BIT: Using Bitmap Index to Speed Up NCBI Taxonomy Computing

Chuan Hu
huchuan@cnic.cn
Computer Network Information Center, Chinese Academy of Science
University of Chinese Academy of Sciences
Beijing, China

Jiawei Cai
jwcai@cnic.cn
Computer Network Information Center, Chinese Academy of Science
University of Chinese Academy of Sciences
Beijing, China

Zihao Zhao*
airzihao@gmail.com
Huawei Technology Company
Beijing, China

Zhihong Shen†
bluejoe@cnic.cn
Computer Network Information Center, Chinese Academy of Science
Beijing, China

ABSTRACT

The National Center for Biotechnology Information (NCBI) Taxonomy is extensively used in biomedical and ecological research. Typical demands include computing the lowest common ancestor, determining descendant relationships, and listing the descendants of a node. However, existing tools often suffer from inefficient runtime performance. To address this challenge, our paper introduces a novel indexing method, BIT, designed specifically for tree-like data. BIT first encodes the tree-like structure into a bit-vector using the Polychotomic encoding algorithm, subsequently storing the bit-vector in a bitmap. By employing parallel bit operations, BIT significantly accelerates the speed of typical computational tasks. Experimental results on public datasets demonstrate that BIT outperforms baseline systems in task execution performance.

CCS CONCEPTS

• **Applied computing → Bioinformatics**.

KEYWORDS

NCBI Taxonomy, Bitmap Index, Tree Data.

ACM Reference Format:
Chuan Hu, Jiawei Cai, Zihao Zhao, and Zhihong Shen. 2024. BIT: Using Bitmap Index to Speed Up NCBI Taxonomy Computing. In *36th International Conference on Scientific and Statistical Database Management (SSDBM 2024), July 10–12, 2024, Rennes, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3676288.3676294

*Zihao Zhao finished his work at Computer Network Information Center, Chinese Academy of Science.
†Zhihong Shen is the corresponding author of this paper.

1 INTRODUCTION

Taxonomy, the science of classification, plays a fundamental role in the fields of biology and ecology by providing a structured way to name and organize the diversity of life. The National Center for Biotechnology Information (NCBI) Taxonomy [6, 20] is a critical resource that supports numerous scientific disciplines by offering a comprehensive database of taxonomic data [14–16, 26, 28].

In the database, all organisms are arranged into a structured taxonomy tree with a single root. Each node within this tree, referred to as a TaxNode, is assigned a specific taxonomy name, a unique identifier known as a TaxId, and a taxonomic rank. The entire lineage of any given taxonomy entry is represented as a sequence of TaxNodes linked through ancestor-descendant relationships within this taxonomy tree. Researchers rely on this taxonomy to perform various computational tasks, such as determining descendant relationships (**T1**), calculating the lowest common ancestor (**T2**), and listing the descendants associated with a given Taxonomy Identifier (TaxId) (**T3**).

There is a widespread need for tools to perform operations on NCBI taxonomies [1, 2, 12, 13, 19, 21], especially when dealing with large-scale data or complex queries. Computational inefficiency stems mainly from traditional data processing methods that are not optimised for the hierarchical and interconnected structure of taxonomic data.

Recognizing the need for more efficient computational methods in taxonomic research, our study introduces a novel approach to enhance data manipulation and retrieval tasks associated with the NCBI Taxonomy. This approach centers around a new indexing method specifically designed for tree-like data structures, which we have named BIT (**B**itmap **I**ndex for **T**axonomy).

As shown in Figure 1, BIT first maps tree nodes to bit-vectors using a tree bit-vector encoding algorithm (*Polychotomic Algorithm* [7]). The bit-vectors contain the gene information carried by each node. As shown in Figure 1.A, the algorithm consists of two phases: the reconstruction phase makes some structural adjustments to the tree, and the encoding phase assigns bit vectors to each node from top to bottom. The positions of 1 in the bit-vector of each node is called the genes of this node. The encoded bit-vectors and genes have the following characteristics: Each node's genes

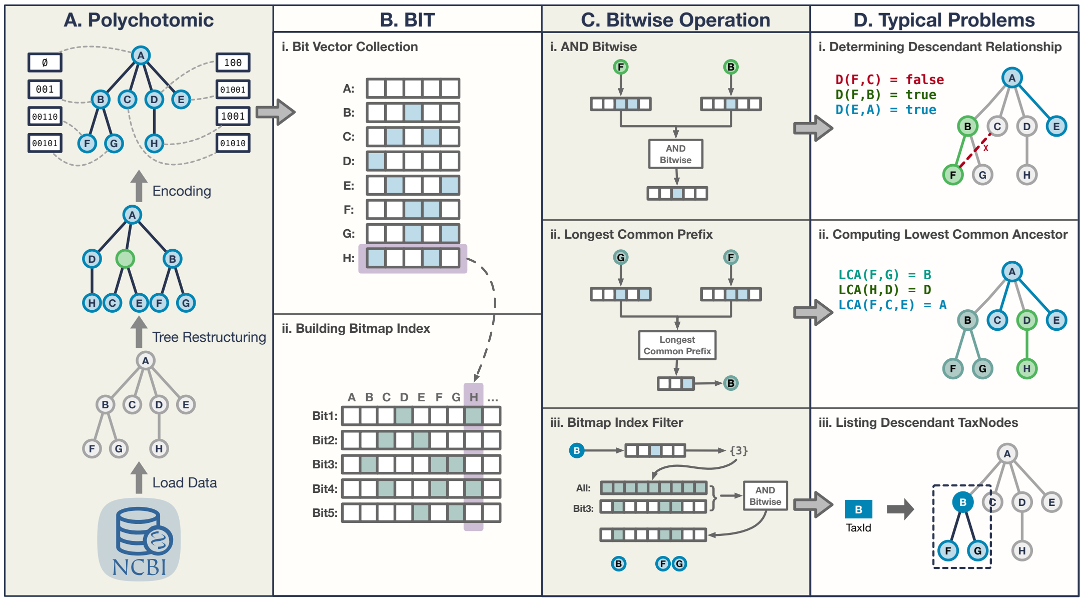

Figure 1: A. After reading the data from NCBI, it goes through two phases of the encoding algorithm (restructuring and encoding) to assign bit codes to each node in the tree. B. The BIT collects and constructs a series of bitmap indices from the encoded nodes and vectors. C. BIT accelerates computational tasks through bitwise operations rather than traditional tree traversal. D. Some example of typical problems on the taxonomy database: Determining Descendant Relationship, Computing LCA(lowest common ancestor) and Listing Descendants of Given TaxIds.

include the genes of all its ancestors. The details of the algorithm we describe in Section 3.

Subsequently, the bit-vectors of all nodes are stored in several bitmap indexes (Figure 1.B). With bit-vectors and bitmap indexes, typical tree queries, such as determining inheritance relationships and searching for common ancestors, can be completed through bitwise operations (Figure 1.C). These operations are much faster in computers than traversing. Additionally, the parallelism of bitwise operations and the compressibility of bitmap indexes make BIT more time and space-efficient than other methods such as creating a tree in memory.

We provide a comprehensive comparison of BIT's performance against standard baseline systems using a series of experiments conducted on public NCBI Taxonomy datasets. Our results demonstrate the superiority of BIT in executing essential taxonomic tasks, thereby underscoring its potential to transform taxonomic research and application across various scientific domains.

In summary, we make the following contributions:

- **Methods:** We have designed a method based on bitmap indexing. The method stores the bit-encoded tree in a bitmap. Based on the properties of bit encoding, the typical tree computation task is converted to vector and bitmap based computation task to speed up the computation performance.
- **Toolkit:** Based on the above approach, we have developed a open source tool to handle the tasks involved. The tool can easily read NCBI data for calculation. By deserialising the bitmap, the tool has faster loading speed and smaller memory cost.
- **Experiment** We evaluated our method using real datasets. The experimental results indicate that BIT significantly outperforms the related baseline tools in typical computational tasks.

We further describing related background and concepts in Section 2. We describe the detailed process of the *Polychotomic* encoding algorithm in Section 3. We introduce BIT methods in Section 4. We present the details of our tool implementation in Section 5. Section 6 demonstrates the experimental. We then discuss related work in Section 7 and conclude in Section 8.

