On the Granularity of Trie-based Data Structures for Name Lookups and Updates

Chavoosh Ghasemi*, Hamed Yousefi*, Kang G. Shin, and Beichuan Zhang

Abstract—Name lookup is an essential function, but a performance bottleneck in both today's and future network architectures. *Trie* is an excellent candidate data structure and has been widely used for looking up and updating names. However, the granularity of trie—at bit, byte (character), or component level—can dramatically affect the network performance in terms of memory usage and packet-processing speed, which has not yet been studied adequately.

To fill this gap, we first show that the choice of trie's granularity for name lookups and updates (i.e., insertions and removals) is not a trivial problem due to the complex performance trade-offs involved. We also introduce a new tool, called *NameGen*, which uses a Markov-based name learning model and generates pseudo-real datasets with different tunable name characteristics. We compare different trie granularities based on a collection of datasets and performance metrics, highlight the strengths and weaknesses of each granularity, and draw a conclusion on the choice of granularity. Surprisingly, our experimental evaluation finds that there are only two key rules to choose the proper trie's granularity for any kind of dataset: (I) bit-level trie is the choice when the memory requirement is a real concern, else (II) character- and component-level tries are preferred for faster lookups and updates when dealing with names composed of short and long components, respectively.

Index Terms—Name Lookup; Trie; Implementation Granularity; Named Data Networking.

1 Introduction

Numerous emerging applications, such as firewalls, intrusion detection systems, load balancers, search engines, peer-to-peer file-sharing and content distribution networks, rely heavily on name-based services for packet forwarding, filtering, mapping, etc. Moreover, significant changes in the Internet usage have led to the rapid development of *information-centric networking* (ICN) architectures, such as *named data networking* (NDN) [37]. NDN makes a fundamental shift in the semantics of network service from delivering the packet to a given destination IP address to fetching data identified by a given name. Thus, name lookup and update (i.e., insertion and removal of names) are core functions to the performance of both today's and future network architectures [19], [23], [38], [29], [36].

Unlike IP addresses of fixed length (32 or 128 bits), names are variable-length with no upper bound. This leads to two main issues in designing packet forwarding engine. (This paper focuses on lookups and updates in forwarding tables, where name lookup hinges on longest-prefix matching (LPM), as it is easily applicable to other name-based services, such as filtering, mapping, etc.)

Memory Usage: The size of a name-forwarding table is, in general, much larger than that of an IP-forwarding table, because each name prefix is likely to be significantly longer than an IP prefix. Moreover, the number of name prefixes is orders-of-magnitude greater than that of IP prefixes [33]. (Taking the domain names as an example, there are already 330 million registered domain names in the Internet as of September 2017.) These factors together result in the forwarding tables with a much larger memory footprint than the current IP-forwarding tables with up to 700K IP prefix entries. Therefore, a memory-efficient data structure for the name-forwarding table is essential.

Processing Speed: Since names are of variable and unbounded length, lookups of and updates to the name-forwarding table may take longer than those to the IP-forwarding table. More frequent updates to the name tables due to updates of forwarding rules or publication and withdrawal of contents also make name processing a bottleneck in the forwarding engine. To support the service at a global scale, fast lookup, insertion, and removal of names in large forwarding tables are essential.

Trie, especially Patricia trie, is an excellent candidate data structure and has been widely used for LPM [31], [32], [11], [34], [14], [17], [24], [29], [36]. However, different granularities (i.e., bit, character, or component level) of trie incur different time and space overheads/complexities, introducing a trade-off between memory usage and name processing speed. On the one hand, removing the redundant information among name prefixes improves memory efficiency. (If two names[1] share a common sequence of bits/characters/components starting from the beginning of names, the common sequence will be stored only once in

* *Co-primary authors.*
- *C. Ghasemi is with the Department of Computer Science, The University of Arizona. E-mail:* chghasemi@cs.arizona.edu
- *H. Yousefi is with the Department of Electrical Engineering and Computer Science, The University of Michigan. E-mail:* hyousefi@umich.edu
- *K.G. Shin is with the Department of Electrical Engineering and Computer Science, The University of Michigan. E-mail:* kgshin@umich.edu
- *B. Zhang is with the Department of Computer Science, The University of Arizona. E-mail:* bzhang@cs.arizona.edu

1. Without loss of generality, we assume the names are hierarchically structured and composed of multiple components separated by delimiters (usually slashes '/'). Flat names are a special case of hierarchical names with only one component.

the trie.) A finer-grained structure is expected to further compress forwarding tables and minimize the memory footprint. On the other hand, traversing a trie can be slow because accessing each level of the trie requires a memory access that cannot exploit CPU caches. A coarser-grained structure reduces the depth of the trie, thus expected to speed up the processing functions.

However, our in-depth evaluation shows that the above expectations do not always hold and the choice of granularity is not trivial as many direct and indirect factors can influence the trade-off significantly (see Section 2.2 for more on this). Thus, we take a deeper look into the choice of granularity, as it plays a key role in the performance of different name-processing functions.

Specifically, this paper makes the following three main contributions.

1) To the best of our knowledge, this is the first to consider the impact of granularity and make a comprehensive comparison of three granularities of trie-based name lookup and update structures (i.e., bit-, character-, and component-level tries). Our implementations, oupen - sourced and available at Github[2], establish an important baseline for performance evaluation in future studies, especially for information-centric networks. A collection of metrics are also introduced to compare different granularities and evaluate their cost–performance trade-offs.
2) We develop a new tool, called `NameGen`, to generates a collection of large datasets with different name characteristics. `NameGen` is important for studying name-processing structures for three distinct features. First, it enables end-users to tune the name characteristic (e.g., number and length of components), which plays a critical role in the performance and choice of name-processing structures. Second, by using a Markov model for learning real datasets, extracting information, and generating close-to-real names randomly, it removes the need to rely on available small datasets or to generate huge datasets of fully random names for network evaluation. Finally, it provides consistency of datasets for future research. That is, having the characteristics of a given dataset, one can generate another different dataset with the same characteristic, making the related results/studies fairly comparable. All of these features are especially important for future Internet architectures as there is currently no standard for content names. Without loss of generality, we focus on NDN names as an important use-case[3].
3) We provide in-depth experimental evaluation of the interplay between the performance metrics. These results highlight the unique strengths and weaknesses of each granularity of trie-based structures. Not only a clear trade-off between speedup and memory usage is made, but also the effect of dataset characteristics on different granularities is demonstrated. According to the results, the bit trie is the choice when the memory requirement is a real concern. However, it results in the worst processing speed and is significantly outperformed by character and component tries. Surprisingly, character and component tries show different relative performances in different datasets. In summary, the character trie is the choice for processing speed when the names are composed of short components; otherwise, the component trie is preferred for faster lookups and updates. Thus, only one of the many factors that can affect the absolute performance is found to dictate the merit of the structures in terms of speed: component length. In other words, the length of components (especially the initial ones) plays a decisive role in the choice of granularity, while the number of components only magnifies the relative performance improvements of character and component tries, and the length of names/prefixes only affects the absolute results.

2. https://github.com/chavoosh/TrieGranularity

3. As a prominent design under the ICN paradigm, NDN [37] has gained significant momentum with participation from academia and industry. Companies including Huawei [7], [25] and Cisco [9], [20], [26], [27] have made substantial R&D efforts on different aspects of NDN in recent years.

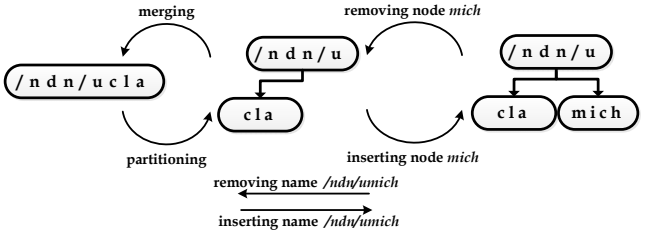
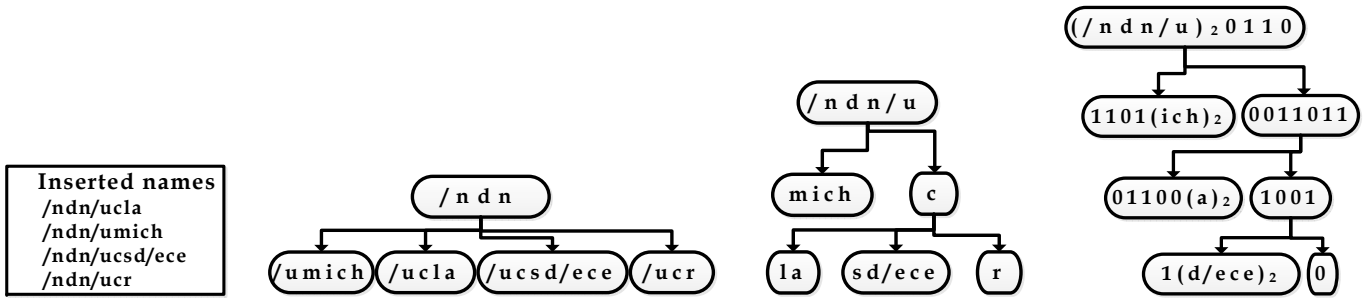


Fig. 1: Partitioning and merging are direct results of name insertion and removal, respectively (a character-level trie as an illustrative example).

The rest of this paper is organized as follows. Section 2 details the design and implementation of different granularities of trie-based lookup and update structures, and compares them based on a collection of performance metrics. Section 3 unveils the design of `NameGen` and presents the characteristics of different generated datasets. Section 4 evaluates the performance of different trie granularities in terms of memory-usage and speed of different processing functions, and also draws a conclusion on the choice of the granularity. Section 5 reviews the related work. Finally, the paper concludes with Section 6.

