The PH-Tree Revisited

Tilmann Zäschke

www.phtree.org
zaeschke@gmx.org
28 October 2015

ABSTRACT

We present several algorithms and improvements for the PH-Tree [13], a general purpose multi-dimensional index. The algorithms include a skyline query, nearest neighbour queries (kNN queries), range queries and a special `update()` operator for moving data. The improvements include a solution for the problem of large nodes, resulting in much better scalability with high dimensionality k when updating the tree. Another improvement uses generic data preprocessing to avoid problems with special cases of clustered datasets such as the CLUSTER 0.5 dataset in [13]. Finally, we discuss how the PH-Tree can be used to store hyper-rectangles instead of points and how to perform efficient queries on stored hyper-rectangles.

1. INTRODUCTION

The PH-Tree[1] is a multi-dimensional index that belongs to the family of quadtrees but provides much better space efficiency and scalability with higher dimensionality. The PH-Tree also scales very well with large datasets, in some cases larger sets with $N > 10^6$ actually perform better than smaller datasets. The PH-Tree also provides implicit Z-ordering by combining its quadtree features with the approach of interleaving bits as it done in critbit-trees (also known as binary prefix tries). However, the PH-Tree goes beyond a simple combination of quadtrees and critbit trees by using unique and efficient algorithms for multi-dimensional hypercube navigation which allow processing of up to 64 dimensions in parallel on CPUs with 64bit registers.

The contributions of this paper are as follows:

- Discussion of the trees properties.
- Solution to the insertion performance problem for increasing k.
- Solution to performance problems with special cases of clustered data with $k \geq 10$.
- kNN queries, range queries and moving objects.
- Storage and querying of rectangle shapes.
- Performance tuning recommendations.

[1]The complete source code is available at `http://www.phtree.org`

http://dx.doi.org/10.13140/RG.2.1.1594.0567.

The PH-Tree was originally published in [13]. First, in Section 2, we recapitulate the PH-Tree as presented in the original paper. Section 3 discusses first general properties of the PH-Tree and then specific explanations about non-intuitive behaviour. Section 4 discusses some structural improvements to the original version that solve for example the problem of slow updates with increasing k. In Section 5 we explain how data preprocessing can be used to avoid performance problems with special cases of clustered data. Then, in Section 6 we explain new algorithms, such as kNN queries, range queries and `update()`. After that we explain in Section 7 how the tree can efficiently be used to store rectangles instead of points. Then we give in Section 8 suggestions for getting the best performance out of the PH-Tree. Section 9 discusses open questions and possible improvements. Finally, we give some concluding remarks in Section 10.

1.1 Related Work

One resource for related work is the original PH-Tree publication [13]. Additional work on data preprocessing for the PH-Tree is available in [3]. Parallelization and cluster computing with the PH-Tree have been explored in [12]. In addition we would like to point out the very useful ELKI framework [7] for research in k-dimensional data mining.

1.2 Terminology

Most terminology is described in [13]. However, some terminology has been updated or added:

Range Queries & Window Queries.

The original paper uses the term *range query* for queries on a (hyper-)rectangular section of the data. This conflicts with other definitions, such as in [7], which use the same term for queries on a (hyper-)spherical section of the data, i.e. everything within a given (euclidean) range. In the text at hand we use *window query* for queries on (hyper-)rectangles and *range query* for queries on (hyper-)spheres.

HC, AHC, LHC, BHC, NI.

The original paper distinguished two possible data representations in a node with HC (HyperCube) and LHC (Linearized HyperCube). We now define three representations:

- AHC: Array HyperCube (formerly HC)
- LHC: Linearized HyperCube
- BHC: Binary Hypercube (also sometimes called NI), see Section 4.1

The acronym *HC* now simply refers to a node's hypercube, irrespective of it's representation with AHC, LHC or BHC.

The *HC-address* is the address of a sub-node or postfix in a node's HC.

Prefix & Infix.

The bits 'between' a node $n$and it's parent node are called *infix*, as illustrated in Figure 1. With *prefix* we refer to all bits 'above' a node's HC, i.e. the prefix of a node $n$is the concatenated bit-sequence of the infixes and HC-addresses of all parent nodes of $n$plus the infix of n.

The original paper used 'prefix' interchangeably for a node's infix and it's complete prefix.

2. THE ORIGINAL PH-TREE

The PH-Tree is essentially a quadtree [4] that uses hypercubes, prefix-sharing [8] and bit-stream storage [6]. The approach of the PH-tree avoids the need for several concepts that are often considered necessary for efficient multi-dimensional trees.

First, many proposed structures split the space in each node in only one dimension in a round-robin fashion. Contrary to that, we build on the quadtree and split the space in each node in all dimensions, which makes the access, such as queries, virtually independent of the order in which the dimensions are stored. This also tends to reduce the number of nodes in the tree, because each node can contain up to 2^k children instead of 2. At the same time, the maximum depth of the tree is independent of k and equal to the number of bits in the longest stored value, i.e. 8 when storing byte values.

Second, many structures aim to balance the tree in order to avoid degenerated trees which are inefficient in terms of performance and space requirements. The PH-tree however is unbalanced (see also the BV-tree [5]), which has the advantage that there is no need for rebalancing and the tree is stable with respect to insert or delete operations. This is useful for concurrency and when stored on disk, because it limits the number of pages that need to be rewritten. The PH-tree avoids problems with degeneration by inherently limiting imbalance, i.e. the maximum depth of the tree, during construction.

Third, the PH-tree does not aim for maximum node occupancy in order to reduce the number of required nodes. Instead of avoiding nodes, the PH-Tree efficiently reduces the size overhead of nodes.

Trees of the kD-tree family split the space in each node into two subspaces along one of the dimensions. Contrary to these, the PH-tree splits the space in each node in all dimensions and uses hypercube navigation within a node to locate sub-nodes and entries. One advantage of hypercubes is that, once the values of a k-dimensional point have been interleaved into a bit-stream, they require only a constant time operation, i.e. an array look-up, to navigate to the sub-node or stored entry. This is useful for point queries, insertion, deletion and locating the starting point of window queries. For comparison, binary trees have to traverse up to k nodes in order to progress one bit in every dimension.

While hypercubes provide superior performance for navigation inside a node, their space complexity of $O(2^k)$ becomes increasingly prohibitive for large k. Since we prioritise space efficiency over performance, the PH-tree uses hypercubes only when they do not negatively impact on space requirements.

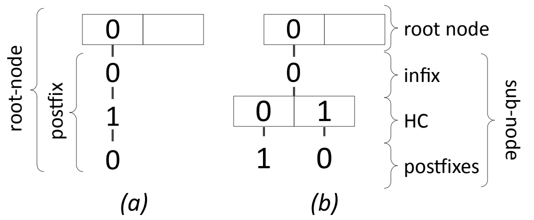


Figure 1: A sample 1D PH-tree with one 4-bit entry (a) and two 4-bit entries (b)

Compared to some other tree structures, another advantage is that the PH-tree can store non-metric spaces in the sense that all dimensions are treated independently and that it has no notion of distance. This makes it also suitable for discrete non-floating point data.

In order to explain the PH-tree, we start with a simple one-dimensional example in Sect. 2.1, which we extend to higher dimensions in Sect. 2.2. Section 2.3 discusses the storage of floating point values. Then we discuss space efficiency, query efficiency and update efficiency in Sect. 2.4, Sect. 2.5 and Sect. 2.6 respectively.

2.1 The 1D-PH-Tree

The PH-tree stores *entries*, which are sets of *values*. For example, a 2D point is stored as one entry with two values, each representing one dimension of the point. We first consider a 1D entry with just one value. The value is stored in its binary representation as a *bit-string*. For example, Fig. 1a shows the bit-string 0010 representing the number 2 when stored as a 4-bit value. In practice, the length of values is typically 8, 16, 32 or 64 bits for common data types. The first bit of any value in the tree is stored in the root node. Subsequent bits are stored in nodes further down the tree. For 4-bit values, the depth of the trees is thus limited to 4. Generally, the maximum node-depth $z_{n,max}$ of the tree is limited to the number of bits per value, the bit-width $w \geq z_{n,max}$. Besides the node-depth, we will also use the bit-depth z_b with $1 \leq z_b \leq w$ to refer to a bit-position. For example, $z_b = 1$ refers to the first bit of a value, $z_b = w$ to the last bit.

The split-box on top of Fig. 1a represents an array for fast look-up of references to entries and sub-nodes. Each array element is empty or holds either one entry or one sub-node. In the 1D-case, all entries starting with a 0 can be found below the left box, all starting with a 1 can be found below the right box. Entries that are attached to an array field without further sub-nodes, such as the 010, are called a *postfix*.

In Fig. 1b we see an example where a second value 0001 has been added to the tree. Since it starts with 0, it is also inserted below the left box in the root node. Since any node can hold only one reference in each position, the postfix is replaced by a sub-node. Because the two values differ only at $z_b = 3$, the common 0 at $z_b = 2$ is stored in the *infix* of the sub-node. The sub-node also has two postfixes, 1 and 0.

