Efficient Z-Ordered Traversal of Hypercube Indexes

Tilmann Zäschke (ETH Zurich, Emineo)
Moira C. Norrie (ETH Zurich)
Multi-Dim Indexing

Some indexes use a **tree** of non-overlapping quadrants
- Quadtrees
- PH-Tree
- ...

→ Hierarchy of hyperquadrants / hypercubes

**Navigation in hypercubes**
Hypercube

- \( k \)-dimensional binary cube
- Each bit for one dimension
- Enumerate corners with \( k \) bits:
  - \( 011... \)
  - = position in linear array

process 64 dimensions in \( O(1) \)

z-order / morton order

(Wikipedia, Goffrie, CC BY-SA 3.0)
Queries: Find all $h \in I$ from all $h \in N$

$k=2$ → 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
Each node has a **list** of subnodes

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

→ Check 1 overlap: \(O(k)\)

→ Check up to \(2^k\) overlaps: \(O(k \times 2^k) = \Theta(k \times N)\)

Same for range queries and exact match queries
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: $m_0$ & $m_1$

HC encoding approach: Use bit masks with $k$ bits
(idea: The mask can tell us whether a quadrant matches)

$m_0 = 00$; $m_1 = 00$;

for each ($k$) {
  if (queryMin($k$) $\geq$ center($k$))
    $m_0[k] = 1$;
  if (queryMax($k$) $\geq$ center($k$))
    $m_1[k] = 1$;
}

→ 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: \( m_0 \& m_1 \)

Some properties of \( m_0 \) and \( m_1 \)

Start/End

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

\[ \rightarrow \text{For exact match search this means } m_0 == m_1 \rightarrow O(k \times 2^k) \text{ become } O(k)! \]

Number of intersecting quadrants = |I|

\[ \text{nBits1} = \text{count}_1\_\text{bits}( m_0 \& m_1 ); \ // \ & = \text{XOR} \]

\[ \text{sizeOfI} = 1 << \text{nBits1}; \ // \ 2^n \text{ Bits}1 \]
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 ($((h | m_0) != h)$ {
    return false;
}

Reject $h$ if it has `1' where $m_1$ has a `0':
if ($((h & m_1) != h)$ {
    return false;
}

Combined: $\text{isInI} = ((h | m_0) & m_1) == h$;
Algorithm #1: isInI(h, m0, m1)

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

Summary 1
Alg. #0: Calculate min/max: $\Theta(k)$
Alg. #1: Check any quadrant in $\Theta(1)$
- Exact match query: $m_0 = m_1$
  $\Rightarrow \Theta(k + 1)$
- Window query: Check $m_1 - m_0 \leq 2^k$ overlaps:
  $\Rightarrow \Theta(k) + O(2^k) * \Theta(1) = O(k + 2^k)$
  Naive: $O(k * 2^k)$
Algorithm #2: $\text{inc}(h, m_0, m_1)$

Can we ‘jump’ from one $h \in I$ to the next?

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

Example: $\text{inc}(01) \rightarrow 11$.

If query intersects 00/10: $\text{inc}(00) \rightarrow 10$

If query intersects only $x$: $\text{inc}(x) \rightarrow ?$
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 → 11
(00 → setFixedTo1 → 01 → add1 → 10 → resetFixed → 10)

Code:

```c
h = h | (~m1); //pre-mask
h++; //increment
h = (h & m1) | m0; //post-mask
```
Algorithm #2: inc($h$, $m_0$, $m_1$)

Summary 2

#0: Calculate min/max: $\Theta(k)$ per node
#2: Increment in $\Theta(1)$ per $h \in I$

→ Window query:

Naive: $\Theta(k \ast |N|) = O(k \ast 2^k)$
With isInI(...): $\Theta(k + |N|) = O(k + 2^k)$
With inc(...): $\Theta(k + |I|)$

Note: if ($|I| > |N|$) then isInI() is faster than inc()!
Algorithm #3: succ($h, m_0, m_1$)

Alg #2:
Gives next valid $h$ based on a valid $h \in I$

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 $\Theta(1)$
PH-Tree: Z-Ordered Traversal

T. Zäschke, M. Norrie, ETH Zurich
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

![Graph showing performance comparison for different algorithms.](image-url)
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 + \( |I| \) \( O(k)/\text{node} \)
• \( \text{isInI()} \) Check if quadrant intersects query \( O(1)/q \)
• \( \text{inc()} \) Next intersecting quadrant after \( h \in I \) \( O(1)/q \)
• \( \text{succ()} \) Next intersecting quadrant after any \( h \) \( O(1)/q \)

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

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