The PH-Tree – A Space-Efficient Storage Structure and Multi-Dimensional Index

Revised Version – 28 June 2014

Tilmann Zäschke
zaeschke@inf.ethz.ch

Christoph Zimmerli
zimmerli@inf.ethz.ch

Moira C. Norrie
norrie@inf.ethz.ch

Institute for Information Systems, Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

We propose the PATRICIA-hypercube-tree, or PH-tree, a multi-dimensional data storage and indexing structure. It is based on binary PATRICIA-tries combined with hypercubes for efficient data access. Space efficiency is achieved by combining prefix sharing with a space optimised implementation. This leads to storage space requirements that are comparable or below storage of the same data in non-index structures such as arrays of objects. The storage structure also serves as a multi-dimensional index on all dimensions of the stored data. This enables efficient access to stored data via point and range queries. We explain the concept of the PH-tree and demonstrate the performance of a sample implementation on various datasets and compare it to other spatial indices such as the kD-tree. The experiments show that for larger datasets beyond 10^7 entries, the PH-tree increasingly and consistently outperforms other structures in terms of space efficiency, query performance and update performance. For some highly skewed datasets, it even shows super-constant performance, becoming faster for larger datasets.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Trees; E.2 [**Data**]: Data Storage Representations; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*

General Terms

Algorithms, Experimentation, Performance

Keywords

Multi-dimensional index, space efficiency, spatial index, patricia-trie, hypercube, quadtree, skewed data

1. INTRODUCTION

Multi-dimensional numerical data is used in many applications, for example in spatial contexts such as geo-information systems, astronomy or ray-tracing. The data often has two or three spatial dimensions plus any number of additional dimensions such as a possible node identifier in geo-information systems.

In this paper we present the PATRICIA-hypercube-tree (PH-tree) which combines binary PATRICIA-tries [16, 17] with a multi-dimensional approach similar to quadtrees while being navigable through hypercubes. The prefix sharing used in PATRICIA-tries leads to reduced space requirements, while the use of hypercubes allows navigation to sub-nodes and entries much more efficiently than binary trees because it is largely independent of the number of dimensions.

The internal structure of the PH-tree is determined only by the data, not by order of insertion or deletion of entries. Rebalancing is conceptually not possible but imbalances are inherently limited. For updates to the tree, having no rebalancing allows the changes required on the tree to be confined to at most two nodes. One node is updated and possibly a second one is either created or deleted, while at most one entry is moved between the two nodes. Our approach is mainly aimed at in-memory storage with integrated indexing and query requirements. However, in the case of datasets with many dimensions, the PH-tree is also suitable for persistent storage because the nodes of the PH-tree get large enough to be split efficiently to fit into disk-pages. With many dimensions, the efficiency for updates, queries and storage space makes the PH-tree suitable to be used not only as an extension for indexing data, but also as a primary storage layout for databases.

To evaluate our approach, we compared its performance with freely available implementations of other structures such as kD-trees and critical-bit-trees. We measured insertion, deletion, range queries and point queries on both synthetic data and real-world data with the the result that the PH-tree outperforms other solution consistently in terms of space and performance for datasets with 10^7 entries or more.

2. RELATED WORK

Starting from the simple binary search tree, the list of algorithms and data structures devised for various use-cases is long and differences are often only found in the details. In the domain of multi-dimensional data, we can differentiate between point access methods (PAM) and space access methods (SAM) as proposed in the survey by [8]. A more recent survey with fine-grained classification is given in [15].

The main representatives of the PAM category are kD-trees and quadtrees. kD-trees [2] are k-dimensional binary trees where the inner nodes of the tree describe axis-aligned hyper-planes that divide the space into two half-spaces corresponding to its child-nodes. The axis along which the split is performed is the same for all nodes on a certain tree level. Usually, dimensions are then switched between levels in a round-robin fashion. Since each node represents an inserted point, the tree's structure is dependent on the order of insertion and deletion operations and prone to degeneration. Rebalancing the tree is tricky since the usual rotation operations of other binary trees are not applicable to the relation between split-dimension and tree level. Extensions to the basic kD-tree have been proposed for externalisation through combination with B-tree ideas (kD-B-tree [20]), avoiding empty nodes by avoiding splitting nodes that are not full on insertion (hB-tree [14]) or by the logarithmic method of a forest of kD-trees (Bkd-tree [19]). Other versions of the kD-tree extend its capabilities into the SAM area by handling the problem of regions overlapping split-planes in different ways, for example the SKD-tree [12] or the Extended kD-tree [4].

In contrast to the popular kD-tree, the quadtree [6] splits the space in all dimensions at each node. This means that, in the 2-dimensional case, each inner node has 4 children, one for each quadrant. In our PH-tree, we take the same approach of splitting the space in all dimensions at each node in the tree. One recent extension to the quadtree is the IQ-tree [5], an indexing mechanism for filtering geo-textual data by combining a 2D quadtree with an inverted file for keywords, which allows them to efficiently perform localised searches for tagged geo data. Quadtrees are rarely used outside 2D or 3D problems because they tend to require a lot of memory due to their propensity for requiring many and large nodes. In the PH-tree, we counter this by using a combination of other approaches.

One of these other approaches is the PATRICIA-trie [16], where strings are stored in a prefix-sharing method which is usually much more space efficient than storing each key individually. PATRICIA-tries also have the advantage that they do not depend on insertion order. In general, any kind of data can be stored in such a tree by taking the bit representation of the data and using the tree to manage bit-strings. Such bit-tries are also known as crit(ical)-bit-trees.

In order to store keys with multiple dimensions, the values of the different dimensions can be interleaved into a single bit-string in a round-robin fashion as discussed in [17, 13].

While [13] provides a thorough complexity analysis of queries in binary tries, [17] extend this idea by allowing keys that consist of variable length text as well as numeric data. Furthermore, they implement an approximate range search algorithm that reduces the number of visited nodes by up to 50% by slightly reducing query accuracy close to the edges of the query hyper-rectangle. While their approach is presented for a binary tree, it should also work for the PH-tree and we see it as a desirable future extension.

