The PH-Tree
A Multi-Dimensional Index

Tilmann Zäschke

PH-tree presentation 2022 © 2022 by Tilmann Zäschke is licensed under CC BY 4.0. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/

PH-tree vs Quadtree

The PH-tree is similar to a quadtree in the sense that:

- It uses a hierarchy of nodes to organize data
- Each node is a square and has four quadrants (eight in 3D, in general 2^{dim} quadrants), i.e. each node splits space in all dimensions.
- Nodes are split into sub-nodes when they contain too many points.

However, the PH-tree does various things differently in order to

- improve scalability with higher dimensions than 2D or 3D,
- avoid “deep” trees when storing strongly clustered data,
- avoid nodes with < 2 entries (except the root node), and
- reduce reshuffling of data when nodes are split/merged.

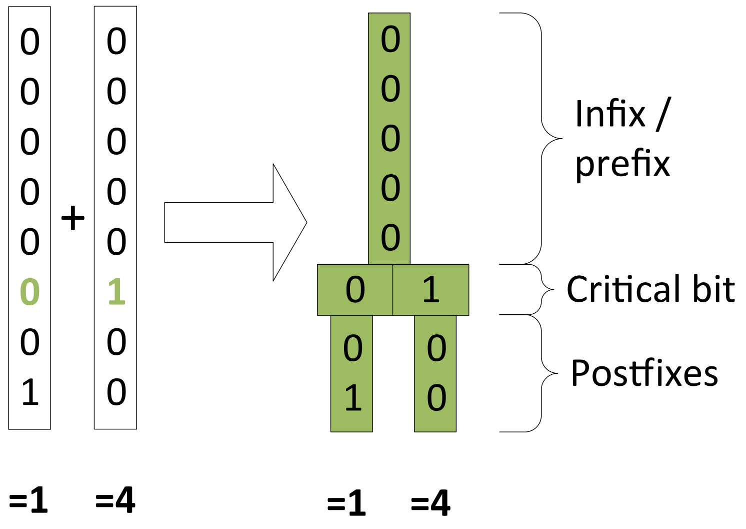
PH-tree vs Quadtree

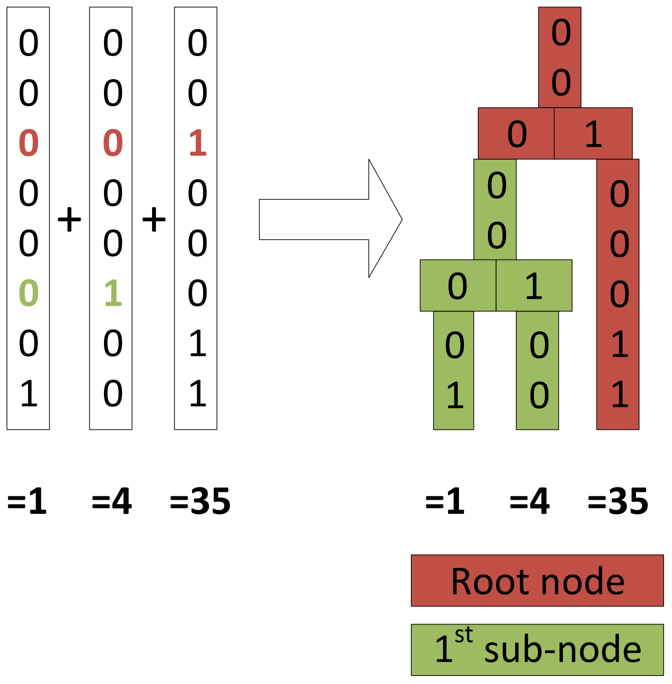
Differences in appearance to quadtrees

- The PH-tree works with integers (it works fine with floating point numbers as well, as we discuss later)
- The PH-tree's "highest"possible node always has (0,0) as center and an endge length $\mathbf{l_{max} = 2^{32}}$ (for 32 bit coordinates). This node may not exist in most trees, but all nodes are aligned as if it existed, e.g. no other node overlaps with (0,0).
- In a PH-tree, child nodes always have an edge length $\mathbf{l_{child} = l_{parent} / 2^{y}}$, with **y** being a positive integer such that $\mathbf{l_{child}}$ is always an integer >= 1, in fact $\mathbf{l_{child}}$ is always a power of 2. This limits the depth of a PH-tree to 32.
- Quadrant capacity = 1.

1D PH-Tree

Example of a 1-dimensional PH-tree with 8-bit coordinates (we use **bit representation** (= base 2) for most examples):
First we add (1) and (4) to the tree, then we add (35).





→ The 1D PH-Tree is equivalent to a CritBit tree or digital PATRICIA trie

→ **Limited depth & imbalance**

→ **No rebalancing**

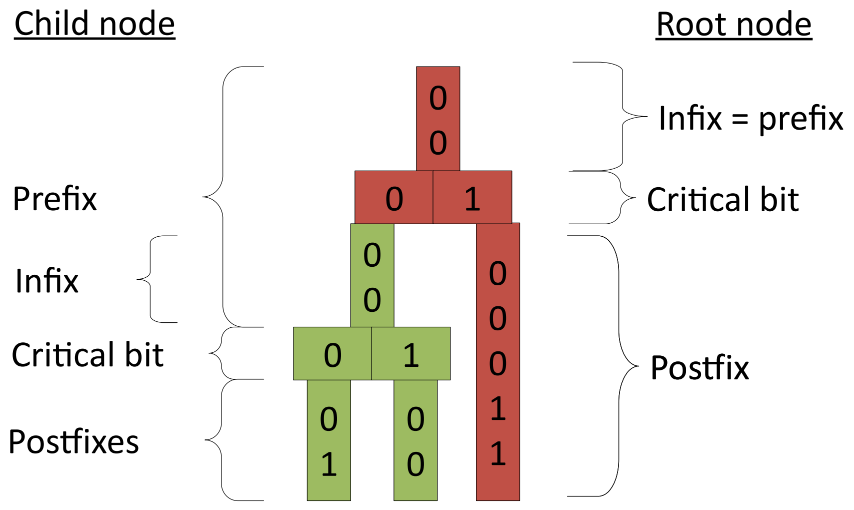
Some terminology

From the viewpoint of a node, every point (=key) is divided into the following sections:

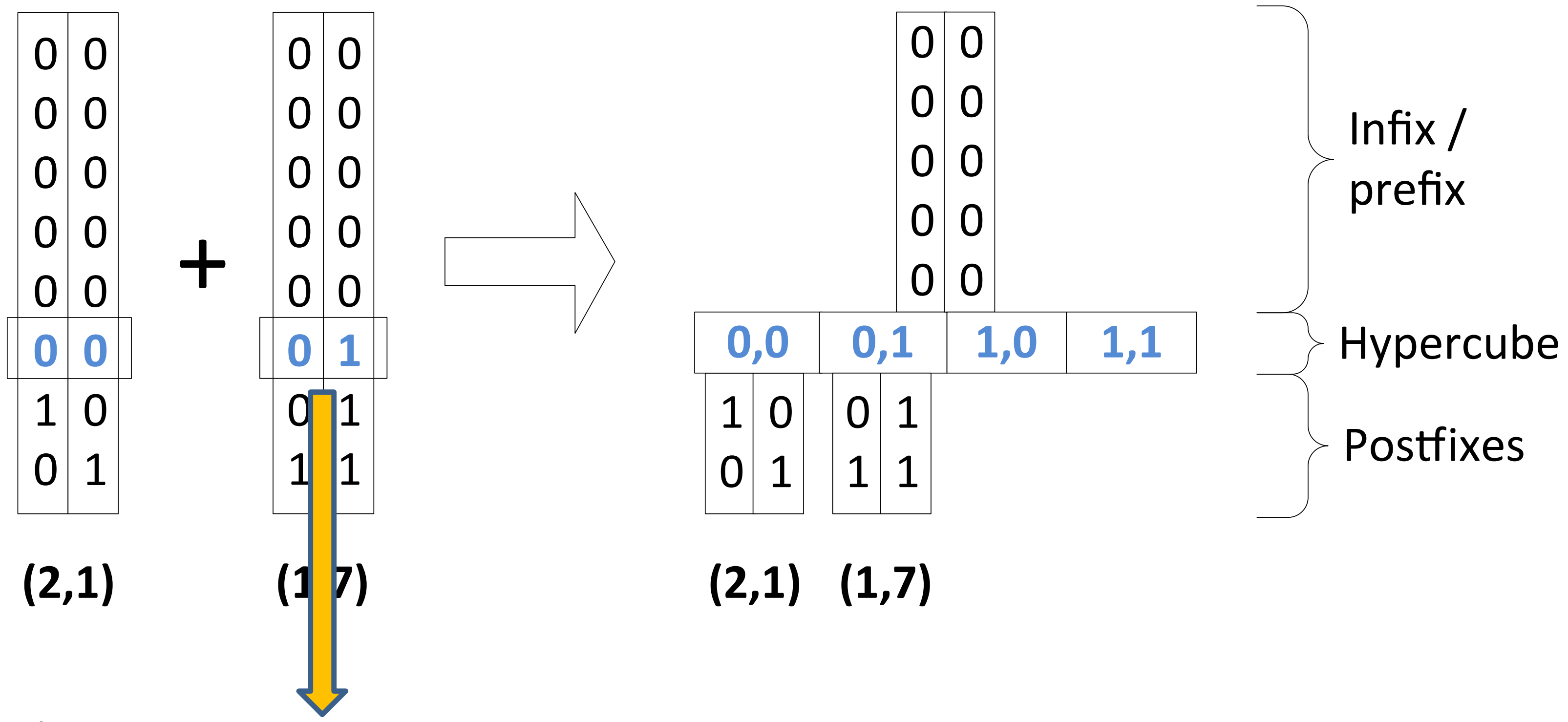
Infix: all bits above the current node
Prefix: all bits between the current node and it's parent.
Critical bit(s): the bit(s) that represent the HC address of the point/key.
Postfix: all bits below the current node (usually only if there is no child node, otherwise called "infix of child").

