Efficient Z-Ordered Traversal of Hypercube Indexes

Tilmann Zäschke[1], Moira C. Norrie[2]

Abstract: Space filling curves provide several advantages for indexing spatial data. We look at the Z-curve ordering and discuss three algorithms for navigating and querying k-dimensional Z-curves. In k-dimensional space, a single hyper-'Z'-shape in a recursive Z-curve forms a hypercube with 2^k quadrants. The first algorithm concerns efficient checking whether a given quadrant of a local hyper-'Z' intersects with a global query hyper-box. The other two algorithms allow efficient Z-ordered traversal through the intersection space, either based on a predecessor from inside the intersection (second algorithm) or from any predecessor (third algorithm). The algorithms require an initialisation phase of $\Theta(k)$ for encoding the intersection envelope of the local hyper-'Z' with a range query. Using this envelope, all three algorithms then execute in $\Theta(1)$. The algorithms are however limited by the register width of the CPU at hand, for example to $k < 64$ on a 64 bit CPU.

Keywords: Multi-dimensional index, spatial index, PH-tree, binary hypercube, Z-curve, Z-ordering, window queries

1 Introduction

We discuss the problem of efficiently traversing partitions of binary hypercubes. This problem occurs for example in certain multi-dimensional indexing structures when executing window queries. Part of the problem is to efficiently determine for increasing dimensionality k whether points or regions in the trees intersect with a k-dimensional query hyper-box. In the case of indexing structures such as the UB-tree [Ba97, Ma99], the BUB-tree [Fe02], the theoretical HQ-tree [Sk06], or the more recent BQ-tree [ZYG11] or PH-tree [ZZN14], the multidimensional data is often (partially) interleaved to form bitstrings. Depending on the encoding, these bitstrings impose a natural Z-order on the stored keys, hence they are called *Z-addresses*. When a Z-address is sliced into sub-strings of k bits, each slice can be seen as the address (called H-address) of a quadrants of a k-dimensional binary hypercube. If we define that every quadrant can contain at most one child (sub-node or point) then nodes may often have up to 2^k children[3] and every child has its own H-address.

Window queries are usually implemented as a type of node traversal while avoiding nodes and their children if they cannot potentially intersect with the query box. To perform efficient window queries, it is therefore necessary to traverse all necessary nodes and sub-nodes while efficiently avoiding any unrelated nodes. Just as the tree is represented by a hierarchy of nodes, i.e. binary hypercubes, each query box can be seen as the envelope of all potentially matching values and thus can also be seen as a (convex) hierarchy of binary

[1] ETH Zurich, Department of Computer Science, Universitätsstrasse 6, 8092 Zurich, zaeschke@inf.ethz.ch
[2] ETH Zurich, Department of Computer Science, Universitätsstrasse 6, 8092 Zurich, norrie@inf.ethz.ch

[3] The advantage of such large nodes may not be intuitive but has been explained and demonstrated in [ZZN14, Zä15, Ch15, BCA15, LH14].

hypercubes. Window queries therefore represent the problem of intersecting hierarchies of binary hypercubes and then traversing these intersections. Related to the window queries are some algorithms for skyline [BKS01] queries, i.e. queries for non-dominated vectors. Some of the more recent algorithms [Ch15, BCA15, LH14] partition the space hierarchically around pivot points and then use H-addresses to reduce the search space.

In this paper, we discuss three algorithms for efficiently navigating intersections of k-dimensional binary hypercubes. All three algorithms can be used for traversing intersections, but they exhibit different strength depending on the representation of the hypercube in memory and on the size of the intersection compared to the size of the hypercube. The 1st algorithm has been used before but without explanation [ZZN14] or only partial explanation in [Ch15, BCA15]. The 2nd and 3rd algorithm are contributions of this paper.

The first algorithm **isInI**(h, I) checks whether a H-address h lies in the intersection I of a node N with a query hyper-box. This is useful when we traverse the whole hypercube $\mathbb{H}^k$, for example if $|I|$ is approaching $|\mathbb{H}^k| = 2^k$.

The second algorithm **inc**$(h_{I,a})$ takes a H-address $h_{I,a} \in I$ and returns its successor $h_{I,b} > h_{I,a}$ in I. This allows efficient traversal of I if $|I| \ll 2^k$ by omitting any $h \notin I$.

The third algorithm **succ**(h) takes any H-address h and returns the next smallest $h_I > h$ that lies in I. **succ**(h) is a generalisation of **inc()** that accepts any $h \in \mathbb{H}^k$ which can be useful for large hypercubes where it can make sense to switch between the first and second algorithm.

All algorithms require encoded intersection information which can be calculated in $\Theta(k)$ for each binary hypercube of the tree. Using this intersection information, all three algorithms complete in $\Theta(1)$ on a CPU with at least $(k + 1)$ bits per register, for example for $k < 64$ on a 64 bit CPU. This differentiates us from earlier work which usually processes only one bit at a time, for example [KPS91].

Algorithms similar to **isInI**(h, I) have already been presented in research [ZZN14, Ch15, BCA15, LH14] where they are used to skip quadrants of the binary hypercube. However, while using **isInI**(h, I) is efficient if only few quadrants have to be skipped, the algorithm does not scale well with larger dimensionality k where only a minority of exponentially growing number of quadrants should be visited. For these cases we present the novel algorithms **inc**$(h_{I,a})$ and **succ**(h) which allow jumping directly to the next valid quadrant.

This paper is structured as follows. After presenting related work in Sect. 2 we define terminology in Sect. 3. Sect. 4 describes how I is calculated, followed by the sections 5, 6 and 7 which describe the three algorithms. After that, in Sect. 8, we provide a discussion how and when the algorithms could be used, followed by a concluding discussion in Sect. 9.

2 Related Work

Window queries on Z-curves [OM84] are relevant for efficient implementations of multi-dimensional indexes such as the Universal B-tree (UB-tree) [Ba97, Ma99, Ra00], the

BUB-tree [Fe02] or the PH-tree [ZZN14, Zä15][4], see for instance [Sa06] for other examples. As proposed by [TH81], these trees interleave some or all bits of each dimension of a stored k-dimensional point $p = p_0, p_1, ...p_{k-1}$ into one bit string. We call this bitstring *Z-address* z since it can be seen as a coordinate in a space-filling Z-curve, i.e. the lexicographic ordering of values encoded in Z-addresses is the Z-ordering, see [Sa94] for a discussion. A comparison of Z-curves with other space filling curves is given in [MAK03].

In the following we assume for simplicity that all attributes p_i of an entry p have the same number of bits w, however the algorithms can be adapted to values with different lengths with few changes. For a k-dimensional tree with Z-ordering we can split the Z-address into w chunks of k bits called *H-addresses* h which represent addresses in binary hypercubes. In effect, an H-address h_v is a cross-section through all $p_{0 \le i < k}$ of a point p, taking the vth bit from each p_i.For example, a 2-dimensional point $p = p_0, p_1$ with $p_0 = 100$ and $p_1 = 010$ can be interleaved to a Z-address $z = 100100$. z is then sliced into $w = 3$ chunks of $k = 2$ bits each where the three chunks represent the H-addresses in the binary hypercube of the root node $h_{d=0} = 10$, an inner node $h_{d=1} = 01$ and a leaf node $h_{d=2} = 00$.

