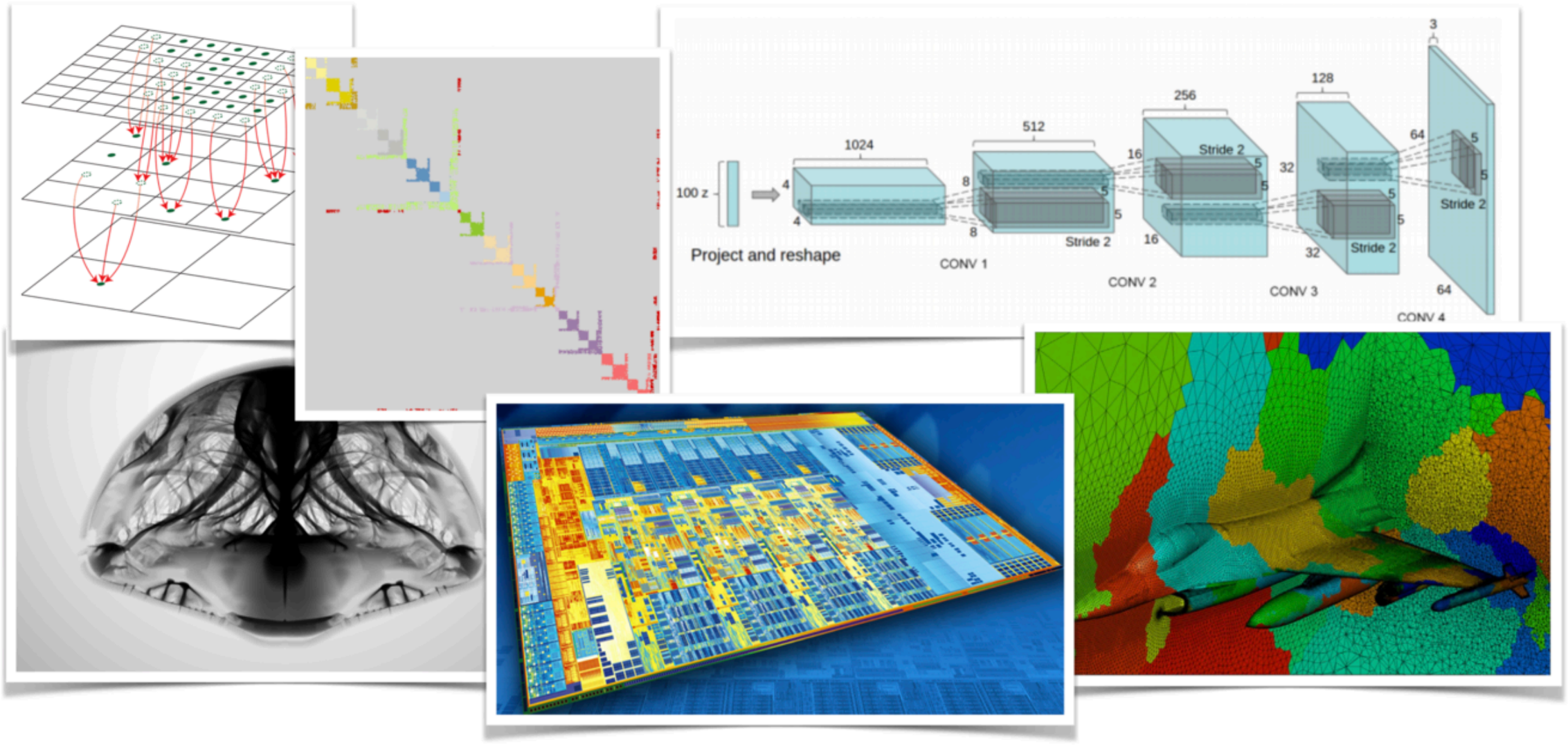


Lecture 23: Introduction to Sparse Grids. Basic concepts of OpenVDB/NanoVDB

Tuesday April 13th 2023

Logistics

- Programming Assignment #4 released last Sunday, due next Monday.Next week: Sparse grids and operations (NanoVDB)

- Midterm and HW1 *almost* done grading, will post by tomorrow early AM

- Guest lectures by Prof. Matt Sinclair on Apr 25, 27 and May 2nd

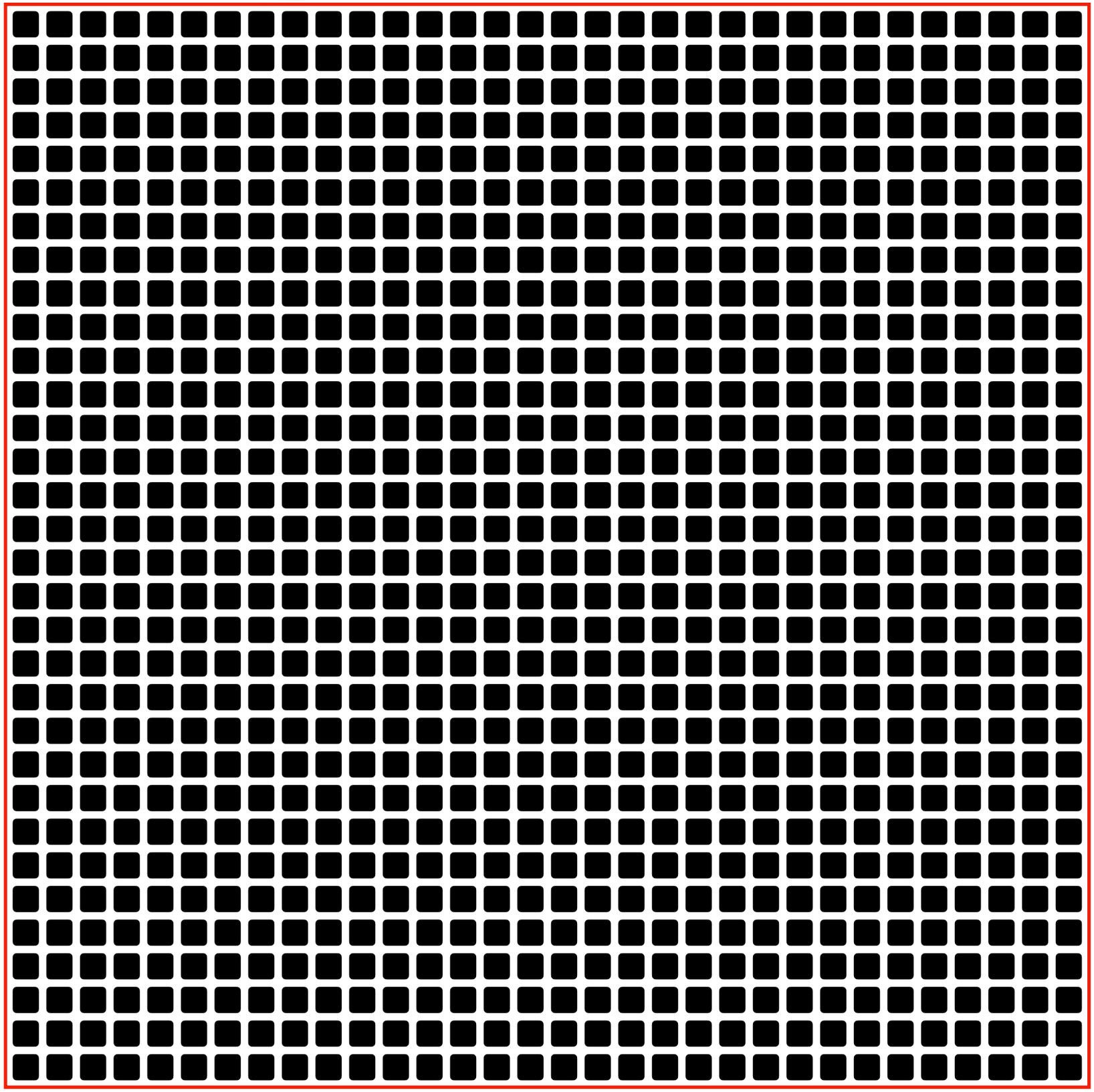
Today's lecture

- Sparse grids, theory and concepts
- We will focus on OpenVDB and NanoVDB (https://www.openvdb.org/)
- Today we will examine data structures and design of data structures within these APIs, Thursday we'll look at code

Sparse grids (intro concepts)

We previously saw operations on dense grids (stencils, etc)

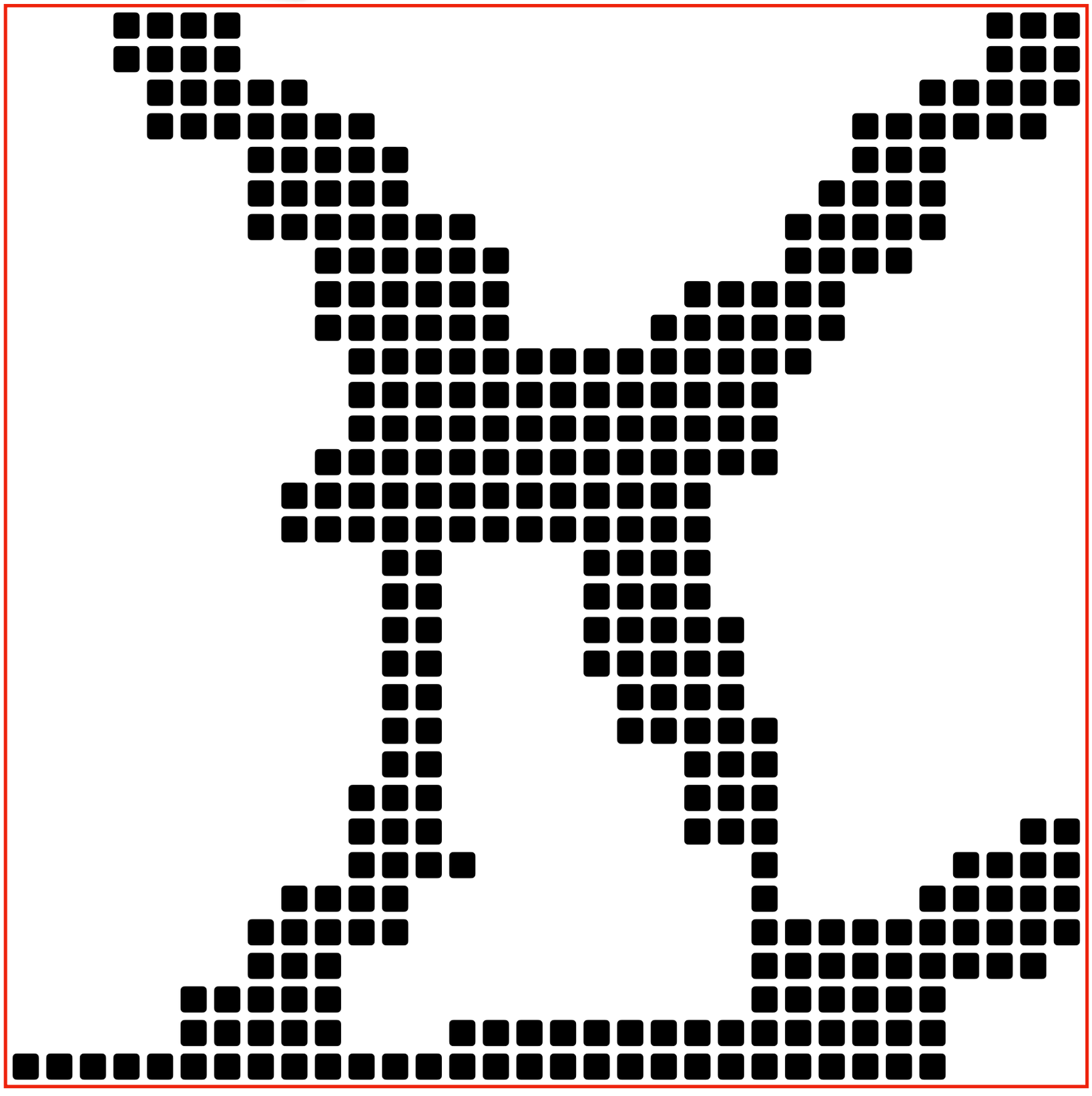
Stencils/matrices were potentially sparse (meaning that stencils were local) but the grid itself was dense



Sparse grids (intro concepts)

We now turn our attention where the grids we operate on only have information we care about in a subset of their spatial extent

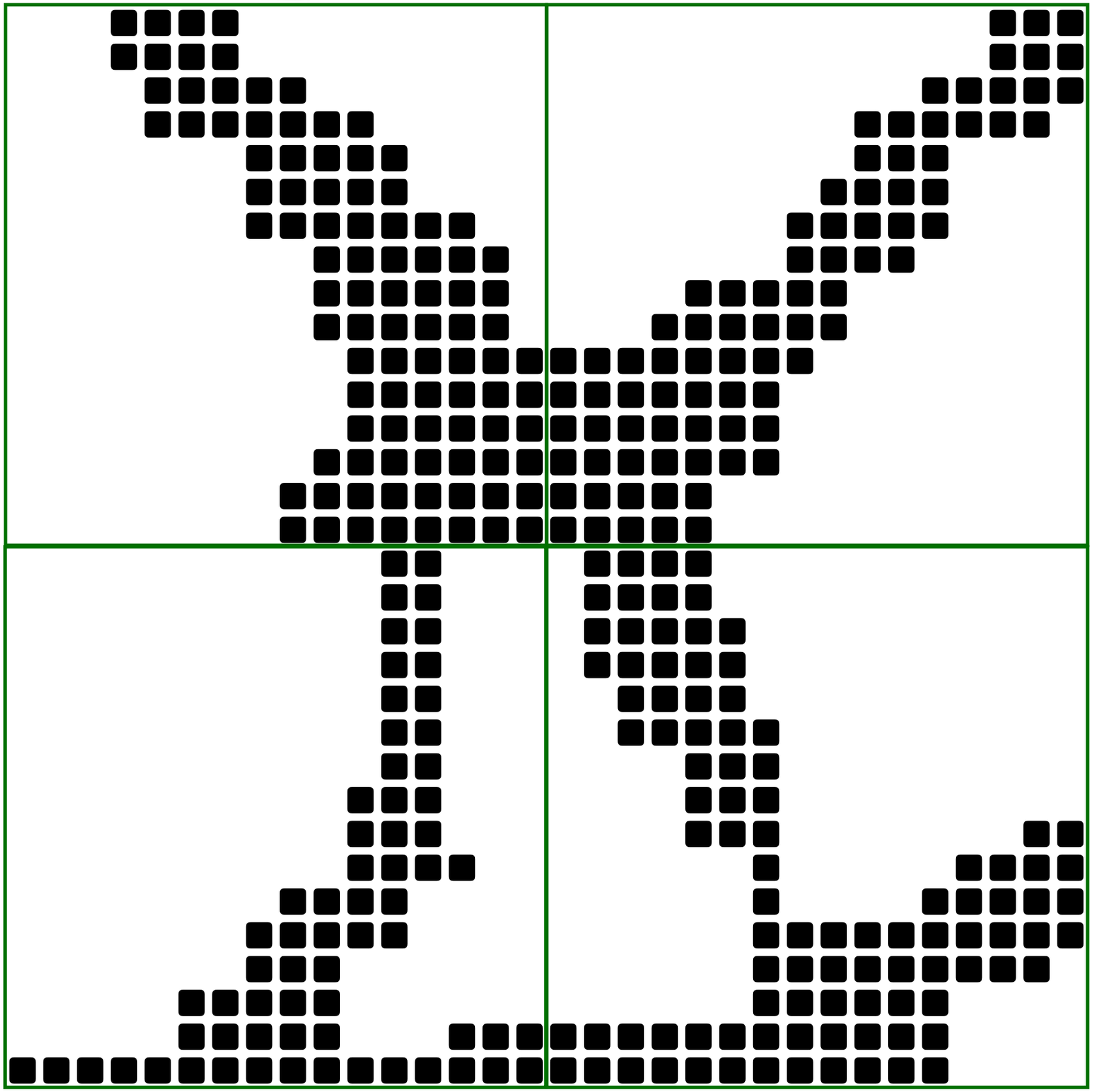
(e.g. covering 10%, 1% or even less of the "dense" grid that they live within)



Sparse grids (intro concepts)

First concept (quadtree in 2D, octree in 3D)

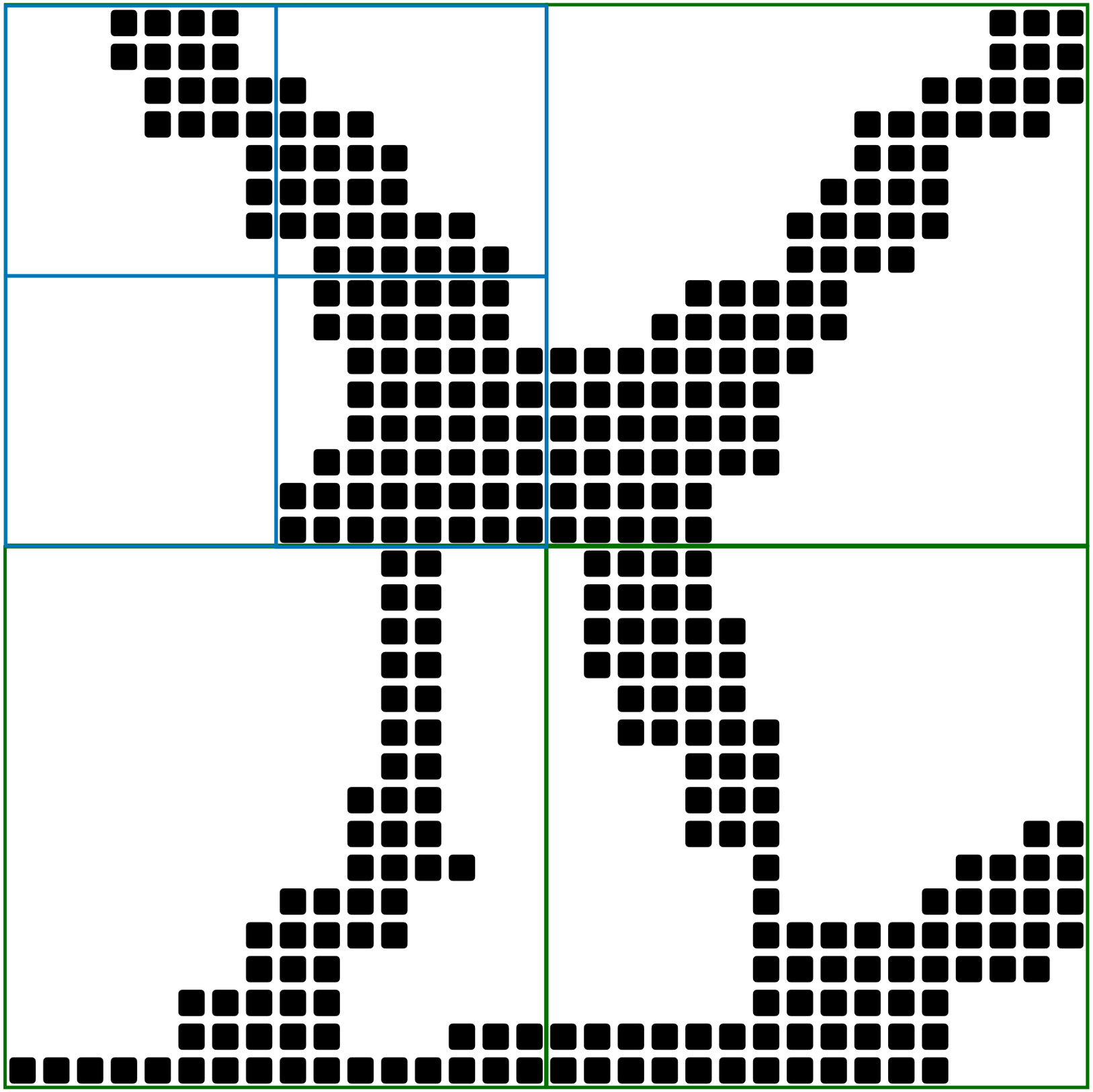
Recursively subdivide each square (cube) of the dense domain into 4 (8) children



