

Received June 18, 2021, accepted July 25, 2021, date of publication July 30, 2021, date of current version August 11, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3101578

# An Entropy-Based Approach: Compressing Names for NDN Lookup

TIANYUAN NIU<sup>1,2</sup> AND FAN YANG<sup>1,2,3,4</sup>

<sup>1</sup>State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China

<sup>2</sup>Beijing Key Laboratory of Network System Architecture and Convergence, Beijing University of Posts and Telecommunications, Beijing 100876, China

<sup>3</sup>Purple Mountain Laboratories, Nanjing 211111, China

<sup>4</sup>Peng Cheng Laboratory, Shenzhen 518055, China

Corresponding author: Fan Yang (yfan@bupt.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1800602, and in part by Science and Technology on Information Transmission and Dissemination in Communication Networks Laboratory Research Project under Grant SKX192010028.

**ABSTRACT** NDN (Named Data Networking) is one of the most popular future network architecture, a “clean slate” design for replacing the traditional TCP/IP network. However, the lookup algorithm of FIB entry in NDN is the bottleneck of the current NDN. Owing to the unique identifier of content name, whose length is variable, the size of FIB entries is proliferating, and the effectiveness of lookup algorithms is low. This paper proposed an entropy-oriented name processing mechanism, compressing the content names effectively by bringing in an encoding scheme. This mechanism can be split into two parts: name compression and lookup. The first part compressed the content names and converted them into a kind of code with a smaller size by considering the information redundancies of content names; the second part built a compact structure to minimize the memory footprint of FIB entries with keeping the high lookup performance. This mechanism outperformed many traditional name lookup algorithms, had better flexibility and cost less memory footprint.

**INDEX TERMS** Named Data Networking, lookup algorithms.


## I. INTRODUCTION

Name Data Networking (NDN) [1] is one of the most valuable future network technologies, which is a clean-slate architecture for replacing the traditional one based on TCP/IP, concentrating on the content itself rather than the location. In NDN, the content name, the equivalent to IP address in traditional TCP/IP network architecture, identifies the content uniquely through all data transmission. However, the number of content names is countless whose lengths are also boundless [1], [2], thereby confronting two critical problems: the low efficiency of the packet forwarding and the high memory footprint of FIB entries.

Current lookup algorithms are designed for conventional IP forwarding, and the precedent hardware optimizations are only applied to the fixed-length IP addresses whose lengths are fixed. In contrast, content names consist of numerous variable-length components, indicating that many IP-based lookup algorithms [3]–[5] cannot be utilized in name-based lookup algorithms. Furthermore, there are many naming

conventions [2] and various kinds of them have various features. Because of this phenomenon, it is impossible to customize quantities of optimized algorithms to fit those various features completely. However, such a problem has been ignored: the designers concentrate on the significance of content rather than a fundamental use of a network—data transmission. From this aspect, content names carry far more information than an identifier of a data packet needs. Given that, the best way to solve this problem is to compress the content names according to Shannon Information theories [6], keeping both the singularity and compactness for an identifier.

In this paper, we combat the challenges mentioned above and design a super-effective NDN name encoding algorithm and a relative lookup algorithm. In this system, we develop the Paradigm Huffman Trie (PHT) to compress content names and CompactTrie to adapt the characteristics of PHT and accelerate the lookup operation. PHT, based on Paradigm Huffman Encoding [7], can eliminate the redundancy of content names from the perspective of information theory to compress the names composed by various-length components effectively and convert them into uniform codes.

The associate editor coordinating the review of this manuscript and approving it for publication was Giovanni Merlino .

CompactTrie is designed for constructing an efficient structure to accelerate the lookup operation, which adopts the features of PHT codes. Meanwhile, our design also includes insertion and deletion operations to provide scalability and flexibility for the lookup algorithm. The experiment shows that the lookup delay of our design is around 10 milliseconds with different size datasets, and the memory footprint is approximately 2 to 3 Megabytes, both lower than other lookup algorithms we tested. The evaluation results reveal the high efficiency, flexibility, and scalability of our work.

The rest of this paper is structured as follows: Section II describes related work, such as trie-based and hash-based lookup algorithms. Then, Section III analyzes the characteristics of the current content name and the necessity of compressing them, motivates our design principle, and introduces a special kind of Huffman Encoding Algorithm. What is more, Section IV, V and VI presents our specific method for name compression, an efficient data structure to adapt compressed names and accelerate the lookup operation of FIB entries, and the update policy for our methods. Section VII contains a detailed performance evaluation for name compression and compares our design with other name lookup algorithms from the perspective of lookup delay and memory footprint. In the end, Section VIII concludes our proposed methods, analyzes the advantages and disadvantages of our work, and illustrates the future work for our next step.

## II. RELATED WORK

The lookup algorithm of NDN can be divided into two categories in [8]: trie-based [9]–[14], hash-based [15]–[21], bloom filter-based [22]–[24] and other kind of lookup algorithm [25]–[28]. Considering the demand of Longest Prefix Match (LPM) in different lookup methods, trie-based algorithms, such as NPT [9], NCE [10], MATA [11], BPT [12], can collaborate with the LPM method effectively owing to the feature of trie structure. In contrast, hash-based algorithms need to construct an effective data structure to fit LPM method [15], [16]. Trie-based lookup algorithm is apt to operate insertion, deletion and update, alongside with its low memory footprint [10]. However, the time complexity of trie-based lookup operation usually relies on the depth of the trie structure and the length of a name.

Trie-based method is a common category in NDN name lookup, and specifically, it can be separated as four sub-categories [8], [28]: component-based [9], encoded component-based [10], [14], [29], character-based [11], [13] and bit-based [12]. Their data structures are all based on trie structure, and the difference is only the granularity when inserting a name into a trie. Trie structure is widely adopted by FIB lookup in NDN because it can store the logic relation of names and support the LPM naturally. In addition, trie-based methods often consume less memory but take more time to search the trie, thus impelling researchers to propose optimized structures to overcome the defections of the trie. Two optimized trie-based algorithms are introduced below to

comprehend what measures can be taken for better lookup performance.

The first Trie-based that our method compares with is bitmap-based pNPT [29]. This method uses priority Name Prefix Trie and separates on-chip processing from off-chip processing, while most of the LPM procedures are performed with on-chip memories. Priority trie replaces empty nodes with leaf nodes, which contain the longest prefix names, hence reducing the LPM lookup process. pNPT consumes less memory than NPT because there are no empty nodes in the priority trie structure. In the meantime, the lookup process is more efficient on account of the priority trie structure. Compared with classic Trie-based schemes, pNPT employs an encoded bitmap and an edge table structure to store Trie node and edge information, both of which are on on-chip memory for efficient name lookup.

The second Trie-based method is NameTrie [13]. NameTrie introduces an encoding mechanism, called minASCII, utilizing the unused bit in ASCII codes to represent much valuable information. NameTrie structure is a bit-level trie structure and exploits two tables to store node and edge information, respectively. Each of them makes a great effort to optimize the structure for efficient lookup. The first structure is *NameNodes*, and it stores the node information of NameTrie with a minASCII encoding mechanism. This mechanism utilizes unused codes to replace common symbols in NDN names, such as delimiter and End Of Piece (EOP) marker, and Most Significant Bit of all characters to store pointer information, which is unused as well. Since the implementation and search procedures of pointers between two nodes in Trie structures are complicated and inefficient, the second structure *EdgeHT* is employed to store edge information. *EdgeHT* is a hash table and can provide high-speed lookup for the location of the next name piece.

Compared to trie-based algorithms, hash-based lookup algorithms are quite the opposite in many aspects. Unlike trie-based methods, hash-based algorithms usually utilize extra data structure to satisfy the demand of LPM, which contains different hash elements to distinguish name prefixes by length. The time complexity of the hash table lookup operation is  $O(1)$ , so the time complexity of a whole lookup operation depends on the extra data structure destined to perform LPM. However, hash-based lookup algorithms have some flaws, such as high memory footprint, lack of flexibility, and hash collision. Regardless of these defects, many NDN name lookup algorithms still employ the hash-based method as their first choice because the time complexity is quite essential. Two typical hash-based name lookup algorithms will be introduced briefly below to know about the current work on this research.

