KUBERNETES IP-TABLES TIME COMPLEXITY USING TRIE TREE AND RADIX TREE IMPLEMENTATION

Kishore Kumar Jinka[1] , Dr. B.Purnachandra Rao[2]
[1]GlobalLogic Inc, VA, USA kjinkaiitb@gmail.com
[2]Sr.Solutions Architect, HCL Technologies, Bangalore, Karnataka, India. pcr.bobbepalli@gmail.com

Abstract

Kubernetes (K8s) is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. Developed originally by Google and now managed by the Cloud Native Computing Foundation (CNCF), Kubernetes has become the de facto standard for container management due to its scalability, flexibility, and reliability in running production-grade workloads. Containers package applications and their dependencies in isolated environments, ensuring that they run the same regardless of the host environment. Docker is one of the most well-known container platforms, but others like rkt and CRI-O are also compatible with Kubernetes. Service abstraction refers to how Kubernetes abstracts the way applications running inside the cluster are exposed to the outside world or internally within the cluster. A Service in Kubernetes is an abstraction layer that defines a logical set of Pods and a policy by which to access them.

*The main goal of the service abstraction is to decouple the application logic from the actual deployment of Pods, allowing the application to scale or self-heal without requiring manual updates to other parts of the infrastructure. In Kubernetes, IP Tables plays a key role in how networking is managed, particularly in terms of routing traffic to Pods and Services. Kubernetes uses IPTables (via the Linux kernel) in several key components to ensure smooth communication within the cluster and to external systems. Kubernetes uses IPTables to implement the **Service** abstraction. When you create a Service, Kubernetes sets up IPTables rules to route traffic to the correct set of Pods.*

For a ClusterIP service, Kubernetes creates IP Tables rules that intercept traffic to the service's IP and port, then routes the traffic to one of the Pods that match the service's selector. This enables round-robin load balancing between Pods. Existing kuberenets is using Trie tree implementation for IP tables for matching the search criteria. The time complexity of Trie tree implementation is O(m) where m is the length of the key. In this paper we will prove that the time complexity improvement of ip tables by using the radix tree implementation..

***Keywords**: Kubernetes (K8S), Cluster, Nodes, Deployments, Pods, ReplicaSets, Statefulsets, Service, IP-Tables, Trie Tree, Radix Tree, Load Balancer, Service Abstraction.*

INTRODUCTION

Kubernetes consists of several components that work together to manage containerized applications. Master Node: This controls the overall cluster, handling scheduling and task coordination.API Server: Frontend that exposes Kubernetes functionalities through RESTful APIs. Scheduler: Distributes work across the nodes based on workload requirements..Controller Manager: Ensures that the current state matches the desired state by managing the cluster's control loops.etcd: Kube-proxy: Manages network communication within and outside the cluster.

Pod is the smallest deployable unit in Kubernetes, encapsulating one or more containers [1] with shared storage and network resources. All containers in a pod run on the same node.Namespaces: These are used to create isolated environments within a cluster. They allow teams to share the same cluster resources without conflicting with each other. Deployment: A higher-level abstraction that manages the creation and scaling of Pods. It also allows for updates, rollbacks, and scaling of applications. ReplicaSet [2] ensures a specified number of replicas (identical copies) of a Pod are running at any given time. StatefulSet: Designed to manage stateful applications, where each Pod has a unique identity and persistent storage, such as databases. DaemonSet: Ensures that a copy of a Pod is running on all (or some) nodes. This is useful for deploying system services like log collectors or monitoring agents.Job: A Kubernetes resource that runs a task until completion. Unlike Deployments or Pods, a Job does not need to run indefinitely.CronJob: Runs Jobs at specified intervals, similar to cron jobs in Linux.

LITERATURE REVIEW

Kubernetes Cluster

A **cluster** [3] refers to the set of machines (physical or virtual) that work together to run containerized applications. A cluster is made up of one or more **master nodes** (control plane) and **worker nodes**, and it provides a platform for deploying, managing, and scaling containerized workloads.

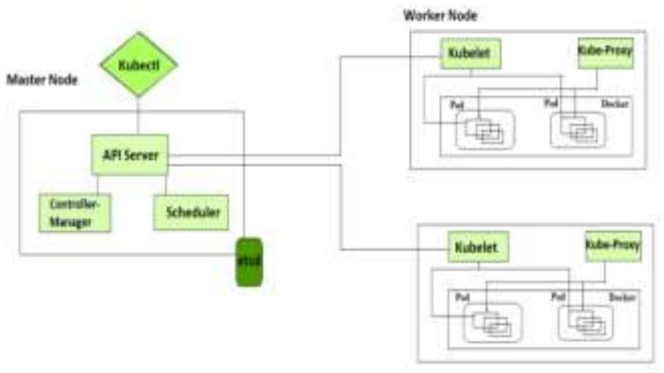


Fig 1. Kubernetes cluster Architecture

Client kubectl will connect to API server [4] (part of Master Node) to interact with Kubernetes resources like pods, services, deployment etc. Client will be authenticated through API server having different stages like authentication and authorization. Once the client is succeeded though authentication [5] and authorization (RBAC plugin) it will connect with corresponding resources to proceed with further operations. Etcd [6] is the storage location for all the kubernetes resources. Scheduler will select the appropriate node for scheduling the pods unless you have mentioned node affinity (this is the provision to specify the particular node for accommodating the pod). Kubelet is the process which is running on all nodes of the kubernetes cluster and it will manage the mediation between api server and corresponding node. Communication between any entity with master node is going to happen only through api server.

Key Components of a Kubernetes Cluster:

Control Plane (Master Node):

API Server exposes Kubernetes APIs. All interactions with the cluster (e.g., deploying applications, scaling, etc.) go through the API server.

Etcd is a distributed key-value store [7] that holds the state and configuration of the cluster, including information about pods, services, secrets, and configurations.

Controller Manager ensures that the cluster's desired state matches its actual state, by managing different controllers (like deployment, replication, etc.).

Scheduler is the one which Assigns workloads to worker nodes based on resource availability, scheduling policies, and requirements.

Worker Nodes:

Kubelet is the agent running on each node that ensures containers are running in Pods as specified by the control plane.

Container Runtime [8] is the software responsible for running containers (e.g., Docker, containerd).

Kube-proxy manages network traffic between pods and services, handling routing, load balancing, and network rules.

How a Kubernetes Cluster Works:

Pods: The smallest deployable units in Kubernetes, consisting of one or more containers. They run on worker nodes and are managed by the control plane.

Nodes: Physical or virtual machines in the cluster that host Pods and execute application workloads.

Services: Provide stable networking and load balancing for Pods within a cluster.

Cluster Operations:

Kubernetes clusters can automatically scale up or down by adding/removing nodes or pods.

Resilience: Clusters are designed for high availability and can automatically restart failed pods or reschedule them on healthy nodes.

Kubernetes ensures traffic is evenly distributed across Pods within a Service.

Self-Healing: The control plane continuously monitors the state of the cluster and acts to correct failures or discrepancies between the desired and current state.

Service Abstraction:

Service Abstraction [9] in Kubernetes provides a way to define a logical set of Pods and a policy by which to access them. This abstraction enables communication between different application components without needing to know the underlying details of each component's location or state.

Stable Network Identity: Services provide a stable IP address and DNS name that can be used to reach Pods, which may be dynamically created or destroyed.

Load Balancing: Kubernetes services automatically distribute traffic to the available Pods, providing a load balancing mechanism. When a Pod fails, the service can route traffic to other healthy Pods.

Service Types: Kubernetes supports different types of services:

ClusterIP: The default type, which exposes the service on a cluster-internal IP. Only accessible from within the cluster.

NodePort: Exposes the service on each Node’s IP at a static port (the NodePort). This way, the service can be accessed externally.

Kubernetes automatically provisions a load balancer for the service when running on cloud providers.

ExternalName maps the service to the contents of the externalName field (e.g., an external DNS name).

Iptables Coordination:

Iptables [10] is a user-space utility program that allows a system administrator to configure the IP packet filter rules of the Linux kernel firewall. In the context of Kubernetes, iptables is used to manage the networking rules that govern how traffic is routed to the various services.

SNo	IP Address	Port
1	10.3.4.3, 10.3.4.5,10.3.4.7	8125
2	10.3.5.3, 10.3.5.5,10.3.5.7	8081
3	10.3.6.3, 10.3.6.5,10.3.6.7	8080
4	10.3.2.3, 10.3.2.5,10.3.2.7	5432
5	10.3.7.3, 10.3.7.5,10.3.7.7	6212
6	10.3.8.3, 10.3.8.5,10.3.8.7	6515

Table 1: IP Tables Storage Structure

Key Functions:

Traffic Routing: Iptables rules direct incoming traffic to the correct service IP based on the defined service configurations.