Sparse grids (intro concepts)

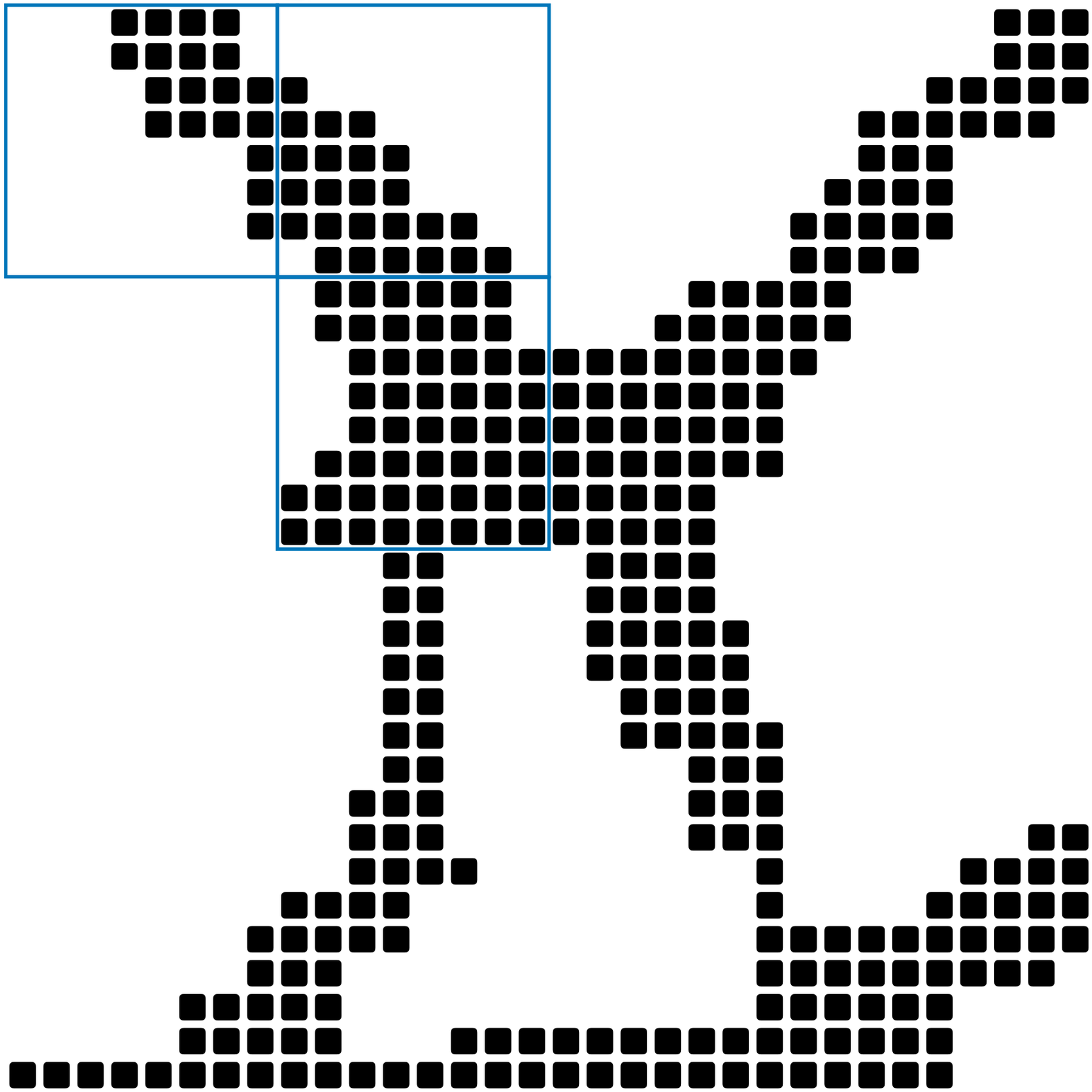
First concept (quadtree in 2D, octree in 3D)

Recursively subdivide each square (cube) of the dense domain into 4 (8) children



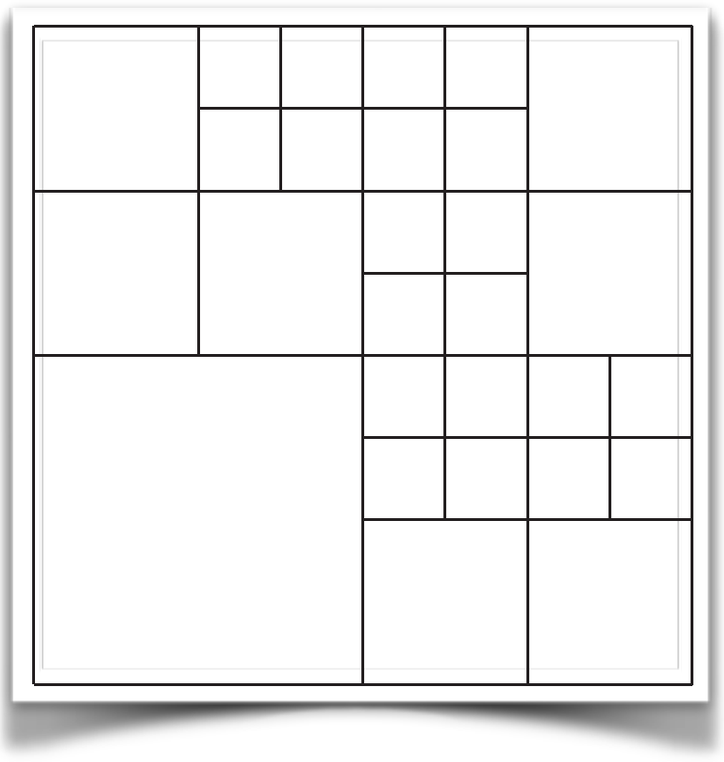
Sparse grids (intro concepts)

In the course of refinement, if any boxes are empty, discard them!

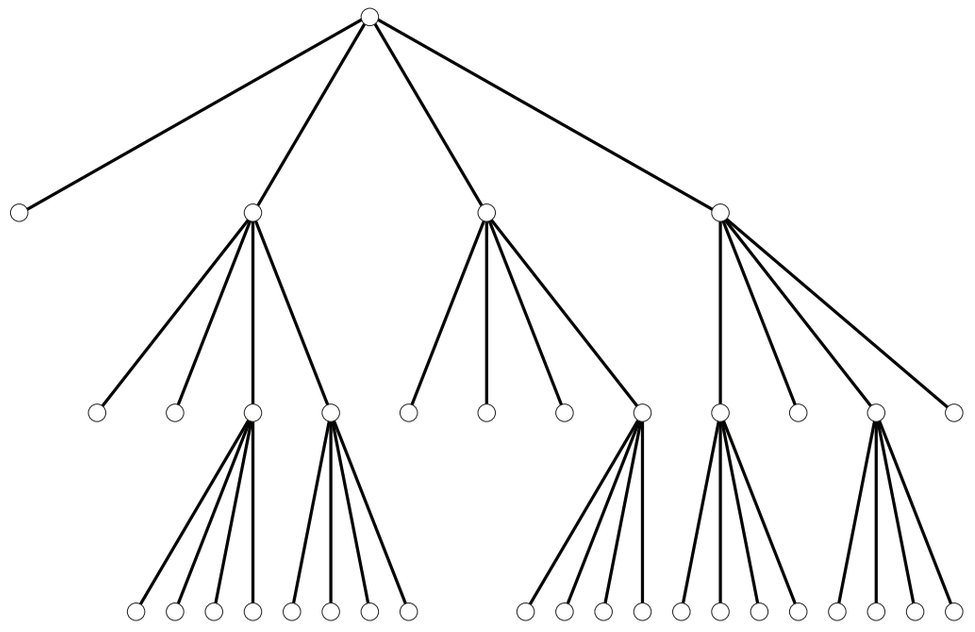


Sparse grids (intro concepts)

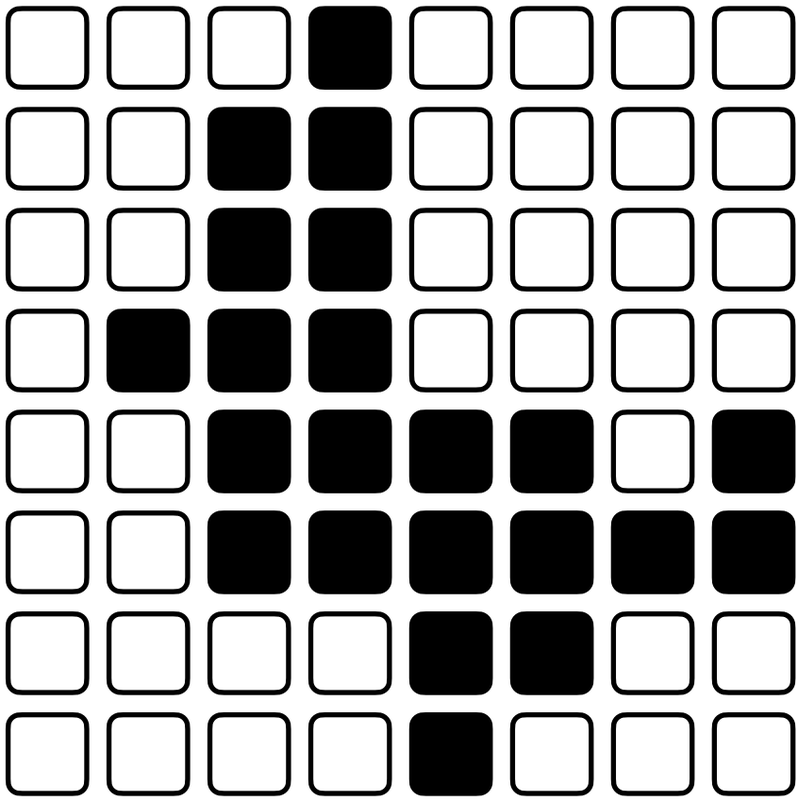
Conceptual
Octree



Underlying
Data Structure

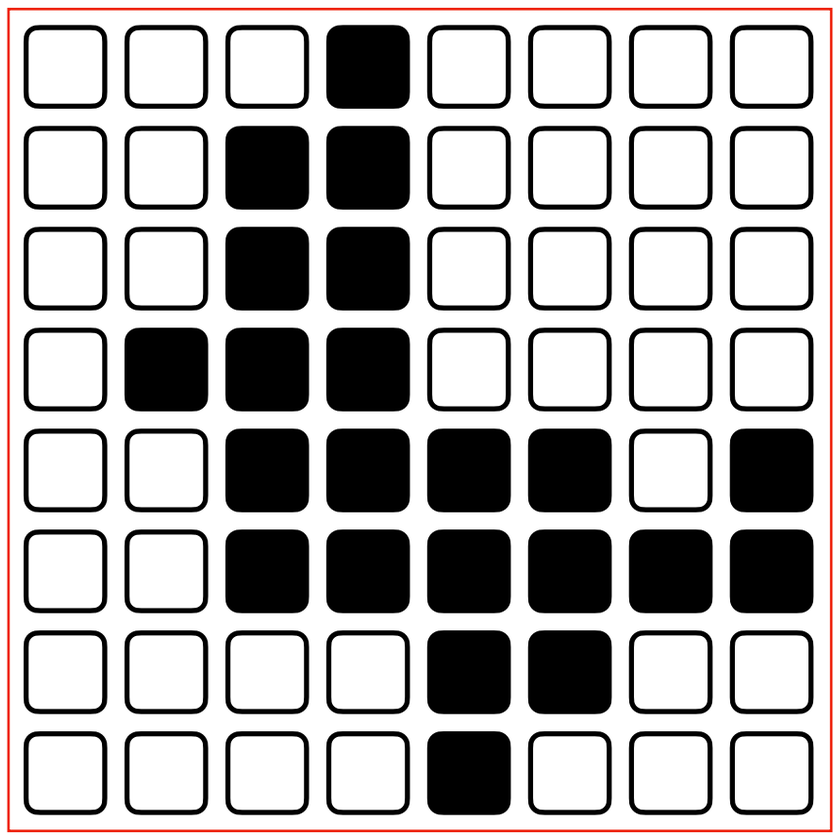


Sparse grids as quadtrees - a specific example



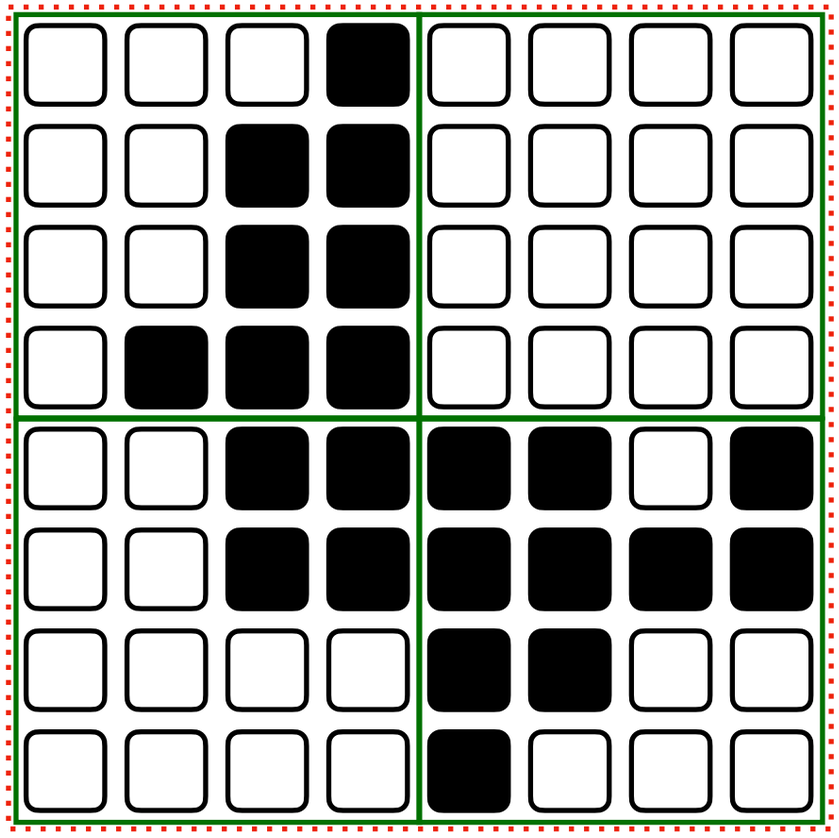
Consider this sparse pattern of occupancy in an 8x8 "host" grid

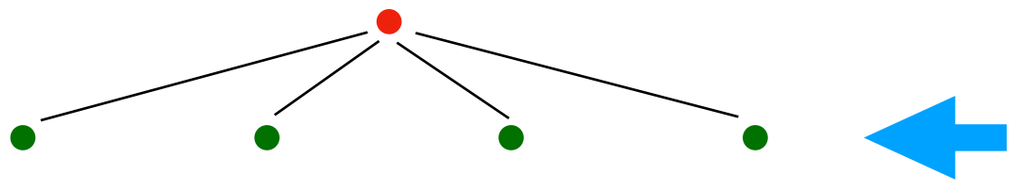
Sparse grids as quadtrees - a specific example



We will progressively build a "tree" structure that will mirror the nesting of rectangular regions in the host grid! This red node (dot) corresponds to the entire 8x8 grid domain

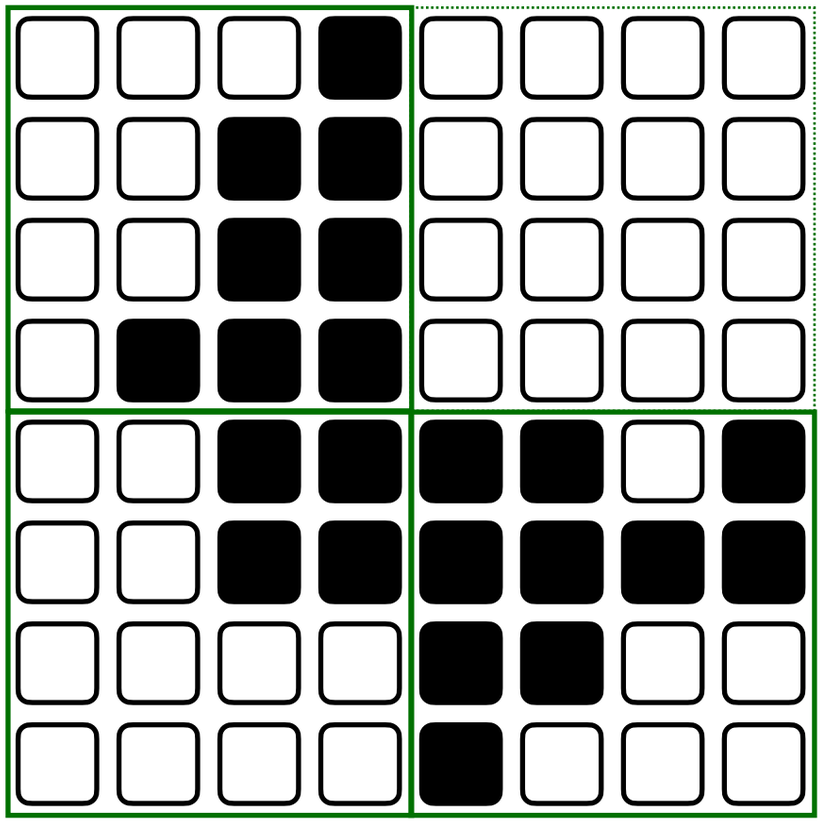
Sparse grids as quadtrees - a specific example

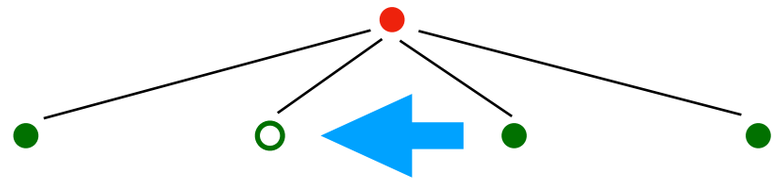




We subdivide the 8x8 box into 4 4x4 sub-boxes. We associate this in the tree by 4 “child” nodes of a tree, that we link to the “root” node (corresponding to the 8x8 box)