The PH-tree is also based on the idea of prefix-sharing. However, unlike other PATRICIA-tries, the PH-tree is not a binary tree but a quadtree. This reduces the number of nodes in the tree by allowing up to 2^k entries per node. Within these nodes, a hypercube-based addressing scheme for the sub-nodes provides efficient access for higher dimensional data. For example, locating a key in a 16-dimensional boolean dataset requires visiting up to 16 nodes in a binary tree. In a hypercube quadtree, because the interleaved bit-string directly represents the array position, locating a key requires accessing only one node followed by a simple array look-up inside the node.

Another approach to reducing space requirements is to serialise the indexing structure into a single bit stream [9], which significantly reduces the storage overhead for managing objects in memory. Unlike the referenced paper, we apply such serialisation, not to the whole storage structures, but to each node separately.

Finally, there are PAM structures such as the LSM-tree proposed in [18] which are optimised for high update rates as they occur in transaction processing systems. However, the LSM-tree supports only 1-dimensional data and is optimised for persistent storage while the k-dimensional PH-tree is mainly aimed at in-memory indexing.

The most prominent SAM structure is the R-tree [10]. Modern forms of the R-tree include the PR-tree [1] and the Hilbert-tree [11], which optimise disk-IO, and the X-tree [3], which improves performance for high-dimensional data. SAM structures are primarily aimed at handling region data. While they can also be used to store points by using regions with size 0, they can not compete with PAM structures in this domain.

3. THE PH-TREE

The PH-tree is essentially a quadtree [6] that uses hypercubes, prefix-sharing [16] and bit-stream storage [9]. 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 [7]), 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 dimen-

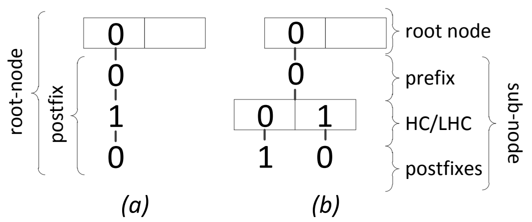


Figure 1: A sample 1D PH-tree with one 4-bit entry (a) and two 4-bit entries (b)

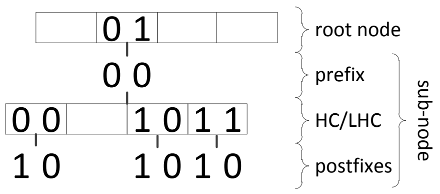


Figure 2: A sample 2D PH-tree with three 4-bit entries: (0001, 1000), (0011, 1000), (0011, 1010)

sions 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 range 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.

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. 3.1, which we extend to higher dimensions in Sect. 3.2. Section 3.3 discusses the storage of floating point values. Then we discuss space efficiency, query efficiency and update efficiency in Sect. 3.4, Sect. 3.5 and Sect. 3.6 respectively.

3.1 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 *prefix* of the sub-node. The sub-node also has two postfixes, 1 and 0.

3.2 The kD-PH-Tree

For the 1D case, our approach resembles the binary PATRICIA trie [17]. 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 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 (HC) and the position numbers are hypercube addresses. Below the root node is the prefix of the sub-node, consisting of a 0 for both values at $z_b = 2$. The HC of the sub-node references three postfixes, which represent the three entries in the tree.

The size of the HC 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 HC has 2^{64} fields. In order to reduce memory requirements for high values of k, we store sparsely filled HCs not directly but create a linear representation (LHC). Figure 3 shows on the left the HC 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 HC> $\rightarrow$ <postfix/sub-node>. The table is sorted by the HC 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 HC-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 HC representation, because it is almost completely filled and requires less space than the equivalent LHC representation. To determine whether HC 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

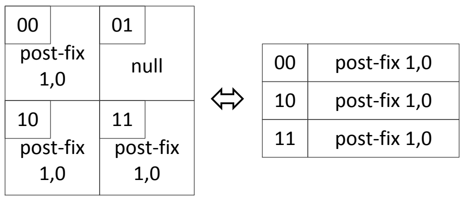


Figure 3: HC (left) and LHC (right) representation of references in a node

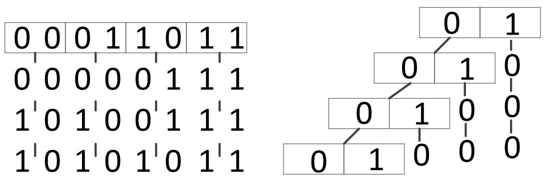


Figure 4: The two space worst cases of the PH-tree

node. The HC 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 HC 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 $O(\log n_p) \leq O(k)$.