NAT (Network Address Translation): Iptables can be configured to rewrite the source or destination IP addresses of packets as they pass through, which is crucial for services that need to expose Pods to external traffic.

Connection Tracking: Iptables tracks active connections and ensures that replies to requests are sent back to the correct Pod.

Service and IP Table:

Service Request: A request is sent to the service's stable IP address.

Kubernetes Networking [11] uses iptables to manage the routing of this request. It sets up rules to map the service IP to the IP addresses of the underlying Pods.

Load Balancing: Ip tables distributes incoming traffic among the Pods that match the service's selector, ensuring load balancing. Return Traffic: When a Pod responds, iptables ensures that the response goes back through the same network path, maintaining connection tracking.

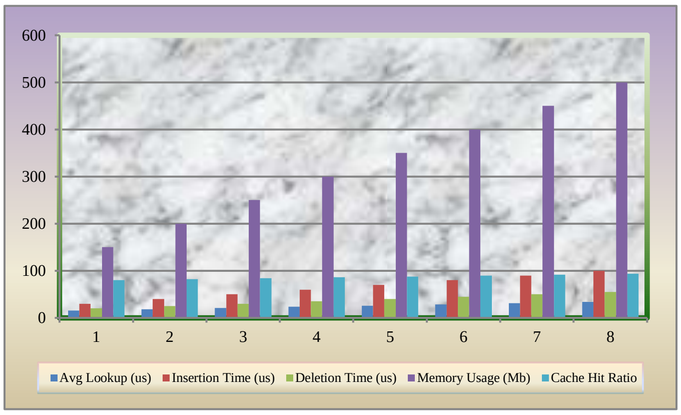
Service abstraction in Kubernetes provides a simplified and stable interface for accessing application components, while iptables coordination ensures that the network traffic is efficiently routed to the right Pods. Together, they form a robust networking framework that is fundamental to the operation of Kubernetes clusters.

Three node , four node , five node , six node , seven node , eight node , nine node and ten node clusters have been configured with 32 CPU, 64 GB and 500GB for master node and 24 CPU , 32 GB and 350 GB for all worker nodes. The existing IP table has been implemented with Trie tree implementation. A **Trie Tree** [12], also known as a **Prefix Tree** [13] [24] [25] [26], is a specialized tree data structure used to store associative data structures, often to represent strings. The key characteristic of a Trie is that all descendants of a node share a common prefix of the string associated with that node. This structure is particularly useful for tasks that involve searching for prefixes, such as auto complete systems, dictionaries, and IP routing tables.

SNo	Size	Avg Lookup (us)	Insertion Time (us)	Deletion Time (us)	Memory Usage (Mb)	Cache Hit Ratio
3	30000	15.6	30	20	150	80
4	40000	18.2	40	25	200	82
5	50000	20.8	50	30	250	84
6	60000	23.4	60	35	300	86
7	70000	26.0	70	40	350	88
8	80000	28.6	80	45	400	90
9	90000	31.2	90	50	450	92
10	100000	33.8	100	55	500	94

Table 2: IP Tables Trie Tree

Please find the number of entries in IP table , avg lookup time, Insertion time , deletion time , memory usage and cache hit ratio for different cluster configurations. For IP table size 30k the avg lookup time is 15.6 us , insertion time is 30 us, deletion time is 20 , memory usage is 150 Mb and cache hit ration is 80% . IP table size 40k the same parameters are 18.2, 40us, 25us, 200Mb and 82%. IP table size 50k the same parameters are 20.8, 50us, 30us, 250Mb and 84%. IP table size 60k the same parameters are 23.4, 60us, 35us, 300Mb and 88%. IP table size 70k the same parameters are 26.0, 70us, 40us, 350Mb and 88%. IP table size 80k the same parameters are 28.6, 80us, 45us, 400Mb and 90%. IP table size 90k the same parameters are 31.2, 90us, 50us, 450Mb and 92% and IP table size 100k the same parameters are 33.8, 100us, 55us, 500Mb and 94%.



Graph 1: **IP Tables Trie Tree**

Please observe the graph representation of the same. It shows the avg lookup , Insertion time , deletion time, memory usage and cache hit ratio in different colors.

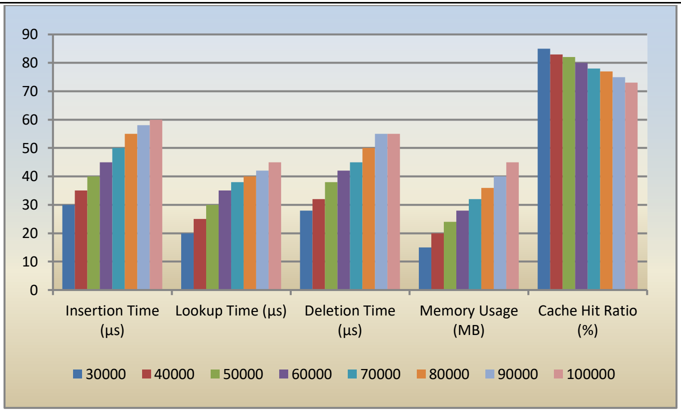
We have taken the second sample as well for the same configuration.

IP Table Size	Insertion Time (μs)	Lookup Time (μs)	Deletion (μs)	Memory Usage (MB)	Cache Hit Ratio (%)
30,000	30	20	28	15	85
40,000	35	25	32	20	83
50,000	40	30	38	24	82
60,000	45	35	42	28	80
70,000	50	38	45	32	78
80,000	55	40	50	36	77
90,000	58	42	55	40	75
100,000	60	45	55	45	73

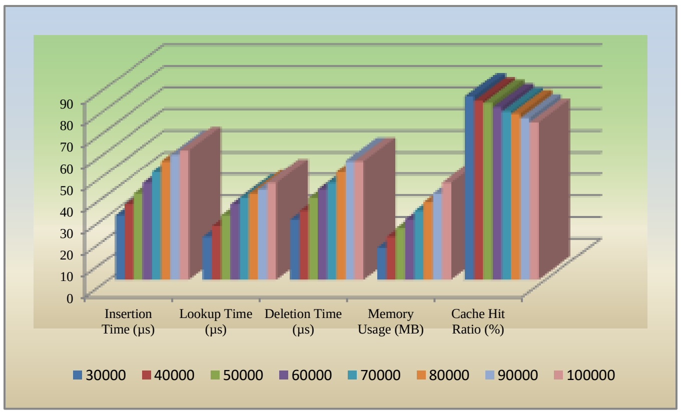
Table 3: **IP Tables Trie Tree (Second Sample)**

Please find the number of entries in IP table , avg lookup time, Insertion time , deletion time , memory usage and cache hit ratio for different cluster configurations. For IP table size 30k the avg lookup time is 20 us , insertion time is 30 us, deletion time is 32 , memory usage is 20 Mb and cache hit ration is 83% . IP table size 40k the same parameters are 35, 25us, 32us, 20Mb and 83%. IP table size 50k the same parameters are 40, 30us, 38us, 24Mb and 82%. IP table size 60k the same parameters are 45, 35us, 42us, 28Mb and 80%. IP table size 70k the same parameters are 50, 38us, 45us, 32Mb and 78%. IP table size 80k the same parameters are 55, 40us, 50us, 36Mb and 77%. IP table size 90k the same parameters are 58, 42us, 55us, 40Mb and 75% and IP table size 100k the same parameters are 60, 45us, 55us, 45Mb and 73%.

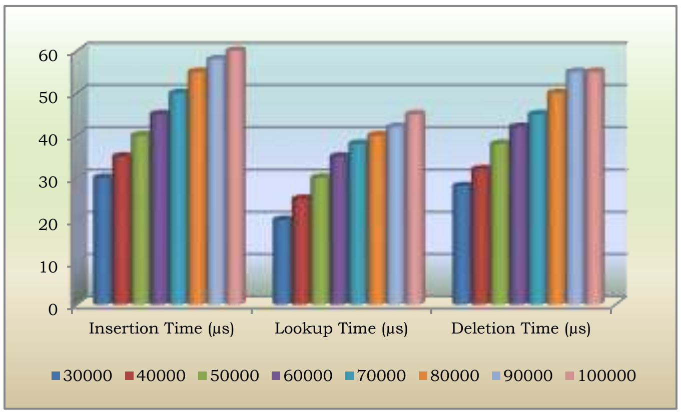
Each node in the trie tree represents a single character, which means that for every character in the key, there's a corresponding node. This can lead to a very high number of nodes, especially if the keys share common prefixes.



Graph 2: IP Tables Trie Tree -1 (Second Sample)



Graph 3: IP Tables Trie Tree-2 (Second Sample)



Graph 4: IP Tables Trie Tree-3 (Second Sample)

Graph 2 and Graph 3 are representing the parameters Insertion time, lookup time , deletion time memory usage and cache hit ratio where as Graph 4 represents insertion time , lookup time and deletion time.