2 X-LEVEL PATRICIA TRIE

As mentioned earlier, trie has been used widely for implementation of forwarding tables thanks to its memory efficiency. Using this structure, searching for names that share the same prefix only needs to save one copy of the shared part. For example, to store two name prefixes */ndn* and */ndn/umich*, the string */ndn* is stored only once. Trie naturally supports both exact and longest-prefix match (LPM). It organizes the name prefixes in a tree structure. Each node in the tree contains part of name prefix, and the nodes are connected via links for name traversal. However, traversing tries can be slow because accessing each level of the trie requires a memory access that cannot exploit CPU caches. To alleviate this problem, merging the intermediate nodes with only one child can effectively compress the traversing paths. This structure, referred to as *path compressed trie* or *Patricia trie* [21], not only reduces the number of nodes, thus saving memory footprint, but also reduces the depth of the trie, thus speeding up name-processing. The nodes (except for the root) in this data structure may have 0 (i.e., leaf nodes)



(a) a small set of prefixes (b) component-level trie (c) character-level trie (d) bit-level trie

Fig. 2: x-level tire representation. Accordingly, different choices of granularity result in completely different tries. $(str)_2$ is a binary representation of str, e.g., $(ndn/u)_2$ 01100011 01110010 collectively represent *ndn/ucr* in bit-level trie.

or > 1 children. We will henceforth focus on Patricia trie, and hence use terms 'trie' and 'Patricia trie' interchangeably.

Name processing functions: Name Lookup is the core function of a forwarding engine. The names also need to be inserted in or removed from the forwarding tables frequently due to forwarding rules' updates or content publishing and withdrawal [33].

To look up a name, we traverse the trie from its root to its descendant nodes until an exact match is found or the search fails, returning the LPM. Lookup is the first step of insertion and removal of a name.

For insertion, once lookup is terminated, we allocate a new node to store the remaining unmatched part of a name. Node partitioning is a direct result of insertion. In case of mismatch at a node, node-partitioning is triggered, which splits the corresponding node (and the name) from the differentiation point. It shifts the unmatched part of the node as well as all of its children one level down while creating a new child corresponding to the unmatched part of the name. Fig. 1 illustrates the node partitioning caused by storing a new name */ndn/umich*, which shares the same prefix with an already inserted name */ndn/ucla*.

Removal is the reverse operation of insertion. Once the name lookup finds an exact match, we delete its associated leaf node. If the parent is left with a single child, they will be merged recursively. If lookup fails to find an exact match, we determine that the removal is invalid, as the name does not exist in the trie. Note that the end of the input names needs to be marked to prevent false positives. For example, without knowing the End Of Name (*EON*), searching for name */ndn* in a trie, where name */ndn/ucla* has already been inserted, returns */ndn*. However, this is a false positive because name */ndn*, as an independent name, has never been inserted in the structure. To mitigate this problem, */ndn[EON]* will be searched, which will fail as expected. Thus, an exact match cannot occur in the middle of the structure. However, to simplify the presentation, we intentionally skip using *[EON]* at the end of names in the examples. Fig. 1 shows the reverse operation of insertion, where removing name */ndn/umich* comes with merging operation, storing name */ndn/ucla* as a whole in a single node.

Granularity: There are numerous implementations of name-based forwarding tables using trie structures, which can be grouped as bit-, character-, and component-level tries, depending on whether they treat a name as a sequence of bits, characters (bytes), or name components (delimited by slash /), respectively. Fig. 2 illustrates 3 Patricia tries built with different granularities for the same input names. Accordingly, different choices of granularity result in completely different tries (in terms of depth, width, number of nodes, etc.)

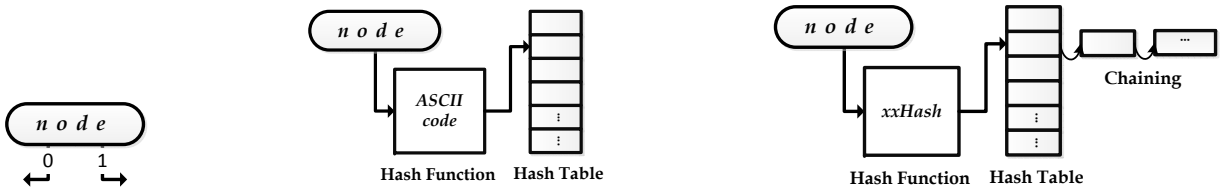
Although the binary trie implementation is often adopted by IP routing tables, LPM in name-forwarding tables is usually of component granularity. However, this should not be translated to using component-level trie as the best choice since it does not always offer the best performance according to our extensive evaluation. The granularity of trie creates a complex trade-off in lookup/update performance while various direct and indirect factors can influence this trade-off. Actually, different choices of granularity offer different memory usage and speed in storing and processing the names. Finally, the granularity used to implement the trie structure is critically important to meeting the performance need, but it is non-trivial to make an optimal choice of granularity. In the next section, we present the implementation details of these three Patricia trie structures, and discuss the effect of granularity on a collection of performance metrics. As performance depends strongly on implementation granularity, we try to keep different implementations as close to each other as possible for a fair comparison of the structures.

2.1 Implementation Granularity

Fig. 3 gives an abstract view of the architecture of each node in bit-, character-, and component-level tries.

Bit Level: Bit-level trie is a binary Patricia trie which defines a name as a sequence of bits. The set of bits in the nodes from the root to a leaf node together form a complete name prefix. Each node has two pointers: the left pointer to the child starting with 0 and the right pointer to the child starting with 1. Thus, in case of a bit mismatch, the child to branch to can be determined without any processing.

Character Level: This trie defines a name as a sequence of US-ASCII printable characters (i.e., bytes). Thus, the number of children at each node cannot exceed 94 (codes 33–126). To keep track of children, i.e., to implement the trie edges, we employ a hash table at each node, as using linked list drastically decreases the speed due to linear probing (i.e., checking the children, one-by-one, at each node until finding the target child). The ASCII codes are used as keys, while the hash function is simply modulus. To minimize the memory footprint, the initial size of hash table is 1 while expanding



(a) bit-level trie (b) character-level trie (c) component-level trie

Fig. 3: x-level trie implementation.

gradually for collision resolution. In case of a collision, a re-hash function is called, doubling the hash table size until the mapped bucket becomes empty. For example, suppose the size of hash table is 16, and key 33 is added to this table which sits in bucket 1 (equals 33 $\bmod$ 16). Then, adding key 65 causes a collision because bucket 1 is occupied. After doubling the hash table size, key 33 sits in bucket 1 again, so we cannot still add key 65. The size is doubled once more. This time key 33 resides in bucket 33 and key 65 sits in bucket 1, resolving the collision. The maximum size of hash tables in character trie is 128 ($2^7 > 94$). According to this implementation, finding the child's address to branch to at each level requires only one off-chip memory access (to fetch the mapped bucket from memory and read the address of the child).

Component Level: This scheme of trie defines a name as a sequence of components which are delimited by slashes '/'. Like in character trie, we utilize a hash table starting with the size of 1 at each node to keep track of its children. Unlike in character trie, the number of children of each node is not bounded in component trie. A re-hashing function is called whenever the load factor—i.e., the number of records (equal to the number of children of the node) divided by the number of buckets—exceeds a specified threshold. To control frequent rehashing which is time consuming and wastes a lot of memory space, we also employ chaining for collision resolution. Hence, linear probing is likely to happen in a given bucket. Unlike character trie, in this structure, finding the child to branch to at each level needs at least two off-chip memory accesses (to reach the very first element of the chain at the mapped bucket). Actually, the mapped bucket that stores the address of the first element in the chain is fetched via the first off-chip memory access. The second access comes with fetching and reading the first element. From there on, traversing any other element in the chain also requires one off-chip memory access. (According to our experiments on different hashing choices in Appendix A.1, load factor 10 represents the most-balanced trade-off and is used for the rest of our experiments.) We employ xxHash [6], as a very fast hash algorithm, to deploy the hash table. For traversing each part of a name towards the children, the hash of its first component also needs to be stored in the hash table for both collision resolution and fast chain traversal. For example, for name */ndn/ucsd/ece* in Fig. 2(a), the hash table in node */ndn* needs to store not only the address of node */ucsd/ece*, but also the hash of component *ucsd* in the related bucket.

2.2 Discussion

As mentioned earlier, *memory usage* and *speed* are the major performance metrics of a trie structure. However, there are other factors which indirectly affect the performance: *compactness*, *depth*, and *internal overhead* (e.g., the number of nodes and width) of the trie structure.

Compactness represents the amount of redundant information that a data structure can avoid storing. For example, in Fig. 2, to store two name prefixes */ndn/ucla* and */ndn/umich*, a component-level trie can store the string */ndn* only once. It however needs to store both components */ucla* and */umich* in their entirety because its trie nodes must fall on the boundary of name components. Instead, the character-level trie will be able to store */ndn/u* only once. On the other hand, the binary trie can avoid storing redundant prefixes in bit granularity, thus saving more memory. The binary trie stores $(/ndn/u)_2 0110$ only once, where $(str)_2$ is a binary representation of str.

The binary trie for IP tables has a depth of at most 32 or 128. However, a name-based binary trie has an unbounded depth. Obviously, bit-level tries have the largest depth, which may lead to longer lookup times as traversing each level usually requires to access the main memory once.

The total number of nodes is also an important overhead metric not only in processing but also in memory usage. A binary trie creates a maximum number of nodes. Moreover, as the trie is implemented by pointers from parent node to child nodes, then all the leaf nodes will have null pointers, which is a significant memory overhead.

Another major contributor to the overhead is width, or the number of children per parent node. It greatly affects the processing needed to decide which branch to go to at each level of the trie. In a bit-level trie, each node has up to 2 children, and checking which child node to go to is a quick, single bit operation. However, in a component-level trie, the number of children of each node could be unbounded, thus figuring out which child to branch to will require more time and potentially more memory. Although the hash table is a very fast data structure which helps quickly find the track of the name for traversal, handling collisions and the unbounded chaining incurred by the component-level trie are time-consuming and reduce the name-processing speed. Moreover, the hash tables keep a certain percentage of their buckets empty (determined by the re-hash threshold) that magnifies their memory footprint.