3.3 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) {
    long raw = Double.doubleToRawLongBits(value);
    if (value < 0.0) {
      return raw ^ 0x7FFFFFFFFFFFFFFFL;
    }
    return raw;
}
```

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 $f_1 > f_2$. 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.

3.4 Space Efficiency

The PH-tree serialises most of the data of each node into a single bit-string [9]. 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. 3.2, 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 HC and $O(n * k * w)$ for LHC as the overhead for nodes. Since we assume a fully filled root node, we use the HC representation with $n = 2^k$ which results 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 can be seen in one of the experiments that we present in Sect. 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 prefix 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

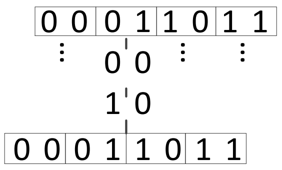


Figure 5: Best case for a sub-node with maximum prefix-sharing and maximum entries per node. Only the second sub-node is shown in full

a k-dimensional prefix of $w-2$ bits and is filled with the maximum of $n=2^k$ entries in HC 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 Sect. 4.3.6.

3.5 Query Efficiency

The PH-tree supports two types of queries, point queries and range queries. Point queries take an entry as parameter and check whether an equivalent entry already exists or not. Range 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 HC 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 HC 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 prefix and continues the search in the sub-node. The sum of bits extracted for prefixes and postfix is less than $w*k$ which we simplify to $O(w*k)$. When we add interleaving, query-internal search and prefix/postfix extraction, we get $O(w*k+w*1+w*k)$ for HC 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 $k\le 64$, 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.

Range queries start with a point query that locates the starting node, defined as the 'lower left' corner of the query range. 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 HC 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 range queries inside a node. If the node lies completely inside the query range, then the query iterator can simply iterate through all elements of the HC or LHC. In the case of HC, 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 postfix or sub-node is thus a constant time operation. If the node is only partly inside the query range 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 range. 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 HC/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 range 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.

3.6 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. 3.5. 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 HC 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.

4. EXPERIMENTAL EVALUATION

In this section we first explain our experimental set-up, then present the experiments together with an analysis of the results.

4.1 Experiment Set-Up

The experiments were executed in order to measure insertion speed, memory consumption, point queries and range queries. As hardware we used a desktop PC with 32GB RAM and an Intel i7-3770K 3.50GHz CPU. All tests were executed in main memory. All algorithms are implemented in Java and ran on Oracle JDK 1.7.0_25 64bit with -Xmx28G -XX:+UseConcMarkSweepGC. The JVM process contained the tested algorithm as well as the set of raw data in the shape of an array of floating point values. However, the figures below always refer to the memory consumption of the tree alone, measured using Java's internal memory figures[1]. Each test was executed in its own JVM, however each test was preceded by a warm-up run that executed all operations on a separate instance of the tested index with 100,000 random entries. After the loading phase, a series of `System.gc()` calls was performed to avoid interference of garbage collection with the following query tests.

For comparison we used two freely availably kD-tree implementations which we call KD1[2] and KD2[3]. While they have very similar behaviour, each has its own strengths and neither was consistently better than the other. We also measured the performance of two critical-bit-trees (bit-wise PATRICIA-tries) which we call CB1[4] and CB2[5]. In order to store k-dimensional entries, we interleaved the k values of each entry into a single bit-stream as for example done in [13]. The two implementations were not prepared for efficient range queries on interleaved values which we therefore did not measure. All algorithms, including the PH-tree, are single-threaded.

Each test was executed three times, the diagrams show the statistical averages of these three runs.

4.2 Datasets

For experimental validation of the PH-tree we used a 2D real-life dataset and several synthetic datasets based on the performance tests by [1].

As a real-life dataset we used the United States Census Bureau 2010 TIGER/Line KML dataset[6]. It consists of poly-lines that describe map features of the United States of America. For the tests, we extracted all points from counties belonging to mainland USA, which resulted in $36.8 * 10^6$ points. Then we removed all duplicates, resulting in $18.4 * 10^6$ unique points. The resulting x/y coordinates ranged between about $-125 \leq x \leq -65$ and $24 \leq y \leq 50$. We ignored