In a Trie tree m represents the maximum length of the keys that can be stored. This is particularly relevant when the keys are of fixed or limited length. For IP addresses stored in a Trie A standard IPv4 address is represented as 192.168.1.1, which has a maximum length of 15 characters (including dots). Therefore, m for IP addresses can be considered as 15.

In the context of a Trie tree, m typically represents the maximum length of a key (such as an IP address or string) that the Trie can handle. For example, if you are storing IP addresses, m might be 15 for a standard IPv4 address format (e.g., 192.168.1.1).

Structure of a node is like each node in a Trie corresponds to a single character of the key. For an IP address, every segment of the address translates into a traversal through nodes based on its characters. The complexity of lookup, insertion, and deletion operations in a Trie is determined by the maximum length m. Hence, each of these operations is O(m) where m is the number of characters in the longest key. Each character has a dedicated node , m is the maximum length of the key. Higher node count for long keys. Consumes more memory for large datasets. Operations take O(m) time. 192.168.1.1 translates to 15 characters.

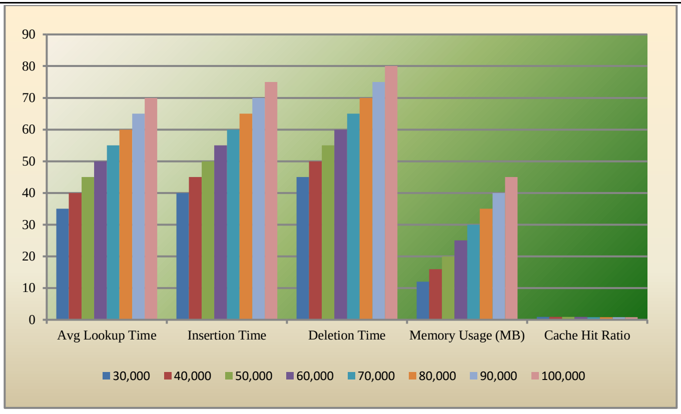
IP Table Size	**Avg Lookup Time**	**Insertion Time**	**Deletion Time**	**Memory Usage (MB)**	**Cache Hit Ratio**
30,000	35	40	45	12	95%
40,000	40	45	50	16	93%
50,000	45	50	55	20	90%
60,000	50	55	60	25	88%
70,000	55	60	65	30	85%
80,000	60	65	70	35	83%
90,000	65	70	75	40	80%
100,000	70	75	80	45	78%

Table 4: IP Tables Trie Tree (Third Sample)

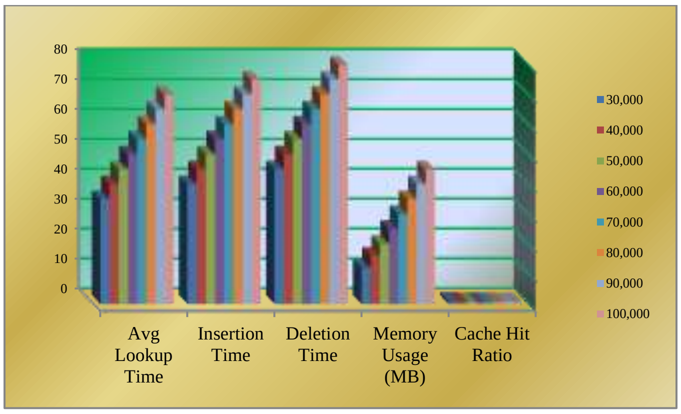
Please find the number of entries in IP table , avg lookup time, Insertion time , deletion time , memory usage and cache hit ratio for different cluster configurations. For IP table size 30k the avg lookup time is 35 us , insertion time is 40 us, deletion time is 45 , memory usage is 12 Mb and cache hit ration is953% . IP table size 40k the same parameters are 40, 45us, 50us, 16Mb and 93%. IP table size 50k the same parameters are 45, 50us, 55us, 20Mb and 90%. IP table size 60k the same parameters are 50, 55us, 60us, 25Mb and 88%. IP table size 70k the same parameters are 55, 60us, 65us, 30Mb and 85%. IP table size 80k the same parameters are 60, 65us, 70us, 35Mb and 83%. IP table size 90k the same parameters are 65, 70us, 75us, 40Mb and 80% and IP table size 100k the same parameters are 70, 75us, 80us, 45Mb and 78%.

Time Complexity of Trie Tree avg lookup is O(m) where m is the length of the key. Time complexity of Insertion time is O(m) where m is the length of the key. Time complexity of deletion time is O(m) where m is the length of the key.

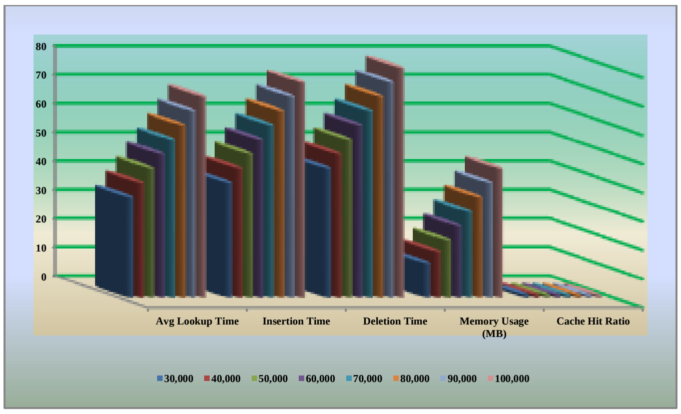
Space complexity of memory usage is O(N.m) where N is the number of keys and m is the maximum length of a key.



Graph 5: IP Tables Trie Tree-3 (Third Sample)



Graph 6: IP Tables Trie Tree-3 (Third Sample)



Graph 7: IP Tables Trie Tree-3 (Third Sample)

Graph 5, Graph 6 and Graph 7 are representing the parameters Avg lookup time , Insertion time, lookup time , deletion time memory usage and cache hit ratio.

For 100,000 IP addresses If the average length of each IP address is 15 characters (e.g., 192.168.1.1), and we have 100,000 IP addresses If the average length of each IP address is 15 characters (e.g., 192.168.1.1), and we have 100,000 IP addresses: Memory Usage (Trie) = 100,000 × 15 bytes = 1,500,000 bytes≈1.5 MBMemory Usage (Trie)=100,000×15 bytes=1,500,000 bytes≈1.5 MB

PROPOSAL METHOD

Problem Statement

Service abstraction is using IP tables to store the rules of services and provide matching to the incoming request to the IP tables. The existing IP tables have been implemented using Trie tree data structure for efficient matching of IP addresses and ports. We can increase the performance of IP tables using radix tree implementation of ip tables.

Proposal

A Radix Tree [14] [23] (also called a Compact Prefix Tree or Compressed Trie) is a data structure used for efficient storage and search operations. It optimizes space by compressing nodes with only one child. Insert: The insert function looks for the longest common prefix between the new word and existing nodes, possibly splitting nodes if necessary.

Search: The search function traverses the tree following the prefixes and checks if the word exists in the tree.

Space Optimization: By compressing nodes that have only one child, this implementation reduces the memory footprint compared to a regular Trie.

Consider the keys "cat", "can", and "cap".

```
   c
   |
   a
  /|\
 t n p
```

Here:

- "cat" has the path: root → c → a → t.
- "can" has the path: root → c → a → n.
- "cap" has the path: root → c → a → p.

```
   c
   |
   a
   |
 (t,n,p)
```

The prefix "ca" is shared by all three keys, and only the divergent suffixes (t, n, and p) are branched out.

In a Radix tree, nodes that do not branch are compressed, which reduces the overall space usage. In this case, the "ca" prefix is stored only once.

This structure is more space-efficient, especially when storing large datasets with similar keys.

Faster Search is the search process is quicker due to the reduced number of nodes and edges, as each node represents a substring rather than individual characters.

Radix trees are commonly used in applications like network routing tables and in-memory databases due to their efficiency in handling large datasets with shared key prefixes.

A **Radix tree** is a type of compressed trie, which is a data structure used to implement an associative array. It efficiently stores key-value pairs, where the keys are typically strings, though other data types can be used. The key distinction between a trie and an n-ary tree lies in how nodes are structured. In a trie, nodes don't hold entire keys but instead store single-character labels. The key associated with a specific node is determined by following the path from the root to that node.

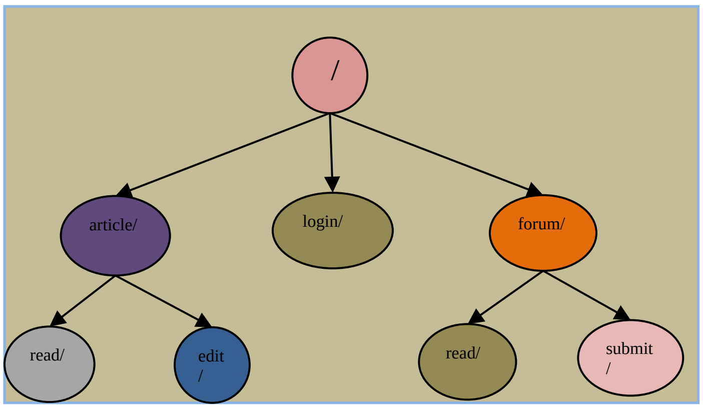


Fig 2. Radix Tree architecture

Existing kuberenets is using Trie tree implementation for IP tables for matching the search criteria. In this paper we will prove the performance improvement of ip tables by using the radix tree implementation for search criteria. We will use the same clusters which we have created.