2 BACKGROUND

2.1 NCBI Taxonomy

The NCBI Taxonomy [6] dataset is a comprehensive resource provided by the National Center for Biotechnology Information (NCBI) that encompasses the classification and nomenclature of organisms. Established in the early 1990s, the NCBI Taxonomy serves as a foundational framework for organizing and categorizing living organisms based on evolutionary relationships. The NCBI Taxonomy provides a hierarchical classification of organisms, ranging from the highest taxonomic ranks (e.g., domains and kingdoms) to species level. Each entry in the taxonomy database represents a taxonomic unit, identified by a unique Taxonomy Identifier (TaxId). TaxIds are used to query and retrieve taxonomic information from the dataset. The taxonomy dataset is organized as a hierarchical tree structure, with each node representing a taxonomic unit at a specific rank (e.g., genus, species). The dataset is available in various formats, including flat files and XML, to accommodate different data access and analysis needs.

The relevant terms are explained below:

- **Taxonomy Identifier (TaxId)**: A unique numerical identifier assigned to each taxonomic unit.
- **Scientific Name**: The formal scientific name of the organism.
- **Rank**: The taxonomic rank of the organism (e.g., species, genus, family).
- **Parent TaxId**: The TaxId of the immediate parent taxonomic unit.
- **Lineage**: The hierarchical path from the root of the taxonomy tree to the specific taxonomic unit.

2.2 Problem Setting

Formally, a tree can be viewed as a partially ordered set $T = (X, \preceq)$, where X is a set of nodes and $\preceq$ is an order relation (transitive, reflexive and anti-symmetric relation). $x \prec y$ means that x is the immediate child of y, y is the immediate parent of x. It is defined as

$$x \preceq y, x \neq y, \neg\exists z \mid x \preceq z \preceq y, x \neq z \neq y.$$

In the following we will formally describe three typical problems.

T1. Determining Descendant Relationship Let $D(x, y)$ denote the function that determines the descendant relationship between two taxonomic nodes x and y in a taxonomic hierarchy. The function returns true if x is a descendant of y, and false otherwise.

$$D(x,y) = \begin{cases} TRUE & x \preceq y \\ FALSE & otherwise \end{cases}$$

For example, as shown in Figure 1.D.i, node F is a descendant of node B and node E is a descendant of node A, hence $D(F, B) = $ *true*, $D(E, A) = $ *true*. and node F is not related to node C, hence $D(F, C) = $ *false*.

T2. Computing Lowest Common Ancestor In the realm of graph theory and computer science, the concept of the Lowest Common Ancestor (LCA), also known as the least common ancestor, pertains to determining the lowest, or deepest, node within a tree, denoted as T, that serves as a common ancestor to two given nodes. In this context, each node is considered to be a descendant of itself. Consequently, if there exists a direct path from one node to the other, the lower node along this path is identified as the lowest common ancestor. Suppose node p is the nearest common ancestor of nodes x and y:

$$x \preceq p, y \preceq p, \nexists z \mid x \preceq z \prec p, y \preceq z \prec p$$

T2 expands the number of nodes compared to the original definition of LCA. Given a set of taxonomic nodes $S = \{x_1, x_2, ..., x_n\}$ and a taxonomic tree represented as T, the Lowest Common Ancestor (LCA) task aims to find the lowest common ancestor of all nodes in S. Let $LCA(S)$ represent the function that computes the lowest common ancestor of the entities in S. As shown in Figure 1.D.ii, the LCA of node F and node G is their parent (node B). node D is the ancestor of node H, so their LCA is node D. The LCA of nodes F, C and E, are the root node (node A).

T3. Listing Descendant TaxNodes For a given taxonomic node x in a taxonomic tree represented as T, the task involves listing all descendant taxonomic nodes of x. Let $List(x)$ represent the function that lists all descendant taxonomic nodes of x in the taxonomic tree T.

$$List(x) = \{y \mid y \preceq x\}$$

For example, as shown in Figure 1.D.iii, the black dashed box shows all the descendants of node B (the dark blue node).

2.3 Bit-Vector Encoding of Tree

Bit-Vector Encoding is an efficient method for representing trees in a compact binary format, which allows for effective storage, retrieval, and manipulation of tree structures. This encoding technique assigns a unique binary sequence to each node in the tree, reflecting its position and relationship with other nodes [9].

Let $T = (X, \preceq)$ be a tree and $B_n = (2[n], \subseteq)$ be the Boolean lattice of size n (That is, the 01 vector space of length n). The tree T has a bit-vector encoding of size n if it can be embedded into B_n. In other words, a bit-vector encoding is a mapping ϕ between the elements of T and the elements of B_n, such that for any two elements x and y from X:

$$x \preceq y \Longleftrightarrow \phi(y) \subseteq \phi(x).$$

This encoding will assign, to each element x of T , a bit-vector V_x of size n, where its ith bit is set to one if i belongs to $\phi(x)$, and zero otherwise. In some encoding algorithms, it is common to call elements in the set $\phi(x)$ as **genes** of x. **In simple terms, if x is a descendant of y, then x contains all of y's genes**.

In a bit-vector encoding, the root of the tree is assigned the null set. Every other node in the hierarchy has some non-null subset of $\{1, ..., n\}$ (its gene) associated with that node. The encoding of a node is the union of the gene of the node with the encoding of its parents. Thus, the encoding of a node is the union of the gene of a node with genes of its ancestors. Such a coding scheme is called a hierarchical encoding. Bit-vector encodings have been a subject of much research [4, 7, 18].

2.4 Bitmap Index

A bitmap index is one of the most efficient indexing methods, using bitmaps to represent the presence or absence of a value in a table's column [25, 29]. It has shown excellent performance in applications like data management and data warehousing [27]. In a bitmap index, each bit corresponds to a value in a column: a bit set to 1 indicates the value is present, while 0 indicates its absence. When executing a query, the bitmap index quickly identifies rows that meet the query condition, enabling the database engine to eliminate non-matching rows rapidly. The remaining rows are processed to produce the final result set.

Bitmap indexes are particularly useful for databases with low cardinality, where the number of unique values in a column is relatively small compared to the total number of rows. Techniques such as binning [22, 24] and compression [3, 5, 8, 10] can make bitmap indexes suitable for high cardinality data. Bitmap compression results in highly compressed indexes, which require less disk space than other indexing methods, offering significant storage cost savings for large databases. Moreover, bitmap indexes are exceptionally fast for processing queries involving multiple columns due to their ability to perform logical operations (AND, OR, NOT) on the bitmaps to quickly identify the rows that meet the conditions [17].

3 POLYCHOTOMIC ENCODING ALGORITHM

Filman [7] presented *Polychotomic Encoding*, a quick algorithm for creating bit vector encoding. *Polychotomic* consists of two stages: **restructuring** and **encoding**. The purpose of restructuring the tree before applying the encoding algorithm is to reduce the number of bits required. Then encoding algorithm chotomic genes to the children in the transformed tree.

3.1 Tree Restructuring

For a node x with children $[a, b, ...]$, the algorithm first (recursively) computes the weight (number of bits needed to represent) of each child:

- Nodes with no children have weight 0;
- Nodes with one child, weight 1 more than that child;
- Nodes with two children weigh 2 more than the heavier child.

For a node with more than two children, the algorithm sorts the children by weight and selects the two "lightest" children, call them a (the lightest) and b (the second lightest). *Polychotomic* attempts to combine the two smallest children into a single node of weight two more than the heavier: It will constructs a new node y (of weight b + 2), changes the parentage of a and b to be y, and inserts y into the child-set of x in their place. But before combining, it checks to make sure that doing so would not create a new heaviest child. If it does not, it builds the new node and iterates. If it would, it stops joining children and uses a multi-bit, chotomic encoding for all the remaining children.