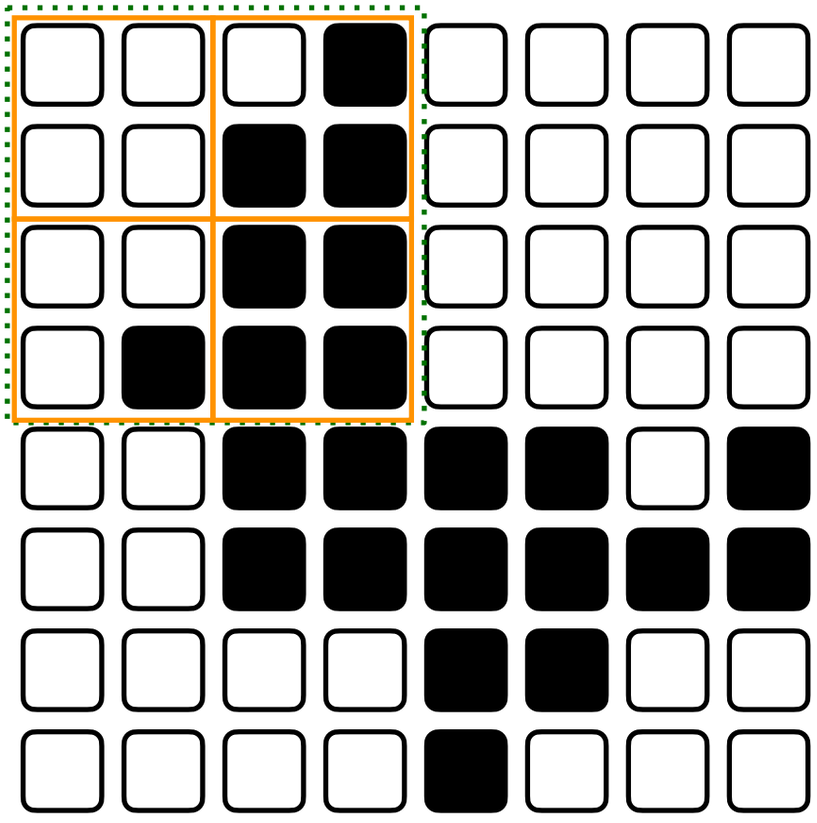
Sparse grids as quadtrees - a specific example

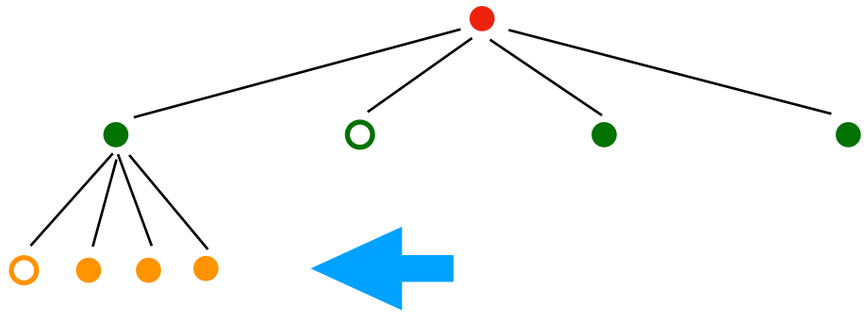




Momentarily forget about the 8x8 box, and focus on the 4x4 ones. We observe that one of them is actually "empty" of any active contents! We show that with a hollowed out node (and will ultimately remove this altogether from the tree)

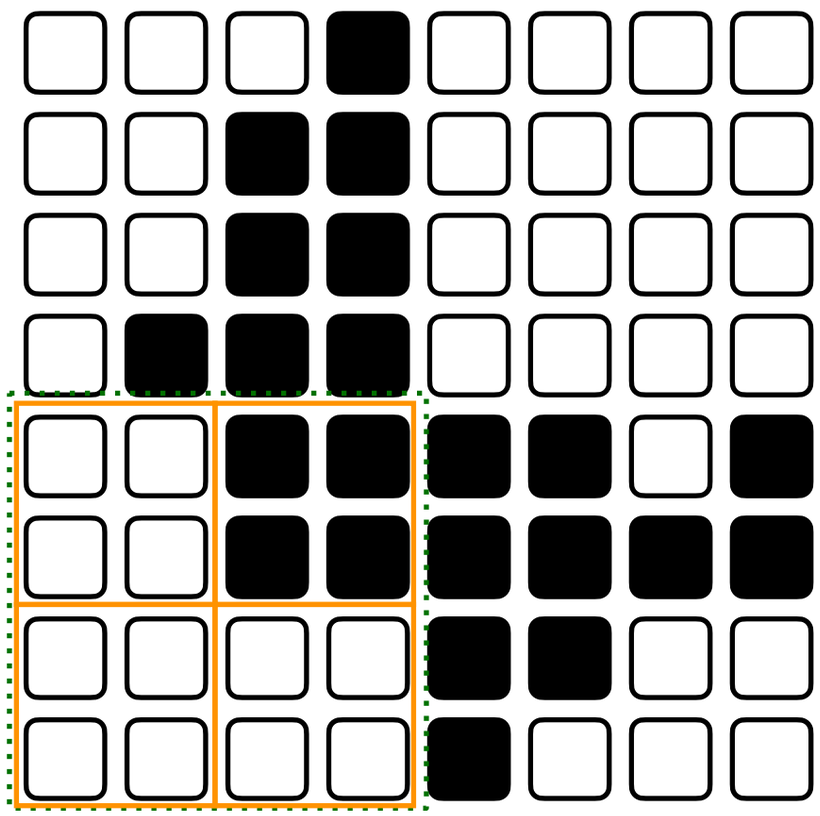
Sparse grids as quadtrees - a specific example

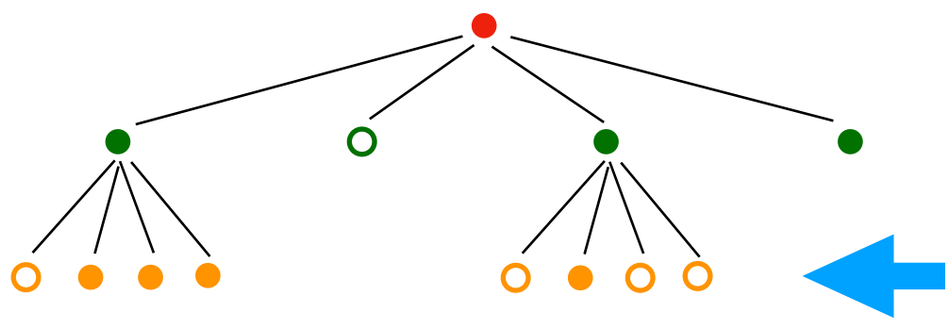




We continue the subdivision of any non-empty 4x4 boxes, this time into 4 2x2 boxes. As before, some of them might be empty, so we mark them with hollow nodes on the tree.

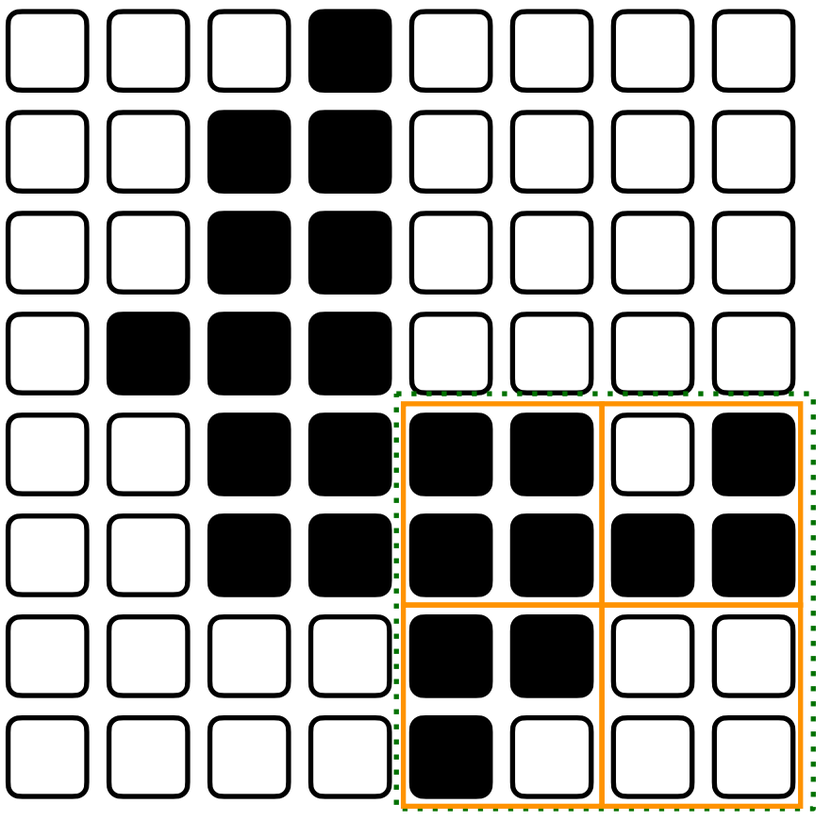
Sparse grids as quadtrees - a specific example

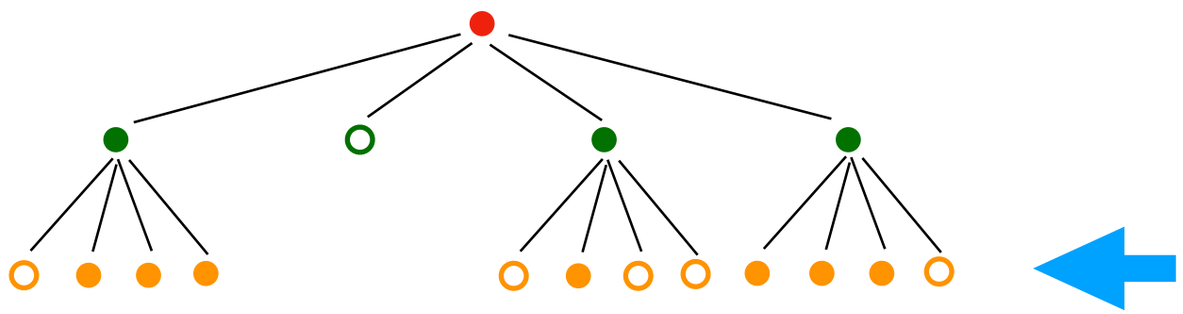




And we repeat this subdivision with any non-empty 4x4 boxes that remain in the tree structure

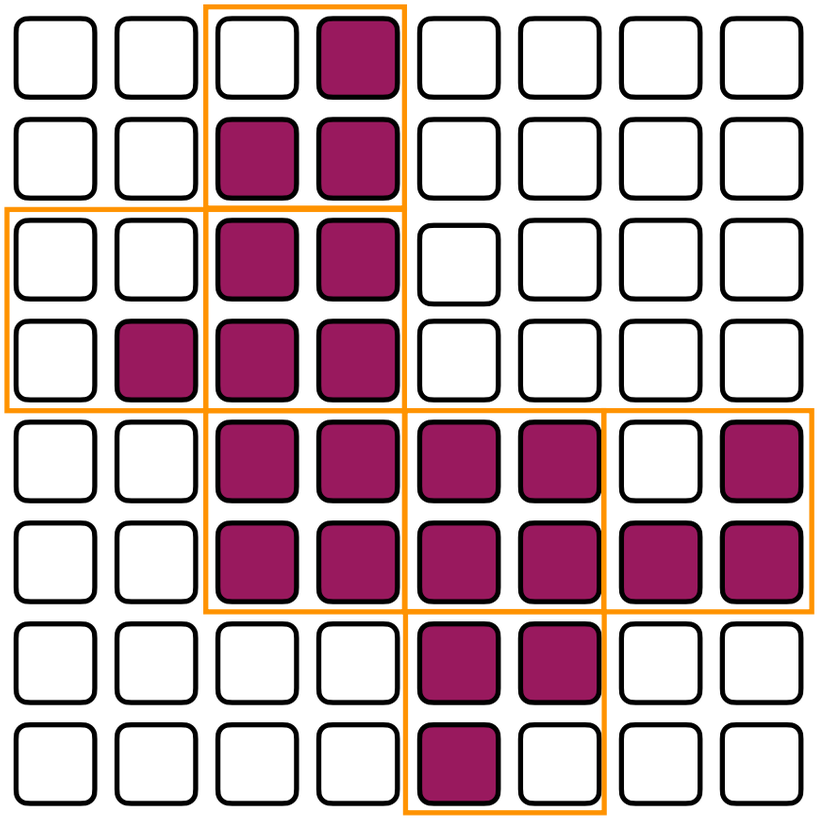
Sparse grids as quadtrees - a specific example

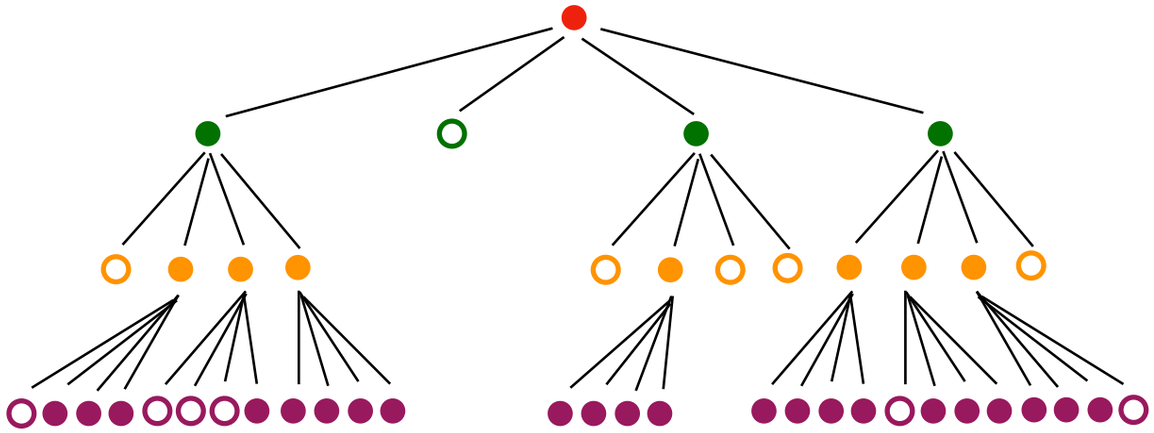




And we repeat this subdivision with any non-empty 4x4 boxes that remain in the tree structure

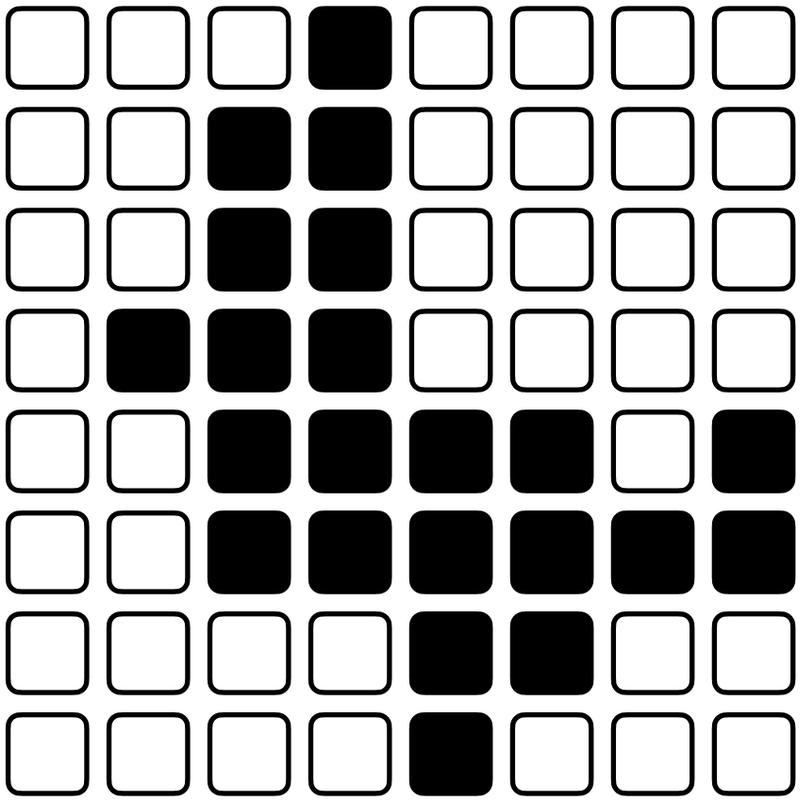
Sparse grids as quadtrees - a specific example

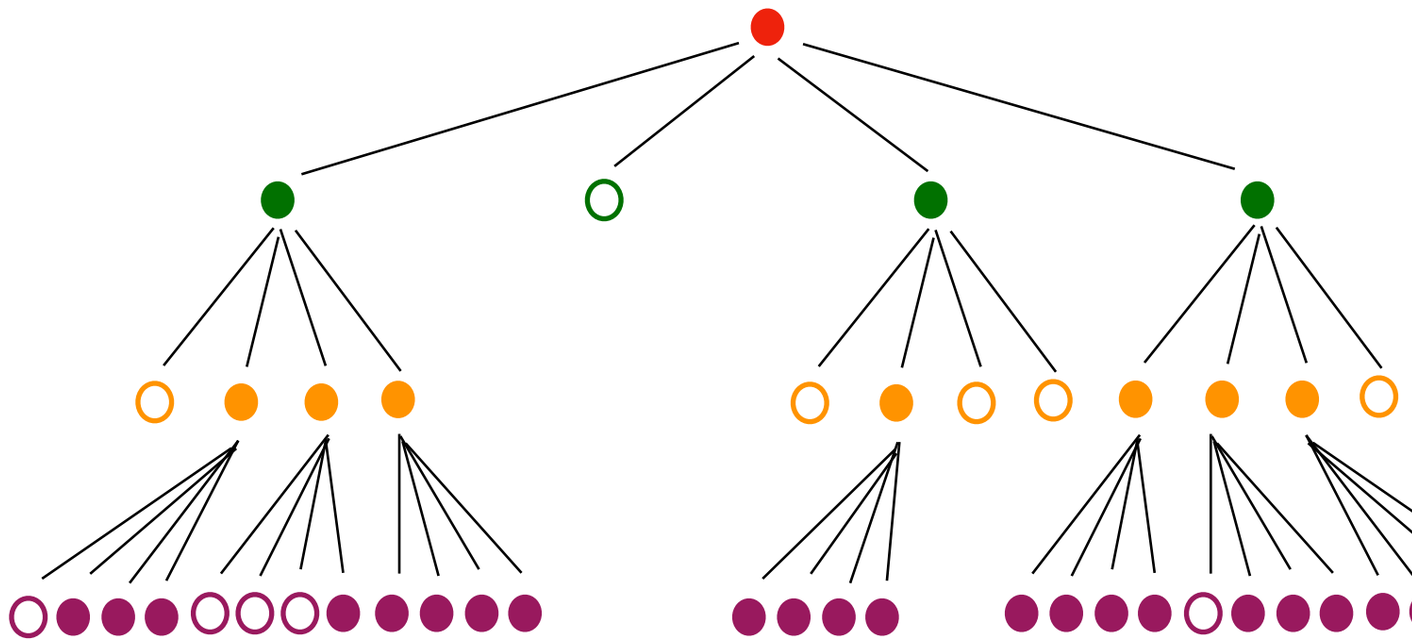




And, finally, we even subdivide the 2x2 boxes into 4 1x1 boxes (now each corresponding to just a single voxel from our sparse grid)