The first hash-based algorithm adopts a two-phase LPM method, uses a virtual prefix to improve the performance, and can resist DOS attack [15]. It scatters the original name set into some sub-sets, differentiated by the number of components in each name, and each sub-set is stored as a hash table whose index is the original name. The lookup operation first

picks a sub-set with a short length name, whose number of components is specified as  $M$ , to initiate the lookup process. For the prefix longer than  $M$ , a virtual prefix that is the same as the previous  $M$  components of the current name is added to the routing table. The number of original name components is denoted as  $MD$ . In the first phase of the LPM, it will start from the sub-set with  $M$  components. If this fails, the lookup operation will continue to other subsets with shorter name components until it succeeds. If there is no match in the first phase, then the entire lookup operation fails. When the first phase is successful, the second phase begins and proceeds the lookup operation at the sub-set with  $MD$  components. Choosing an appropriate factor  $M$  will reduce the number of lookup operations, and the only phase one of LPM will be needed in most cases. However, the virtual prefix brought by this algorithm expands the size of FIB entries, which is enormous for the first time, and this leads to a high memory footprint.

The second one is based on the binary search tree, inside whose node is composed by hash table [16]. Like the first hash-based algorithm, this algorithm also scatters the original name set by the number of components but organizes the sub-sets as a binary search tree. A binary search runs on the node of this tree structure during each lookup operation to locate the LPM result. On each node, if there is a matching prefix in the corresponding hash table, then the right sub-tree is recursively traversed to find a longer prefix; if it does not match, the longest prefix must appear in the left sub-tree; hence it is necessary to traverse the left sub-tree recursively. The time complexity of the lookup operation is linearly related to  $\log(k)$ , where  $k$  is the number of components in a name. However, this algorithm also needs to bring in the virtual prefix to solve the missing lookup because of the non-sequential traversal of the binary search tree, which means this algorithm also results in a high memory footprint.

The time complexity of the hash table lookup is  $O(1)$ , but an extra structure or mechanism is needed to effectively guarantee the LPM and aggregate name entries. Therefore, the algorithms which are discussed above construct the global data structure according to the length of name components with a local hash table. Although both algorithms maintain good performance on the time scale, they consume much more memory than other name lookup algorithms. In addition to the high memory footprint, some critical problems in hash-based algorithms are hash collision and poor scalability. In order to solve these problems, a hash table structure needs to occupy more space, and memory usage is wasteful from a certain view.

### III. AN ENTROPY-BASED APPROACH

The existing lookup algorithms confront the following problems: 1) the strategy of exchanging space for time leads to the high memory footprint, and the FIB of NDN is exponentially expanded; 2) content names, which are the identifiers of packets in data transmission, include the pragmatic level information. However, only the grammatical level is needed;

3) There are many naming conventions, which means the characteristics of NDN FIB entries are not unique. The first point has been described in I, and the following two issues will be analyzed below.

The content name belongs to the application layer in the traditional TCP/IP architecture and contains human-readable pragmatic information. However, when NDN uses the content name as the unique identifier for data transmission, pragmatic information is redundant, from the perspective of Shannon theories [6]. Even for IP FIB entries, the data structure has a redundant part [3]. Data transmission requires only simple grammatical information to complete the forwarding behavior, which is the ultimate problem of NDN FIB entries being much larger than IP. In addition, for the sake of data transmission, the identifier of transmitted data only needs to be globally or locally unique.

In summary, the NDN architecture requires a name processing mechanism, i.e., the capability to compress names into grammatical information, thus more appropriate for data transmission. Lossless data compression in communication systems is adequate to address such needs. After using the lossless data compression, we will have the following advantages in NDN:

- 1) The capability to compress the original name to make it appropriate for data transmission;
- 2) A translatable encoding algorithm to assure the local or global uniqueness of a name;
- 3) After encoding or compression, the content name has a unified form, and it is easy to design the general lookup algorithm of FIB entries.

Therefore, the design proposed in this paper proposes an encoding mechanism and maps the name to a binary code by some rules. The encoded name, or code, has a unified form, which is the new identifier of content in place of the original name. In the meantime, no matter what naming convention is, names in the data packet are displayed as a binary string consisting of 0 and 1. Furthermore, we design a compact structure and an efficient name lookup algorithm according to the characteristics of encoded names.

As described above, the unique identifier of the current NDN packet, the content name, contains abundant pragmatic information, which is unnecessary for data transmission. Specifically, as for English, a name is generally composed of 26 letters, but a valid name only needs part and arranges them in a fixed order. Compared to invalid names in which letters are arranged randomly, the valid names constitute a tiny percentage in the set where each name is composed of random order letters. Hence, the valid names constitute a finite rule set. For an NDN name, the length and number of name components are infinite, meaning that both the name component set and the name set are infinite sets. However, according to the characteristics of content names, the name component should be a finite set, and the length of a name component should also be limited. It is necessary to encode and compress names or components for better performance in data transmission. Therefore, the algorithm presented in this

paper introduces an encoding mechanism to ensure proper and efficient compression of names and local or global uniqueness.

To operate the longest prefix match, the level of compression is the component level instead of the name level. Many names in a namespace share the same prefix, so hierarchical names can be aggregated in FIB. At the same time, the existing URL database, which is generally considered to be the representation of content names, has the following rules: 1) a name component appears in the same level of multiple names, and there are frequency differences between them; 2) Some name component only appears following a specific name prefix; 3) For a specific name prefix, components in the next level constitute a finite set. Considering that the IP addresses can be compressed into a dictionary tree [30], also known as the trie, name sets in the namespace can also be aggregated into a trie at the granularity of component, where each node in the trie represents a name component. According to the characteristics of URL databases, if the name set in the namespace is constructed as a trie, the child nodes of each node can form a finite set, and the frequency of these child nodes is different.

According to the above analysis, a simple encoding algorithm can be obtained. Since the frequency of each name component following the same prefix is different, the entire name component set in the namespace can be encoded using the paradigm Huffman algorithm [31], an improved Huffman algorithm [7] which is one of the most excellent encoding algorithms. However, there are some problems with that encoding algorithm. For example, since the name component sets in the namespace is huge, the length of a code with low frequency is longer than the original name before encoding, and some name component only appears after a specific prefix. Therefore, it is not appropriate to perform global name component encoding.

Reconsidering the characteristics of names in the URL database, if a name prefix can guarantee global uniqueness, the remaining name components can surmount the globally unique restriction and vice versa. According to this feature, in the dictionary tree constructed by name sets in the namespace, child nodes under each node constitute a finite set and are independent of other sets. If this feature is true, the finite set of name components represented by child nodes under each node in the dictionary tree can be independently encoded without encoding the entire set of name components in the complete namespace. Therefore, the problem that code is too long after encoding is solved. A rigorous proof of the above conclusions will be given below.

Suppose a name  $N$  can be expressed in the following form:  $N = C_1/C_2/\dots/C_m$ , where  $C_i$  represents the  $i$ -th name component and belongs to a finite set, and  $m$  represents the number of current components a name has. The information contained in  $N$  can be expressed by Equation (1):

$$H(N) = H(C_1 C_2 \dots C_m) \quad (1)$$

where  $H(\cdot)$  represents the information entropy function [6].

From Equation (1), the information entropy of a name can be represented by the joint information entropy of name components. At the same time, the name components at all levels are not independent. Therefore,  $H(C_1 C_2 \dots C_m)$  can be expressed in another form, as shown in Equation (2):

$$H(C_1 C_2 \dots C_m) = H(C_1) + H(C_2|C_1) + \dots + H(C_m|C_{m-1} \dots C_2 C_1) \quad (2)$$

Equation (2) can be obtained from the additivity of entropy. In Equation (2), the right side of this equation is the conditional entropy except for the first term, which means that except for the first-level name component, the information quantity of all other name components is the amount of information in a prefix. When the prefix provides enough information, the information in the prefix is needed exclusively.