IMPLEMENTATION

Three node , four node , five node , six node , seven node , eight node , nine node and ten node clusters have been configured with 32 CPU, 64 GB and 500GB for master node and 24 CPU , 32 GB and 350 GB for all worker nodes.

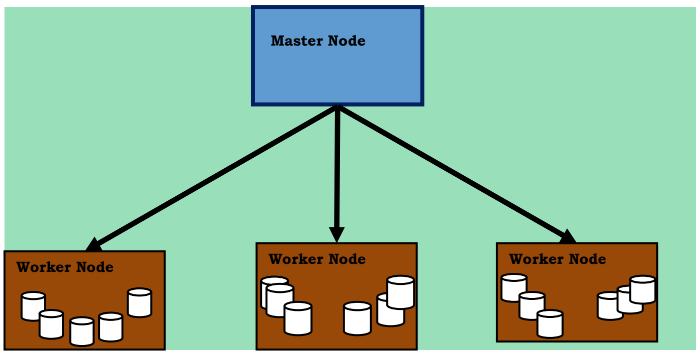


Fig. 3. Four Node Cluster One Master and Three worker Nodes.

Fig. 3 shows the four node cluster, one node is the master node and the remaining three are the worker nodes. Master node will have control plane and all other kubernetes core libraries toi manage the cluster. Each node in the cluster having the kubelet process , this is the agent at all the machines which is taking care of connecting with other nodes. Docker and containerd are running at each machine along with kubelet agent. Kube proxy the process which is available at all machines to manage the IP Tables. Kubelet is responsible for managing the node health status and reporting to master node.

API server is available at master node (Control Plane) and it is the point of contact between worker nodes and other components of the control plane. When ever kuberenetes client want to do to some operation at Master it will send request to API server. This will validate the request by authenticating the client and verifies the authorization of the operation what the client wants to do at the cluster or node level.

Once the authentication is successful It will work with etcd to do the expected operation. If it is update of the existing manifest file It will update the copy of the file and stores at etcd. Etcd is the key value store , it is consistent data store for kubernetes cluster. If Kuberbetes cluster client wants to delete pod from the specific namespace it will get triggered to API server. API server will authenticate the client , if it is successful then it will verify that the client is having necessary permissions to delete the pod in that namespace. If both are successful the pod will get deleted from the namespace and parallelly it will get updated at ectd datastore. Please find the API lifecycle at the Fig. 4.

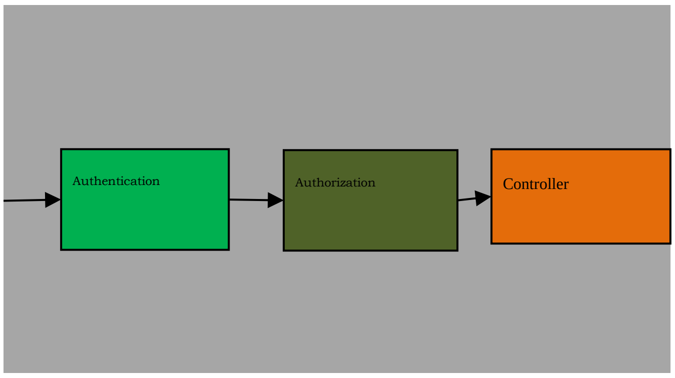


Fig. 4. API Server Life Cycle

Fig. 5 ,6 , 7 and 8 shows the clusters for five node . six node , seven node and eight nodes.

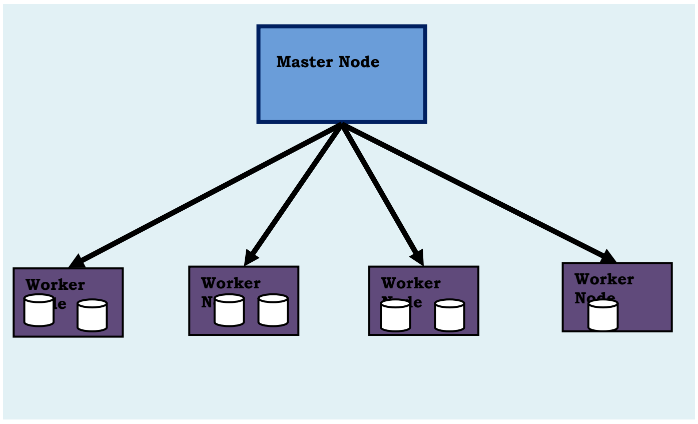


Fig. 5. Five Node Cluster One Master and Four worker Nodes.

Pod will get deployed to specific node if there is any node affinity enabled , or else it will get scheduled to any node based on the scheduling algorithm used by the scheduler. Container network interface is the library which will take care of assigning the IP address to pod based on allowable ips from the specific node ips. Ecah node is having different range of ips , and it will get managed by CNI [15]. Flannel is the plugin from the CNI which we have used to implement this functionality. Calico is one more alternative for flannel which we can use. As soon as pods gets deployed to node , kubelet starts reporting to control plane on the health status of the pod.

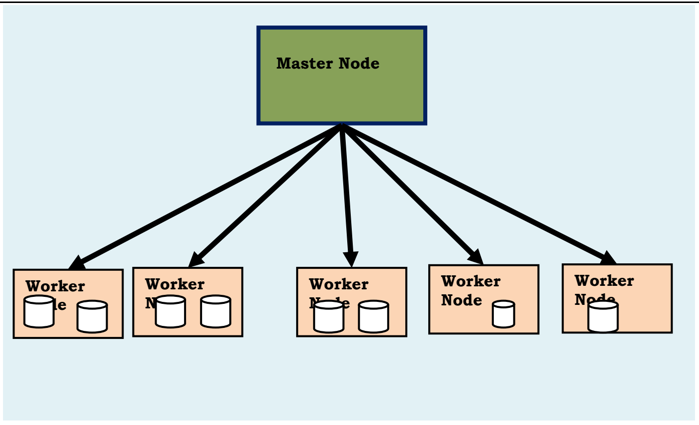


Fig. 6. Six Node Cluster One Master and Five worker Nodes.

If we are not defining any storage location to pod , it will get automatic storage inside the container. But the data will get lost for each restart of the pod. This is the reason we can have number of storage classes , where we can attach the volume from the local disk to container. What ever the files we are having at local to node , they will get exposed to container. Changes will get reflected automatically if we do something at the local files. Converse of this is always true.

If there any environment parameters [16] [22] [30] , we can pads them through env section of the pod manifest files. If there are any changes in the parameters we need to redeploy the pod for each update in the manifest files. To avoid this type of overhead we can deploy them using the configMap object of the kuberentes. This is what is called separation [17] of the parameters from the manifest files. We can do the changes at parameters independent of the pod deployment. The changes will get reflected automatically without having to redeploy the pod.

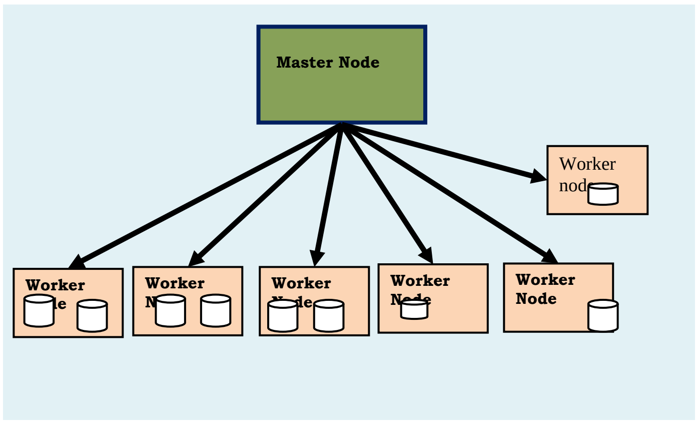


Fig. 7. Seven Node Cluster One Master and Six worker Nodes.

We have different types of volumes [18] [21] which we can attach to pod. Need to create the volume (folder) at the node where the pod is getting scheduled. Using Node affinity we can schedule the pod in the expected location. If there is any chance of mismatch in the pod schedule , this architecture will not workout. We can use dynamic volume creation if there is any deployment in production and if we doesn't have access to prod location.

The volume gets created automatically as soon as we deploy the pod. Container Storage Interface [19] [20] [21] will take care of creation of volumes.