2.2 The kD-PH-Tree

For the 1D case, our approach resembles the binary PATRICIA trie [9]. For trees with dimensionality $k > 1$ however, we do not interleave the bits from the k values of each entry, but use a different approach. The bit-strings of the

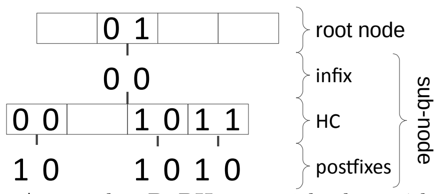


Figure 2: A sample 2D PH-tree with three 4-bit entries: (0001, 1000), (0011, 1000), (0011, 1010)

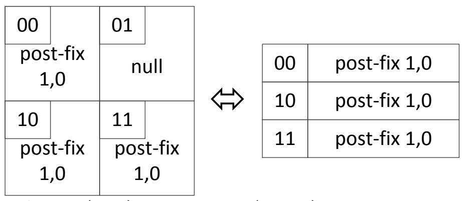


Figure 3: AHC (left) and LHC (right) representation of references in a node

values of one entry are stored in parallel as depicted in Fig. 2. On the top is the root node with a single reference at the second position. The position is calculated from the first bit (0 and 1) of each of the two values of the 2-dimensional entry. Using this approach, the array of references effectively becomes a hypercube (array hypercube – AHC) and the position numbers are hypercube addresses. Below the root node is the infix of the sub-node, consisting of a 0 for both values at $z_b = 2$. The AHC of the sub-node references three postfixes, which represent the three entries in the tree.

The size of the AHC is 2^k for a k-dimensional tree. For high dimensionality, it is likely to be only sparsely filled, for example for $k = 64$ the AHC has 2^{64} fields. In order to reduce memory requirements for high values of k, we store sparsely filled AHCs not directly but create a linear representation (LHC). Figure 3 shows on the left the AHC representation of the sub-node of Fig. 2 and on the right the equivalent LHC representation. The LHC representation consists of a table of value pairs that map <address in AHC> → <postfix/sub-node>. The table is sorted by the HC address (HyperCube address) to allow binary searches. In the example, the stored postfixes all contain two bits, 0 and 1, which represent the last bit of each value from each entry.

The PH-Tree switches automatically between LHC- and AHC-representation depending on which requires less space. For example, in Fig. 2 the root node on top has a reference to a sub-node at HC address 01. The bottom node has three references to postfixes at the HC addresses 00, 10 and 11. The top-node would be stored in LHC representation because it is sparsely filled, the bottom node would be stored in AHC representation, because it is almost completely filled and requires less space than the equivalent LHC representation. To determine whether AHC or LHC representation is used, we calculate and compare the size of both. We define n_s with $0 \leq n_s \leq 2^k$ as the number of sub-nodes, n_p with $0 \leq n_p \leq 2^k$ as the number of postfixes and l_p with $0 \leq l_p \leq w$ as the length (in bits) of the postfixes in a node. The AHC representation has fixed space requirements of $O(2^k)$ bits for sub-nodes and $O(l_p * 2^k)$ bits when storing postfixes. LHC representation requires space in the order of $O(n_s * k)$ for sub-nodes and $O(n_p * k * l_p)$ for postfixes. In a future implementation, a relaxed switching condition could prevent nodes from oscillating between AHC and LHC with each insert/delete operation.

Considering look-up speed, locating an element with known HC address is effectively an array look-up with $O(1)$. LHCs are sorted arrays that can be accessed via binary search, allowing random access with $O(\log n_p) \leq O(k)$.

2.3 Floating Point Values

All datasets used in the following experiments store 64 bit floating point values. However, the PH-Tree understands only bit-strings, which it sorts as if they were integer values. In order to store floating point values, these have to be converted such that sorting the bit-string representation results in the same order as ordering the floating point values directly. Since floating point numbers are stored using the IEEE 754 format, we applied the following conversion function (Java code):

```
long c(double value) {
    if (value == -0.0) {
      value = 0.0;
    }
    if (value < 0.0) {
      long lb = Double.doubleToRawLongBits(value);
      return (~lb) | (1L << 63);
    }
    return Double.doubleToRawLongBits(value);
}
```

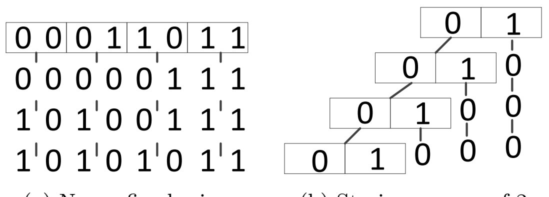
This conversion function has the property that for $i_1 = c(f_1)$ and $i_2 = c(f_2)$, $i_1 > i_2$ will be true if and only if $f_1 > f_2$, except for -0.0, which is eliminated. This sortability property allows the PH-Tree to perform search operations on stored entries independent of whether they represent floating point numbers or not. When reading entries from the PH-Tree, the inverse conversion can be applied to turn the result back into an IEEE floating point value.

2.4 Space Efficiency

The PH-Tree serialises most of the data of each node into a single bit-string [6]. This has two advantages. First, it reduces space needs by avoiding multiple arrays for different purposes, each of which is an object with its own memory management overhead. Second, values can be stored such that they use exactly the number of bits that they require. For example storing a boolean requires only a single bit.

With respect to space requirements, there are two effects that degrade space efficiency and cause worst case scenarios. One effect is a lack of prefix-sharing which prevents space-reduction. The other effect is a bad node-to-entry ratio. A large ratio is bad because it means having a large overhead of nodes stored in memory compared to stored values.

The first worst case scenario is that no prefix-sharing happens, for instance in a tree that has only a root node and no sub-nodes as shown in Fig. 4a. In this case all values are stored without prefix-sharing plus the overhead of the root node itself. In this case, the size of the tree is the size of the stored data $n * k * w$ plus the size overhead of the node. With the results from Sect. 2.2, the number of sub-nodes $n_s = 0$, $n_p = n$ and a postfix length $l_p = w - 1$ we get $O(w * 2^k)$ bits for AHC and $O(n * k * w)$ for LHC as the overhead for nodes. Since we assume a fully filled root node, we use the AHC representation with $n = 2^k$ which re-


(a) No prefix sharing (b) Storing powers of 2
Figure 4: The two space worst cases of the PH-tree

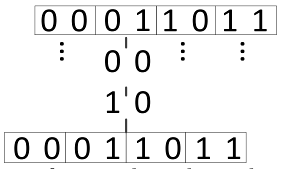


Figure 5: Best case for a sub-node with maximum prefix-sharing and maximum entries per node. Only the second sub-node is shown in full

sults in a total space requirement of $O(n*k*w+w*2^k) = O(n*k*w+w*n)+O(n*k*w)$ which is equivalent to the $O(n)$ of kD-trees.

The second worst case scenario is that the tree has a low entry to node ratio $r_{e/n} = n/n_{node}$ which results in significant memory consumption from the storage overhead of nodes. Each node has at least two sub-references such as sub-nodes or locally stored entries in the form of postfixes. Since every tree with $n > 1$ has more entries than nodes, we get $r_{e/n} > 1.0$ for $n > 1$. For example Fig. 4b shows a 1D PH-Tree with $r_{e/n} = n/n_{node} = 5/4 = 1.25$ (the bottom node contains two entries). However, for this extreme scenario to occur, two conditions apply. First, the depth of the tree is constrained to w, which is typically 16, 32 or 64 bit. This means that this scenario gets increasingly unlikely with growing n. Second, this scenario requires the data to have a special property that every entry deviates from a common shared prefix at a different bit position. This requirement is for example fulfilled in Fig. 4b which shows a tree with the entries $\{0000, 0001, 0010, 0100, 1000\} = \{0, 1, 2, 4, 8\}$, where each value deviates at a different position from the maximum shared prefix 000. If data does not have this power-of-two property or something equivalent, the worst case cannot occur. This scenario is more likely to occur in trees with large k, because it is sufficient if an entry deviates in one dimension from all other entries to cause the creation of a separate node. This occurs for example for the CLUSTER$_{0.5}$ dataset as discussed in Section 3.2.4. The combination of a lack of prefix sharing and a bad entry-to-node ratio is also unlikely, because the first requires a flat tree and the second requires few entries per node which can only be fulfilled in a minimum tree with one entry that consists of a single boolean value.

The best case occurs when all sub-nodes in a tree are fully filled and have the longest possible prefix, as the sub-node shown in Fig. 5. The root node on the top has $2^k = 4$ sub-nodes, where one is shown only, which have a infix of length 2 and contain 4 entries, one in each position of their 2D hypercubes. The length of the postfixes is 0. All values of the sub-node share the same prefix and differ from each other only in the last bits. Generally, each sub-node has a k-dimensional prefix of $w - 2$ bits and is filled with the maximum of $n = 2^k$ entries in AHC representation. In this case, the space requirement for the node and the tree as a whole is $O(k*(w-2)) + O(2^k)$ which results in $O(w*k+n)$. For comparison, a plain array requires $O(w*k*n)$.

