesh

\Rightarrow



Reduce
vertices
by factor 2

\Rightarrow



Refine
surface
from
enclosed
links

Meshes

data structure

Vertices and links are reduced as detailed for point clouds and for graphs, the challenge with meshes is handling surface element reduction.

A similar approach as for graphs – enclosed triangles at higher levels will form surfaces if a surface existed between the underlying groups of vertices at the prior level.

Basic Utilities

Structuring the basic I/O and
selection/filtering utilities
for basic and some
composite elements

Zarr-vectors-py

A **python utility toolbox** for data reading (points, streamlines, graphs, meshes) and eventually for handling and filtering of multiscale vector elements.

Under construction – Aims to be working well in Summer 2026.

Github (temp. location):

<https://github.com/Andrew-Keenlyside/zarr-vectors-py>

The screenshot shows the GitHub repository page for **zarr-vectors-py**, which is marked as **Private**. The repository has 0 Watchers, 0 Forks, and 0 Stars. The main branch is **main**, with 1 Branch and 0 Tags. A search bar is present with the text "Go to file". There are buttons for "Add file" and "Code".

The commit history shows a recent update to the **readme** by **Andrew-Keenlyside** 4 days ago, with 3 Commits. The file list includes:

File	Commit	Time
assets	initial commit	4 days ago
tests	initial commit	4 days ago
zarr_vectors	initial commit	4 days ago
README.md	update readme	4 days ago
pyproject.toml	initial commit	4 days ago

The **README** section contains a **Note** stating: "This package is under development and will change. It will also be migrated to another location once completed." Below the note is a logo for **Zarr-vectors**, which features a 3D cube connected to a network of nodes and lines.

The **Tools for Zarr Vectors Data** section describes the package: "zarr-vectors-py is a Python package for reading, writing, and managing large-scale vector geometry data in the zarr vectors format — a chunked format built to mirror Zarr v3 for multiscale points, lines, streamlines, graphs, skeletons, and meshes." It also mentions alignment to the Zarr_vectors specification by Forest Collman, Allen Institute for Brain Sciences, with a [Link to specification](#) and a [GitHub](#) link.

The **Install** section provides the command: `pip install zarr-vectors`.

The right sidebar contains the **About** section, which describes the project as "Utilities and tools for working with data in the Zarr-vectors data structure". It lists 0 stars, 0 watching, and 0 forks. The **Releases** section states "No releases published" with a [Create a new release](#) link. The **Packages** section states "No packages published" with a [Publish your first package](#) link. The **Contributors** section lists 1 contributor: **Andrew-Keenlyside**. The **Languages** section shows **Python** at 100.0%. The **Suggested workflows** section, based on the tech stack, includes suggestions for "Publish Python Package", "Django", and "Python Package using Anaconda", each with a **Configure** button.