frontiers in ICT

ORIGINAL RESEARCH
published: 02 August 2019
doi: 10.3389/fict.2019.00015

A Composite Structure for Fast Name Prefix Lookup

Jiawei Hu and Hui Li**

School of Electronic and Computer Engineering, Shenzhen Graduate School, Peking University, Shenzhen, China

Name-based forwarding plane is a critical but challenging component for Named Data Networking (NDN), where a hash table is an appealing candidate for data structure utilized in FIB on the benefit of its fast lookup speed. However, the hash table is flawed that it does not naturally support the longest-prefix-matching (LPM) algorithm for name-based forwarding. To support LPM in the hash table, besides the linear lookup, random search (such as binary search) aims at increasing the lookup speed by reconstructing the FIB and optimizing the search path. We propose a composite data structure for random search based on the combination of a hash table and a trie; the latter is introduced to preserve the logical associations among names, so as to recycle memory and prevent the so-called backtracking problem, thus enhancing the lookup efficiency. Our experiment indicates the superiority of our scheme in lookup speed, the impact on memory consumption has also been evaluated.

Keywords: named data networking (NDN), random search, data structure, longest prefix matching (LPM), trie (prefix tree)

OPEN ACCESS

Edited by:
Pere Barlet-Ros, Universitat Politecnica de Catalunya, Spain

Reviewed by:
Yi Wang, Southern University of Science and Technology, China
Giorgos Dimopoulos, Independent Researcher, Barcelona, Spain

***Correspondence:**
Jiawei Hu
jiaweihu39@gmail.com
Hui Li
lih64@pkusz.edu.cn

Specialty section:
This article was submitted to Big Data Networks, a section of the journal Frontiers in ICT

Received: *09 February 2019*
Accepted: *04 July 2019*
Published: *02 August 2019*

Citation:
Hu J and Li H (2019) A Composite Structure for Fast Name Prefix Lookup. Front. ICT 6:15. doi: 10.3389/fict.2019.00015

1. INTRODUCTION

Named data networking (NDN; Zhang et al., 2010) is a networking paradigm concentrating on the content itself rather than its endpoints. Instead of their destination addresses, packets are forwarded based on hierarchically structured names like URL to facilitate multicasting and in-network caching. The transmission mechanism in NDN is consumer-driven and performed by two types of packets, namely, Interest and Data. An interest packet carries the name of the desired content and is sent to the network by a consumer for a request if the Interest encounters a producer node which has the requested content. A Data packet of this content is generated and follows the reverse path taken by the Interest to travel back to the requesting consumer.

In NDN, the Forwarding Information Base (FIB) stores the forwarding information for Interest packets, filling the analogous rule as with IP. Since the NDN FIB is keyed by hierarchically-structured content names, as in IP, the Interest packets are forwarded based on the longest prefix matching (LPM) results, with the requested names as the lookup keys.

The LPM methods implemented for IP forwarding have achieved a remarkable success. Benefitted from core concepts such as prefix expansion and bitmap representations (Doeringer et al., 1996; Degermark et al., 1997; Srinivasan and Varghese, 1998), for fixed length IPs, modern LPM scheme can handle Internet-scale rulesets efficiently within a few megabytes of memory. However, as the packet forwarding in NDN is much more complex compared to IP, traditional LPM approaches do not work well for name-based forwarding; namely, an efficient and scalable NDN forwarding scheme requires to settle the following challenges:

1. **Variable name formats.** Most traditional forwarding solutions are designed for IPs and have the complexity proportional to the length of prefixes. Unlike a fixed-length IP address, since an NDN name has unbounded length and more complex structure (So et al., 2012, 2013; Yuan et al., 2012), directly applying these schemes yields low performance.
2. **Large forwarding table size.** Referring to DNS works in So et al. (2013) estimated the scale of the name-based FIB, which is orders of magnitude larger than IP routing tables today. Without the assistance of the compression method, the NDN forwarding table can far exceed the capacity of the commodity device.
3. **Frequent update.** Besides the updates caused by network topology and routing policy changes, an NDN FIB requires to handle contents' publishing, deletion and in-network caching, making the update rate much higher than that in IP protocol.

As a result, forwarding plane design has become a critical but challenging work especially in vast scale networks. The evaluation criteria of FIB design contain operation speed, memory consumption, as well as the support for an intelligent and adaptive forwarding strategy in the future. Currently, trie (prefix tree) and hash table are two data structures which are widely used as an index for FIB:

Trie is an ordered tree-like structure which is usually used to handle string matching. Based on the benefits from its structural characteristics, trie can reduce memory consumption by aggregating redundant prefixes in names, and the LPM search can be naturally supported since trie stores the logic relations of the prefixes. However, trie has a drawback that its has low lookup efficiency, as the search speed is linearly relative to the expected value of trie depth which is associated with the unbounded name length.