A set of genes are chotomic if no element of the set is a subset of another. The genes $\{1\}$ and $\{2\}$ (represented as the two bit bit-vectors [10] and [01]) are a two-element, two bit chotomic set; the genes $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$, and $\{3, 4\}$ ([1100], [1010], [1001], [0110], [0101], and [0011]) represent a set of six four-bit chotomic genes. The function C defines the number of different chotomic genes using n bits:

$$C(n) = \binom{n}{\lfloor n/2 \rfloor} = 2^{\lfloor n/2 \rfloor} \prod_{1 \le i \le \lceil n/2 \rceil} (2i-1)/i \tag{1}$$

The function sp [23] is the "inverse" of C—the number of bits needed to create different chotomic genes.

$$sp(n) = \min\{k \mid C(k) \ge n\} \tag{2}$$

Formally, the elements of a sequence S are $[x_1, \ldots, x_n]$, where the x_i are sorted. Thus, x_1 of a sequence is the smallest element of that sequence; the x_2 is the second smallest. The rightmost element of a sequence is the largest. The use of two ellipses (e.g., $[x_1, \ldots x_i \ldots x_n]$) indicates that we can't specify where an element goes in the sequence's sort. The cardinality of S is $|S|$. The Polychotomic Encoding weight function is

$$\mathcal{P}(S) = \begin{cases} 0 & |S| = 0 \\ 1 + x_1 & |S| = 1 \\ 2 + x_2 & |S| = 2 \\ \mathcal{P}([x_3, \ldots, x_2 + 2, \ldots, x_n]) & |S| > 2, x_2 + 2 \le x_n \\ x_n + sp(n) & |S| > 2, x_2 + 2 > x_n \end{cases} \tag{3}$$

Because $sp(0) = 0, sp(1) = 1, sp(2) = 2$, this function can be simplified to

$$\mathcal{P}(S) = \begin{cases} \mathcal{P}([x_3, \ldots, x_2 + 2, \ldots, x_n]) & |S| > 2, x_2 + 2 \le x_n \\ x_n + sp(n) & otherwise \end{cases}$$

Next, combining Figure 2 as an example, the *Case* mentioned in the figure and the following text refers to which of the five cases from Function 3 the calculation corresponds to. The entire calculation process is bottom-up:

(1) Initially, as shown in the first sub-figure of Figure 2, since all leaf nodes have no children, we set the weight for all leaf nodes to 0 (**Case 1**).
(2) For the left sub-tree, node B has two children, thus fitting **Case 3**, with $P = 2 + x_2$. Here, x_2 represents the weight of the second smallest child, and since both children have a weight of 0, $x_2 = 0$, hence $P(B) = 2$.
(3) The node D has one child, corresponding to **Case 2**, thus $P(D) = 1 + x_1 = 1$.
(4) When calculating the weight for the root node A, the child nodes are first sorted by their weights.
(5) According to the algorithm, we first prioritize merging smaller nodes but without creating a new node with a greater weight. At this time, the smallest two node weights are $x_1 = 0$ and $x_2 = 0$, and the weight of the merged node would be $x_2 + 2 = 4$ (**Case 3**). The merge does not create a larger node ($x_2 + 2 \le x_4$), thus fitting **Case 4**, and the merge operation is performed.
(6) The weight of the root node is recalculated thereafter. Now there are three child nodes, sorted with weights $x_1 = 1, x_2 = 2, x_3 = 2$ (sorting is omitted in the figure). If the smallest two nodes are merged now, the new node's weight would be $x_2 + 2 = 4$, creating a new larger node. Thus, it does **not fit Case 4**, and no merging is performed. According to **Case 5** of the formula, $P(A)$ is calculated as $x_3 + sp(3) = 5$, where $sp(3) = 2$ can be calculated using Proposition 1.

3.2 Encoding

The encoding process involves gradually assigning bit vectors from the root node to the leaf nodes. As shown in Figure 3, this phase consists of the following steps, which we will exemplify in conjunction with Figure 3. After restructuring the tree, the gene of the root node is considered **empty(∅)**. Then, the following process is executed recursively, starting from the root node:

(1) Count the number of children n, and calculate the number of bits k required for creating n different chotomic genes by the $sp(n)$ function. For example, in Figure 3, the root node has three children, needing three bits to create the chotomic genes.
(2) Use the $code(k)$ function to create chotomic genes in k bits (the genes may be more than the number of children; create as needed). In Figure 3, the root node created 3 chotomic genes ([100],[010],[001]).
(3) **Append** these genes to the parent's gene and assign them to the children.

This phase is completed when all nodes from the top down have had their genes assigned. Genes are also assigned in this phase for

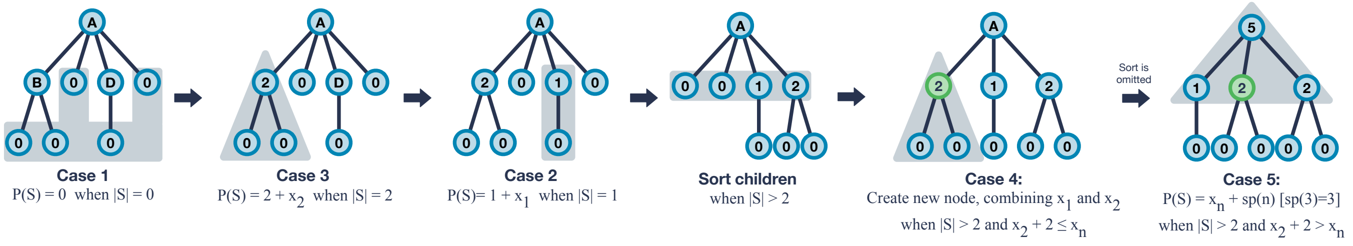
Case 1
P(S) = 0 when |S| = 0
Case 3
P(S) = 2 + x2 when |S| = 2
Case 2
P(S)= 1 + x1 when |S| = 1
Sort children
when |S| > 2
Case 4:
Create new node, combining x1 and x2
when |S| > 2 and x2 + 2 ≤ xn
Sort is omitted
Case 5:
P(S) = xn + sp(n) [sp(3)=3]
when |S| > 2 and x2 + 2 > xn

Figure 2: The process of restructuring tree in *Polychotomic*. The blue nodes in the figure indicate the native nodes in the tree. The green nodes represents the newly created nodes. The number in the node indicates its weight. The gray box indicates the process of calculating *weight*, and the *case* in the figure indicates cases in the Eq. 3.

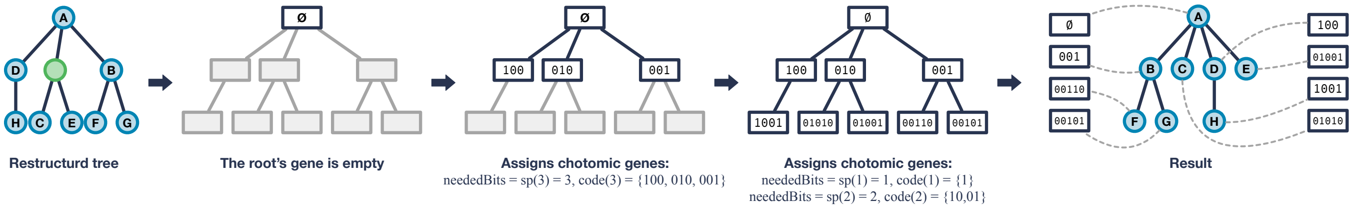
Restructurd tree
The root's gene is empty
Assigns chotomic genes:
neededBits = sp(3) = 3, code(3) = {100, 010, 001}
Assigns chotomic genes:
neededBits = sp(1) = 1, code(1) = {1}
neededBits = sp(2) = 2, code(2) = {10,01}
Result

Figure 3: The process of encoding in *Polychotomic*. The black 0/1 numbers indicate the newly assigned genes of this node based on the genes of the parent node.

nodes added in the previous phase, and these temporary nodes can be discarded when the tree is restored to its initial structure (as in the last sub-figure in Figure 3).

Algorithm 1 presents the pseudo-code of *Polychotomic*. According to the description in [7], in the worst case, for a "straight line" hierarchy composed single-child nodes, bit vector encoding needs one bit for each node except the root. Thus, the worst case space complexity of all bit vector encoding algorithms is proportional to$|T|^2$. In practice, the required space seems more on the scale of $|T|\log(|T|)$.

3.3 Implementation of Polychotomic

3.3.1 sp function and code function. The purpose of the *sp* function is to compute the number of bits required for n elements, that is, to calculate the smallest k such that $c(k) \geq n$. In *Polychotomic*, the *sp* function is computed through Proposition 1, which provides the upper and lower bounds for $sp(n)$ and constrains the range of values in two integers.

Proposition 1 (Habib [11]). *Let* $n \geq 2, c(n) = \binom{n}{\lfloor n/2 \rfloor}$ *and* $sp(n) = \min\{k \mid c(k) \geq n\}$ *.Then,* $sp(n) \in$

$$\left\{\left\lfloor \log_2(n) + \frac{\log_2(\log_2(n))}{2} + 1\right\rfloor, \left\lfloor \log_2(n) + \frac{\log_2(\log_2(n))}{2} + 2\right\rfloor\right\}.$$

However, in practical computations, this method is not fast, because the k involving many logarithmic calculations. Therefore, we have designed a new method for computing the *sp* function to accelerate the calculation process.