Suppose there are two names consisting of two components,  $N_1 = A_1 A_2$ ,  $N_2 = B_1 B_2$ , and it is known that these names are independent of each other, thus, they satisfy Equation (3):

$$H(N_1 N_2) = H(N_1) + H(N_2) \quad (3)$$

If the names are represented by name components, Equation (3) can be expressed as Equation (4):

$$H(A_1 A_2 B_1 B_2) = H(A_1 A_2) + H(B_1 B_2) \quad (4)$$

According to the additivity of entropy, Equation (4) can be expressed as Equation (5):

$$H(A_1 B_1) + H(A_2 B_2|A_1 B_1) = H(A_1) + H(A_2|A_1) + H(B_1) + H(B_2|B_1) \quad (5)$$

The prefixes obtained by known conditions are independent of each other, and Equation (6) is available:

$$H(A_1 B_1) = H(A_1) + H(B_1) \quad (6)$$

Comparing two equations above, Equation (7) can be derived:

$$H(A_2 B_2|A_1 B_1) = H(A_2|A_1) + H(B_2|B_1) \quad (7)$$

Equation (7) shows that if two names with two components are independent, the name components with the different prefix are also independent of each other, and this conclusion can also be extended to the case where the number of name component is  $M$ , so the original proposition is proved. From this, we can get Theorem 1:

*Theorem 1:* The finite sets are independent if the prefix is different.

After the rigorous proof, Theorem 1 can be applied to the design of the actual encoding algorithm. Our goal is to compress content names into Huffman codes and use them to replace names in FIB entries, thus acquiring a better performance in memory footprint. The specific design and implementation of this encoding algorithm will be described thoroughly in the next section. Before that, we will introduce the paradigm Huffman algorithm briefly for the sake of a deep comprehension of the encoding mechanism that we design.





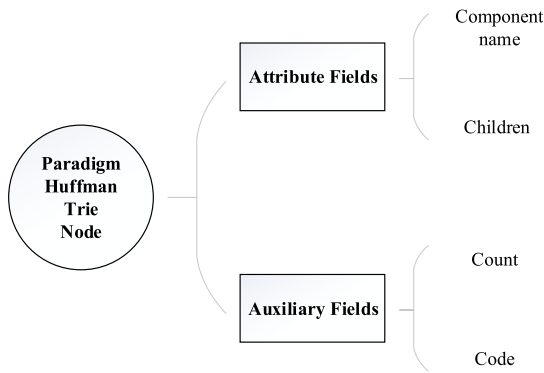


FIGURE 2. Trie node diagram.

**Algorithm 1** PHT Construction

```

Input: Name set  $S$ 
Output: Root node  $R$ 
Initialize root node  $R$ , component queue  $Q$ 
Initialize pointer  $p$  pointing to  $R$ 
for all name components  $c \in S$  do
    Enqueue  $c$  into  $Q$ 
end for
while  $Q \neq \emptyset$  do
    Dequeue a name component  $n$  from  $Q$ 
    if  $Search(p.children, n)$  then
        Increment  $p.count$  and  $p \rightarrow p.children$ 
    else
        Insert  $n$  into  $p.children$  and  $p \rightarrow p.children$ 
    end if
end while
    
```

need to be inserted. A pointer is needed to mark the position to be inserted. At initialization, the pointer points to the root node of PHT. Depending on whether it exists, the insertion can be split into two steps: search and create. The first step is to determine whether there is a prefix of the inserting name in PHT, and the second step is to create a new trie node when the search fails. The granularity of each search and creation is on the component level.

Each search first accesses the child nodes of the current node to distinguish whether the inserting name component already exists in the child nodes. If it does, the *count* field of the current name component will be incremented, the pointer points to the matching child node, and the insertion operation of the next component level continues.

When the creation begins, a new PHT node will be initialized, whose fields will be initialized too. The *component name* is set as the current name component, and other fields are set as default values. Then, this new node will be added to the child nodes of the current node and make the pointer points to the new node. After that, the creation operation completes.

**B. CALCULATION AND ASSIGNMENT OF NAME CODE**

According to the theory in Section III, the code set should be limited to a finite name set where all components have a common prefix. Therefore, when calculating the code of a specific component, it should be calculated at its parent node, and then the codes are assigned to each child node. Since all the nodes need encoding in PHT, the entire PHT needs to be traversed. The traversal method adopted by this algorithm is Breadth-First Search (BFS), and the encoding method is paradigm Huffman coding presented in Section III. The specific calculation and allocation are shown in Algorithm 2.

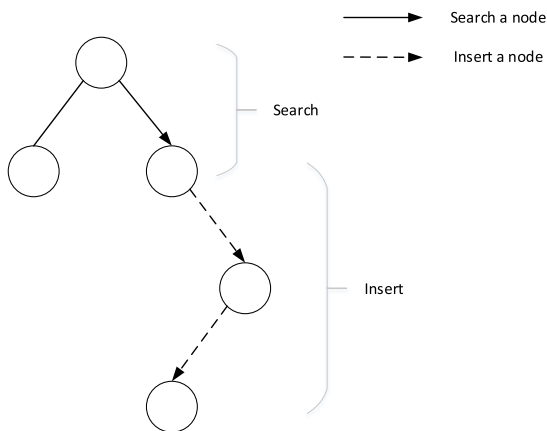


FIGURE 3. Construction diagram.

so all name components need to be inserted. However, partial insertion is quite the opposite. A prefix of the inserting name exists in PHT, so only the remaining components of the name

**Algorithm 2** Calculation and Assignment

```

Input: Root node  $R$ 
Output: None
Initialize component queue  $Q$ 
Initialize pointer  $p$  pointing to  $R$ 
Enqueue  $R$  into  $Q$ 
while  $Q \neq \emptyset$  do
    Dequeue a node  $n$  from  $Q$ 
    for all child node  $ch \in n.children$  do
        Put  $ch.count$  into a set  $freqSet$ 
    end for
     $lenSet = LengthCalculation(freqSet)$ 
     $codeSet = ParadigmHuffmanEncoding(lenSet)$ 
    for all code  $c \in codeSet$  do
        Assign  $c$  to child node correspondingly
    end for
    for all child node  $ch \in n.children$  do
        Enqueue  $ch$  into  $Q$ 
    end for
end while
    
```

For a traversal process, it is firstly distinguished whether the node is a leaf node. If it is not, traverse will continue; if it is, all child nodes of the current node are traversed to obtain the *count* field in each child node and store them as frequency set. The frequency set, length set, and codes correspond to the child nodes through the subscript. The length of a code can be calculated according to the frequency set and stored in the length set. In the meantime, codes can be calculated according to the length set. Finally, the codes are sequentially assigned to the code field for all child nodes according to the subscript. After traversing all the nodes in PHT, each node corresponding to a name component has its code, whose length is much shorter than that of the original name component.

### C. CODE TRANSITION

Because the codes of name components store at PHT nodes, we need to traverse and record every root-leaf route to acquire the intact encoded names, transitioning from original name to code. The traversal method adopted by this algorithm is Depth First Search (DFS) for accessing the leaf node as soon as possible. In the meantime, we choose the stack to record each root-leaf route to ensure the integrity of each encoded name with a common prefix.

---

#### Algorithm 3 Code Transition

---

**Input:** Root node  $R$

**Output:** Name code  $N$

Initialize stack  $K$

Push  $R$  into  $K$

**while**  $K \neq \emptyset$  **do**

    Pop a node  $n$  from  $K$

**for all** child node  $c \in n.children$  **do**

        CodeTransition( $c$ )

**end for**

**end while**

Print route

---

The code transition process is shown in Algorithm 3. The algorithm runs a DFS operation on the entire PHT, where the DFS operation is implemented recursively. A node is pushed into the stack in a code transition process, which records each root-leaf route, and the route of its child nodes is recursively output. The recursive terminal condition is that a leaf node is addressed. The top element of the stack, which is the leaf node that has been output, is deleted when the recursion terminates. When addressing the leaf node, the route is output according to the stack. All elements in the stack are popped and saved to an array, but the first item in the array is the original top of the stack, which is the last name component, so the array needs to be inverted to ensure the order of a name. Finally, each item in the array is concatenated into a string, thus composing an encoded name. After completing one transition, the algorithm will proceed to the next recursive operation.