In each binary hypercube, we define H-addresses $h \in \mathbb{H}^k = \{0, 1, ..., 2^k - 1\}$ as the lexicographically ordered sequence of points which, when each connected with their predecessor and successor, form a k-dimensional binary Z-curve. For example, lets assume a tree consisting of a hierarchy of nodes where each node represents a hypercube. In every of the tree's hypercubes/nodes, if a quadrant of a hypercube contains geometrically more than one of the points in the tree, then the quadrant simply references a sub-hypercube/node of the same size as the quadrant. These sub-hypercubes split the space recursively further until every quadrant contains at most one point.

For window queries, the query hyper-box (the term *box* denotes that it is not necessarily a cube) is intersected with the hypercubes of the nodes, where the quadrants of the nodes are identified by H-addresses. The idea is that when we search for entries that match the query, we want to traverse only those H-addresses/quadrants in a node that intersect with the query hyperbox and who can potentially contain matching sub-nodes or points.

The naive approach is to iterate through all non-empty quadrants in a node and test whether they intersect with the query. How the H-address of a quadrant can be checked for intersection in constant time is described in [ZZN14]. This works well if the intersection I occupies a large part of the node's hypercube. For $k = 2$ it cannot be smaller than $\frac{|I|}{\mathbb{H}^k} \ge \frac{1}{2^k} = \frac{1}{4}$, but for growing k and small selectivity the naive approach becomes exponentially inefficient. An improved version with min/max quadrants (see Sect. 4) is described in [Ni13]. However, they still appear to check all quadrants between min and max in order to skip them.

In this paper we propose an approach that, instead of iterating through all available quadrants, generates only H-addresses that intersect with the query and then checks whether the H-address exists in the hypercube of the current node. The algorithm requires $\Theta(k)$ initialisation effort for each visited node and then generates only H-addresses that are part of the intersection in $\Theta(1)$ for each H-address on a CPU with more than k bits per register.

[4] [ZZN14] uses the term 'range queries' for queries on rectangular windows.

More recently, binary hypercube partitioning is also used by skyline algorithms [BKS01], i.e. algorithms that determine all non-dominated vectors in a dataset. Unlike the index trees above, they usually do not split the space in half according to the binary representation of the coordinates, but instead split at specially calculated 'pivot' points [Ch15, BCA15, LH14]. However they still use H-addresses and partially describe algorithms similar to our initial **isInI**(h, I) algorithm. Their description and use of **isInI**(h, I) is partial in the sense that they only consider cutting away the 'upper' half of the cube in each dimension. As discussed in Sect. 6, cutting away the lower half works mostly, but not completely, symmetric.

3 Terminology

In this work we refer to *constant time* operation as anything that can be executed on a computer's CPU in a constant amount of CPU operations. For example, all $\Theta()$ and $O()$ references refer to execution complexity on a CPU with at least $(k + 1)$ bits per register, unless stated otherwise.

Definition 1. *(Point p) A point p represents an entry in a tree structure. p is a k-dimensional point where each dimension is represented by a value* $p_{0 \le i < k}$ *with w bits, i.e. each value* p_i *is an integer with* $0 \le p_i < 2^w$. *For simplicity we ignore negative integers and floating point values, even though they are also supported by the algorithms, possibly with minor modifications*[5]*. We also assume that all values* p_i *have the same number of bits w.*

Definition 2. *(Z-address z) A Z-address z is a bitstring consisting of the* $k \times w$ *bits of a point p. The first k bit of z represent the first (highest order) bit of each value* p_i*, the next k represent the second bit, and so forth.*

Definition 3. *(H-address h) Let* $h \in \mathbb{H}^k = \{0, 1, ..., 2^k - 1\}$. *A hypercube address h is any subsequence of k bits starting at a multiple of k of a bitstring z. Any z thus can be seen as a sequence of w H-addresses which designate a point p in the hierarchical hypercube of the index tree. Inside the node of a tree, each h acts as a key to a point that is stored in the node or to a subnode.*

Definition 4. *(Sets I, N and R)* $N \subseteq \mathbb{H}^k$ *denotes the set of all h that are stored in a node, either in the form of points or subnodes.* $I \subseteq \mathbb{H}^k$ *is the set of h that potentially contains points or subnodes that intersect with the query box.* $h \in I$ *do not necessarily exist, i.e. I may or may not be a subset of N. Finally,* $R \subseteq \mathbb{H}^k$ *is the result set containing all h that lie in I and in N, i.e.* $R = I \cap N$.

We denote the bitwise binary operations as follows: '&' (AND), '|' (OR), '~' (NOT) and '⊕' (XOR). The listings use the same notation except ˆ for ⊕.

4 Encoding the Shape of *I*

In order to efficiently traverse an intersection I, we use an efficient encoding of the shape of I. We do this by means of two bit sequences, m_0 and m_1. m_0 and m_1 each consist of k bits

[5] See [ZZN14] for a related discussion.

where each bit in m_0 specifies whether the '0'-half of that dimension of the hypercube is part of I or not. Accordingly, m_1 specifies the '1'-half of the dimension.

Definition 5. *(Range filters m_0, m_1). We encode the intersection I of the hypercube, with the query box in two bit strings, m_0 and m_1. For each dimension $i : 0 \leq i < k$, m_0 has a '0' at position i iff the '0' quadrant of that dimension is part of the intersecting body. Accordingly, m_1 has a '1' at position i iff the '1' quadrant of dimension i is part of the intersection.*

Definition 6. *($m_{0,i}$ & ~$m_{1,i} \equiv 0$). We define that m_0 and m_1 never restrict on the same dimension. In other word, m_0 and ~m_1 never have a '1' bit at the same position. If they would restrict on the same dimension, neither '0' nor '1' would be allowed for $h \in I$ at that position and the intersection $I = \emptyset$ would be empty. In this case, the current node should have never been entered because it can not possibly contain any results.*

Corollary 1. *($m_0 \leq m_1$). $m_0 \leq m_1$ follows implicitly from Def. 5 and Def. 6 because each bit in m_0 is necessarily smaller or equal to the according bit in m_1. If m_0 is bitwise smaller or equal to m_1, then $m_0 \leq m_1$.*

Corollary 2. *(m_0, $m_1 \in I$ and $\forall h : (h \in I \rightarrow m_0 \leq h \leq m_1)$). m_0 and m_1 are valid H-addresses with $m_0, m_1 \in I$. If we were to construct a minimum value $h_{min} \in I$, we would set all bits to '0', except those bits that are required to be '1'. This is identical to how we construct m_0, therefore $m_0 = h_{min}$ and hence $m_0 \in I$. The argument for m_1 goes analogous. While m_0 and m_1 define the minimum and maximum values for h through lexicographic ordering, not all values $h \in [m_0, m_1]$ are in I, i.e. $h \in I \rightarrow h \in [m_0, m_1]$ is a one way implication. $\forall h : (h \in I \rightarrow h \in [m_0, m_1])$ but $\neg\forall h : (h \in [m_0, m_1] \rightarrow h \in I)$. As a result, $|I| \leq |[m_0, m_1]|$.*

It is important to understand that m_0 and m_1 play a double role by encoding the extent of the intersection I while also being the minimum and maximum possible values for any $h \in I$. Figure 1 shows an example with a 3-dimensional (hyper)-cube split in two parts, the front with $y = 1$ and the back with $y = 0$. If we assume an I that consists only of the half of the cube that has $y = 1$ while $x, z = 0, 1$ are unconstrained then we would get $m_0 = 010$ for constraining y's lower dimension and $m_1 = 111$ because there are no other constraints. As we can see, m_0 and m_1 represent the numerical minimum and maximum of the intersection I which consists of the front of the cube.