First, we create an array *spArr* such that $spArr[i] = c(i)$ for $i \geq 2$, and set $spArr[0] = 0$ and $spArr[1] = 1$ for $i = 0$ and $i = 1$. The elements of the array represent the number of different chotomic genes using the number of the element's index bits. Then, we just need to search this array to find the index of the element that is just greater than n. We can use binary search to speed up the search process. Based on our experience, setting the size of *spArr* to 30 is sufficient ($spArr[30] = 155, 117, 520$). According to our validation, this method is approximately 20 times faster than the method that calculates through Proposition 1.

The implementation of *code* function is also similar to this, using a precomputation approach to store the chotomic genes that can be created by k bits in $codeArr[k]$, thereby avoiding redundant calculations. Since *codeArr* needs more space, its size is set to 20 (if this is insufficient, larger values will be calculated and saved).

4 BITMAP INDEX FOR TREE

In this section, we will describe the process of designing a bitmap index system for tree data (BIT) using the algorithm outlined in Section 3 and introduce how to use BIT in some typical tree query tasks. Table 1 summarizes our notations used in this section.

4.1 Bitmap Index Design

Following the execution of the encoding algorithm, every node within the tree possesses a corresponding bit vector. Subsequently, we proceed to assemble a bitmap index utilizing these vectors. In this indexing process, each bitmap is associated with a single bit in the bit vectors, with each bit within the bitmap corresponding to a particular node. The value of each bit within the bitmap signifies whether the node possesses the corresponding value in its respective bit vector. Conceptually, this operation bears resemblance to the transpose of a matrix, wherein the rows and columns of the original matrix are interchanged.

Algorithm 1: Polychotomic Encoding Algorithm

```
Function polychotomic(root):
    Input: root, the root node of the tree.
    polyWeight(root);
    encoding(root,0);
end

Function polyWeight(node):
    Input: node, the tree node;
    Output: weight, the weight of this node;
    if node has no children then
        return 0;
    end
    for child in node's children do
        child.weight ← polyWeight(child);
    end
    sort children of node by weight in ascending order;
    S ← node.children.weight;
    n ← S.size;
    if n > 2 and S[1] + 2 leq S[n-1] then
        create node N;
        Move the first two children($c1$, $c2$) of the node to $N$;
        N.weight ← $c2$.weight + 2;
        insert $N$ into node's children;
        polyWeight(node);
    else
        return S[n-1] + sp(n);
    end
end

Function encoding(node, nextFreeBit):
    Input: x, tree node; nextFreeBit, the index of next free
           bit.
    neededBits ← sp(x.children.size);
    for child in x.children as c in code(nextFreeBit,
      neededBits) do
        child.gene ← c;
        encode(child, nextFreeBit + neededBits);
    end
end

Function sp(n):
    Input: n, the number of different chotomic genes.
    Output: k, the number of bit need.
    initialization: spArr ← [], spArr[0] ← 0, spArr[1] ← 1;
    for i from 2 until MAX do
        spArr[i] ← $\binom{i}{\lfloor i/2 \rfloor}$
    end
    Binary search k in spArr, such that spArr[k-1] < n ≤
      spArr[k];
end
```

Notation	Meaning
G_x	the genes set of node x
V_x	the bit-vector of node x
$V[i]$	the $i-th$ bit in bit-vector V
$\mathcal{V}$	the set of bit-vectors
B_i	the bitmap with index i
$B[j]$	the $j-th$ bit in bitmap B
$\mathcal{B}$	the set of bitmaps

Table 1: Notations

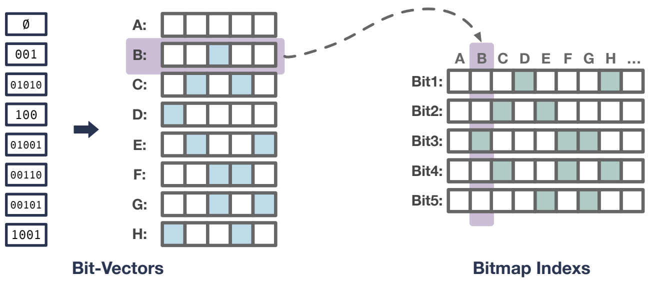

Figure 4: The diagram of the construction process of BIT.

If we have n tree nodes and each node's bit vector contains k bits: $\mathcal{V} = [V_1, V_2, ..., V_n]$. We create k bitmaps $\mathcal{B} = [B_1, B_2, ..., B_k]$, and the $j-th$ bit of the bitmap B_i represents the value of the $i-th$ bit in the vector of the $j-th$ tree node.

$$B_i[j] = V_j[i], \forall i, j \in \mathbb{Z}, 0 < i \leq k, 0 < j \leq n \tag{4}$$

As demonstrated in Figure 4, the node B's bit-vector $V_b = [00100]$, thus, the second bit of the third bitmap is true(1), $B_3[2] = 1$.

4.2 Query in BIT

This subsection will outline how to utilize bit-vector and bitmap index to speed up typical tree query tasks. By using these techniques, such tasks can be accomplished using bitwise operations, which are parallel executed quickly on computers.

4.2.1 Determining Descendant Relationship. We provided a brief introduction to this query in Section 2, and in this section, we will offer a more detailed description. Based on the encoding algorithm, if y is an descendant of x, then the genes of y must contain all of the genes of x(see in Section2.2).

$$y \leq x \iff G_x \subseteq G_y \tag{5}$$

And

$$G_x \subseteq G_y \iff G_x \cap G_y = G_x$$

When genes are represented as bit-vectors (V_x and V_y), the aforementioned subset intersection can be calculated using bitwise AND operations.

$$y \leq x \iff G_x \subseteq G_y \iff V_x \wedge V_y = V_x$$

We only need to do one bitwise AND operation and one comparisons on the vectors of the ancestor nodes to determine the inheritance relationship between them. For example, as shown in

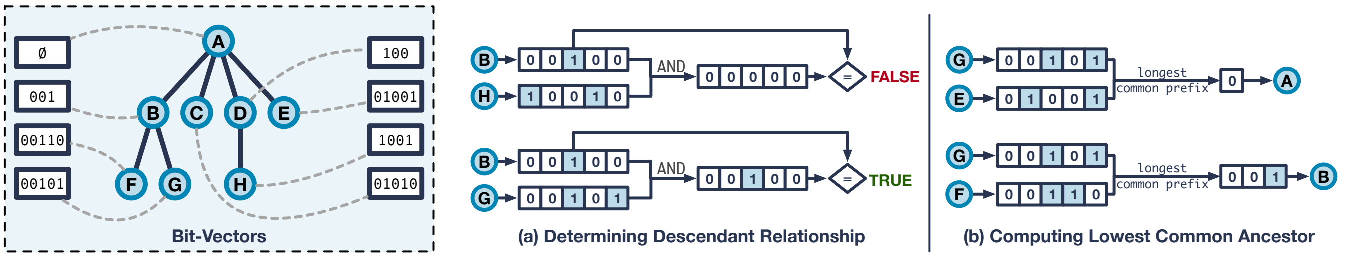
Bit-Vectors
(a) Determining Descendant Relationship
AND
FALSE
TRUE
(b) Computing Lowest Common Ancestor
longest common prefix

Figure 5: Illustrations of BIT handling typical tree queries.

Figure 5(a), the genes of node B is $\{3\}$ ($V_b = [00100]$), and the genes of node G is $\{3,5\}$ ($V_g = [00101]$). Because

$$V_b \wedge V_g = [00100] = V_b,$$

node b is an ancestor of node g.

