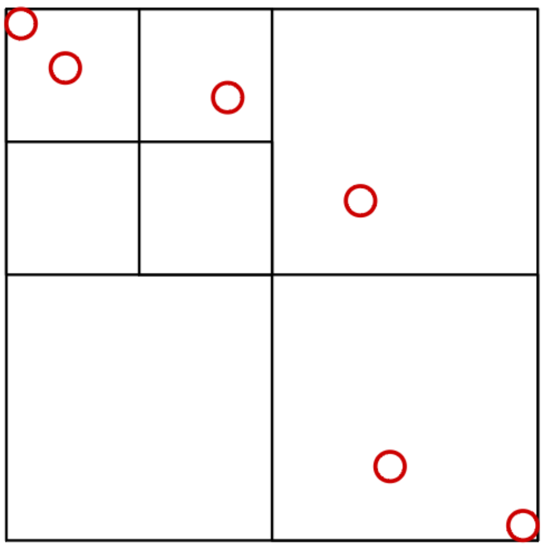
Efficient Z-Ordered Traversal of Hypercube Indexes

Tilmann Zäschke (ETH Zurich, Emineo)
Moira C. Norrie (ETH Zurich)

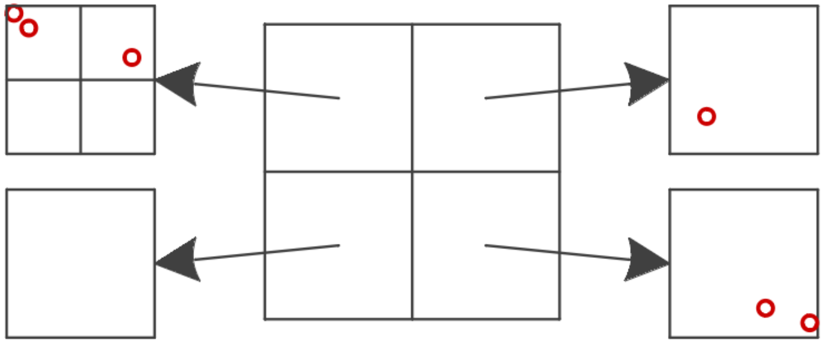
Multi-Dim Indexing



Some indexes use a <u>tree</u> of <u>non-overlaping</u> <u>quadrants</u>

- Quadtrees
- PH-Tree
- ...

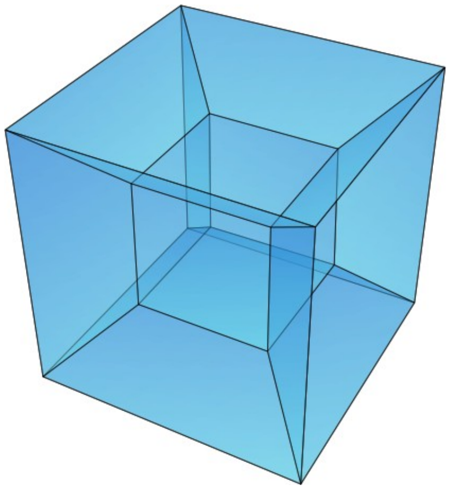
→ Hierarchy of hyperquadrants / hypercubes



<u>Navigation in hypercubes</u>

Hypercube

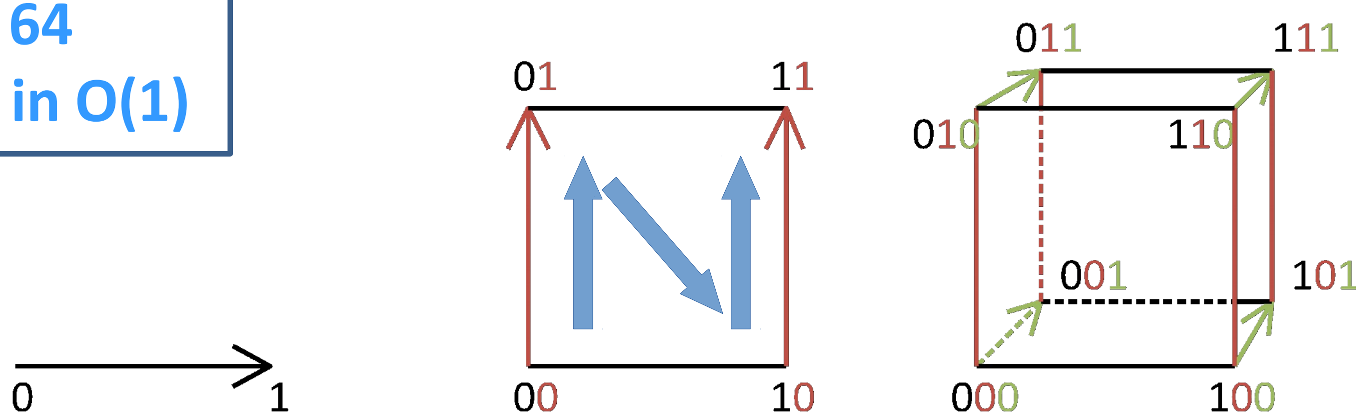
- ***k***-dimensional binary cube
- Each bit for one dimension
- Enumerate corners with ***k*** bits:
  = **011**…
  = position in linear array