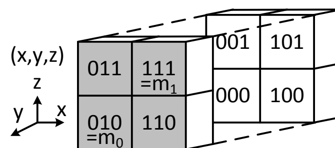

Fig. 1: Hypercube with $k = 3$ where I (grey area) is constrained to y = '1'

Corollary 3 (Size of $N : |N|$)**.** *The size of N can be calculated from m_0 and m_1. Let $n_{1,m0}$ be the number of '1' bits in m_0 and $n_{0,m1}$ be the number of '0' bits in m_1. Then the number of dimensions with restriction $k_r = n_{1,m0} + n_{0,m1}$. Since N extends only in the non-restricted dimensions, N becomes $\mathbb{H}^{k-k_r}$ and the size of N becomes $|N| = 2^{k-k_r}$.*

Since counting '1' bits is a constant time operation on typical modern CPUs, the size $|N|$ can be efficiently computed with (Example in Java):

```
//mask to avoid bits with i>=k
long mask = ~(-1<<k);
//count `1'-bits
int k_r = Long.bitCount(m0 | ((~m1) & mask));
//power of 2^(k-k_r)
int sizeN = 1 << (k - k_r);
```

If we look at the Z-ordered traversal in I, we see that the gaps between h values in I are irregular, see for example the gap between '011' and '110' in Fig. 1. A dimensional restriction on a highly significant bit will cause a big gap while a less significant bit will cause a small gap.

Corollary 4 (Gap size)**.** *We assume an N with a single restriction on one dimension i with* $0 \leq i < k$. *If we want to traverse the gap then we need to move on by* 2^i *entries before we find the next valid entry. The size of the gap, i.e. the number of invalid entries, is* 2^i. *If multiple gaps are crossed at the same time, the total width of the gap is the sum of the individual gaps. The largest possible gap occurs if all restrictions are crossed at the same time. In this case, without further proof, the size of the gap is simply the number resulting from* $(m0 \mid \sim m1) + 1$.

5 Algorithm 1: isInI(h,I)

The **isInI**() algorithm checks whether an H-address h intersects with I. To specify the shape of I, we use m_0 and m_1 because they carry all relevant information and uniquely specify I. The function has therefore the parameters **isInI(h,m0,m1)**. We define:

Definition 7. *(Bitstring* b_X *and bits* $b_{X,i}$*) We define b as the bitstring of a given integer number. We use* b_h, b_{m0}, b_{m1} *to indicate that we discuss the bit representation of h,* m_0 *and* m_1 *respectively.* $b_{h,i}$, $b_{m0,i}$, $b_{m1,i}$ *is the ith bit of b, starting with* $i = 0$ *for the least significant bit.* $0 \leq i < k$ *irrespective of the number of leading '0' bits.* $b_{\{h,m0,m1\},i}$ *is undefined for* $i < 0$ *and assumed to be '0' for* $i \geq k$.

While I is continuous in space, see grey area in Fig. 1, it is not contiguous in terms of the numeric sequence of its h-addresses. Therefore, it is not sufficient to check if $m_0 \leq h \leq m_1$. For example, as shown in the figure, the grey intersection with the query box consists of the non-contiguous sequence $\{010, 011, 110, 111\} = \{2, 3, 6, 7\}$.

A naive algorithm could check for each bit $b_{h,i}$ (with $0 \leq i < k$) in h whether it is compatible with the according bits $b_{m0,i}$ in m_0 and $b_{m1,i}$ in m_1. If $b_{m0,i} = 1$ and $b_{h,i} = 0$ or $b_{m1,i} = 0$ and $b_{h,i} = 1$, then $h \notin I$ otherwise $h \in I$. This approach would require $2 * k$ bit comparisons.

A constant time version is presented in [ZZN14], however with little explanation how or why this works. The algorithm in [ZZN14] is as follows:

```
boolean isInI(long h, long m0, long m1) {
  if ((h | m0) != h) {
    return false;
  }
  if ((h & m1) != h) {
    return false;
  }
  return true;
}
```

List. 1: **isInI(), original version from [ZZN14]**

Lemma 1. ***isInI***(h, m_0, m_1) *= true iff* $h \in I$.

Proof. We limit the proof to one individual bit b_i because ⊕ (XOR), & (AND), | (OR) and ~ (NOT) are bitwise operations that process one bit from each operand without interfering with other bits. Therefore, if the proof works for a single bit, it implicitly works for any number of bits. The first check in Lst. 1 (‘`if (((h | m0) & m1) != h)`’) can only return `false` if $h_i \mid m_{0,i} \neq h_i$. This can only happen if $(h_i = 0) \wedge (m_{0,i} = 1)$, which indicates, see Def. 5, that h refers to the ‘0’ half of the hypercube while only the ‘1’ half is part of I. Accordingly, for the second term `h & m1 != h`, this can only return `false` if $(h_i = 1) \wedge (m_{1,i} = 0)$, which indicates, again see Def. 5, that h refers to the ‘1’ half of the hypercube while only the ‘0’-half is part of I. As a result, **inc()** returns `false` iff $h \notin I$. □

Our optimisation is almost trivial, it uses the fact that $m_{0,i}$ and $m_{1,i}$ can never restrict on the same dimension because $m_0\&(\sim m_1) \equiv 0$, see Def. 6. Using this fact, the code can be simplified to

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

List. 2: **isInI(), final version**

In other words, if m_0 requires any bit to be ‘1’ which is not set in z, or if m_1 requires any bit to be ‘0’ which is ‘1’ in h, then the comparison with the original h will fail and the function reports a mismatch.

Lemma 2. $h|m_0\&m_1 = h \Leftrightarrow h \in I$

Proof. Building on Lemma 1, Lst. 2 can only behave different from Lst. 1 if m_0 and m_1 would affect the same bit and m_0 would be the inverse of m_1, i.e. if `(h | m0) != h` but `((h | m0) & m1) == h`. However, m_0 and m_1 can never affect the same bit, see Def. 6, hence both algorithms show identical behaviour. □

From the code it is obvious that it executes in constant time as long as all values do not have more bits than a CPU register.

6 Algorithm 2: inc(h)

In cases where I is much smaller, see Corollary 3, than the surrounding binary hypercube, i.e. $|I| \ll 2^k$, checking each entry with **isInI(h,I)** is not very efficient. Instead, it is desirable to have a way of directly generating h-addresses that are part of the intersection. One approach is an algorithm that takes one $h_{in} \in I$ as input and generates the next bigger $h_{out} \in I$, thus allowing to generate a complete set of all $h \in I$. To this end we propose the **inc**(h_{in}, m_0, m_1) algorithm that produces all $h \in I$. The starting value for h_{in} would be m_0, i.e. **inc**$(h_{in} = m_0, m_0, m_1)$.