In summary, the highest memory compression in the bit-level trie is achieved at the cost of maximal trie depth, which can degrade the speed. Moreover, the component-level trie reduces the trie depth at the cost of higher width. This can lead to the generation of very fat tries since each node may be associated with a large number of children, incurring a higher internal overhead and slowing down the processing speed. The character trie, on the other hand, generates a taller trie along with a larger number of nodes than the

component trie, which negatively affect its memory saving and speed. Thus, the optimal choice of the granularity is not trivial and requires a large set of experiments. Moreover, the characteristic of the input dataset plays a critical role in the choice of granularity. We develop a new tool that generates a variety of pseudo-real datasets. Each dataset has a specific name characteristic (e.g., number and length of components), giving a clear perspective of the unique strengths and weakness of each trie structure.

We analyze both the time and space complexities of the name processing functions for different trie implementations (see Appendix A.2).

3 NameGen and Dataset Generation

In this section, we describe the process of generating datasets of NDN-like names, and introduce the tools we developed. As NDN names employ URL-like structure [37], [33], we first use our simple and fast URL collector tool, providing a dataset of URLs for feeding `NameGen`, our name generator tool. Next, we present the design of `NameGen` which provides the user with the opportunity of generating a collection of datasets with different name characteristics. Such a tool is important for future studies of name-based structures. Although we focus on generating NDN names as the primary use-case of the current study, our automatic name generator is general enough to be used for other name generating applications by feeding the related inputs.

3.1 URL Collection

To choose between different granularities, we need to compare them against different large datasets each with a specific name characteristic. Actually, we want to find how fast and memory efficient different trie structures work for names with small and large number and length of components. We could not find a widely-used available URL dataset which is good enough even to be fed for learning and then generating diverse results in `NameGen`. The term *good* means a dataset which has (1) a reasonably large size (at least 1M URLs), (2) a diverse number of components, (3) a diverse length of components, (4) Top-Level Domain (TLD) diversity (as they serve as the first component in NDN names), and (5) prefix diversity (i.e., not a large number of the names should share the long prefixes). For example, even collections of open-access datasets of Alexa [3], blacklist [1], and DMOZ [2] could not meet all the above requirements.

Thus, we decided to design a URL collector on our own and collect a large-enough dataset of URLs. Our tool is developed on top of Scrapy [5], which is a popular, fast, and open-source web crawling and web scraping framework written in Python. It is widely used by major information technology companies (e.g., Lyst, CarrierBuilder, and ScienceDirect) to crawl websites and extract structured data from their pages for a wide range of purposes, from data mining to monitoring and automated testing. We customized the spider classes in Scrapy to parse responses, extract scraped links, and return new requests to its engine to follow links recursively. The output may include duplicate names which are removed by our tool. Thus, our input dataset includes only the unique URLs for learning.

TABLE 1: URL dataset specifications

# of names	2,000,000
Avg. length of names	65.99
Max length of names	1000
Avg. length of components	13.94
Max length of components	985
Avg. # of components per name	4.62
Max # of components per name	20

We start by crawling top 500 websites in Alexa list. The output is a dataset of 2M unique URLs. The specifications of this dataset are given in Table 1. To extract more information from the dataset, Fig. 4 shows the distribution of the number and length of components. The dataset is shown to meet all our requirements to equip `NameGen` with the possibility of generating a broader spectrum of possible datasets, each with a specific user-requested characteristic.

Note that without loss of generality, we simply discard the resulting URLs with more than 20 components and with the size of more than 1000 characters as both of these situations rarely (i.e., < 0.0001) happen in our dataset. These statistics will later be used to show the high accuracy of our name generation tool in keeping the characteristics of the input names.

3.2 NDN-like Name Generation

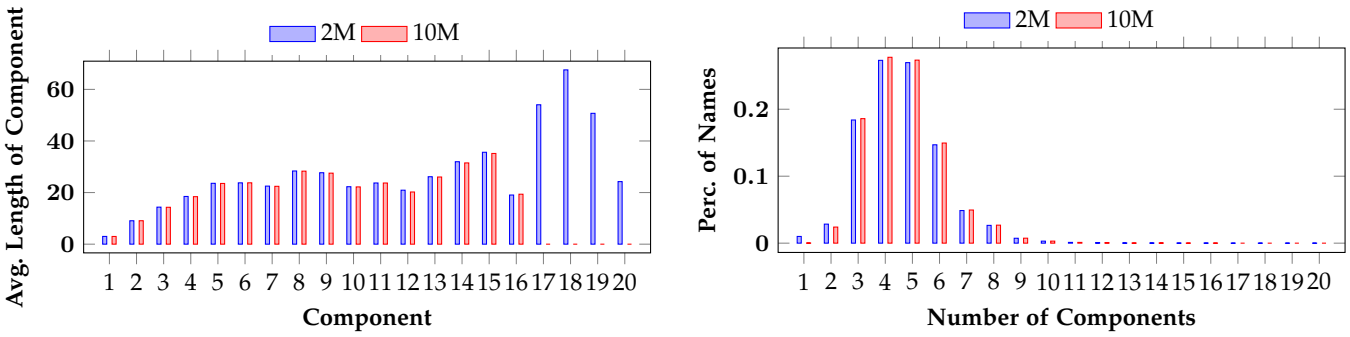
`NameGen` provides accurate results and a very handy and reasonable set of parameters, thus facilitating comparison of different data structures. It runs in three consecutive phases: parsing, learning, and generation.

Parsing: The overall parsing process has two interleaved phases. Phase 1 targets to dump/pass invalid URLs, while Phase 2 reformats the URLs into NDN-like names prepared for learning. As mentioned earlier, an NDN name is a hierarchical URL-like name containing a sequence of components delimited by slashes '/'. The current *de facto* NDN URI adopts the percent-encoding method, so NDN names include US-ASCII upper- and lower-case letters (A-Z and a-z), digits (0-9), and four specials: PLUS (+), PERIOD (.), UNDERSCORE (_), and HYPHEN (-), as well as the PERCENT symbol (%) [4].

In Phase 1, during parsing, the URLs are checked by a validator function which simply dumps the invalid URLs (e.g., those including at least a non-ASCII character). Without loss of generality, we also assume that the name length and the number of components do not exceed 1000 characters and 20 components, respectively. Thus, any input URL violating these conditions is simply dumped as well.

In Phase 2, to create NDN-like names from URLs, `NameGen` rewrites each valid URL by the following rules: (1) remove application protocols ('http(s)://', 'ftp://', 'www.', etc.). NDN is an alternative paradigm to replace the Internet layer, so its ideas should not be limited to the World Wide Web; (2) rewrite '//' by '/' if happens during parsing a name as a component of length zero is meaningless; (3) replace '.' in TLDs, such as '.com' and '.org', by '/'[4], and move TLDs to the beginning of the name; and (4) add '/' to the beginning of a name if does not exist and remove '/' from the end if happens. An example transformation is to rewrite URL *https://umich.edu/file1* by NDN name */edu/umich/file1*.

4. The complete list of TLDs can be found online on *https://data.iana.org/TLD/tlds-alpha-by-domain.txt*.



(a) Distribution of length of components

(b) Distribution of number of components

Fig. 4: Dataset statistics and high accuracy of our tool: dataset of 2 million URLs and generated dataset of 10 million URL-like names.

Note that although the parsing may be modified according to the user requirements, our learning phase is fixed for every type of the input dataset.

Learning: `NameGen` is a Markov-based random name generator written in C. A Markov model is an effective way of generating a new record based on the statistical probabilities in a set of sample input records. In its simplest form (a.k.a. order-1 Markov), the model estimates the probability of occurrence of a letter/character in a given position basing the preceding character, which is constituted in the learning process. This way, Markov can be applied to predict the value of i-th element in a given sequence depending on the value of element i-1 [30].

The transition matrix T defines the number of occurrences of each character in the function of all possible preceding characters. The occurrence possibility is then calculated by devising each number by the sum of all the others in the same row. Formally, the probability of occurrence of x_i after x_j ($1 \leq i, j \leq 95$) is calculated by $P(x_i|x_j) = \frac{T[x_j][x_i]}{\sum_{i=1}^{95} T[x_j][x_i]}$. The reason for value 95 is that the most significant bit (MSB) of all characters in the NDN URI scheme is unused, as the ASCII code of printable characters ranges from 33 to 126. Therefore, codes outside of this range (i.e., 0–32 and 127–255) are not included in input names—which means a total of 94 possible characters. Adding one more state for ∅ as the starting state, the associated Markov model has a 95 × 95 transition matrix. To obtain a more refined model, `NameGen` also uses an order-2 Markov for learning, where the number of occurrences of each character after each group of 2 characters in the input names is counted [15]. For example, the name */edu* breaks down into the following order-2 chains: ∅∅ →e, ∅e →d, and ed→u.

The learning process continues for all input names while filling out the transition matrix. Then, `NameGen` can generate random NDN-like names by traversing the Markov chains. Note that the model is trained independently for different components of a name to provide the maximum similarity in each component. Besides the name characters, `NameGen` also learns the number and length of components, which is very important for automating generation and/or keeping the characteristics of new names.

Generation: `NameGen` takes advantage of Markov models and uses a reference collection of names to learn the possible sequences of characters. It offers the user two options for determining the number and length of the components: (1) learning-based (automatic) generation, and (2) optional generation in a possible range based on the general output of the learning process. In both cases, the character generation is according to a learned Markov model. However, the number and length of components can be generated according to learning phase (the former case) or interest of the user (the latter case). Next, `NameGen` generates new names, component by component. In the order-2 model, it starts by state ∅∅ to generate the starting character of each new component. It then moves forward character-by-character while examining the chains and picking a new character from the pool of characters followed the previous 2-character group in the reference dataset until reaching the length of the component. Clearly, transitions between states are based on the possibility of their occurrences, i.e., `NameGen` does not generate something that has not already been happened.