A a stored **point** is also called **key** or **coordinate**.

d is the number of dimensions of.
w is the number of bits,
e.g. 32 or 64.



2D PH-Tree Insertion



(01) = position in linear array

The extracted bits (01) at the depth of the node give us directly the index in the array of quadrants!

Complexity:	Calculate index:	*O*(*d*)	(d = number of dimensions)
	Find child:	*O*(1)	

Excursion: **Hypercube**

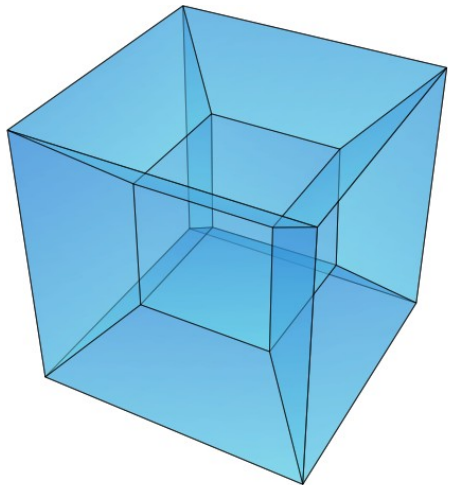
d-dimensional binary cube (binary Hamming Space)

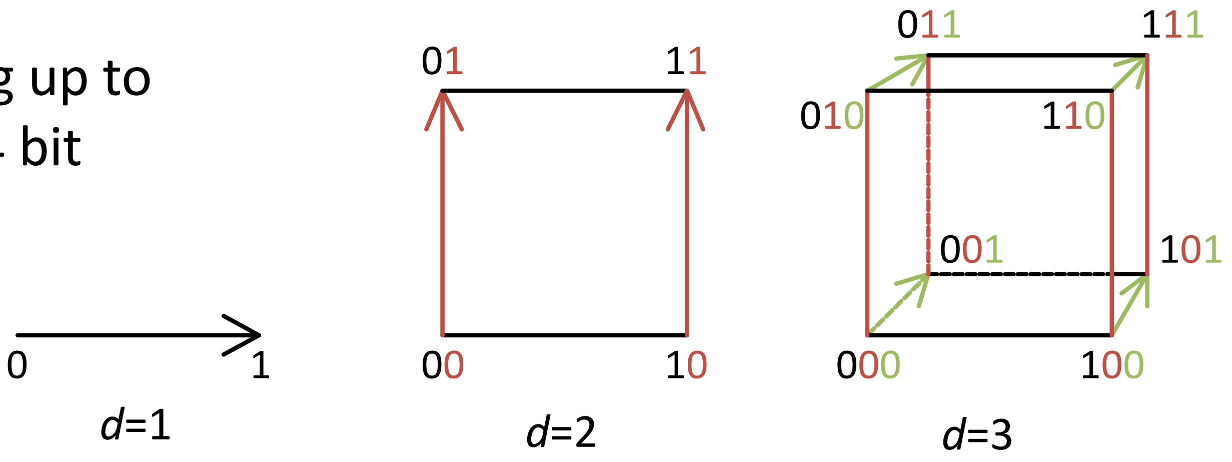
One bit for every dimension.

Hypercube (HC) address: e.g. **011**…

Idea: quadrants in a node can be numbered and accessed with their HC address.

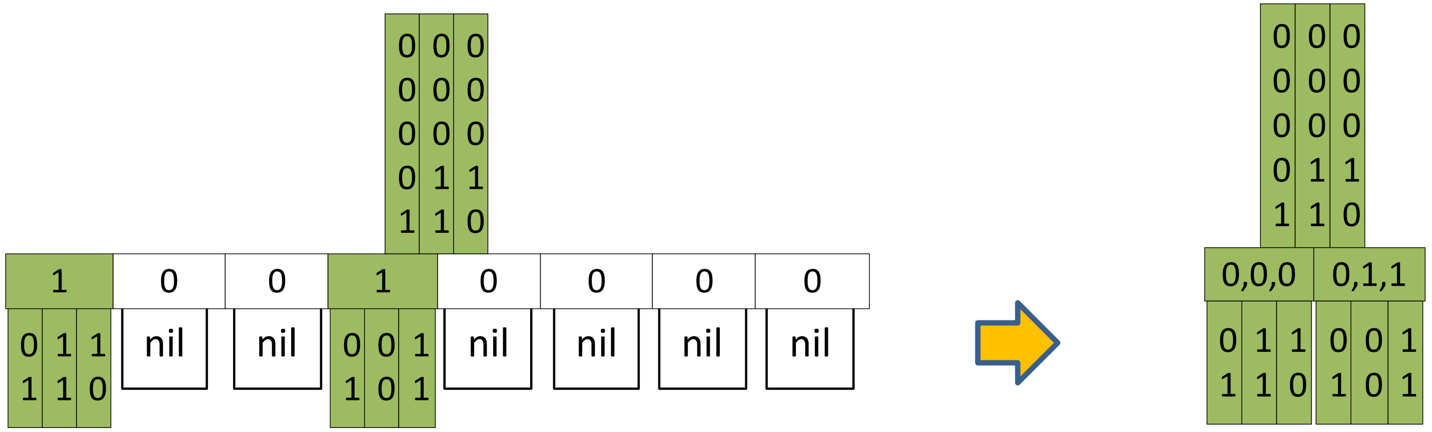
This allows processing up to 64 dimensions in a 64 bit CPU register in ***O(1)***!


(Wikipedia, Goffrie, CC BY-SA 3.0)



List HC & B+tree HC

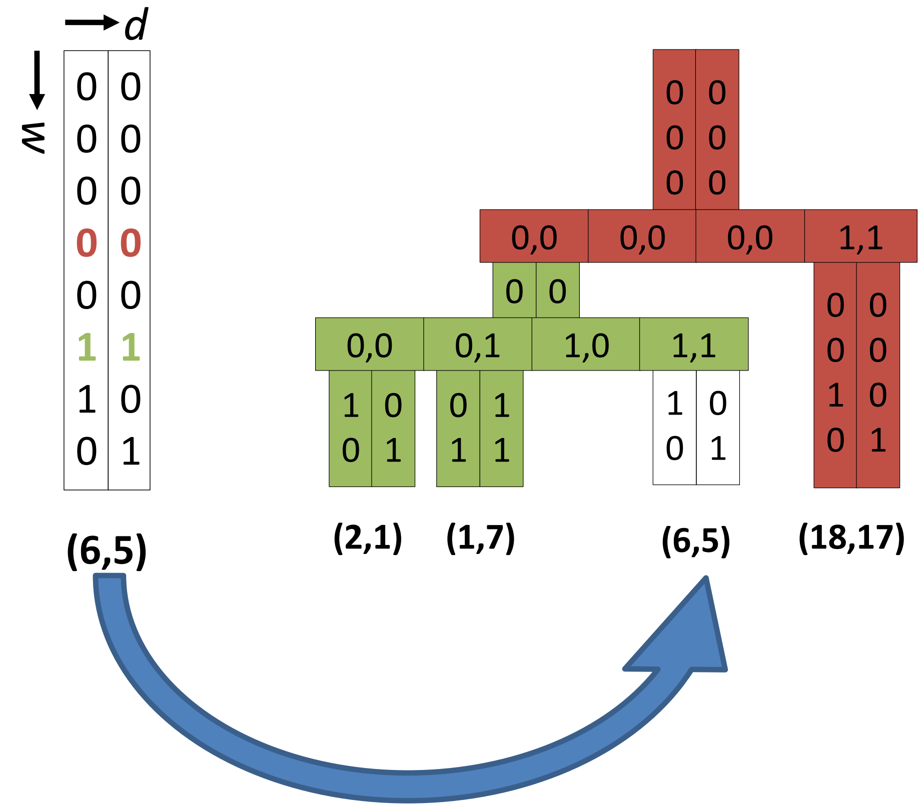
Arrays with **2^d** quadrants don't scale well with dimensionality. For higher dimensions the nodes store quadrants in a sorted list or B+tree. The three storage modes are called AHC (array), LHC (linear list) and BHC (B+tree).



array
Space: $O(w*d*2^d)$
Search: $O(1)$

sorted list
$O(w*d*n)$
$O(\log n)$

Insertion Example #2



Insertions (and deletions) can navigate the tree quickly by extracting the array position from the key.