[1]We compared `Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();` before and after tree loading
[2]`http://home.wlu.edu/~levys/software/kd/`
[3]`http://www.savarese.com/software/libssrckdtree-j/`
[4]`https://github.com/rkapsi/patricia-trie`
[5]`https://github.com/jfager/functional-critbit/`
[6]`https://www.census.gov/geo/maps-data/data/tiger.html`

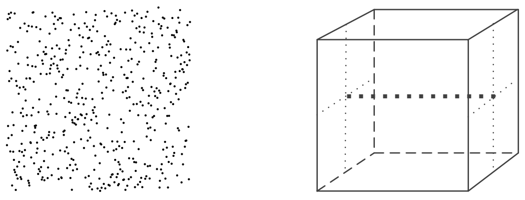

(a) 2D CUBE (b) 3D CLUSTER

Figure 6: The CUBE and CLUSTER datasets

the third dimension in which all points were at 0.0. While the dataset appears to be aimed at 32bit single precision, we use 64bit double precision because one of the kD-tree implementations supported only double precision.

We also used two types of synthetic datasets. The CUBE dataset is a set of up to 100,000,000 points distributed uniformly at random between 0.0 and 1.0 and independently in every dimension. An example is shown in Fig. 6a. The coordinates are 64 bit double precision floating point values that were converted to 64 bit integers using the conversion function from Sect. 3.3.

The CLUSTER is an extension of the 2D synthetic dataset described in [1]. The dataset consists of a line of 10000 evenly spaced clusters of points. Each cluster extends 0.00001 in every dimension and is filled with evenly distributed points. The line of clusters stretches from 0.0 to 1.0 on the x-axis and is parallel to the axis of every other dimension with an offset of 0.5, except for space measurements where we also test with an offset of 0.4. In total, the CLUSTER dataset contains up to 50,000,000 points. An illustration of the CLUSTER dataset is depicted in Fig. 6b.

All points are generated randomly, however all tests use the same set of randomly generated data.

In the following figures, CLUSTER is abbreviated to CL and CUBE to CU.

4.3 Experimental Results

4.3.1 Loading

For each test, an empty tree was loaded with the amount of entries indicated on the x-axis, the resulting total time was divided by the number of entries. The diagrams therefore show the average loading time per entry in μs (10^{-6} seconds).

For the TIGER/Line datasets in Fig. 7a, the kD-trees show a somewhat irregular performance which is probably caused by the fact that the data is loaded for US-county after US-county, where different counties have very different data distribution properties that affect loading performance. The PH-tree and the CB-trees, however, show almost identical and consistently *decreasing* insertion times for a growing tree. This is owed to the increasing prefix-sharing, which means that, with a growing tree, the postfix gets smaller and can be serialised more quickly. The PH-tree improves even slightly faster than the CB-trees, which is owed to the increasing switching from LHC to HC in most of the nodes which happens due to the small dimensionality of $k = 2$. For HC nodes the point query performance and update performance are both $O(k * w)$.

Loading was also tested using the CUBE dataset with varying amounts of 3D 64bit entries as shown in Fig. 7b. The figure shows that the PH-tree is on par with the kD-trees and about 50%-80% faster than CB-trees when loading 10^6 entries. However, when loading more entries, the loading time per entry of the PH-tree increases only slightly in parallel to the CB-trees, while the kD-trees get increasingly slower.

For the CLUSTER dataset in Fig. 7c, the performance of the PH-tree is similar to the TIGER/Line dataset and shows virtually constant behaviour.

These results are in line with the prediction in Sect. 3.6 that insertion time is largely independent of the number of objects and only depends on the number of nodes, which is limited by w and the number of dimensions k, which are both constants. The resulting insertion complexity is $O(w * k)$.

4.3.2 Point Queries

For the TIGER/Line data, all point queries are located inside the minimum and maximum values of each coordinate. For the synthetic data, all queries are located between 0.0 and 1.0 in each dimension. Point queries were created randomly, having a 50% chance of querying an existing data point or otherwise querying a random coordinate in the allowed query range. Each test performed 1,000,000 point queries.

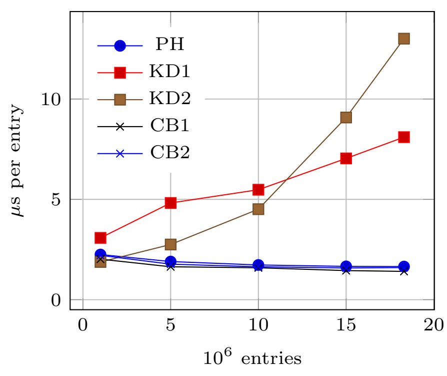
Fig. 8a, 8b and 8c show the query performance for the TIGER/Line, CUBE and CLUSTER datasets for varying amounts of entries. For better visualisation, Fig. 8a also shows the times of the PH-queries multiplied by 10. Compared to the other trees, the PH-tree performs consistently better, except for very small datasets, showing very little decrease in performance for large datasets.

4.3.3 Range Queries

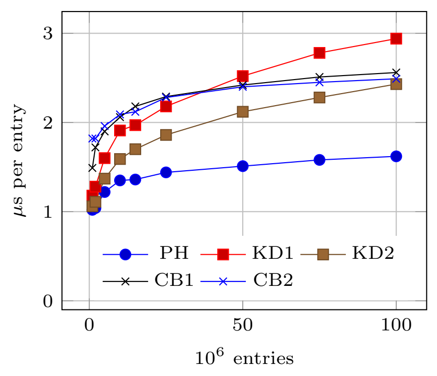
For the TIGER/Line and CUBE datasets, range queries are rectangles or k-dimensional cuboids where all edges have random length, except one randomly chosen edge that is adjusted so that the query covers 1% of the area of TIGER/Line data or 0.1% of the volume of CUBE data. The CLUSTER queries are cuboids that extend from 0.0 to 1.0 in every dimension except for the x-axis where they have an extension of 0.01% and are randomly located between 0.0 and 0.1. This was done because, with growing k, it becomes increasingly unlikely that a random cuboid intersects with the row of clusters in the centre of the data space. Having the query cuboids extend from 0.0 to 1.0 in most dimensions solves this problem. The diagrams show the average query execution time divided by the number of returned entries.

