Human Behavior, Development and Society
E-ISSN 2651-1762, Vol 25 No 2, August 2024

Exploring the Potential of k^n-Tree for Efficient Representation of n-ary Relations

Sebastián Alexis Moraga, Asia-Pacific International University, Thailand

Date Received: 28 May 2024 Revised: 16 July 2024 Accepted: 19 July 2024

Abstract

The objective of this experimental study was to investigate the scalability of k^n-trees, a compact data structure designed for representing n-ary relations, compared to a baseline based on a plain representation of adjacency lists. A literature review of compact data structures was conducted, focusing on k^n-trees and their potential for efficient n-ary data representation. To assess scalability, experiments comparing k^n-tree performance against the baseline using set intersection as a benchmark were conducted. Results demonstrated superior k^n-tree scalability in terms of time and memory, especially for high-dimensional and clustered datasets. On average, k^n-trees were eight times faster and consumed 35 times less memory than the baseline. The study also analyzed the impact of the order parameter k on performance, revealing a trade-off between space efficiency and query time. This study provides valuable insights into the practical applicability of k^n-trees for managing and querying high-dimensional data.

Keywords: *Compact data structures, k^n-tree, n-ary relations, scalability*

Introduction

The exponential growth of high-dimensional and multi-attribute data necessitates efficient data management solutions. Traditional data structures often incur significant memory overhead, hindering the scalability of processing and analysis. Compact data structures, which aim to minimize space while maintaining query efficiency, have emerged as a promising approach to tackle this challenge. The k^n-tree, a compact representation for n-ary relations, holds potential for modeling multi-dimensional data, particularly in scenarios where space efficiency is crucial.

Previous work has explored the k^2-tree, a specific case of the k^n-tree for 2-ary relations, but further investigation is needed to assess the scalability of k^n-trees for higher dimensions and varying data distributions. This research addresses this gap by delving into the scalability of k^n-trees, focusing on their performance in set intersection – a critical operation in data analysis. To evaluate the scalability of k^n-trees, a series of experiments were conducted comparing their performance against a baseline using a plain representation of adjacency lists. The set intersection operation served as a benchmark to assess the efficiency of both representations. The insights gleaned from these experiments shed light on the practical applicability of k^n-trees for managing and querying high-dimensional data in real-world scenarios.

Literature Review

Compact data structures have emerged as a powerful paradigm for representing data with minimal space that makes them particularly valuable for handling the ever-growing amount of data. This section provides details on these data structures, dissecting their design, implementation basis, operations, and complexity. While not comprehensive, this review focuses on the foundational compact data structures found in the literature.

Fixed-Size Arrays

The traditional approach of using a fixed-size array A with n elements encoded in w bits can be wasteful if the data requires less than w bits for representation. To address this problem, Navarro (2016) proposed determining the minimum number of bits, denoted by ℓ, required to encode the elements in an array, where $\ell \leq w$. Then, the data can be packed within words of w bits.

The implementation utilizes a virtual bit array B that stores actual data elements, with a total size of ℓn bits to accommodate all elements of A. Each element in B is encoded using ℓ bits. Additionally, a word array W of integers of w bits is used, large enough to pack all encoded elements from B. The packing strategy utilizes bit-level operations. Individual elements in B are placed in W, starting from the most significant bit. The specific location within a word in W depends on the element's position j in B. The formula $r = \big((j-1)\ mod\ w\big) + 1$ is used to calculate this starting position (r) within a word in $W\left[\left\lceil \frac{j}{w} \right\rceil\right]$.

This design allows for reading and writing individual elements. On the one hand, reading a bit at position j in B involves calculating the word index in W and performing bitwise operations to isolate the desired bit. On the other hand, reading a whole encoded integer within a single word in W requires calculating the chunk to be read, and performing bitwise operations to extract the relevant ℓ bits. If the encoded integer spans two words in W, additional calculations are needed to handle bits across words. Writing a value involves clearing the existing data in W before writing the new value. The clearing process also utilizes bitwise operations and mechanisms to handle bits across words.

In summary, the space of a fixed size array is $O\left(\left\lceil \frac{\ell n}{w} \right\rceil w\right)$, and time for reading and writing is $O(c)$.

Variable-Size Arrays

Variable size arrays (Raman et al., 2007) address the limitation of fixed size arrays by allowing elements to have different bit lengths in B. There are two main design approaches for variable size arrays. The first uses a separate array P to store the length information for each encoded element. Then, P is used to determine the starting and ending positions of elements within B. The other approach is using a strategy for self-describing length. This approach embeds the length information within the encoded element itself. This eliminates the need for a separate data structure for storing lengths. However, it might introduce some overhead for encoding/decoding lengths.

In both approaches, the element locations in B differ from fixed size arrays. Instead of fixed intervals, the starting position of$A[i]$ depends on the sum of encoded lengths ℓ, i.e., $\sum_{j=1}^{i-1} \ell_j$. The implementation involves a main bit array B, and a pointer array P. The array B stores the actual encoded data elements, with each element i using variable bits ℓ_i. The array P acts as an index of B.

There are two variations for P. The first one is implementing P as an array of sampled pointers. This approach partitions the array A into consecutive blocks of k elements. Then, P stores the starting position for each block in B. To access $A[i]$, the block that it belongs to is calculated and then the corresponding pointer in P is used to find the starting position in B. The last step is looking for the target element within that block. Additionally, if the encoding in B uses γ-coding (Elias, 1975), this property can be leveraged to efficiently skip to the desired element within a block by reading only the codeword headers. The other variation is implementing P along with another array P' of dense pointers (Ferragina & Venturini, 2007; Raman et al., 2007) to avoid the need for calculating the sum of preceding lengths during access. P' stores offsets within B, directly pointing to the starting position of each $A[i]$.