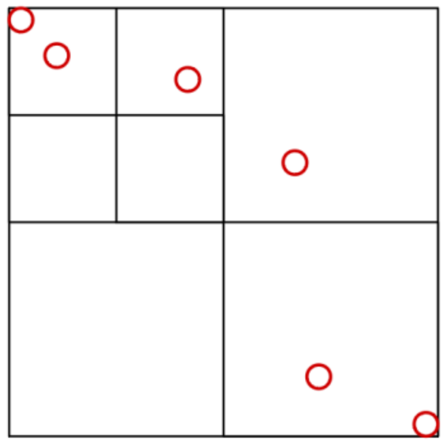
(Wikipedia, Goffrie, CC BY-SA 3.0)

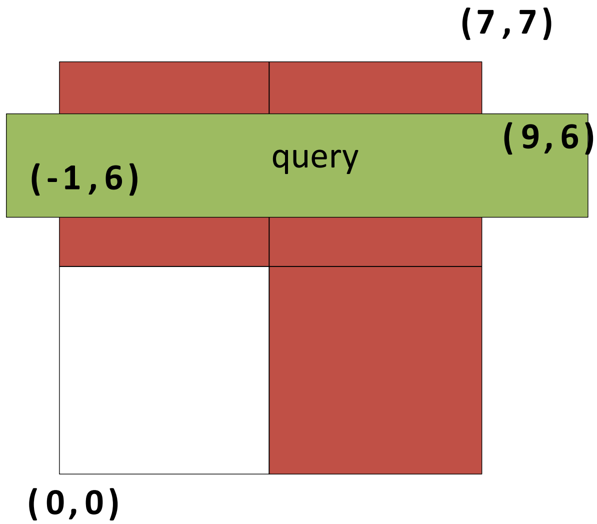
process 64 dimensions in O(1)

z-order / morton order



Queries: Find all $\boldsymbol{h} \in \boldsymbol{I}$ from all $\boldsymbol{h} \in \boldsymbol{N}$





***k*=2** → The number of dimensions

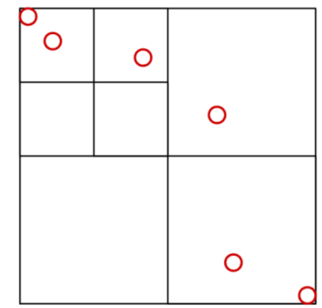
***I*=2** → The intersection, i.e.
the set of all quadrants that intersect with a query

***N*=3** → The node, i.e. the set of all occupied quadrants

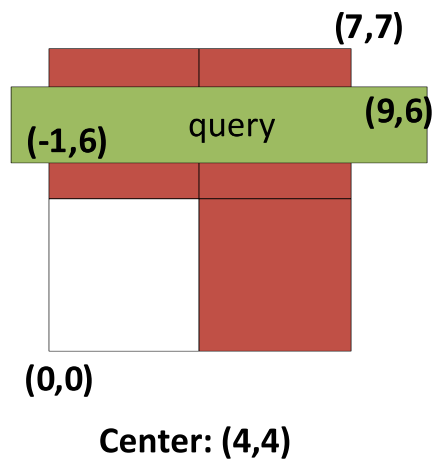
h → The hypercube address of a quadrant,
equal to its ID or position in an array, has ***k*** bits

Quadtree – Naïve Approach – List-QT

<u>Each node has a **list** of subnodes</u>



```
for each (quadrant) {
  if (overlap(quadrant, query)) {
    traverseSubnode(quadrant);
  }
}
```



→ Check 1 overlap: ***O(k)***

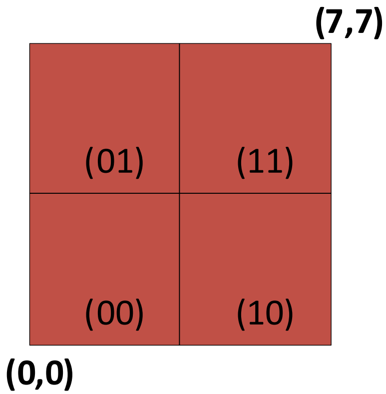
→ Check up to 2^k overlaps: ***O(k * 2^k) = Θ(k*N)***

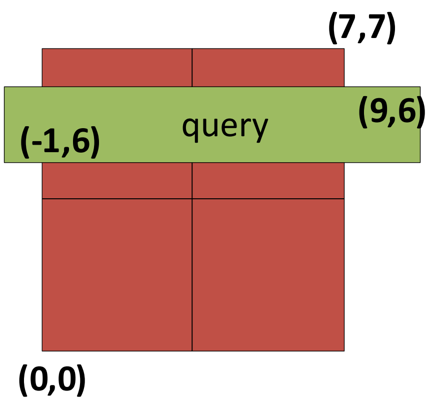
Same for range queries and exact match queries

Quadtree – Naïve Approach – Array-QT

Z-ordered **array** of subnodes
array position = z-address: [00, 01, 10, 11]=[0,1,2,3]

```
for each (quadrant) {
   if (quadrant != null &&
          overlap(quadrant, query)) {
      traverseSubnode(quadrant);
   }
}
```