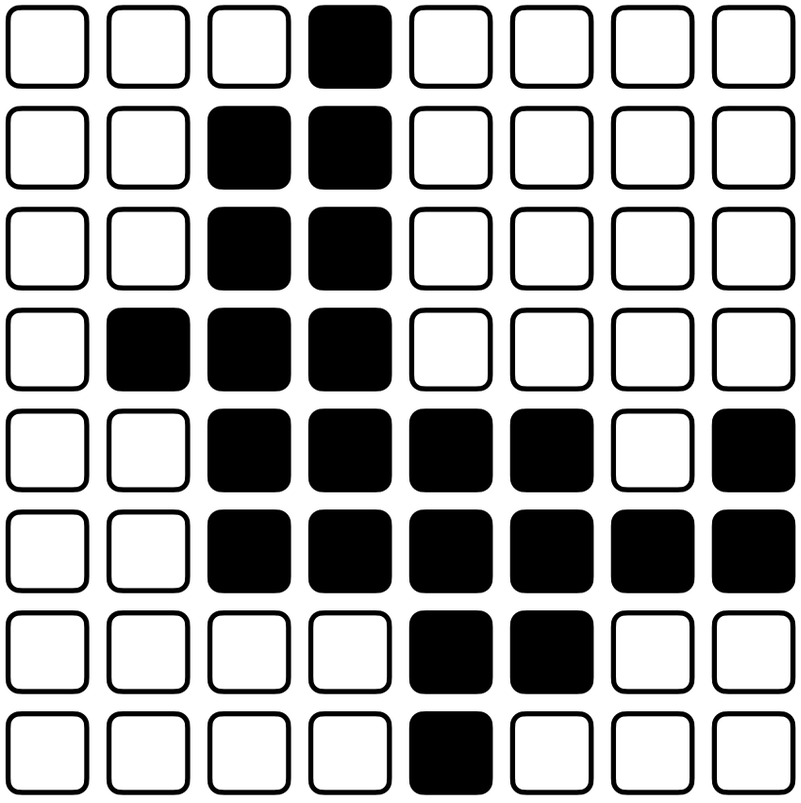
Sparse grids as quadtrees - a specific example

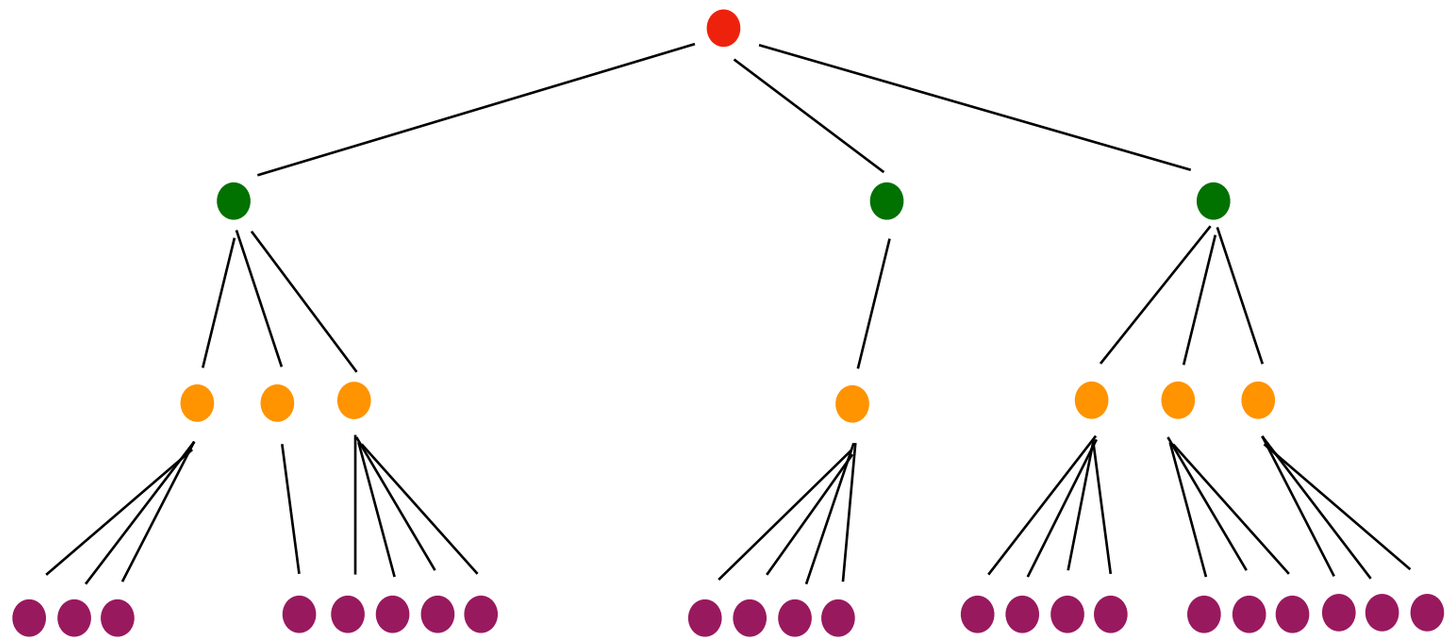




At the end of this process, every "active" voxel in the sparse grid we started with, is represented with exactly one "leaf" node in the tree representation we constructed!

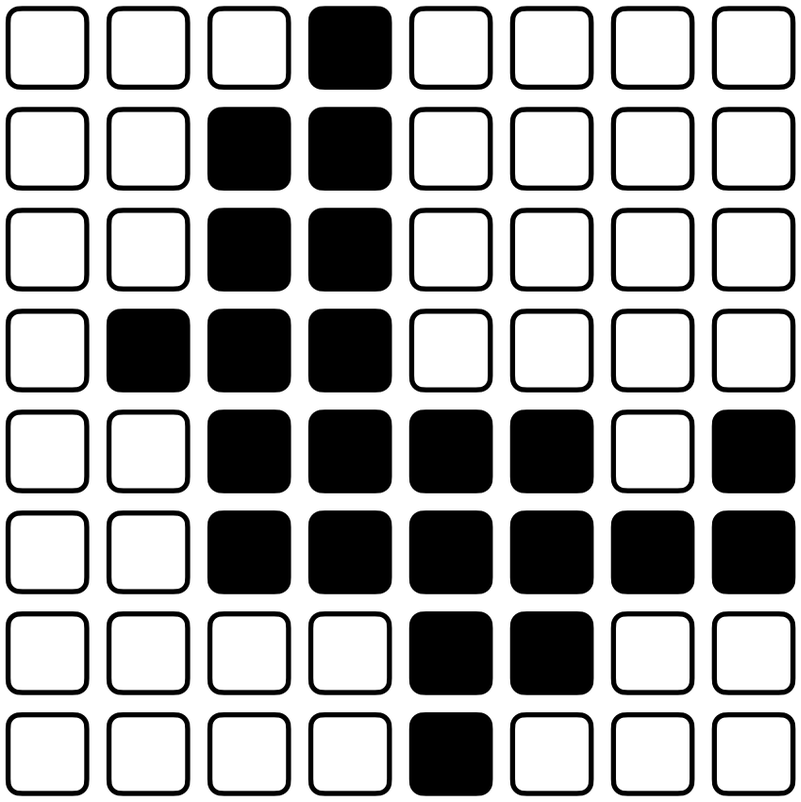
Sparse grids as quadtrees - a specific example

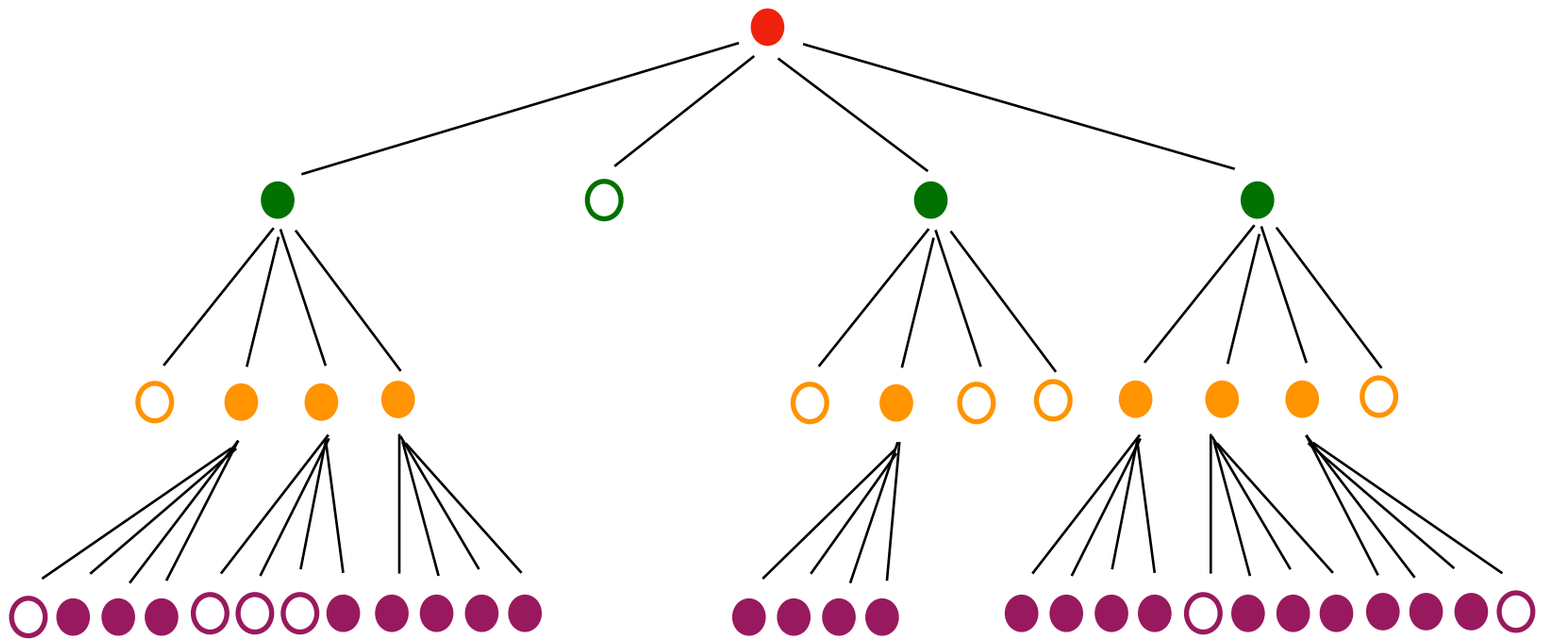




*We could also consider that the tree doesn't actually include the "hollow" nodes, we could prune those away. In this case, the leaves are exactly as many as the active voxels. We also see that even in the worst case scenario, we would not be using more than N*logN nodes in the tree to store N active voxels (In practice, it's really O(N))*

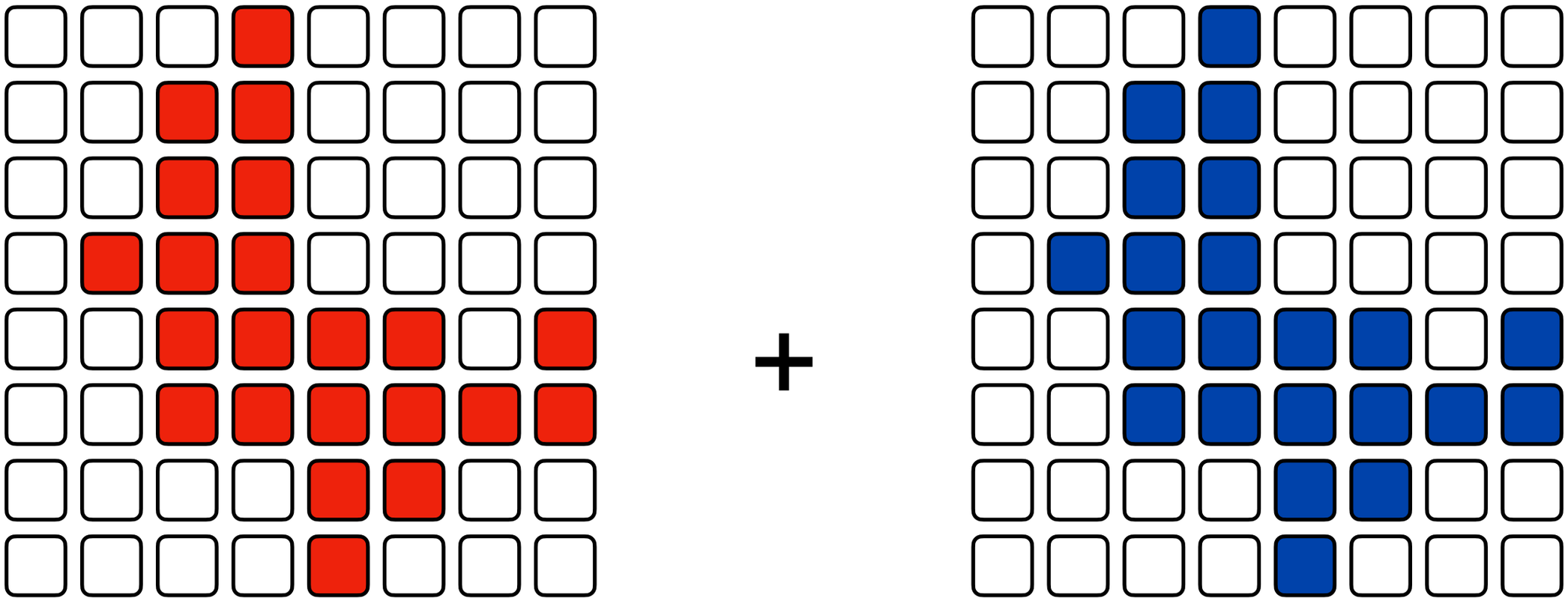
Sparse grids as quadtrees - a specific example





We might stick with this representation that shows "terminal" nodes however; this has the benefit of every node having <u>exactly</u> 4 children in all cases

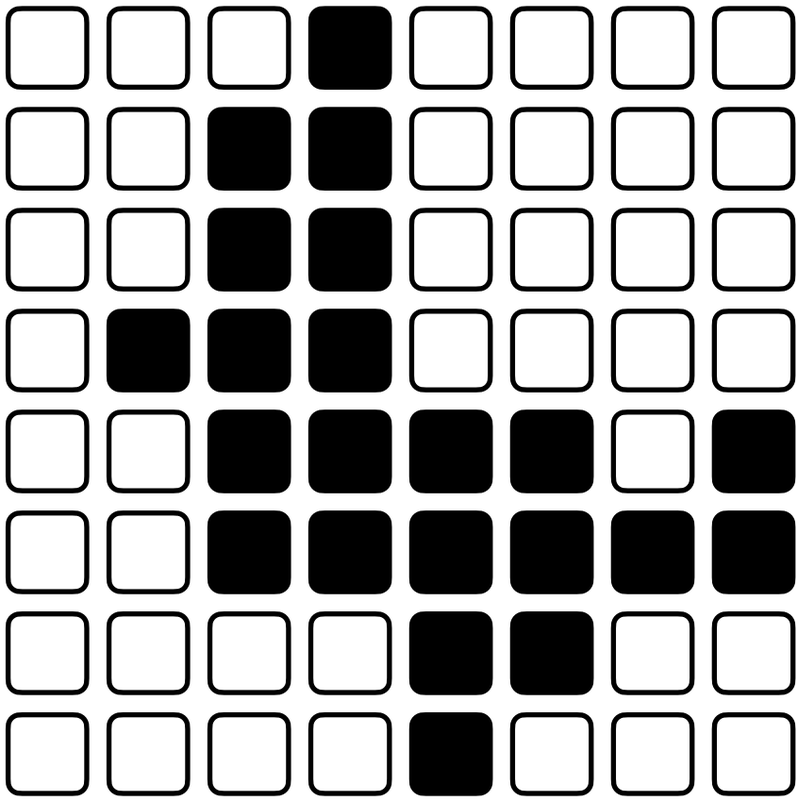
Sparse grids as quadtrees - a specific example

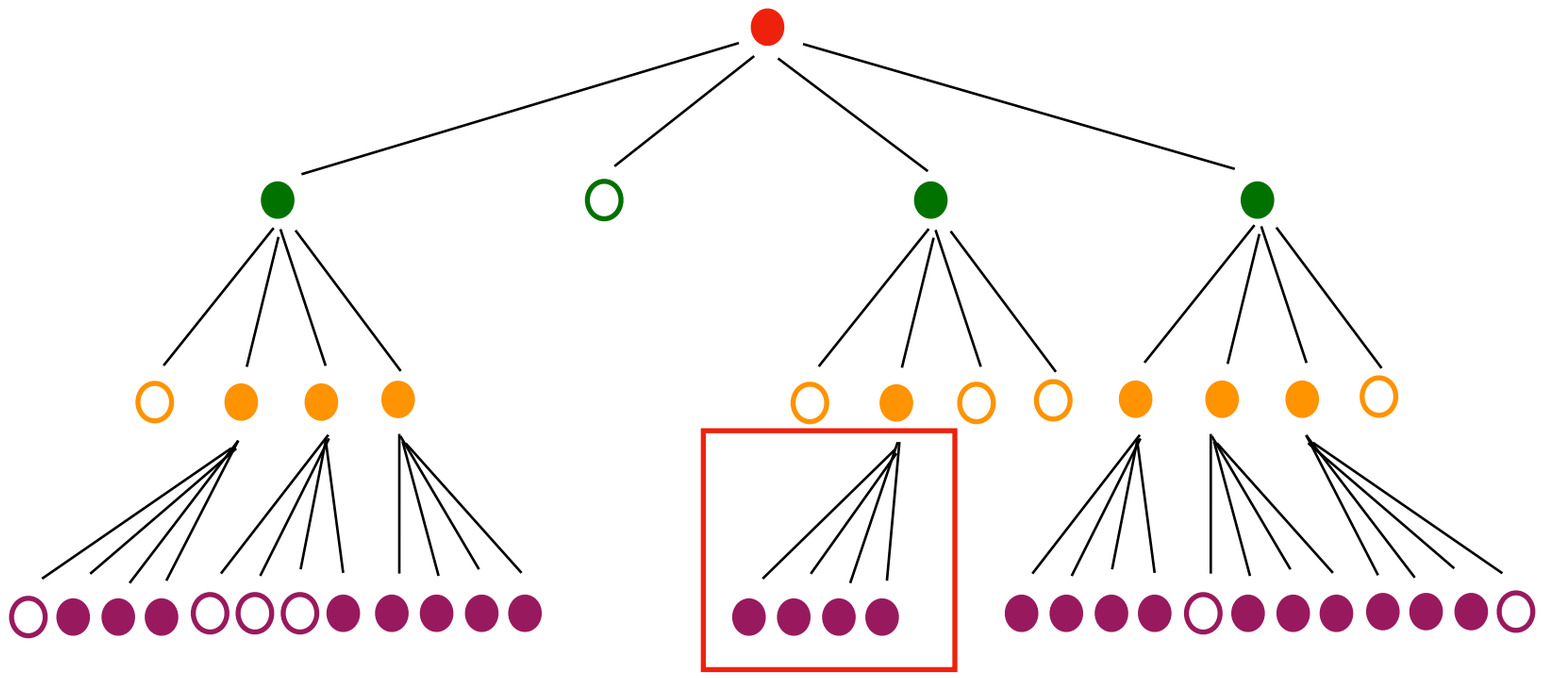


When considering how we operate on such sparse representations, it's useful to concentrate on the 2 types of operations that we used in the Laplacian Solver (they're surprisingly ubiquitous and representative of typical use cases!)