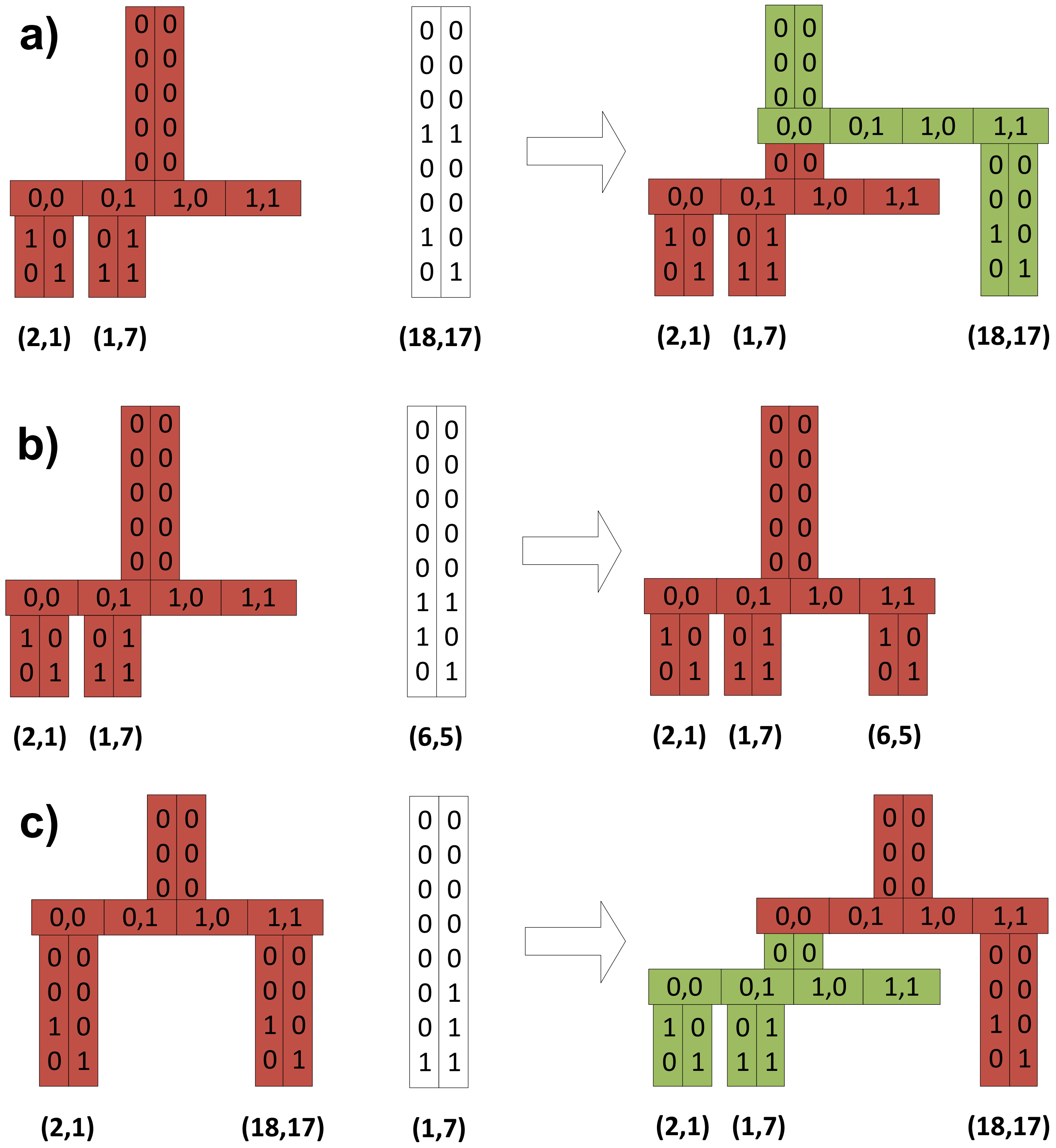
Insertion

3 cases for insert:

a) prefix collision

b) free postfix slot

c) postfix collision



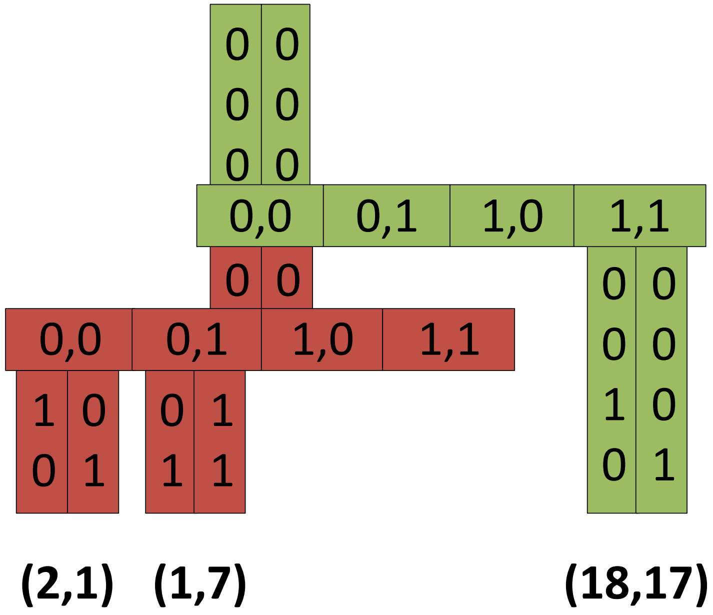
Hypercube navigation – point access

Point query

How do we find a key
(=does a given key exist)?

For each node:

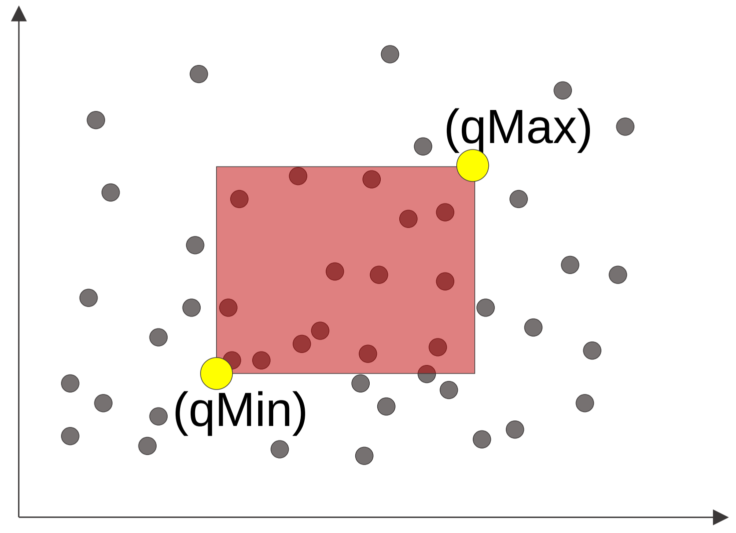
- compare infix/prefix
- extract HC-address from current key
- if entry at HC-address exists:
  access subnode or key/postfix



Hypercube navigation – window queries (WQ)

Window queries (WQ):

Find all points that lie inside a rectangular query window defined by `qMin` and `qMax`.



Example:

```
    qMin = (-1,6);
    qMax = (9,6);
    query(qMin, qMax);
```

For each node, naïve approach:
Iterate through all quadrants, compare each quadrants prefix with `qMin`/`qMax`.

There are 2^d quadrants.

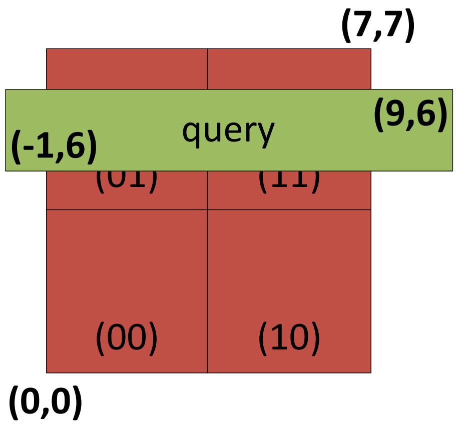
Can we do better?

Window queries – navigation in nodes

Example: `qMin/qMax = (-1,6)` / `(9,6)`

Example query execution on one node:

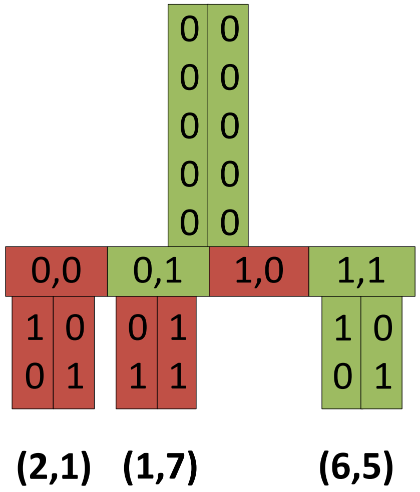
(0,0) =	(2,1)	→ mismatch
(0,1) =	(1,7)	→ postfix mismatch
(1,0) =	(1xx,0xx)	→ mismatch
(1,1) =	(6,5)	→ **match**!



Naïve approach
For each node, iterate through all quadrants:

- calculate each quadrants corners
- compare each corners with `qMin`/`qMax`.

→ ***O(d * 2^d^)*** per node



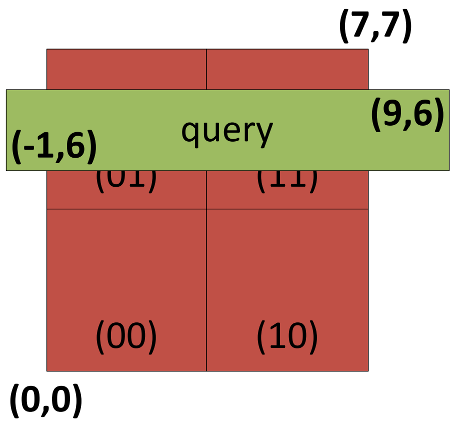
Can we do better?