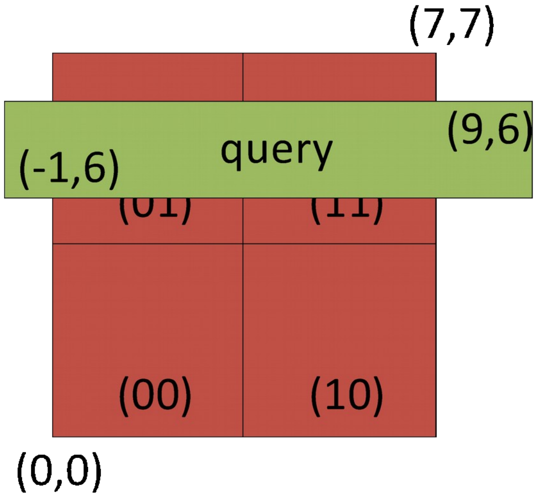
→ Check 1 overlap: ***O(k)***
→ Check all 2^k overlaps: ***O(k * 2^k)***
→ Same for range queries and exact match queries

Algorithm #0: m_0 & m_1

<u>HC encoding approach: Use bit masks with *k* bits</u>

(idea: The mask can tell us whether a quadrant matches)

```
m0=00; m1=00;
for each (k) {
  if (queryMin(k) >= center(k))
    m0[k] = 1;
  if (queryMax(k) >= center(k))
    m1[k] = 1;
}
```

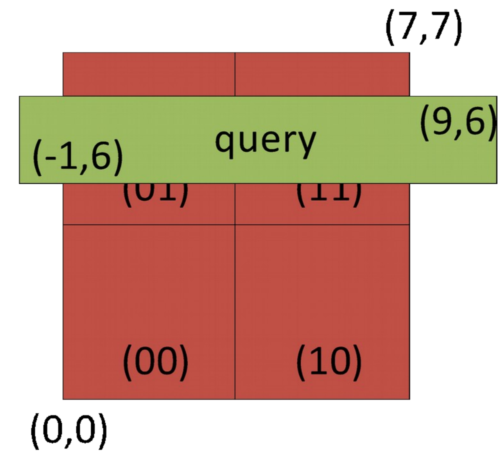


→ Example: m_0 = 01; m_1 = 11;

lo-mask m_0: *'1' indicates that low quadrants can be skipped.*

hi-mask m_1: *'0'indicates that high quadrants can be skipped.*

Algorithm #0: m_0 & m_1



<u>Some properties of m_0 and m_1</u>

<u>Start/End</u>

m_0/m_1 are the IDs/positions of the first and last intersecting quadrant

→ For exact match search this means
m_0==m_1 -> $\boldsymbol{O(k*2^k)}$ become $\boldsymbol{O(k)}$ **!**

[~~00~~, 01, 10, 11]

m_0 m_1

<u>Number of intersecting quadrants = |***I***|</u>

```
nBits1 = count_1_bits( m0 ^ m1 ); // ^ = XOR
sizeOfI = 1 << nBits1;            // 2^n Bits1
```

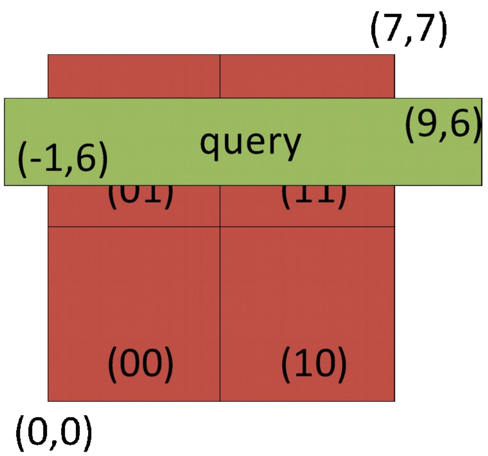
Algorithm #1: isInI(h, m0, m1)

Test if quadrant ***h*** is part of intersection ***I***:

Reject ***h*** if it has \`0’ where $m_0$ has a \`1\`:

```
if ((h | m0) != h) {
     return false;
}
```

(00 | 01 = 01) -> false
(01 | 01 = 01) -> true
(10 | 01 = 11) -> false
(11 | 01 = 11) -> true



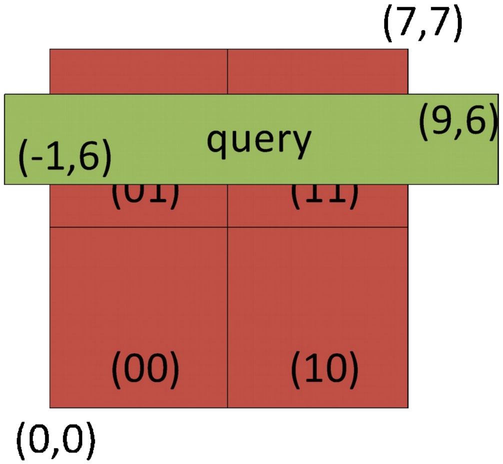
Reject ***h*** if it has \`1’ where $m_1$ has a \`0\`:

```
if ((h & m1) != h) {
     return false;
}
```