4.2.2 Computing Lowest Common Ancestor. We first consider the simple case, i.e., computing the LCA of two nodes.We formally describe the definition of LCA in Section 2. Suppose node p is the nearest common ancestor of nodes x and y:

$$x \preceq p, y \preceq p, \nexists z \mid x \preceq z \prec p, y \preceq z \prec p$$

As per Formula 5, the genes of the p must exist in the genes of both x and y:

$$G_p \subseteq G_x, G_p \subseteq G_y,$$

so the ancestors of x and y must exist in this set:

$$p \in \{q \mid G_q \subseteq G_x \cap G_y\}$$

Since the nodes closer to the root have fewer genes, it becomes meaningless for subsequent genes to be the same when the two nodes have different genes. In other words, the genes of the common ancestor must be smaller than any gene where these two nodes differ.

$$G_p = \{\alpha \mid \alpha \in G_x \cap G_y, \alpha < \forall \beta \in G_x \Delta G_y\}$$ [1]

For example, as shown in Figure 4, the node G's genes is $\{3,5\}$([00101]) and the node E's genes is $\{2,5\}$([01001]). Although both have gene 5, since differences between the two gene start from 2, gene 5 will not exist in their common ancestor (Nodes with gene 5 also do not exist). Therefore the genes of the LCA of node G and node E are empty, i.e., the root node A. For node F, its genes are $\{3,4\}$([00110]). There are no different genes before the common genes $\{3\}$ of F and G, so $\{3\}$ exists in the common ancestor B.

As bit-vectors, the bit-vector of P is a prefix of the longest common prefix of the bit-vectors of X and Y, as shown in Figure 5. In some cases, the nodes corresponding to the prefixes might be virtual nodes generated during the restructuring process. Therefore, in actual retrieval, genes from the tail end of the prefix will be gradually removed until a tree node is found.

4.2.3 List Descendants. The two queries mentioned above involve performing calculations and comparisons of bit vectors. Additionally, there are traversal queries, such as querying all descendants of a specific node and tree pruning traversal. These queries can be processed using **gene filters**, which rely on the bitmap indexs.

[1]Δdenotes the symmetric difference : $G_x \Delta G_y = G_x \cup G_y - G_x \cap G_y$

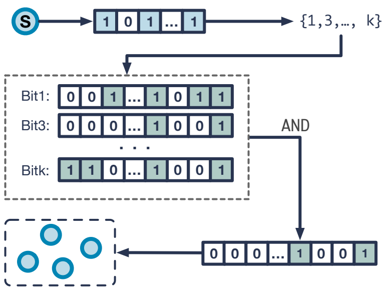

Figure 6: Illustrations of Gene Filter.

Given a genes set G_s, the gene filter needs to retrieve all nodes containing these genes:

$$\mathbf{GF}(G_s) = \{n \mid G_s \subseteq G_n\}$$

A naive approach is to traverse all nodes' bit-vectors and use the previously mentioned method to determine inheritance relationships. However, with the bitmap index, this can be achieved through a few bitwise operations.

Suppose the gene set $G_s = \{g_1, g_2, ..., g_k\}$ consists of k genes. Since genes and bit vectors correspond:

$$G_s \subseteq G_n \iff \forall g \in G_s, V_n[g] = 1$$

As mentioned earlier, in the bitmap index, a particular bitmap represents the existence of a gene in the nodes. Thus, if a node contains the genes in g, it is equivalent to the node's position being true in the bitmap index corresponding to the gene. Combined with Eq. 4, so

$$G_s \subseteq G_n \iff \forall g \in G_s, B_g[n] = 1$$

Therefore, the corresponding bitmap $\mathcal{B}_S = \{B_{g1}, B_{g2}, \ldots, B_{gk}\}$ was first selected from the bitmap set $\mathcal{B}$ based on the gene location of G_S. Then do AND bitwise operation on all bitmaps in $\mathcal{B}_S$ to get B'. At this point, the position corresponding to the index in B' that is 1 is the filtered result. This is demonstrated in Figure 6.

$$\mathbf{GF}(G_s) = \{n \mid B'[n] = 1\}$$

The gene filter can not only filter the included genes but also the excluded genes. The approach is similar, but the bitmap for the excluded genes needs to undergo a bitwise NOT operation before being calculated with other bitmaps using the bitwise AND operation.

4.3 Complexity Analysis

In this section, we analyze the time complexity and space complexity of BIT creation and various BIT-based computations. Let N represent the number of nodes in the tree.

4.3.1 creation. The creation process begins with the Polychotomic encoding. The time complexity of this step is $O(N \log N)$, and the space complexity is $O(N \log N)$. Next is the construction of the BIT, both the time and space complexities are $O(N \log N)$.

In addition to the bitmap, we maintain several hash tables for mapping codes and IDs. The time and space complexities for constructing these hash tables are both $O(N)$. Therefore, the overall time complexity for the creation process is $O(N \log N)$, and the space complexity is also $O(N \log N)$.

4.3.2 determining descendant relationship. First, obtaining the bit encodings of the two nodes has a time complexity of $O(1)$ due to the use of hash tables. Then, performing an AND operation on the two bit vectors has a time complexity of $O(\log N)$. There is no additional space overhead, so the space complexity is $O(1)$, and the overall time complexity is $O(\log N)$.

4.3.3 LCA. or computing the LCA of K nodes, first, the bit vectors of K nodes are obtained, which has a time complexity of $O(K)$. Then, the longest common prefix of the K bit vectors is computed, with a complexity of $O(K \log N)$. There is no additional space overhead, so the space complexity is $O(1)$, and the overall time complexity is $O(K \log N)$.

4.3.4 list descendants. For listing all descendants of a node, first, the bit vector of the input node is obtained, which has a time complexity of $O(1)$. Then, the bit positions that are set to 1 in this vector are iterated through, and the corresponding bitmaps are collected, with a time complexity of $O(\log N)$. Next, AND operations are performed on these bitmaps, which has a time complexity of $O((\log N)^2)$. The additional space overhead is due to the need for a new bitmap to store the result, so the space complexity is $O(\log N)$.

4.3.5 Comparison of algorithms of other tools. Other tools use tree traversal methods to perform computational tasks. The time and space complexity for constructing the tree is $O(N)$. Since BIT involves calculating bit vectors and bitmaps, its construction time and space complexity are greater than that of tree construction. Determining a descendant relationship through tree traversal involves traversing up from the deeper node and making judgments, with a time complexity of $O(h)$, where h is the tree depth, and the worst-case complexity is $O(N)$. In contrast, BIT's method is more stable. For computing LCA for K nodes using tree traversal, the complexity is $O(KN)$. Listing descendants using tree traversal has a time complexity of $O(N)$.

5 IMPLEMENTATION

In this section, we present a brief overview of the development of a versatile tool built upon the methodologies outlined in the preceding sections. Leveraging Java as the implementation language, this tool is designed to offer enhanced functionality and usability. It is provided in the form of a Java Archive (JAR) file, facilitating seamless integration into Java-based projects. BIT is an open source project[2] that is hosted on Github and allows for collaborative development and contributions from the community. The

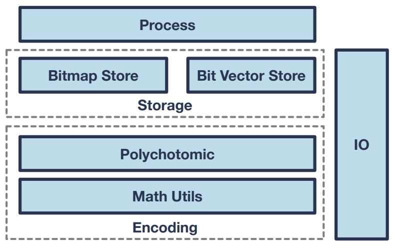

Figure 7: Architecture of BIT

Architecture of BIT is shown in Figure 7. The Encoding module corresponds to Section 3 and includes the Polychotomic algorithm and some mathematical tool functions (e.g. sp, c). The Storage module is responsible for storing bitmaps and bit vectors. The Processing module provides functions T1-T3 and is the implementation of Section 4. IO is responsible for input, output and serialisation. In our implementation, we adopt a specific approach to handle the storage and manipulation of bit-vectors within our Java-based system. To represent bit-vectors, we employ Java byte arrays, which offer a convenient and efficient means of storing binary data. Byte arrays allow us to allocate memory in a compact and contiguous manner, enabling seamless access and manipulation of individual bits within the vector.

For the actual implementation of compressed bitmaps, we turn to *RoaringBitmap*[3], a specialized data structure tailored for compressed bitmap operations in Java environments. *RoaringBitmap* offers a powerful set of features optimized for efficient storage and fast operations on large sets of integers.

6 EXPERIMENT

6.1 Experiment Setup

6.1.1 Baselines. We used 3 systems as baseline to evaluate the performance of BIT:

- Taxonkit [21] is a versatile command-line toolkit that provides efficient and practical operations for manipulating and formatting taxonomic data.
- ETE [13] (The Environment for Tree Exploration) is a Python toolkit designed for the analysis, manipulation, and visualization of tree, offering comprehensive tools for evolutionary research.
- Taxopy [1] is a Python library that streamlines the access and manipulation of taxonomic information, simplifying tasks such as species name resolution and data retrieval from multiple taxonomic sources.