## V. COMPACTTRIE DESIGN AND LOOKUP ALGORITHM

According to the encoding algorithm in IV, the name components are encoded by the paradigm Huffman algorithm, and each name component also satisfies the characteristics of paradigm Huffman codes. Therefore, the features of paradigm Huffman codes can be considered when designing the lookup algorithm for better performance on memory footprint.

### A. COMPACTTRIE DATA STRUCTURE

The data structure of the lookup algorithm is based on the dictionary tree. Fig. 4 shows the data structure inside a CompactTrie node. The *base* and *offset* fields store the encoding and decoding information of the paradigm Huffman algorithm; the *output\_flag* field represents the forwarding information corresponding to a prefix, and if it does not exist, the default value of this field is 0; the *children* field represents the set of child nodes in current node; the *len\_max* and *child\_max* fields represent the maximum range of the *base/offset* fields and the *children* field, respectively. There is no need for storing the encoded name component because combining the *base*, *offset* and *children* fields by some rules can acquire the original name component. The function and usage of each field will be elaborated in the following part.

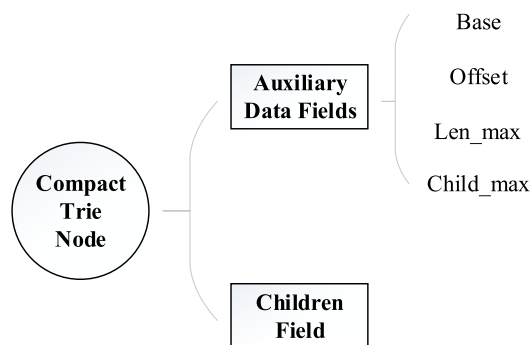


FIGURE 4. Compacttrie node diagram.

*Base, offset:* The *base* field represents the smallest code in the set of codes of different lengths, and the *offset* field represents the relative position of the smallest code in the entire code set, both of which are accessed by the code length. At the same time, these two fields store the information of child nodes, so matching a component at a certain level means finding the position of a node at the next level. With *base* and *offset* field, the original position of a child node corresponding to the next name component can be calculated based on the principles in Section III. In other words, decoding the code of a name component means a search on FIB entries in lookup algorithms.

*Len\_max, child\_max:* Considering that the fields in CompactTrie nodes need to be adjusted dynamically and updated frequently, the *base*, *offset*, and *children* fields are all

constructed as dynamic lists. When added or deleted, the list is expanded or compressed, so two parameters are needed to represent the size of dynamic lists for adjustment and update. As can be seen from Section III, the size of *base* and *offset* field are the same, so the *len\_max* field can be used by them together. The *child\_max* field represents the size of *children* field. The fields presented above are called auxiliary data fields.

### B. INSERTION

The insertion operation can be split into two parts: the insertion of CompactTrie nodes and the update of fields inside the nodes. The insertion of a node is relatively simple and similar to the construction operation in Section IV, as shown in Algorithm 4. When inserting a name, there are two types of insertion: full insertion and partial insertion. Full insertion means no prefix of the inserting name exists in CompactTrie. Thus, a list of new CompactTrie nodes will be created, and each component will be inserted; partial insertion means a prefix of the inserting name already exists in CompactTrie. At this point, the action is to update the fields of nodes in the prefix and insert the suffix of that name, specifically, create a list of new CompactTrie nodes corresponding to the name components in that suffix. When all name components have been inserted, this insertion operation ends.

---

#### Algorithm 4 Insertion

---

**Input:** Name  $N = \{c_i | i = 1, 2, \dots, L\}$ , root node  $R$

**Output:** None

```

Initialize a node pointer  $p$  pointing to  $R$ 
for all name component  $c \in N$  do
  Calculate the length  $l$  of  $c$ :  $l = \text{CalculateLength}(c)$ 
  Calculate the subscript  $i$  of  $c$ :
   $i = \text{CalculateSubscript}(l)$ 
  Adjust the auxiliary and children fields of current node:
   $\text{Adjust}(p)$ 
  if  $p.\text{children}_i = \emptyset$  then
    Initialize a new node  $p.\text{children}_i$ 
  end if
   $p = p.\text{children}_i$ 
end for

```

---

Since each node in CompactTrie contains the information of its child nodes, when a certain level of components is inserted, the information of the corresponding parent node will be updated. The update of fields inside a node only involves four auxiliary data fields and *children* field which represents the set of child nodes. The update of auxiliary data fields and *children* field will be presented below.

#### 1) AUXILIARY DATA FIELDS

When updating the auxiliary data fields, two values will be calculated: the length of an encoded component and the position of a child node according to Equation (11). First, the update condition of the *len\_max* field is that the length of an encoded component is larger than the *len\_max* field of

the current node, and if the condition is met, the *len\_max* field will be replaced by this value.

Updates of *base* and *offset* depend on the length of an encoded component which is defined as *len*, and its unit is bit. On account of dynamic updates for auxiliary data fields, there will be four states:

- 1) The *base* and *offset* fields of current *len* do not exist, so the *base* and *offset* fields need to be updated;
- 2) The current encoded component is smaller than the *base* value with same *len*, so the *base* and *offset* fields need to be updated;
- 3) The current encoded component is larger than every value with same *len*, so the *offset* field needs to be updated;
- 4) The *base* and *offset* fields do not need to be updated;

Considering the circumstances presented above, we define a variable *len\_offset* to record the distance that data in *offset* and *children* fields need to be moved. The initial value of *len\_offset* is zero, and its value are determined when adjusting the *base* and *offset* fields. For the first three cases, the core issue is the adjustment of *base* and *offset* fields, and *len\_offset* represents the scale of the expansion when the *children* field needs to expand. Since the original *children* field is compact between codes with different lengths, adjusting the *base* field and the *offset* field is necessary when a new child node is inserted.

Under the first circumstance, as shown in Fig. 5, the *base* and *offset* fields in a node of specific length are null, so the *base* of that specific length will be set as the code value. As for the update of *offset*, owing to the characteristics of paradigm Huffman encoding algorithm [32], the *offset* of that specific length is set to the position of the first code in the next bunch of *offset* values with longer length. In the meantime, *len\_offset* will be set as 1, because the distance of adjacent code with different lengths is 1.

Under the second circumstance, as shown in Fig. 5b, the code of the current component is the first and smallest in the bunch of values with the same length, so the original *base* of specific length will be replaced by the current code. Besides, the *offset* does not need to be updated because the position of the first code of the specific length does not change. Then the *len\_offset* is set to the original *base* with specific length minus code of current component.

Under the third circumstance, as shown in Fig. 5c, the code is the biggest among *base* of specific length, so both the *base* and *offset* fields do not need to be updated. The only thing to do is set the *len\_offset* as the code of current component minus the largest *base* value of specific length.

Because codes with the same length are sparse, given the last circumstance, the current component can be inserted directly without adjusting *base* and *offset* fields. In this case, *len\_offset* remains the same and *children* field do not need to expand.

Owing to the movement of child nodes under the first three circumstances, every *offset* value whose length is larger than the current specific length needs to be added *len\_offset*.