Note: the ideas discussed on the following pages are partially quite complex. For a full discussion and proof please refer to [2].

Window Queries – minPos & maxPos

For each node we calculate two values `minPos` and `maxPos` as follows:

```
for (d = 0; d <= dimensions; d++) {
     center = calcCenterOfNodeFromPrefix()
     minPos[d] = qMin[d] >= center ? 1 : 0;
     maxPos[d] = qMax[d] >= center ? 1 : 0;
}
```



query: `qMin`(-1,6) / `qMax` (9,6)

center: (00000**1**00,00000**1**00)$_2$ = (4,4)$_{10}$

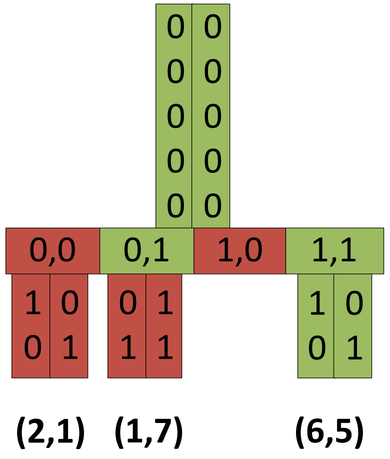
`minPos[0] = (-1 ≥ 4) = 0`

`minPos[1] = (6 ≥ 4) = 1`

`maxPos[0] = (9 ≥ 4) = 1`

`maxPos[1] = (6 ≥ 4) = 1`

→ `minPos` / `maxPos`: (**0**,**1**)/(**1**,**1**)



Note: `minPos[d]`/`maxPos[d]` refer to their `d`'s *bit*.

Window queries – navigation in nodes

How can we use `minPos`/`maxPos`?

→ `minPos` / `maxPos`:   (0,1)/(1,1)

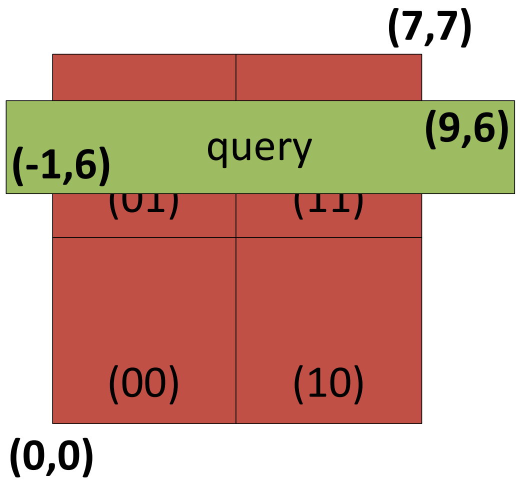
Idea #1

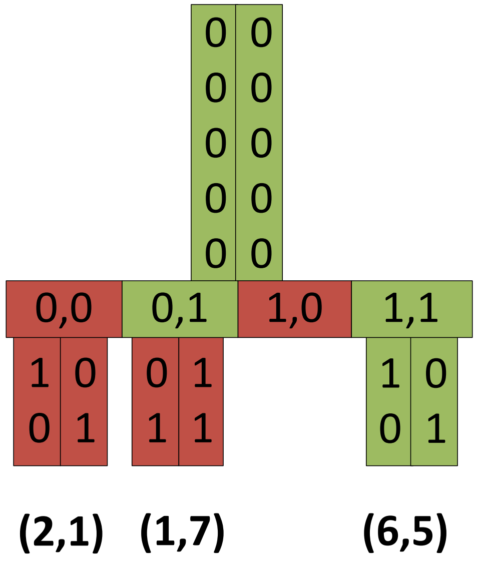
`minPos` and `maxPos` are the
lowest and **highest** overlapping quadrant!

In our example this removes only the first quadrant from iteration, but it may remove more. Also, it is cheap, calculated in ***O(d)*** per node!

However, we still have $\boldsymbol{O(d + d * 2^d)}$ per node

Can we do better?





Window queries – Check quadrant

Idea #2
A fast way to check if a quadrant at `pos` is applicable:

```
isValid = ((pos | minPos) & maxPos) == pos);
```

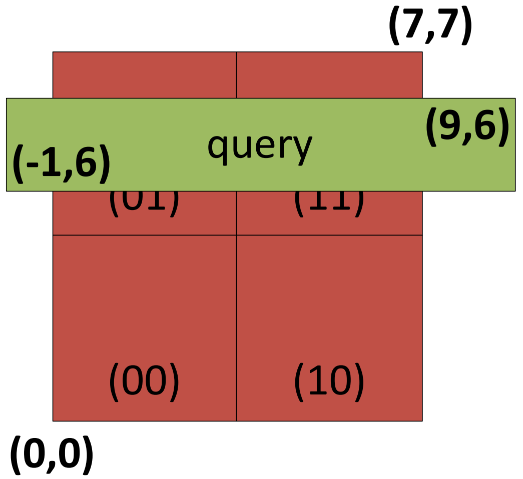
`minPos/maxPos` = (01)/(11)

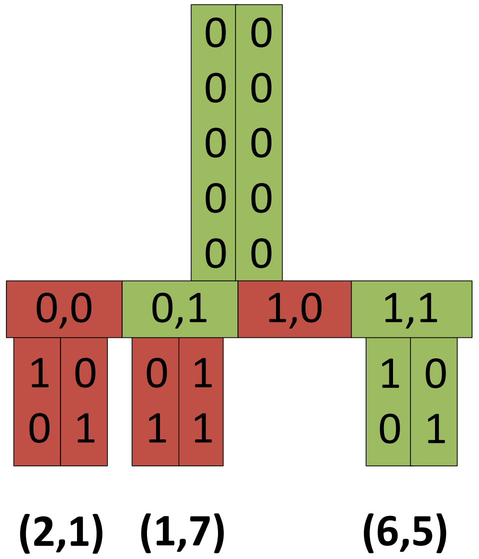
pos (0,1): `(01 | 01) & 11 = 01` ✓
pos (1,0): `(10 | 01) & 11 = 11 != 10` ✗
pos (1,1): `(11 | 01) & 11 = 11` ✓

`minPos` has `1` if positions with '`0`' should be skipped
`maxPos` has `0` if positions with '`1`' should be skipped

This reduces complexity from ***O(d + d*2^d)*** to ***O(d + 2^d)***

Can we do better?





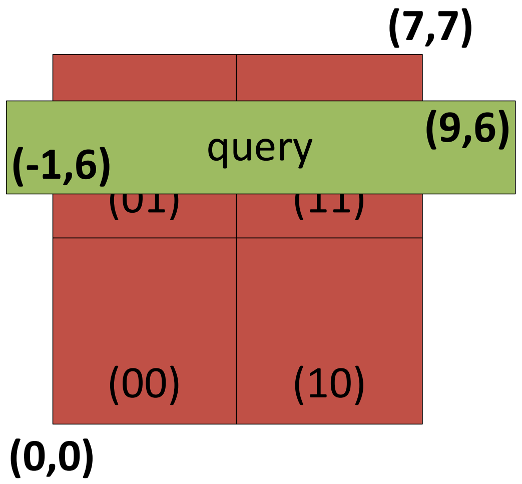
Window queries – Find next quadrant

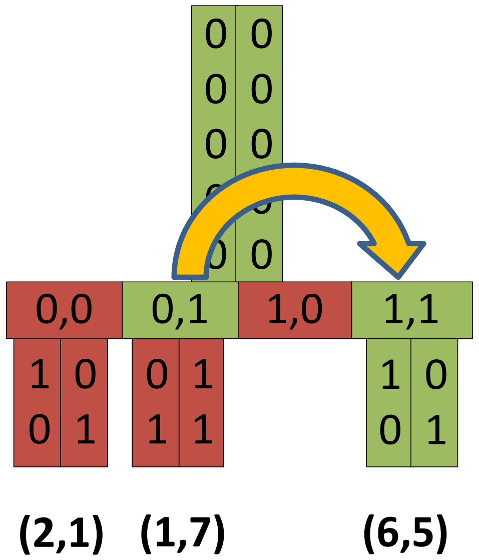
Idea #3
Find a function
`pos_next = inc(pos, minPos, maxPos)`
that returns the next matching quadrant.

If inc() executes in ***O(1)***, then complexity is still ***O(d + 2^d)***, however it is also optimal in the sense that the complexity is at the same time:
O(d + number_of_matching_entries).

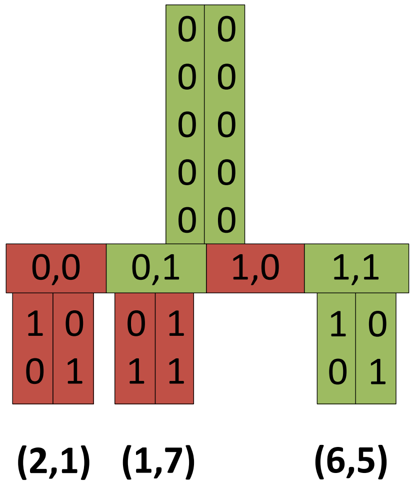
This is better than quadtrees, octrees or R-Trees which all have
O(d * number_of_ALL_entries) per node.

The next slide gives an overview over how `inc()` works. A full explanation and proof is available in [2].