Hash table is also an appealing candidate for FIB because of its advantages of fast lookup and simple implementation. However, hash table-based FIB schemes have the following two problems:

1. **Hash table consumes more memory than structures like a trie or a bloom filter**, as whole name string needs to be stored to handle hash collisions and ensure proper forwarding. Current implementations mainly focus on shortening name length to resolve this problem, such as replacing the name by its fingerprint, hash value, or encoded version (Yuan et al., 2012, 2017).
2. **Hash table does not naturally support the LPM search.** The simplest solution for this problem is linear search, in which HT lookup starts at the longest length of the name prefix and is iterated decreasingly by component granularity until the LPM is found. The other way refers to random search (Waldvogel et al., 1997) aimed at reducing the lookup time by rearranging the search order, such as 2-stage LPM (So et al., 2013) or binary search (Yuan and Crowley, 2015). In spite of the improvement to the search speed, most random search methods rely on the reconstruction of FIB and may lead to the so-called backtracking problem, causing a false negative, as well as the memory waste since many useless entries can not be timely removed.

This paper concentrates on the optimization for LPM approaches in NDN, improving the random search method for the hash-tabled FIB scheme, aiming at solving the so-called backtracking problem and the memory waste caused by outdated entries. Our scheme features (1) a reconstructed hash table with entries that can be classified into 3 types, which determine whether a backtracking process is required when a random search miss occurs; thus the false negative error can be prevented. (2) an auxiliary trie structure storing logic relations between names to dynamically modify entries' types upon table change, while the extra memory consumption of the trie is restricted. (3) corresponding operation algorithms for the composite structure above. This paper is organized as follows: section 2 introduces the background and related works. Our scheme is presented in section 3. The evaluation result is in section 4.

2. BACKGROUND AND RELATED WORKS

2.1. Requirement of NDN Forwarding Plane

In NDN, Data should be retrieved by a hierarchical tokenized name with the format like "/c1/c2/c3." As some parts of the name cannot be informed or inferred beforehead, the consumer sends Interests carrying only a prefix of the content name to the network, and any Data under this prefix can be returned by NDN protocol. e.g., An Interest with the name "/com/ndn/document/file01.txt" can be replied by any Data whose name is covered by this prefix, such as "/com/ndn/document/file01.txt/segment01." Thus, same as IP protocol, the name matching algorithm in NDN's Forwarding Information Base (**FIB**) is the Longest Prefix Matching (**LPM**), performed with the Interest name as the search key. Therefore, the FIB can find the most accurate forwarding information for the Interest.

In the meanwhile, NDN forwarding is challenging in scalable applications for several reasons: (1) an NDN name has unbounded variable length, making a lookup operation time-consuming; (2) The amount of NDN FIB entries is larger than that of IP by orders of magnitude; (3) NDN FIB has to be updated more frequently as a result of content publishing and deletion, making the forwarding plane design a difficult but essential task.

2.2. Hash Table-Based FIB Schemes

Due to the requirement of the frequent processing operation in NDN forwarding plane, hash table has become a potential candidate for FIB because of its advantage of fast lookup. To support the LPM matching algorithm, in a typical Interest forwarding scenario, the hash table (HT) lookup starts from the longest length of name prefix and decreases progressively by component granularity. The process iterates until it finds the LPM. However, this linear search method is undesirable not only for its under performance but also for security. Since a non-matching interest with the name of n components requires n times FIB lookup, NDN router is more vulnerable to DoS attacks where Interests with the long non-matching nonce name are injected to the network.

Methods have been proposed to reduce HT lookup times through the optimization of the searching path. The work in

So et al. (2013) introduces a 2-stage LPM scheme enhancing the DoS resistance. During the name insertion period, not only the name itself, but also a corresponding prefix with the certain short length (M component), which forms a virtual entry, is inserted into FIB. The search process starts from length M and either continues to proper prefixes (if not found) or restarts from a longer prefix to perform a linear search. Besides, works in Wang et al. (2013, 2014) optimize the 2-stage name lookup process by rearranging the search order according to the distribution of a prefix component number.

Besides the schemes mentioned above, works in Yuan and Crowley (2015) proposed a lookup method with the binary search of hash tables. With all shorter prefixes of names inserted as marker entries, the hash tables form a balanced binary search tree, in which each HT stores prefixes with a certain length. Regarding each tree node, the shorter prefixes are stored in the left subtree, while the longer prefixes are stored in the right one.

The works in Wang et al. (2017) applied a dynamic programming algorithm based on the statistic of prefix component number distribution, to optimize the binary search above. For each component number M, with a prior probability of a length M prefix to be matched, the dynamic programming can recursively calculate which length to be checked next so as to optimize the search path.

2.3. The Backtracking Problem

The aforementioned schemes use **random search** (Waldvogel et al., 1997) rather than a linear method to support efficient name lookup. However, the random search also causes the so-called backtracking problem. Taking the binary search as an example, if FIB stores 2 names "/c1" and "/c1/c2/c3/c4," prefixes "/c1/c2" and "/c1/c2/c3" are inserted into the table as marker entries to support binary search. Then the search route for "/c1/c2/c3/c5" will be:

$$/c1/c2 \xrightarrow{HIT} /c1/c2/c3/c5 \xrightarrow{MISS} /c1/c2/c3 \xrightarrow{HIT} \text{End}$$

If the algorithm terminates just after the binary search, the result is NO FOUND because all HIT entries in the search route are markers, leading to a false negative since the longest prefix matching for it is "/c1."

To handle this case, IP protocol proceeds to the last matching entry and introduces a backtracking search find the LPM Nevertheless, as mentioned above, in NDN, backtracking for every mismatch is impractical for both efficiency and security. Works in Yuan and Crowley (2015) considered to let each virtual entry store the forwarding information inherited from its non-virtual longest matching prefix, without giving implementation details, which requires keeping the consistency between a marker entry and its corresponding LPM especially in a dynamic environment.