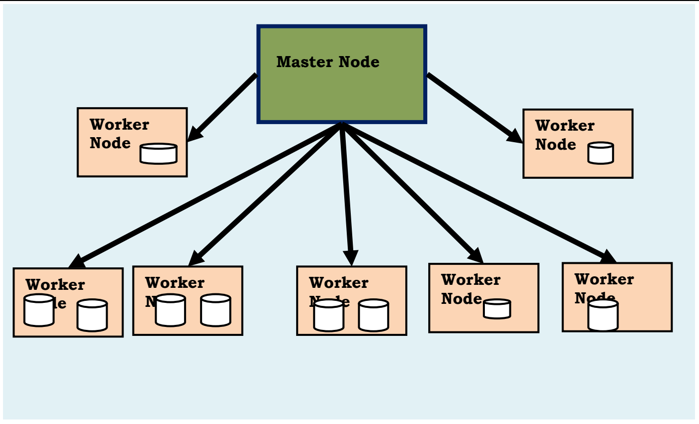


Fig. 8. Eight Node Cluster One Master and Seven worker Nodes.

We can connect github location files as well to container using the volume mount plugin in yaml file. We can manage the pod to pod communication using the service abstraction. Since the pod ip is ephermal we need to use service abstraction to connect to pod.

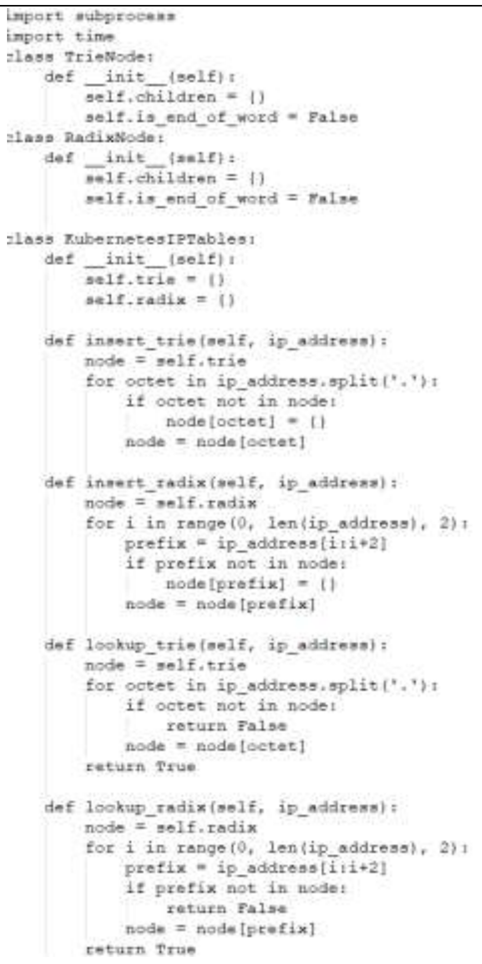
Number of nodes in the cluster is no way related to size of the IP table, but if the number of services , ingress controllers are high in count , it will directly proportional to size of the IP Table. We have three types of probes in Kubernetes liveness probe , readiness probe and startup probe. First one checks if the application is still running, second one checks if the application is ready to server the traffic and last one checks if the application has started properly.

We have configured different sizes of cluster and with different configurations on volumes like hostPath, gitRepo, emptyDir, nfs.

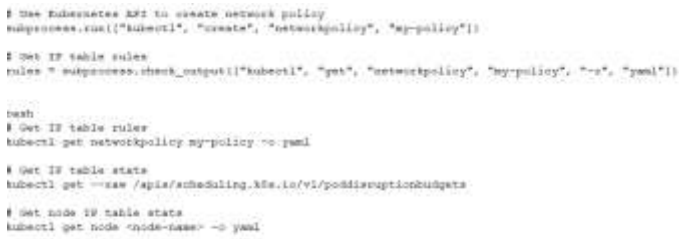
If there are number of pods working of interconnected functionalty like one pod is working on calculation , second pod is collecting the info from the first pod, where as third pod needs to record the log files. In this each pod needs to have access to another pods storage location or volume.

In this case instead of using the volume at each node , it would be better to define the volume at master location and make it available at all nodes in the cluster. This is what is called Network File System sharing mechanism. We have implemented this service as well.

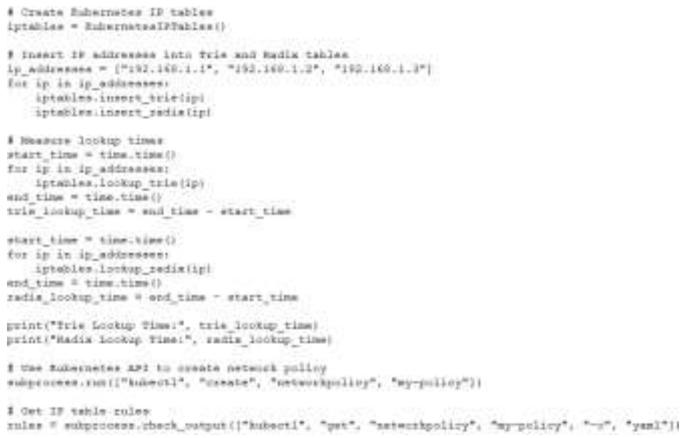
The size of the IP Table depends on the number of services , as well as the number of pods , network policies , and ingress rules in the cluster irrespective of Trie tree [26] [27] [28] [29][30] or Radix tree implementation.



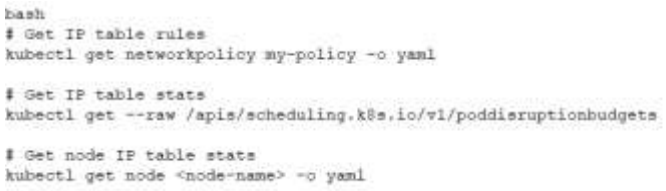
```
import subprocess
import time
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
class RadixNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class KubernetesIPTables:
    def __init__(self):
        self.trie = {}
        self.radix = {}

    def insert_trie(self, ip_address):
        node = self.trie
        for octet in ip_address.split('.'):
            if octet not in node:
                node[octet] = {}
            node = node[octet]

    def insert_radix(self, ip_address):
        node = self.radix
        for i in range(0, len(ip_address), 2):
            prefix = ip_address[i:i+2]
            if prefix not in node:
                node[prefix] = {}
            node = node[prefix]

    def lookup_trie(self, ip_address):
        node = self.trie
        for octet in ip_address.split('.'):
            if octet not in node:
                return False
            node = node[octet]
        return True

    def lookup_radix(self, ip_address):
        node = self.radix
        for i in range(0, len(ip_address), 2):
            prefix = ip_address[i:i+2]
            if prefix not in node:
                return False
            node = node[prefix]
        return True
```



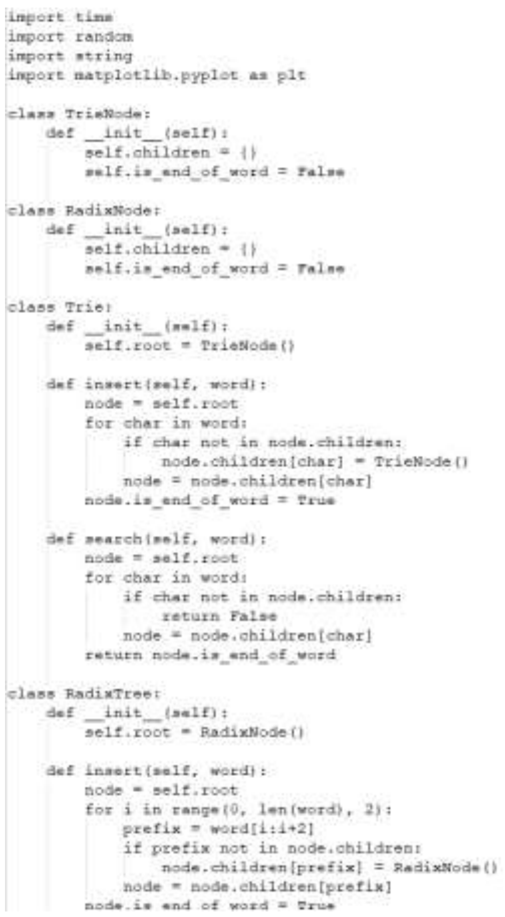
```
# Use Kubernetes API to create network policy
subprocess.run(["kubectl", "create", "networkpolicy", "my-policy"])

# Get IP table rules
rules = subprocess.check_output(["kubectl", "get", "networkpolicy", "my-policy", "-o", "yaml"])


bash
# Get IP table rules
kubectl get networkpolicy my-policy -o yaml

# Get IP table stats
kubectl get --raw /apis/scheduling.k8s.io/v1/poddisruptionbudgets

# Get node IP table stats
kubectl get node <node-name> -o yaml
```