Window queries – find next quadrant



Example: `inc(01) => (11)`

Use a trick:

→ in `pos`, set all bits to '`1`' where only one value is allowed (constrained bits).

```
pos = pos | minPos | (~maxPos) = 01 | 01 | 00 = 01;
```

→ now, if we add '`1`' for increment, any change to a constrained bit will cause an overflow and update the next higher bit.

```
pos++ => 01 + 1 = 10;
```

→ then we set all the constrained bits back to their previous value.

```
pos = (pos & maxPos) | minPos = (10 & 11) | 01 = 11;
```

→ `inc()` finds next slot in ***O(1)*** for $d < 64$

Window queries – the power of minPos / maxPos

For each node calculate `minPos`/`maxPos`: ***O*(*d*)**

Use these values:

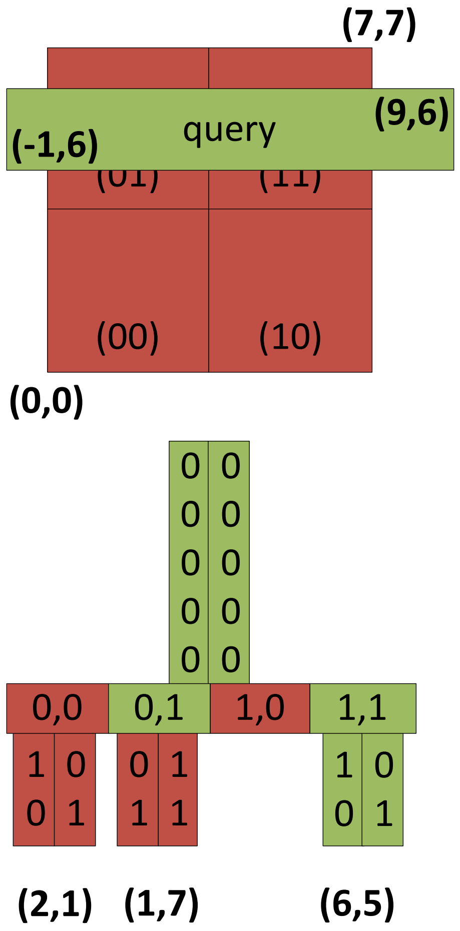
- as **start/end** point for iteration
- **check** whether any quadrant matches: ***O*(1)**

```
isValid = ((pos | minPos) & maxPos) == pos);
```

- **get next** valid quadrant: ***O*(1)**

```
long r = v | (~maxPos);
return ((++r) & maxPos) | minPos;
```

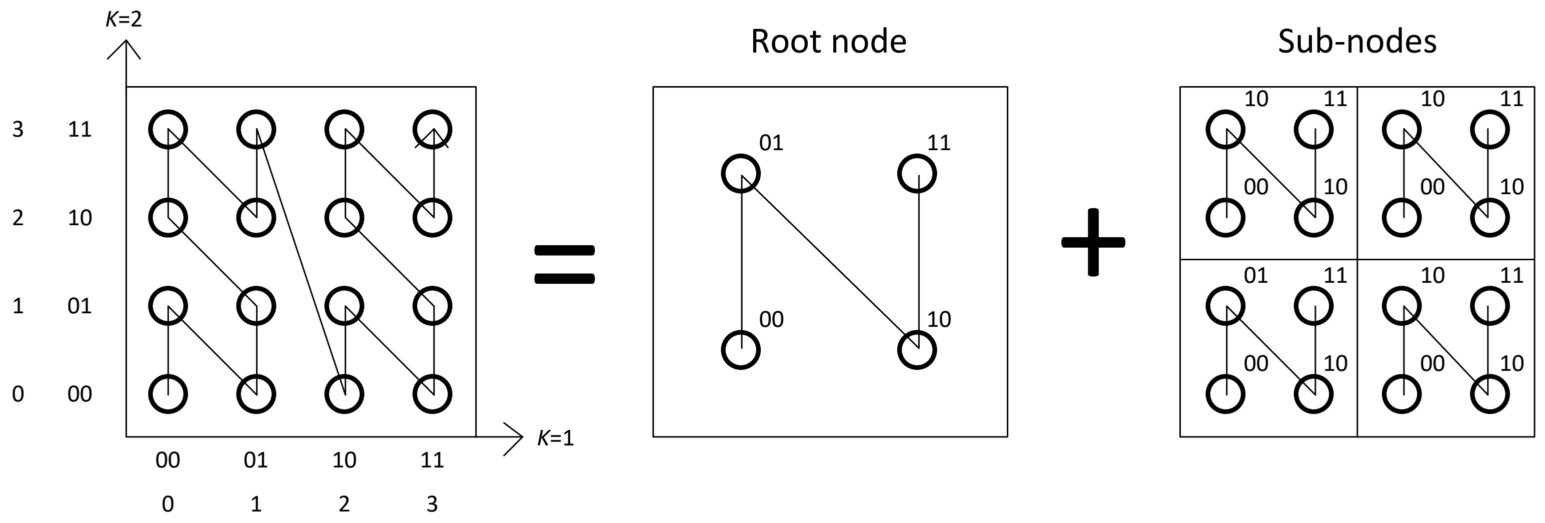
Note: in practice, inc() is rarely useful because it usually requires a O(log n) lookup (LHC/BHC) to find the nex pos. check() is much cheaper because it can just use an iterator.



This explanation is obviously very superficial. For details please refer to the original papers [1], [2], and see http://phtree.org

Window query – result ordering

When traversing the tree as described above (depth-first & traverse nodes in order of their HC-address) the result follows a Z-oder curve (also called Morton order):



Floating point keys – IEEE conversion

The PH-tree can natively only store *integer* keys. The following conversion allows **fast** and **lossless conversion** from floating-point to integer *and back*.

Java:

```java
long encode(double value) {
    long r = Double.doubleToRawLongBits(value);
    return (r >= 0) ? r : r ^ 0x7FFFFFFFFFFFFFFFL;
}
```

C++:

```cpp
std::int64_t encode(double value) {
    std::int64_t r;
    memcpy(&r, &value, sizeof(r));
    return r >= 0 ? r : r ^ 0x7FFFFFFFFFFFFFFFL;
}
```

Essentially, this takes the bit-representation of a floating point value and treats it as integer (after some conversion for negative numbers). This preserves the total ordering of values which is all that is required for insert(), remove(), point_query() and window_query().

Other operations, such as nearest_neighbor_query(), need to convert coordinates back to floating point data.

Floating point keys – multiply conversion

Another common conversion is strategy is to multiply the floating point value with a constant and casting it to an integer:

```
long encode(double value) {
    return (long) value * 100;
}
```

This is also **fast** an preserves **a certain amount of precision**.

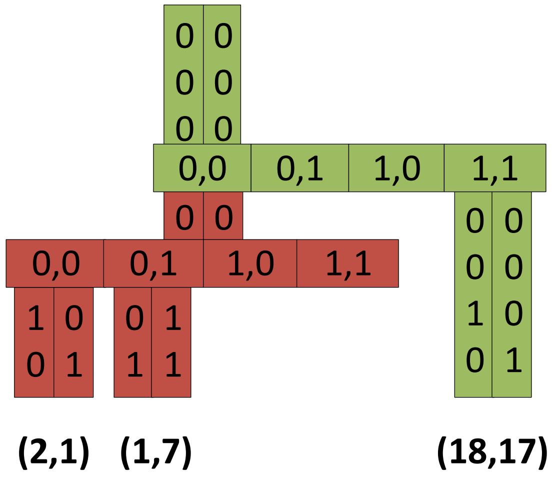
The main advantage resulting index tends to be **faster**. This effect hasn't been fully investigated, but there are two effects:

- The tree is more "dense", i.e. the length of the infix is more often zero. That allows skipping comparison of infixes when navigating the tree
- If the implementation supports prefix-sharing (bit-streaming): Normal integer values tend to have longer common prefixes, allowing for more prefix sharing, resulting in lower memory usage and better cache usage.

One (slight?) problem is that the rounding affects the precision of all operations: insert(), remove(), point_query(), window_query(), nearest_neighbor_query(), … .

Prefix sharing (bit-streaming)

All entries in a node share a common prefix.
E.g. on the example on the right, all entries in the red node share a common prefix `(00000,00000)`$_2$.



To save memory, we can store the prefix (actually infix) only once per node.

Using the example on the right:

- normal storage: 2 keys * 2 dimensions * 8 bit = 32 bit.
- prefix sharing:
  (1 prefix * 2 dim * 1 bit) + (2 keys * 2 dim * 2 bit) = 2 + 8 bit = 10 bit

The bits above the infix can be taken from the parent node. The bits above the postfix are equal to the array position.

This approach can safe a lot of memory, especially when used with multiply-conversion. With the Java implementation we saw performance improvements of 20%-30% for all operations.