2.4. The Memory Consumption Problem

Another problem of the random search is the memory consumption, resulting from the insertion of virtual entries as well as the memory waste caused by outdated entries: using the example above, if name "/c1/c2/c3/c5" is deleted, "/c1/c2" and "/c1/c2/c3" become useless and should be removed for memory saving. However, due to the diversity of name components, it is extremely hard to determine whether a virtual entry is useful only if to traverse the entire table; Our scheme focuses on the cumulation of outdated virtual entries, and the FIB compression methods are not involved in the present study.

3. ALGORITHM DESIGN

3.1. Hash Table

According to the evaluation result in So et al. (2013), among widely-used hash functions, CityHash[1] shows the best performance in terms of both computational cost and hash collision probability. SipHash (Aumasson and Bernstein, 2012) is also an appealing candidate which is a bit slower than CityHash but provides higher resistance to hash flooding DoS attack. In our scheme for evaluation, CityHash is selected since we concerned more about the operation speed rather than security in the current period of study.

`std::unorder_map` is selected as the implementation of hash table, since it is chained-based and required only one hash function, reducing the computational overhead compared to open addressing or methods requiring multiple hash functions.

3.2. Reconstruction of the FIB

As is mentioned above, by component granularity, all shorter prefixes of a name ought to be stored in the table to make the binary search feasible, thus requiring the process called the FIB reconstruction. In a reconstructed FIB, table entries are classified into the following two types:

- **Real Entry.** Each real entry stands for a name referring to a real existing file, and all the entries are real before the reconstruction.
- **Non-real Entry.** In the FIB reconstruction, if a real entry's shorter prefix does not exist in the table, a corresponding non-real entry should be inserted for it. A non-real entry does not refer to any data of real existence and can not be used to guide the interest forwarding. The only usage of it is to support the random search.

The lookup for name n starts at n's prefix with N_0 components, namely, a prefix of length N_0, in which N_0 is given by the algorithm. If there is any match, the algorithm will select n's prefix of length $N_1 > N_0$ as the next one to search. Otherwise the lookup proceeds to length $N_1 < N_0$.

Taking the binary search as an example: If the FIB stores forwarding information for name n = "/c1/c2/c3," then n's prefixes "/c1," "/c1/c2" should be inserted as non-real entries when they are absent in the table. Let L and H be the lower and upper bound of the binary search, and $M = \lfloor (L + H)/2 \rfloor$ be the length to look up. The search path for name n' = "/c1/c2/c3/c4/c5/c6" is as follows:

[1] CityHash https://code.google.com/p/cityhash/.

Set L to 1, H to the length of the name $\rightarrow$

$(L = 1, H = 6, M = 3)$ /c1/c2/c3 $\xrightarrow{HIT\ (L=M+1)}$

$(L = 4, H = 6, M = 5)$ /c1/c2/c3/c4/c5 $\xrightarrow{MISS\ (H=M-1)}$

$(L = 4, H = 4, M = 4)$ /c1/c2/c3/c4 $\xrightarrow{MISS\ (H=M-1)}$

$(L > H)$ End

The last match n_{lm} (in this case is /c1/c2/c3) is returned as the search result for n', thus giving the implementation of the LPM algorithm.

However, in spite of the improvement to the search efficiency, the random search also has drawbacks described below:

- **The backtracking problem.** As shown in section 2.3, if the random search for n ends with a non-real entry name n_{lm} as the last match, there may also exist the longest matching real entry for n. In this case, the backtracking is required. The algorithm has to visit n_{lm} and perform an LPM search for it, leading to large computational overhead.
- **The outdated non-real entry problem.** As shown in section 2.4, if a non-real entry name n has no real entry inherited from it, n should be removed for space saving. However, it is impossible to check whether a non-real entry is redundant only if to traverse the entire table. As a result, the cumulation of outdated non-real entries aggravates the memory consumption problem of the NDN forwarding plane.

To handle the backtracking problem, when the random search ends with a non-real entry as the last match, we divide non-real entries into two types based on whether a backtracking process is required:

- **Virtual Entry.** A Non-real entry is virtual if none of its proper prefixes has a corresponding real entry in the table. When the last matching entry of the random search is virtual, the search process terminates.
- **Semi-virtual Entry.** Otherwise, this non-real entry is semi-virtual and the backtracking is required.

3.3. Structure Design

In our FIB design, besides the hash table, we introduce a component granularity trie (prefix tree) structure to record the logical relationships among names:

Figure 1 illustrates the primary data structure of our design: The hash table is utilized for fast name lookup, the search key in each table entry (n, e) is the interest name n, while the value is the corresponding trie node e.

A trie node stores the entry type (real, virtual, or semi-virtual), the forwarding information of this name and the pointers to parent and child nodes. Since name prefixes are not duplicated stored in the trie, extra memory consumption of this scheme can be restricted. Each edge in trie refers to a name component, while node stands for the name which is the combination of components on the path from the root to this node; Root is set to virtual, which does not belong to any name.