This design supports reading and writing individual elements. However, accessing requires additional computation to look up starting bits; this can be reduced by using dense pointers. In summary, for variable size array with sampled pointers and γ-coding, the time cost is $O(k \lg(x))$, where x is a γ-coded element. Its space cost is $O(n \lg(w))$ bits, with $k \cong \ln(2)$. For variable size array with dense pointers, the time cost is $O(c)$, and space is $O(n \lg(w))$.

Bitvectors

The bitvector (Jacobson, 1989) is a bit array designed to use space close to its zero-order empirical entropy, i.e. $n\mathcal{H}_0(B)$ bits. It achieves this by dividing the bit array into blocks of size b and encoding them as a pair (c_i, o_i), where c_i is the class (number of 1s) of the block, and o_i is the offset within c_i that identifies a specific permutation among what is allowed by c_i.

The implementation uses an array C for classes, and an array O for corresponding offsets. The lengths of the offsets are determined by the class and stored in an array L, where $L[c]$ gives the number of bits needed to encode a class c offset.

The bitvector supports $access(B, i)$ to retrieve the i-th bit in B; $rank(B, i)$ to count the number of occurrences of a specific bit (0 or 1) up to position i; and, $select(B, j)$ to find the position of the j-th occurrence of a specific bit (0 or 1).

The space is $n\mathcal{H}_0(B) + \frac{n}{b}\lg(b) + O\left(\frac{n}{b}\right) + wb^2$ bits (Raman et al., 2007; Pagh, 2001), where n is the number of bits, and w is the word length. The worst-case time for all operations is $O(b)$, but it can be improved (Clark, 1997; Jacobson, 1989) by using precomputed tables for encoding and decoding. In practice, the $O(b)$ time cost is often sufficient, as the block size b is typically chosen to be small.

Wavelet Trees

Wavelet trees (Grossi et al., 2003) are designed to represent and query sequences over larger alphabets. They hierarchically decompose the alphabet, creating a binary tree structure where each node corresponds to a range of symbols. The tree's root represents the entire alphabet, and each leaf represents a single symbol. Internal nodes store bitvectors that indicate whether a symbol in the sequence belongs to the left or right half of the alphabet range at that level.

Wavelet trees support $access(S, i)$ to retrieve the symbol at position i in the sequence S; $rank_c(S, i)$ to count the number of occurrences of symbol c up to position i by traversing the path from the root to the leaf of c, accumulating the counts from the bitvectors along the way; and, $select_c(S, j)$ to find the position of the j-th occurrence of symbol c by traversing the tree from the leaf of c to the root, using the bitvectors to determine the position at each level.

The space cost is $n \lg(\sigma) + o(n \lg(\sigma)) + O(\sigma w)$ bits, where n is the length of the sequence, σ is the alphabet size, and w is the word size. The $o(n \lg(\sigma))$ term is on account of the extra space required for $rank$ and $select$. The worst-case time complexity for all operations is $O(\lg(\sigma))$.

The wavelet tree can use compressed representations for the bitvectors, achieving space close to $n\mathcal{H}_0(S)$, while maintaining the same time cost (Barbay & Navarro, 2013; Golynski et al., 2008).

Sequences

The sequence $S[1, n]$ is designed as a generalization of bitvectors. It supports $access$, $rank$, and $select$ operations while minimizing space usage. Two primary approaches are employed. One uses a permutation-based representation (Golynski et al., 2006) along with bitvectors to achieve efficient operations. It is particularly suitable for large alphabets. The other uses wavelet trees (Grossi et al., 2003), enabling efficient operations through binary $rank$ and $select$ on bitvectors. When using permutation-based representation, the sequence is divided into chunks. Each one is represented using a permutation and a bitvector. The bitvectors store the frequency of each symbol within the chunk, while the permutation encodes symbols' order. This enables efficient $rank$ and $select$ operations within chunks.

When using wavelet trees, it recursively partitions the alphabet into halves, creating a binary tree structure. Each node is associated with a bitvector that indicates whether a symbol belongs to the left or right half at that level. This enables efficient operations on account of the structures used.

Both implementations support $access(S, i)$ to retrieve the symbol at position i in the sequence; $rank_c(S, i)$ to count the number of occurrences of symbol c up to position i; and, $select_c(S, j)$ to find the position of the j-th occurrence of symbol c.

In summary, for permutation-based representation, the space cost is $n \lg(\sigma) + n\, o(\lg(\sigma))$ bits. The time cost (Grossi et al., 2010) is $O(\lg(\lg(\sigma)))$ for $access$, $O(\lg(\lg(\sigma)))$ for $rank$, and $O(c)$ for $select$. For a representation using wavelet trees, space using a plain approach is $n \lg(\sigma) + o(n \lg(\sigma)) + O(\sigma w)$ bits, and $n\,\mathcal{H}_0(S) + o(n \lg(\sigma)) + O(\sigma w)$ bits for a compressed approach (Barbay and Navarro, 2013). Time cost is $O(\lg(\sigma))$ for $access$, $O(\lg(\sigma))$ for $rank$, and $O(\lg(\sigma))$ for $select$ both approaches. In practice, wavelet trees are often preferred for small to moderate alphabet sizes due to its lower space overhead. For larger alphabets, permutation-based representation can be more space-efficient, especially when combined with alphabet partitioning techniques.

Level-Order Unary Degree Sequence (LOUDS)

The Level-Order Unary Degree Sequence (LOUDS) representation, first reported by Jacobson (1989) and implemented in Delpratt et al. (2006), is designed to encode the topology of an ordinal tree by using a bitvector to store the unary degree sequence of the tree in a level-wise order. The unary degree sequence is made of $1s$ followed by 0, where the number $1s$ are the number of children of a node.