The problem is that I is not a necessarily contiguous sequence of H-addresses but can have numerous gaps of different length. For example, the I in Fig. 1 is $\{010, 011, 110, 111\} = \{2, 3, 6, 7\}$ which has a gap of two in the middle but no gap between the other elements.

That means, depending on the current h_{in}, **inc()** has to add a different $\delta h_{in,out}$ to get h_{out}. We can see that $\delta h_{in,out}$ is related to m_0 and m_1. If the ith bit (counting i from the lowest bit) of m_0 is 1 ($m_{0,i} = 1$) and the ith bit of h_{in} is '0' after adding '1', then we must add an additional 2^i in order to switch the bit back to an acceptable value. The algorithm for m_1 is similar. Such an algorithm that has to verify each bit obviously executes in $O(2 * k)$ since h, m_0 and m_1 have each k bits. Based on the example in Fig. 1, the transition from $h = 011$ to $h = 110$ would look like this:

```
--> start: h=011
h <- h+1 = 100
--> conflicts with m1, bit i=1
h <- h+(2^1) = 110
--> okay
```

An obvious approach to achieve better runtime than $O(k)$ is to find a way to calculate $\delta h_{in,out}$. While we are not aware of a way to do this, we achieve the desired result by exploiting the CPU's *add* operation. The problem is that, if we add '1' to a bit, a possibly resulting overflow should not necessarily go to the immediate next higher bit, but to the next higher bit that is unrestricted and that can therefore be flipped. The trick is to let the CPU's *add* operation skip over the bits that are restricted by m_0 and m_1. To do this, we set all bits that should be skipped to 1. Any overflow will then be forwarded to the next '0' bit in constant time. Since the input h is already a valid H-address (per definition) the bits restricted in m_0 are already set to '1' and we only need to set the restricted bits from m_1 to '1', because they will all be '0' due to the restriction. This can easily be done with

```
h = h | (~m1); //set filtered bits to `1'
```

If we now increment h by one, all bits on which we have restricted create an overflow, unless the overflow is swallowed by a lower order bit that was '0'.

Now we just have to make sure that we turn h back into a valid value by setting and unsetting the restricted bits. This can be done by

```
h = (h & m1) | m0; //restore filtered bits
```

The resulting function is:

```
long inc(long h_in, long m0, long m1) {
  long h_out = h_in | (~m1); //pre-mask
  h_out++;                   //increment
  h_out = (h_out & m1) | m0; //post-mask
  return h_out;
}
```

List. 3: **inc(), improved version**

This function produces an ordered sequence of $h \in I$ with $\Theta(1)$ per h. Note that this function returns $h_{out} \leq h$ if the incoming $h_{in} = m_1$. The reason is that in a practical implementation, m_1 should have all bits b_i for $i \geq k$ set to '0', which means that $h_{out} \leq h$.

Corollary 5 (Stop condition)**.** *If the input is $h_{in} = m_1$ then $h_{out} = m_0$, unless the CPU performs signed computation and the overflowing bit causes a sign change to a negative value. Also, in the special case of $m_0 = m_1 \rightarrow h_{out} = h_{in}$ because $h = m_1 \rightarrow h_{out} = m_0 = m_1 = h_{in}$.*

We split up the proof of **inc()** in three parts. First we show that **inc()** wraps around from m_1 to m_0, i.e. that **inc**$(m_1, I) = m_0$.

Lemma 3. ***inc***$(h_{in} = m_1, m_0, m_1) = m_0$. *This wrap around condition means that the highest possible values $h_{in} = h_{max} \in I$, which is m_1, results in the lowest possible value, m_0. This condition ensures that h_{out} =**inc()** always produces $h_{out} \in I$, however, as we will see, at the cost of breaking the $h_{in} > h_{out}$ rule (Lemma 5).*

Proof. In the first operation all bits are set to '1': $h_{out} = m_1 \mid \sim m_1 = \{1\}^k \rightarrow h_{out} = 2^k - 1$. In the second step we add 1, resulting in $h_{out} = 2^k$, which is all '0' with a leading '1' at position $i = k + 1$. In the third step, the $\&m_1$ operation sets the leading bits to '0', resulting in $h_{out} = 0$, and the $h_{out} = h_{out} \mid m_0$ results in $h_{out} = m_0$. □

Now we show that **inc()** always returns an $h \in I$.

Lemma 4. $\forall h_{out} : h_{out} =$ ***inc***$(h_{in}, m_0, m_1) \rightarrow h_{out} \in I$

Proof. For any H-address $h_{out} =$ **inc**$(h_{in}, m_0, m_1) \rightarrow h_{out} \in I$ for all valid m_0, m_1 and $h_{in} \in I$ because the masking $h\&m_1 \mid m_0$ (post-masking) ensures that only the valid bits remain set, i.e. that $h_{out} \in I$. In other words, let $C(h) = h\&m_1 \mid m_0$ be the function used in the the post-masking step in **inc()** and in **isInI()**, see Lemma 1. C is idempotent since '&' and '|' are idempotent bitwise operations. Since **inc()** applies $C(h)$ as the last step, a subsequent check with **isInI()** results in $C(C(h))$. However, $C(C(h)) \equiv C(h)$, hence $\neg\exists h : C(C(h)) \neq C(h)$ which means that $\forall h_{out} : h_{out}$ =**inc()**$\rightarrow h_{out} \in I$. □

Finally we show that **inc**() always returns the direct successor, i.e. that there is no valid $h \in I$ that lies between any given input value and output value. This implies that $h_{out} > h_{in}$ for $h_{in} < m_1$.

Lemma 5. *$\neg\exists h_x : (h_{in} < h_x < h_{out}) \wedge (h_{out} = \textbf{inc}(h_{in}, m_0, m_1))$, or simply $h_{in} \prec h_{out}$ where '$\prec$' indicates direct predecessor relationship in I.*

Proof. First we consider the case that the least significant bit $b_{h,0} = 0$ right before the increment (after the pre-masking). $b_{h,0}$ can only be '0' if m_0 does not restrict on this bit, i.e. $m_{0,0} = 0$, otherwise $h_{in} \notin I$ would not be a valid input argument. Neither can m_1 restrict on the first bit, otherwise the initial pre-masking $h \mid \sim m_1$ would have set it to '1'. During the increment, we add '1', resulting in a bitstring that represents an integer that is trivially '1' larger than h_{in}. The post-masking has no effect on $b_{in,0}$ because we established that neither m_0 nor m_1 can have a restriction on that bit. They also cannot change any of the other bits of h_{out} because these have not changed and are identical to the bits of h_{in} which comply by definition with all restrictions imposed by m_0 and m_1. Since adding '1' to $b_{h,0}$ cannot affect any other bits, nor is it affected *by* other bits, it can be generalised to adding a '1' bit to any '0' bit, i.e. for any $b_{h,i}$.