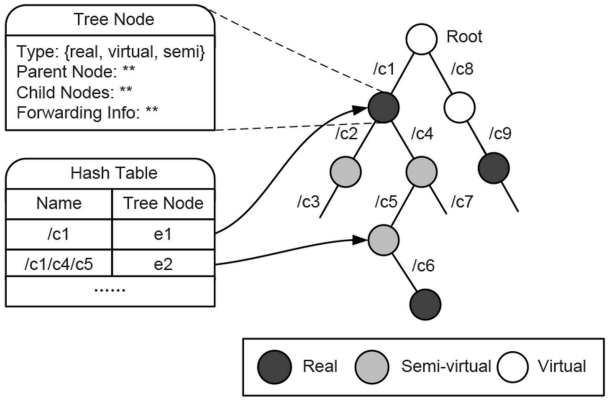

FIGURE 1 | Data structure of HT based-FIB combined with trie.

The motivations to introduce the trie structure are as follows:

- It is possible that non-real entry type varies with table operations of other names, e.g., virtual entry "/c1/c2/c3" becomes semi-virtual upon the insertion of real entry "/c1/c2." To update the entry type timely in a dynamic table, the inheritance relationships among names should be recorded.
- Trie gives the solution to the outdated non-real entry problem: if a leaf node is non-real in the trie, then this entry is redundant and should be removed.
- Trie can speed up the backtracking in the search algorithm, since each node has only one parent and no name or component matching is required in this process.

Here we present the operation algorithms of our method:

3.4. Operation Details

Insertion. There are 2 cases when a new name n is coming:

1) As shown in **Figure 2A**, if there exists non-virtual entry (n, e) for this name in table, e is set to real, and all virtual nodes in e's subtree are set to semi-virtual.
2) As presented in **Figure 2B**, if an entry for n is not found, a real entry is created for n. To ensure that all n's proper prefixes have corresponding entry in the table, the algorithm performs a backward search for its proper prefixes, and create non-real entries for prefixes non-existing in the table. The process continues until the found of n's LPM entry in the table or the arrival of the root. All non-real entries created in insertion are set to semi-virtual if the LPM entry is real or semi-virtual. Otherwise, they are set to virtual.

Deletion. There are 3 cases when deleting name n:

1) For (n, e), if e is not a leaf and e's parent is real or semi-virtual, then e is set to semi-virtual.
2) As **Figure 3A** indicates, if e is not a leaf and e's parent is virtual, then set e to virtual instead of removing it. Furthermore, for each semi-virtual node e^* in e's subtree, when there are no real nodes on the path from e^* to e, e^* is also set to virtual.

Algorithm 1: Key Insertion Algorithm

Input:
H: HT and trie-based FIB.
n: n = "/c1/c2/.../cN" is the name to insert.
f: The corresponding forwarding information of n.
Output:
H: HT and trie-based FIB, with n inserted.

```
1: lookup n in HT
2: if n is the name of a real entry (n, e) then
3:   update e's forwarding information with f
4: else if n is the name of a non-real entry (n, e) then
5:   /* as Figure 2A */
6:   set e's type to real, e's forwarding information to f
7:   for each virtual entry (∼, e*) in e's subtree do
8:     set e*'s type to semi-virtual
9:   end for
10: else
11:   /* as Figure 2B */ create entry (n, e_N) and insert it to HT
12:   set e_N's type to real, e_N's forwarding information to f
13:   for i = N − 1 to 1 do
14:     lookup n_i ="/c1/c2/.../ci" in HT
15:     if n_i is the name of an entry (n_i, e) then
16:       add e_{i+1} to e's child list, set e_{i+1}'s parent to e
17:       if e is virtual then
18:         set e_j(i < j < N)'s type to virtual
19:       else
20:         set e_j(i < j < N)'s type to semi-virtual
21:       end if
22:       return
23:     else
24:       create entry (n_i, e_i) and insert it to HT
25:       add e_{i+1} to e_i's child list, set e_{i+1}'s parent to e_i
26:     end if
27:   end for
28:   add e_1 to root's child list, set e_1's parent to root
29:   set e_j(0 < j < N)'s type to virtual
30: end if
```

3) As **Figure 3B** depicts, if e is a leaf, delete (n, e), then iteratively check n's non-real proper prefixes, and remove it when it is a leaf.
Thus, correctness of entry type can be kept upon table modifications, giving adaption to the dynamic environment.

3.5. Support to Random Search

Insert and delete operations may be a bit costly since our scheme concentrates on the optimization for the search process, for FIB requires frequent read operations and fewer write operations. We take a binary search, which is the simplest method of random search, as the example. Besides, advanced schemes in Wang et al. (2013, 2014, 2017) can also be supported.