The implementation uses a bitvector. The first two bits are set to 10, and then the tree is traversed in level-order, appending the unary degree sequence of each node to the bitvector. The bitvector results of $n + 1$ $0s$ (one 0 per node, plus the initial one), and n $1s$ (one 1 per edge – $n - 1$ –, plus the initial one). Therefore, the bitvector length is $2n + 1$.

LOUDS supports operations such as $root(v)$ to retrieve the root of the tree; $isleaf(v)$ that checks if a node v is a leaf; $parent(v)$ that returns the parent of a node v; $child(v, t)$ that returns the t -th child of a node v; $childrank(v)$ that returns the $rank$ of a node v among its siblings; $fchild(v)$ and $lchild(v)$ that return the first/last child of a node v; and, $nsibling(v)$ and $psibling(v)$ that return the next/previous sibling of a node v.

The space cost is $2n + 1$ bits. If the bitvector uses constant time $rank$ and $select$, the time cost for all operations is constant (Clark, 1997), but using $3.6n$ additional bits. If compressed counterparts are used, $2.65n$ of extra bits are added only at expense of speed.

The cardinal variation of LOUDS is designed to represent cardinal trees, where each node has a fixed set of child types (σ in total), and a node might have a child of each type. This variation represents each node using σ bits in the bitvector. The k -th bit indicates whether the node has a child of type k (1) or not (0). For example, in a binary tree with $\sigma = 2$, a node with two children is 11, a node with only a left one is 10, a node with only a right one is 01, and a leaf is 00.

LOUDS supports operations of ordinal LOUDS plus $labeledchild(v, l)$ that returns the child of node v with label l, if it exists (equivalent to $child(v, t)$ in cardinal trees); $childrenlabeled(v, l)$ that checks if the child of node v with label l exists (returns $true$ or $false$); and $childlabel(v)$ that returns the label of the edge leading to node v.

The space cost of cardinal LOUDS is σn bits. The time cost of most operations is $O(c)$, assuming constant-time $rank$ and $select$ on the bitvector. However, for large values of σ, the space overhead can be significant. In such cases, compressed representations of the bitvector can be used to reduce the space to $n\ \lg(\sigma) + O(n)$ bits, but this may increase the time of some operations to $O\ (\lg(\sigma))$.

Balanced Parenthesis (BP)

The Balanced Parenthesis (BP) representation (Jacobson, 1989) is designed to encode ordinal trees using a balanced sequence of parentheses. Each node is represented by a pair of matching parentheses, with the opening parenthesis marking the node's first visit in a depth-first traversal and the closing parenthesis marking the completion. This representation captures the hierarchical structure of the tree, where the nesting of parentheses corresponds to the parent-child relationships.

BP is implemented using a bitvector, where 1 is an opening parenthesis and 0 is a closing one. The bitvector is preprocessed to support parenthesis queries, such as $close$ (where an opening parenthesis closes), $open$ (where an closing parenthesis opens), $enclose$ (index of the parent node), $fwdsearch$ (next index of a given value), $bwdsearch$ (previous index of a given value), rmq (range minimum query – used to find the position of the minimum value within a given range), rMq (range maximum query – used to find the position of the maximum value within a given range),$mincount$ (it counts how many times the minimum occurs within a given range), and $minselect$ (position of a given minimum), which are used to navigate and query the tree structure efficiently.

BP additionally supports navigation operations ($root$, $fchild$, $lchild$, $nsibling$, $psibling$, $parent$, $ancestor$); structure operations ($isleaf$, $nodemap$, $nodeselect$, $preorder$, $preorderselect$, $postorder$, $postorderselect$); depth- and subtree-related operations ($depth$, $subtree$, $isancestor$, $levelancestor$); leaf-related operations ($leafnum$, $leafrank$, $leafselect$); and the hierarchy operator lca (lowest common ancestor).

The space cost is $2n + o(n)$ bits (Munro & Raman, 2001), where n is the number of nodes in the tree. The time cost for all the operations is $O(\lg(n))$.

Depth-First Unary Degree Sequence (DFUDS)

The Depth-First Unary Degree Sequence (DFUDS) representation (Benoit et al., 2005), similarly to ordinal LOUDS, is designed to encode the topology of an ordinal tree. However, it uses a depth-first unary degree sequence. Each node is represented by a sequence of $1s$ (indicating the number of children) followed by a 0 . This sequence is concatenated in depth-first order to form a bitvector. The key property of DFUDS is that all nodes in a subtree are contiguous in the bitvector, and the net excess (Benoit et al., 2005; Jacobson, 1989) within any subtree is -1 .

The first three bits of the bitvector are set to 110 , and then the tree is traversed in preorder, appending the unary degree sequence of each node to the bitvector. The bitvector results of $n + 1$ $0s$ (one 0 per node, plus the initial one), and $n + 1$ $1s$ (one 1 per edge – $n - 1$ – plus the two initial ones). Therefore, the bitvector length is $2n + 1$ bits.

DFUDS supports navigation operations ($root$, $fchild$, $lchild$, $nsibling$, $psibling$, $parent$); structure-related operations ($isleaf$, $nodemap$, $nodeselect$, $preorder$, $preorderselect$); subtree-related operations ($subtree$, $isancestor$); leaf-related operations ($leafnum$, $leafrank$, $leafselect$); and the lowest common ancestor operator (lca).