In summary, with the number of entries being n, all worst cases result in space requirements of $O(w*k*n)$. Both worst cases and the best case can occur only with a very limited number of elements and very specific data characteristics. For higher numbers of entries and real-world data, they are unlikely to be approached.

The average case cannot be established as a simple function of n because it not only depends on the type of distribution, such as uniform distribution or Gaussian distribution, but also on the value range in each dimension. For example, if we assume a constant $w = 32$, data ranging from 0 to $1,000,000$ has a different storage requirement from data that ranges from 0 to $1,000$ because the latter allows better prefix-sharing. Furthermore, not only the width of the range affects storage requirements, but also the location. Storing data between 0.09999 and 0.10001 is more efficient than storing data between 0.49999 and 0.50001, even though the range is the same. The reason is that, in IEEE floating point representation, the latter causes a change in the exponent which is encoded in the higher order bits. This reduces prefix sharing and, for higher dimensions, reduces the likelihood of multiple entries sharing one node which leads to a worse entry-to-node ratio and thus requires more storage space. This example is discussed in detail in Section 3.2.4.

2.5 Query Efficiency

The PH-Tree supports two types of queries, point queries and window queries. Point queries take an entry as parameter and check whether an equivalent entry already exists or not. Window queries take as parameter a query-rectangle defined by a 'lower left' point and an 'upper right' point. The query returns an iterator over all points in the query-rectangle. Query efficiency depends as much on the type of query as on the characteristics of the stored data.

For point queries, the values of the entry first need to be interleaved into a single bit-string. The PH-tree uses a naive algorithm for interleaving that requires $O(w*k)$. During the actual search, point queries require the traversal of at most w nodes, which is the maximum depth of the tree. In AHC nodes, the sub-node or postfix can be found in $O(1)$, because the relevant k bits of the interleaved value directly represent the position in the AHC array of the node. In the case of LHC, a binary search is done over at most $\log_2(2^k) = k$ elements, resulting in a total worst case complexity of $O(w*k+w*k)) = O(w*k)$ for locating the sub-node or postfix. The query either extracts and compares the postfix or enters the sub-node, checks the sub-node's infix and continues the search in the sub-node. The sum of bits extracted for infixes and postfix is less than $w*k$ which we simplify to $O(w*k)$. When we add interleaving, query-internal search and infix/postfix extraction, we get $O(w*k+w*1+w*k)$ for AHC and $O(w*k+w*\log k+w*k)$ for LHC, resulting in $O(w*k)$ for both approaches.

It should be noted that the binary search for LHC requires extracting the keys from the bit-stream for each search step. The length of the key is k, which means that, for $k \leq 64$, the key can be extracted in constant time on a 64bit CPU, or, using modern SIMD instruction sets, the 64 bit limit can be extended to 512 bits. This means, if k is much larger than 64 (512) bits, the extraction approaches $O(k)$ resulting in a search effort of $O(k^2)$ per binary search and a total complexity of $O(w * k^2)$ for point queries.

Window queries start with a point query that locates the starting node, defined as the 'lower left' corner of the query window. Then, for each candidate node, all postfixes and sub-nodes that potentially intersect with the query need to be traversed. Even if the point query fails, the last visited node is the starting point for the query. Moreover, the HC address of the failed check can be used as starting address inside the node's AHC or LHC.

The worst case occurs when a query restricts only one or few dimensions out of k and if the values in these dimensions share long prefixes compared to other dimensions, because all their higher bits are the same. An extreme example would be a query on a dimension whose only values are 00000000 and 00000001 which is equivalent to storing boolean values. If the query constrains only this dimension, for example to 00000001, then, in order to see whether any entry matches the query, the algorithm has to fully read this value for all entries which means traversing all nodes in the tree. This results in a full scan with $O(n)$. This is the same as the worst case of kD-trees.

Another worst case occurs when many entries are postfixes of the same node. The most extreme case is the same as one of the space complexity worst cases, as depicted in Fig. 4a.

In the best case, location of the starting node is followed by a series of matches until the upper range of the query is reached. In this case the effort consists of locating the starting node and decoding the matching entries which results in $O(w * k) + O(w * k * n_{matches}) = O(w * k * n_{matches})$ or $O(w * k)$ per resulting entry.

Similar to space-complexity, the average query complexity cannot be established as a simple function of n, because it depends on different characteristics of the data. However, as the experiments in the next section show, query complexity tends to vary between $O(\log n)$ and $O(1)$ for low k.

Conceptually, the PH-Tree differs from binary trees such as kD-trees or PATRICIA-tries in that it splits in each node in all k dimensions. This has several consequences. First, the maximum depth of the tree in terms of nodes is limited to w instead of $w * k$ for binary trees. Second, for large k, the maximum number of nodes in a PH-Tree approximates $n_{node} = 2^{k+w}$ whereas it approaches 2^{k*w} for binary trees. These two effects reduce the number of nodes that can exist and at the same time the number of nodes that need visiting during a query.

Third, the hypercube representation allows very efficient window queries inside a node. If the node lies completely inside the query window, then the query iterator can simply iterate through all elements of the AHC or LHC. In the case of AHC, some slots may be empty, but the empty slots are limited to usually $\approx 30\%$ or less, because otherwise the node would switch to LHC representation. Finding the next post-fix or sub-node is thus a constant time operation. If the node is only partly inside the query window then the following algorithm assures that matching elements can be found efficiently. First we calculate two bit masks m_L and m_U that encode the lower and upper boundary of the intersection with the query rectangle. The masks each consist of k bits where any bit b_i with $0 \leq i < k$ of m_L is set to 0 iff the query range is less or equal to the lower left corner of the node in the dimension i. Inversely, any bit b_i in m_U is set to 1 iff the query range is equal to or larger than the top right corner of the node. The resulting masks m_L and m_U have several useful properties. First, the masks are effectively the HC addresses of the minimal and maximal possibly matching slots in the AHC/LHC. That means m_L and m_U can be used as start and end values for HC address iteration. This avoids iterating over many slots in the LHC/HC which can not possibly contain an entry that matches the query. Second, during iterations, they can be used on each HC address to simply check whether it fits the query rectangle or not. An HC address h fits if `(h|m`$_L$`) == h && (h&m`$_U$`) == h`.

In summary, the masks allow verification of slot validity in all dimensions in a single operation, assuming k is smaller than the register width of the CPU. This is considerably more efficient than binary trees which, in order to achieve the same, need to traverse a sub-tree that consists of up to 2^k nodes and that is up to $k-1$ nodes deep.

2.6 Update Efficiency

The structure of the PH-tree is determined solely by the characteristics of the stored data, not by the order of updates that are performed. Upon modification, at most two nodes of the tree need to be modified. For example, insertion requires location of the insertion node which is essentially a point query with $O(w * k)$, see Sect. 2.5. If there is already a postfix at the insertion position, the insertion requires creation of a sub-node ($O(1)$) which will contain the new entry. Encoding the new entry and possibly copying an existing postfix from the parent node takes $O(w * k)$. In the case of LHC, inserting values requires shifting parts of the LHC table which consists of up to $w * 2^k$ bits, resulting in $O(w * k + 1 + w * k + w * 2^k) = O(w * (2^k + k))$ for inserts if all nodes use LHC. Insertion in AHC nodes does not require shifting data, therefore the resulting worst case is $O(w * k)$.

Update complexity is usually measured on the number of entries. For the PH-tree, which currently does not allow duplicates, the maximum number of entries n_{max} is 2^{k*w}, encompassing all possible combinations of w bits in k dimensions. The worst case insertion complexity of $O(w * k)$ can therefore be seen as $O(\log n_{max})$. For comparison, kD-trees have an average update complexity of $O(\log n)$ and a worst case complexity of $O(n)$.

In summary, it can be seen that any modifications to the tree are largely independent of the number of entries in the tree. The number of entries in the affected nodes does play a role, but this could be improved by splitting the node into chunks, which is work in progress. The tree is not balanced, which has the advantage that no costly rebalancing can occur. At the same time, degeneration of the tree is inherently limited to w, the bit-width of the stored values. For example, when storing 32 bit integer values, the maximum depth of the tree is 32. Finally, each update affects at most two nodes, one being modified and possibly a second one being added or removed, which is useful for concurrent processing.

3. GENERAL PROPERTIES

The PH-Tree has a number of advantageous properties:

- **Static**. The tree structure is independent of the insertion order. There is no rebalancing. All modifications are local, not insertion, update or deletion will ever affect more than two nodes. This results in a number of advantages:
  - Good for **parallelisation**. Since updates are al-

ways local to at most two nodes, high concurrency with few conflicts is easily possible.

  – Good for **persistent storage**. Since there is no rebalancing, there is no risk of having to rewrite a lot of data to disk after an update operation (insert, update, delete)on the tree.

- **Limited depth**. The tree cannot degenerate. The depth of the tree is limited to the bit-precision of the key data, i.e. usually 64 for 64bit integer of floating point keys. The number of dimensions k does *not* affect the depth of the tree.