Figure 4 presents search process for "/c1/c2/c3/c6/c7," where the search route is:

$$/c1/c2 \xrightarrow{HIT} /c1/c2/c3/c6 \xrightarrow{MISS} /c1/c2/c3 \xrightarrow{HIT} \text{End}$$

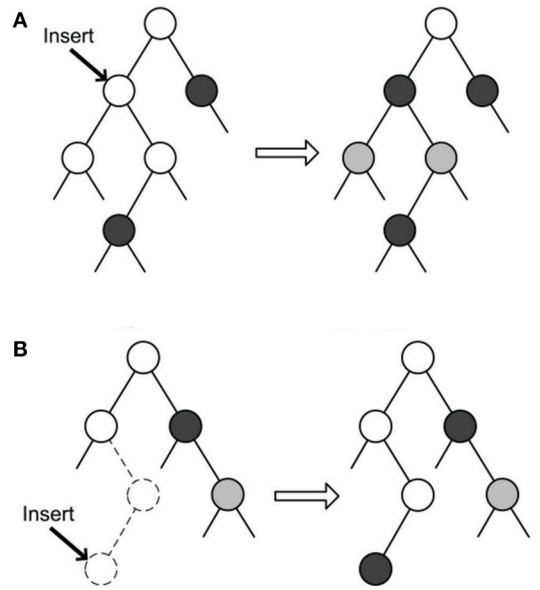

FIGURE 2 | Cases in Key Insertion. **(A)** Case when there exists non-virtual entry for this name in table. **(B)** Otherwise.

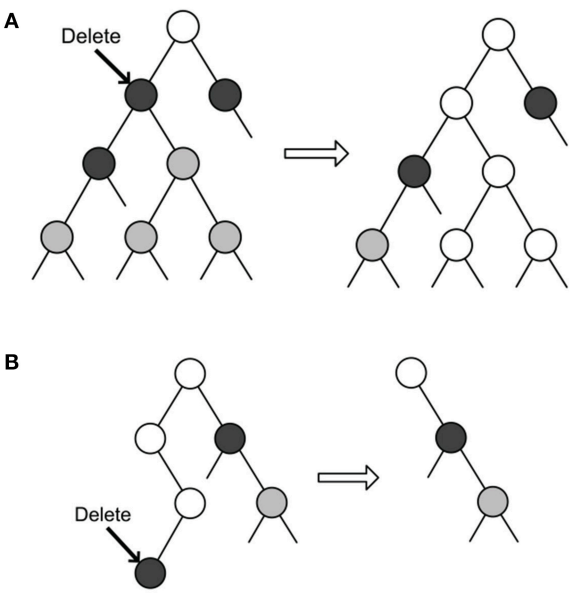

FIGURE 3 | Cases in Key Deletion. **(A)** Case when name to delete is not leaf and has virtual parent. **(B)** Case when name to delete is leaf.

At the termination, the last match "/c1/c2/c3" is semi-virtual, indicating the existence of a real proper prefix of "/c1/c2/c3" in the table. Subsequently, entering the backtracking process, which starts at the last match and iteratively visits the parent until encountering a real entry, whose forwarding information is returned as the search result.

The backtracking process can be timesaving since each node has only one parent and no name or component matching is required. In another case, if the binary search ends with a virtual entry as the last match, NO_FOUND can be ensured and the

Algorithm 2: Key Deletion Algorithm

Input:
H: HT and trie-based FIB.
n: *n* = "/c1/c2/.../c*N*" is the name to delete.
Output:
H: HT and trie-based FIB, with *n* deleted.

```
lookup n in HT,
if n is not the name of a real entry then
  return
end if
if for n's entry (n, e), if e is not a leaf then
  set e's forwarding information to N/A
  if e's parent is semi-virtual or real then
    set e's type to semi-virtual
  else
    /* As Figure 3A, here uses BFS */
    create a empty queue q and insert e into it
    while q is not empty do
      e* = q.pop()
      set e*'s type to virtual
      insert all e*'s semi-virtual child nodes into q
    end while
  end if
else
  /* As Figure 3B */
  remove e from its parent's child list
  delete entry (n, e) in HT
  for i = N − 1 to 1 do
    for n_i ="/c1/c2/.../ci" and its entry (n_i, e_i)
    if e_i is non-real and e_i' is a leaf then
      remove e_i from its parent's child list
      delete entry (n_i, e_i) in HT
    else
      return
    end if
  end for
end if
```

search process terminates. Thus, DoS resistance and the efficiency improvement are provided.

4. EVALUATION

4.1. Discuss About False Negative

Since NDN is a future Internet architecture and not deployed at a large scale yet, it is hard to gain an abundant supply of data from real NDN traffic traces. Therefore, we can not obtain a realistic NDN FIB with sufficient size and logical relationships among names, based on which we could numerically analyze the influence of false negative error. As the substitute, we demonstrate a typical case and discuss its causing reason.

A false negative error occurs if:

- There exist two real names n_1 and n_2 in the table, where n_1 is a proper prefix of n_2.

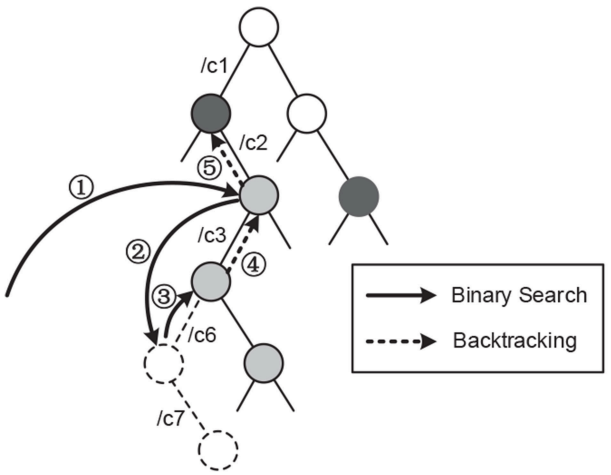

FIGURE 4 | Search Process for Name "/c1/c2/c3/c6/c7".