The space cost is $2n + 1$ bits. Time of most operations is $O(\lg(n))$, assuming constant-time operations on balanced parentheses. However, $child$ and $children$ can be performed in constant time, making DFUDS particularly efficient for navigating towards specific children.

k²-Tree

The k^2-tree (Brisaboa et al., 2014) is designed to represent clustered graphs, where nodes can be divided into subsets of as many edges as possible. It recursively partitions the adjacency matrix of the graph into k^2 submatrices, representing it as a k^2-ary tree. Empty submatrices are not represented, leading to space savings. The tree structure is encoded as cardinal LOUDS bitvector. Each internal node in the tree is represented by k^2 bits, indicating the presence or absence of its children. Leaf nodes are either 1 (representing an edge) or 0 (no edge). The bitvector is preprocessed to support $rank$ and $select$, enabling efficient navigation and querying.

The choice of the order parameter, k, involves a trade-off between space efficiency and query performance. A higher k typically leads to better space efficiency, as it results in a shallower tree with fewer internal nodes. However, it also increases the fan-out of each node, potentially increasing the time required to traverse the tree during queries.

For smaller matrices with side size $s = 2^m$, where $m \in \mathbb{Z}$, a smaller k value ($k = 2^i$, with $i \in \mathbb{Z}$ and $i = 1$) might be more suitable, as the space savings from a higher k may be insignificant compared to the increased query time. Conversely, for larger matrices, with side size of the form $s = 4^m$, where $m \in \mathbb{Z}$, a higher k value ($k = 2^i$, with $i > 1$) could be advantageous. The space savings from a shallower tree can become more significant as the matrix size grows, outweighing the potential increase in query time. Moreover, if a dataset is clustered, a higher k value can effectively capture the clustered structure, leading to further space savings and potentially faster query times due to reduced tree traversal.The k^2-tree supports $adj(G, v, u)$ that checks if there is an edge between nodes v and u; $neigh(G, v)$ that returns the list of neighbors of node v; and $rneigh(G, v)$ that returns the list of reverse neighbors of node v.

The space cost is influenced by different factors, such as number of nodes and edges, distinct paths and nodes, representation of the bitvector, and grid size. For instance, using a sparse bitvector, the space could result of $O\left(e \lg\left(\frac{n^2}{e}\right) + e \lg(k)\right)$ bits, where n is the number of nodes, e is the number of edges, and k is the tree order. In practice, the space is often much lower for clustered graphs due to unrepresented empty submatrices. For that reason, several studies have explored the alternative of graph clustering and partitioning (Hernández & Navarro, 2014; Chierichetti et al., 2009; Maserrat & Pei, 2010; Boldi et al., 2011; Claude & Ladra, 2011; Grabowski & Bieniecki, 2014).

The time cost is $O(\lg_k(n))$ for $adj(G, v, u)$, $O(n)$ for $neigh(G, v)$, and $O(n)$ for $rneigh(G, v)$. The $O(n)$ time cost for $neigh$ and $rneigh$ is a worst-case bound.

k^n-Tree

The k^n-tree (de Bernardo et al., 2013; de Bernardo, 2014) is designed to represent n-ary relations encoded in n-dimensional matrix (hypermatrix). It extends the k^2-trees to higher dimensions by recursively partitioning a hypermatrix into k^n equal-sized sub-hypermatrices. This hierarchical partitioning allows for efficient representation of clustered data, where most of the $1s$ are concentrated in a few partitions. The tree is encoded using a cardinal LOUDS bitvector, following the same principles of the k^2-tree. However, each internal node is represented by k^n bits.

The k^n-tree supports $checkcell(C)$ that checks if the cell at coordinate C contains a 1; and $range(S, E)$ that reports all the cells within the range defined by starting coordinate S and ending coordinate E that contain a 1.

Similarly to k^2-tree, the k^n-tree space cost is influenced by different factors (number of nodes and edges, distinct paths and nodes, representation of the bitvector, and grid size). For instance, using a sparse bitvector, the space is $O\left(r \lg\left(\frac{s^n}{r}\right) + n\, r \lg(k)\right)$ bits, where r is the number of relations ($1s$ in the hypermatrix), s is the standardized size of the hypermatrix ($s = 2^{\lceil \lg(s_{max}) \rceil}$, with s_{max} as the largest dimension), and k is the tree order. The time cost of $checkcell(C)$ is $O(h) = O(\log_k(s))$. For $range(S, E)$, it depends on the hypermatrix distribution. In the worst case, it can be as high as $O(s^n)$, but for clustered data it can be much lower, as the k^n-tree can efficiently skip empty partitions.

Graphs

The representation of general graphs provides efficient support for various operations while minimizing space. Two approaches are proposed. The first approach represents each row of the graph's adjacency matrix as a sparse bitvector (Navarro, 2016) that optimizes space usage, especially for graphs with fewer edges. The other approach uses a permutation-based sequence (Claude & Navarro, 2011) for storing the matrix's adjacency lists as a concatenation of each node's neighbor list into a single sequence.

When using sparse bitvectors, 1 indicates the presence of an edge and 0 indicates its absence. Operations like adj (checking for an edge), $neigh$ (retrieving neighbors), and $outdegree$ (counting outgoing edges) are implemented using bit-vector $access$, $rank$, and $select$. When using sequence, a bitvector is used to mark the starting position of each node's neighbor list within the sequence. This allows for efficient $access$ on the sequence, and $rank$ and $select$ on the bitvector. Additionally, reverse neighbors and indegree can be computed efficiently using $rank$ and $select$ on the sequence.

Both representations support $adj(G, v, u)$ that checks if there is an edge from node v to node u; $neigh(G, v)$ that returns the list of neighbors of node v; and outdegree(G, v) that returns the outdegree of node v. The sequence representation additionally supports $rneigh(G, v)$ that returns the list of reverse neighbors of node v; and indegree(G, v) that returns the indegree of node v.