- **z-order**. Tree traversal occurs generally in z-order. That means all query results are z-ordered. This can often be useful, for example for skyline queries (non-domination queries), because

- The **nodes** of the tree are **strictly quadratic/cuboid** and **strictly non-overlapping**, except with parent nodes and child nodes. This simplifies searches and insertion.

- **bit-level encoding**. For many internal operations the tree uses a bit-transposed version of the key, see [13]. This allows processing up to 64 dimensions in parallel on a regular 64bit CPU, or more dimensions if the CPU registers support more bits.

- Preference for **clustered data**. As demonstrated in [13] prefers clustered data over evenly or randomly distributed data. For example query performance of the $CLUSTER_{0.4}$ dataset barely changes between $k = 2$ and $k = 15$ [13]. The weakness with certain datasets can be avoided using data preprocessing as shown in Section 5.

- **Space efficiency**. Due to prefix sharing and other techniques, the PH-Tree requires for increasing k less memory than most other known index structure. For $k \geq 7$ it requires even less space than a plain `long[]` or `double[]` that contains the same data.

- **Scaling with dimensionality**. The tree scales quite well with dimensionality. As described above, certain datasets show no degradation in query performance, even with $k = 15$. We did not conduct window query experiments with $k > 15$ yet.

- **Large nodes**. The nodes of the PH-Tree may contain up to 2^k children. While this can also cause problems (see 'disadvantages' below), it has the advantage that, compared to some other trees, typical queries have to traverse considerably fewer nodes. See also Section 3.1.2.

Known disadvantages include:

- **Complexity** of implementation. Due to the bit-level encoding and processing, large parts of the code contain less than intuitive bit-level operations. The total code base of the tree is also considerable (several 1000 lines of code).

- **CPU heavy**. Due to the bit-level encoding, the PH-Tree tends to be quite heavy on the CPU.

- **Large Nodes (solved)**. The PH-Tree's structure implies nodes with up to 2^k children. In the initial version of the tree, this translated to possibly very large nodes that slowed down insertion and deletion operations. However, unlike the original version, the current version of the tree does not store nodes in two single large arrays but splits them up if they get too big. This solves all problems mentioned in the original publication [13], see Section 4.1.

- **Garbage Collection (solved)**. The PH-Tree does not store keys as separate objects but encodes them in large arrays. As a result, queries have to create lots of array objects in order to return the extracted keys from the tree. This can put significant load on the Java garbage collector if the extracted keys are discarded afterwards. This problem can mostly be avoided by using the solution discussed in Section 8.

- **Bad performance with clustering around IEEE exponent change (solved)**. The old tree showed bad query performance if the data was strongly clustered floating point value, such as 0.5, with an exponent change in the IEEE representation. The recommended workaround was to shift data a bit, for example to 0.4. A more generic solution is to use an integer preprocessor as discussed in Section 5.

In our experience, the tree should be useful with large, ideally somewhat clustered, datasets with $\geq 10^6$ entries or more. During queries we found that the tree is comparatively fast at finding the first result, but somewhat slow at extracting results, i.e. with result size $n_{res} > 10$ the time complexity of queries $\approx O(n_{res})$, while the query initialisation cost tends to be negligible[2].

3.1 Structural Properties

3.1.1 Number of Nodes

Every node (except the root node) has at least two children, where children can be key/value pairs or sub-nodes. This means a tree with N entries, has $1 \leq n_{nodes} \leq N$ nodes.

Usually, if n_{nodes} approaches N, the tree tends to get slow for most operations such as updates or queries. In such cases data preprocessing may improve tree performance, see Section 5.

3.1.2 Maximum Number of Nodes

The total maximum number of nodes $n_{nodes,max}$ in a tree, depending on dimensionality k and bit-width w (usually $w = 64$) can be calculated as follows. We start with assuming a fully filled tree, i.e. a tree that contains all possible keys from the value space $S = \{0,1\}^{k*64}$ with $|S| = 2^{k*64}$.

$$n_{nodes,max} = \sum_{i=0}^{w-1} (2^{i*k}) \tag{1}$$

with

$$\sum_{i=0}^{n-1} (ar^i) = a\left(\frac{1-r^i}{1-r}\right) \tag{2}$$

[2]We compared this with an X-tree [1] which performed worse for small queries but better for queries with $n_{res} > 1000$

and

$$r = 2^k \quad (3)$$

leads to

$$n_{nodes,max} = \sum_{i=0}^{w-1}(2^{ki}) = \left(\frac{1-2^{kw}}{1-2^k}\right) = \left(\frac{2^{kw}-1}{2^k-1}\right) \quad (4)$$

For comparison, nodes in a Critbit tree are

$$n_{nodes,max} = n_{entries,max} - 1 = 2^{kw} - 1 \quad (5)$$

$$\Rightarrow \mathcal{O}(n_{entries,max}) = \mathcal{O}(2^{kw}) \quad (6)$$

In summary, $n_{max,PH} * (2^k - 1) = n_{max,crit-bit}$, i.e. a crit-bit tree may have up to almost 2^k as many nodes as a PH-Tree. This works to the advantage of the PH-Tree because, by implication, a typical query will have to traverse a lot less nodes to retrieve the results.

3.2 Behaviour

This section contains some more detailed explanations for non-intuitive performance behaviour, mostly as seen in the experiment section in [13].

3.2.1 Shrinking memory consumption

With the CUBE dataset, 3D, 4D and 5D datasets require *less* space than 2D datasets. The explanation is that with growing k, nodes can have more children (up to 2^k), thus reducing number of nodes.

For example, an entry requires $k*64+4$ bit = $k*8+4$ byte (key + reference to value)[3], while a node requires about 88 byte (39/40 + ba=16 + sub=16 + values=16 ⇒ 88).

If we now increase k from $k = 2$ to $k = 3$, the entry size increases by 8 byte (64bit) per entry. However, since we can save up to 50% of the nodes, i.e. up to 0.5 nodes per entry. In other word, going from $k = 2$ to $k = 3$ adds 8 bytes per entry but saves up to $0.5 * 88 = 44$ bytes per entry by reducing the number of nodes.

3.2.2 Increasing performance for 2D TIGER insertion

Figure 6a shows the average insertion times for entries for an increasing tree size when inserting the TIGER/Line dataset [4]. Interestingly, the diagrams show *increasing performance* when the tree grows from $1 * 10^6$ to $5 * 10^6$ and further.

This can be explained with a more *full* tree which is more densely populated with nodes. This results in nodes being 'closer' together, i.e. the infix length becomes '0'. This speeds up the tree because with infix length = 0 we can avoid infix checking, which is quite expensive, when navigating the tree.

The explanation in the original paper with increased HC use is probably wrong, the stats in Table 1 show no indication of any HC use.

There are more reasons, but they are likely to have little impact. The first reason is that more elements mean

[3]A reference in Java with <32GB RAM is 32bit

[4]The diagrams are taken from [13], the data is available as KML files [11]

N	bytes /entry	nodes	postfix length	postfix AHC	sub-node AHC
1000000	67	843750	36	0	0
5000000	67	4239657	35	0	0
10000000	68	8565765	35	0	0
15000000	68	12848525	35	0	0
18351888	68	15644278	35	0	0

Table 1: Tree quality for TIGER 2D / 64bit. From left to right: number of entries N, average bytes per entry, number of nodes, average postfix length of nodes, number of nodes with AHC representation for postfixes and for sub-nodes (postfixes and sub-nodes are stored separately).

that the post-fixes get shorter, that means they tend to require less array elements from which they are extracted or inserted. This mainly means that we can take shorter code-paths in the bit-level extraction/insertion methods. Also, it may slightly improve memory locality.

Another possible reason is that shorter post-fixes increase the likelihood that a post-fix matches a given query, i.e. the nodes are more likely to completely lie inside the query rectangle. This means that queries are less likely to encounter mismatching elements. In other words, the higher bits are still there, but they are encoded in the node's HC. This means they are stored only once (easier on the cache), and can be checked faster or even avoided with HC iteration.

Finally, the postfixes get shorter which means that insertion is less likely to cause an overflow in the byte-array. This avoids internally an array allocation (or array pool access) and a full array copy.

3.2.3 Increasing performance for CLUSTER queries

Figure 7c shows that the PH-Tree shows *increasing* query performance with increasing number of entries in the tree.

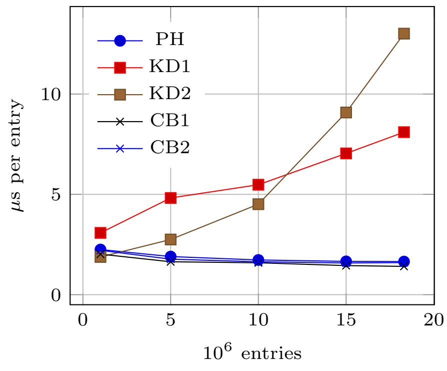
One reason is the same as for increasing insertion performance for the TIGER/Line dataset in Section 3.2.2, i.e. there are less and less infixes to process. However, for higher dimensional datasets it only works for the CLUSTER 0.5 queries. As we can see in Section 3.2.4, Tables 2 and 3, CLUSTER 0.5 is a somewhat degenerated tree, because it contains on average 0.65 nodes per entries, i.e. a lot of nodes. With an increasing number of entries, the tree recovers somewhat, i.e. we get on average more keys per node. This results in the same increased 'closeness' of nodes that avoids infix generation.