Algorithm 3: Search Algorithm (Binary)

Input:
H: HT and trie-based FIB.
n: *n* = "/c1/c2/.../c*N*" is the search key.
Output:
f: the forwarding information of *n*'s LPM entry.

```
/* Binary Search */
L = 1, H = N
e_LPM = root
while L ⩽ H do
  M = (L + H)/2
  lookup n_M ="/c1/c2/.../cM" in HT
  if n_M is the name of a entry (n_M, e) in table then
    L = M + 1, e_LPM = e
  else
    H = M − 1
  end if
end while
if e_LPM is virtual then
  return NO_FOUND
else
  /* As Figure 4 */
  while e_LPM is not real do
    e_LPM = e_LPM's parent
  end while
  return e_LPM's forwarding information
end if
```

- The interest has name n_3, and n_1 is a proper prefix of longestCommonPrefix(n_2, n_3).

Namely, if the FIB stores the forwarding information about a file as well as its proper prefix *n*. e.g., "/cn/edu/pku/document/file01/segment01" and "/cn/edu/pku." Subsequently, the request for other data under *n* may cause a false negative error. For example, Upon interest name "/cn/edu/pku/document/file02/segment01," the binary search

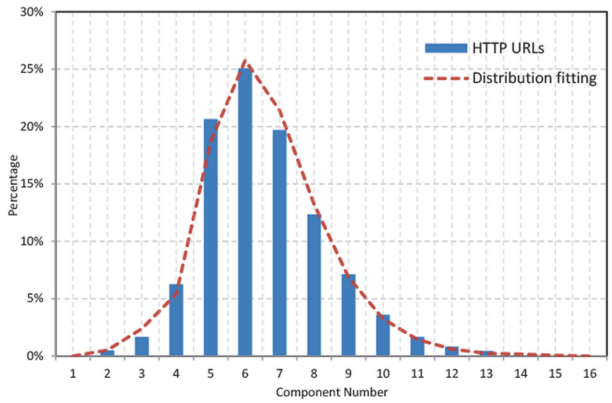

FIGURE 5 | Component number distribution of HTTP URLs.

ends at a non-real entry "/cn/edu/pku/document/." If the algorithm ends with no backtracking, an error occurs since the LPM is "/cn/edu/pku" and not found.

4.2. Experimental Setup

Then, we evaluate the experimental performance of our scheme, based on the statistical analysis of HTTP URL length.

Test set. Due to the following reasons, we selected HTTP URLs to generate artificial dataset of NDN names for our evaluation:

1. As is mentioned above, building a large and realistic NDN routing table is a difficult task since NDN has not been deployed at Internet scale. Therefore, previous work usually translate existing Internet traces to generate NDN dataset (Yuan et al., 2012; So et al., 2013; Yuan and Crowley, 2015).
2. Most current works in NDN assumes URL-like hierarchical names whose components are delimited by slashes, based on NDN names' similarities to URLs, we presumed that NDN forwarding table has a similar long-tailed component distribution. The domain name was not considered since their component amounts are mostly 2 or 3 and follow a heavily skewed distribution (So et al., 2013; Wang et al., 2013).
3. Since the performance of the lookup algorithm is sensitive to the length of query names, for the convenience of numerically evaluating our scheme, we generated artificial NDN names with the variable expected component number.

An HTTP URL is converted into an NDN name in the following way: (1) The file name extensions and query part after the character "?" are excluded; (2) The domain name is reversed by component granularity; (3) Both character "/" and"." are treated as component delimiters. e.g., "doc.pku.edu.cn/file/01.txt" becomes "cn/edu/pku/doc/file/01."

In our test, we obtain HTTP URLs from web traffic traces in[2] and analyze their statistical properties, base on which we generate test sets of interest name for our evaluation.

[2]Internet Traffic Archive. ftp://ircache.net/Traces/DITL-2007-01-09/.

TABLE 1 | Lookup performance (always MISS).

N	Linear search		Binary search		
	Average lookup times	Throughput	Average lookup times	Throughput (No backtrack)	Throughput (Algorithm 3)
6	6	100%	2.34	236%	235%
7	7	100%	2.54	250%	247%
8	8	100%	2.70	267%	266%
9	9	100%	2.85	280%	279%
10	10	100%	2.96	301%	299%

Figure 5 illustrates the component number N's distribution of 4 million access logs from web proxy caches, while the average component number is $\bar{\lambda}_0 = 6.58$. The red line is the following distribution $\rho(N; \bar{\lambda}_0)$ utilized to fit the statistical result:

$$2\rho(N;\bar{\lambda}_0) = \begin{cases} 0 & (0 \leqslant N \leqslant 1) \\ \text{Poisson}(N-2;\bar{\lambda}_0-2) & (2 \leqslant N \leqslant 4) \\ \text{Poisson}(N-2;\bar{\lambda}_0-2) & \\ \quad +\text{Poisson}(N-5;\bar{\lambda}_0-5) & (5 \leqslant N) \end{cases}$$

In which $\text{Poisson}(N;\lambda)$ is the Poisson distribution with the expectation value λ:

$$\text{Poisson}(N;\lambda) = \frac{\lambda^N}{N!}e^{-\lambda} \ (0 \leqslant N)$$

To generate an interest name whose component number has the expectation value of $\bar{\lambda} > 5$. (1) Its component number N is derived from $\rho(N;\bar{\lambda})$, which is a stretched version of the distribution above by substituting $\bar{\lambda}$ for $\bar{\lambda}_0$, with its expectation value changed to $\bar{\lambda}$. (2) Its first component is selected randomly from 10 common domain name suffixes (e.g., com, org) in a uniform way, while other components are selected uniformly from a set of 10 thousand unique components collected from URLs and averagely have 8.62 characters.