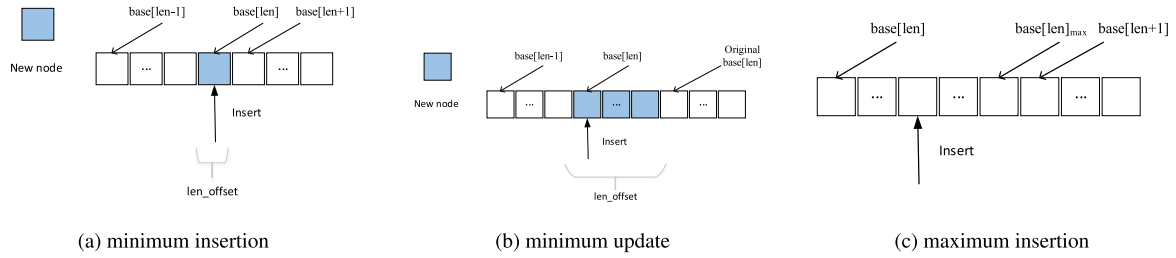


FIGURE 5. Auxiliary fields adjustment in insertion.

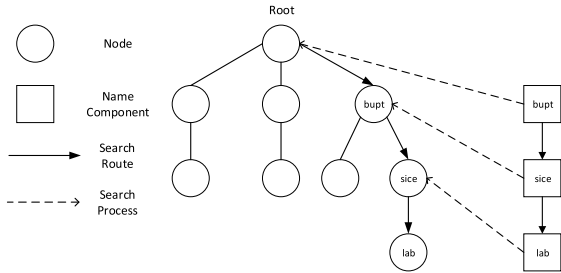


FIGURE 6. Lookup process.

## 2) CHILDREN FIELD

Before adjusting the children field, the position of the child node, which corresponds to the inserting name component, has been calculated, and there are two results. The first one is that the result of the calculation is larger than the *child\_max* field which is the maximum position of this node. When it happens, the *child\_max* field is set to the position of current node and *children* field need to expand until its size equals to *child\_max*. Another situation is that the *children* field does not need to expand. The variable *len\_offset* is the distance that all the nodes whose subscripts are larger than that of the current node need to move. After this action, the current node will be inserted to its position calculated above.

## C. LOOKUP

The first input parameter of the lookup algorithm is the root node of CompactTrie, which represents the entrance of it; the second parameter is the name that needs to be searched.

First, the lookup algorithm declares a node pointer, responsible for tracking the path of a lookup. The entire algorithm maintains a loop and searches a matching component cyclically. In each loop, the length of the current encoded component is first obtained, facilitating the calculation of the child node subscript in the next step. Then, according to Equation (11), the calculation for the subscript of a child node is performed, and then this child node is accessed. If the child node exists, the node pointer moves to this child node and the search for the next component will continue; if the child node does not exist, according to the rule of LPM, the algorithm will directly return the forwarding information in the current node. Fig. 6 shows an example of lookup operation.

During each loop, the algorithm will calculate two parameters: the length of an encoded component and the subscript

## Algorithm 5 Lookup

**Input:** Name  $N = \{c_i | i = 1, 2, \dots, L\}$ , root node  $R$

**Output:** Forwarding information  $F_w$

Initialize a node pointer  $p$  pointing to  $R$

**for all** name component  $c \in N$  **do**

    Calculate the length  $l$  of  $c$ :  $l = \text{CalculateLength}(c)$

    Calculate the subscript  $i$  of  $c$ :

$i = \text{CalculateSubscript}(l)$

**if**  $p.\text{children}_i = \emptyset$  **then**

        Output  $p.\text{fwd}$  as  $F_w$

**else**

$p = p.\text{children}_i$

**end if**

**end for**

Output  $p.\text{fwd}$  as  $F_w$

of a child node. If these two parameters exceed their ranges, the lookup fails. Meanwhile, if the loop ends at the root node, the lookup fails too. From the procedure of Algorithm 5, we can figure that: the computational complexity inside the node is  $O(1)$ , and outside the node depends on the height of CompactTrie, which means the overall computational complexity is  $O(h)$ , where  $h$  represents the height of CompactTrie.

## D. DELETION

Deletion also requires two levels of operation: the deletion of nodes in the global CompactTrie and the update of auxiliary data fields and *children* field inside the nodes. Since the deletion of a name depends on the premise that the name exists in the table, the recursive method is used to delete a name in the CompactTrie. The current name component is searched for each recursion, as described in Section V-C. The recursive termination condition is that all name components are found.

The deletion algorithm is shown in Algorithm 6. The global deletion of a node is implemented by the recursive method, and the position of the next name component is searched in each recursion. If the position of the next name component exists, the child node in this position is recursively deleted; if it does not exist, this name does not exist in the table, and the deletion operation fails.

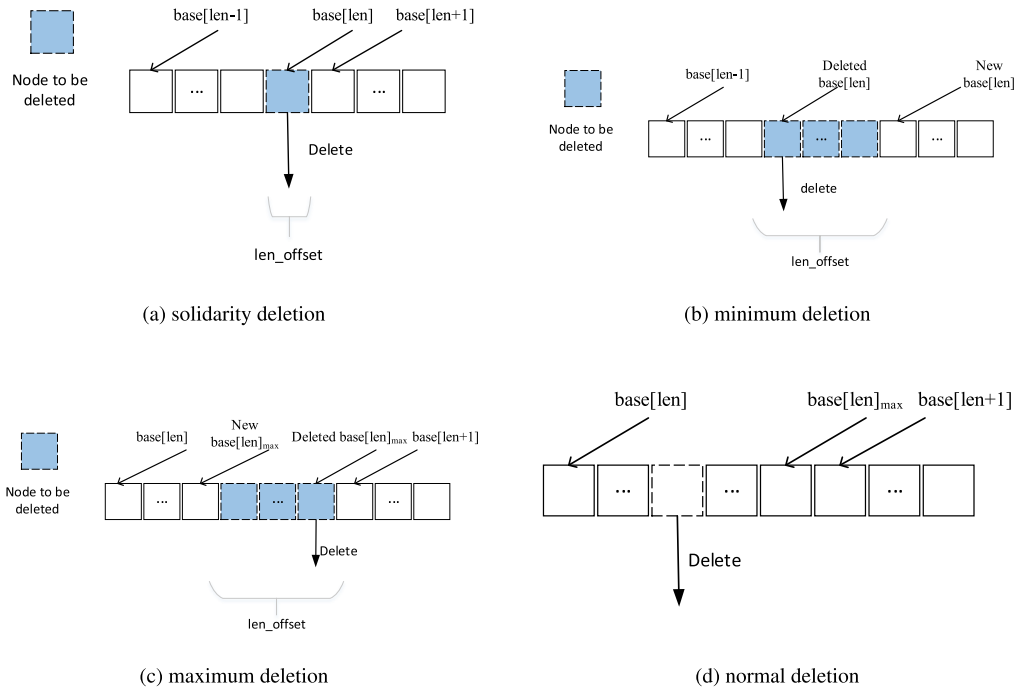


FIGURE 7. Auxiliary fields adjustment in deletion.

**Algorithm 6** Deletion

**Input:** Name  $N = \{c_i | i = 1, 2, \dots, L\}$ , root node  $R$

**Output:** None

```

Initialize a node pointer  $p$  pointing to  $R$ 
for all name component  $c \in N$  do
    Calculate the length  $l$  of  $c$ :  $l = CalculateLength(c)$ 
    Calculate the subscript  $i$  of  $c$ :
     $i = CalculateSubscript(l)$ 
    if  $p.children_i \neq \emptyset$  then
        Recursively delete  $p.children_i$ 
        Adjust the auxiliary and children fields of current
        node:  $Adjust(p)$ 
    else
        Deletion fails
    end if
end for
Recursion returns
    
```

Each recursion will return a value that represents the number of child nodes of the current node. When deleting the current node, the number of child nodes is acquired. If it is not 1, it means that the prefix of the current name component is also the prefix of other names, and these nodes cannot be deleted. When processing the leaf node, the search is successful, and the deletion can be started.

After acquiring the number of child nodes, updating auxiliary data fields and the child node field will commence. Deletion operation compresses *children* field, so we also

define a variable to maintain the distance that data need to move for *children* field, named as *len\_offset*. The initial value of *len\_offset* is zero, and it is determined when adjusting *base* and *offset* fields. The update of auxiliary data fields and child node field will be presented below.