Next, we show that adding '1' to any '1' bit $b_{h,i}$ also works. After the pre-masking step, a bit $b_{h,i}$ can be '1' either because it was '1' in h_{in} or because it was set to '1' during pre-masking because m_1 has a restriction on that dimension. Now, adding '1' to a '1' bit in $b_{h,i}$ will cause an overflow and result in $b_{h,i} = 0$ and '1' added to the next higher bit $b_{h,i+1}$. The overflow may cascade through several bits until it adds '1' to a '0' bit in $b_{h,j}$, with $j > i$, which we treated already in the first part of this proof. If neither m_0 nor m_1 impose any restrictions, the post-masking will not change h_{out} and h_{out} is trivially the +1-successor of h_{in}. Since the highest modified bit $b_{h,j}$ was a '0' bit, $h_{out} > h_{in}$ holds as established above. We also know from Lemma 4 that $h_{out} \in I$.

Finally, is h_{out} the direct successor of h_{in} or, in other words, could there be a value h_x with $h_{in} < h_x < h_{out}$? Any $h_x > h_{in}$ must have at least one '1' in a position i_x where h_{in} has a '0', otherwise it cannot be greater than h_{in}. This is only possible if '1' and '0' are actually possible values for that position, which means that neither m_0 nor m_1 impose a restriction on that position. If the position in question is $i_x = 0$, then h_{out} is the immediate successor as shown in the first part of this proof. If $i_x > 0$ then h_{out} is also the successor of h_{in} because, as shown above, the algorithm will overflow until it hits a bit that is '0'. All bits before that (i.e. all that cause an overflow) are either '1' in h_{in} or they can have only one state, which means h_x could not be different from h_{in} at that position without violating the boundary imposed by m_0 and m_1. This means there cannot be an h_x with $h_{in} < h_x < h_{out}$. □

7 Algorithm 3: succ(h,i)

The third algorithm works similar to **inc(h,I)**, but accepts as input arbitrary $h \in \mathbb{H}^k$ and not only $h \in I$. We start with a version of the algorithm that treats three cases separately before we present a more compact but less intuitive version without branching.

The 3-cases implementation first checks whether $h \in I$ which means that **inc()** can be used from incrementation (1st case). If $h \notin I$, it finds out which bits collide with m_0 (2nd case) or m_1 (3rd case) and, depending on where the most significant of the colliding bit comes from, uses two approaches to generate an output $h_{out} \in I$. The details are given in the proof to Lemma 6.

```
long succ(long h, long m0, long m1) {
  if (isInI(h, m0, m1)) {
    return inc(h, m0, m1); //1st case
  }

  long coll = ((h | m0) & m1) ^ h;
  long diffBit = maxBit(coll);
  long mask = diffBit > 0 ? diffBit-1 : 0;

  long confM0 = (~h) & m0;
  long confM1 = h & ~m1;

  if (confM0 > confM1) { // 2nd case
    h &= ~mask;
    h |= m0;
    return h;
  }

  //increment - 3rd case
  long out = h | ~m1;    //pre-masking
  out += coll & ~m1 ;    //increment
  out = (out & m1) | m0; //post-masking
  return out;
}
```

List. 4: Implementation of **succ()**

Note that we use a function **maxBit(x)** that returns a "*value with at most a single one-bit, in the position of the highest-order ("leftmost") one-bit in the specified value 'x'. Returns zero if the specified value has no one-bits in its two's complement binary representation, that is, if it is equal to zero*"[6]. Modern CPUs typically provide a constant time instruction for this operation.

Lemma 6. *h_{out} =**succ(**h_{in}, m_0, m_1**)** $\rightarrow h_{out} \succ h_{in}$ **with** $h_{out} \in I$. That means **succ()** always returns the next possible $h \in I$, i.e. $\neg\exists h_x : h_x \in I \wedge h_{in} < h_x < h_{out}$.*

Proof. We determine which bits in h_{in} conflict with m_0 or m_1, i.e. in which dimensions h_{in} lies outside I. Let i_{m0} and i_{m1} be the position (the rightmost and least significant bit is at $i = 0$) of the most significant bit $b_{h,i}$ that poses a conflict with m_0 and m_1, respectively, or $i = -1$

[6] See javadoc of `Long.highestOneBit()` in JDK 7 by Oracle Inc.

if no conflict exists. We now consider three scenarios. If $i_{m0} = i_{m1}$ then $i_{m0} = i_{m1} = -1$, because m_0 and m_1 can never conflict on the same bit, see Def. 6. If there is no conflict we can simply apply **inc()** and finish.

As second case we consider $i_{m0} > i_{m1}$. In this case, $b_{i,m0}$ is set to '0' even though it would need to be set to '1' to intersect with I. To create the next highest valid $h \in I$ we simply set all bits $b_i : i \le i_{m0}$ to their respective minimum, i.e. we set all bits b_i with $i \le i_{m0}$ to the values of the last i_{m0} bits of the known minimum m_0. The result is the smallest $h \in I$ with $h > h_{in}$, i.e. $h > h_{in}$, see also Corollary 2.

As third and last case we consider $i_{m0} < i_{m1}$ where the most significant conflicting bit at i_{m1} has a '1' instead of the required '0'. To find the next higher $h \in I$, we first apply the pre-masking from **inc()**, then we add a '1' at the conflicting position i_{m1}, i.e. we add $2^{i_{m1}}$, then we set all bits b_i with $i \le i_{m1}$ to their respective minimum defined by m_0 and finally apply the post-masking from **inc()**. By means of an overflow during the addition, the conflicting bit is set to '0' and the addition ensures that the resulting number is larger than h_{in}. By setting the trailing bits to their minimum, we ensure that we do not skip any $h \in I$, i.e. we ensure that $h > h_{in}$. To ensure that adding '1' works fine for the higher order bits b_i with $i > i_{m1}$, we apply the pre-masking and post-masking from algorithm **inc()** which ensures that the resulting value is indeed $h \in I$, see Lemma 4. □

While the algorithm in Lst. 4 works fine, it can be optimised by avoiding the three branching statements. Lst. 5 shows an optimised version with less instructions and without branching. First, we calculate two values, `confM0` and `confM1`, that have exactly one bit set to '1', either at the most significant position where a conflict occurs or as the least significant bit. We then calculate two masks, `maskM0` and `maskM1`, that are filled with '0' up to and including the '1' bit of `confM0` and `confM1`. After the '1' bit, they are filled with '1'. If no conflict occurs, the masks are all '0'. Then we start the incrementation with the known pre-masking step. This is especially necessary to bring the most significant conflicting bit into a consistent state. Then, in the new *masking* step, we remove all bits below the most significant conflicting bit. After that, we add `confM0 | ~maskM1`. If no conflict occurs, `confM0` is '1' and we add '1'. Otherwise we add '1' at the most significant conflicting position with m_1, unless m_0 has a more significant conflicting position, in which case the mask `~maskM0` ensures that nothing is added. Finally we do the post-masking and return the result. It is easy to see that the algorithm completes in constant time.

```
long succ(long h, long m0, long m1) {
  long confM0 = maxBit((~h) & m0 | 1);
  long confM1 = maxBit(h & ~m1 | 1);
  long maskM0 = confM0-1;
  long maskM1 = confM1-1;

  //increment
  long out = h | ~m1;       //pre-masking
  out = out & ~(maskM0 | maskM1); //mask
  out += confM1 & ~maskM0; //increment
```

```
    out = (out & m1) | m0;    //post-masking
    return out;
}
```

List. 5: Optimised no-branch **succ()**

8 Application Example

For the remainder of the paper, we use the PH-tree [ZZN14, Zä15] as a running example for the applicability of the established algorithms. The PH-tree is a index that forms a hierarchy of binary hypercubes. The root node is a binary hypercube and each corner is connected with a child hypercube which recursively have hypercubes connected to their corners. The position of the corner of the k-dimensional binary hypercube in the root node encodes the first bit of all values p_i of a point p, this effectively interleaves the bits of all p_i. The first child hypercube encodes the 2nd bit, and so forth. When storing values with a precision of 64 bit, the tree has a depth of 64 nodes. For simplicity we assume that all values p_i have the same precision and we ignore the fact that the trees may, for optimisation, create nodes only if at least two corners have children. Since each hypercube represents one bit of each dimension of a stored point, the tree can be at most 64 nodes deep for 64 bit values.