The functions they support are shown in Table 2.

6.1.2 Dataset. We use a public dataset in our experiments. The NCBI Taxonomy Database [20] is a comprehensive and curated database developed and maintained by the National Center for Biotechnology Information (NCBI). This dataset contains approximately

[2]https://github.com/HC-teemo/BIT
[3]https://github.com/RoaringBitmap/RoaringBitmap

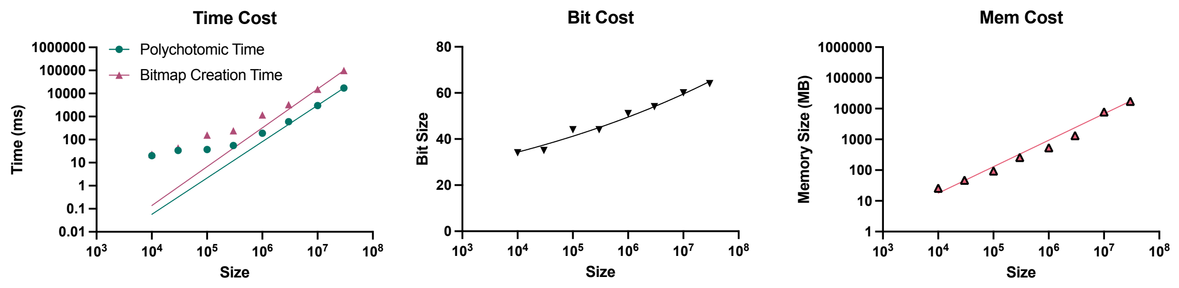

Figure 8: Cost of BIT creation at different data scales

Table 2: Comparison of related tool functions

	ETE	Taxopy	TaxonKit	BIT
List Descendants	✓		✓	✓
LCA	✓	✓	✓	✓
Determining Descendant	✓*	✓*	✓*	✓

* Implemented using LCA.

2.4 million nodes. We are using the latest version (2024-04-14) of taxdump, which can be downloaded from NCBI website[4].

We also did the following processing on top of the NCBI Taxonomy to generate other data. Firstly, starting from the root node, subtrees of depth n are extracted to simulate different sizes of datasets, which we named NCBI-hn (e.g. NCBI-h3, which denotes data 3 levels down from the root node). In addition, we split each rank of the classification tree separately, with the aim of sampling data at a given rank for testing, which we named NCBI-rn (e.g. NCBI-r3 denotes nodes of rank 3).

6.1.3 Hardware and Software. Our experiments were performed on a server containing four Intel(R) Xeon(R) Gold 6230R CPUs running at 2.10GHz, we used NVMe SSD as data storage. We also utilized the latest versions of taxonomy comparison tools, which are: ETE 3.1.3 [5] , Taxopy 0.12.0 [6], Taxonkit 0.16.0 [7].

6.2 Algorithm Evaluation

This section evaluates the construction time and space cost of the algorithm. We then compare the performance of the proposed method with traditional tree traversal methods for three types of queries. Given that the NCBI dataset has a fixed size, we generated trees of varying scales for experimental data, as shown in Table 3.

6.2.1 Cost of Creation. We recorded the encoding and indexing times, the required number of bits, and the maximum memory usage across datasets of different sizes. The results are shown in Table 3.

[4]https://ftp.ncbi.nih.gov/pub/taxonomy/
[5]http://etetoolkit.org/docs/latest/changelog/index.html
[6]https://pypi.org/project/taxopy/
[7]https://bioinf.shenwei.me/taxonkit/

The data from the table is visualized in charts, as shown in Figure 8. The X-axis represents the size of datasets on a log10 scale. The left chart shows the construction time in milliseconds, with the Y-axis also on a log10 scale. The middle chart indicates the number of bits required for encoding, with the Y-axis on a linear scale. The right chart displays memory consumption, with the Y-axis on a log10 scale. As seen from the charts, the bit size approximately follows a log(n) trend, and both time metrics approximate an $n\log(n)$ trend (less apparent in smaller datasets). Memory consumption also approximates an $n\log(n)$ trend. These results are roughly consistent with our expectations.

Table 3: Cost of BIT creation at different data scales

Nodes Number	Polychotomic Time (ms)	Bitmap Creation Time (ms)	Bit Number	Memory Cost(MB)
10K	20	23	34	26
30K	34	43	35	47
100K	37	157	44	94
300K	55	240	44	259
1M	188	1172	51	535
3M	595	3272	54	1,330
10M	3,000	15211	60	7,800
30M	17,244	99374	64	17,310

6.2.2 Algorithm Performance Comparison. In this section, we compare the performance of the BIT algorithm and tree traversal in three types of queries. The results are depicted in Figure 9.

On the left sub-figure, the performance comparison for listing descendants is presented. From the figure, it can be observed that for smaller datasets, the performance of BIT is inferior to tree traversal. This is because the BIT process is more complex, with noticeable overheads such as hash mapping. As the dataset size increases, the execution time of BIT becomes smaller than tree traversal, as the BIT method is less influenced by the dataset size. Parallelization in *list* primarily occurs during the AND operations on multiple bitmaps (where AND operation satisfies the associative property). For smaller datasets, parallelization performs worse due to the additional overhead of parallel processing, such as thread management, synchronization, and context switching, which typically outweighs the performance gains. Parallelization demonstrates its advantages under larger dataset sizes.

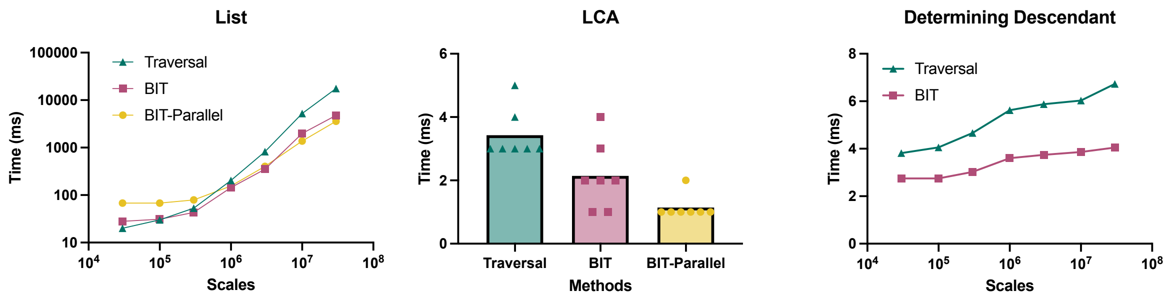

Figure 9: Algorithm Performance Comparison

In the middle sub-figure, the performance of computing the LCA is shown. We calculated the LCA for 512 nodes across 1000 sets of different test data. Due to the short execution time of LCA tasks, there is significant fluctuation influenced by the test data. Therefore, the chart does not distinguish dataset sizes. Each point in the chart represents the execution time of a single run, while the bar chart represents the average time. It can be observed that BIT consumes less time than tree traversal for LCA computation. Parallelization in LCA mainly occurs during multiple node LCA calculations, where common prefix operations on multiple vectors are required. Thus, the performance of parallelization mainly depends on the number of input nodes K, with parallelization showing no advantage when K is small.

On the right sub-figure, the performance of determining ancestors is depicted, recorded over 10,000 runs. Overall, the time taken by BIT is less than tree traversal, and the increase is more gradual. We analyze that in larger datasets, due to cache locality, tree traversal may lead to decreased cache hit rates. In contrast, the main overhead of BIT is bit vector operations, and bit vectors are contiguous in memory (as byte arrays), which helps maintain cache coherence.

6.3 Tools Creation Cost Evaluation

In this section, we compare the creation cost of BIT and other tools. Since the other tools do not allow for manual dataset settings, this experiment is conducted using the complete NCBI Taxonomy data.

Table 4: Comparison of the creation cost to other tools

Tools	Creation Time (s)	Memory Used (GB)
BIT-init	27	1.3
BIT	0.7	0.67
Taxonpy	7.1	0.87
TaxonKit	3.2	0.82
ETE-init	129.4	3.0

The results are shown in Table 4. At this stage, the processing procedures of different tools are not entirely the same. Taxonpy and TaxonKit have similar processing procedures, primarily involving reading the NCBI data files (either CSV or dump files) and loading them into memory. The data structure used by these two tools for tree construction is a map (or dictionary in Python) from `taxonId` to `parentID`. Their creation times are of the same order of magnitude, and the memory usage is also comparable.