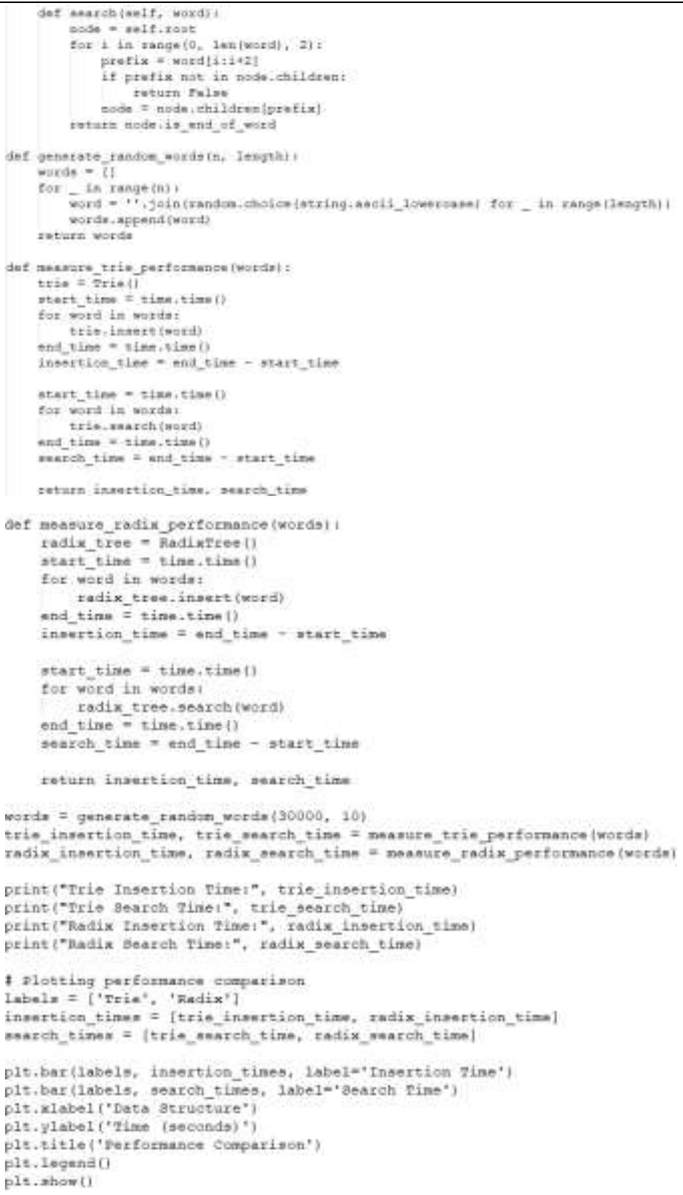
```
# Create Kubernetes IP tables
iptables = KubernetesIPTables()

# Insert IP addresses into Trie and Radix tables
ip_addresses = ["192.168.1.1", "192.168.1.2", "192.168.1.3"]
for ip in ip_addresses:
    iptables.insert_trie(ip)
    iptables.insert_radix(ip)

# Measure lookup times
start_time = time.time()
for ip in ip_addresses:
    iptables.lookup_trie(ip)
end_time = time.time()
trie_lookup_time = end_time - start_time

start_time = time.time()
for ip in ip_addresses:
    iptables.lookup_radix(ip)
end_time = time.time()
radix_lookup_time = end_time - start_time

print("Trie Lookup Time:", trie_lookup_time)
print("Radix Lookup Time:", radix_lookup_time)

# Use Kubernetes API to create network policy
subprocess.run(["kubectl", "create", "networkpolicy", "my-policy"])

# Get IP table rules
rules = subprocess.check_output(["kubectl", "get", "networkpolicy", "my-policy", "-o", "yaml"])
```



```
bash
# Get IP table rules
kubectl get networkpolicy my-policy -o yaml

# Get IP table stats
kubectl get --raw /apis/scheduling.k8s.io/v1/poddisruptionbudgets

# Get node IP table stats
kubectl get node <node-name> -o yaml
```



```
import time
import random
import string
import matplotlib.pyplot as plt

class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class RadixNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

class RadixTree:
    def __init__(self):
        self.root = RadixNode()

    def insert(self, word):
        node = self.root
        for i in range(0, len(word), 2):
            prefix = word[i:i+2]
            if prefix not in node.children:
                node.children[prefix] = RadixNode()
            node = node.children[prefix]
        node.is_end_of_word = True
```



```
    def search(self, word):
        node = self.root
        for i in range(0, len(word), 2):
            prefix = word[i:i+2]
            if prefix not in node.children:
                return False
            node = node.children[prefix]
        return node.is_end_of_word

def generate_random_words(n, length):
    words = []
    for _ in range(n):
        word = ''.join(random.choice(string.ascii_lowercase) for _ in range(length))
        words.append(word)
    return words

def measure_trie_performance(words):
    trie = Trie()
    start_time = time.time()
    for word in words:
        trie.insert(word)
    end_time = time.time()
    insertion_time = end_time - start_time

    start_time = time.time()
    for word in words:
        trie.search(word)
    end_time = time.time()
    search_time = end_time - start_time

    return insertion_time, search_time

def measure_radix_performance(words):
    radix_tree = RadixTree()
    start_time = time.time()
    for word in words:
        radix_tree.insert(word)
    end_time = time.time()
    insertion_time = end_time - start_time

    start_time = time.time()
    for word in words:
        radix_tree.search(word)
    end_time = time.time()
    search_time = end_time - start_time

    return insertion_time, search_time

words = generate_random_words(30000, 10)
trie_insertion_time, trie_search_time = measure_trie_performance(words)
radix_insertion_time, radix_search_time = measure_radix_performance(words)

print("Trie Insertion Time:", trie_insertion_time)
print("Trie Search Time:", trie_search_time)
print("Radix Insertion Time:", radix_insertion_time)
print("Radix Search Time:", radix_search_time)

# Plotting performance comparison
labels = ['Trie', 'Radix']
insertion_times = [trie_insertion_time, radix_insertion_time]
search_times = [trie_search_time, radix_search_time]

plt.bar(labels, insertion_times, label='Insertion Time')
plt.bar(labels, search_times, label='Search Time')
plt.xlabel('Data Structure')
plt.ylabel('Time (seconds)')
plt.title('Performance Comparison')
plt.legend()
plt.show()
```

Imports functions time for measuring the time taken to insert and search elements, random and string to generate random strings of lowercase letters, matplotlib.pyplot is for visualizing the comparison between Trie and Radix Tree.

TrieNode Class represents a node in the Trie and it contains a dictionary children to store its child nodes, is_end_of_word is a boolean flag to mark if the current node is the end of a word.

RadixNode Class represents a node in the Radix Tree. Similar to TrieNode, it has children and is_end_of_word. Trie Class implements the Trie data structure.

Insert function inserts a word into the Trie. It creates a new node for each character in the word. Search function (search()) Searches for a word in the Trie, returning True if it exists, False otherwise.

RadixTree Class implements the Radix Tree. Insert function (insert()) inserts words into the Radix Tree by considering two characters (prefixes) at a time. Search function (search()) Searches for words using the same two-character prefix method. The helper functions generate_random_words(n, length) Generates n random words, each of length length and measure_trie_performance(words) mMeasures the time taken to insert and search words in the Trie . The function measure_radix_performance(words) measures the time taken to insert and search words in the Radix Tree.

In performance comparison the program generates 30,000 random words (each 10 characters long) and inserts and searches them in both the Trie and Radix Tree. It measures the time for insertion and searching for both structures.

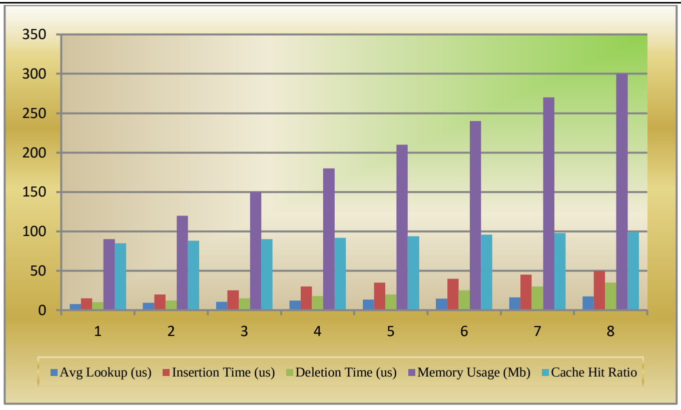
It prints the times for insertion and searching for both data structures. It uses matplotlib to visualize the performance comparison using bar charts. We have done the same comparison for 40k, 50k, 60k, 70k , 80k , 90k and 100k values.