1) AUXILIARY DATA FIELDS

The first auxiliary data field that needs adjusting is *len\_max*. The conditions of adjusting the *len\_max* field are harsh, including: 1) the deleted name component has the maximum length among all components at the same level; 2) the deleted name component has the maximum subscript; 3) with the maximum length, there is only one name component, and it is the deleted one. The new *len\_max* will be set as the maximum length after deleting the current name component. As for the *base* and *offset* fields, there are three circumstances when updating them:

- 1) Delete the first component of a specific length, which is the minimum code. The *base*, *offset* and *children* fields need to be adjusted;
- 2) Delete the last component of a specific length, which is the maximum code. The *children* field needs to be adjusted;
- 3) All fields do not need to be updated;

Under the first circumstance, as shown in Fig. 7a, if the deleted name component is the only one of specific length, the *base* and *offset* fields of such length will be eliminated and *len\_offset* will be set as 1. Otherwise, as shown in Fig. 7b, the *base* of specific length is set as the first value that is

TABLE 1. Dataset specification.

Dataset/ $10^4$	1	2	3	4	5	6	7	8	9	10
Average length of names	68.42	29.79	47.55	48.85	58.73	55.11	51.18	49.23	50.74	69.69
Average length of components	13.81	5.37	10.14	8.92	16.68	12.54	11.75	11.47	12.22	20.33
Average number of components	4.82	4.71	4.94	5.12	3.85	4.32	4.17	4.19	4.18	3.43

bigger than the deleted one, the *offset* keeps constant, and the *len\_offset* is set as the subtraction of new *base* value and original *base* value of specific length.

Under the second circumstance, as shown in Fig. 7c, the *base* and *offset* fields do not need to be updated, and the *len\_offset* will be set as the subtraction of deleted code and the first code that is smaller than the deleted one.

For the last circumstance, as shown in Fig. 7d, all fields do not need to be updated. After adjusting all the fields, all the *offset* values whose lengths are larger than the current length need to subtract the *len\_offset* to keep the compactness of CompactTrie structure.

## 2) CHILDREN FIELD

The movement of *children* field depends on whether the *len\_offset* is modified. If *len\_offset* remains zero, there is no operation of *children* field. Otherwise, all nodes after the deleted node need to move forward by *len\_offset*, and *child\_max* need to subtract the same value too.

## VI. UPDATE POLICY

On account of frequent content updating, an update policy is needed to guarantee the accuracy of name codes which is calculated by the PHT encoding algorithm, as shown in Fig. 8. When a Content Provider adds or deletes a content, all name codes will be recalculated by the PHT encoding algorithm, plus the added or minus the deleted content names. Considering the independence between different levels of name components, adding or deleting a content name does not impact other name codes, so the content can be updated by its own Content Provider without notifying other Content Providers. After updating the name codes, the Content Provider will encapsulate the new name codes into an update message and deliver it to adjacent nodes, in which the FIB entries will be updated according to the new name codes. Meanwhile, each node will deliver this update message to its adjacent downstream nodes to renew their own FIB entries until they finish updating. The update message will carry a *timestamp* field to avoid duplicate updates.

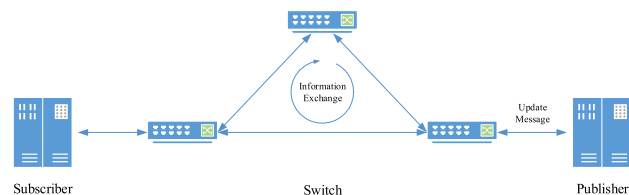


FIGURE 8. Update policy.

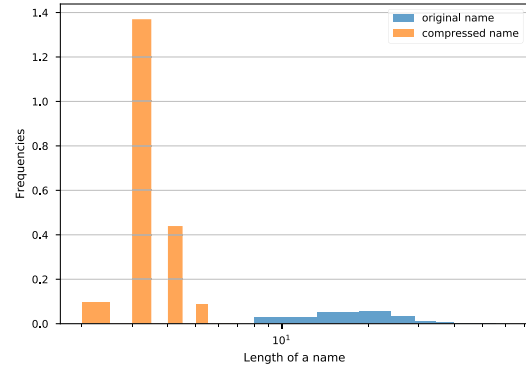


FIGURE 9. Comparison of original and compressed names.

## VII. EVALUATION

### A. EXPERIMENT SETUP

**Dataset:** we use a database provided by Sogou Labs [34], including millions of Internet URLs, to test the performance of our design. Moreover, in order to approach the format of NDN names according to NDN naming conventions, we exploit an NDN name generator called NameGen [35], to convert the URLs into appropriate NDN names. NameGen is a name generating tool by using a Markov model for learning real datasets, and it can generate a huge dataset having the same characteristics of a given dataset. The names generated by NameGen are hierarchical names with a common prefix, which satisfies our demands on the lookup algorithm. We generated many datasets, including different number of NDN names, to test the adaptability and flexibility of algorithms. The specification of these datasets are shown in Table 1.

**Hardware:** our test runs on a virtual machine with a 3.40 GHz Intel i5-7500 4-core CPU and 4 GB of RAM. All 4 cores share 1 MB of L2 cache and 6 MB of L3 cache. All algorithms are implemented by the C programming language. The testing method counts the completion time with different datasets and calculates the memory footprint of each lookup algorithm.

### B. COMPRESSION PERFORMANCE

The compression performance is mainly reflected on the ratio of encoded name size to original name size. Specifically, we focus on the Average Name Compression Rate, denoted as ANCR, and the Distribution of Component Compression Rate, denoted as DCCR. These two indicators are used to illustrate how efficient our design is from a macro and micro perspective, respectively. Fig. 9 shows the ANCR curves

with different databases, where the x-axis marks the number of name components in different databases, and the y-axis marks the ANCR. In Fig. 9, the Average Compression Rate is derived from Equation (12):

$$\begin{aligned}
 ANCR &= len(name) \times 8 \div len(code) \\
 CCR &= len(component) \times 8 \div len(code) \quad (12)
 \end{aligned}$$

The granularity of measuring the size of both original and encoded names is bit level. For a character in an original NDN name, it is typically 8-bit long encoded by ASCII and longer by other code systems, which means the memory footprint of an NDN name is at least 1 Bytes multiplied by the length of this name; for an encoded name which is a code generated by our design, the memory footprint of this code is its own size, the bit level. Simultaneously, if the code is not 8-bit aligned, it will be padded zeros before the Most Significant Bit (MSB) to guarantee the efficiency of memories.

In Fig. 9, we randomly choose a dataset generated above to find out the difference of original and compressed names, thus evaluating the performance of PHT encoding algorithm. From Fig. 9, the length of original names usually exceeds 10, but that of compressed names belongs to around 5, which shows that our design can compress a content name into a code with a much smaller size efficiently. A conclusion can also be speculated that the length of compressed names is approximately equal to the number of components.

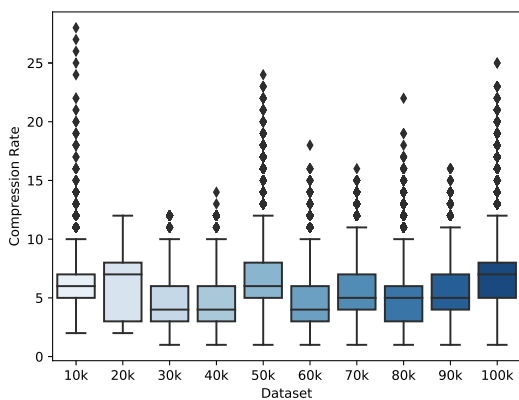


FIGURE 10. Distribution of component compression rate.

Fig. 10 shows the statistical features with all datasets by box plot, where the x-axis marks the dataset size and the y-axis marks the component compression rate. In Fig. 10, the compression rate of each component is derived from PHT structure, where both original and encoded names are stored, and it is easy to calculate the component compression rate.