BIT has two scenarios: The first is the initial creation, which includes reading the CSV file storing the tree data, encoding using the Polychotomic algorithm, and constructing the bitmap index. BIT serializes these computed data into binary files for storage. The second scenario involves BIT directly reading and deserializing these binary files into memory. In most cases, since the NCBI data does not update frequently, BIT only needs to load the binary files without repeating the indexing process. As can be seen from the results, the initial creation of BIT takes a considerable amount of time. And BIT loads data directly faster and with less memory consumption than the other two tools.

The process for ETE is somewhat unique. ETE, upon first use, loads the NCBI dump data, parses it, and creates a local SQLite database to store the tree (on disk). This process consumes a significant amount of time and memory. In subsequent uses, ETE accesses this database to execute operations, meaning that ETE requires almost no creation time and memory during use.

6.4 List Performance Evaluation

The experiment aimed to evaluate the performance of BIT in handling *List Decendants* (**T3**) tasks, contrasting with Taxonkit and ETE systems. We simultaneously inputted 1, 10, 100, 1000, and 10000 taxids, listed their descendant lists, and recorded the execution time. Each experiment was conducted 10 times, and the average was taken to reduce errors. The ids used for testing were sampled from the 6th rank (r6) of the taxonomy tree.

The experimental results are shown in Figure 10(List), the scales of both X and Y axes are logarithmic. The running time of ETE for computing 10000 taxids was excessively long and therefore not recorded. The execution time and input quantity of all three methods generally exhibit linear relationships (Taxonkit-E indicates exclusion of loading time). In terms of performance, the average execution time for Taxonkit is 10-53ms/taxId, which is approximately 4 times faster than ETE (70-200ms/taxId). The performance of BIT is significantly superior to other methods by about **1 to 2 orders of magnitude** (1-3ms/taxId), especially when dealing with large-scale data. This is because bitwise operations do not result in frequent memory addressing issues associated with processing large amounts of data.

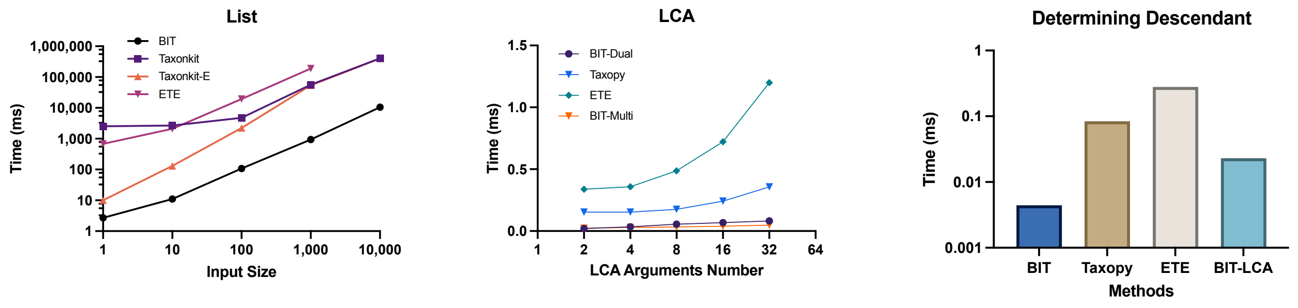

Figure 10: Performance of three Tasks

6.5 LCA Performance Evaluation

This experiment aimed to evaluate the performance of BIT in handling the *Lowest Common Ancestor* (**T2**) task, compared to systems such as Taxopy and ETE. Taxonkit was not included in this comparison. This decision was based on the short execution time of the LCA task, neither of which supports batch processing. Taxonkit's proportionally high data preprocessing time and its fluctuation prevent accurate measurement of actual LCA computation times.

In this experiment, m taxids (where $m = 2, 4, 8, 16, 32$) were inputted to calculate their LCA, and the execution time was recorded. Each experiment was conducted 10 times to calculate an average, thus minimizing error. The IDs for testing were sampled using the following rule: initially, three nodes were sampled from the 10th rank (r10), followed by sampling 20 descendants from each of these three nodes (resulting in a total of 60 nodes). m taxids (without repetition) were then randomly selected from these 60 nodes to serve as the test inputs.

The experimental results are illustrated in Figure 10(LCA), where the x-axis represents the number of TaxIds (m) for computing the LCA on a log_2 scale, and the y-axis represents the execution time in milliseconds. Two different strategies for BIT are presented in the chart. `BIT-Dual` denotes the approach of performing multiple calculations using binary LCA (`lca(id1, id2)`), while `BIT-Multi` represents the method of employing multiple input LCA (`lca(id[])`).

In terms of efficiency, ETE exhibits the slowest performance, with a time of 0.34 ms at $m = 2$. Taxopy is approximately twice as fast as ETE, with an execution time of around 0.15 ms. BIT outperforms both Taxopy and ETE by 6-14 times (BIT-Dual and BIT-Multi yield the same result when $m = 2$, thus not distinguishing between them). Due to BIT's LCA algorithm being based on bitwise operations for the longest common prefix computation, it can process multiple calculations simultaneously. BIT-Multi, by rapidly computing common prefixes, effectively avoids unnecessary redundant calculations, resulting in the fastest performance.

6.6 Determining Descendant Performance Evaluation

The aim of this experiment is to evaluate the performance of determining descendant relationships (**T1**). As other systems do not support this specific computational task, alternative methods supported by them are utilized. The first method employs the `LCA` operation: if $LCA(A, B) = A$ and $A \neq B$, then B is considered a descendant of A. The second method utilizes the `list` operation: if $B \in list(A)$, then B is regarded as a descendant of A. Since in a tree, the computational space for upward LCA calculation is smaller than that for downward list calculation, we opt to use LCA for the T1 task in this experiment.

The experimental results are presented in Figure 10(Determining Descendant), where the x-axis represents different methods, and the y-axis represents the execution time (in milliseconds). It is evident that BIT outperforms the other two systems by 1-2 orders of magnitude in terms of performance. To highlight the advantage of the BIT algorithm, we have incorporated an LCA-based algorithm similar to the ones used in the other two systems for comparison purposes. It can be observed that the method employed by BIT is faster than the BIT-LCA-based methods. This is because the BIT method only requires a single AND operation and comparison, in contrast to the more complex calculations involved in the LCA-based methods utilized by the other systems.

7 RELATED WORK

In this section, we will introduce some tools and programs for analyzing and processing NCBI and other sourced taxonomy data.

The Entrez Programming Utilities (E-utilities) [19] offer a stable interface to the NCBI Entrez query and database system. With nine server-side programs, E-utilities use a fixed URL syntax to retrieve data from NCBI's 38 databases, enabling customized data pipelines and simplified queries via a command-line interface.

Taxize [2] is an open-source R package providing programmatic access to taxonomic data from 13 web sources. It simplifies retrieving and resolving taxonomic names with functions that interact with multiple data sources via web APIs, enhancing data manipulation, visualization, and analysis in research workflows. Both E-utilities and Taxize use Web APIs for data access and lack robust local testing methods; E-utilities also lacks an open-source version.

The Environment for Tree Exploration (ETE) [12] revolutionizes hierarchical tree analysis with its Python toolkit, offering automated manipulation, diverse analysis options, and customizable visualizations. ETE supports various tree formats, facilitating phylogenetic and clustering tree analysis. ETE 3 [13] introduces improvements and standalone tools for comparative genomics and phylogenetics, integrating seamlessly with the NCBI taxonomy database.

TaxonKit [21] is a command-line toolkit for efficient processing of NCBI taxonomy data. With seven core subcommands, it offers functionalities like TaxIds querying and change tracking. TaxonKit can also support GTDB taxonomy by converting data into the NCBI taxonomy dump file format, addressing features like filtering TaxIds by rank range and converting lineages into custom formats for metagenomic analysis reports.

Taxopy [1] is a Python package for NCBI-formatted taxonomic databases, offering features like lineage retrieval, lowest common ancestor determination, and taxonomy name extraction from TaxIds. It allows creating Taxon objects for detailed classification information and facilitates identifying shared classifications among lineages, with adjustable stringency and agreement level evaluation.

Since E-Utilities and Taxize perform computations by calling remote service APIs, they were not chosen as comparison objects in this experiment.