Combined: `isInI = ((h | m0) & m1) == h;`

Algorithm #1: isInI(h, m0, m1)

```
boolean isInI(int h, int m0, int m1) {
  return ((h | m0) & m1) == h;
}
```



Summary 1

Alg. #0: Calculate min/max: $\boldsymbol{\Theta}(k)$

Alg. #1: Check any quadrant in $\boldsymbol{\Theta}(1)$

Exact match query: $m_0 = m_1$

→ $\boldsymbol{\Theta}(k + 1)$

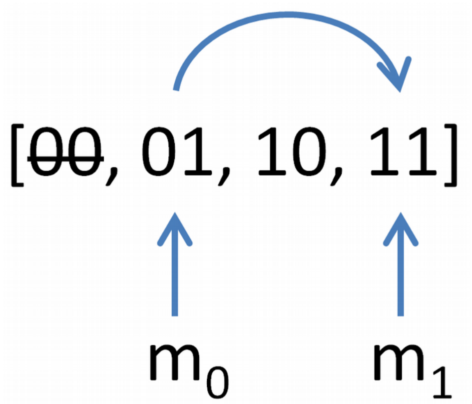
Window query: Check m_1-m_0 ($\leq 2^k$) overlaps:

→ $\boldsymbol{\Theta}(k) + \boldsymbol{O}(2^k) * \boldsymbol{\Theta}(1) = \boldsymbol{O}(k + 2^k)$

Naive: $\boldsymbol{O}(k * 2^k)$

Algorithm #2: inc(h, m_0, m_1)

Can we 'jump' from one ***h*** ϵ ***I*** to the next?

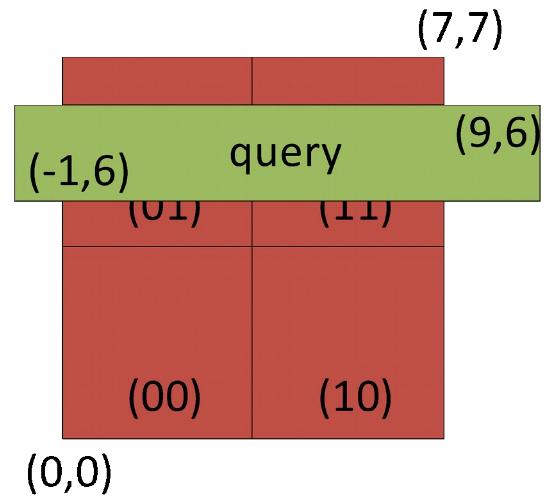


In any valid ***h*** some bits may be restricted to be either 0 or 1.

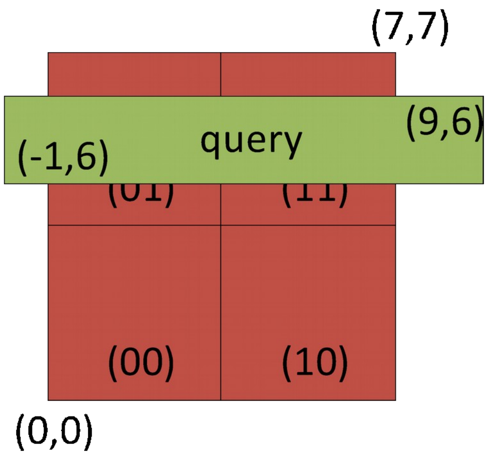
Example: inc(0**1)** **→** 1**1**.

If query intersects 00/10: inc(00) → 10

If query intersects only x: inc(x) → ?



Algorithm #2: inc(h_{in}, m_0, m_1)



1) Set all `fixed bits’ to `1’.
2) Add 1 -> The overflows on all fixed bits `forward’ increment to higher bits.
3) Set all fixed bits to their fixed state.

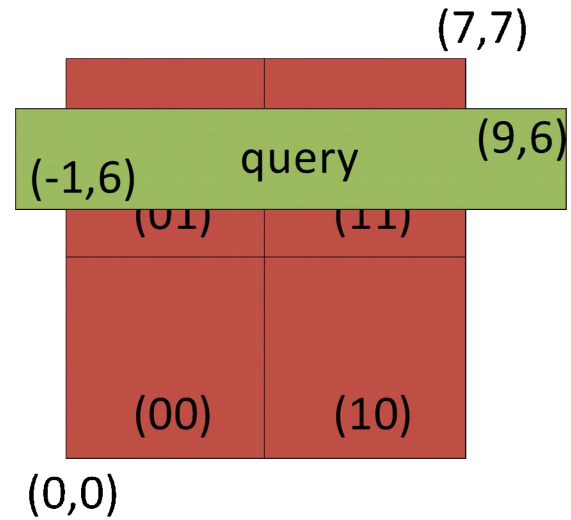
01 → setFixedTo1 → 01 → add1 → **10** → resetFixed → 1**1**

(00 → setFixedTo1 → 0**1** → add1 → **10** → resetFixed → 10)

Code:

```
h = h | (~m1);           //pre-mask
h++;                     //increment
h = (h & m1) | m0;       //post-mask
```