The first type of operation is a point-wise operation, like an addition or SAXPY on an index-by-index basis. We are assuming that the two sparse arrays/grids we are adding (or operating on) have exactly the same sparsity pattern (this is the typical case)

Sparse grids as quadtrees - a specific example

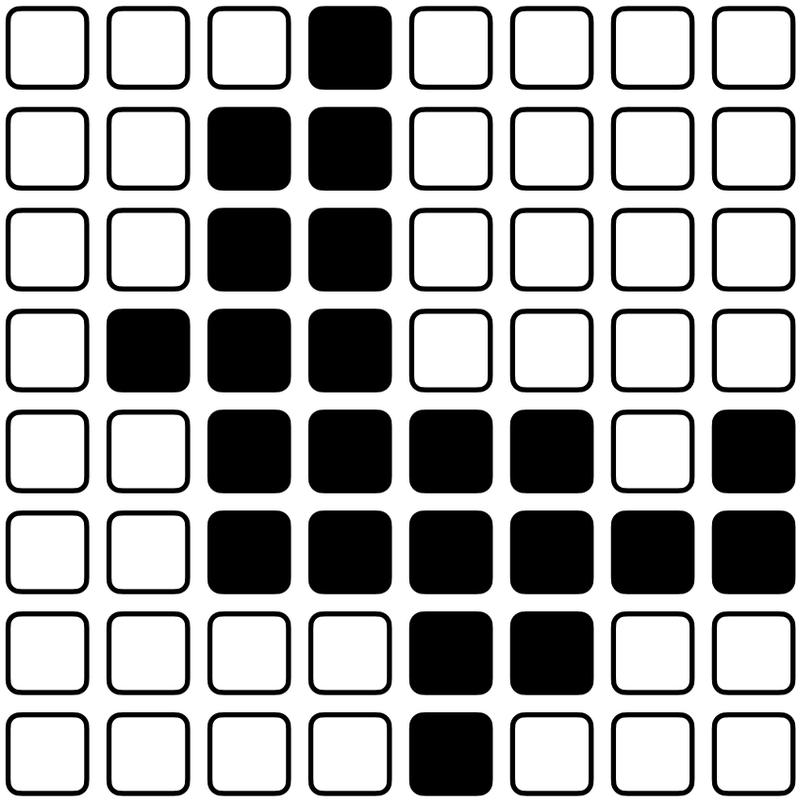


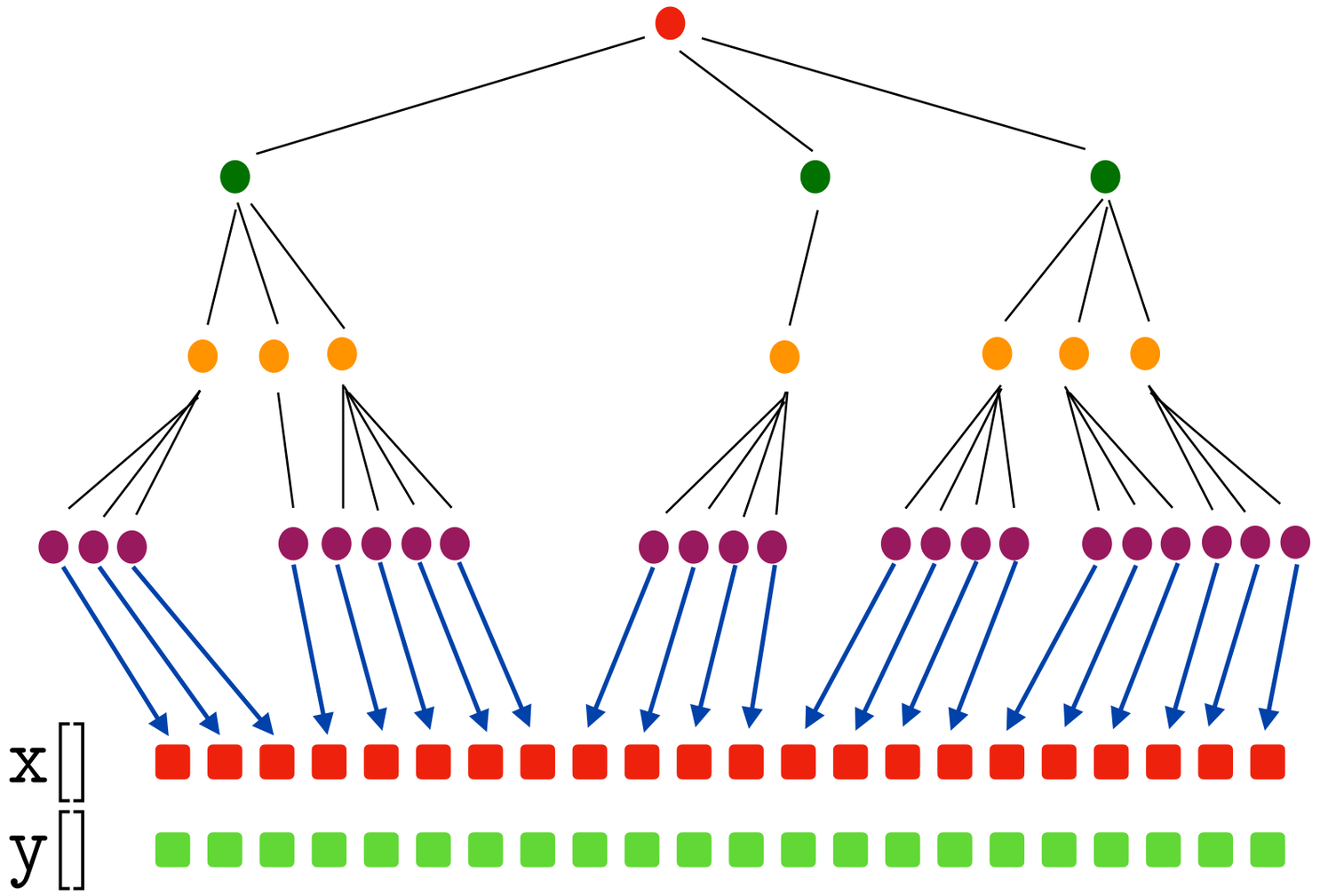


The possible problem with efficient streaming (add/saxpy) operations is that data that is stored on leaf nodes of the tree are typically heap-allocated, and accessed through pointers (inefficient caching and prefetching)

```
struct Leaf{
   float value00;
   float value01;
   float value10;
   float value11;
}
```

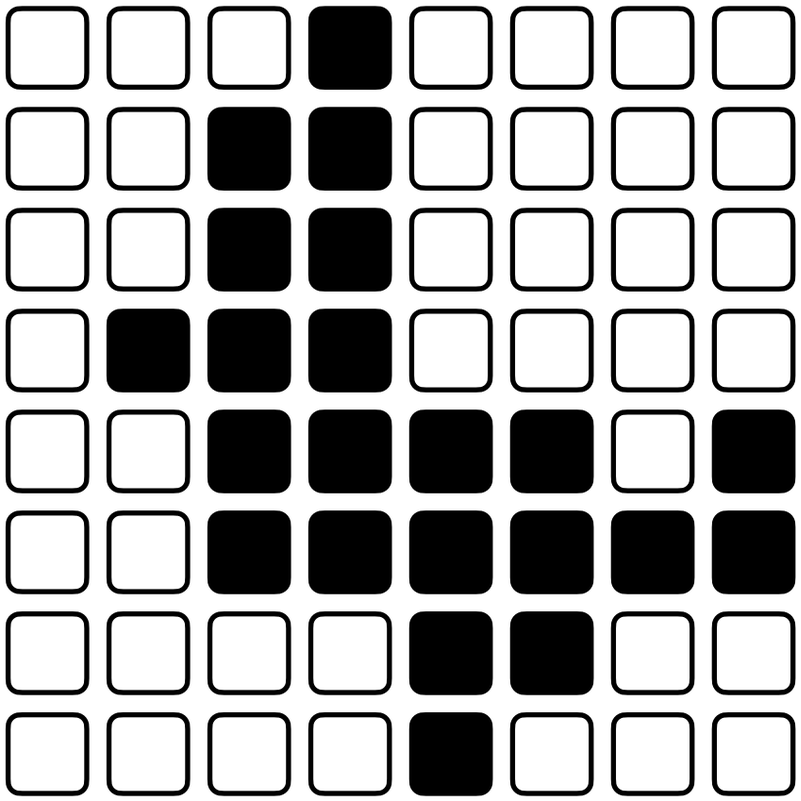
Sparse grids as quadtrees - a specific example

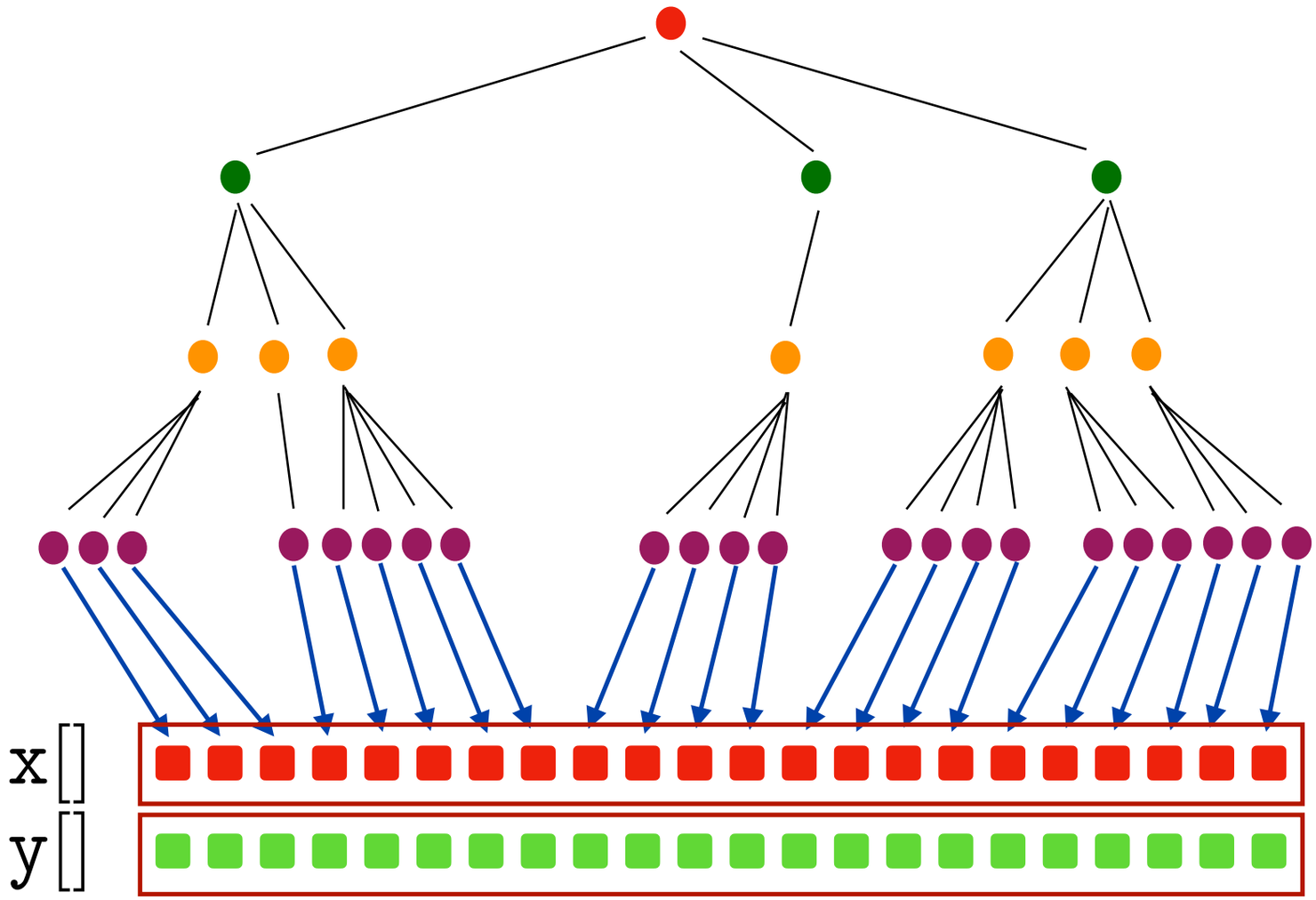




A possible remedy: At leaf nodes don't store the values, but pointers to a sequential/ linearized array that stores a "flattened" representation of all the leaf-level values

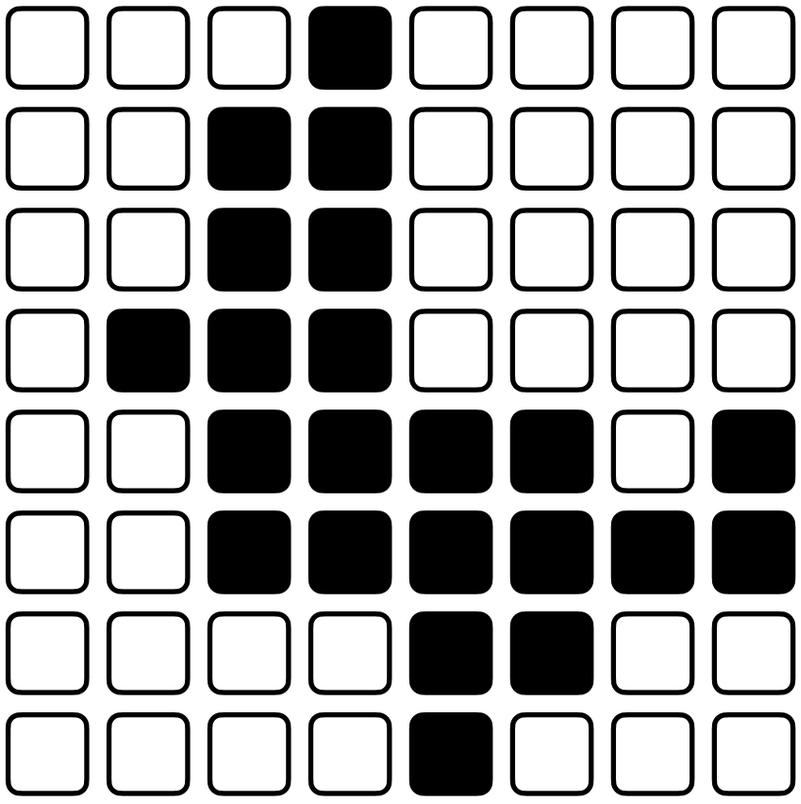
Sparse grids as quadtrees - a specific example

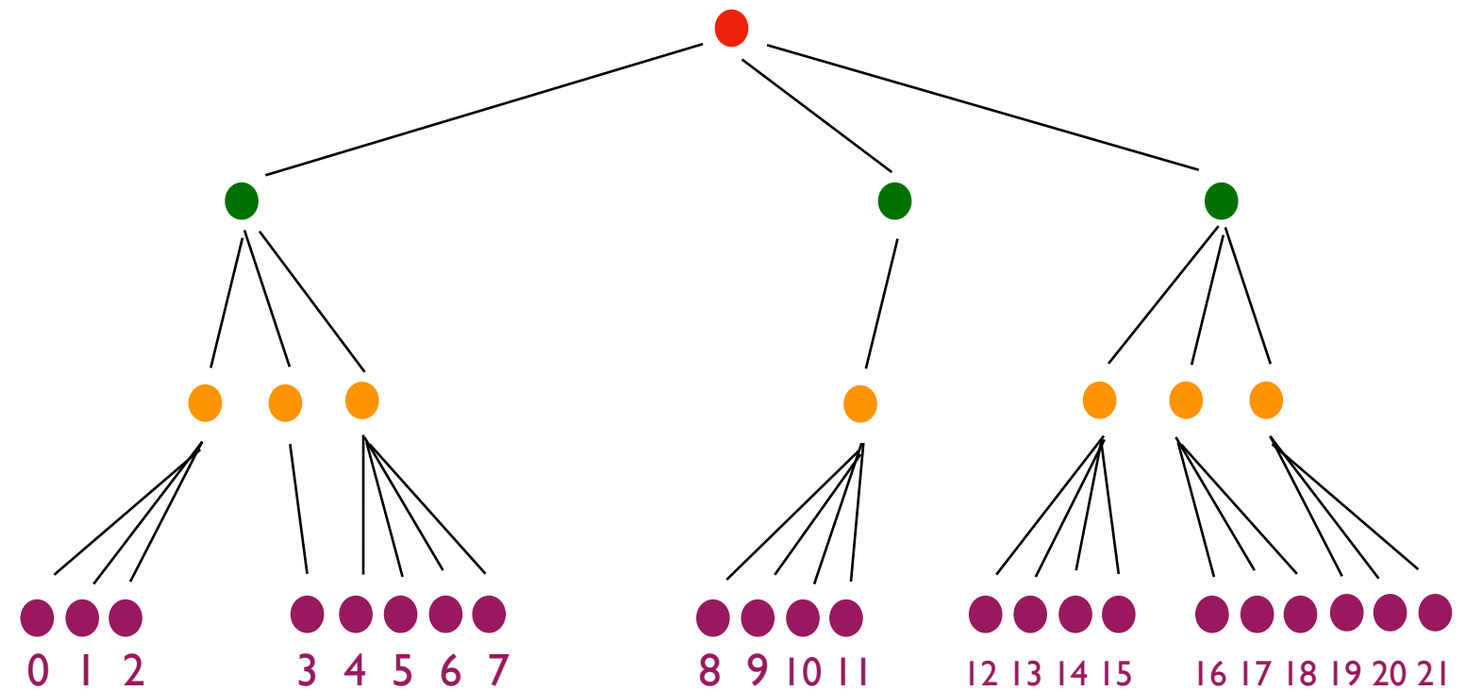


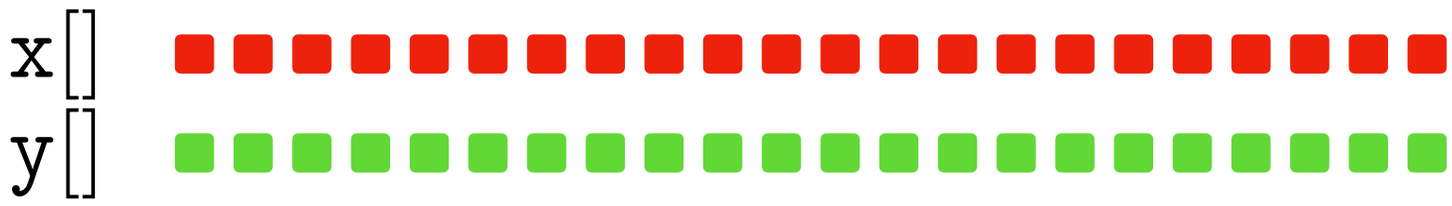


In this case, streaming (point-wise) operations can simply proceed by operating on the linearized/flattened array representation of the leaf-level data