Rectangles & boxes as key

The plain PH-tree can only store points (vectors) as keys. However, storing axis-aligned boxes as keys can efficiently be done by putting the two defining corners (minima and maxima) of a box into a single ***2*d*** key, for example by concatenating them:

`k = { min₀, min₁, …, min_{d-1}, max₀, max₁, …, max_{d-1}}`

$k = \{ min_0, min_1, …, min_{d-1}, max_0, max_1, …, max_{d-1}\}$

Example: a 2D box is stored in a 4D point: `(2,3)/(4,5) → (2,3,4,5).`

This works trivially for lookup, insert and remove operations. Window queries need to be converted from ***d*** d-dimensional vectors to ***(2∗d)***-dimensional vectors. For a window query that matches all boxes that are *completely inside* the query box, the query keys are:

```
kmin = { min0, min1, …, mind-1, min0, min1, …, mind−1 }

kmax = { max0, max1, …, maxd-1, max0, max1, …, maxd−1 }
```

Or, for a window query that matches all boxes that *intersect* with a query box:

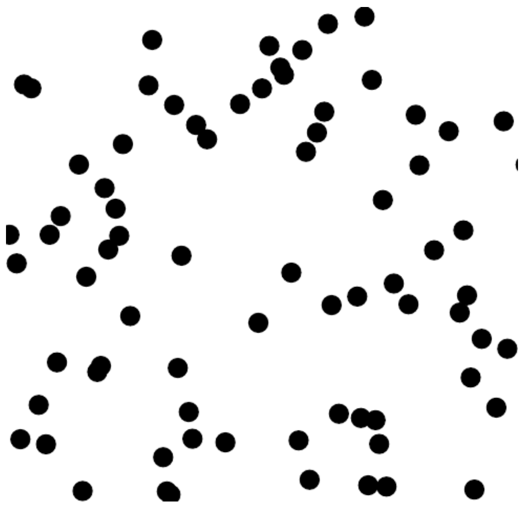
```
kmin = { −∞,     −∞, ...,     −∞, min0, min1, …, mind−1 }

kmax = { max0, max1, ..., maxd−1,   +∞,   +∞, …,    +∞ }
```

Example: qMin/qMax = `(3,4)/(4,5) → (−∞, −∞, 3, 4)/(4, 5, +∞, +∞)`

This is usually managed internally by the implementation. For more details please see [3].

Experimental evaluation (Java only)

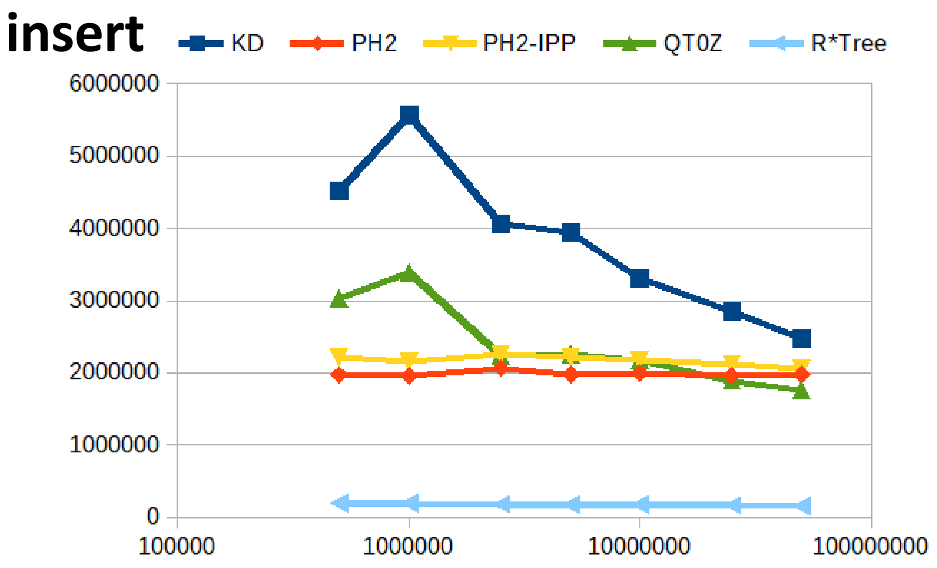
The following slides show experiments with strongly clustered dataset.
Each blob consists of 100s or 1000s of points (floating point coordinates).

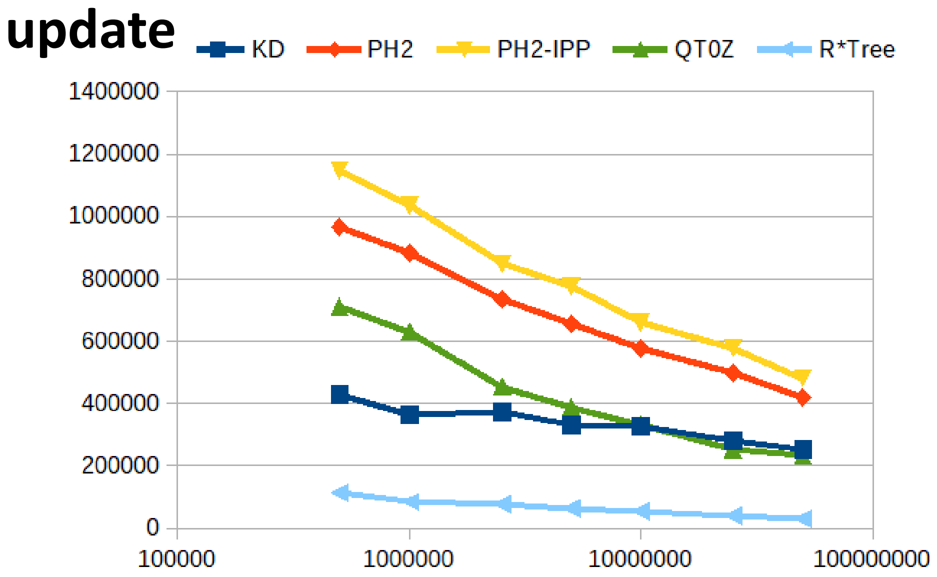


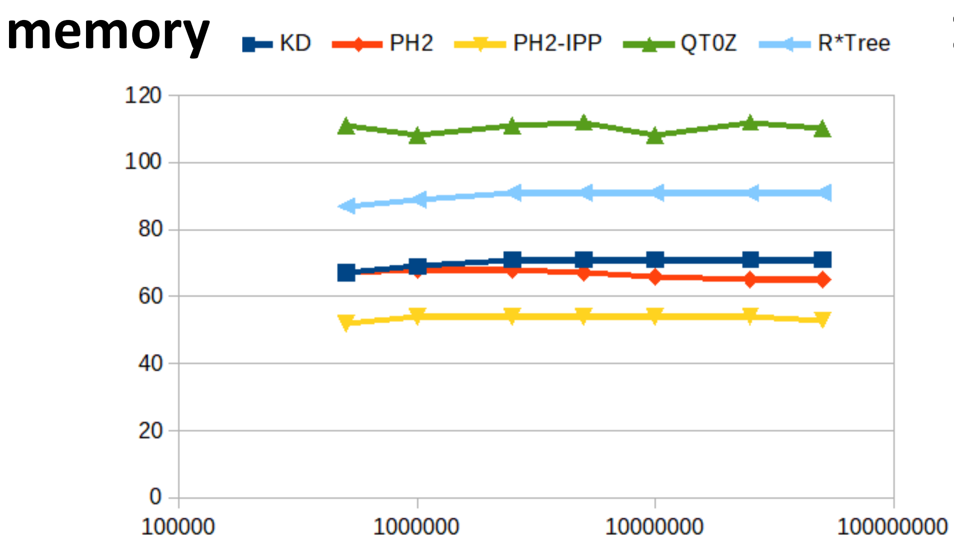
We compare multiple indexes: a kd-tree (KD), a quadtree (QT0Z), a R*tree (R*Tree), and two PH-trees, one with default conversion (PH2) and one with Integer-Multiply conversion (PH2-IPP). All are written in Java and available from:
https://github.com/tzaeschke/tinspin-indexes
Details and more experiments are available at:
https://github.com/tzaeschke/TinSpin/blob/master/doc/benchmark-2017-01/Diagrams.pdf

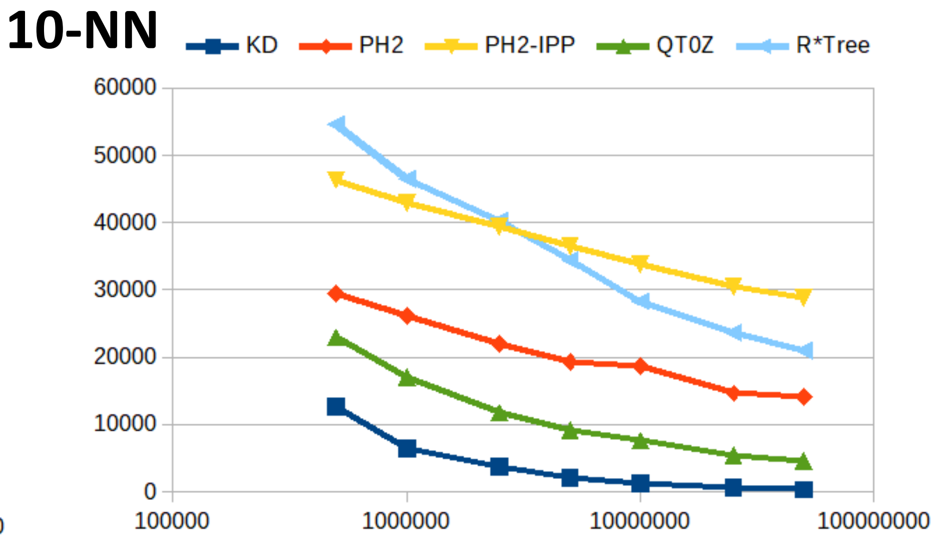
3D, varying size: 500’000 – 50’000’000

x-axis: # of points in index; y-axis: operations per second (for ‘memory’: bytes per entry)