Geometrically, the PH-tree bipartitions the space recursively into hypercubes of decreasing size which are represented by nodes. Division is limited such that each node contains at least two points, either directly stored in the node or in subnodes. When traversing the tree one node after another, the data points p are returned in Z-order according to the interleaved value z = **interleave**$(p_0, ..., p_{k-1})$. An example of a two-dimensional tree ($k = 2$) with 3-bit values ($w = 3$) is shown in Fig. 2. The numbers are the interleaved z values, the squares represent the nodes. The outer square is the root node, it contains $2^k = 4$ child nodes, which each contain 4 leaf nodes with 4 points each.

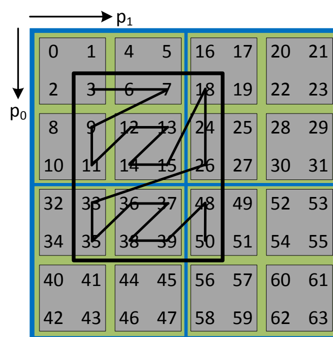

Fig. 2: z values and nodes of a tree with $k = 2$ and $w = 3$ with one root node (blue), its child nodes (green) and 16 leaf nodes (grey). The black rectangle is the querybox intersecting with binary hypercube nodes in z-ordered tree.

The PH-tree supports three storage representations for internally representing the sub-nodes and points stored in a node: Array Hypercube (AHC), List Hypercube (LHC) and Nested Tree Hypercube (NTHC).

In AHC representation, the entries of a node are stored in an array of size 2^k. This representation requires memory in the order of $\Theta(2^k)$, but can be efficient if the nodes contains many entries. The array addresses are effectively the Z-addresses which allows very fast lookup in $\Theta(1)$. The cost of full traversal is $\Theta(2^k)$, insert and delete execute in $\Theta(1)$.

In LHC representation, all entries are stored in a sorted list ordered by their Z-addresses. This approach is often more memory efficient but requires a binary search for random access with $O(\log_2 |N|)$. A full traversal costs $O(|N|)$.

With NTHC, entries are stored in a nested tree, see [Zä15]. NTHC is only used for very large nodes to speed up insertion and deletion. Like LHC, the cost of random access is about $O(\log |N|)$ and full traversal is $O(|N|)$. For the purpose of the calculations below, NTHC behaves approximately like LHC, except for higher base cost for all operations. Hence, we discuss below only LHC and imply that any conclusion also apply to NTHC.

One property of the PH-tree is that all entries in a node share the same prefix, i.e. the leading bits of the Z-values in a node are identical. This is called the *node-prefix*. Figure 3 shows on the left an example with the prefixes for the 4 sub-nodes of a root node with $k = 2$. The prefix is essentially the coordinate in the root's hypercube where the sub-node is attached. On the right the figure shows the prefixes of the leaf nodes, consisting of the prefix of their parent plus their own position in the parent. A prefix always contains a multiple of k bits.

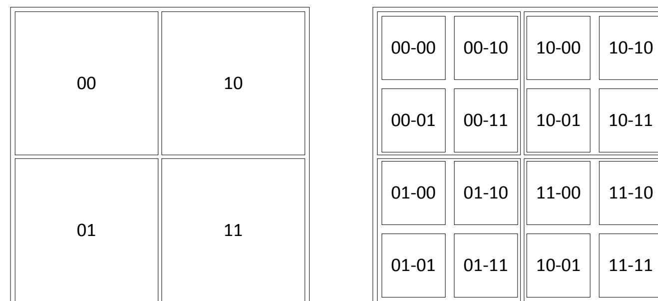

Fig. 3: Prefixes (positions in global space) of the direct children of the root node (left) and their respective children (right)

8.1 Window Queries

Window queries are specified by a 'lower left' and 'upper right' corner q_{min} and q_{max}. To locate matching points inside the query box, the PH-tree traverses the tree by locating the node that contains the interleaved Z-value of q_{min}. The tree then iterates through the stored Z-values of each node. Once it reaches the end of a node it returns to the parent node and traverses the next child node. The algorithm stops once the current Z-value is $\geq q_{max}$. Figure 2 shows an example with a query box with z_{min} = **interleave**$(001, 001) = 3$ to z_{max} = **interleave**$(101, 100) = 50$. The algorithms in this paper can be used to efficiently traverse the content of a node so that only those child nodes and points are traversed that potentially intersect with the query box.

For a window query with a query box defined by q_{min} and q_{max}, we intersect the query box iteratively with the binary hypercubes of each node. Starting with the root node, we need to identify those quadrants of the root's hypercube that can intersect with the query box. A simple approach is to iterate through all quadrants (represented by H-adresses), calculate the boundaries of each quadrant and check whether they overlap with the query box. Overlapping occurs if the lower boundary $p_{min} \leq q_{max,i}$ or the upper boundary $p_{max} \geq q_{min,i}$. To translate the boundaries of a hypercube relative into the global space, we need apply the prefix of the node. A naive implementation may look like this:

```
boolean isInI(h, qMin, qMax, prefix) {
  for (0 <= i < k) {
    //convert the i'th bit of h to int
    int pMin = hToMinInt(prefix, h, i);
    int pMax = hToMaxInt(prefix, h, i);
    if (pMax < qMin[i] || pMin > qMax[i]) {
      return false;
    }
  }
  return true;
}
```

List. 6: naive isInI($z, qMin, qMax, prefix$)

The `hToMinInt()` and `hToMaxInt()` in the above algorithm hide the fact that `p` needs to be reconstructed before it can be compared to `qMin` and `qMax`, i.e. we need to construct a minimum and maximum k-dimensional integer `p` consisting of the k-dimensional `prefix` and h. The difference between the two functions is that, if we are not in the leaf level, `hToMinInt()` fills all remaining bits with '0' while `hToMinInt()` fills them with '1'. On the leaf level, there are no remaining bits to be filled. Figure 4 shows an example where a node $[8, 11]$ intersects with a query so that only $\{9, 11\}$ lie in I. The middle part of the figure shows how the tree encodes the values, with $00 - 10$ as the node's prefix and the four h-values 00, 01, 10 and 11. The right part shows the same h-values and the resulting m_0 and m_1 which encode the intersection I (grey area) and, at the same time, represent the minimum and maximum h-values in I.

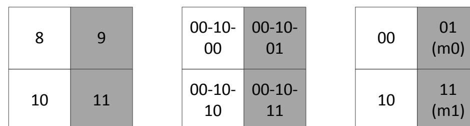

Fig. 4: Intersection I (grey area) of node with a query-box (left), according bit-encoding with prefix $00 - 10$ (middle) and resulting quadrants with m_0 and m_1 (right)

In the following sections we discuss how and when the three proposed algorithms can be used for window queries.