It is desired to increase the order of Markov, but it increases the time complexity of the program for handling a larger size of the transition matrix (95^k for an order-k Markov, where the possibility of occurrence of element x_i is defined by $P(x_i|x_{i-1}, \ldots, x_{i-k})$). Thus, order-2 is a consistently good choice. Moreover, our experiments and others' findings [8], [10] confirm that a large k can cause the generator to regurgitate a lot of the original training input names. This is desirable if the goal is to produce English text or dictionary words, but not for our goal of producing real-like 'random' names.

To prove the correctness of `NameGen` in safely keeping the statistics of the input dataset, Fig. 4 also shows the characteristics of a generated dataset of 10 million names. `NameGen` is shown to work very accurately as the input and generated dataset show very close statistics. Note that the names in input dataset rarely have components 17–20 as shown in Fig. 4(b). Although these last-mile components may also be generated in a new dataset, they can produce the names with the length of more than 1000, which will be discarded as invalid names. That is the reason behind length zero for these components in the generated dataset as shown in Fig. 4(a).

3.3 Datasets

The name characteristic is an important factor in the performance of the trie structures. Thus, besides the fully learning-based dataset (also interchangeably called random dataset),

TABLE 2: Dataset specifications

	SS	LS	SL	LL
# of components	2–5	15–20	2–5	15–20
Length of components	2–5	2–5	50-100	50–100

TABLE 3: System Configuration

CPU	Intel Xeon E5-2683, 2.10 GHz
L1 cache	32 KB
L2 cache	256 KB
L3 cache	40960 KB
DRAM	8×16 GB DDR4 2400 MHz
OS	Ubuntu 16.04.2 LTS

we generate four other types of datasets each with a specific name characteristic: SS, LS, SL, and LL. The first letter stands for the number of the components ('S' for small and 'L' for large) while the second letter stands for the length of the components ('S' for short and 'L' for long). For example, LS represents a dataset of names having a large number of components, each with a short length. Each dataset contains 10M NDN-like names. The collection of datasets occupies more than 15 GB. Table 2 summarizes the specification of each dataset in terms of number and length of components. Note that the generation of names having a length of more than 1000 is possible in both random and LL datasets as the components are generated independently. As mentioned earlier, without loss of generality, we simply dump these names, respecting the maximum length rule for our valid names.

4 Performance Evaluation

We have conducted extensive experiments to evaluate the performance of different trie granularities in terms of memory usage and name processing (i.e., insertion, lookup, and removal) speed. The effect of determining factors, such as trie depth and width, is also demonstrated. We use an Intel Xeon E5 2.10GHz two-socket server platform, with 128GB 2400MHz DDR4 memory, and compare the performance of different granularities. The detailed system configuration is shown in Table 3.

We first present the performance result for the random (fully learning-based) dataset. We then show this is not enough for a clear conclusion on the choice of granularity, as the name characteristics (i.e., the number and length of the components) also play a critical role in the performance. To this end, we carefully evaluate different granularities of trie-based structures under datasets SS, LS, SL, and LL, while highlighting their unique strengths and weaknesses[5].

4.1 On the random dataset

Given below are our testing scenarios and our observations of how different trie granularities perform in terms of speed and memory usage of various processing functions on the random dataset.

5. The absolute performance numbers will differ on different software switches/routers, but the relative merits of the different designs should be preserved. Due to the wide range of technologies and designs employed by hardware platforms, the relative merits of this comparison might not hold on real routers.

TABLE 4: Trie characteristic under random dataset

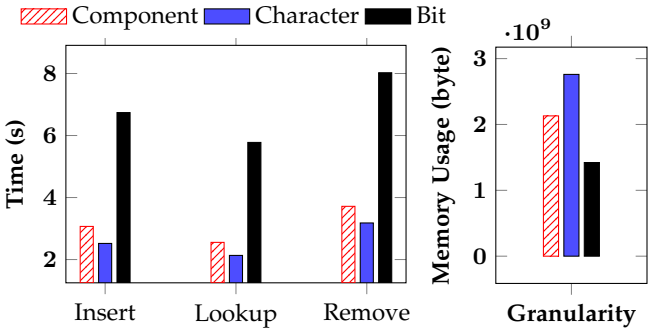
Trie	Component	Character	Bit
Max Height	5	29	119
Avg. Height	2.33	10.95	56.58
# of Leaves	5000000	5000000	5000000
# of Nodes	5246590	7102548	9934489
Avg. Width	21.28	3.38	2
Avg. Hash Table Size	3.85	21.57	-
Avg. Chain Length	3.55	1	-

4.1.1 Speed

Testing scenario: we randomly choose 5M and 1M names from the target dataset (here the random dataset). The former is stored in each trie structure. The latter set is then looked up, inserted, and removed, respectively, in/from the structures. Thus, we report the cumulative cost of 1M name operations for each processing function. Due to randomness, the selected set of 1M may share some names with the constructed trie. To mitigate the effect of randomness, we run all experiments 10 times and report the average. This testing scenario may raise two questions which need to be clarified: (1) why do we measure the cost of each function cumulatively, i.e., for the set of 1M names, not for a single name (or a small set of names)? This is because, although the latter approach seems more straightforward, it is highly susceptible to the changes of name characteristics (such as the length of components) and does not reflect the actual cost of the function itself. Moreover, it cannot mask sudden volatile system changes (e.g., those by OS tasks) on the machine; and (2) why don't we insert, look up, and remove a set of names, respectively, starting from an empty trie, and show the performance results? This is to ensure that the cost of each function is not biased by the changes of trie size. Actually, if name insertion starts with an empty trie and gradually fills up the trie, it runs pretty fast at the beginning while slowing down as the trie getting occupied. Obviously, the costs of inserting a name into an empty trie and a trie with millions of names are very different. Reporting the insertion time this way is misleading, as it is biased by the changes of trie size. The same problem occurs to the name-removal process: it starts slowly with a full trie and speeds up gradually by removing more names, until the trie gets empty. Unlike name insertion and removal, the lookup process cannot benefit the changes of trie size as looking up a set of names does not add or remove any name to/from the trie. Therefore, reporting the construction time (starting with an empty trie and inserting all names) as the insertion time and that of destruction (starting with the full trie and emptying it) as removal time cannot be compared to lookup time. Thus, we run the functions for a set of names (1M names) on already constructed tries to make sure that the cost of each function is kept roughly fixed and the changes in trie size do not affect the results.

Note that, as mentioned earlier, the load factor of component-level trie is set to its optimal trade-off point between speed and memory, which is 10 as discussed in Appendix A.1. Some essential findings are as follows. For simplicity and clarity of the figures, they depict the processing time, which is inversely proportional to speed. Note that sign '>' means 'better' in terms of speed.

Observation 1: *lookup > insertion > removal.*



(a) Speed (b) Memory

Fig. 5: Performance of different trie structures under the random dataset.

Lookup is the initial step of insertion and removal, so it is clearly the fastest processing function. Insertion starts in case of a mismatch even in the middle of traversing the trie. In our testing scenario, removal happens after insertion. Thus, for removal, all 1M names already exist in the structures. Therefore, it needs to traverse the whole names from the root to the children in order to find the exact matches and remove the corresponding leaf nodes, taking more time than the insertion process where the exact matches fail for a part of the names in the middle (sometimes close to the root) of the trie. Finally, as evident from Fig. 5(a), lookup is always better than insertion, than removal in terms of speed. One may ask how insertion and removal perform independent of their current sequence in our testing scenario. As expected for these two reverse functions, our results show that insertion speed is very close to removal speed (see Appendix A.3 for more on this). However, we prefer showing the removal results based on the current testing scenario (i.e., lookup, insertion, and removal, respectively). This way we differentiate the curves of insertion and removal in the figures, which provides a better view of how different trie granularities perform for these two update functions.

Observation 2: *character > component > bit.*

On bit-level trie, which yields the worst processing speed: Table 4 shows the characteristics of each trie structure after storing 5M names. The average height of bit trie is an order-of-magnitude higher than the other two structures. This causes bit trie to yield the worst performance (see Fig. 5(a)) as traversing from each level to the next one in the trie requires one off-chip memory access. We observe an interesting comparison between the impact of the computational overhead and the number of off-chip memory accesses on speed-up. Bit trie incurs much less computational overhead at each node than component and character tries, as in case of bit mismatch, the child to branch to can be determined without any processing. However, reducing the number of off-chip memory accesses has a drastically higher impact on speed than reducing the computational overhead at nodes.

Component vs. bit trie: The height of component trie is shorter while its width is greater than two other structures because the names are less likely to share their beginning components with each other than their beginning characters or bits. Thus, most of the inserted names will remain shallow at the top, increasing the width of top-level nodes (especially the root). (The same reasoning can be applied to compare bit and character tries.) The main idea behind component trie, i.e., expanding the width and minimizing the height, is therefore in sharp contrast to that behind bit trie, i.e., narrowing the width and maximizing the height. If component trie could determine which child to branch to at each level with zero off-chip memory access (like bit trie), it would minimize lookup speed. Although using a hash table is to mimic this behavior, it still requires a few off-chip memory accesses to find the desired children to continue name traversal. By employing a hash table at each node, the width of a node (i.e., the number of its children) will not matter more; instead, the chaining length plays the key role in the performance. By increasing the size of the chain, the number of off-chip memory accesses will increase as traversing each item in the chain triggers one memory access. This is the main drawback of the component trie, which limits its overall speed. As evident from the figure, it still outperforms bit trie on average by 126%.

Character vs. bit trie: Although determining the next child to branch to in bit trie incurs no off-chip memory access, character trie is far better in terms of speed due to its much shorter height. In general, character trie lies somewhere between bit and component tries, where (i) it tries to minimize internal overhead at each node by employing hash tables with no chaining (which mimics zero processing in bit trie), and at the same time (ii) it creates a shallower trie structure, like the component trie. Character trie could aggressively outweigh bit trie, on average, by 170%.