Sparse grids as quadtrees - a specific example

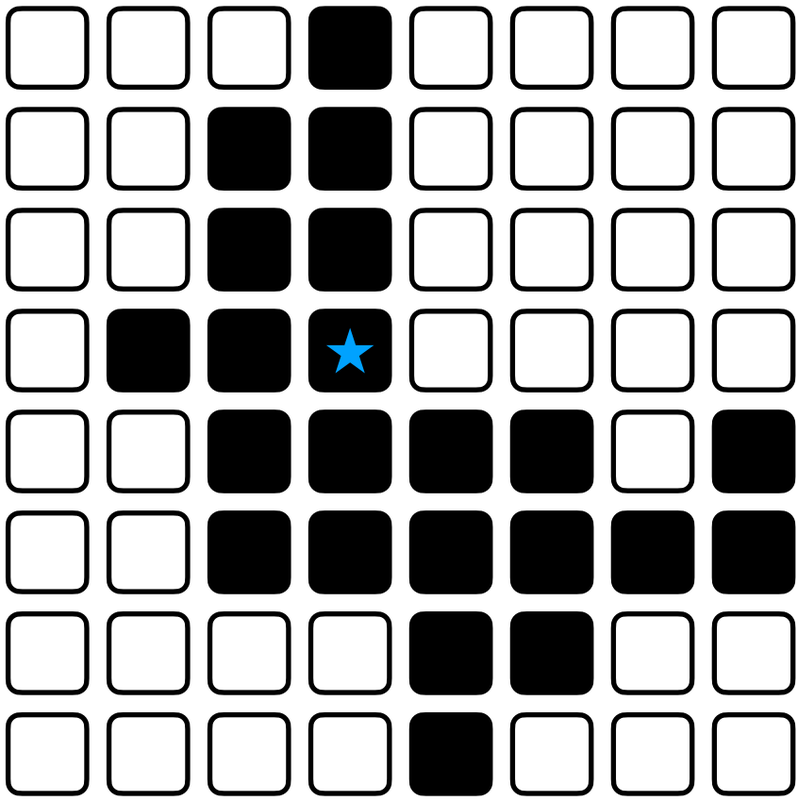


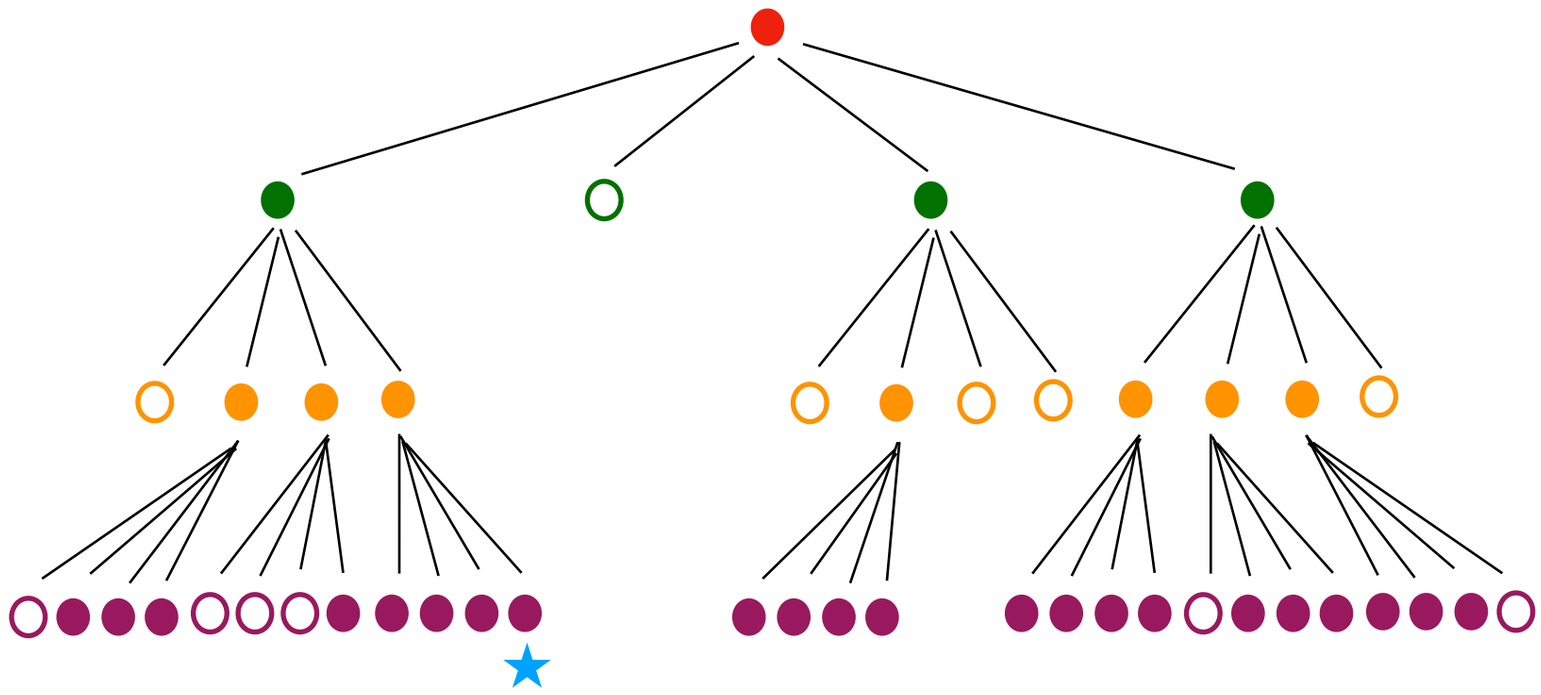




Another interpretation: Instead of the tree holding the actual data, it holds "Indices" into a separate linearized array storage. This concept (which we will call an Index-Grid) will be used a lot in our specific approach!

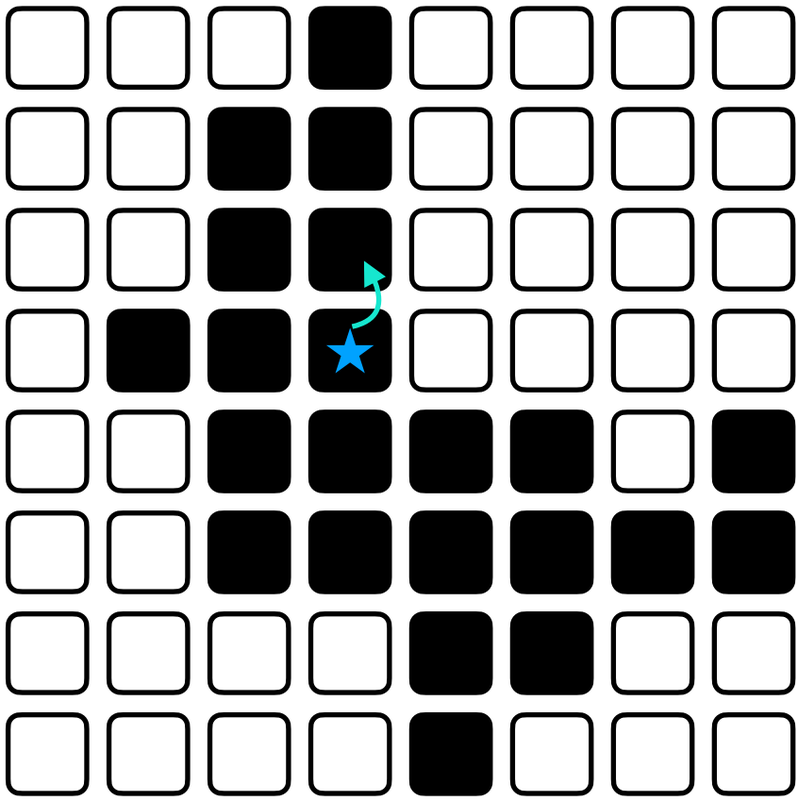
Sparse grids as quadtrees - a specific example

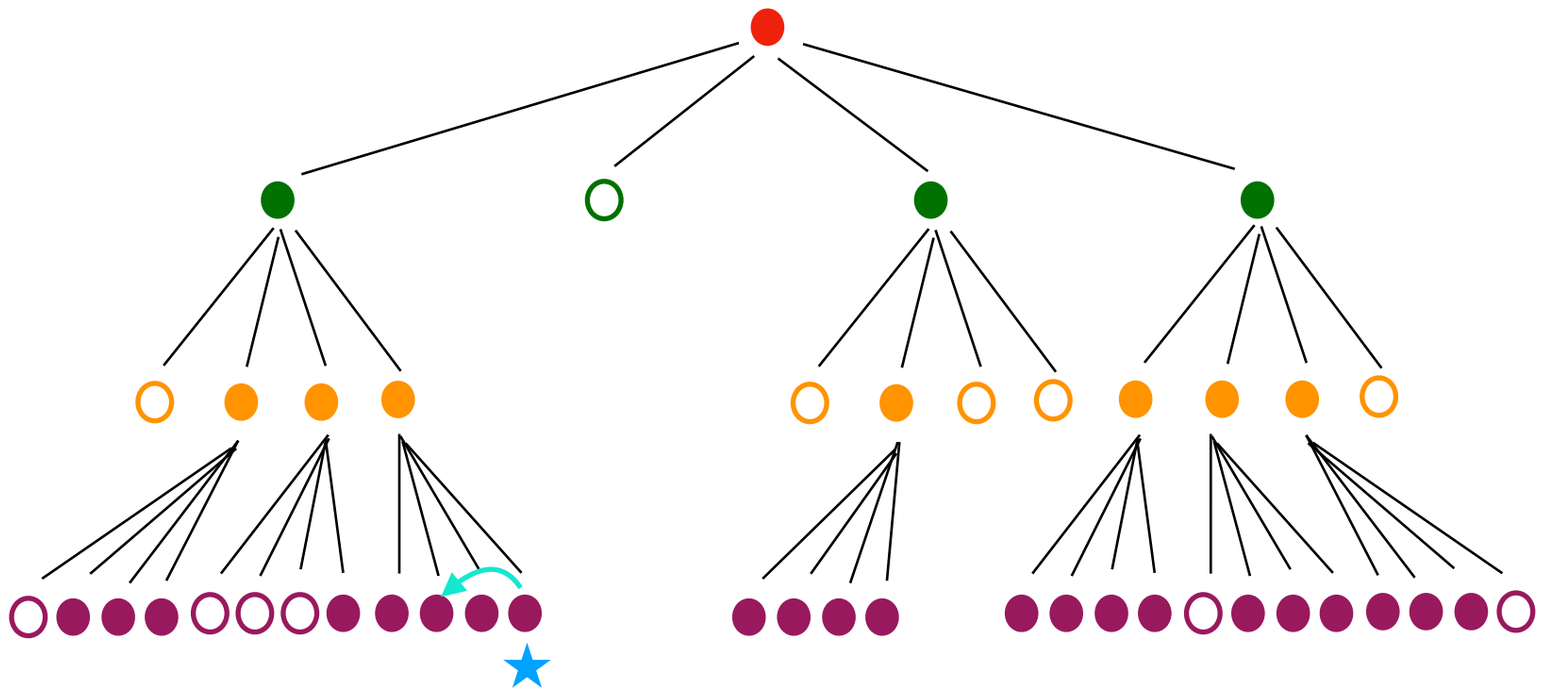




The second type of operation we care to optimize is a stencil application. Consider here just the case of the Laplacian!

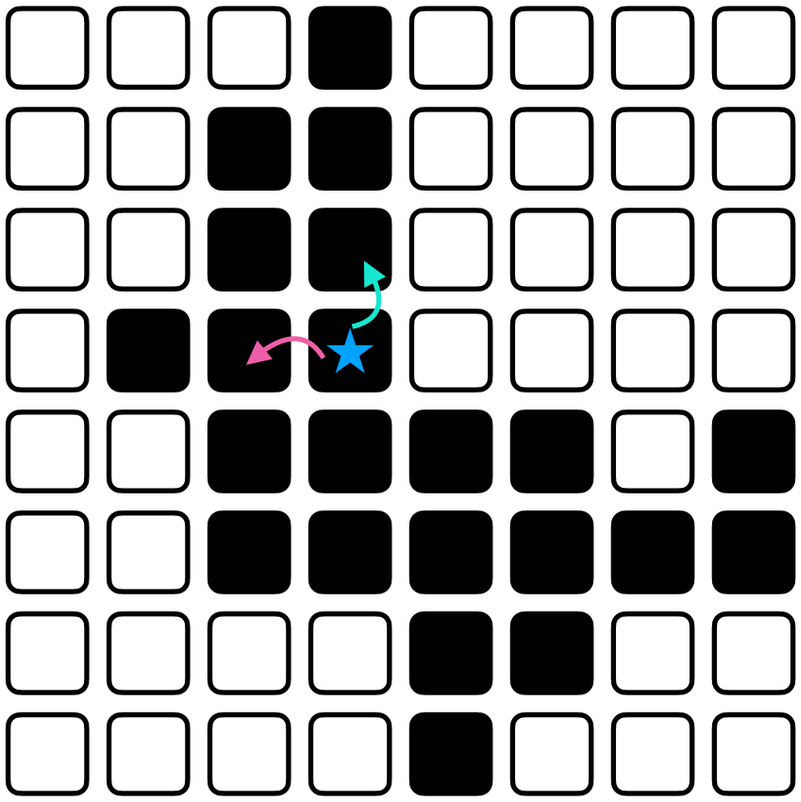
Sparse grids as quadtrees - a specific example

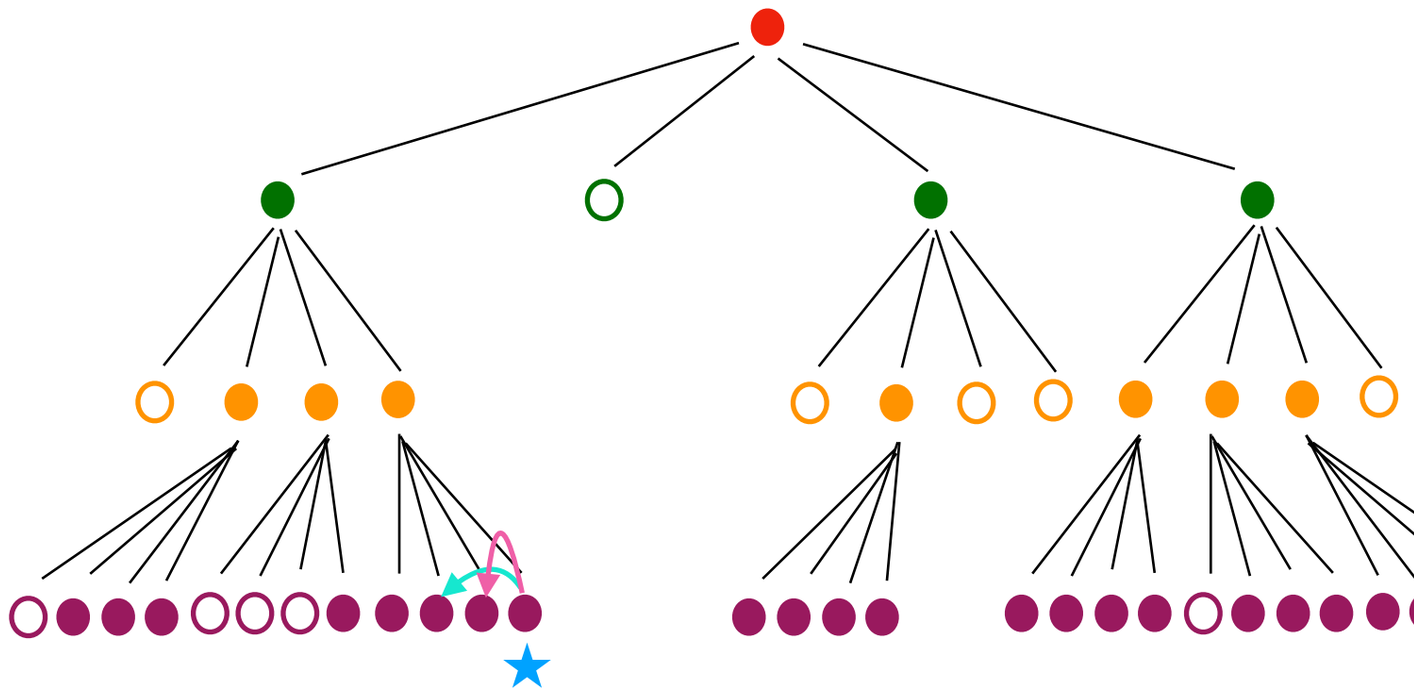




The second type of operation we care to optimize is a stencil application. Consider here just the case of the Laplacian!