8.2 isInI()

The **isInI()** algorithm is useful when iterating through the H-addresses of an AHC hypercube where most elements can be expected to lie inside I, i.e. $|I|$ approaches 2^k. Since m_0 and m_1 can serve as minimum and maximum values for the iteration, it is sufficient if a suitable majority of the elements in $[m_0, m_1]$ can be expected to be in I. The total cost C_{node} of traversing a whole node in order to identify all potential matches is $C_{AHC,isInI} = 2^k \times (c_{isInI} + c_{array-lookup})$, where c_{isInI} denotes the cost of a single call to **isInI()**. The main advantage is that **isInI()** is very fast because it contains very few operations.

In the case of LHC, full traversal means traversing a list ($c_{list-next}$ per element) of $|N|$ elements. This results in a total cost $C_{LHC,isInI} = |N| \times (c_{isInI} + c_{list-next})$. Therefore, with LHC, **isInI()** is useful if $|I|$ approaches $|N|$ but inefficient for $|I| \ll |N|$.

8.3 inc()

In the following, $\{z_1, z_2, ...\}$ designates a sequence of z values returned that can be constructed from a node's prefix and the H-addresses generated by **inc()**. $[z_1, z_2]$ is a subnode that potentially contains all z-values between z_1, z_2, assuming that these z values are present in the tree.

Figure 2 shows a tree with a query box Q and the ordering of points in the query result. The aim of **inc()** is to return only h values from a node's intersection I so that a traversal for a window query only checks sub-nodes and points that potentially intersect with I. In the example, on the level of the root node (blue), **inc()** returns all children (green nodes), because they all intersect with Q. On the intermediate level (green nodes) it will return all children (grey leaf nodes) for the first node, the two leaves nodes $\{[16, 19], [24, 27]\}$ for nodes for the second and $\{[32, 35], [36, 39]\}$ for the third node and only the first leaf node $\{[48, 51]\}$ for the last green node. On the leaf level (grey nodes) it returns $\{3\}$, $\{6, 7\}$, $\{9, 11\}$ and $\{12, 13, 14, 15\}$ from the top left quarter of the tree, and so forth for the remaining tree.

In other words, **inc()** returns from a node only z values that potentially intersect with the query box Q. However, it does not guarantee that the nodes actually contain any point $p \in Q$. For example, the algorithm will check the top left grey leaf node $\{[0, 3]\}$ even if 3 is not actually stored in the tree. Please note that while the example in Fig. 2 looks quite simple, the size of the nodes grows with 2^k, which means that simply checking all z in a given node becomes prohibitively expensive for large k. Even checking only all actually existing h and the according z in a node, if such a list is available, can be expensive if the node intersects only with a small part with Q.

Using **inc()** in an AHC node costs $C_{AHC,inc} = |I| \times (c_{inc} + c_{array-lookup})$. When using LHC, **inc()** is more expensive with $C_{LHC,inc} = |I| \times (c_{inc} + c_{list-lookup})$ where $c_{list-lookup}$ is a $O(\log |N|)$ operation for performing a binary search on the sorted list of subnodes and point in the node. Obviously, **inc()** tends to work well if $|I|$ is small compared to the number of entries in the node $|N|$ or compared to the maximum size of the node $|\mathbb{H}^k| = 2^k$. The

disadvantage is that this approach always traverses $|I|$ elements even if $|I|$ is much bigger than number of entries in the node, i.e. if $|I| \gg |N|$.

8.4 succ()

We saw that **isInI()** and **inc()** both have strengths and weaknesses depending on the size of N, I and $\mathbb{H}^k$. We also saw in Corollary 4 that the gaps in the value space of I that occur during traversal of a node can be quite irregular and large. One idea for improvement is therefore to traverse areas with large gaps with **inc()** while using full traversal with **isInI()** on the (mostly) contiguous stretches of I. The distance to the next h can be calculated with $\Delta_h = \mathbf{inc}(h, m_0, m_1) - h$, at least for $h \in I$. If Δ_h is large, then we can decide to simply add Δ_h to the current h and continue traversal there. The problem is that the current h may not be from I, because we reached it via full traversal with **isInI()**. This is where **succ()** can be used, because it works with any $h \in \mathbb{H}^k$ as input. This is especially useful in the case of LHC, where random access is expensive due to the required binary search. The complexity of **succ()** is the same as **inc()**, however it has a higher base cost due to the additional operations.

8.5 Algorithm Selection

A detailed cost analysis is beyond the scope of this paper, however we provide a short guide on how to decide whether full traversal with **isInI()** or traversal of the intersection with **inc()** should be more efficient. In the case of AHC we established two relationships:

$$C_{AHC,isInI} = 2^k \times (c_{isInI} + c_{array-lookup}) \quad (1)$$

$$C_{AHC,inc} = |I| \times (c_{inc} + c_{array-lookup}) \quad (2)$$

The full traversal with **isInI()** is more efficient than using **inc()** if $C_{AHC,isInI} \le C_{AHC,inc}$. Lets assume that $c_{array-lookup}$ is negligible (ignoring memory access costs) and further estimate that $2 \times c_{isInI} = c_{inc}$ because it has roughly half as many instructions. As a result we see that full iteration should be used if:

$$\begin{aligned} 2^k \times (c_{isInI} + c_{array-lookup}) &\le |I| \times (c_{inc} + c_{array-lookup}) \\ \Rightarrow \quad 2^k \times c_{isInI} &\le |I| \times (2 \times c_{isInI}) \\ \Rightarrow \quad 2^{k-1} &\le |I| \end{aligned} \quad (3)$$

That means we should use **inc()** as soon as at least one dimension is restricted. The size $|N|$ can be calculated as discussed in Corollary 3.

In the case of LHC, **isInI()** should be used if $C_{LHC,isInI} \le C_{LHC,inc}$. Again, we estimate that $2 \times c_{isInI} = c_{inc}$. We also assume that $c_{list-next}$ is negligible (ignoring memory access

costs) and that $c_{list-lookup}$ costs $(\log |N|) \times (2 \times c_{isInI})$ for the binary search that requires twice as many operations for each step as **isInI()**. This means that full iteration should be used if:

$$
\begin{aligned}
& |N| \times (c_{isInI} + c_{list-next}) && \leq && |I| \times (c_{inc} + c_{list-lookup}) \\
\Rightarrow \quad & |N| \times c_{isInI} && \leq && |I| \times (2 \times c_{isInI} + (\log |N|) \times (2 \times c_{isInI})) \\
\Rightarrow \quad & |N| && \leq && |I| \times (2 + (\log |N|) \times 2) \\
\Rightarrow \quad & \tfrac{|N|}{2 \times (1 + \log |N|)} && \leq && |I|
\end{aligned} \tag{4}
$$

For **succ()**, the decision whether full traversal should be used or not can be made not only once per node but for each traversal step. However, this implies the additional cost of calculating the size of the gap and the expected number of possible non-matches. The size of the gap negatively affects the cost of full traversal, however if there are very few entries in the node, then full traversal may still be cheaper because the next element may be beyond the gap, thus allowing faster traversal than with **inc()**. The probability of finding any elements in a gap can be more accurately calculated by also taking minimum and maximum values m_0 and m_1 into account. This can be further refined by considering the number of elements that have already been traversed compared to the spatial fraction of the node that has been traversed, i.e. $n_{found}/|N|$ vs $h/2^k$. At the same time, the complexity of the analysis means that hybrid traversal with **succ()** should probably not be used unless it promises considerable advantages. This results in considerable complexity and, as indicated above, we consider it future work that is outside the scope of this paper.