Component vs. character trie: The main benefit of character trie over component trie is fast branching to the desired children for name traversal. This is because the collision in this trie is resolved by re-hashing without chaining as discussed in Section 2.1 (that is also why the chaining length for character trie is always 1 as shown in Table 4). Although character granularity results in a taller trie, incurring a higher number of off-chip memory accesses, component trie has 3.5x higher chain length than character trie. Besides the high internal overhead of level-by-level children traversal alongside the parsing time to extract components, such an increase in the chain length can significantly increase lookup and update times in component trie. Finally, the results indicate that character trie can outperform component trie by an average of 20% in terms of lookup and update speed, respectively.

The significant improvement of character over component trie on the random dataset may mislead to using character trie as the best choice. However, the effect of name characteristic is not considered well. A more in-depth evaluation of the granularities for different types of datasets in the next section provides a clear conclusion on the merit of different granularities in terms of processing speed. Accordingly, character trie does not always show the best speed. Our later conclusion on the choice of granularity will fully support the current results and simply prove the advantage of using character trie for the random dataset.

4.1.2 Memory usage

Here, we compare different data structures on how they save memory for storing the set of 5M names randomly chosen from the random dataset. Note that sign '>' means 'better' in terms of memory usage.

Observation 1: *bit > component > character.*

Bit trie removes a higher redundancy (enjoying a better compactness). Besides, despite having a larger number of nodes, its structure is very simple and memory efficient as each node has at most two pointers towards its children. This is in sharp contrast with two other structures which utilize hash tables at each node for fast children traversal. Thus, binary trie shows the best performance in terms of memory usage (see Fig. 5(b)). Compared with component trie, although character trie is more compact for removing redundant data, it has almost 35% more nodes as well as a much larger size of the hash tables in each node as shown in Table 4. Thus, component trie outperforms character trie on average by 30% in terms of memory usage.

TABLE 5: Trie characteristic in component trie

Dataset	SS	LS	SL	LL
Max Height	5	5	2	2
Avg. Height	2.76	2.83	1	1.000004
# of Leaves	5000000	5000000	5000000	5000000
# of Nodes	5554240	5543842	5000002	5000009
Avg. Width	10.02	10.19	2500000.5	555556.44
Avg. Hash Table Size	1.93	1.92	262144.5	58255.11
Avg. Chain Length	3.60	3.66	5.77	3.06

TABLE 6: Trie characteristic in character trie

Dataset	SS	LS	SL	LL
Max Height	18	18	48	48
Avg. Height	9.71	9.86	13.1	13.1
# of Leaves	5000000	5000000	5000000	5000000
# of Nodes	6952939	6992335	7363865	7363527
Avg. Width	3.56	3.51	3.12	3.12
Avg. Hash Table Size	20.58	20.35	15.37	15.33
Avg. Chain Length	1	1	1	1

TABLE 7: Trie characteristic in bit trie

Dataset	SS	LS	SL	LL
MAX Height	89	88	164	162
Avg. Height	43.61	46.98	54.92	54.93
# of Leaves	5000000	5000000	5000000	5000000
# of Nodes	9607879	10000000	10000000	10000000
Avg. Width	2	2	2	2

4.2 The effect of dataset characteristics

In this section, we present a more in-depth analysis of different trie granularities. Actually, we explore how the statistical characteristics of the namespace in terms of two very relevant features, i.e., (1) length of components for each name, and (2) number of components for each name, can affect lookup and update functions. Four more types of datasets (SS, LS, SL, and LL) are taken into account to determine the rule behind the choice of granularity. We follow the same testing scenario as discussed for the random dataset. To also include the effect of trie size on the performance, for each dataset, we look up, insert, and remove 1M random names, respectively, in/from a trie of size of 1M–5M randomly chosen names (by step 1M). Further essential findings are observed and reported as follows.

4.2.1 Speed

Observation 1: *processing time is ascending for the trie size.*

Storing a higher number of names (from 1M to 5M) results in increasing the size of the trie (both in width and height). A larger trie needs more time for traversal, which causes higher lookup and update times as shown in Fig. 6. Although this is a general behavior, one may ask about the reason behind fluctuating the performance values for insertion time in component trie. The answer is behind re-hashing effect. This is more evident as a spike of insertion time for the component trie of size of 2M names in Fig. 6(c). This observation indicates when a majority of the nodes almost reach their load factor threshold, thus needing to double their hash tables' sizes, which is a very time-consuming operation. From here on, insertion performs faster because not only the hash tables are big enough—and hence there is probably no need for re-hashing, but also probably less names from the set of 1M names need to be inserted in a larger trie. Finally, this trend may sometimes be broken, but in general, keeps its increasing behavior.

Observation 2: *processing time is ascending from Fig. 6(a) to Fig. 6(d).*

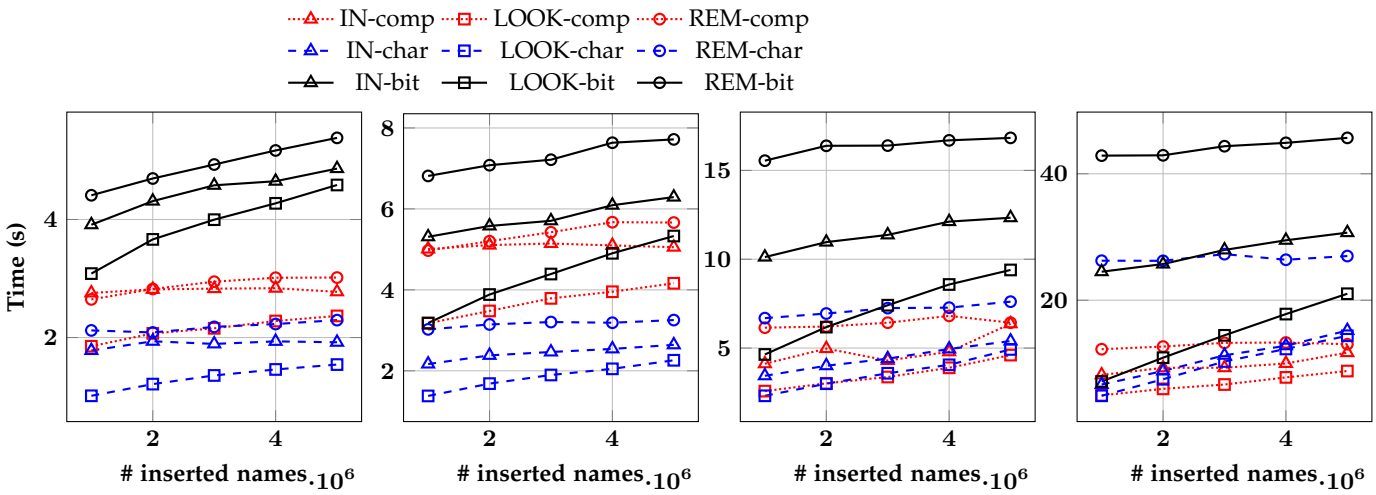
Since the total length of the names in LS is higher than SS, it takes more time to look up (so insert and remove) them in the constructed trie. Actually, traversing longer names is more time-consuming. This increase in lookup and update times is evident from the figures, as the name length expands from LS to SL (where the name length changes in intervals 30–100 and 100–500, respectively). Finally, dataset LL, having the longest names, incurs the maximum processing time.

Observation 3: *character > component for SS and LS, while component > character for SL and LL.*

Tables 5-7 show the characteristics of each trie structure after storing 5M names. For each dataset, bit trie shows the worst performance as its height is an order-of-magnitude higher than two other structures, so incurring a huge number of off-chip memory accesses (see Fig. 6). Surprisingly, character and component tries show different relative performance in different datasets. Here, considering the specific characteristic of each dataset, we analyze them, separately, as follows.

SS and LS: Dataset SS has a similar characteristic to the random dataset (where each name has on average 4 components of length 13), causing close results for almost the same reasons. Actually, despite the higher height of character trie, it outperforms component trie on average by 64% for its higher chaining length and internal overhead of level-by-level children traversal. For LS, increasing the number of components has no significant impact on the structure of the tries. The reason is behind the 'Patricia' type of the tries, where the parents swallow their sole children. Thus, its only effect is to magnify the improvements. As evident from Fig. 6(b), character trie shows a significant improvement (103%) over component trie.

SL and LL: As component trie treats a name as a sequence of components, the names with long components are less likely to share any prefix with each other. The components in SL and LL datasets are very long (of 50-100 characters), causing almost all names to remain at the top of the trie (which results in an average height of 1 in component trie). Anyway, although names in these two datasets share almost zero component with each other, they share many characters, increasing the height of character trie. Thus, 13x taller trie in character trie can dramatically decrease its speed. As the second interesting fact, component trie detects lookup



(a) SS (b) LS (c) SL (d) LL

Fig. 6: Speed of different trie structures under different datasets.

fails much faster than character trie during lookups, especially for names with longer components. Thus, it avoids checking many unnecessary characters for LPM. For example, assume that we need to look up */ndn/file...xyw* while */ndn/file...xyz* has already been inserted in the structure. As discussed in Section 2.1, component-level trie uses a hash function at each node to find the next node to branch to during name traversal. It starts from the root, hashes the first component (i.e., *ndn*) and uses the generated key to find which node to branch to. After branching to node *ndn*, it hashes the second component (i.e., *file...xyw*). However, none of the children have the generated key, so component-level trie raises lookup failure without checking even a single character of the second component. Instead, searching the same name in character-level trie keeps continuing lookup until the character *'z'* to figure out the name does not exist in the structure. This behavior effectively increases the gap between the processing speed of character and component tries. Consequently, unlike in SS and LS, component trie takes the best place in terms of lookup speed and outperforms character trie in SL dataset. The improvement is more highlighted for LL dataset (26% as evident from Fig. 6(d)), where the number of components increases as well. In general, the results show that the length of components dictates the choice of granularity; while the number of components only magnifies the relative performance improvements, and the length of names/prefixes only affects the absolute results.