The space of a bitvector representation is $e \lg\left(\frac{n^2}{e}\right) + O(e)$ bits (worst-case entropy for directed graphs), and the time complexity of adj is $O(\lg(n))$, $O(c)$ for $neigh$, $O(\lg(n))$ for outdegree. The operation $rneigh$ is not space-efficiently supported.

For a sequence, the space is $e \lg(n) \left(1 + o(1)\right) + O(n)$ bits. The time of adj is $O(\lg(\lg(n)))$, $O(\lg(\lg(n)))$ for $neigh$, $O(c)$ for outdegree,$O(c)$ for $rneigh$, and $O(\lg(\lg(n)))$ for indegree.

Note that the time costs for sequence assume constant-time $rank$ and $select$ on the bitvector. In practice, these operations can be implemented efficiently, resulting in fast graph operations.

Findings

Compact data structures efficiently handle various data types. Fixed-size arrays suit numeric datasets with narrow ranges, while variable-size arrays cater to wider ranges. By combining these structures, one can efficiently encode n-ary relations like geographic data (latitude, longitude, and altitude) or customer purchases (customer ID, product ID, and purchase amount).

Bitvectors serve as fundamental building blocks for compact data structures and can encode binary relations, such as friendships in a social network. Wavelet-trees represent n-ary relations as strings over a larger alphabet, enabling efficient queries but potentially impacting space efficiency if the alphabet becomes too large (Barbay et al., 2014). The k^n-tree is a promising structure for representing n-ary relations in GIS, recommender systems, and RDF data. However, scalability concerns exist regarding its exponential space growth with dimensionality (de Bernardo et al., 2013; de Bernardo, 2014; Navarro, 2016), especially for non-clustered data.

To validate scalability concerns, experiments were conducted to compare the time and space performance of a specific query on synthetic n-ary datasets represented as k^n-trees against plain representations of adjacency lists as baseline. Synthetic datasets were chosen for their controlled nature, allowing systematic variation of parameters to isolate the effects of individual factors on scalability.

Multiple datasets were generated with varying sizes ($s \in [12,32,64,128]$), dimensions ($n \in [2,3,4]$; while 2-ary relations are common, many applications involve 3-ary or higher-order relations), and densities ($d \in [0.125,0.25,0.5,0.75]$; this wide range of sparsity levels helps to reflect the variability of real-world datasets), distributed randomly (to simulate scenarios where relationships between entities are uniformly distributed), randomly along the main diagonal, and in clusters (it helps to simulate scenarios where relationships are grouped together in specific regions with $c \in [1,2,4,8]$). Those parameter ranges were chosen to simulate various real-world scenarios and to stress-test the k^n-tree under different conditions. A systematic variation of them, with two samples per variation, yielded 576 different samples. Experiments were conducted on pairs of samples with equivalent size, dimension, and order $k \in [2,4]$, resulting in 6,912 baseline and 10,368 k^n-tree experiments.

To measure the scalability, the set operator intersection was used as benchmark, since it is a relevant operation that requires traversing the entire data, thus revealing advantages and disadvantages in terms of space and time efficiency of each type of encoding. This choice aligns with the work of Quijada-Fuentes et al. (2019), where the authors focused on set operations to evaluate the performance and scalability of k^2-trees on 2-ary relations.

The intersection algorithm for k^n-tree is shown in Algorithm 1. The algorithm receives two k^n-trees A and B, along with other related parameters, and returns a k^n-tree C as the intersection between A and B. The algorithm recursively traverses the trees A and B level by level, comparing corresponding bits in the bitmaps. For each level, it iterates through the potential k^n child positions, recursively finding the intersection of children if they exist. If there are no more levels, it directly compares leaves (actual relation values). Pointers p_A and p_B are used to navigate the bitmaps, skipping over non-existent children (empty sub-hypermatrices). The results are accumulated in a bitmap C representing the intersection at each level. Ultimately, the algorithm returns whether t is not empty (negation of the flag ε) used to set bits in earlier recursion levels as the algorithm returns.

Algorithm 1 *Set Intersection for k^n-Tree*

Inputs: k^n-tree bitmaps A and B; p_A is an array of pointers to bits in A, one per level of A; p_B is an array of pointers to bits in B, one per level of B; C is an array of h bitmaps that represents the intersection between A and B at each level; constants k and h related to the input k^n-trees; l is the current visited level in both k^n-trees A and B; A and B have the same height h; A and B have the same size s; A and B have the same order k.

Output: k^n-tree C as the intersection of A and B.

```
Function intersection($A, B, p\_A, p\_B, C, k, h, l$)
    $t \leftarrow \emptyset$ // Variable $t$ is a bitmap that keeps a result for the current level $l$.
    $\varepsilon \leftarrow 1$ // $\varepsilon$ is a flag to mean that $t$ does not contain any 1.
    For $i \in [0, k^n - 1]$ do
        If $l < h$
            If $A_{p_{A(l-1)}} \wedge B_{p_{B(l-1)}}$ then
                $t_i \leftarrow$ intersection($A, B, p_A, p_B, C, k, h, l + 1$)
            Else
                $t_i \leftarrow 0$
                skip_node($A, p_A, l + 1, A_{p_A(l-1)}, k^n, h$)
                skip_node($B, p_B, l + 1, B_{p_B(l-1)}, k^n, h$)
        Else
            $t_i \leftarrow A_{p_A(l-1)} \wedge B_{p_B(l-1)}$
        $\varepsilon \leftarrow \varepsilon \wedge \neg t_i$
        $p_A(l-1) \leftarrow p_A(l-1) + 1$
        $p_B(l-1) \leftarrow p_B(l-1) + 1$
    If $\neg\varepsilon$ then
        $C_{l-1} \leftarrow C_{l-1} \parallel t$ // Concatenate $C_{l-1}$ with bitmap $t$.
    Return $\neg\varepsilon$
```

The skip-node function used in Algorithm 1 helps skip subtrees of a k^n-tree (represented as a bitmap) that are not part of an intersection. If a node in the tree has children, the function skips over the parts of the bitmap representing those children using rank1 function to count $1s$ within a range of the bitmap to determine if there are any existing children.