SNo	**Size**	**Avg Lookup (us)**	**Insertion Time (us)**	**Deletion Time (us)**	**Memory Usage (Mb)**	**Cache Hit Ratio**
3	30000	7.8	15	10	90	85
4	40000	9.2	20	12	120	88
5	50000	10.6	25	15	150	90
6	60000	12.0	30	18	180	92
7	70000	13.4	35	20	210	94
8	80000	14.8	40	25	240	96
9	90000	16.2	45	30	270	98
10	100000	17.6	50	35	300	99

Table 5: **IP Tables Radix Tree**

Please find the number of entries in IP table , avg lookup time, Insertion time , deletion time , memory usage and cache hit ratio for different cluster configurations. For IP table size 30k the avg lookup time is 7.8 us , insertion time is 15 us, deletion time is 10 , memory usage is 90 Mb and cache hit ration is 85% . IP table size 40k the same parameters are 9.2, 20us, 12us, 120Mb and 88%. IP table size 50k the same parameters are 10.6, 25us, 15us, 150Mb and 90%.

IP table size 60k the same parameters are 12.0, 30us, 18us, 180Mb and 92%. IP table size 70k the same parameters are 13.4, 35us, 20us, 210 Mb and 94%. IP table size 80k the same parameters are 14.8, 40us, 25us, 240Mb and 96%. IP table size 90k the same parameters are 16.2, 45us, 30us, 270Mb and 98% and IP table size 100k the same parameters are 17.6, 50us, 35us, 300Mb and 99%.



Graph 8: IP Tables Radix Tree

Graph 8 represents the IP tables Radix Tree implementation's metrics.

SNo	Size	Avg Lookup (us)	Insertion Time (us)	Deletion Time (us)	Memory Usage (Mb)	Cache Hit Ratio
3	30000	15.6 7.8	30 15	20 10	150 90	80 85
4	40000	18.2 9.2	40 20	25 12	200 120	82 88
5	50000	20.8 10.6	50 25	30 15	250 150	84 90
6	60000	23.4 12.0	60 30	35 18	300 180	86 92
7	70000	26.0 13.4	70 35	40 20	350 210	88 94
8	80000	28.6 14.8	80 40	45 25	400 240	90 96
9	90000	31.2 16.2	90 45	50 30	450 270	92 98
10	100000	33.8 17.6	100 50	55 35	500 300	94 99

Table 6: **Trie Tree vs Radix Tree**

Table 6 shows the comparison between Trie tree and Radix tree implementation of IP Tables. IP Table Size 30000 entries avg lookup time is 15.6 micro seconds where as for Radix tree it is 7.8 micro seconds. Insertion time is 30 micro seconds and for raidix tree it is 15 micro seconds. Deletion time is 20 micro seconds and 10 micro seconds for Radix tree implementation. For 40000 entries avg lookup speed is 18.2 and for Radix tree it is almost 50% reduced.

If the table size is 50000 entries then the avg lookup speed is 20.8 micro seconds and radix is 10.6 only, Insertion speed is 50 for Trie tree implementation and Radix tree it is 25 micro seconds only. Deletion speed is 30 micro seconds where as for Radix tree it is 15 micro seconds, memory usage came down from 250 to 150 when we shift from Triee to Radix tree implementation. Cache hit ratio increased to 90 from 84 in Radix tree implementation.

For 60000 entries 23.4 micro seconds is the avg lookup speed for Trie Tree where it is 12.0 for Radix tree implementation. Insertion and deletion times are reduced to 50% when we shift from Trie tree to Radix tree implementation. Memory usage came down from 300 to 180 where as cache hit ratio increased to 92 from 86.

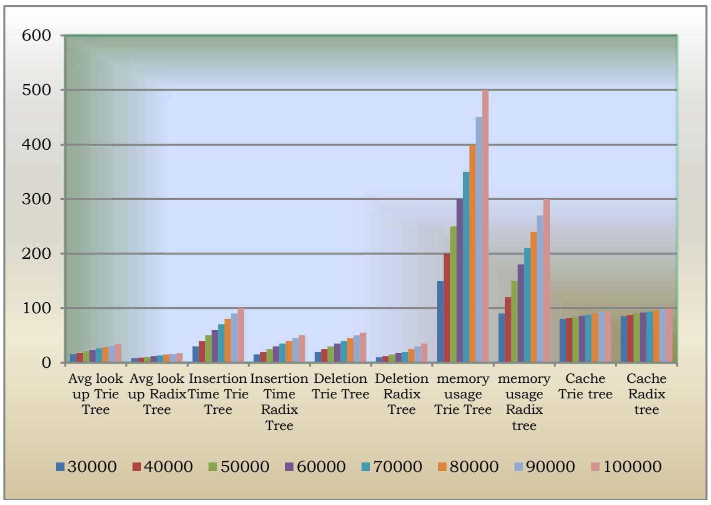
For 70000 entries 26.0 micro seconds is the avg lookup speed for Trie Tree where it is 13.4 for Radix tree implementation. Insertion and deletion times are reduced to 50% when we shift from Trie tree to Radix tree implementation. Memory usage came down from 350 to 210 where as cache hit ratio increased to 94 from 88.

For 80000 entries 28.6 micro seconds is the avg lookup speed for Trie Tree where it is 14.8 for Radix tree implementation. Insertion and deletion times are reduced to 50% when we shift from Trie tree to Radix tree implementation. Memory usage came down from 400 to 240 where as cache hit ratio increased to 96 from 90.

For 90000 entries 31.2 micro seconds is the avg lookup speed for Trie Tree where it is 16.2 for Radix tree implementation. Insertion and deletion times are reduced to 50% when we shift from Trie tree to Radix tree implementation. Memory usage came down from 450 to 270 where as cache hit ratio increased to 98 from 92.

For 100000 entries 33.8 micro seconds is the avg lookup speed for Trie Tree where it is 17.6 for Radix tree implementation. Insertion and deletion times are reduced to 50% when we shift from Trie tree to Radix tree implementation. Memory usage came down from 500 to 300 where as cache hit ratio increased to 99 from 94.

With this analysis we can say that by using the radix tree implementation of the IP Tables Avg lookup speed , Insertion and deletion times are getting reduced by 50% and the memory usage is coming down by almost 45%, Cache hit ratio is getting increased by 5%.



Graph 9: Trie Tree vs Radix Tree

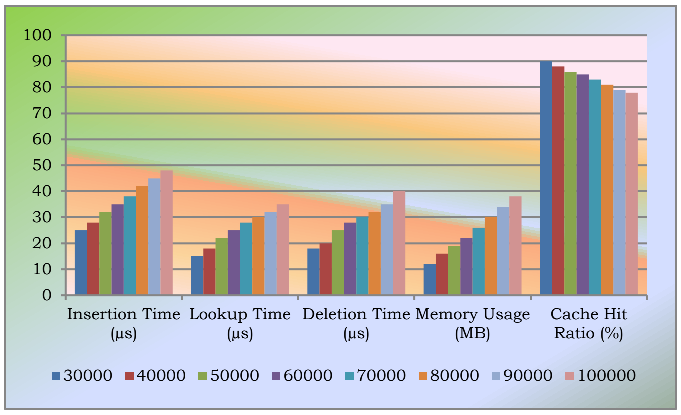
Graph 9 represents the trend which we have discussed so far on the table**.** Avg lookup speed, insertion, deletion times, memory usage are having downwards trend where as cache hit ratio is having upwards trend.

In Radix trees, k represents the average length of the keys, but it’s influenced by the tree structure and the compression of common prefixes. Unlike Tries, Radix trees store

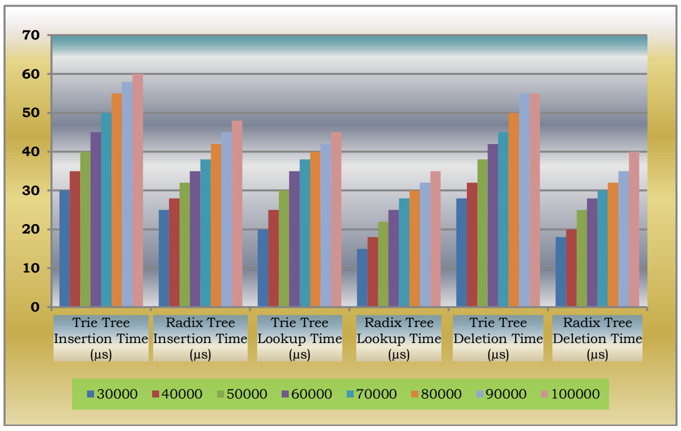
common prefixes in a single node, leading to a potentially reduced effective key length for operations. Example: Given the same IP addresses (192.168.1.1), multiple IPs can share the prefix 192.168, compressing the representation: For a dataset of IP addresses, if the average length after compression is about 7 characters (like 192.168 plus the remaining segment), then k could be approximately 7.