Methodology. To make the case simple, std::unorder_map is selected with hash function CityHash[1], and the forwarding information is set to be empty. The table for test contains 2 million random-generated names, and each kind of operation is executed for 0.5 million times. The experiment is repeated for 10 times with different seed, and the average value is taken as the result of performance evaluation.

4.3. Efficiency Test

Lookup Test. For the lookup operation, we concern about: (1) The effectiveness of the random search; (2) The impact of the backtracking on the lookup speed. Therefore, we compare linear search, binary search without backtracking (which may cause a false negative) and Algorithm 3 in our evaluation.

Firstly, we use names that do not exist in the table as the lookup keys, with the expectation value N of the component number. The performance metrics selected are

TABLE 2 | Lookup performance (always HIT).

M	N	Linear search		Binary Search		
		Average lookup times	Throughput	Average lookup times	Throughput (No backtrack)	Throughput (Algorithm 3)
3	6	3.98	100%	2.84	126%	123%
	7	4.99	100%	3.02	131%	130%
	8	6.02	100%	3.17	151%	150%
	9	7.00	100%	3.31	168%	167%
	10	8.01	100%	3.45	185%	183%
4	6	3.03	100%	2.90	85%	85%
	7	4.00	100%	3.06	104%	101%
	8	5.02	100%	3.21	123%	124%
	9	6.02	100%	3.35	141%	139%
	10	7.03	100%	3.47	159%	159%

TABLE 3 | Insertion and Deletion performance.

M	Insert throughput		Delete throughput	
	HT	Algorithm 1	HT	Algorithm 2
3	100%	63%	100%	74%
4	100%	59%	100%	75%
5	100%	59%	100%	74%
6	100%	61%	100%	71%
7	100%	58%	100%	74%

the average lookup times and the operation throughput, using the linear search as the baseline. **Table 1** present the lookup performance with different parameters N, indicating the considerable improvement of binary search for no-matching interest since it does not need to traverse the entire name.

We test the case of search hit, with the following hypothesis: (1) Since the distribution of domain names is skewed and hard to fit (over 70 percent of domain names have 3 components), we assume that the component number of names in FIB obeys the similar distribution with HTTP URLs. Here we use Poisson($N - 1; \lambda - 1$); (2) the expected component number M of FIB names is smaller than that of interest names, namely $M < N$.

Considering that most domain names have 2 or 3 components, and NDN interest names are possibly to be longer than HTTP URLs, we select $M = 3, 4$ for evaluation. **Table 2** reveals that the superiority of binary search increases with $N - M$, and the backtracking process has little time overhead compared to HT lookups.

Insertion and Deletion Test.

Table 3 compares the time consumption of Algorithm 1 from the basic HT insertion, as well as that between Algorithm 2 and the basic HT deletion. If Algorithm 1 or 2 inserts or deletes k entries during the execution phrase, its cost time is divided by k for comparison. As is shown writing operations is slower as the sacrifice to speed up reading.

4.4. Scalability Test

Compared to other data structures, HT induces higher memory consumption since the entire key has to be stored to handle hash collision, aggravating the challenge in NDN application at large scale. Below we test the impact of our scheme on the memory cost. The FIB name still follows the distribution Poisson($N - 1; \lambda - 1$) with expected component number M. The FIB has a size of 2 million entries, with all forwarding information set to be empty.

TABLE 4 | Upbound of non-real entries' extra memory cost.

	M = 3	M = 4	M = 5	M = 6
Extra memory cost	116%	166%	216%	265%

Extra memory cost of non-real entries. To obtain an upbound of extra memory cost, we make each real entry name not be a prefix of any other real names in the table. **Table 4** presents the test result and indicates the major disadvantage of random search application in NDN: Even for relatively small expected component number M, the non-real entries double the FIB size in the worst case, emphasizing the necessity of FIB compressing approach like a footprint-based hash table, or the combination with the bloom filter.

Extra memory cost of trie structure. With M kept the same, compared to the hash table (forwarding info as the table value), our structure averagely consumes 15.7 Bytes per entry, mainly resulting from three 32-bit pointers for trie (parent, next sibling, first child) and 1 Byte for entry type.

5. DISCUSS AND FUTURE WORK

As is evaluated, the main shortcoming of the binary search implemented for Hash table is the extra memory cost of non-real entries, in the worst case, the FIB size is expanded for four times, much aggravating the storage problem of NDN forwarding plane.