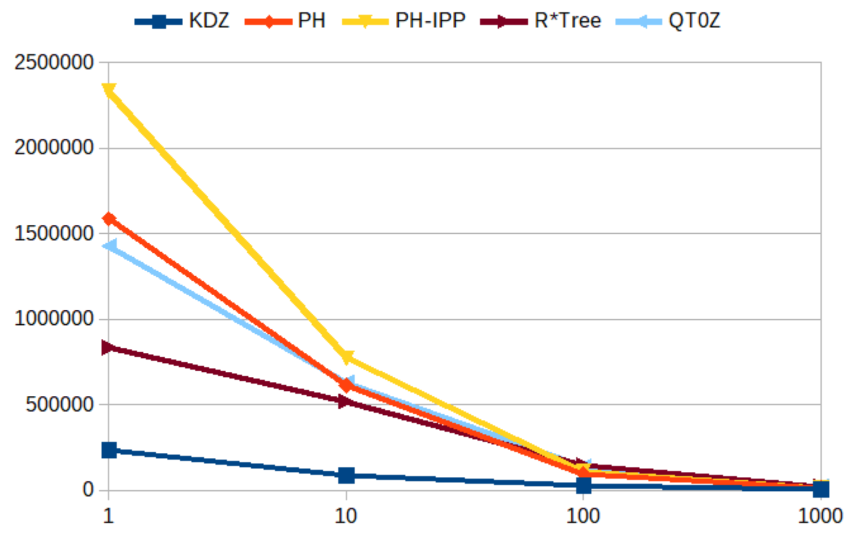


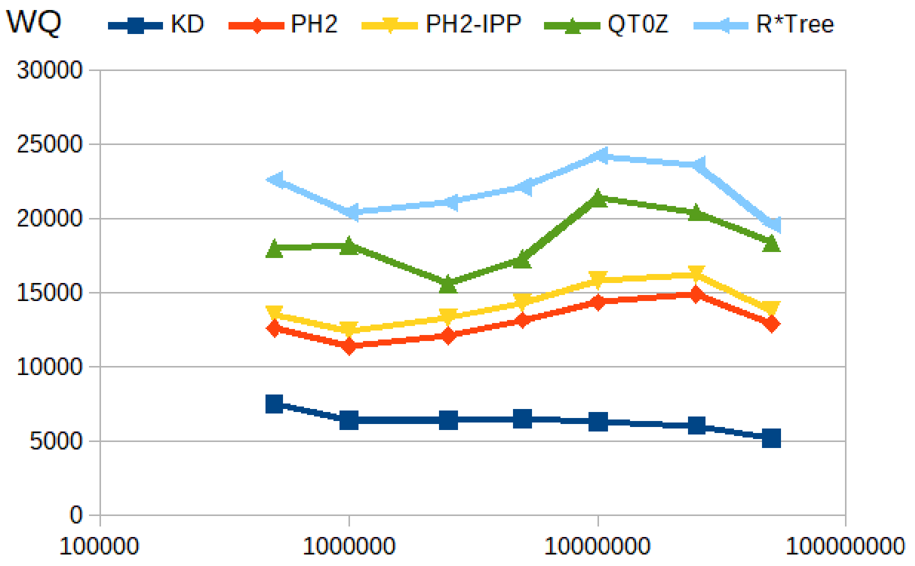
Window Queries – 3D

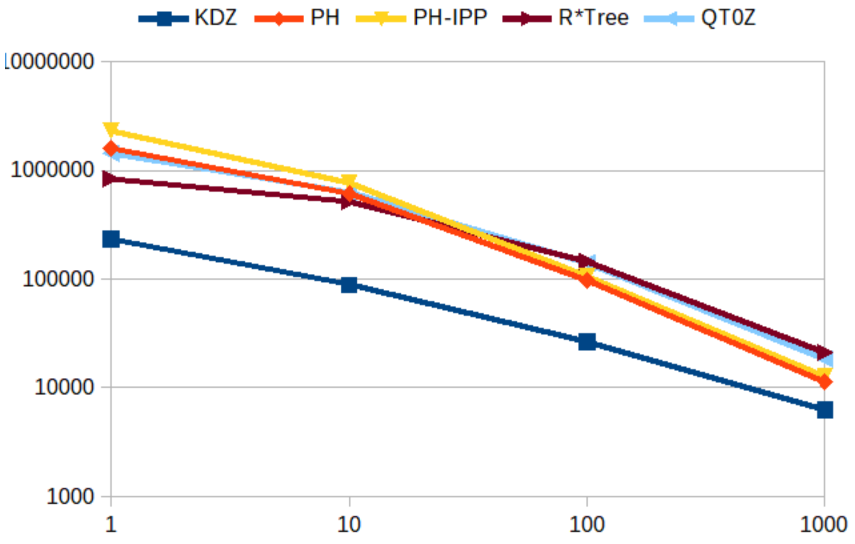
The number of points in a query result have a strong effect on query performance.

On the bottom we vary the index size.

On the right we vary the query window size on a dataset with 1M points, top uses logarithmic scale and bottom linear scale.





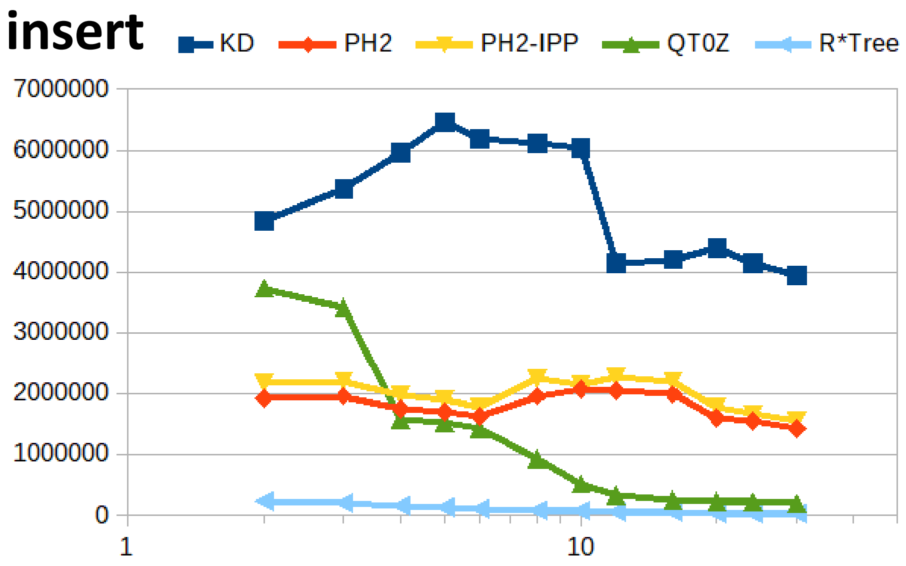


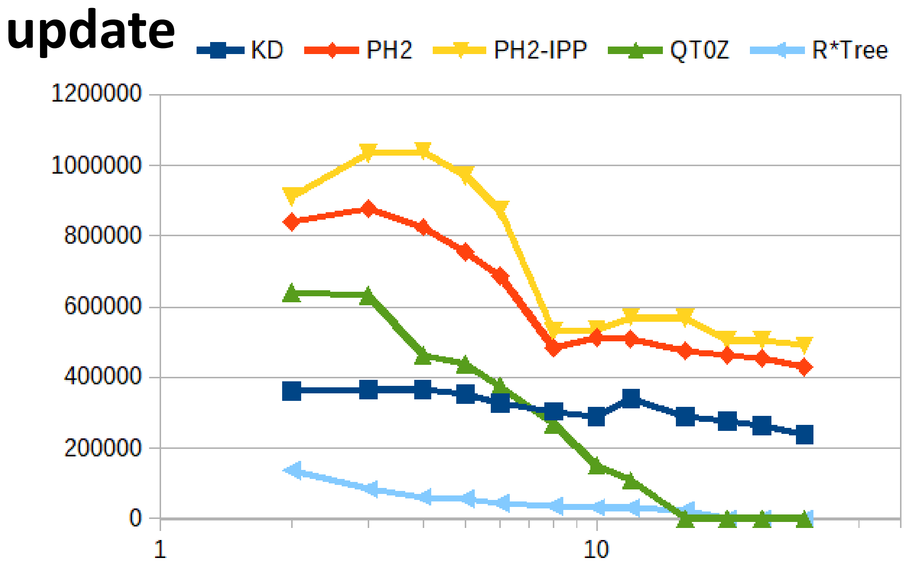
x: # of points in index; **y**: queries/sec; **constant**: 1000 results per query

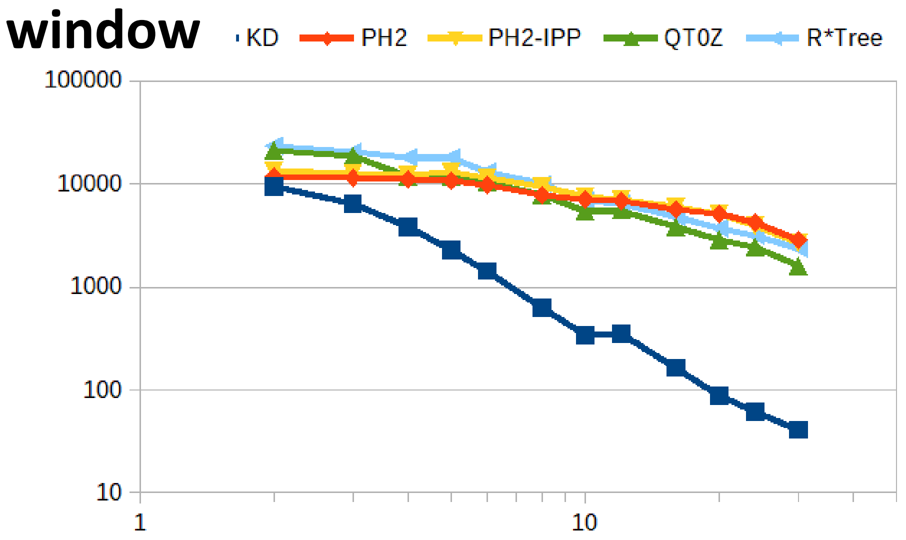
x: # of points in **query result**; **y**: queries/sec; **constant**: 10^6 points in index