Another reason is specific to query execution. While executing a query, the PH-Tree traverses a lot of nodes and initialises an iterator for each node. This initialisation is time consuming. This is not a problem if a node contains a lot of children, but if there are very few children per node, then the overhead of initialising the iterator affects query performance. Similar to the infix processing, the cost of initialisation gradually disappears if the tree contains more data and on average more children per node.

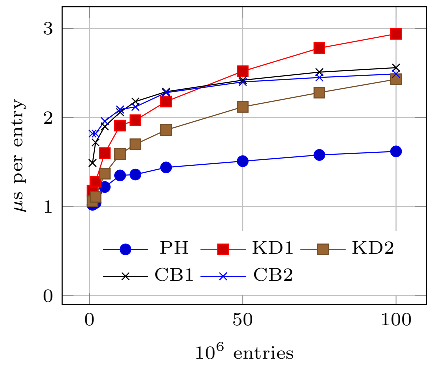
3.2.4 Performance difference on CLUSTER 0.4 & 0.5

Figure 8 shows that window queries on the PH-Tree perform much better on the CLUSTER dataset if the chain of clusters is shifted from 0.5 in space to 0.4 in space. As indicated in [13], this has to do with the IEEE encoding of floating point values.

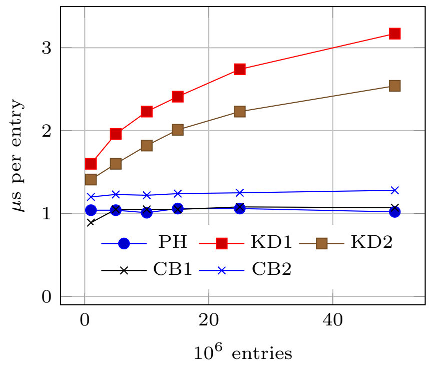
The tables 2 and 3 show the tree shape for trees with



(a) 2D TIGER/Line

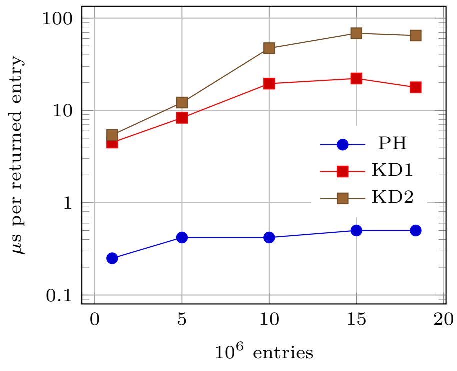


(b) 3D CUBE

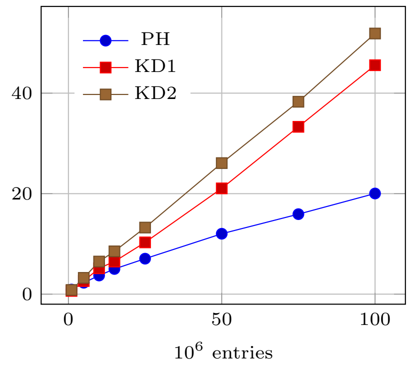


(c) 3D CLUSTER

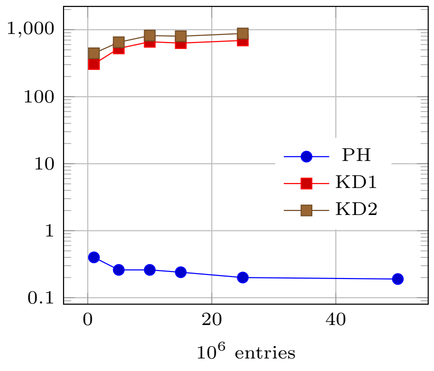
Figure 6: Insertion times per entry for the TIGER/Line, CUBE and CLUSTER datasets



(a) 2D TIGER/Line



(b) 3D CUBE



(c) 3D CLUSTER

Figure 7: Window query time for the TIGER/Line, CUBE and CLUSTER datasets

k	bytes /entry	nodes	postfix length	postfix AHC	sub-node AHC
2	52	6413880	32	54502	543004
3	44	4842617	34	128	92483
4	42	3899630	35	0	19813
5	43	3256625	35	0	9308
6	48	3364499	36	0	4669
8	53	2509507	36	0	0
10	61	2284548	37	0	0
15	74	553896	37	0	0

Table 2: Tree quality for CLUSTER 0.4

k	bytes /entry	nodes	postfix length	postfix AHC	sub-node AHC
2	52	6502433	32	52389	503281
3	46	5139209	34	110	71077
4	46	4406574	35	0	2956
5	49	4143194	35	0	4
6	56	4351824	36	0	4
8	81	6362178	36	0	4
10	112	8958219	38	0	4
15	159	9901860	44	0	2

Table 3: Tree quality for CLUSTER 0.5

10,000,000 entries, 64 bit floating point values and CLUSTER 0.4/0.5 data. We can see that with $k = 15$, $CL_{0.5}$ has 20 times more nodes than $CL_{0.4}$ (the difference is less pronounced with $k = 10$ with factor 4). At the same time, `postLen` becomes much larger which means up that nodes are, in average, closer to the root node. Even for smaller k, we can see that $CL_{0.5}$ has less AHCs, however, it has at least 4 (2). This may suggest a few densely populated nodes high up in the tree, followed by more nodes which are also high up (`postLen`, the length of the postfix, is the opposite of average depth). High up nodes result in long postfixes, which are slightly more expensive to extract.

However, the main effect comes probably from the distribution of node population. With CLUSTER 0.5, there are almost one node per key in the tree, i.e. rarely more than two children per node. In this case, iterating over the tree becomes very expensive due to the large fixed cost of initialising iteration over a given node, see Section 3.2.4. The ratio of node to key is even made worse by the fact that there are for $k = 15$ two nodes in AHC mode, which can only occur if they have 2^{k-1} children.

Section 5 discusses how this problem can efficiently be avoided with data preprocessing.

4. STRUCTURAL IMPROVEMENTS

4.1 Updates with $k > 3$

The original PH-Tree showed degrading insertion performance for increasing k. The reason is that all children in a node is stored in a single `byte[]`, and the number of children is limited only by 2^k, which can result in a large array. Inserting or removing children in a node may require large

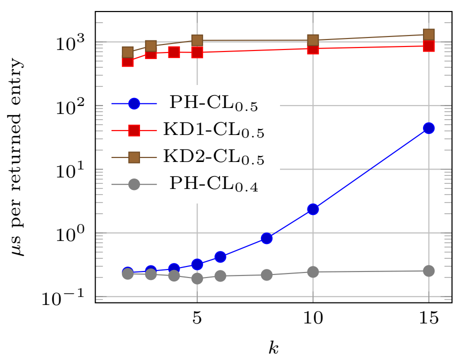


Figure 8: Query execution times for varying k and 10^7 64 bit entries and CLUSTER 0.4 and 0.5 datasets. KD query performance is identical for CLUSTER 0.4 and 0.5.

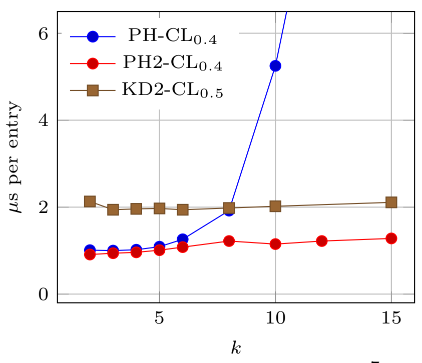


Figure 9: Insertion times for varying k and 10^7 64 bit entries for CLUSTER datasets using the original tree (PH) and the new version (PH2).

parts of the `byte[]` to be shifted, which becomes increasingly expensive.

The first optimisation was to use `long[]` instead of `byte[]`. This significantly reduces the loop count for copying arrays.

The second optimisation was to use a different representation that allows fast insertion and deletion by avoiding a single large `byte[]` for the data. The current implementation uses a binary tree, hence we call it BHC representation[5]. The BHC representation is only used in trees with $k \geq 7$ and only in nodes with over 500 sub-nodes or over 50 postfixes. These values were determined experimentally. Using BHC for queries is generally a bit slower than AHC or LHC representations. The difference for the sub-node and postfix threshold of 500 vs 50 stems from the different memory requirements. A sub-node generally requires k bits for the HC-address plus 4 bytes for the reference to the sub-node. A postfix requires much more data, k bits for the HC-address plus k∗postfix-length bits for the value. For $k \geq 7$ and an average postfix length of, for example, 32 bits, this results in $k + k * 32 = 7 + 7 * 32 = 231$ bits ≈ 29 byte.

While BHC may not be the most efficient representation in terms of memory and query performance, the negative impact is low because there are typically very few nodes that use this representations. An alternative representation is proposed in Section 9.1.