8 CONCLUSION

Our study aims to accelerate the use of NCBI taxonomy in biomedical tasks, such as computing the lowest common ancestor, determining descendant relationships, and listing node descendants. Existing tools often struggle with inefficient runtime performance. To address this, we introduce a novel indexing method called BIT, specifically designed for handling tree-like data structures. Using the Polychotomic encoding algorithm, BIT encodes the tree structure into bit-vectors stored in bitmaps. By leveraging parallel bit operations, BIT significantly enhances computational speed. Our experimental evaluations with public datasets confirm that BIT outperforms baseline systems in task execution performance.

Future work could focus on the following aspects: 1) Algorithm Optimization to Reduce Memory Usage: Further efforts can be directed towards optimizing the BIT algorithm to minimize memory consumption, enhancing its scalability for handling larger datasets. 2) Expanded Tool Functionality: There is a need to incorporate additional commonly used features into the BIT toolkit, such as advanced querying options and support for diverse dataset. 3) Enhanced Usability: Improvements in usability, such as implementing the toolkit in Python and providing it as a packaged library.

ACKNOWLEDGEMENT

This work was supported by the National Key R&D Program of China(Grant No.2021YFF0704200) and Informatization Plan of Chinese Academy of Sciences(Grant No.CAS-WX2022GC-02)

REFERENCES

[1] Antônio Camargo. 2022. apcamargo/taxopy: v0.10.1 (v0.10.1). https://doi.org/10.5281/zenodo.6993581

[2] S.A. Chamberlain and E. Szöcs. 2013. taxize: taxonomic search and retrieval in R. *F1000Research* 2 (2013).

[3] Zhen Chen, Yuhao Wen, Junwei Cao, Wenxun Zheng, Jiahui Chang, Yinjun Wu, Ge Ma, Mourad Hakmaoui, and Guodong Peng. 2015. A survey of bitmap index compression algorithms for big data. *Tsinghua Science and Technology* 20, 1 (2015), 100–115.

[4] Pierre Colomb, Olivier Raynaud, and Eric Thierry. 2008. Generalized polychotomic encoding: a very short bit-vector encoding of tree hierarchies. In *Modelling, Computation and Optimization in Information Systems and Management Sciences: Second International Conference MCO 2008, Metz, France-Luxembourg, September 8-10, 2008. Proceedings*. Springer, 77–86.

[5] François Deliège and Torben Bach Pedersen. 2010. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *Proceedings of the 13th international conference on Extending Database Technology*. 228–239.

[6] Scott Federhen. 2012. The NCBI taxonomy database. *Nucleic acids research* 40, D1 (2012), D136–D143.

[7] Robert E Filman. 2002. Polychotomic encoding: A better quasi-optimal bit-vector encoding of tree hierarchies. In *ECOOP 2002—Object-Oriented Programming: 16th European Conference Málaga, Spain, June 10–14, 2002 Proceedings 16*. Springer, 545–561.

[8] Francesco Fusco, Michail Vlachos, Xenofontas Dimitropoulos, and Luca Deri. 2013. Indexing million of packets per second using GPUs. In *Proceedings of the 2013 conference on Internet measurement conference*. 327–332.

[9] Kaoutar Ghazi, Laurent Beaudou, and Olivier Raynaud. 2018. Algorithms for a Bit-Vector Encoding of Trees. In *Intelligent Computing & Optimization*. Springer, 418–427.

[10] Gheorghi Guzun, Guadalupe Canahuate, David Chiu, and Jason Sawin. 2014. A tunable compression framework for bitmap indices. In *2014 IEEE 30th international conference on data engineering*. IEEE, 484–495.

[11] Michel Habib, Lhouari Nourine, Olivier Raynaud, and Eric Thierry. 2004. Computational aspects of the 2-dimension of partially ordered sets. *Theoretical Computer Science* 312, 2-3 (2004), 401–431.

[12] J. Huerta-Cepas, J. Dopazo, and T. Gabaldón. 2010. ETE: a python Environment for Tree Exploration. *BMC bioinformatics* 11 (2010), 1–7.

[13] J. Huerta-Cepas, F. Serra, and P. Bork. 2016. ETE 3: reconstruction, analysis, and visualization of phylogenomic data. *Molecular biology and evolution* 33, 6 (2016), 1635–1638.

[14] Justin Kuczynski, Jesse Stombaugh, William A Walters, Antonio González, J Gregory Caporaso, and Rob Knight. 2012. Using QIIME to analyze 16S rRNA gene sequences from microbial communities. *Current protocols in microbiology* 27, 1 (2012), 1E–5.

[15] Yong-Xin Liu, Yuan Qin, Tong Chen, Meiping Lu, Xubo Qian, Xiaoxuan Guo, and Yang Bai. 2021. A practical guide to amplicon and metagenomic analysis of microbiome data. *Protein & Cell* 12, 5 (2021), 315–330. https://doi.org/10.1007/s13238-020-00724-8

[16] Alessio Milanese, Daniel R. Mende, Lucas Paoli, Guillem Salazar, Hans-Joachim Ruscheweyh, Manuel Cuenca, ..., and Shinichi Sunagawa. 2019. Microbial abundance, activity and population genomic profiling with mOTUs2. *Nature communications* 10, 1 (2019), 1014.

[17] Patrick E O'Neil. 2005. Model 204 architecture and performance. In *High Performance Transaction Systems: 2nd International Workshop Asilomar Conference Center, Pacific Grove, CA, USA September 28–30, 1987 Proceedings*. Springer, 39–59.

[18] Olivier Raynaud and Eric Thierry. 2001. A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. In *ECOOP 2001—Object-Oriented Programming: 15th European Conference Budapest, Hungary, June 18–22, 2001 Proceedings 15*. Springer, 165–180.

[19] Eric Sayers. 2010. A General Introduction to the E-utilities. Entrez Programming Utilities Help [Internet]. Bethesda (MD): National Center for Biotechnology Information (US).

[20] Conrad L Schoch, Stacy Ciufo, Mikhail Domrachev, Carol L Hotton, Sivakumar Kannan, Rogneda Khovanskaya, Detlef Leipe, Richard Mcveigh, Kathleen O'Neill, Barbara Robbertse, Shobha Sharma, Vladimir Soussov, John P Sullivan, Lu Sun, Seán Turner, and Ilene Karsch-Mizrachi. 2020. NCBI Taxonomy: a comprehensive update on curation, resources and tools. *Database* (Jan 2020). https://doi.org/10.1093/database/baaa062

[21] W. Shen and H. Ren. 2021. TaxonKit: A practical and efficient NCBI taxonomy toolkit. *Journal of genetics and genomics* 48, 9 (2021), 844–850.

[22] Arie Shoshani, Luis M Bernardo, Henrik Nordberg, Doron Rotem, and Alex Sim. 1999. Multidimensional indexing and query coordination for tertiary storage management. In *Proceedings. Eleventh International Conference on Scientific and Statistical Database Management*. IEEE, 214–225.

[23] Emanuel Sperner. 1928. Ein satz über untermengen einer endlichen menge. *Mathematische Zeitschrift* 27, 1 (1928), 544–548.

[24] Kurt Stockinger. 2002. Bitmap Indices for Speeding Up High-Dimensional Data Analysis. In *Database and Expert Systems Applications*. 881–890.

[25] Kurt Stockinger and Kesheng Wu. 2007. Bitmap indices for data warehouses. In *Data Warehouses and OLAP: Concepts, Architectures and Solutions*. IGI Global, 157–178.

[26] Duy Tin Truong, Eric A. Franzosa, Timothy L. Tickle, Matthias Scholz, George Weingart, Edoardo Pasolli, ..., and Nicola Segata. 2015. MetaPhlAn2 for enhanced metagenomic taxonomic profiling. *Nature methods* 12, 10 (2015), 902–903.

[27] Tzu-Hsuan Wei, Soumya Dutta, and Han-Wei Shen. 2018. Information guided data sampling and recovery using bitmap indexing. In *2018 IEEE Pacific Visualization Symposium (PacificVis)*. IEEE, 56–65.

[28] Derrick E. Wood and Steven L. Salzberg. 2014. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology* 15 (2014), 1–12.

[29] Kesheng Wu, Sean Ahern, E Wes Bethel, Jacqueline Chen, Hank Childs, Estelle Cormier-Michel, Cameron Geddes, Junmin Gu, Hans Hagen, Bernd Hamann, et al. 2009. FastBit: interactively searching massive data. In *Journal of Physics: Conference Series*, Vol. 180. IOP Publishing, 012053.