8.6 Experimental Evaluation

Please note that the purpose of this section is exclusively to confirm the theoretic evaluation. Comparative performance test of the PH-tree with other indexes can be found in [ZZN14]. For the experiments we configured the PH-Tree in a special AHC-only mode. The AHC only mode ensures that the nodes in the tree do not change their representation during insert/update/delete (CUD) operations. This is useful in situations with very frequent CUD operations. Figure 5 shows the result for varying dimensionality of a cube-shaped dataset with 10^5 randomly distributed points between [0.0, 1.0] in every dimension, this is equivalent to the CUBE datasets described in [ZZN14]. The results show that, as expected, the use of **inc()** can increase performance considerably, especially for increasing dimensionality which allows for small $|I|$ compared to $|N|$ in the nodes' hyper cubes.

When allowing LHC mode, AHC nodes become much rarer and the performance gain is much less notable. As discussed in Sect. 8.3, **inc()** is expected to make much less of a difference in LHC nodes because random access in LHC mode requires a binary search with $O(\log n)$ base cost per access. AHC becomes increasingly rare with larger dimensionality because nodes have increasingly rarely enough children to justify AHC mode for optimal

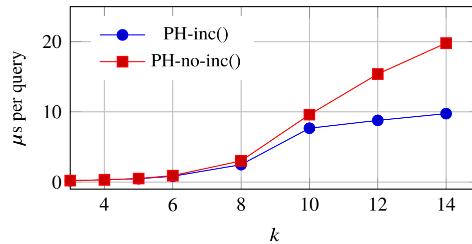

Fig. 5: Query execution times for 10^5 entries with varying dimensionality k of randomly distributed points between [0.0, 1.0] in every dimension. Each query returned on average 1000 entries.

memory usage. As explained [ZZN14], optimal memory consumption is the normally used as decisive factor for AHC vs LHC representation.

We acknowledge that the PH-Tree is not an ideal testbed, and that the non-AHC representation may be rarely used. However, the results clearly demonstrate that the algorithms work and can efficiently avoid worst case cost scenarios.

9 Conclusion

We discussed three algorithms, including two which we developed, that are useful for traversing partitions of k-dimensional binary hypercubes. By exploiting the 'parallel processing' of up to 64 bits in standard CPU registers, all three algorithms execute in constant time for $k < 64$.

For validation we implemented **isInI()** and **inc()** in a specially adapted version of the PH-tree[7]. The algorithms behaved correctly and with the expected performance complexity where our improvements showed increasing effect on the performance with increasing k. This resulted in 20% to 50% reduced query time for $k \geq 10$. Proper evaluation of **succ()** is considered future work.

10 Acknowledgements

This research was partially funded by the Hasler Foundation, Switzerland.

References

[Ba97] Bayer, R.: The universal B-tree for multidimensional indexing: General concepts. In: Intl. Conf. on Worldwide Computing and Its Applications. WWCA '97', pp. 198–209, 1997.

[BCA15] Bøgh, K.S.; Chester, S.; Assent, I.: Work-Efficient Parallel Skyline Computation for the GPU. Proc. of the VLDB Endowment, 8:962–973, 2015.

[7] The PH-tree source code is available from `http://www.phtree.org`

[BKS01] Borzsony, S.; Kossmann, D.; Stocker, K.: The Skyline Operator. In: Proc. 17th Intl. Conf. on Data Engineering. ICDE ’01. IEEE, pp. 421–430, 2001.

[Ch15] Chester, S.; Sidlauskas, D.; Assent, I.; Bøgh, K.S.: Scalable Parallelization of Skyline Computation for Multi-Core Processors. In: Proc. 31st IEEE Intl. Conf. on Data Engineering. ICDE ’15, 2015.

[Fe02] Fenk, R.: The BUB-tree. In: Proc. of 28th Intl. Conf. on Very Large Data Bases. VLDB ’02, 2002.

[KPS91] Kirschenhofer, P.; Prodinger, H.; Szpankowski, W.: Multidimensional Digital Searching and Some New Parameters in Tries. Technical Report CSD TR 91-052, Purdue University, Indiana, USA, 1991.

[LH14] Lee, J.; Hwang, S.-W.: Scalable Skyline Computation Using a Balanced Pivot Selection Technique. Information Systems, 39:1–21, 2014.

[Ma99] Markl, V.: Processing Relational Queries using a Multidimensional Access Technique. Dissertations in Database and Information Systems-Infix, 59, 1999.

[MAK03] Mokbel, M.F.; Aref, W.G.; Kamel, I.: Analysis of Multi-Dimensional Space-Filling Curves. GeoInformatica, 7(3):179–209, 2003.

[Ni13] Nishimura, Shoji; Das, Sudipto; Agrawal, Divyakant; El Abbadi, Amr: \ mathcal {MD}-HBase: design and implementation of an elastic data infrastructure for cloud-scale location services. Distributed and Parallel Databases, 31(2):289–319, 2013.

[OM84] Orenstein, J.A.; Merrett, T.H.: A Class of Data Structures for Associative Searching. In: Proc. of the 3rd SIGACT-SIGMOD Symp. on Principles of Database Systems. PODS ’84, pp. 181–190, 1984.

[Ra00] Ramsak, F.; Markl, V.; Fenk, R.; Zirkel, M.; Elhardt, K.; Bayer, R.: Integrating the UB-Tree into a Database System Kernel. In: Proc. of Intl. Conf. on Very Large Data Bases. VLDB ’00, pp. 263–272, 2000.

[Sa94] Sagan, H.: Space-filling Curves. Springer, 1994.

[Sa06] Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, 2006.

[Sk06] Skopal, T.; Krátkỳ, M.; Pokornỳ, J.; Snášel, V.: A New Range Query Algorithm for Universal B-trees. Information Systems, 31:489–511, 2006.

[TH81] Tropf, H.; Herzog, H.: Multidimensional Range Search in Dynamically Balanced Trees. Angewandte Informatik, 2:71–77, 1981.

[Zä15] Zäschke, T.: , The PH-Tree Revisited. `http://www.phtree.org`, 2015.

[ZYG11] Zhang, J.; You, S.; Gruenwald, L.: Parallel Quadtree Coding of Large-scale Raster Geospatial Data on GPGPUs. In: Proc. of 19th ACM SIGSPATIAL Intl. Conf. on Advances in Geographic Information Systems. GIS ’11. ACM, pp. 457–460, 2011.

[ZZN14] Zäschke, T.; Zimmerli, C.; Norrie, M.C.: The PH-Tree: A Space-Efficient Storage Structure and Multi-Dimensional Index. In: Proc. of Intl. Conf. on Management of Data. SIGMOD ’14, pp. 397–408, 2014.