The amount of non-real entries required can be reduced by the improvement of the lookup algorithm. Taking the following method, which is a refined version of binary search, as an example:

1. For a real name, all shorter prefixes with even numbered length should be stored in the FIB. e.g., if the table stores name "/c1/c2.../cN", then "/c1/c2", "/c1.../c4", "/c1.../c6"... are also to be stored. Like the binary way, if there exists no corresponding entry for a prefix, then a non-real entry should be inserted for it in the FIB reconstruction.
2. In the search process, L and H (the lower and upper bound) are always even. If the query name has length N, at the beginning, $L = 2, H = 2\lfloor N/2 \rfloor$.
3. The length to lookup in each iteration cycle is:

$$M = 4\lfloor \frac{L+H}{4} \rfloor$$

   Which is also an even number. On the lookup hit, $L = M + 2$, otherwise, $H = M - 2$. The iteration ends if $L \geqslant H$.
4. After the iteration, if there exist no last matching prefixes, then we lookup "/c1." On the lookup hit, "/c1" is returned as the LPM. Otherwise, NO_FOUND is returned.

5. In the other case, if the last matching prefix is "/c1.../c*lm*", then we lookup "/c1.../c($lm + 1$)". On the lookup hit, "/c1.../c($lm + 1$)" is returned as the LPM. Otherwise, "/c1.../c*lm*" is returned.

As is shown, with the computational complexity logarithmical to the length of the query name, the algorithm above reduces the non-real entry amount by a half. Extending to more general case, we can choose integer sequence $\{N_1, N_2...\}$ as the lengths of non-real prefixes to be inserted. e.g., In the binary case, $N_1 = 1, N_2 = 2...$; In the method above, $N_1 = 2, N_2 = 4...$; Our future work includes selecting N_i to optimizing the computational and storage cost.

The statistic of name length can also contribute to the improvement of search algorithm. In the binary way, the middle of the search range is selected as the next target M to lookup; With the assistance of name lengths' prior probability, we are able to select M minimizing system's information entropy, thus shortening the search path.

6. CONCLUSION

We propose a composite data structure based on the combination of trie and hash table, to solve the problems of backtracking and outdated non-real entries in the random search.

The experiment indicates that the introduction of trie structure exerts restricted influence on the search speed and memory consumption, and the performance of random search increases with longer names in interest and shorter names in FIB. However, the random search sacrifices write speed and memory efficiency for fast lookups, aggravating the memory cost problem in NDN forwarding plane. Further optimization to memory cost and the search algorithm is required, as well as larger-scale experiments of NDN implementation.

AUTHOR CONTRIBUTIONS

JH contributed to the core concept of this structure and the corresponding algorithm. HL as the professor, provided valuable advice for the design and the implementation of our scheme.

REFERENCES

Aumasson, J.-P., and Bernstein, D. J. (2012). *Siphash: A Fast Short-Input PRF*. Cryptology ePrint Archive, Report 2012/351. Available online at: http://eprint.iacr.org/

Degermark M., Brodnik A., Carlsson S., and Pink S. (1997). "Small forwarding tables for fast routing lookups," in *Proceedings of the ACM SIGCOMM '97*, (Cannes: ACM).

Doeringer W., Karjoth G., and Nassehi M. (1996). Routing on longest-matching prefixes. *IEEE ACM Trans. Netw.* 4, 86–97. doi: 10.1109/90.503764

So W., Narayanan A., and Oran D. (2013). "Named data networking on a router: Fast and dos-resistant forwarding with hash tables," in *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (San Jose, CA).

So W., Narayanan A., Oran D., and Wang Y. (2012). "Toward fast NDN software forwarding lookup engine based on hash tables," in *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (Austin, TX: ACM), 85–86.

Srinivasan V., and Varghese G. (1998). "Faster IP lookups using controlled prefix expansion," in *ACM SIGMETRICS Performance Evaluation Review,* Vol. 26 (Madison, WI: ACM), 1–10.

Waldvogel, M., Varghese, G., Turner, J., and Plattner, B. (1997). "Scalable high speed IP routing lookups," in *Proceedings of the ACM SIGCOMM '97* (Cannes).

Wang Y., Qi Z., Dai H., Wu H., Lei K., and Liu B. (2017). "Statistical optimal hash-based longest prefix match," in *Symposium on Architectures for Networking and Communications Systems* (Beijing).

Wang Y., Tai D., Zhang T., Lu J., Xu B., Dai H., et al. (2013). "Greedy name lookup for named data networking," in *ACM SIGMETRICS Performance Evaluation Review* (Pittsburgh, PA), Vol. 41, 359–360.

Wang Y., Xu B., Tai D., Lu J., Zhang T., Dai H., et al. (2014). "Fast name lookup for named data networking," in *IEEE 22nd International Symposium of Quality of Service* (Hong Kong).

Yuan H., and Crowley P. (2015). "Reliably scalable name prefix lookup," in *Symposium on Architectures for Networking and Communications Systems* (Oakland, CA), 111–121.

Yuan H., Crowley P., and Song T. (2017). "Enhancing scalable name-based forwarding," in *Symposium on Architectures for Networking and Communications Systems* (Beijing), 60–69.

Yuan H., Song T., and Crowley P. (2012). "Scalable NDN forwarding: Concepts, issues and principles," in *2012 21st International Conference on Computer Communications and Networks (ICCCN)* (Munich: IEEE), 1–9.

Zhang L., Estrin D., Burke J., Jacobson V., Thornton J. D., Smetters D. K., et al. (2010). *Named Data Networking (ndn) Project*. Xerox Palo Alto Research Center-PARC, Tech. Rep. NDN-0001.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.