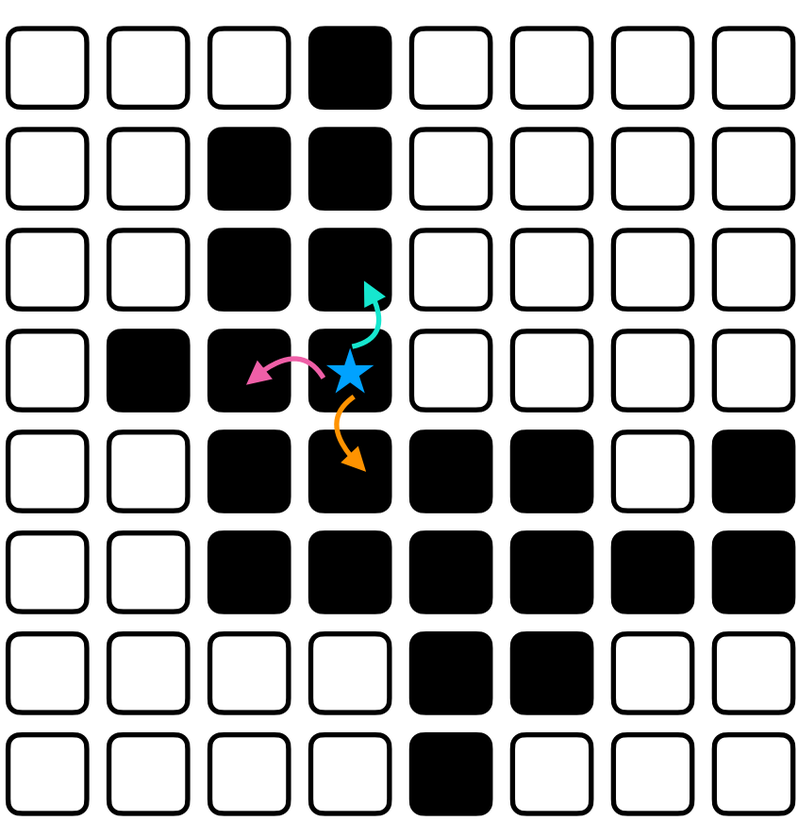
Sparse grids as quadtrees - a specific example

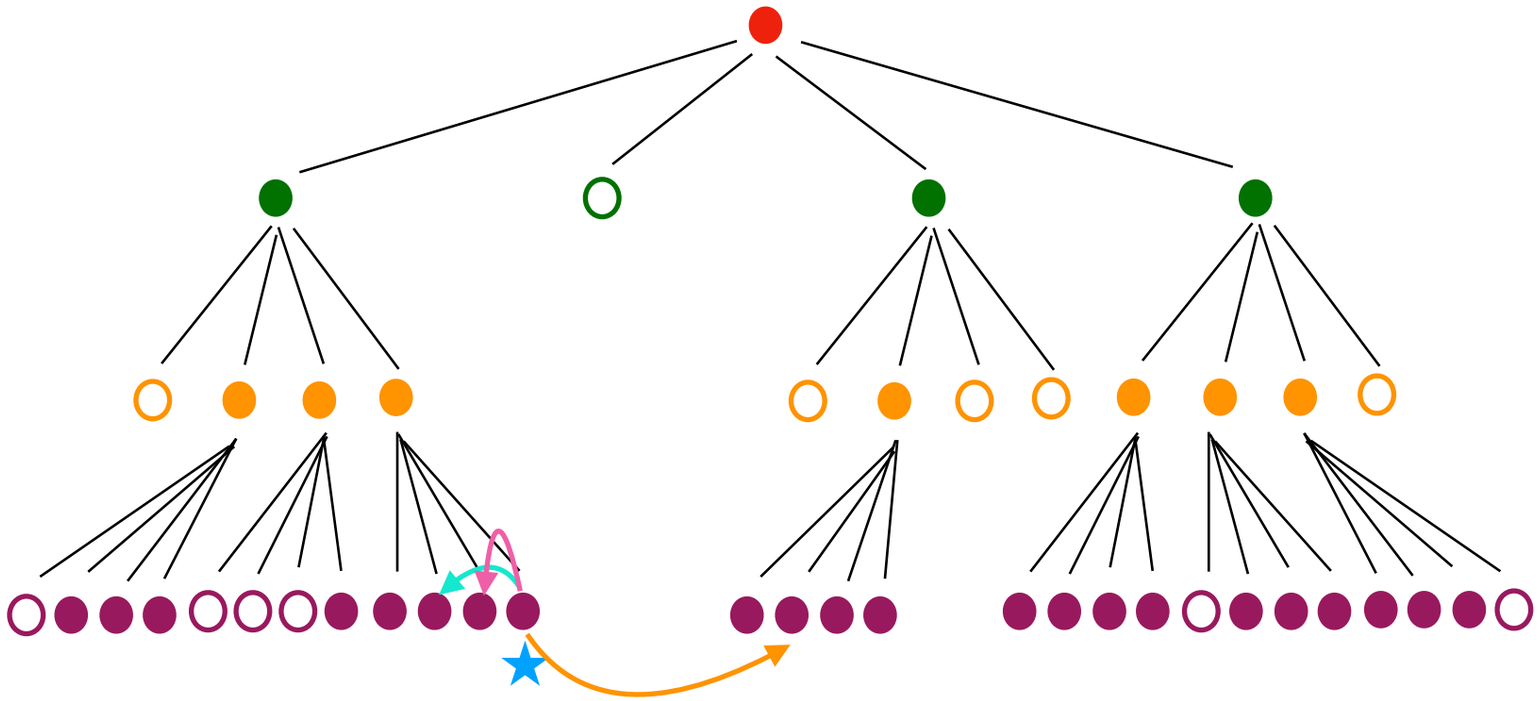




The second type of operation we care to optimize is a stencil application. Consider here just the case of the Laplacian!

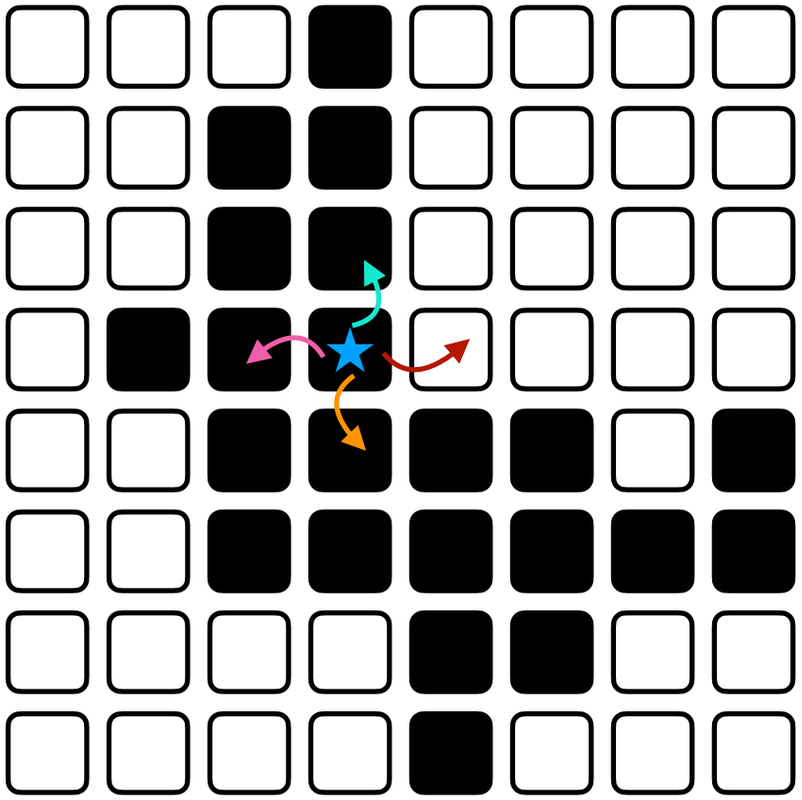
Sparse grids as quadtrees - a specific example

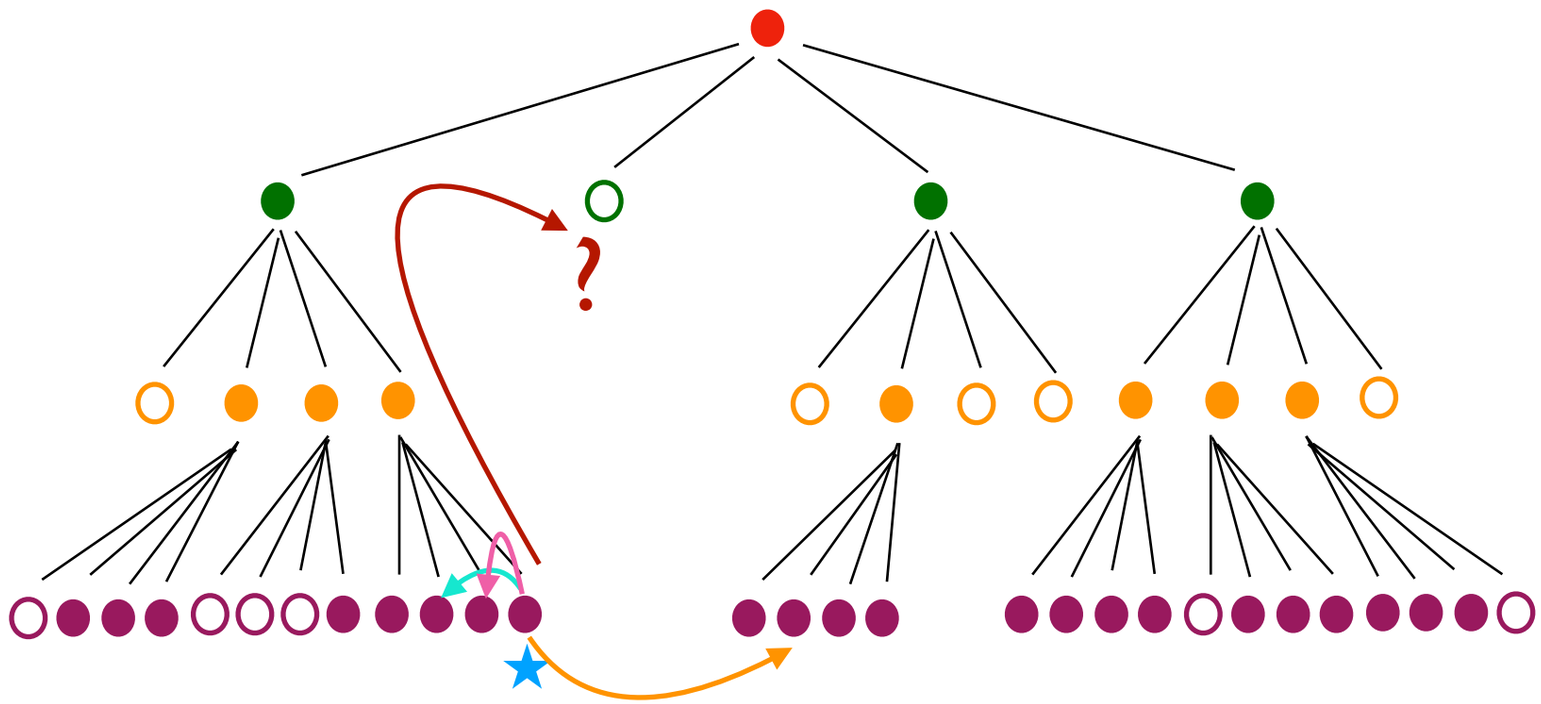




The second type of operation we care to optimize is a stencil application. Consider here just the case of the Laplacian!

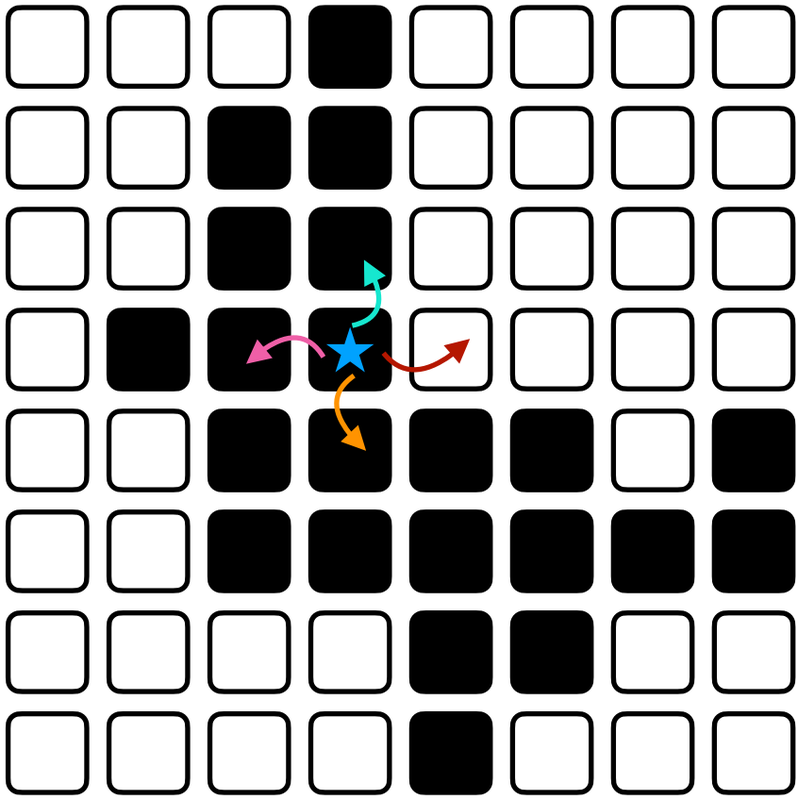
Sparse grids as quadtrees - a specific example

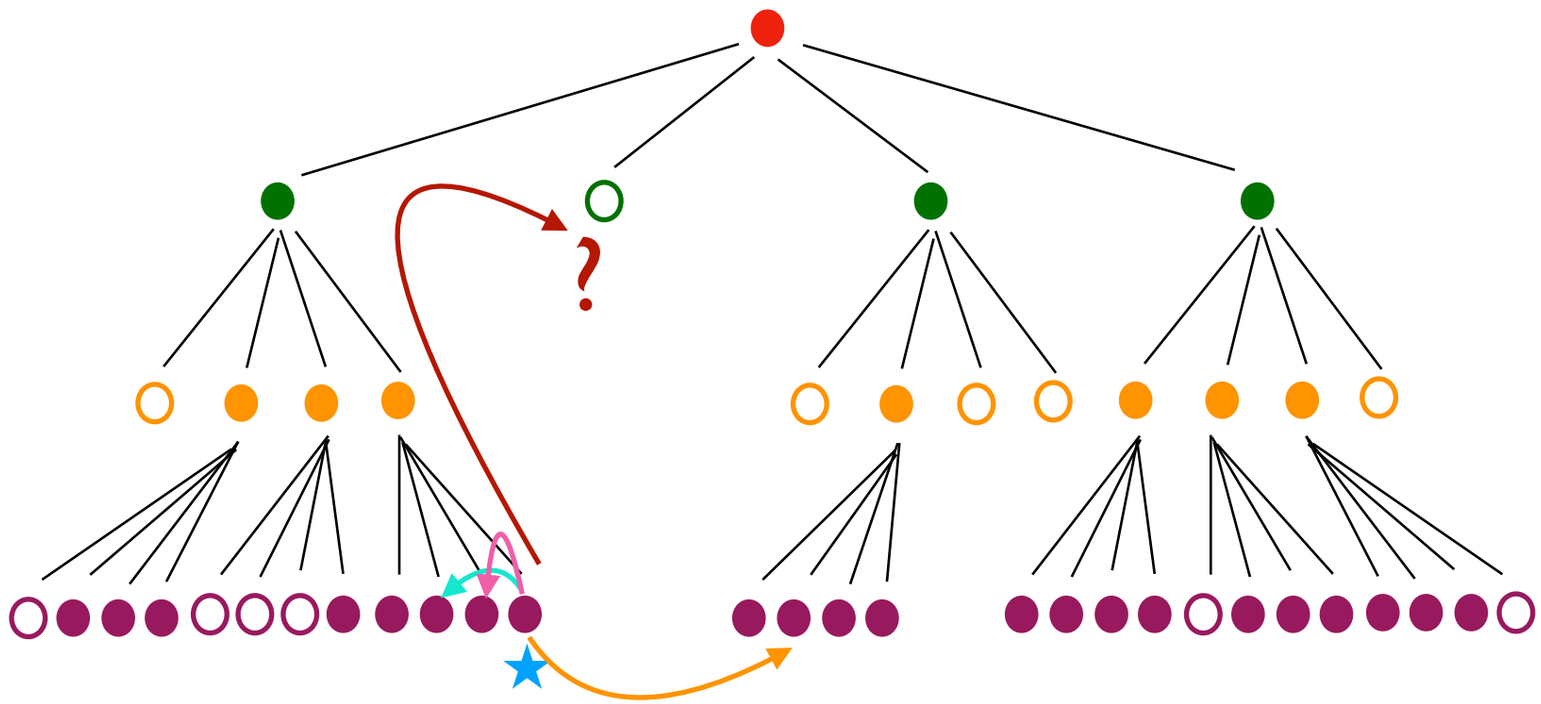




Accessing neighbors is dramatically more expensive than dense grids! It requires an expensive (pointer dereferencing) and unpredictable tree traversal

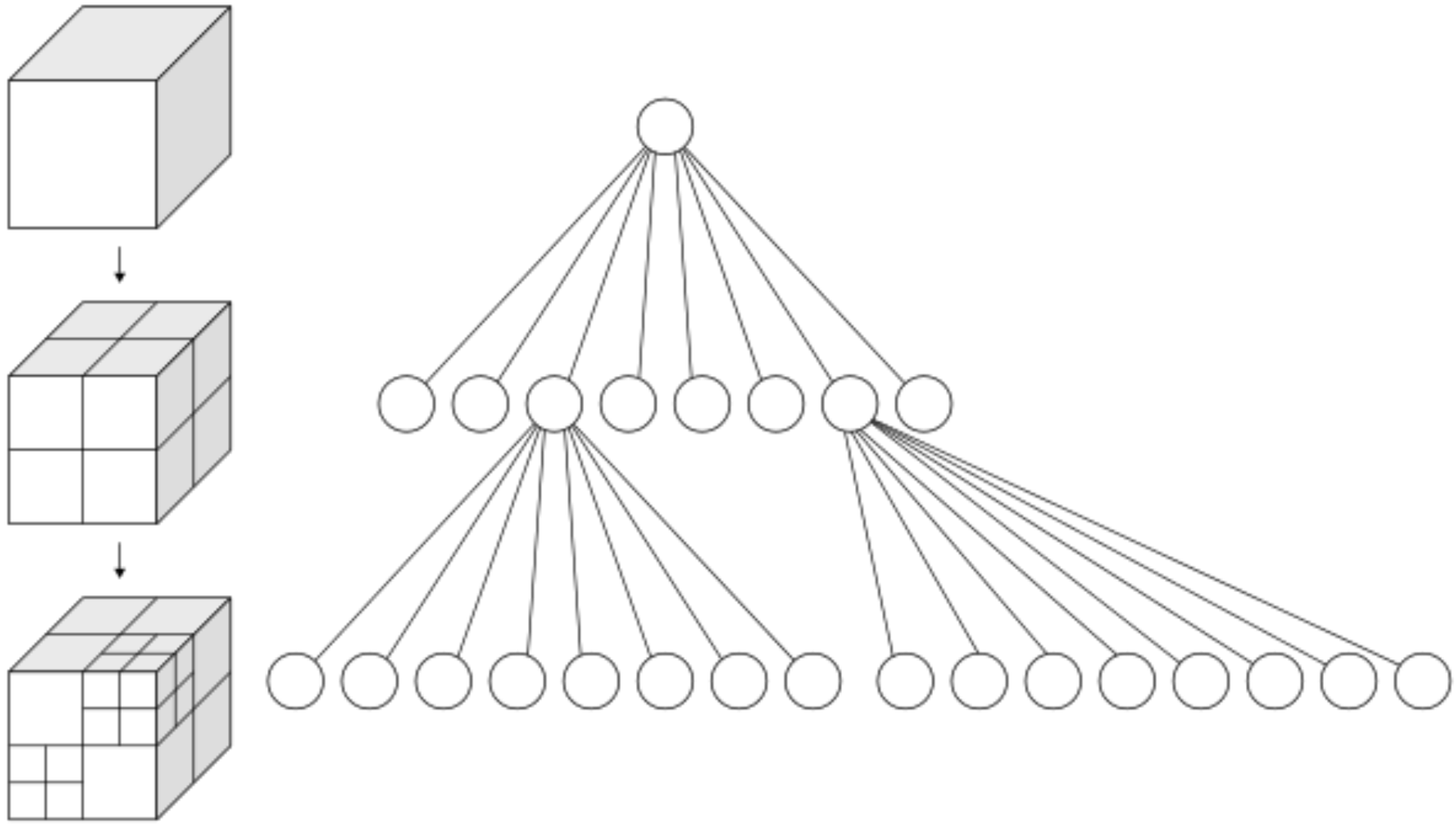
Sparse grids as quadtrees - a specific example





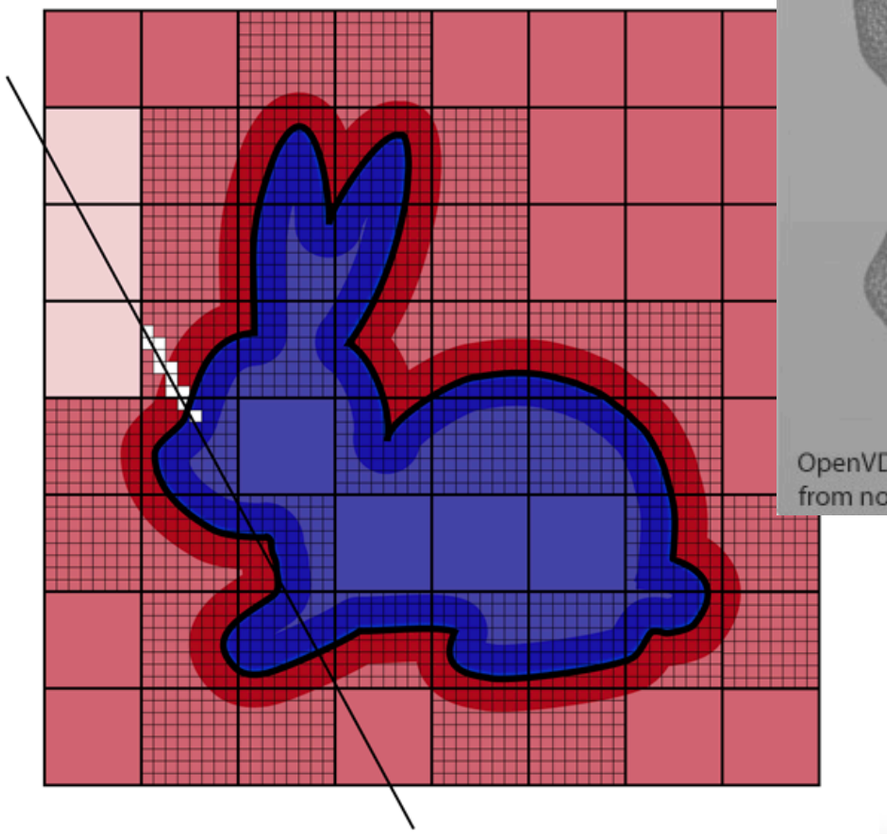
Easy and streamlined access of stencil neighbors is the primary obstacle to efficiency for sparse grid structures!