The cost of re-hashing is also a key factor which highly impacts name insertion speed. Generally speaking, re-hashing is a costlier operation in component trie than in character trie for two reasons: (1) traversing the chained items one-by-one in component trie and shifting them to a new hash table is much time-consuming, and (2) there is no upper bound for the hash table size in component trie, meaning that re-hashing should be done on larger hash tables. Unlike a faster lookup in component trie compared to character trie in dataset SL, it works worse in insertion. This is the effect of re-hashing which frequently happens during insertion of new names in component trie. For dataset LL, the significant improvement of lookup in component trie compensates this result. However, removal in character trie always takes more time than component trie since the exact match on the already inserted names in a much taller trie is required.

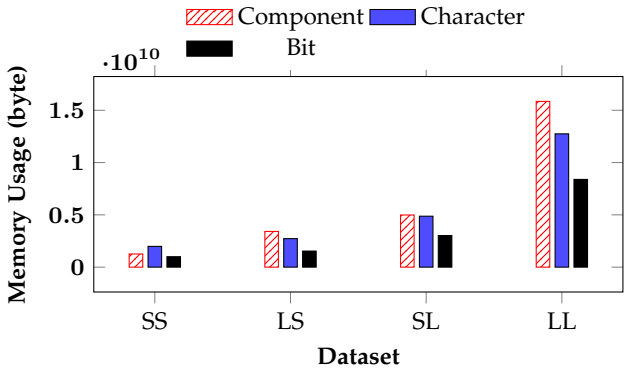
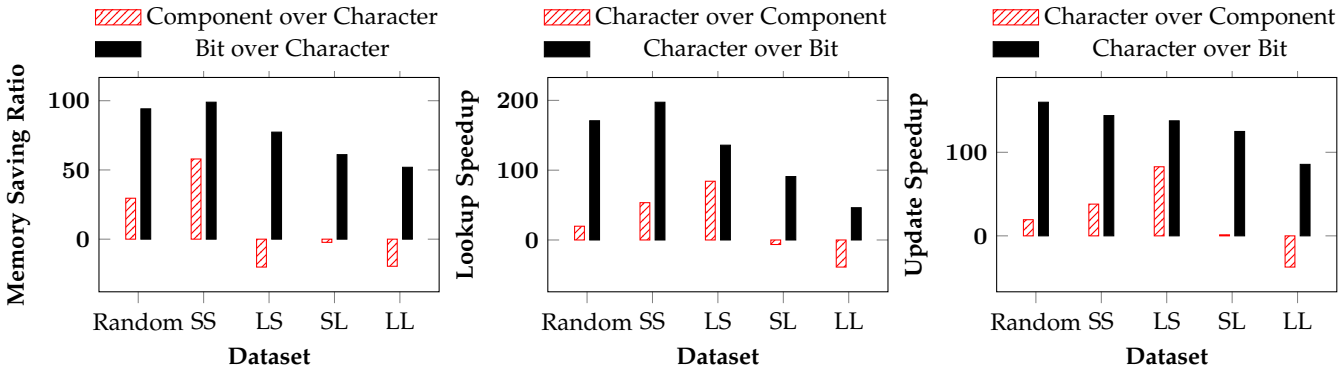


Fig. 7: Memory usage of different trie structures under different datasets.

4.2.2 Memory usage

Observation 1: *Increasing number and length of components waste memory significantly in component trie.*

As the name length increases, more memory is required to store the names. Fig. 7 depicts the increasing memory usage from SS to LL. For the same reason discussed for the random dataset, bit trie always shows the best memory usage. However, a deeper look at the results in Fig. 7 shows that increasing the number of components dramatically increases the memory usage in component trie. On the other hand, the length of components has a significant impact on the memory usage of component trie. Actually, when the component length is large, almost all the names share no prefix in component granularity (component trie has an average height of 1). This means no opportunity for removing redundancy while storing the names in the structure, which maximizes memory footprint.



(a) Memory saving ratio (b) Lookup speedup (c) Update speedup

Fig. 8: Concluding results on the choice of granularity. ("x over y" represents how x outperforms y.)

4.3 Discussion and Conclusion

Fig. 8 shows the concluding results on lookup and update speed as well as memory usage of component, character, and bit tries for the set of 5M names. Neither component trie nor character trie is shown to be able to match the memory efficiency of bit trie. It outperforms character and component tries by more than 70% on average. Thus, bit trie is the choice when the memory requirement is a real concern. On the other hand, it shows the worst performance in terms of processing speed as is outperformed by character and component tries by 128% and 98% on average, respectively. Surprisingly, character and component tries show different relative performances in different datasets: the former works, on average, 80% better for datasets of short-component names (SS and LS), while the latter shows a better performance by 14% on average for datasets of long-component names (SL and LL). As a result, the length of names/prefixes does not affect the choice of granularity. Instead, among name characteristics, the length of components plays a decisive role in the processing performance, while the number of components has a supportive impact. In other words, the length of components (especially the initial ones) determines the choice of granularity, while the number of components only magnifies the relative performance improvements of character and component tries, and the length of names/prefixes only affects the absolute results. Finally, in terms of speed, the character trie is the choice when the names are composed of short components; otherwise, the component trie is preferred for faster lookups and updates.

Having a clear conclusion on the choice of granularity, let's return to the random dataset to see if our conclusion supports its results. The characteristic of the dataset (as shown in Table 1 and Fig. 4), where both the number and length of components are much closer to those in dataset SS, directs us towards using character trie to achieve the fastest speed. The result, as shown in Fig. 5, confirms the correctness of our conclusion.

5 Related Work

Existing name lookup and update designs are mostly based on hash table or trie. Several studies employed lookup structures based on hash tables [16], [19], [23], [38]. Since the hash table only supports exact matching, it needs multiple lookups to find the LPM—this problem is often referred to as "prefix seeking". For example, for a name */ndn/ua/cs/conf*, the LPM process can start from the shortest prefix (i.e., */ndn*) and increment by one component at a time, start from the longest (i.e., */ndn/ua/cs/conf*) and decrement by one component at a time, start from a particular length and then continue to shorter prefixes or restart at a larger prefix [28], or even start from the middle and do a binary search for the longest matching prefix [35]. Besides the prefix seeking, another problem of the hash-table-based approach is its large memory footprint. Since hash tables need to store the name strings for the purpose of collision resolution and verification, the names often need to be stored in their entirety. Even when multiple names share the same prefixes, these shared prefixes are still stored multiple times, thus consuming more memory. Several schemes [12], [18], [22] used bloom filters along with hash tables to improve the lookup speed and memory usage. Their benefit of reduced memory usage often comes at the expense of key verification and collision resolution. False positives will prevent the packets from being forwarded to the right destination. Finally, it is unattractive to incur such overheads (both in memory usage and number of memory accesses) to the system in order to use an exact matching structure for LPM. This inherent mismatch causes structural complexity for higher performance.

Trie, especially Patricia trie, is widely used in forwarding table implementation for its memory efficiency. Using this structure, searching for names that share the same prefix only needs to save one copy of the shared part. It also naturally supports both exact match and LPM. There are several existing proposals that implement name-based forwarding tables using trie structures. Parallel Name Lookup (PNL) [31] established component-level Name Prefix Tree (NPT) while speeding up lookups by selectively grouping trie nodes and allocating them to parallel memories. Name Component Encoding (NCE) [32] improves NPT by encoding name components and then looking up these codes to find the LPM. It uses the State Transition Arrays (STA) to implement both the Component Character Trie (CCT) and Encoded Name Prefix Trie (ENPT), thus effectively reducing the memory requirement while accelerating lookup speed. He *et al.* [11] used ENPT along with an improved version of STA, called *Simplified STA*, and a hash-table-based code allocation function. However, the encoding function is the

performance bottleneck in both designs. Ghasemi *et al.* [13] proposed a new character-level trie data structure to store and index forwarding table entries efficiently and to support fast packet forwarding. They combined the power of arrays and hash tables in an optimized manner to speed up name lookups and updates while using a new scheme to encode control information without using additional memory. Wang *et al.* [33] dealt with this bottleneck by leveraging the massive parallel processing power of GPU to achieve faster table lookups. Quan *et al.* [24] proposed an Adaptive Prefix Bloom filter (NLAPB) in which the first part of a name is matched by a bloom filter, and the later part is processed by a trie. Recently, Song *et al.* [29] proposed a bit-level Patricia trie that can compress the forwarding table significantly in order to fit it into high-speed memory modules. Yuan *et al.* [36] adopted the main idea of [29] and proposed hash-table-based and trie-based data structures, accordingly.

Despite a number of implementations of tries in the literature, none of them studied the choice of granularity, which can dramatically affect the network performance in terms of memory usage and packet-processing speed.

6 Conclusion

The choice of granularity for lookup/update structures can greatly influence the network performance, which has not yet been studied well. This paper makes three main contributions: (1) our implementations of different trie schemes establish the baselines for performance evaluation (see Appendix A.4); (2) a new tool, `NameGen`, is developed to learn real (even small) datasets and generate a large collection of datasets with different (learning-based and/or user-defined) name characteristics in terms of both number and length of components, which is important especially for evaluation of new network architectures like NDN; and (3) our extensive experiments highlight the unique strengths and weaknesses of each scheme of trie-based structures and draws a conclusion on the choice of granularity.

Surprisingly, only one of the many factors that can affect the absolute performance is found to dictate the merit of the structures in terms of speed: component length. In other words, the length of components (especially the initial ones) plays a decisive role in the choice of granularity, while the number of components only magnifies the relative performance improvements of character and component tries, and the length of names/prefixes only affects the absolute results. In conclusion, the character trie is the choice for speed when the names are composed of short components; otherwise, the component trie is preferred for faster lookups and updates. However, in terms of memory usage, the bit-level trie provides the best performance, thus making it the choice when the memory requirement is a real concern.