Algorithm #2: inc(h, m_0, m_1)



<u>Summary 2</u>

#0: Calculate min/max: $\boldsymbol{\Theta}(k)$ per node

#2: Increment in $\boldsymbol{\Theta}(1)$ per $\boldsymbol{h} \in \boldsymbol{I}$

→ Window query:

Naive:	$\boldsymbol{\Theta}(k * \lvert\boldsymbol{N}\rvert) = \boldsymbol{O}(k * 2^k)$
With isInI(...):	$\boldsymbol{\Theta}(k + \lvert\boldsymbol{N}\rvert) = \boldsymbol{O}(k + 2^k)$
With inc(...):	$\boldsymbol{\Theta}(k + \lvert\boldsymbol{I}\rvert)$

Note: if ($\lvert\boldsymbol{I}\rvert > \lvert\boldsymbol{N}\rvert$) then isInI() is faster than inc()!

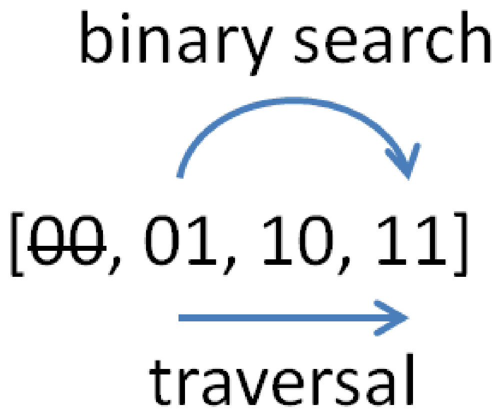
Algorithm #3: succ(h, m_0, m_1)

Alg #2:

Gives next valid ***h*** based on a **valid *h*** ∊ ***I***

Alg #3:

Gives next valid ***h*** based on **any *h***



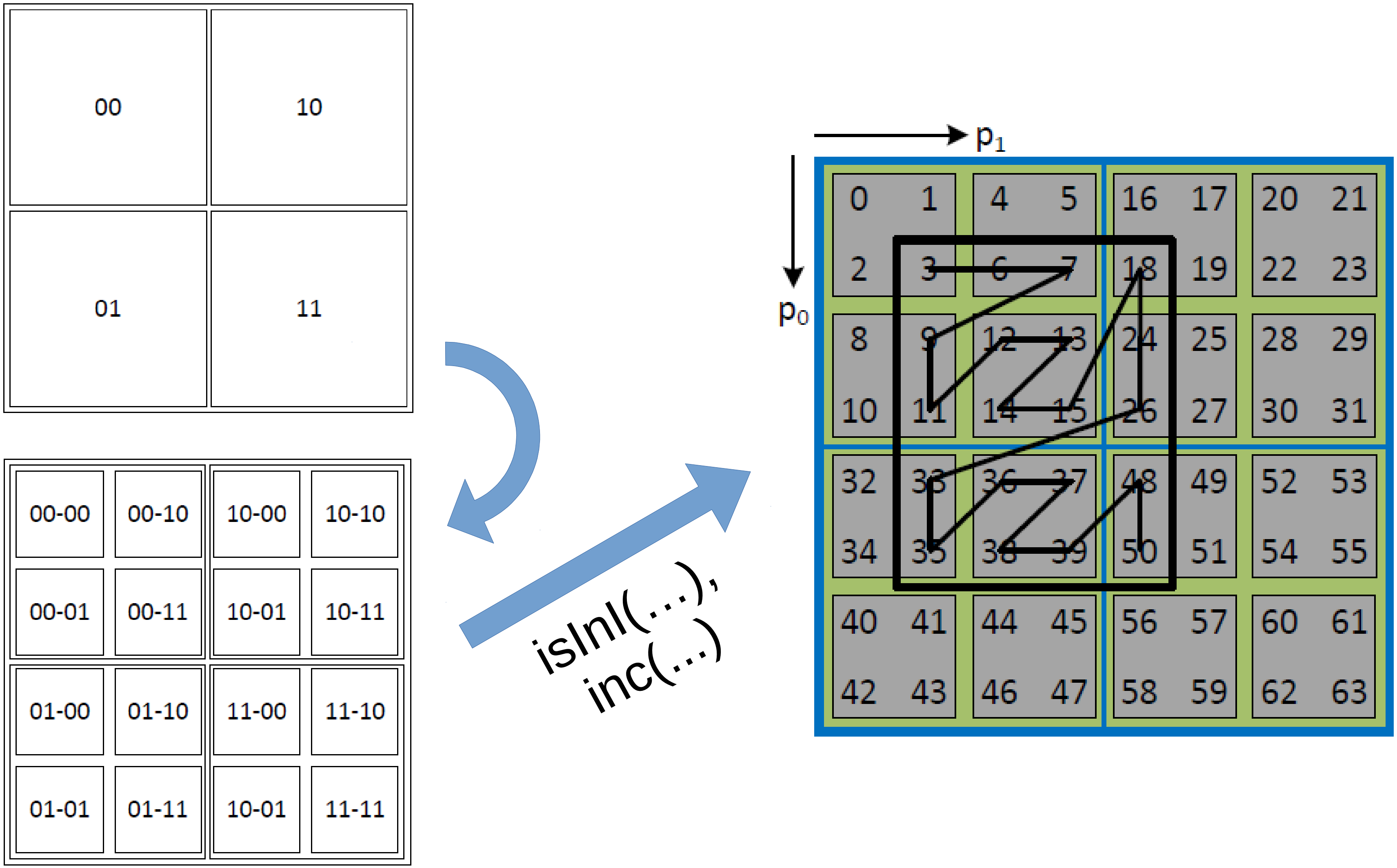
Motivation:

Query may change/move during execution

Decide on the fly to switch from isInI() to inc()

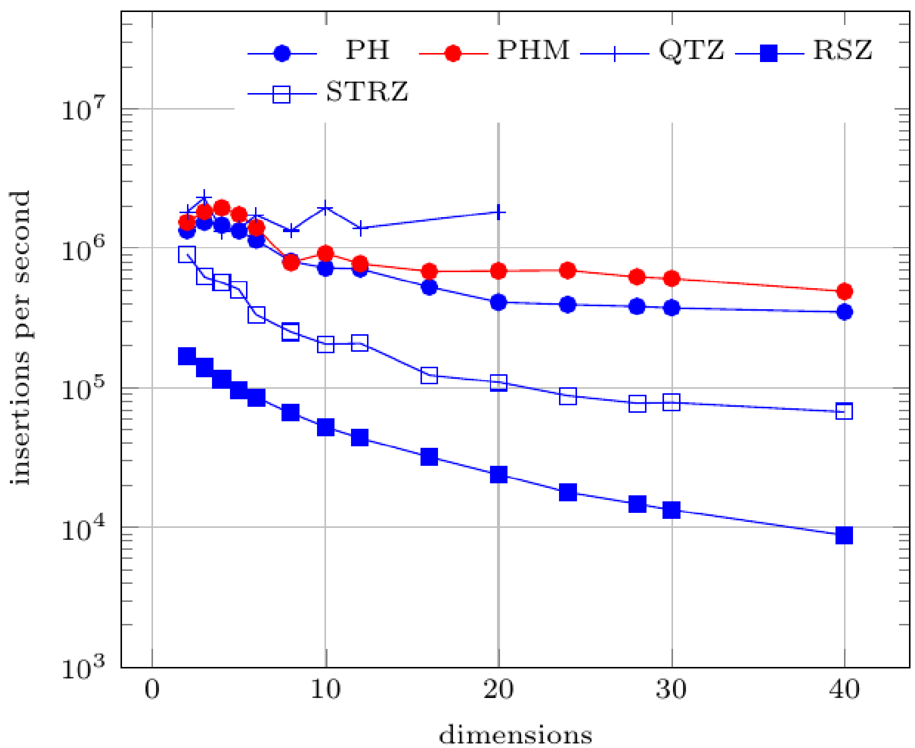
Not shown here, executes in $\boldsymbol{\Theta}(1)$