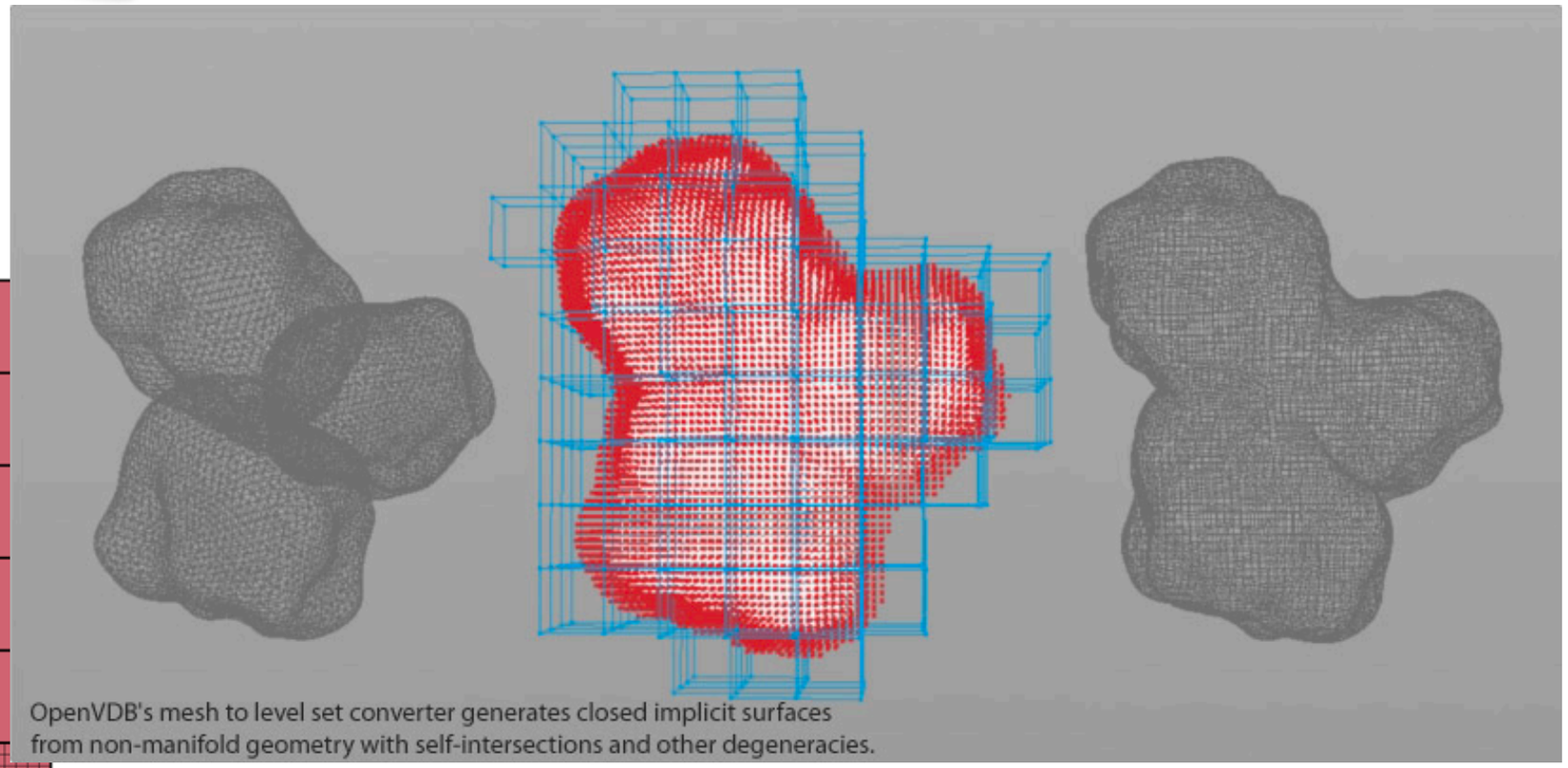
2D - > 3D?



In 3D the same hierarchical subdivision concept applies! Each cube gets split in 8 children, a 2x2x2 arrangement of cubes of half-edge-size!
The resulting data structure is called an octree.
Same considerations apply as in 2D

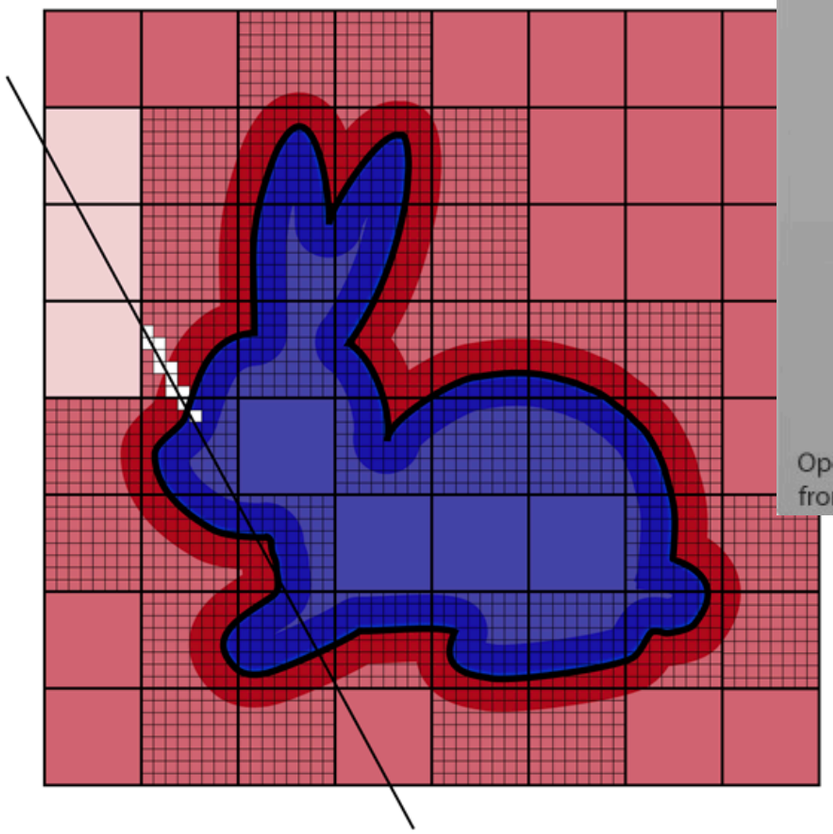
OpenVDB/NanoVDB

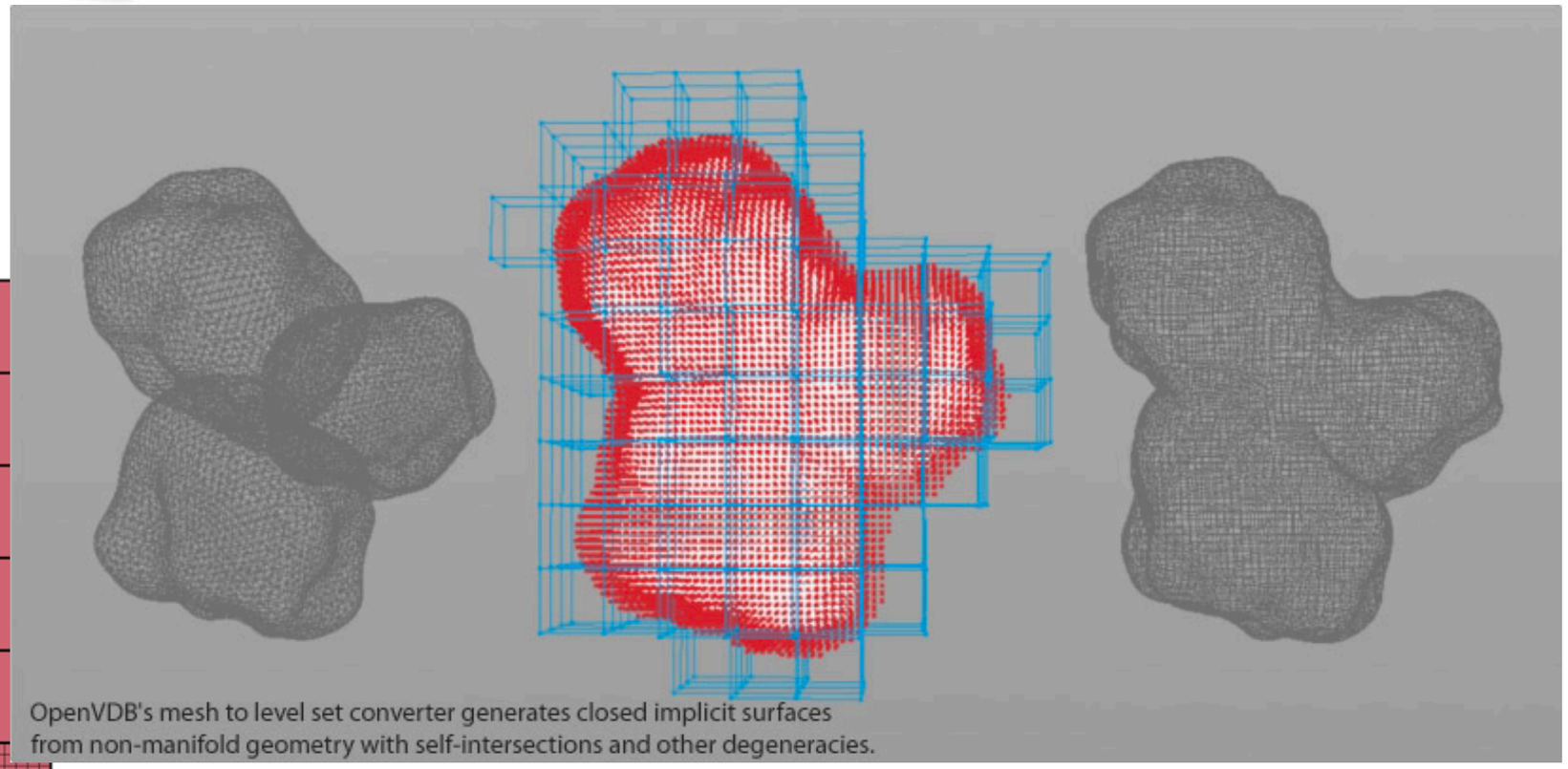




The data structure we will use to address some of these issues is OpenVDB (or its variant NanoVDB). The key idea is that it's a hierarchical (tree-like) sparse storage structure, but instead of each cube being split into 2x2x2 smaller "child" cubes, it's being split into 8x8 (in 2D) or 8x8x8 (in 3D) children at every level of the tree! I.e. each tree node has 512 children!

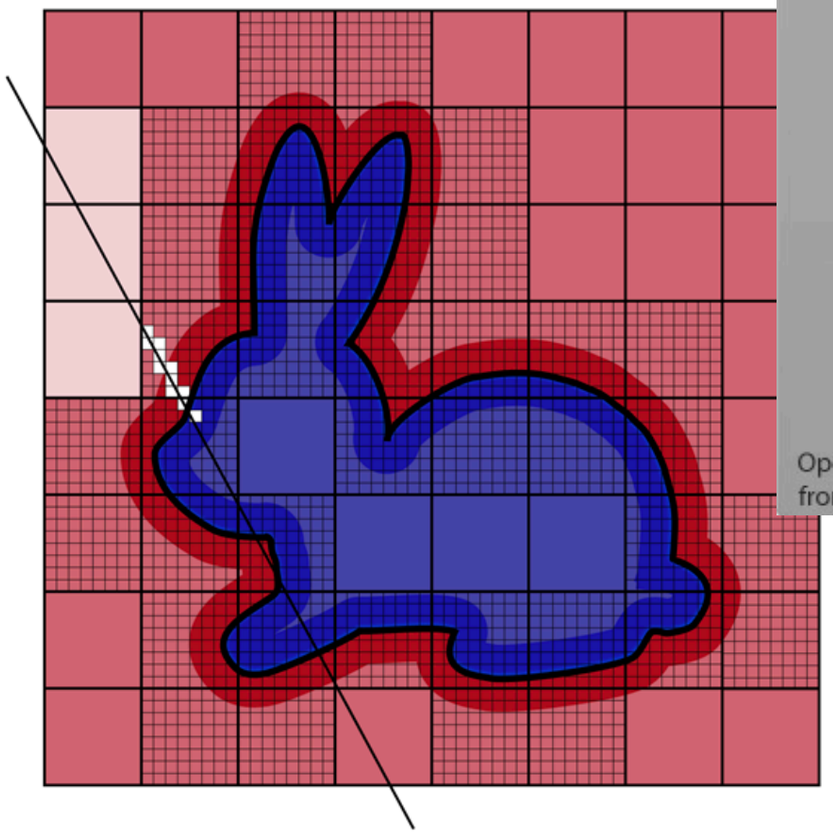
OpenVDB/NanoVDB

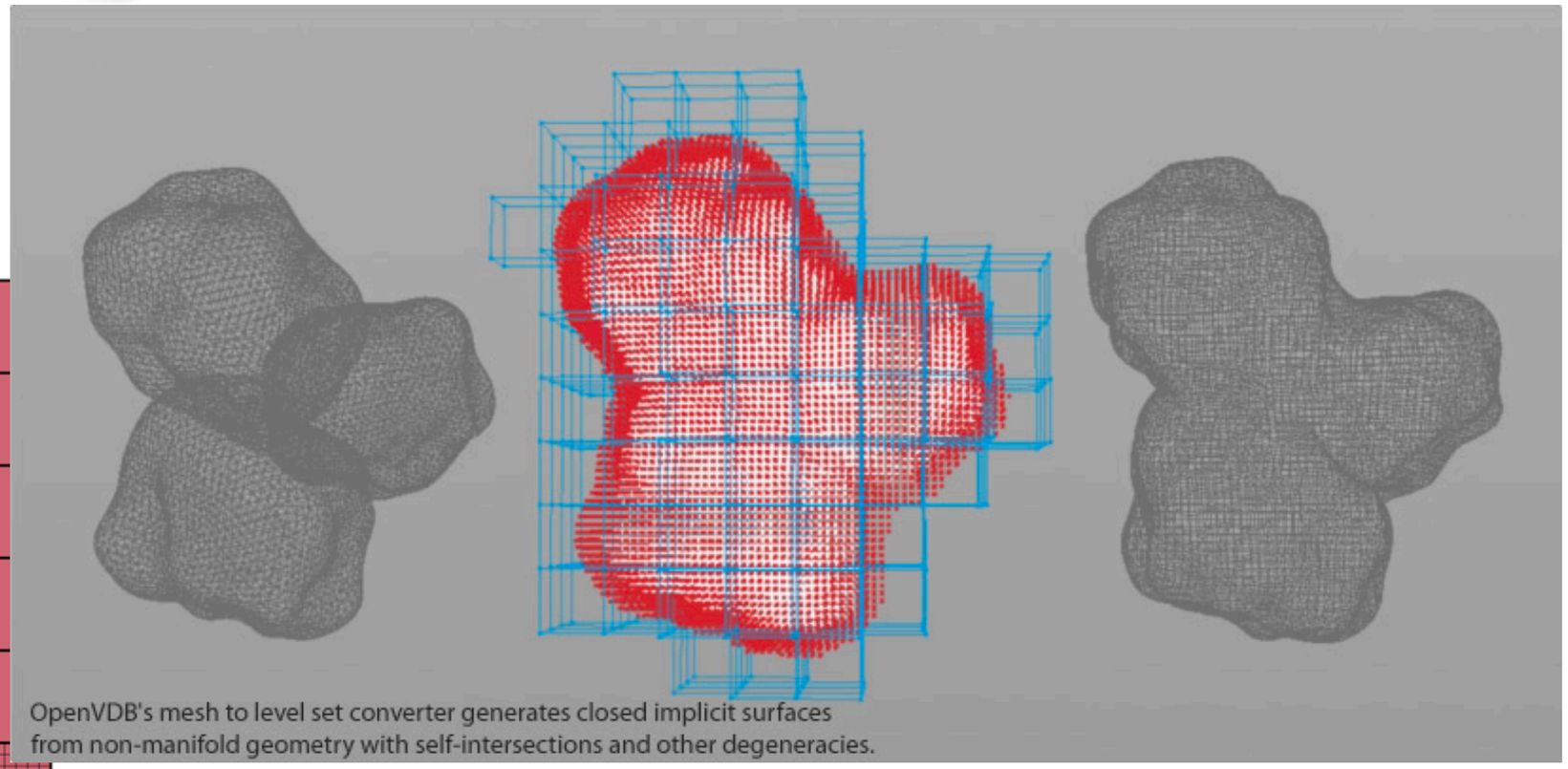




Benefit: At each leaf node of the VDB tree, we don't just have a single pixel (or 4/8 children), but a reasonably-sized 8x8x8 subgrid! This gives us plenty of opportunity to do most stencil operations within that grid!

OpenVDB/NanoVDB





OpenVDB's mesh to level set converter generates closed implicit surfaces
from non-manifold geometry with self-intersections and other degeneracies.

Optimization #2: VDB has caching structures that allow quick (or rather quick) access to nearby neighbors of a leaf-level grid.