In Fig. 9a, the KD1 and KD2 trees show slightly irregular but decreasing performance by a factor of 5 to 10 for the TIGER/Line dataset. The PH-tree starts with $0.25\mu s$ about 20 times faster than the KD-trees and slows down only by a factor of 2 to $0.5\mu s$ per returned entry for $1.8 * 10^7$ entries. For the CUBE dataset, all three trees show linear scaling with growing tree size as shown in Fig. 9b. However, while the PH-tree shows initially the same performance as the kD-trees, it progresses at a different angle and is about 2.5 times faster for $n = 10^8$.

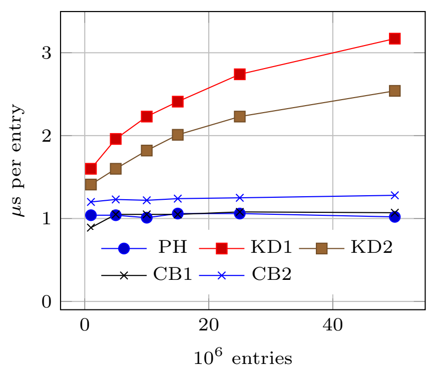
Figure 9c finally shows the CLUSTER dataset for which we tested the queries in kD-trees only for up to $5*10^6$ entries because of the long query execution time. The performance of the KD-trees decreases from ≈ 300 or 440 to $25‘000\mu s = 25ms$ per returned entry, while the performance of the PH-tree *increases* with the number of elements in the tree, in



(a) 2D TIGER/Line

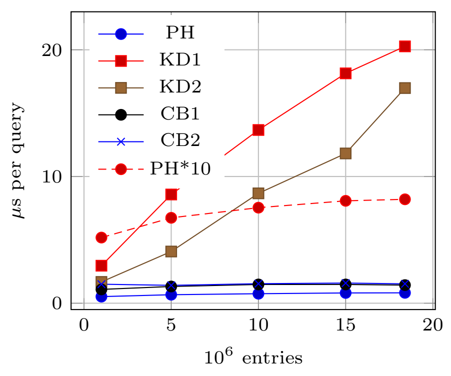


(b) 3D CUBE

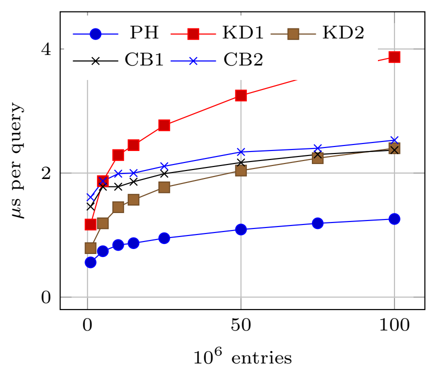


(c) 3D CLUSTER

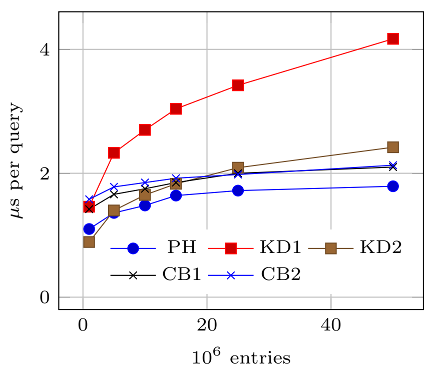
Figure 7: Insertion times per entry for the TIGER/Line, CUBE and CLUSTER datasets



(a) 2D TIGER/Line



(b) 3D CUBE



(c) 3D CLUSTER

Figure 8: Point query times for the TIGER/Line, CUBE and CLUSTER datasets

this case from from $0.40\mu s$ to $0.19\mu s$, which is comparable to the performance with the TIGER/Line dataset. Similar to the update performance, we attribute this to the increasing use of HC-nodes which provide excellent performance.

Range queries on CB-trees are not shown because they resulted in nearly full scans approaching $O(n)$ complexity for the available implementations. While it is possible to provide more efficient range queries, there are some inherent limitations with such queries in CB-trees, for example the susceptibility to the ordering of dimensions because they split the space in each node along one dimension. For high k, if a query has low selectivity on the dimensions of the upper nodes, then all sub-nodes must be searched to the depth where the query has a high selectivity. This makes it difficult to find the correct starting node and may result in traversing many nodes unnecessarily when building the result set. This is also visible from the high-k point query performance of CB-trees discussed in Sect. 4.3.7. For large k, the nodes that need to be searched can be significant, especially because the CB-tree has up to 2^{k*w} nodes on up to $k*w$ levels and cannot rebalance itself as the kD-tree can. For comparison, as mentioned before, the HC/LHC approach of the PH-tree is more independent of the ordering of dimensions and limits the tree depth to w.

4.3.4 *Unloading*

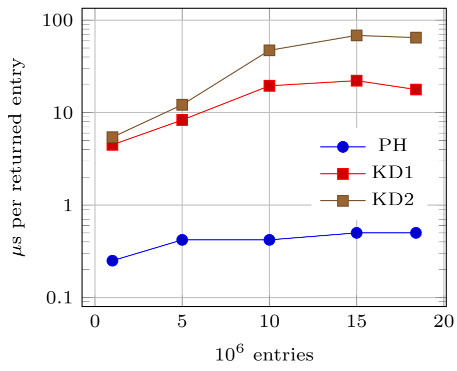
Due to space limitations, we do not show the results for tree unloading. The results are very similar to tree loading, but a bit faster. The only exception is the KD2-tree, which appears to have a slightly faulty implementation as it scales exponentially with k, which became apparent in high dimensional tests.

The PH-tree is consistently about 10% faster for delete operations compared to insert operations. At least two properties contribute to this speed-up. First, compared to insert operations, delete operations require allocation of smaller objects (byte[]), if at all, to hold smaller amounts of data. Second, our implementation of shift-left (used by deletion) copy operation is faster than shift-right (used by insertion).

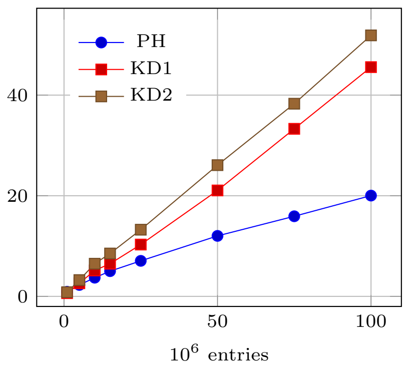
4.3.5 *Space*

The space consumption is measured by comparing the memory consumption of the JVM process before and after index creation. The resulting numbers consistently differed less than 5% from the space calculated by summing up the required bytes of all nodes. For reference the following figures also show two naive storage approaches, the *plain array* and the *object array*. In the plain array, all data is stored in a single `double[]` (C/C++: `long float[]`) of size $k*8*n$ bytes. In the object array, every entry has its own object in memory with k attributes of type `double`. Including memory alignment and an array of references to these objects, the required space is $(k*8+16+4)*n$ bytes.

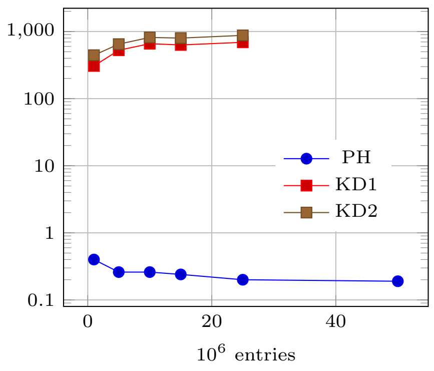
Table 1 shows the memory consumption in bytes per entry for different datasets and storage structures. The table shows the results from the 2D TIGER/Line 64 bit dataset



(a) 2D TIGER/Line



(b) 3D CUBE



(c) 3D CLUSTER

Figure 9: Range query time for the TIGER/Line, CUBE and CLUSTER datasets

	PH	KD1	KD2	CB1	CB2	d[]	o[]
TIGER	68	87	95	79	61	16	36
CUBE	46	95	103	88	69	24	44
CLUST.	43-55	95	103	88	69	24	44

Table 1: Required bytes per entry for $n \geq 5,000,000$ 64 bit entries (d[]=double[], o[]=object[])

10^6 entries	1	5	10	15	25	50
CLUSTER$_{0.4}$	48	45	44	44	43	43
CLUSTER$_{0.5}$	55	48	46	45	44	43

Table 2: Required bytes per entry for the CLUSTER datasets at $k = 3$