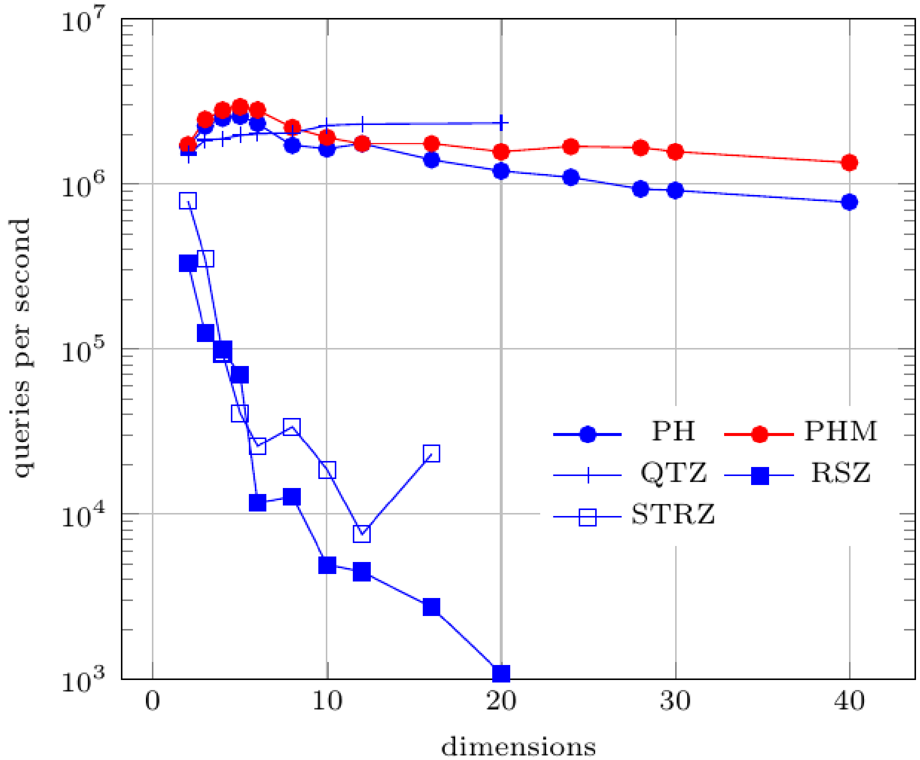
PH-Tree: Z-Ordered Traversal



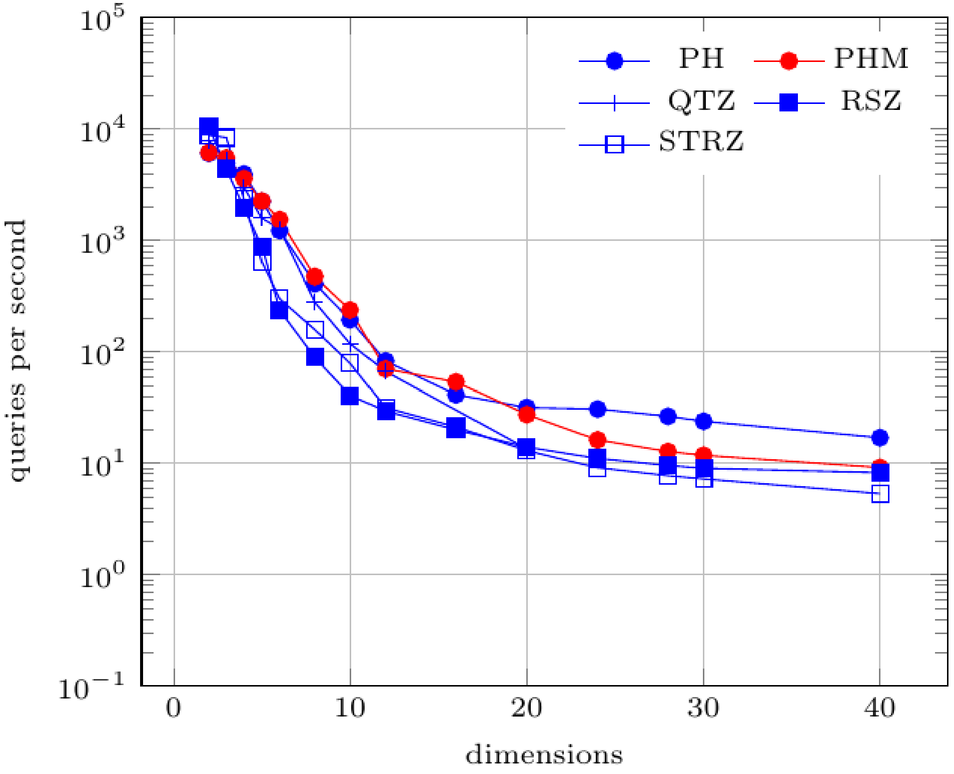
PH-Tree with isInI()

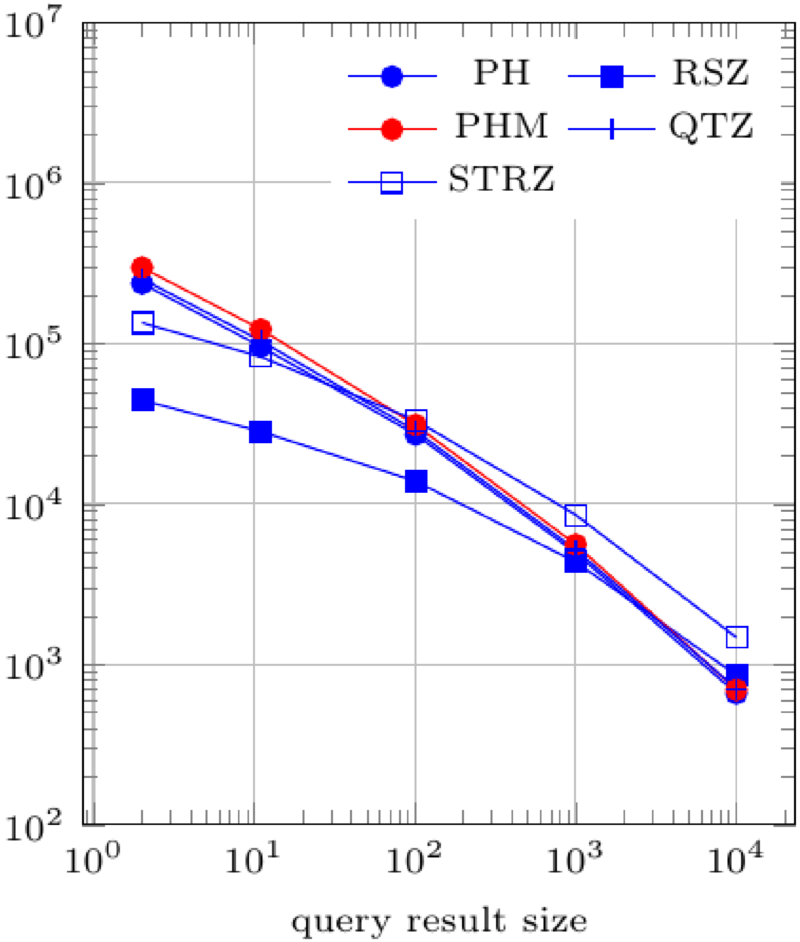
- Shaped like a quadtree, but is actually a bit-level trie
- Splits at every ‘bit’ → at most 64 levels for 64bit data
- Example: 1M points, evenly distributed between [0 ... 1.0]