7 Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1629009 and a Huawei grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

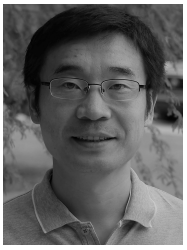
References

[1] Blacklist. http://www.shallalist.de, 2015. [Online].
[2] ODP–Open Directory Project. http://www.dmoz.org/, 2015. [Online].
[3] Alexa the Web Information Company. http://www.alexa.com/, 2017. [Online].
[4] NDN Name Format. https://named-data.net/doc/ndn-tlv/name.html#ndn-name-format, 2017. [Online].
[5] Scrapy: A fast and powerful scraping and web crawling framework. https://scrapy.org/, 2017. [Online].
[6] xxHash–extremely fast non-cryptographic hash algorithm. http://cyan4973.github.io/xxHash/, 2017. [Online].
[7] S. O. Amin, Q. Zheng, R. Ravindran, and G. Wang. Leveraging ICN for secure content distribution in IP networks. In *ACM Multimedia Conference (MM'16)*, pages 765–767, 2016.
[8] J. Bentley. *Programming Pearls (2Nd Ed.)*. ACM Press/Addison-Wesley Publishing Co., 2000.
[9] G. Carofiglio, L. Muscariello, M. Papalini, N. Rozhnova, and X. Zeng. Leveraging ICN in-network control for loss detection and recovery in wireless mobile networks. In *ACM ICN*, pages 50–59, 2016.
[10] H. Crawford and J. Aycock. Kwyjibo: Automatic domain name generation. *Softw. Pract. Exper.*, 38(14):1561–1567, 2008.
[11] H. Dai, B. Liu, Y. Chen, and Y. Wang. On pending interest table in named data networking. In *ACM/IEEE ANCS*, pages 211–222, 2012.
[12] H. Dai, J. Lu, Y. Wang, T. Pan, and B. Liu. BFAST: High-speed and memory-efficient approach for NDN forwarding engine. *IEEE/ACM Transactions on Networking*, 25(2):1235–1248, 2017.
[13] C. Ghasemi, H. Yousefi, K. G. Shin, and B. Zhang. A fast and memory-efficient trie structure for name-based packet forwarding. In *IEEE ICNP*, 2018.
[14] M. Hershcovitch and H. Kaplan. I/O efficient dynamic data structures for longest prefix queries. *Algorithmica*, 65(2):371–390, 2013.
[15] R. Lawrence. Random word generator. 2006.
[16] X. Li and W. Feng. Two effective functions on hashing URLs. *Journal of Software*, 15:179–184, 2004.
[17] Y. Li, D. Zhang, X. Yu, W. Liang, J. Long, and H. Qiao. Accelerate NDN name lookup using FPGA: Challenges and a scalable approach. In *FPL'14*, pages 1–4, 2014.
[18] H. Lim, M. Shim, and J. Lee. Name prefix matching using bloom filter pre-searching. In *ACM/IEEE ANCS*, pages 203–204, 2015.
[19] B. Michel, K. Nikoloudakis, P. Reiher, and L. Zhang. URL forwarding and compression in adaptive web caching. In *IEEE INFOCOM*, volume 2, pages 670–678, 2000.
[20] I. Moiseenko and D. Oran. Path switching in content centric and named data networks. In *ACM ICN*, pages 66–76, 2017.
[21] D. R. Morrison. Patricia— practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
[22] D. Perino, M. Varvello, L. Linguaglossa, R. Laufer, and R. Boislaigue. Caesar: A content router for high-speed forwarding on content names. In *ACM/IEEE ANCS*, pages 137–147, 2014.
[23] Z. G. Prodanoff and K. J. Christensen. Managing routing tables for URL routers in content distribution networks. *International Journal of Network Management*, 14:177–192, 2004.
[24] W. Quan, C. Xu, J. Guan, H. Zhang, and L. A. Grieco. Scalable name lookup with adaptive prefix bloom filter for named data networking. *IEEE Communications Letters*, 18(1):102–105, 2014.
[25] R. Ravindran, A. Chakraborti, S. O. Amin, A. Azgin, and G. Wang. 5G-ICN: Delivering ICN services over 5G using network slicing. *IEEE Communications Magazine*, 55(5):101–107, 2017.
[26] J. Samain, G. Carofiglio, L. Muscariello, M. Papalini, M. Sardara, M. Tortelli, and D. Rossi. Dynamic adaptive video streaming: Towards a systematic comparison of ICN and TCP/IP. *IEEE Transactions on Multimedia*, 19(10):2166–2181, 2017.
[27] M. Sardara, L. Muscariello, J. Augé, M. Enguehard, A. Compagno, and G. Carofiglio. Virtualized ICN (vICN): Towards a unified network virtualization framework for ICN experimentation. In *ACM ICN*, pages 109–115, 2017.
[28] W. So, A. Narayanan, and D. Oran. Named data networking on a router: Fast and DoS-resistant forwarding with hash tables. In *ACM/IEEE ANCS*, pages 215–226, 2013.
[29] T. Song, H. Yuan, P. Crowley, and B. Zhang. Scalable name-based packet forwarding: From millions to billions. In *ACM ICN*, pages 19–28, 2015.
[30] T. V. Vleck. GPW: Password generator. 2006.

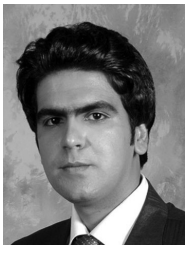
[31] Y. Wang, H. Dai, J. Jiang, K. He, W. Meng, and B. Liu. Parallel name lookup for named data networking. In *IEEE GLOBECOM*, pages 1–5, 2011.
[32] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, and Y. Chen. Scalable name lookup in NDN using effective name component encoding. In *IEEE ICDCS*, pages 688–697, 2012.
[33] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang. Wire speed name lookup: A GPU-based approach. In *NSDI*, pages 199–212, 2013.
[34] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee IP lookup performance with FIB explosion. *SIGCOMM Comput. Commun. Rev.*, 44(4):39–50, 2014.
[35] H. Yuan and P. Crowley. Reliably scalable name prefix lookup. In *ACM/IEEE ANCS*, pages 111–121, 2015.
[36] H. Yuan, P. Crowley, and T. Song. Enhancing scalable name-based forwarding. In *ACM/IEEE ANCS*, pages 60–69, 2017.
[37] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, k. claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang. Named data networking. *ACM SIGCOMM Comput. Commun. Rev.*, 44(3):66–73, 2014.
[38] Z. Zhou, T. Song, and Y. Jia. A high-performance URL lookup engine for URL filtering systems. In *IEEE ICC*, pages 1–5, 2010.

Kang G. Shin received the BS degree in electronics engineering from Seoul National University, Seoul, Korea, and the MS and the PhD degrees in electrical engineering from Cornell University, Ithaca, New York, in 1970, 1976, and 1978, respectively. He is the Kevin and Nancy O'Connor Professor of Computer Science and founding director of the Real-Time Computing Laboratory in the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan. At Michigan, he has supervised the completion of 80 PhDs and also chaired the Computer Science and Engineering Division for three years starting in 1991. From 1978 to 1982 he was on the faculty of the Rensselaer Polytechnic Institute, Troy, New York. His current research focuses on QoS-sensitive computing and networks as well as on embedded real-time and cyberphysical systems. He has authored/coauthored more than 900 technical articles (more than 330 of which are published in archival journals) and more than 40 patents or invention disclosures. He has also received numerous institutional awards and best paper awards. He is a life fellow of the IEEE.

Chavoosh Ghasemi is a PhD student in the Department of Computer Science at the University of Arizona. He graduated from Sharif University of Technology in 2014, majoring in Information and Communication Technology Engineering. His research interests include Information Centric Networks, routing and forwarding protocol design, Content Delivery Networks, name lookup data structures, and video streaming QoS.



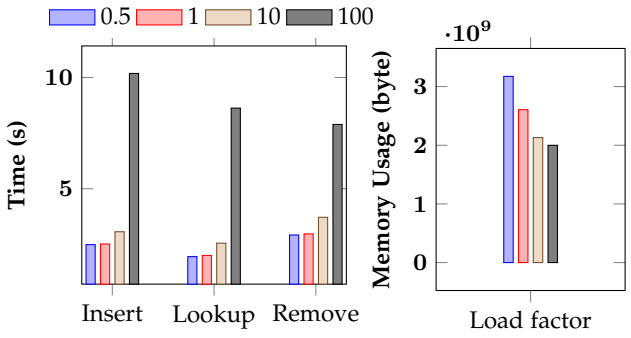
Beichuan Zhang received the B.S. degree from Peking University, Beijing, China, in 1995, and the Ph.D. degree from the University of California, Los Angeles (UCLA), CA, USA, in 2003. He is an Associate Professor in the Department of Computer Science, University of Arizona, Tucson, AZ, USA. His research interest is in Internet routing architectures and protocols. He has been working on named data networking, green networking, inter-domain routing, and overlay multicast. Dr. Zhang received the first Applied Networking Research Prize in 2011 by ISOC and IRTF, and the Best Paper Award at IEEE ICDCS in 2005.



Hamed Yousefi is currently a Staff Researcher at Huawei R&D USA. From 2016 to 2018, he was a Postdoctoral Research Fellow at the Department of Electrical Engineering and Computer Science, University of Michigan. He received his PhD in Computer Engineering from Sharif University of Technology in 2015. From 2013 to 2014, he was a visiting scholar in computer science at the University of Michigan. His research interests include future Internet/network architectures, information-centric networks, content delivery networks, wireless sensing systems and networks, and reliable and real-time networking.

Supplement of "On the Granularity of Trie-based Data Structures for Name Lookups and Updates"

Chavoosh Ghasemi*, Hamed Yousefi*, Kang G. Shin, and Beichuan Zhang



(a) Speed (b) Memory usage

Fig. 1: Choice of load factor in component trie makes a trade-off between performance metrics.