and the 3D CUBE and CLUSTER 64 bit datasets, which were constant for trees between $5 * 10^6 \geq n \geq 10^8$. The only exception is the PH-tree when storing the CLUSTER dataset, which showed significant variation, improving from 55 bytes per entry for $n = 1,000,000$ to 43 bytes per entry at $n = 50,000,000$. This behaviour is discussed in detail in Sect. 4.3.6. For the larger CUBE and CLUSTER datasets, the PH-tree requires about the same amount of memory as a plain `object[]` which is about half the space of the two kD-trees. The two CB-trees require between 50% and 80% more space than the PH-tree.

4.3.6 CLUSTER$_{0.4}$ vs CLUSTER$_{0.5}$

As in the original test proposed by [1], the point clusters of the CLUSTER test centre around 0.5 on every axis except the x-axis. However, this is something of a worst case for the PH-tree in terms of space requirements. For comparison, we performed the same tests with the clusters located at 0.4, which we henceforth call CLUSTER$_{0.4}$. The original version will be called CLUSTER$_{0.5}$. Table 2 shows how the required bytes per entry develop for increasing n. For few entries, the CLUSTER$_{0.5}$ dataset requires 55 bytes per entry, which is 15% more than is needed for CLUSTER$_{0.4}$. It is only for high n that the two datasets approach the same space requirement of 43 bytes per entry.

A difference of 15% does not appear significant, but it is critical for two reasons. First, it shows that the PH-tree differs from kD-trees and CB-trees in that its behaviour depends on the absolute coordinates of the data. Second, if we

k	2	3	5	10	15
CUBE	623	450	284	199	138
CLUSTER$_{0.4}$	684	534	397	139	54
CLUSTER$_{0.5}$	718	629	743	995	932

Table 3: Number (thousands) of nodes in a PH-tree with varying k holding 10^6 64 entries

look at datasets with increasing k, as shown in Fig. 10, we see that the 15% for $k = 3$ increases dramatically for larger k. The CUBE dataset is shown for reference.

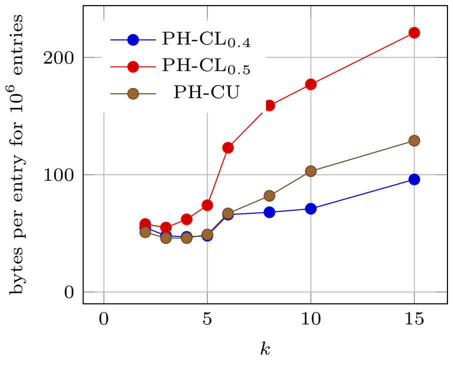


Figure 10: Required bytes per 64 bit entry for $n = 10^6$ and increasing k for the CLUSTER$_{0.4}$, CLUSTER$_{0.5}$ and CUBE datasets for the PH-tree

This increase in required bytes per entry occurs due to the node count of PH-trees with the different CLUSTER datasets. As Tbl. 3 shows, the CLUSTER$_{0.5}$ data set has 932,000 nodes for 1,000,000 entries and $k = 15$, whereas the CLUSTER$_{0.4}$ requires only 54,000 nodes.

This sensibility to coordinate position can be explained with the way that floating point numbers are represented in the PH-tree. As discussed in Sect. 3.3, floating point numbers are converted to integers in order to store them in the PH-tree. The converted numbers are similar to the IEEE representation which stores the sign in the highest bit, followed by the exponent, followed by the fraction.

If we look at the two CLUSTER datasets, they contain point clusters of length 0.00001, which means that they

float	IEEE 64 bit integer	sign	exponent	fraction
0.39999	4600877199177713619	0	0111111.1101	1001.10011001.01101111.10101000.00101110.1000111.11010011
0.40000	4600877379321698714	0	0111111.1101	1001.10011001.10011001.10011001.10011001.10011001.10011010
0.49999	4602678639028661817	0	0111111.1101	1111.11111111.11010110.00001110.10010100.11101110.00111001
0.50000	4602678819172646912	0	0111111.1110	0000.00000000.00000000.00000000.00000000.00000000.00000000

Table 4: IEEE Binary64 bit representation of different floating point values. The '.' mark every 8th bit for easier reading

reach from 0.49995 to 0.50005 for CLUSTER$_{0.5}$ and from 0.39995 to 0.40005 for CLUSTER$_{0.4}$. The important difference, as illustrated in Tbl. 4, is that in the floating point representation, going from 0.4999999 to 0.5 causes a change in the exponent. This means that all points in the CLUSTER$_{0.5}$ dataset may differ in at least one dimension in the exponent, i.e. at the 11th or 12th bit from the left. In the CLUSTER$_{0.4}$ dataset, all points have the same exponent and differ only at the 25th bit. If values differ in their high bits in this way, we get a scenario that resembles the worst case scenario discussed in Sect. 3.4. For $k = 3$, the tree is split at the level of the changed exponent bit into $2^k = 2^3 = 8$ sub-trees. This reduces the density of the sub-trees which causes a worse entry-to-node ratio and reduces prefix-sharing. For $k = 3$ the effect is only marginally noticeable. However for $k = 15$ we get $2^k = 2^{15} = 32768$ sub-trees which results in significantly increased space requirements.

One way to counter this effect is obviously to move data coordinates before storing them in the PH-tree. Another approach is to use integer representation, for example instead of storing floats that represent $[meters]$ one could store integers that represent $[nm] = 10^{-9}[m]$.

In terms of performance, the two CLUSTER variants behave very similarly with less than 10% difference for $k = 3$. For larger k, some differences emerge as we will show in the next section.

4.3.7 Datasets with $k > 3$

The Figures 11 and 12 show the insertion performance for trees with $n = 10^7$ elements and various $k \leq 10$. The PH-tree scales well until about $k = 8$ dimensions, but then, as expected, the large node size inhibits efficient updates to the tree. Element deletion performance is not shown but has very similar behaviour with the exception of test case KD2-CL$_{0.5}$, where the KD2 tree showed exponentially degrading performance for increasing k. To keep the following diagrams simple, we show as reference only the KD2 tree, which performed best from the two kD-trees. Also, as the two diagrams show, the CB-trees, where we show only the best performing CB1-tree, scaling linearly with growing k.