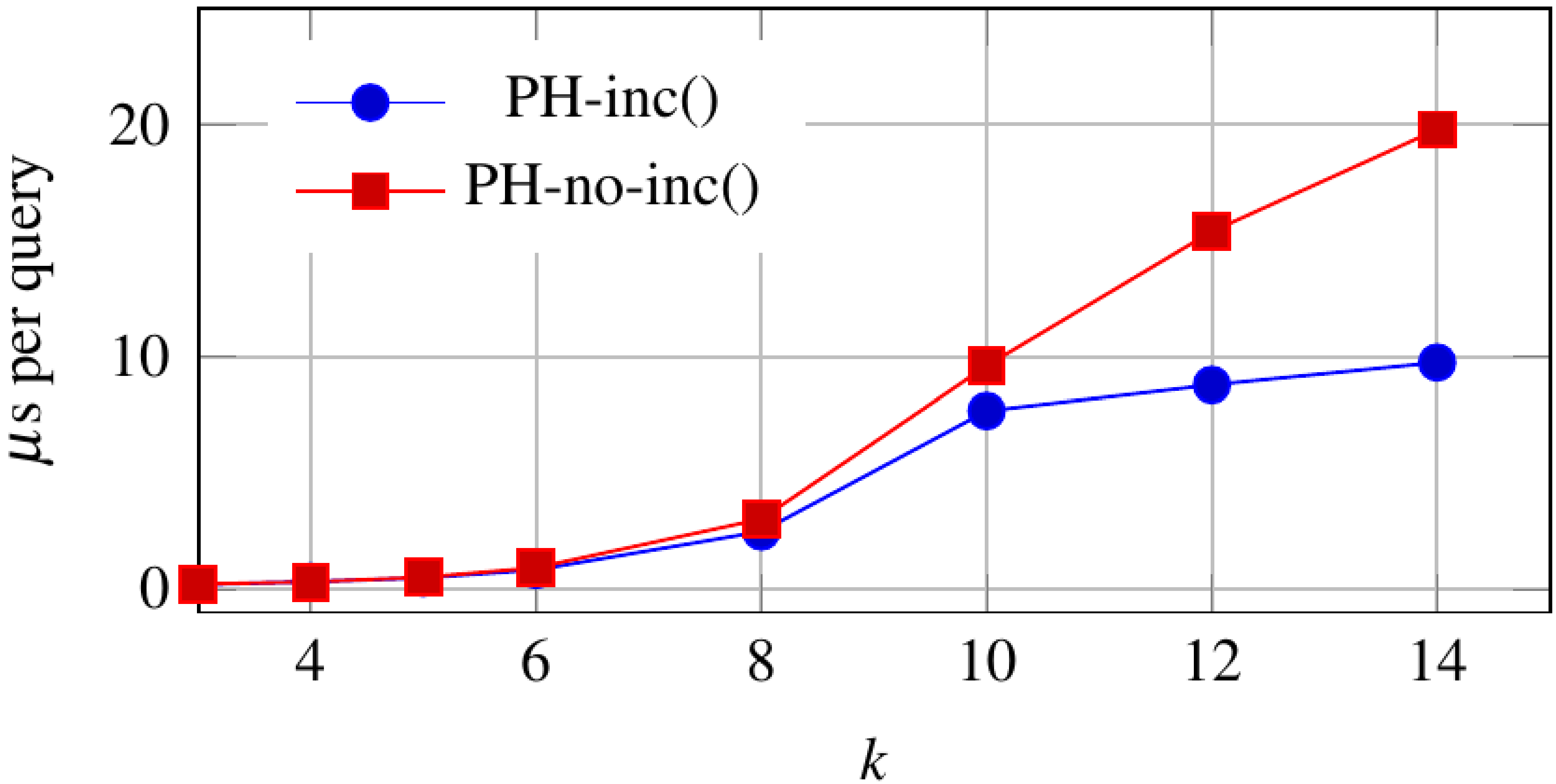


Window Queries over *k* and varying size for 3D





PH-Tree with inc()



10^5 entries, *k*-dim cube, randomly distributed [0...1]

But, PH avoids large nodes anyway (NT), hence no succ()

Summary

3½ Algorithms

- m_0/m_1 lo/hi-mask max + start/endpoint + $|\boldsymbol{I}|$ $\boldsymbol{O}(k)$/node
- isInI() Check if quadrant intersects query $\boldsymbol{O}(1)$/q
- inc() Next intersecting quadrant after $\boldsymbol{h} \in \boldsymbol{I}$ $\boldsymbol{O}(1)$/q
- succ() Next intersecting quadrant after any $\boldsymbol{h}$ $\boldsymbol{O}(1)$/q

$\mathbf{m_1}$ is, for example, used in SkylineQueries, with **isInI($\mathbf{m_1}$**-only)

Navigation in k=60 dimensions often possible in $\boldsymbol{O}(k)$/node