The set intersection algorithm for adjacency lists is shown in Algorithm 2. It finds the intersection of two sorted adjacency lists A and B. It iterates through both lists simultaneously, comparing the current elements from each list. If they match, the element is added to the result; if not, the algorithm advances the pointer of the list with the smaller element. This continues until one of the lists is exhausted, ensuring that all common elements are identified and returned.

Algorithm 2 *Set Intersection for Adjacency Lists*

Inputs: k^n-tree bitmaps A and B; p_A is an array of pointers to bits in A, one per level of A; p_B is an array of pointers to bits in B, one per level of B; C is an array of h bitmaps that represents the intersection between A and B at each level; constants k and h related to the input k^n-trees; l is the current visited level in both k^n-trees A and B; A and B have the same height h; A and B have the same size s; A and B have the same order k.

Output: k^n-tree C as the intersection of A and B.

```
Function intersection($(A, B)$)
    Let $C$ the result, initially empty.
    $\alpha \leftarrow \beta \leftarrow True$
    Loop
        If $\alpha \wedge$ has_more($(A)$) then
            $a \leftarrow$ next($(A)$) // next returns the next n-dimensional tuple.
            $\alpha \leftarrow False$
        If $\beta \wedge has\_more(B)$ then
            $b \leftarrow$ next($(B)$) // next returns the next n-dimensional tuple.
            $\beta \leftarrow False$
        If $\neg\alpha \wedge \neg\beta$ then
            $\alpha \leftarrow True$ if $a < b \vee a = b$ else $False$
            $\beta \leftarrow True$ if $a > b \vee a = b$ else $False$
            If $a = b$ then $C \leftarrow C \parallel a$
        Else Break
    Return $C$
```

Results

Figures 1, 2, and 3 present summarized results for $s \in [16{,}64]$, which are compatible with $k = 4$. Results for $s \in [32{,}128]$ (not compatible with $k = 4$) follow the same trend.

Figure 1a shows that the k^n-tree consistently outperformed the baseline approach in terms of execution time for the intersection operation. This advantage was particularly pronounced for higher dimensional datasets ($n > 2$) and denser datasets (Figure 3a). On average, the k^n-tree performed eight (8) times faster than the baseline.

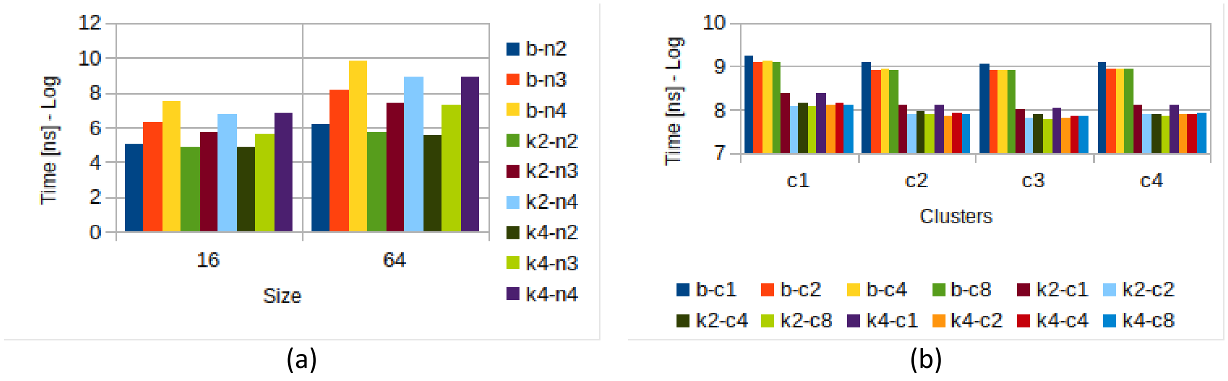
Regarding memory, Figure 2a demonstrated a superior memory scalability of k^n-tree compared to the baseline. This was evident in the lower memory usage of its main data structure across various dataset sizes, dimensions, and densities (Figure 3b). On average, the k^n-tree consumed 35 times less memory than the baseline.

The distribution of data points significantly affected the performance of both the k^n-tree and the baseline. Clustered data distributions were most favorable for the k^n-tree (as expected), leading to the best performance in terms of both time (Figure 1b) and memory usage (Figure 2b). However, even with scattered data distributions, the k^n-tree with $k = 4$ often outperformed the baseline.