Using BHC almost completely resolves the negative side effects of larger k on insertion and deletion while having very little impact on query performance.

4.2 `prefixLen` and `infixLen` in Node Objects

Some of the most accessed values of every node are the `postLen` and `infixLen` which contain the lengths of the postfixes and the infix. These were originally stored in the large `byte[]`. In the new version they are stored in their own `int` fields in each nodes. This adds an overhead of 1 to 7 bytes per node but increases performance by 10-20%. Not that this also introduces a limitation that a node cannot store more than 2^{31} children. However, in practice this is not an issue because even if we have a dataset with more than 2^{31} entries and $k > 31$, they all would have to be formed in a way that they become postfixes of the same node. If that would be the case, the tree would switch to BHC representation, see Section 4.1.

4.3 Summary and Performance

[5]In the code it is called `NI` for node-index

The results of the structural changes are shown in the Figures 9 and 10. For the CLUSTER 0.4 dataset and the CUBE dataset, the new version of the PH-Tree avoids exponentially growing insertion times due to the BHC representations. The CLUSTER 0.5 tests, even when using the integer preprocessor as suggested in Section 5, show almost identical similar results, on average $\approx 5\%$ faster. They are not shown here because they mostly overlap with the PH2 results.

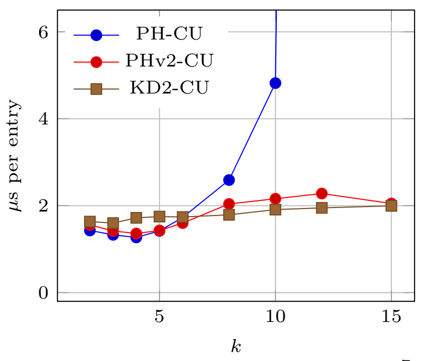


Figure 10: Insertion times for varying k and 10^7 64 bit entries for the CUBE dataset using the original tree (PH) and the new version (PH2).

5. DATA PREPROCESSING

The PH-Tree comes with two main interfaces[6]: `PhTree` for storing `long[]` keys and `PhTreeF` for storing `double[]` keys.

The PH-Tree internally only works with `long[]` keys, so `double[]` keys have to be preprocessed into `long[]` keys before they can be stored. By default this is done with the `EmptyPP` preprocessor which uses an approach described in [13]. This approach fully preserves the precision/accuracy of the `double[]` keys (including `NaN`, $-\infty$ and ∞; but -0 may be converted to $+0$). However, as also described in [13], the resulting performance is not optimal and may significantly decrease for certain datasets.

More details about data preprocessing can be found in [3].

[6]Note that these interfaces are for storing 'point' keys, there are also interfaces for storing rectangle keys.

5.1 Integer Preprocessor

The PH-Tree comes with one other predefined preprocessor, the `IntegerPP`. It simply multiplies and `double` value by a large constant C and converts the result to a `long` integer. The preprocessor takes as argument the constant C.

In our experience, using the `IntegerPP` increases performance by 10%-30% for normal datasets. Problematic datasets such as the $CL_{0.5}$ datasets perform similar to normal datasets, such as the $CL_{0.4}$, see [13].

The disadvantage of `IntegerPP` is the loss in precision. For datasets $[0.0, 1.0]$ we typically use $C = 10^8$ or $C = 10^{11}$ which allows for ≈8-11 fractional precision.

Depending on the dataset, the effect of C can vary considerably. For example, Table 4 shows execution times (load + query) for an experimental skyline [2] query algorithm. Best execution time is achieved for $C = 10^8$, worst for $C = 10^6$. The exact mechanism is not clear, but there two observations can be made. First, there is a string correlation between execution time and the number of nodes. Second, multiplying C by any power of two results in approximately the same performance and a similar number of nodes.

Also, while using $C = 10^8$ almost tripled the performance for a dataset with $k = 10$, performance for dataset with $k = 8$ or $k = 12$ was reduced almost by the same factor.

Note that this is an extreme case, for other datasets and algorithms we usually see much smaller differences in performance with varying C.

	runtime	insert	
C	[sec]	[sec]	nodes
10^6	98	2.44	261178
10^7	31.2	1.19	68328
10^8	15.9	0.91	68328
$1.5 * 10^8$	31.2	1.57	111723
$2 * 10^8$	15.0	0.92	68328
$2.56 * 10^8$	96.8	2.86	261178
$3 * 10^8$	31.2	1.54	111723
$4 * 10^8$	15.3	0.90	68328
10^9	96.7	2.98	261510
10^10	29.7	1.34	102420
10^11	17.8	1.00	68686
10^12	93.7	3.61	260591

Table 4: Performance of an experimental non-domination (skyline) query algorithm. The dataset consists of 10^6 10-dimensional independent points (similar to CUBE). The algorithm finds 106468 non-dominated points. BHC was disabled as an optimisation.

The best value for C will have to be determined experimentally for each dataset. As a general rule we recommend choosing C as small as possible, just big enough to achieve the desired precision. In our experience, the relevant characteristics of a dataset are the distribution of data, the minimum and maximum values in each dimension and the dimensionality k. The size of the dataset has usually no influence on the best value for$C$. Choosing the right C is one of the open questions, see Section 9.

The following code example creates a PH-Tree with an integer preprocessor with $C = 10^8$:

```
PhTreeF.create(dims,
    new IntegerPP(100L*1000L*1000L));
```

5.2 Data Exponent Shifting

As shown in [13], one way of improving performance for the CLUSTER 0.5 dataset is by shifting it to 0.4. For a more generally solution lets assume a dataset with minimum and maximum values $[x_{i,min}, x_{i,max}]$ for each dimension $0 \leq i < k$. The aim is to avoid a change in the exponent between $[x_{i,min}, x_{i,max}]$ in the IEEE representation, i.e. that both have the same exponent in binary representation. This can be achieved by finding an exponent e such that $2^e \leq x_{i,min} \leq x_{i,max} < 2^{e+1}$.

e can be calculated as the ceiling of the binary logarithm of the value range:

$$e = \lceil \log_2 (x_{i,max} - x_{i,min}) \rceil \quad (7)$$

We can then calculate the starting position of our target range with $x_{i,min,new} = 2^e$. If the values are all negative, we choose a negative range with $x_{i,max,new} = 2^e$. It follows that the required displacement d_i for the preprocessor is $d = 2^e - x_{i,min}$ except for negative $x_{i,max}$ where we use $d = -2^e - x_{i,max}$.

For example, using a range of $[0.4, 0.6]$ results in $d = -0.15$, i.e. an effective range of $[0.25, 0.45]$.

In addition, we want to avoid shifting further than necessary. Therefore, we change the algorithm so that d is always positive (for positive $x_{i,max}$) and that we simply shift into the next high region which fits the data range without causing exponent overflow.

We do this positive-only shift because in our experience it offers much better results than allowing $d < 0$. It is not clear why this happens, it seems that avoiding exponent change is only part of the problem. See also Section 9.

Due to the way that IEEE is encoded, the length of a region that has the same exponent is equal to the starting point, i.e. $|2^(e - 1) - 2^e| = 2^e$. This means, to achieve a positive-only shift we calculate $\Delta x = [x_{i,min}, x_{i,max}]$ as above. If $\Delta x \geq x_{i,min}$ (assuming a positive $x_{i,min}$) then we proceed as in Eq. 7, because we will always get a $d > 0$. Otherwise, if $\Delta x < x_{i,min}$, we use $x_{i,min}$ instead of Δx:

$$e = \lceil \log_2 (x_{i,max}) \rceil \quad (8)$$

PH-Tree provides an analysis tool that takes as input a sample dataset and creates a preprocessor that performs data exponent shifting. The data sample set should contain at least the estimated minimum and maximum values for each dimension. There is no problem if the real data later exceeds these minimum and maximum values, as long as outliers are rare. The aim of this preprocessing is only to avoid having data clusters crossing exponent 'borders'.

Example code:

```
double[][] data = {{0.4,0.4}, {0.6,0.6}};
ExponentPP pp =
    ExponentPPAnalyzer.analyze(data);
```

5.3 Results

Figure 11 shows the performance for the CLUSTER dataset for different dimensions k in μs per entry returned by a query. The best performance is achieved by shifting the data to 0.4 (PH2-$CL_{0.4}$). Using generic integer preprocessing instead of shifting shows slightly worse performance ((PH2I-$CL_{0.5}$). Applying both shifting and integer preprocessing (not shown) does not further increase performance.

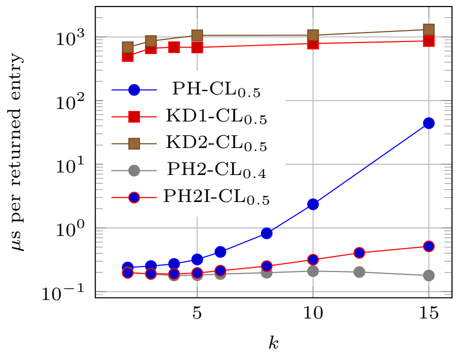


Figure 11: Query execution times for varying k and 10^7 64 bit entries and CLUSTER datasets. KD query performance is identical for CLUSTER 0.4 and 0.5. For the PH-Tree it shows performance of the old version (PH), the new version when data is shifted to 0.4 or when using the integer preprocessor (PH2I).