Point query performance is, as shown in Fig. 13a and Fig. 13b, rather independent of k for the PH-tree and the kD-trees, however the PH-tree is consistently faster. Again, the CB-trees scale linearly and cannot compete with the other trees for increasing k.

As shown in Fig. 13c, range query performance depends on the dataset. CLUSTER range query time for the kD-trees is not shown as it was orders of magnitude slower than the PH-tree. This behaviour is expected from the results shown earlier in Fig. 9c. The PH-tree shows linear scaling with the CUBE dataset due to the prevalent LHC representation. While range query times for CLUSTER$_{0.5}$ grow exponentially for $k > 10$, the better prefix sharing in the CLUSTER$_{0.4}$ dataset requires much less nodes and in-

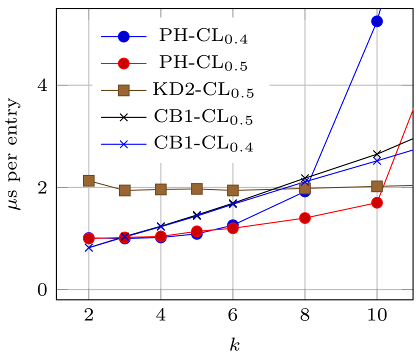


Figure 11: Insertion times for varying k and 10^7 64 bit entries for CLUSTER datasets

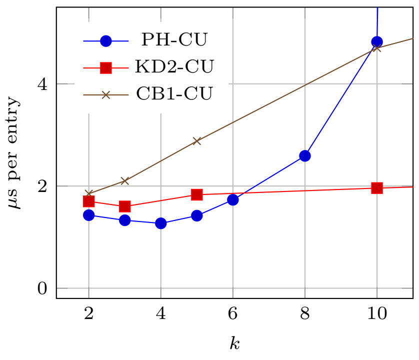
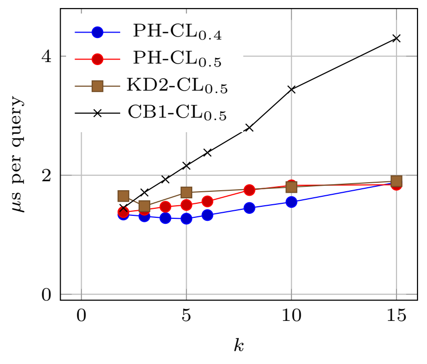


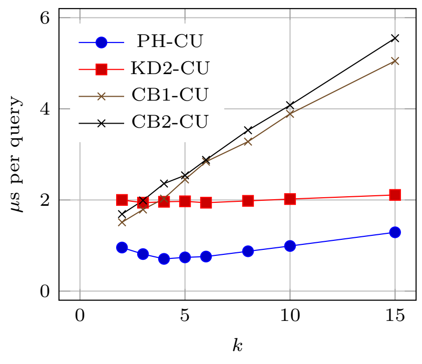
Figure 12: Insertion times for varying k and 10^7 64 bit entries for the CUBE dataset

creased HC representation which allows near constant scaling which consistently requires between 0.19 and 0.25 μs per returned entry.

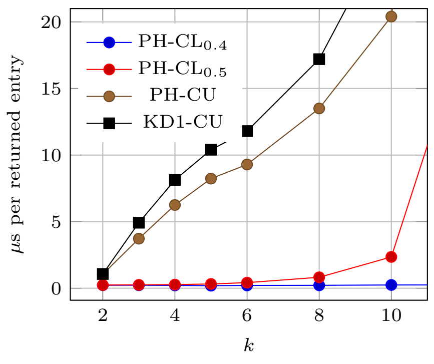
The test results in Fig. 14 and Fig. 15 show the memory consumption per entry when filling the trees with 10^7 entries, using varying numbers of dimensions. The two kD-trees and CB-trees show virtually no difference in space efficiency between storing CUBE and CLUSTER datasets. Unlike the other trees, the space requirements of the PH-tree vary significantly. After starting around 50 bytes per entry for 2D entries, it drops to a low at 4D, which means that storing 10^7 3D, 4D or 5D entries takes less space than storing the same amount of 2D entries. Then, depending on the dataset, space requirements rise again. The best performance is shown with the CLUSTER$_{0.4}$ dataset which requires only 60% of the space required by a `double[]` at $k = 15$. The worst performance is caused by the CLUSTER$_{0.5}$ dataset due to its bad entry-to-node ratio which requires 9,901,854 nodes for 10,000,000 entries. However, even in this case, the



(a) CLUSTER point queries



(b) CUBE point queries



(c) CLUSTER and CUBE range queries

Figure 13: Query execution times for varying k and 10^7 64 bit entries. KD-CL times in (c) are not shown as they range between 500 and 1000 μs per returned entry

PH-tree requires over 15% fewer bytes per entry than the KD1 tree, which is also smaller than the KD2 tree.

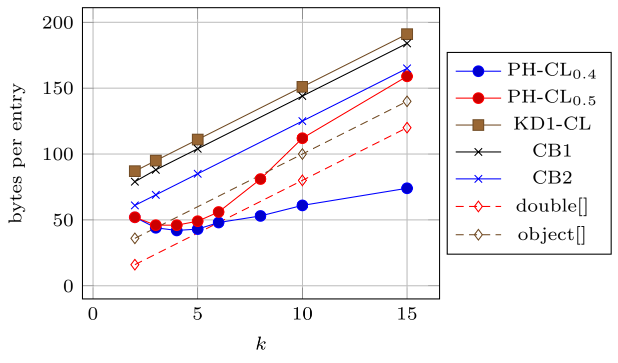


Figure 14: Space requirements per 64bit entry depending on k for 10^7 entries for the CLUSTER datasets

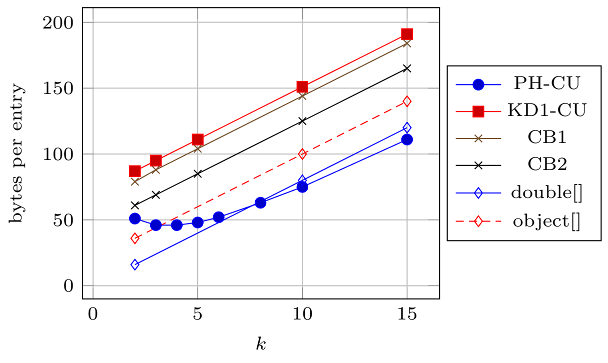


Figure 15: Space requirements per 64bit entry depending on k for 10^7 entries for the CUBE dataset

In summary, the PH-tree performs well with respect to space requirements and can easily compete with un-indexed structures such as `object[]` or even `double[]`. However, the space consumption depends on the characteristics of the dataset. This is different from the other trees where the required bytes per entry were virtually independent of the dataset.

4.4 Results

The results clearly show that the PH-tree is very space efficient, especially for larger datasets. This is not surprising, since neither kD-trees nor the CB-tree use serialisation to save space. What is relevant however is that despite using prefix-sharing and serialisation to a bit-stream, the PH-tree shows excellent performance properties. Update and query performance appears almost independent from the number of entries in the PH-tree. For larger datasets, this results in better performance than the tested kD-tree implementations. However, query performance clearly does depend on the type of data in the tree.

While the PH-tree works well with all tested datasets, it shows its strength with datasets such as TIGER/Line or CLUSTER, where data points are not evenly spread but concentrated in some areas. If such dense datasets are large, the PH-tree can benefit from the increasing prefix-sharing, which saves space, and the increasing prevalence of hypercubes in the nodes, which provide superior performance for navigation and updates. Furthermore, allowing up to 2^k children per node and limiting the maximum tree depth to w reduces the overall number of nodes as well as the number of nodes visited for any kind of queries. The only drawback is that certain dense datasets, such as the CLUSTER$_{0.5}$ dataset, can cause less than ideal space requirements for high values of k. However, the resulting space requirements are still not worse than the space requirements of kD-trees.

It should be noted that we compared our implementation of the PH-tree with publicly available free implementations of kD-trees and CB-trees. We acknowledge that the performance of the trees depends partly on the effort of their respective developers. This may affect some of the measurements but should not change the overall trends.

5. OUTLOOK