Discussion

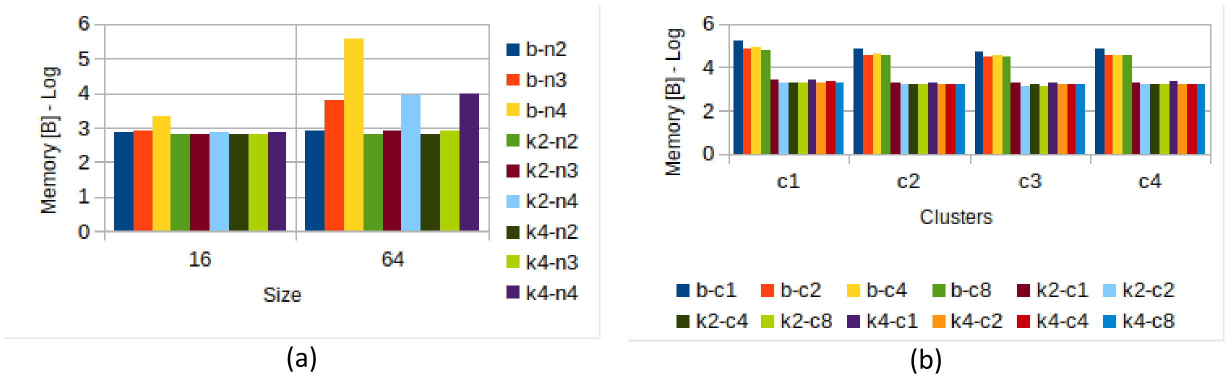
The experimental results align with previous studies (Brisaboa et al., 2014; de Bernardo et al., 2013; de Bernardo, 2014), highlighting the impact of dimensionality and data distribution on k^n-tree performance. Memory usage notably increased with dimensionality, particularly for denser datasets. However, the k^n-tree consistently outperformed the baseline in terms of both time and memory efficiency for higher dimensions, especially with clustered data ($c > 1$). This underscores the k^n-tree's effectiveness in compressing and navigating clustered data, even in high-dimensional spaces. This observation is aligned with the findings of Quijada-Fuentes *et al.* (2019) who observed that for k^2-trees.

Figure 1 *Comparison of (a) Time to Size, and (b) Time to Clusters.*



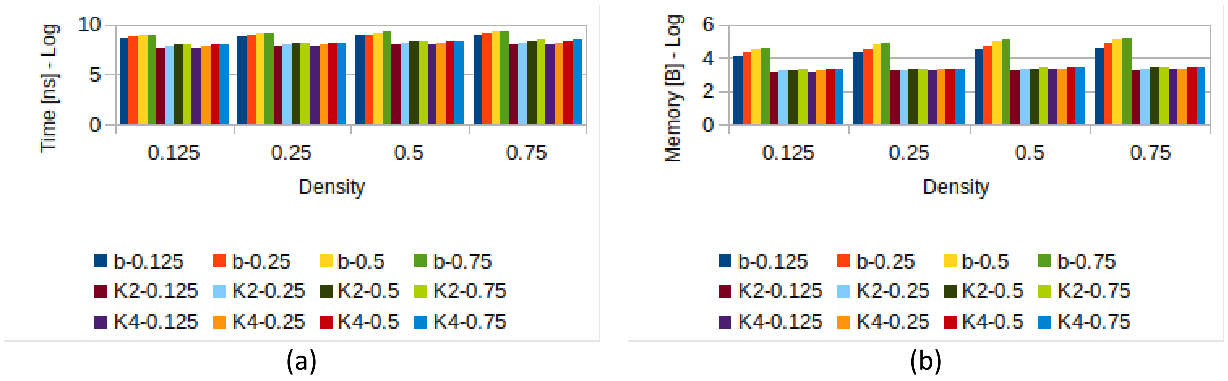
Note. Results in logarithmic scale. Abbreviations: b for baseline; ni for $n = i$; ki for k^n-tree with $k = i$; and ci for $c = i$.

Figure 2 *Comparison of (a) Memory to Size, and (b) Memory to Clusters.*



Note. Results in logarithmic scale. Abbreviations: b for baseline; ni for $\mathrm{n} = \mathrm{i}$; ki for k^{n}-tree with $\mathrm{k} = \mathrm{i}$; and ci for $\mathrm{c} = \mathrm{i}$.

Figure 3 *Comparison of (a) Time to Density, and (b) Memory to Density.*



Note. Results in logarithmic scale. Abbreviations: b for baseline; $\mathrm{ki} - \mathrm{j}$ for k^{n}-tree with $\mathrm{k} = \mathrm{i}$ and $\mathrm{d} = \mathrm{j}$.

Experiments confirmed the trade-off between space and time efficiency with varying order parameter (k) values. While a higher order ($k > 2$) generally led to better space efficiency, it wasn't always optimal. For lower dimensional and sparser datasets, $k = 2$ proved more time and space efficient. However, for clustered and denser data, $k = 4$ was superior. This aligns with findings by Quijada-Fuentes et al. (2019), where k^2-trees with compression of 1s (akin to higher k values) were more efficient for denser datasets, while the original k^2-trees (akin to lower k values) excelled with sparser data.

Conclusion

The empirical evidence from our experiments demonstrated the k^n-tree's effectiveness in handling set intersection operations on high-dimensional or clustered data, showcasing an average of eight (8) times faster execution and 35 times less memory consumption compared to the plain representation of adjacency lists as baseline. These findings highlight the importance of considering dataset characteristics and application requirements when selecting compact data structures. The superior scalability of the k^n-tree for high-dimensional and clustered data suggests its potential for efficient data management in various domains. For instance, in Geographic Information Systems (GIS) dealing with multi-dimensional spatial data or recommender systems modelling user-item interactions as high-dimensional relations, the k^n-tree could offer significant advantages in storage and query processing.

Furthermore, this research confirms the impact of the order parameter k on k^n-tree performance as mentioned by Navarro (2016). Lower k values are more efficient for lower-dimensional or sparser datasets, while higher k values are better suited for clustered and denser data.

However, this study is limited to experiments on synthetic datasets, which may not fully capture the complexities and nuances of real-world data. Additionally, it is focused solely on the set intersection operation, leaving the performance of the k^n-tree for other set operations unexplored. Finally, the range of parameters explored in the experiments was limited.

Future work should evaluate the k^n-tree on real-world datasets, explore its performance for other set operations, and conduct a more comprehensive parameter analysis. Additionally, comparing the k^n-tree with alternative compact data structures (Benoit et al., 2005; Delpratt et al., 2006; Quijada-Fuentes et al., 2019) for n-ary relations would provide a more complete understanding of its strengths and weaknesses. Furthermore, investigating the impact of different clustering algorithms on k^n-tree performance could further enhance its real-world applicability.