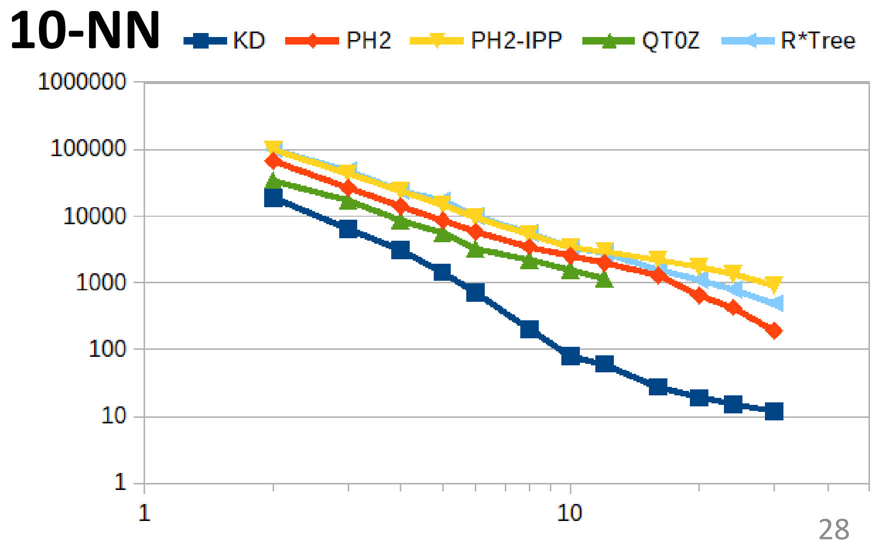
1M entries, varying *d*:   2 – 30

x-axis: # of **dimensions**; y-axis: operations per second









PH-Tree – Summary

General features

→ points or rectangle data
→ integer and floating point data
→ insert/move/remove, point query, window query, kNN search
→ Z-ordered

Static

→ no rebalancing: may benefit concurrency or persistence

Hypercube

→ exploits 64 bit **constant time** operations

Good with **large** and **clustered datasets**

Complex to implement

Source code:	http://www.phtree.org
Test framework:	http://www.tinspin.org

PH-tree

http://www.phtree.org

Test framework with other indexes
(kD-tree, quadtree, R*tree, STR-tree, cover-tree, ball-tree, …)

http://www.tinspin.org

PH-tree presentation 2022 © 2022 by Tilmann Zäschke is licensed under CC BY 4.0. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/

References – PH-tree

[1] **The PH-Tree: A Space-Efficient Storage Structure and Multi-Dimensional Index**; Tilmann Zäschke, Christoph Zimmerli and Moira C. Norrie, Proc. of Intl. Conf. on Management of Data (SIGMOD), 2014

[2] **Efficient z-ordered traversal of hypercube indexes**; PH-Tree: Tilmann Zäschke and Moira C. Norrie, Proc. of 17th BTW 2017, Stuttgart, Germany, 2017

[3] **The PH-Tree Revisited**; T. Zäschke (2015).

[4] **Cluster-Computing and Parallelization for the Multi-Dimensional PH-Index**; Bogdan Vancea (2015).

References – Other

Quadtree:

[5] **Quad Trees A Data Structure for Retrieval on Composite Keys**; R. Finkel and J. Bentley; Acta Informatica, 4:1-9, 1974.

PATRICIA-Trie:

[6] **PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric**; D. R. Morrison; Journal of the ACM, 15(4):514-534, October 1968.

Crit-Bit Tree:

[7] D.J. Bernstein, http://cr.yp.to/critbit.html

Various Slides

P: PATRICIA Trie

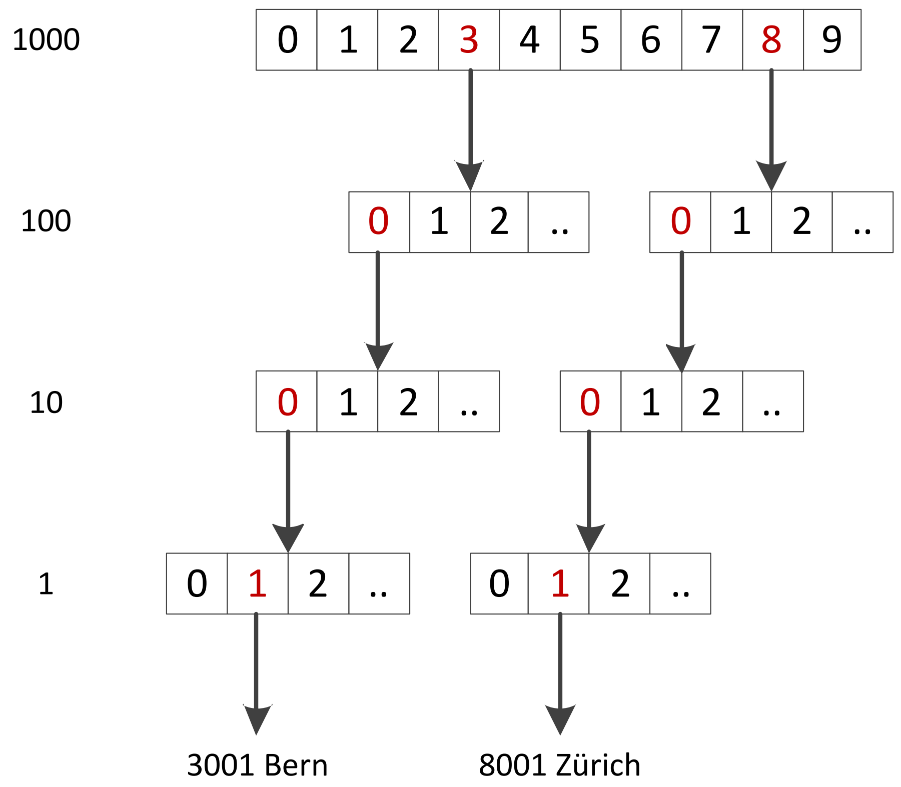
static tree

- concatenate key while walking down the tree
- height max = size of key
- no rebalancing

prefix sharing

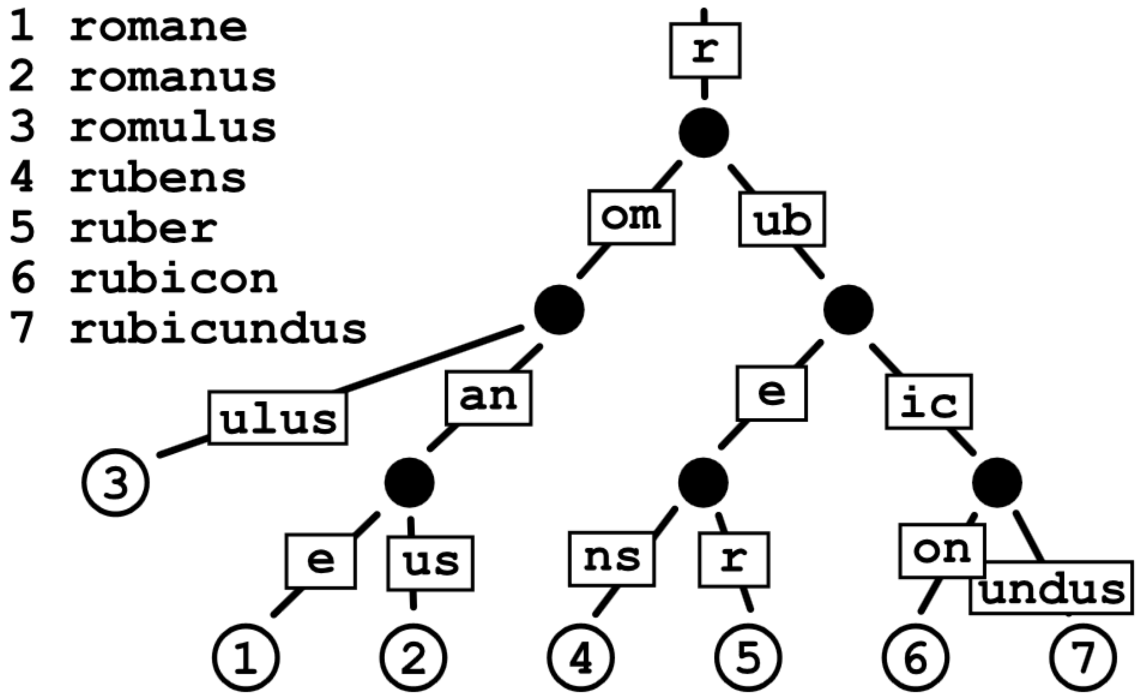
- storage shared with other keys

array of sub-nodes allows fast access



P: PATRICIA-Trie

Practical Algorithm To Retrieve Information Coded In Alphanumeric



→ List of sub-nodes i.o. array

By Claudio Rocchini, CC-BY 2.5