In Fig. 10, although the datasets are generated randomly, and the sizes of them are distinct, the median compression rates of different datasets are around 5, and the 25th percentile is about 3, which means the majority of content name components are compressed efficiently. Compared with Fig. 9, this one shows the compression performance from a macro

aspect to acquire the global perspective of the PHT encoding algorithm. Consequently, Fig. 10 can show the stability and flexibility of our PHT Encoding algorithm, no matter what datasets are used.

### C. LOOKUP PERFORMANCE

There are usually two indices to measure the lookup performance: delay of one lookup operation and memory footprint of FIB entries. Both of them are quantified for time and space complexity. In this part, we evaluate our CompactTrie design, compared with the other two hash-based lookup algorithms [15], [16] and two trie-based methods [13], [29], to examine and analyze the performance of CompactTrie. Fig.11 shows the comparison of lookup delay between our design and the other four algorithms, where the x-axis marks the number of test datasets and the y-axis marks the average lookup delay whose unit is microsecond. We use 10,000 names selected randomly from the test datasets to acquire multiple samples of time overhead for each lookup operation and calculate the average value of those samples to obtain the average delay. In Fig. 11, the average lookup delay of CompactTrie lookup operation is lower than two hash-based algorithms and pNPT method and close to NameTrie. Meanwhile, the average delay of DosResistant proliferates with the growth of dataset size, whereas the other four methods, including ours, grow more smoothly. In general, Our design can keep the delay of one lookup operation relatively low and vary gently as the size of the dataset grows.

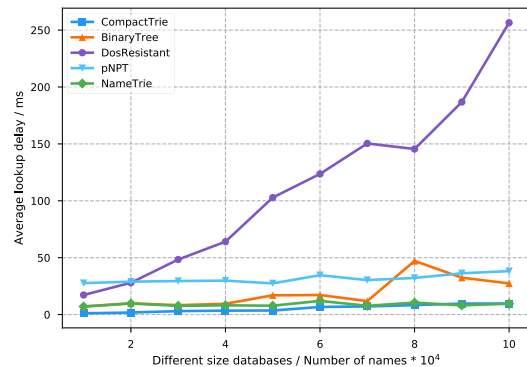


FIGURE 11. Comparison of lookup delay.

The same conclusion can be obtained by analyzing the principle of each algorithm. The time complexity of our design consists of two parts: global search of each name component and local calculation of next name component's position, as shown in Table 2. Considering the time complexity of the local calculation is  $O(o + c - b)$  as described in Section V thoroughly and all those three parameters are constant, the full cost of our design is the global search, and its time complexity is  $O(h)$ . In other words, the frequency of memory access relies on the number of name components. At the same time, neither does our design involve comparing



TABLE 2. Complexity analysis of compacttrie lookup.

	Definition
$b$	Base
$o$	Offset
$h$	Trie Depth
Time Complexity	
Inner-node Calculation	$O(o + c - b)$
Global Search	$O(h)$
Overall	$O(h * (o + c - b))$

characters, nor does it store the corresponding component in the trie node, which will also reduce the lookup delay. For the other two hash-based algorithms, although the time complexity of hash table lookup is  $O(1)$ , an extra structure is needed to satisfy the demand of LPM, thus requiring the extra cost for each lookup. Comparing characters is also performed in these hash-based algorithms for each lookup operation, so the total time complexity of hash-based algorithms is relatively high. Meanwhile, the time complexity of the other two trie-based algorithms relatively low because these two methods take great effort to optimize the structure and minimize the in-node calculation as our work did. For example, minASCII in NameTrie exploits unused ASCII code to represent common NDN symbols, and priority trie in p-NPT replaces blank nodes with leaf nodes by some rules.

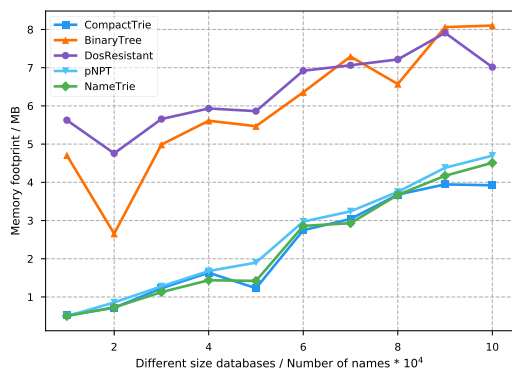


FIGURE 12. Comparison of memory footprint.

Fig.12 shows the comparison of memory footprint among CompactTrie and the other four algorithms. The memory footprint of the DOS-resistant and Binary Tree methods is bigger than the other three algorithms because the size of the hash table needs to extend when a name is inserted to the data structure or the size of datasets grows. Especially for DOS-resistant lookup algorithm, when the component number of an inserted name is larger than existing ones inside the hash table, or a virtual prefix needs to be generated, the size of the hash table dilates. In other words, the memory is pre-allocated in hash-based algorithms, and it has to be

expanded when the existing memory is insufficient, which is wasteful and not flexible. For our CompactTrie design and other two trie-based methods, the size of memory footprint is relatively low, due to the characteristics of the trie structure. Specifically, NameTrie uses the minASCII encoding mechanism, and pNPT uses the priority trie to reduce the memory footprint. As for our CompactTrie design, the auxiliary data fields and children field in CompactTrie nodes can also help lower the memory footprint. In conclusion, trie-based algorithms can provide a lower memory footprint, and our design is efficient at this point.

Our design is better than the other two hash-based lookup algorithms on both lookup delay and memory footprint and other trie-based methods on memory footprint through two comparisons. Therefore, our CompactTrie design accelerates the forwarding of content and reduces the size of name FIB entries. In summary, our design takes both time and space complexity into account and guarantees them at a low level.

VIII. CONCLUSION

This paper mainly studies an NDN name encoding algorithm and a relative optimized lookup method with low time and space complexity. The traditional NDN lookup algorithm focuses on the forwarding and lookup performance, which sacrifices space in exchange for time. Still, our strategy can improve the performance both on lookup delay and memory usage. The size of NDN FIB entries is much larger than that of IP, so the memory footprint problem will be more severe. Considering the validity of data transmission, the PHT and CompactTrie structure designed in this paper exploits an encoding mechanism and fundamentally reduces the size of FIB entries while still maintaining a high performance relatively.

After introducing the encoding mechanism, not only the size of FIB entries is reduced, but also the high scalability and flexibility are kept. At the same time, the encoded content names all have consistent characteristics, regardless of different naming conventions. Given that the current bottleneck of NDN is the soaring size of FIB entries, our design solves this problem and provides high forwarding performance, which allows the NDN architecture to evolve more rapidly. It is also hopeful to replace the traditional TCP/IP networks by providing a more flexible and efficient forwarding engine for current network applications.

However, our design still needs to be improved from many aspects and treat as our future work. First, CompactTrie still exploits pointers as the connection between trie nodes and the pointer is wasteful for memory and hard to be implemented. Using another data structure to store edge information in the trie is a good idea, and we would design and implement it in the future. Second, our whole design still needs a better-established update policy to adapt to the fast update of content names. In this paper, we focus on the design of name encoding and the relative lookup algorithm, so the update policy is quite easy. Consequently, a perfect update policy will be regarded as our future work.