References

Barbay, J., Claude, F., Gagie, T., Navarro, G., & Nekrich, Y. (2014). Efficient fully-compressed sequence representations. *Algorithmica*, *69*(1), 232–268. https://doi.org/10.1007/s00453-012-9726-3

Barbay, J., & Navarro, G. (2013). On compressing permutations and adaptive sorting. *Theoretical Computer Science*, *513*, 109–123. https://doi.org/10.1016/j.tcs.2013.10.019

Benoit, D., Demaine, E. D., Munro, J. I., Raman, R., Raman, V., & Rao, S. S. (2005). Representing trees of higher degree. *Algorithmica*, *43*, 275–292. https://doi.org/10.1007/s00453-004-1146-6

Boldi, P., Rosa, M., Santini, M., & Vigna, S. (2011, March). Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web* (pp. 587–596). https://doi.org/10.1145/1963405.1963488

Brisaboa, N. R., Ladra, S., & Navarro, G. (2014). Compact representation of web graphs with extended functionality. *Information Systems*, *39*, 152–174. https://doi.org/10.1016/j.is.2013.08.003

Chierichetti, F., Kumar, R., Lattanzi, S., Mitzenmacher, M., Panconesi, A., & Raghavan, P. (2009, June). On compressing social networks. In *Proceedings of the 15th ACM SIGKDD International Conference On Knowledge Discovery and Data Mining* (pp. 219–228). https://doi.org/10.1145/1557019.1557049

Clark, D. (1997). *Compact pat trees* [Doctoral dissertation, The University of Waterloo, Canada]. https://uwspace.uwaterloo.ca/bitstream/handle/10012/64/nq21335.pdf

Claude, F., & Ladra, S. (2011, October). Practical representations for web and social graphs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (pp. 1185–1190). https://doi.org/10.1145/2063576.2063747

Claude, F., & Navarro, G. (2011). Self-indexed grammar-based compression. *Fundamenta Informaticae*, *111*(3), 313–337. https://dl.acm.org/doi/10.5555/2361502.2361504

de Bernardo, G. (2014). *New data structures and algorithms for the efficient management of large spatial datasets* [Doctoral dissertation, Universidade da Coruña, Spain]. http://hdl.handle.net/2183/13769

De Bernardo, G., Álvarez-García, S., Brisaboa, N. R., Navarro, G., & Pedreira, O. (2013). Compact querieable representations of raster data. In O. Kurland, M. Lewenstein, & Porat, E. (eds.), *String processing and information retrieval*. SPIRE 2013. Lecture Notes in Computer Science, vol 8214. Springer, Cham. https://doi.org/10.1007/978-3-319-02432-5_14

Delpratt, O. N., Rahman, N., & Raman, R. (2006). Engineering the LOUDS succinct tree representation. In C. Àlvarez & M. Serna (eds.), *Experimental algorithms*. WEA 2006. Lecture Notes in Computer Science, vol 4007. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11764298_12

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, *21*(2), 194–203. https://doi.org/10.1109/TIT.1975.1055349

Ferragina, P., & Venturini, R. (2007). A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, *372*(1), 115–121. https://doi.org/10.1016/j.tcs.2006.12.012

Golynski, A., Munro, J. I., & Rao, S. S. (2006, January). Rank/select operations on large alphabets: A tool for text indexing. In *SODA* (Vol. 6, pp. 368–373). In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA 2006, Miami, Florida, USA.

Golynski, A., Raman, R., & Rao, S. S. (2008, July). On the redundancy of succinct data structures. In J. Gudmundsson (eds.) *Algorithm theory* – SWAT 2008. SWAT 2008. Lecture Notes in Computer Science, vol 5124. Springer, Berlin, Heidelberg. *Scandinavian Workshop on Algorithm Theory* (pp. 148–159). Springer. https://doi.org/10.1007/978-3-540-69903-3_15

Grabowski, S., & Bieniecki, W. (2014). Tight and simple web graph compression for forward and reverse neighbor queries. *Discrete Applied Mathematics*, *163*, 298–306. https://doi.org/10.1016/j.dam.2013.05.028

Grossi, R., Gupta, A., & Vitter, J. S. (2003). High-order entropy-compressed text indexes. In *Proceedings of SODA '03: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. https://dl.acm.org/doi/10.5555/644108.644250

Grossi, R., Orlandi, A., & Raman, R. (2010). Optimal trade-offs for succinct string indexes. In *Automata, Languages and Programming: 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I 37* (pp. 678–689). Springer. https://doi.org/10.1007/978-3-642-14165-2_57

Hernández, C., & Navarro, G. (2014). Compressed representations for web and social graphs. *Knowledge and Information Systems*, 40(2), 279–313. https://doi.org/10.1007/s10115-013-0648-4

Jacobson, G. (1989, October). Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science* (pp. 549–554). IEEE Computer Society. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=63533

Maserrat, H., & Pei, J. (2010, July). Neighbor query friendly compression of social networks. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 533–542). https://doi.org/10.1145/1835804.1835873

Munro, J. I., & Raman, V. (2001). Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, *31*(3), 762–776. https://doi.org/10.1137/S0097539799364092

Navarro, G. (2016). *Compact data structures: A practical approach*. Cambridge University Press.

Pagh, R. (2001). Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, *31*(2), 353–363. https://doi.org/10.1137/S0097539700369909

Quijada-Fuentes, C., Penabad, M. R., Ladra, S., & Gutiérrez, G. (2019). Set operations over compressed binary relations. *Information Systems*, *80*, 76–90. https://doi.org/10.1016/j.is.2018.10.001

Raman, R., Raman, V., & Satti, S. R. (2007). Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, *3*(4), 43–es. https://doi.org/10.1145/1290672.1290680