6. NEW FEATURES

The algorithms for insertion, deletion, point queries and window queries are described in [13]. Here we describe additional features that were not published in the original version of the PH-Tree.

6.1 Moving Objects & `update()`

In order to improve performance for moving objects we added an `update()` method that allows moving a point in the tree.

Lets assume a point v_1 that moves in space to a position v_2. Instead of calling `remove(`v_1`)` followed by `insert(`v_2`)`, the PH-Tree provides an `update(`v_1`, `v_2`)` function. This function tends to be twice as fast as the combination of `remove(`v_1`)` and `insert(`v_2`)` if v_1 and v_2 are in close proximity, i.e. in the same tree node. With growing distance, the performance decreases to that of the combination of `remove(`v_1`)` and `insert(`v_2`)`. Using `update()` is always advisable.

For convenience, `update()` also returns the value that is associated with the key. For example:

```
Object value = tree.update(v1, v2);
```

6.2 kNN Queries

In the PH-Tree, k-nearest neighbour queries (kNN-queries) return the $k = n_{min}$ nearest points to a given centre point c. We use n_{min} instead of k to distinguish it from the use of k as a measure for dimensionality.

Behaviour:

- If c is a key in the tree, it will be returned as part of the result.
- If the tree contains $\leq n_{min}$ entries, it will simply return all entries.
- The tree may return more than n_{min} entries if there are many entries with (almost) equal distance to c.
- However, the tree may choose not to return more than n_{min} entries, even if there are more entries with the same distance as already returned entries.

The algorithm works in two phases. In the first phase we determine an initial estimate for the distance. In the second phase search the tree for entries within that distance. The second phase may be repeated if the initial estimate for the distance was too small.

To estimate the initial distance d we navigate down in the tree to find an entry that is 'close' to the desired centre c of the kNN query. A 'close' entry is one that is in the same node where the c would be or, if there are no entries in the node, in one of its sub-nodes. Note that we do not return c itself in case it exists in the tree, this avoids $d = 0$. From whichever point we find, we calculate the distance to d to the centre point c and use it as initial search distance.

We then use a combination of window query ($Q = \{c_i \pm d\}^k$ and distance-query. Like a normal window query, we traverses only nodes and values that lie in the query rectangle. In addition, we also check every key whether it satisfies the distance constraint and every sub-node whether it can possibly contain any entries that satisfy the distance requirement. In euclidean space the distance constraint forms a sphere.

While iterating through the query result, we regularly sort the returned entries to see which distance would suffice to return n_{min} result, i.e. we take the maximum distance from the n_{min} closest known points. If the new distance is smaller, we adjust the query rectangle and the distance function before continuing the query. As a result, when the query returns no more entries, we are guaranteed to have all neighbours closer than the current known minimum distance.

The only thing that can go wrong is that we may get less than n_{min} neighbours in the second phase if the initial distance was too small. In that case we multiply the initial distance by 10 and run the algorithm again. Note that multiplying the distance by 10 means a 10^k-fold increase in the search volume.

NOTE: The query rectangle is calculated from the centre point and the estimated distance in the `PhDistance.toMBB()` method. The implementation of this method may not work with some types of non-euclidean spaces.

In the PH-Tree, the kNN-query is implemented with an algorithm that draws on the following strengths of the PH-Tree.

- The PH-Tree allows to quickly find a neighbouring point that lies with high likelihood in close proximity to c.
- The non-overlapping nodes (except with parent nodes) of the tree allow efficient window queries and efficient skipping of sub-nodes that do not satisfy the distance requirement.
- The PH-Tree query mechanism allows efficient on-the-fly update of query constraints, such as distance and rectangle, while executing the query.

We evaluated PH-Tree's kNN performance using the ELKI framework [7]. Using the PH-Tree behaved similar to their CoverTree implementation on 8, 10, 20 and 27 dimensional datasets (scale-d-10, scale-d-20, aloi-8d und aloi-27d). While the PH-Tree could not outperform the CoverTree (the fastest available kNN tree in ELKI), it took consistently at most *twice the time* to perform the query while being on average *twice as fast* to load the data into the tree. The PH-Tree may therefore be preferable if loading times are crucial or if the data is updated later on. We also had the impression that the PH-Tree improves performance with larger datasets. This is an impression we also has for other operations, including insertion and window queries [13].

For example:

```
//Find the 10 points closest to 'c'
PhKnnQuery<DATA> query =
    tree.nearestNeighbour(10, c);
while (query.hasNext()) {
  Object x = query.nextValue();
}
```

The PH-Tree kNN queries also allow using custom distance functions.

6.3 Range Queries

A range query returns all points within a given range r of a given centre point c. In euclidean space this translates to a spherical query with radius r.

For a range query, the PH-Tree executes internally a window query with an additional distance function that specifies the range of the query.

7. INDEXING HYPER-RECTANGLES

The PH-Tree can natively only store points. However, it is also possible to store extended shapes in the form of hyper-rectangles defined by a minimum and maximum point. This works by mapping the k dimensional minimum and maximum point into a single $2k$ dimensional point.

For example, to store a 2D rectangle that extends from (x_{min}, y_{min}) to (x_{max}, y_{max}), we store it as a 4D point with $(x_{min}, y_{min}, x_{max}, y_{max})$.

The performance for updates and queries of the PH-Tree for k dimensional rectangle shapes is essentially the the same as the performance for point data with $2k$ dimensions.

Intersection Queries.

Querying for any rectangles that intersect (partially or fully lie inside) the rectangle of a window query works as follows. Lets assume a query rectangle $(a_{min}, b_{min}), (a_{max}, b_{max})$ that we intersect with the xy-rectangle from above. The two rectangles intersect if:

$$\begin{aligned} &(x_{min} < a_{max}) \\ &\wedge(y_{min} < b_{max}) \\ &\wedge(x_{max} > a_{min}) \\ &\wedge(y_{max} > b_{min}) \end{aligned} \tag{9}$$

To perform a range query, we simply have to query for:

$$\begin{aligned} -\infty < x_{min} < a_{max} \\ -\infty < y_{min} < b_{max} \\ a_{min} < x_{max} < +\infty \\ b_{min} < y_{max} < +\infty \end{aligned} \tag{10}$$

That means we have to transform the 2D query rectangle $(a_{min}, b_{min}), (a_{max}, b_{max})$ into a 4D query rectangle with:

$$\begin{aligned} min &= (-\infty, -\infty, a_{min}, b_{min}) \\ max &= (a_{max}, b_{max}, +\infty, +\infty) \end{aligned} \tag{11}$$

The PH-Tree provides an API to perform these transformations: `PhTreeSolid` for `long[]` data and `PhTreeSolidF` for `double[]` data. The following example returns all rectangles that intersect or lie inside the $(2, 3), (7, 8)$ rectangle:

```
double[] min = new double[]{2,3};
double[] max = new double[]{7,8};
pht.queryIntersect(min, max);
```

Inclusion Queries.

The inclusion query, which returns only rectangles that lie completely inside the query rectangle, works similarly. The rules for the 2D case are:

$$\begin{aligned} min &= (a_{min}, b_{min}, a_{min}, b_{min}) \\ max &= (a_{max}, b_{max}, a_{max}, b_{max}) \end{aligned} \tag{12}$$

The following example returns all rectangles that fully lie inside the $(2, 3), (7, 8)$ rectangle:

```
double[] min = new double[]{2,3};
double[] max = new double[]{7,8};
pht.queryInclude(min, max);
```

8. TUNING

The PH-Tree implementation provides a number of opportunities for performance tuning. For example, a very simple performance improvement can be achieved for moving data by using the `update()` method, see Section 6.1. For many types of data, performance improvements can be achieved by using data preprocessing, see Section 5. The remainder of this section discusses how the load on the Java Garbage Collector can be reduced.

The PH-Tree provides several mechanisms to reduce object creation and thus reduce load on the garbage collector. The mechanisms include reusable iterators, reusable query result objects, array pooling and array over-allocation.

8.1 Reusable Iterators

Iterators in the PH-Tree, as they are returned by the various query interfaces, are complex objects. Internally, the iterators reuse most of the allocated objects and require very few non-reusable objects, which means they should have very low impact on GC while running.

However, iterators are not automatically reused, executing many queries can create significant GC load if new iterators are used for each query.

To avoid this problem, all iterators in the PH-Tree have a `reset(...)` method that allows restarting the iterator, if desired with different parameters. Iterators can be reset at any time, independent of whether the previous iteration reached the end or not.

For example:

- `PhExtent.reset()`
- `PhExtentF.reset()`
- `PhQuery.reset(long[] min, long[] max)`
- `PhQueryF.reset(double[] min, double[] max)`
- `PhKnnQuery.reset(int nMin, PhDistance dist, long[] center)`

- `PhKnnQueryF.reset(int nMin, PhDistance dist, double[] center`
- `PhRangeQuery.reset(double range, long[] center)`
- `PhRangeQueryF.reset(double range, double[] center)`

8.2 Reusable Query Results

All iterator in PH-Tree implement the Java `Iterator` interface but also provide additional methods. The normal `next()` method returns the value of the current entry. This has no impact on GC since no objects are created.