APPENDIX

.1 Choice of load factor for component trie

The choice of *load factor*, defined as the number of children in each node divided by the size of its hash table in component trie, incurs different time and space complexities, creating a trade-off between memory usage and speed. A lower load factor results in a higher hash table size, thus increasing the memory footprint. Instead, it reduces chaining length and hence needs less sequential search in chains, improving the lookup speed. On the other hand, a higher load factor saves more memory while increasing the lookup time. (The hash table works like a linked list if the load factor approaches infinity). Fig. 1 shows the speed and memory usage for different load factors (0.5, 1, 10, 100) in component trie where 1M random names are looked up and updated in a trie of size of 5M names. Load factor 10 represents the most balanced trade-off, saving lots of memory while slightly increasing the lookup/update time. In other words, the processing speed for this load factor is very close to those for load factors 0.5 and 1, while using memory like that for load factor 100. Thus, the load factor of component-level trie is set to 10 in the rest of our experiments.

- * *Co-primary authors.*
- *C. Ghasemi is with the Department of Computer Science, The University of Arizona. E-mail:* chghasemi@cs.arizona.edu
- *H. Yousefi is with the Department of Electrical Engineering and Computer Science, The University of Michigan. E-mail:* hyousefi@umich.edu
- *K.G. Shin is with the Department of Electrical Engineering and Computer Science, The University of Michigan. E-mail:* kgshin@umich.edu
- *B. Zhang is with the Department of Computer Science, The University of Arizona. E-mail:* bzhang@cs.arizona.edu

.2 Complexity Analysis

Here, we analyze both the time and space complexities of the name processing functions for different trie implementations. We assume that the total number of names is N and all names are unique.

.2.1 Bit-level trie

Lookup: To look up a name in a bit-level trie, we start from the root and traverse down the trie while checking each node bit-by-bit until a mismatch occurs. The child to branch to at each level is determined with zero processing and is fetched to the cache with 1 off-chip memory access. Assuming the average length of each node is m bits, looking up N names requires $O(N \times m \times \log_2 N)$ time with $O(N \times \log_2 N)$ off-chip memory accesses.

Insertion: After finding LPM, the remaining bits of the queried name will be inserted into a new node. Adding a new node as well as splitting the current node takes $O(\frac{m}{2} + (M - m \times \log_2 N))$ on average and requires 2 off-chip memory accesses to insert the children, where M is the length of the queried name in bits. Thus, the time complexity of inserting N names in a bit-level trie is $O(N \times (\frac{m}{2} + M))$ with $O(N \times \log_2 N)$ off-chip memory accesses.

Remove: After finding the queried name, the corresponding leaf node will be deleted and its parent will merge with the other child. This process requires $O(m \times \log_2 N)$ time to find the node, $O(m)$ time for merging, and 1 off-chip memory access to fetch the remaining child. Thus, removing N names takes $O(N \times m \times \log_2 N)$ with $O(N \times \log_2 N)$ off-chip memory accesses.

Space complexity: Each node in a bit-level trie stores a sequence of bits and two pointers (each 8 bytes) to the left and right children. Thus, the memory footprint of a bit-level trie is $O(N \times m)$ bytes.

.2.2 Character-level trie

Lookup: Traversing the character-level trie downward from the root incurs 2 off-chip memory accesses at each node (including fetching the hash table of the current node and

the child to branch to). Assuming the average length of each node is m bytes and the average height of the trie is $\log_x N$ ($1 < x \leq 94$), the time complexity of looking up N names is $O(N \times m \times \log_x N)$ with $O(N \times \log_x N)$ off-chip memory accesses. Note that x shows the fan-out, which is 3 on average according to our evaluations (see Table 6).

Insertion: After finding LPM, the remaining characters of the queried name will be added to a new node. Assuming the length of the queried name is M bytes and that node splitting is necessary, the time complexity of insertion will be $O(\frac{m}{2} + (M - m \times \log_x N))$ on average with 3 off-chip memory accesses to update the current node's hash table and create two children. At each node, the maximum number of rehashing times is $\log_2 94$ and a constant number of buckets (i.e., < 94) must be handled during each rehashing. Thus, rehashing can be done in constant time. Finally, inserting N names in a character-level trie takes $O(N \times (\frac{m}{2} + M))$ with $O(N \times \log_x N)$ off-chip memory accesses.

Remove: To remove a name, we need to find it and delete the resolved leaf from the trie. If, after deletion of a leaf, the parent only has one child, they will be merged together. Deleting a node requires 2 off-chip memory accesses (to update the parent's hash table and fetch the other child to the cache for merging). Furthermore, merging (if necessary) incurs $O(m)$ time to copy the child's characters to its parent. Thus, the time complexity of removing N names is of $O(N \times m \times \log_x N)$ with $O(N \times \log_x N)$ off-chip memory accesses.

Space complexity: Each node in a character-level trie stores a sequence of characters and one hash table. Each record in a hash table requires 9 bytes, including an index (1 byte) and a pointer (8 bytes). Assuming the average number of buckets in a hash table is h at each node, the space complexity of a character-level trie is $O(N \times (h+m))$ bytes. It is worth noting that the average value of h is 20 according to our evaluations (see Table 6).

.2.3 Component-level trie

Lookup: Name lookup in a component-level trie checks the queried name byte-by-byte while traversing down the trie. At each node, it incurs c+1 off-chip memory accesses, where c is the average load factor of the hash tables. Thus, the number of off-chip memory accesses to look up a name will be $O(c \times log_x N)$. According to our evaluations, x can be very large. Therefore, looking up N names takes $O(N \times m)$ with $O(N \times c)$ off-chip memory accesses, where m is the average node length in bytes.

Insertion: After finding LPM, the remaining components of the queried name will be inserted into a new node. Splitting the current node and adding a new node require $O(M - \frac{m}{2})$ time and $O(c)$ off-chip memory accesses, respectively, where M is the length of the queried name in bytes. Moreover, rehashing at each node incurs $h \times c$ off-chip memory accesses, where h is the average number of buckets in the hash tables. Thus, the total number of off-chip memory accesses to insert N names will be $O(N \times (\log_2 h \times c \times h))$, where $\log_2 h$ shows the number of rehashing times to reach the size of h. Thus, inserting N names takes $O(N \times M)$ with $O(N \times (\log_2 h \times c \times h))$ off-chip memory accesses.

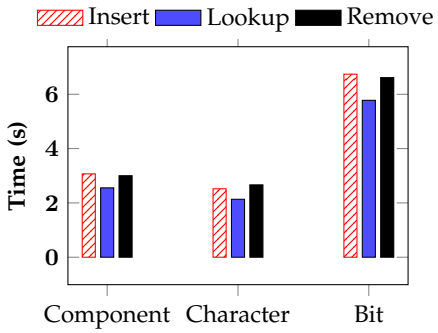


Fig. 2: Removal performs close to insertion if no priority is given in running these two update functions.

Remove: After finding the queried name, the associated leaf node will be deleted. If the parent is left with a single child, the two nodes will be merged. Deleting a leaf node requires $c + 2$ off-chip memory accesses to update the parent's hash table and fetch the node to delete. Furthermore, the merging procedure takes $O(m)$ time with 1 off-chip memory access to fetch the remaining child to the cache. Thus, removing N names requires $O(N \times m)$ time with $O(N \times c)$ off-chip memory accesses.

Space complexity: Each record in the hash table has an index and a singly-linked list of elements. Each element has a pointer to a child (8 bytes), a hash value (8 bytes), and a pointer to the next element in the list (8 bytes). Thus, the average size of a hash table is $h \times (c \times 24)$ bytes. Finally, the average memory footprint of a component-level trie is $O(N \times (h \times c + m))$ bytes.

.3 Insertion vs. removal

Fig. 2 compares different schemes if no priority is given in running the processing functions. Unlike the main testing scenario (see Section **??**) in which names are removed after their insertion, we randomly choose 1M names from the random dataset and remove them from the structures of 5M randomly chosen names. As in the insertion scenario, a half of the names probably exist in the trie (composed of 5M of 10M names in the random dataset). Our results in such a fair scenario (i.e., for an equal number of insertions and removals) show that using each trie scheme, the insertion speed is very close to the removal speed, because insertion and removal work similarly, but in reverse directions. Note that for a smaller size (less than 5M) of trie, the major parts of names do not exist. This will result in a higher insertion time than the removal time as the latter cannot find the exact matches for major parts of the names. Therefore, in such a case, removal performs closer to lookup as the fastest processing function.

.4 Our trie implementations as baselines

Although some efforts have been made to establish a baseline for comparison of name lookup schemes [**?**], no implementation baseline exists to allow comparison between tries of different granularities for both name lookups and updates. One of our major contributions in this paper is the implementation of data structures. Below, we articulate some of the important features that can turn these implementations into the baseline/reference:

1) The data structures are comparable, meaning that they all use the same technologies, libraries, and modules. It is, of course, unfair to compare data structures developed in different environments (like C and JVM) using different technologies.
2) None of the structures are biased, meaning that each structure is developed according to its well-known design and no one is optimized to show better performance. For instance, instead of linked lists, the hash table is utilized in character and component tries, as it is commonly used and intuitive. However, we did not employ any optimization to help a data structure outperform the others.
3) There are no known source code for the tries in different granularities. We have open-sourced our implementation of these data structures along with a comprehensive documentation of each piece of the code at `https://github.com/chavoosh/TrieGranularity`.
4) The source code of each structure is modular, facilitating the modification of its features. For example, one can change the parsing method of the current version of component trie by only replacing the parsing module. This will give developers enough flexibility to employ their own modules without touching the rest of the code.
5) `NameGen` enables end-users to tune the characteristics of outcome dataset, while one can add more parameters and factors based on his requirements and applications.

This package of beyond 20k lines of C code can become a baseline for the researchers and developers for comparison or optimizations.