While the current implementation of the PH-tree works well for a number of scenarios, there are a number of more or less obvious extensions that we would like to investigate in the future.

First, currently all node-data is stored in a single bit-string which makes insert and delete operations slow for $k > 8$. Splitting these bit-strings into sizeable chunks would improve update performance. At the same time, the chunk size could be chosen so that a chunk fits on a disk-page, which would improve performance for a persistently stored PH-tree.

Second, support for nearest-neighbour searches would be desirable. An early prototype implementation indicates that such searches can be efficiently performed.

Third, the fact that at most two nodes are modified with each update makes the PH-tree suitable for concurrent access and updates.

Fourth, it would be desirable to verify the PH-tree further with real world k-dimensional data.

Finally, one of the next steps is to allow different bit-width w for each dimension of an entry. At the same time, the current limit of $w = 64$ could be increased to allow values with arbitrary length. This would also allow the PH-tree to be effectively used as a compact and fully indexed table of a relational database.

6. CONCLUSIONS

We have presented the PH-tree as an approach for combined storage and indexing of multi-dimensional data. The tests showed that the PH-tree is space efficient, requiring significantly less space than structures such as the kD-tree and in certain cases even less space than naive plain array storage. As a multi-dimensional index structure, it showed competitive performance for updates and queries, especially for larger datasets. For large and dense datasets, such as the tested CLUSTER dataset, the PH-tree performs range queries several orders of magnitude faster than the tested kD-trees. For a densely filled tree such as resulting from the TIGER/Line or CLUSTER dataset, the tree showed even super-constant performance for updates and range queries which became faster with a growing number of entries. Compared to CB-trees (binary PATRICIA-tries), the PH-tree shows comparable characteristics for $k = 2$ but scales much better for most datasets with growing k, especially in point query performance. The only exception is insertion performance, where the current implementation becomes slower than CB-trees for $k = 10$ and beyond.

In summary, the combination of multi-dimensional indexing with space efficiency and good performance makes it a useful alternative for many applications.

7. ACKNOWLEDGEMENTS

We would like to thank Dr. Stefania Leone, who provided valuable feedback and support on early versions of this paper. This research was partially funded by the Hasler Foundation, Switzerland.

8. REFERENCES

[1] L. Arge, M. D. Berg, H. Haverkort, and K. Yi. The Priority R-tree: A Practically Efficient and Worst-Case Optimal R-tree. *ACM Transactions on Algorithms*, 4(1):9:1–9:30, March 2008.

[2] J. L. Bentley. Multidimensional Binary SearchTrees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[3] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An Index Structure for High-Dimensional Data. *Readings in multimedia computing and networking*, page 451, 2001.

[4] J.-M. Chang and K.-S. Fu. Extended Kd-tree Database Organization: A Dynamic Multiattribute Clustering Method. *Software Engineering, IEEE Transactions on*, 3:284–290, 1981.

[5] L. Chen, G. Cong, and X. Cao. An Efficient Query Indexing Mechanism for Filtering Geo-Textual Data. In *Proc. of the 2013 ACM Intl. Conf. on Management of Data*, SIGMOD '13, pages 749–760, 2013.

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

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

[8] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[9] U. Germann, E. Joanis, and S. 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.

[10] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proc. Intl. Conf. on Management of Data*, SIGMOD '84, pages 47–57. ACM, 1984.

[11] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. Technical Report TR 93-19, ISR University of Maryland, USA, 1993.

[12] A. K. Khamayseh and G. Hansen. Use of the Spatial kd-Tree in Computational Physics Applications. *Communications in Computational Physics*, 2, 2007.

[13] P. Kirschenhofer, H. Prodinger, and W. Szpankowski. Multidimensional Digital Searching and Some New Parameters in Tries. Technical Report CSD TR 91-052, Department of Computer Science, Purdue University, West Lafayette, Indiana, USA, 1991.

[14] D. Lomet and B. Salzberg. The hB-tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *ACM Transactions on Database Systems*, 15:625–658, 1990.

[15] K. Markov, K. Ivanova, I. Mitov, and S. Karastanev. Advance of the Access Methods. *International Journal on Information Technologies and Knowledge*, 2/2:123–135, 2008.

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

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

[18] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[19] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vittery. Bkd-tree: A Dynamic Scalable kd-tree. In *Proc. Intl. Symposium on Spatial and Temporal Databases*, SSTD '03, pages 46–65, 2003.

[20] J. T. Robinson. The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proc. Intl. Conf. on Management of Data*, SIGMOD '81, pages 10–18, New York, NY, USA, 1981. ACM.