The `PhIterator` provides three additional methods for getting results from an iterator: `nextKey()`, `nextEntry()` and `nextEntryReuse()`.

`nextKey()` and `nextEntry()` are simply convenience methods that return the key or the Entry (key and value). While convenient, they have the disadvantage that they create new objects for each result: The `nextKey()` method creates a new `long[]` or `double[]` object to hold the key while the `nextEntry()` methods creates in addition to a key object and an entry object.

`nextEntryReuse()` avoids the additional objects by reusing key- and entry objects. For example, the iterator may maintain internally only two `long[]` instances, which are returned in alternating order to return the key. That means, after the second iteration, the first key object is 'reused', i.e. overwritten with a new key, and returned again. This avoids creation of any objects that need to be garbage collected later, but obviously comes with some consequences:

- Any returned key or entry is only valid until the next call to any of the `nextXYZ()` methods.
- Writing to any returned object may invalidate subsequent results.
- While currently not the case, in future versions, updating any of the returned objects may invalidate the tree itself.

It depends very much on the use case which of the `nextXYZ()` methods is most useful. The `nextEntryReuse()` method is usually useful if (most of) the entries are only required temporarily, for example if they undergo further filtering or when counting results.

Note that is possible to use the `nextXYZ()` methods interchangeably, for example one could call `next()`, `nextKey()`, `nextEntry()` and `nextEntryReuse()` in any order, with the expected correct result for each method.

8.3 Storing Keys as Values

Another approach to avoiding object creation during query traversal is to store the key as (part of) the value. For example:

```
//insert data
long[] key = ...;
tree.put(key, key);

//read data
PhKnnQuery<long[]> query =
    tree.nearestNeighbour(10, c);
while (query.hasNext()) {
  long[] key = query.nextValue();
  ...
}
```

Of course it is also possible to store as value a composite object that consists of the key and additional data.

8.4 Array Pooling

Another new mechanism to reduce GC load is array pooling. Array pooling works by pooling `long[]` and `Object[]` objects when the tree is updated. The pool consists of s_{array} several sub-pools, one for each array size between 1 and s_{array}. Currently we use a 'rectangular' pool, which means that all sub-pools have the same size s_{pool}. In practice this works fine with pools < 10MB for $s_{array} = 1000$ and $s_{pool} = 100$. Unfortunately, the theoretical memory requirement in this case would be $M \approx s_{array}^2/2 * s_{pool} * 8bytes = 400$MB (excluding the overhead of $1000 * 100 * 16 = 1.6$MB for storing 100,000 array in memory).

A possible improvement would be the implementation of a triangular pool where the size of the sub-pool ranges from s_{pool} for array size 1 to 1 at array size s_{array}. The theoretical maximum memory would be about 1/3 of the square pool, i.e. 133MB.

Array pooling reduces array allocations typically by $50\%-90\%$. Usually there should be no need to reconfigure the pooling mechanism. However, if the need should arise, it can be reconfigured or disabled in the `PhTreeHelper` class.

8.5 Array Over-Allocation

Another approach to reduce the array allocation is to allocate arrays larger than initially necessary.

Using `PhTreeHelper.setAllocBatchSize(x)`, the user can define how arrays are allocated. The parameter `x` signifies multiples of 8 byte, which is the memory alignment in a VM with less than 32GB RAM[7].

Unfortunately, this turned out to be of little help. The problem is that an incremental change in a node, such as adding or removing a postfix, adds or removes typically $k * 8$bytes. So the x would need to be set to at least k for 64bit values (ignoring the fact that due to prefix sharing we may store only 40 bits or so). This is quite expensive on the memory without significant improvements in performance.

9. OPEN QUESTIONS AND OUTLOOK

9.1 Outlook

- Cluster detection. There may be unique advantages of the PH-Tree with respect to cluster detection. For example, the depth of a node (length of the post-fix) indicates proximity of children. Can this be used to detect clusters? Unfortunately, the reverse is not true, proximity does not always result in short postfixes, for example for $63_{10} = 00111111_2$ and $64_{10} = 01000000_2$.
- Currently the tree stores the infixes and postfixes not in transposed (=interleaved) format. If transposition/interleaving would be cheaper (using low level AVX instructions with $O(\log k)$ instead of $O(k)$, then a fully transposed storage may have many nice side effects, for example when checking infixes or postfixes during queries.

[7]This can be configured when starting the JVM with the argument `-XX:ObjectAlignmentInBytes=16`

This could also make sense with the experimental HighK iterator that carries inclusion masks when traversing the tree during a query.

- Persistence / large nodes. When nodes get too big, the tree currently switched to a critbit representation. This works reasonably well, but using a PH-Tree may actually work better. For example, the entries in a 8-dim node could be stored in a 4-dim PH-Tree with depth 2. Such a representation may also be useful when storing a PH-Tree on disk, because the above approach means that no node really contains more than 2^4 entries, or whatever they k-limit of these inner node may be.

9.2 Open Questions

- Why does HC-iteration makes so rarely sense? According to some experiments, HC iteration make most sense with trees around $k = 5$. Why? Similar to this, the non-domination iterator works best for $6 \leq k \leq 8$. Is there a connection? Possibly a bug?
- 64bit alignment does not help. The PH-Tree currently uses a lot of bit-level operation, for example to safe space through prefix sharing. We tried storing the whole keys (all 64bit), which allows 64bit-aligned access and much simpler write/read operation at the cost of 30% increased memory consumption. Unfortunately, this did not improve performance at all but even gave a small decrease $\approx 5\%$. It is not clear why, one explanation is that the PH-Tree performance is actually not limited CPU but by the memory-bus, which means that simplifying the algorithms by at least 50% at the cost of 30% more data is *not* a good deal. Or was our testing faulty?
- Behaviour with $\text{CLUSTER}_{0.5}$ dataset. Section 5 discusses how this can be avoided using data preprocessing. While the exponent change in the middle of the cluster is part of the problem, simply avoiding the exponent shift is not always a solution, see discussion of the `ExponentPP` preprocessor. It is not clear why it is also important to avoid shifting towards 0 during preprocessing.
- Integer preprocessor. Choosing the best multiplier constant requires a lot of experience. It would be interesting to find out why multiplying the constant by 2 or 4 does not change the performance, but multiplying by 2.56 can reduce performance by a factor of 6.5 and increase the number of tree node by a factor of 3, see Section 5.

10. CONCLUSION

The PH-Tree is a general purpose multi-dimensional index. It behaves well with clustered data and large datasets. It also has useful properties such as being very memory efficient and being a static tree without rebalancing.

In this paper we presented solutions to the problems mentioned in the original paper [13] and we proposed algorithms for kNN queries, range queries and `update()` features. We also gave a more in-depth explanation of the tree's behaviour an suggestions for tuning.

Recent updates of the PH-Tree can be found on the website [10].

11. REFERENCES

[1] Stefan Berchtold, Daniel A Keim, and Hans-Peter Kriegel. The X-tree: An Index Structure for High-Dimensional Data. *Readings in multimedia computing and networking*, page 451, 2001.

[2] S Borzsony, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 421–430. IEEE, 2001.

[3] Adrien Favre-Bully. Data Preprocessing and Other Improvements for the Multi-Dimensional PH-Index. Master's thesis, ETH Zurich, Zurich, Switzerland, 2014.

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

[5] Michael Freeston. A General Solution of the n-Dimensional B-Tree Problem. *SIGMOD Rec.*, 24(2):80–91, 1995.

[6] Ulrich Germann, Eric Joanis, and Samuel Larkin. Tightly Packed Tries: How to Fit Large Models into Memory, and Make them Load Fast, Too. In *Proc. of the NAACL HLT Workshop*, pages 31–39, June 2009.

[7] LMU - Ludwig Maximilians Universität München. ELKI 0.6.5. http://elki.dbs.ifi.lmu.de, 2015.

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

[9] Bradford G. Nickerson and Qingxiu Shi. On k-d Range Search with Patricia Tries. *SIAM Journal on Computing*, 37(5):1373–1386, 2008.

[10] Tilmann Zäschke. PH-Tree. `http://www.phtree.org/`, 2015.

[11] U.S. Census Bureau. TIGER/Line 2010 Files. `ftp://ftp2.census.gov/geo/tiger/KML/2010_Proto/`, 2010.

[12] Bogdan Vancea. Cluster Computing and Parallelization for the Multi-Dimensional PH-Index. Master's thesis, ETH Zurich, Zurich, Switzerland, 2015.

[13] T. Zäschke, C. Zimmerli, and M.C. Norrie. The PH-Tree: A Space-Efficient Storage Structure and Multi-Dimensional Index. In *Proc. of Intl. Conf. on Management of Data (SIGMOD '14)*, pages 397–408, 2014.