## REFERENCES

- [1] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. C. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, Jul. 2014.
- [2] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos, "A survey of information-centric networking research," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 2, pp. 1024–1049, May 2014.
- [3] G. Révçri, J. Tapolcai, A. Körösi, A. Majdán, and Z. Heszberger, "Compressing IP forwarding tables: Towards entropy bounds and beyond," *ACM SIGCOMM Computer Commun. Rev.*, vol. 43, no. 4, pp. 111–122, 2013.
- [4] H. Zhian, M. Bayat, M. Amiri, and M. Sabaei, "Parallel processing priority trie-based IP lookup approach," in *Proc. 7th Int. Symp. Telecommun. (IST)*, Sep. 2014, pp. 635–640.
- [5] M. Zec, L. Rizzo, and M. Mikuc, "DXR: Towards a billion routing lookups per second in software," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 5, pp. 29–36, Sep. 2012.
- [6] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, no. 3, pp. 379–423, Jul./Oct. 1948.
- [7] E. S. Schwartz and B. Kallick, "Generating a canonical prefix encoding," *Commun. ACM*, vol. 7, no. 3, pp. 166–169, Mar. 1964.
- [8] Z. Li, Y. Xu, B. Zhang, L. Yan, and K. Liu, "Packet forwarding in named data networking requirements and survey of solutions," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1950–1987, 2nd Quart., 2019.
- [9] Y. Wang, H. Dai, J. Jiang, K. He, W. Meng, and B. Liu, "Parallel name lookup for named data networking," in *Proc. IEEE Global Telecommun. Conf. (GLOBECOM)*, Dec. 2011, pp. 1–5.
- [10] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, and Y. Chen, "Scalable name lookup in NDN using effective name component encoding," in *Proc. IEEE 32nd Int. Conf. Distrib. Comput. Syst.*, Jun. 2012, pp. 688–697.
- [11] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, and H. Wu, "Wire speed name lookup: A GPU-based approach," in *Proc. 10th Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 199–212.
- [12] T. Song, H. Yuan, P. Crowley, and B. Zhang, "Scalable name-based packet forwarding: From millions to billions," in *Proc. 2nd ACM Conf. Inf.-Centric Netw.*, Sep. 2015, pp. 19–28.
- [13] C. Ghasemi, H. Yousefi, K. G. Shin, and B. Zhang, "A fast and memory-efficient trie structure for name-based packet forwarding," in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, Sep. 2018, pp. 302–312.
- [14] O. Karrakchou, N. Samaan, and A. Karmouch, "FCTrees: A front-coded family of compressed tree-based FIB structures for NDN routers," *IEEE Trans. Netw. Service Manage.*, vol. 17, no. 2, pp. 1167–1180, Jun. 2020.
- [15] W. So, A. Narayanan, and D. Oran, "Named data networking on a router: Fast and DoS-resistant forwarding with hash tables," in *Proc. Archit. Netw. Commun. Syst.*, Oct. 2013, pp. 215–226.
- [16] H. Yuan and P. Crowley, "Reliably scalable name prefix lookup," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, May 2015, pp. 111–121.
- [17] R. Shubbar and M. Ahmadi, "Efficient name matching based on a fast two-dimensional filter in named data networking," *Int. J. Parallel, Emergent Distrib. Syst.*, vol. 34, no. 2, pp. 203–221, Mar. 2019.
- [18] Y. Wang, Z. Qi, H. Dai, H. Wu, K. Lei, and B. Liu, "Statistical optimal hash-based longest prefix match," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, May 2017, pp. 153–164.
- [19] H. Khelifi, S. Luo, B. Nour, and H. Mounsla, "A name-to-hash encoding scheme for vehicular named data networks," in *Proc. 15th Int. Wireless Commun. Mobile Comput. Conf. (IWCMC)*, Jun. 2019, pp. 603–608.
- [20] J. Hu and H. Li, "A composite structure for fast name prefix lookup," *Frontiers ICT*, vol. 6, p. 15, Aug. 2019.
- [21] S. Feng, M. Zhang, R. Zheng, and Q. Wu, "A fast name lookup method in NDN based on hash coding," in *Proc. 3rd Int. Conf. Mechatronics Ind. Inform. (ICMII)*. Atlantis Press, 2015, pp. 575–580.
- [22] D. Perino, M. Varvello, L. Linguaglossa, R. Laufer, and R. Boislaigue, "Caesar: A content router for high-speed forwarding on content names," in *Proc. 10th ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, 2014, pp. 137–148.
- [23] Z. Li, Y. Xu, K. Liu, X. Wang, and D. Liu, "5G with B-MaFIB based named data networking," *IEEE Access*, vol. 6, pp. 30501–30507, 2018.
- [24] H. Dai, J. Lu, Y. Wang, T. Pan, and B. Liu, "BFAST: High-speed and memory-efficient approach for NDN forwarding engine," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 1235–1248, Apr. 2016.
- [25] K. Chan, B. Ko, S. Mastorakis, A. Afanasyev, and L. Zhang, "Fuzzy interest forwarding," in *Proc. Asian Internet Eng. Conf.*, Nov. 2017, pp. 31–37.
- [26] J. Kim, M.-C. Ko, M. S. Shin, and J. Kim, "Scalable name lookup for NDN using hierarchical hashing and patricia trie," *Appl. Sci.*, vol. 10, no. 3, p. 1023, Feb. 2020.
- [27] B. Nour, K. Sharif, F. Li, H. Mounsla, and Y. Liu, "A unified hybrid information-centric naming scheme for IoT applications," *Comput. Commun.*, vol. 150, pp. 103–114, Jan. 2020.
- [28] T. Liang, J. Shi, and B. Zhang, "On the prefix granularity problem in ndn adaptive forwarding," in *Proc. 7th ACM Conf. Inf.-Centric Netw.*, 2020, pp. 41–51.
- [29] J. Seo and H. Lim, "Bitmap-based priority-NPT for packet forwarding at named data network," *Comput. Commun.*, vol. 130, pp. 101–112, Oct. 2018.
- [30] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proc. Conf. Comput. Commun., 17th Annu. Joint Conf. IEEE Comput. Commun. Societies Gateway 21st Century (IEEE INFOCOM)*, vol. 3, Mar. 1998, pp. 1240–1247.
- [31] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. Inst. Radio Eng.*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.
- [32] A. Moffat and A. Turpin, "On the implementation of minimum redundancy prefix codes," *IEEE Trans. Commun.*, vol. 45, no. 10, pp. 1200–1207, Oct. 1997.
- [33] A. Moffat and J. Katajainen, "In-place calculation of minimum-redundancy codes," in *Proc. Workshop Algorithms Data Struct.* Springer, 1995, pp. 393–402.
- [34] *Sogou T-Rank*. Accessed: Oct. 9, 2019. [Online]. Available: <http://www.sogou.com/labs/resource/t-rank.php>
- [35] C. Ghasemi, H. Yousefi, K. G. Shin, and B. Zhang, "On the granularity of trie-based data structures for name lookups and updates," *IEEE/ACM Trans. Netw.*, vol. 27, no. 2, pp. 777–789, Apr. 2019.
- [36] M. Fukushima, A. Tagami, and T. Hasegawa, "Efficiently looking up non-aggregatable name prefixes by reducing prefix seeking," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPs)*, Apr. 2013, pp. 340–344.
- [37] H. Yuan, P. Crowley, and T. Song, "Enhancing scalable name-based forwarding," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, May 2017, pp. 60–69.



**TIANYUAN NIU** received the B.S. degree in information and communication engineering from Beijing University of Posts and Telecommunications, Beijing, China, where he is currently pursuing the Ph.D. degree. His research interests include future network architecture, Named Data Networking, cache mechanism in future network device, and P4.



**FAN YANG** received the Ph.D. degree from Beijing University of Posts and Telecommunications. He is currently a Lecturer with Beijing University of Posts and Telecommunications. He has participated in the National 973 Planning Project "Research on Service-Oriented Future Internet Architecture and Mechanism," the National 863 Project "Service-Oriented SDN Architecture and Key Technology Research" and "Future Network System and Structural Research for Service-Oriented Resource Intelligent Scheduling" funded by the National Key Laboratory. He presides over German Telecom and Huawei's next-generation IP routing equipment research, Huawei ultra-high-speed network processor search algorithm research, and other enterprise projects. He has published more than 30 SCI/IEI articles and national invention patents. His research interests include software defined network and high-performance routing and switching technologies. He has won the 2013 National Technology Contribution Individual Award, the Huawei Golden Network Award, and the China Communication Society Science and Technology First Prize.

...