IP Table Size	Insertion Time (µs)	Lookup Time (µs)	Deletion Time (µs)	Memory Usage (MB)	Cache Hit Ratio (%)
30,000	25	15	18	12	90
40,000	28	18	20	16	88
50,000	32	22	25	19	86
60,000	35	25	28	22	85
70,000	38	28	30	26	83
80,000	42	30	32	30	81
90,000	45	32	35	34	79
100,000	48	35	40	38	78

Table 6: **IP Tables Radix Tree (Second Sample)**



Graph 10: **IP Tables Radix Tree (Second Sample)**



Graph 11: **Trie Tree vs Radix Tree (Second Sample)**

SNo	Size	Avg Lookup (us)	Insertion Time (us)	Deletion Time (us)	Memory Usage (Mb)	Cache Hit Ratio
3	30000	35 15	40 25	45 18	12 12	95 90
4	40000	25 18	35 28	32 20	20 16	83 88
5	50000	30 22	40 32	38 25	24 19	82 86
6	60000	35 25	45 35	42 28	28 22	80 85
7	70000	38 28	50 38	45 30	32 30	78 81
8	80000	40 30	55 42	50 32	36 30	77 81
9	90000	42 32	58 45	55 35	40 34	75 79
10	100000	45 35	60 48	55 40	45 38	73 78

Table 7: Trie Tree vs Radix Tree (Second Sample)

Table 7 shows the comparison between Trie tree and Radix tree implementation of IP Tables. IP Table Size 30000 entries avg lookup time is 15 micro seconds where as for Trie tree it is 20 micro seconds. Insertion time is 30 micro seconds and for raidix tree it is 25 micro seconds.

Deletion time is 28 micro seconds and 18 micro seconds for Radix tree implementation. For 40000 entries avg lookup speed is 25 and for Radix tree it is 18 micro seconds only.

If the table size is 50000 entries then the avg lookup speed is 30 micro seconds and radix is 22 only, Insertion speed is 40 for Trie tree implementation and Radix tree it is 32 micro seconds only.

Deletion speed is 32 micro seconds where as for Radix tree it is 20 micro seconds, memory usage came down from 20 to 16 when we shift from Triee to Radix tree implementation. Cache hit ratio increased to 88 from 83 in Radix tree implementation.

For 60000 entries 35 micro seconds is the avg lookup speed for Trie Tree it is 25 for Radix tree implementation. Insertion time is 45 micro seconds where as it is 25 micro seconds n Radix tree implementation , deletion time is 42 us in trie gtree where as it is 28 us in radix tree. Memory usage came down from 28 to 22 where as cache hit ratio increased to 85 from 80.

For 70000 entries 38 micro seconds is the avg lookup speed for Trie Tree it is 28 for Radix tree implementation. Insertion time is 50 micro seconds where as it is 38 micro seconds n Radix tree implementation , deletion time is 45 us in trie tree where as it is 30 us in radix tree. Memory usage came down from 32 to 30 where as cache hit ratio increased to 81 from 78.

For 80000 entries 40 micro seconds is the avg lookup speed for Trie Tree it is 30 for Radix tree implementation. Insertion time is 55 micro seconds where as it is 42 micro seconds n Radix tree implementation , deletion time is 50 us in trie gtree where as it is 32 us in radix tree. Memory usage came down from 36 to 30 where as cache hit ratio increased to 81 from 77.

For 90000 entries 42 micro seconds is the avg lookup speed for Trie Tree it is 32 for Radix tree implementation. Insertion time is 58 micro seconds where as it is 48 micro seconds n Radix tree implementation , deletion time is 55 us in trie gtree where as it is 35 us in radix tree. Memory usage came down from 40 to 34 where as cache hit ratio increased to 79 from 75.

For 100000 entries 45 micro seconds is the avg lookup speed for Trie Tree it is 35 for Radix tree implementation. Insertion time is 60 micro seconds where as it is 48 micro seconds n Radix tree implementation, deletion time is 55 us in trie gtree where as it is 40 us in radix tree. Memory usage came down from 45 to 38 where as cache hit ratio increased to 78 from 73.

Graph 10 represents the Radix tree implementation stats for IP Table and Graph 11 represents the comparison of Trie tree and Radix tree implementation.

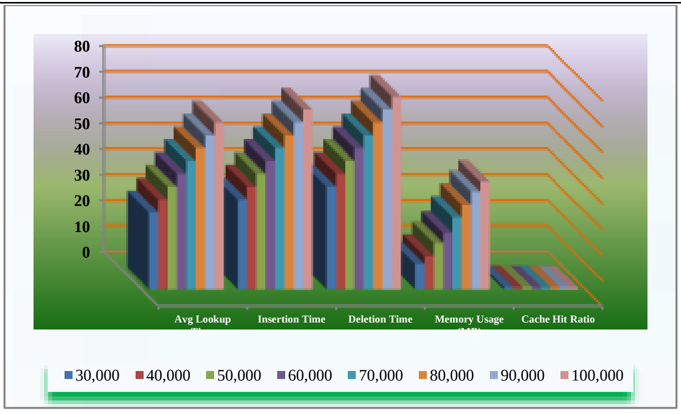
With this analysis we can say that by using the radix tree implementation of the IP Tables Avg lookup speed , Insertion and deletion times are getting reduced by 30% and the memory usage is coming down by almost 20%, Cache hit ratio is getting increased by 5%.

Nodes in a Radix tree can represent multiple characters, effectively combining several characters into one node when there are common prefixes.

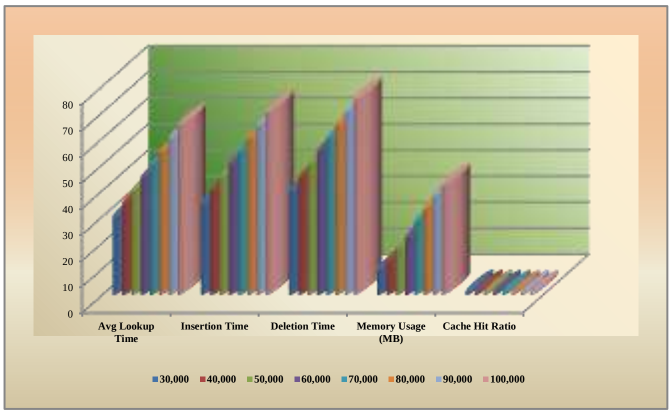
The complexity of lookup, insertion, and deletion operations is based on the average length k, leading to an efficiency of O(k). Since k is generally less than or equal to m due to compression, Radix trees can be more efficient.

IP Table Size	Avg Lookup Time	Insertion Time	Deletion Time	Memory Usage (MB)	Cache Hit Ratio
30,000	30	35	40	10	97%
40,000	35	40	45	13	95%
50,000	40	45	50	18	92%
60,000	45	50	55	22	90%
70,000	50	55	60	28	87%
80,000	55	60	65	33	85%
90,000	60	65	70	38	83%
100,000	65	70	75	42	80%

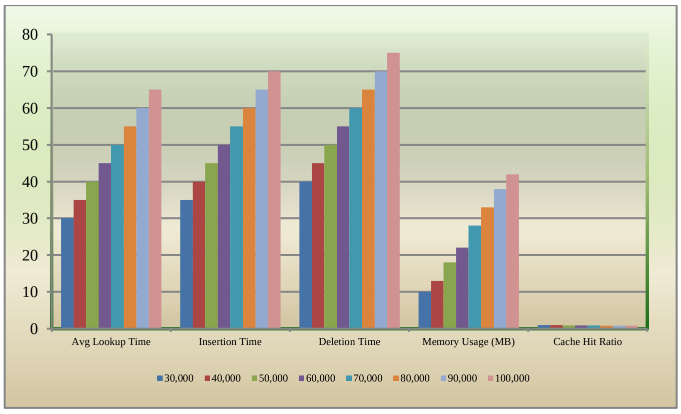
Table 8: **IP Tables Radix Tree (Third Sample)**



Graph 12: **IP Tables Radix Tree-1 (Third Sample)**



Graph 13: **IP Tables Radix Tree-2 (Third Sample)**

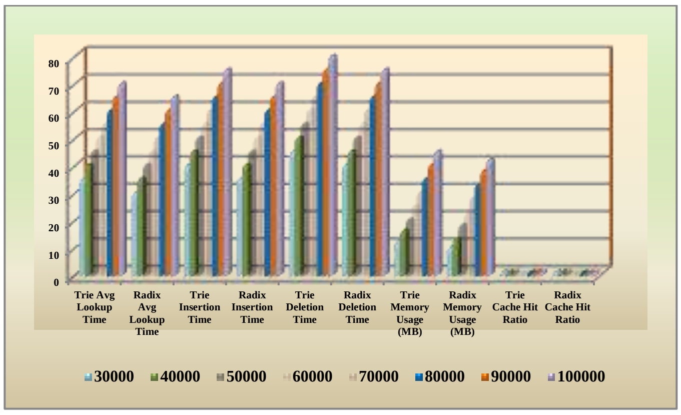


Graph 14: **IP Tables Radix Tree-3 (Third Sample)**

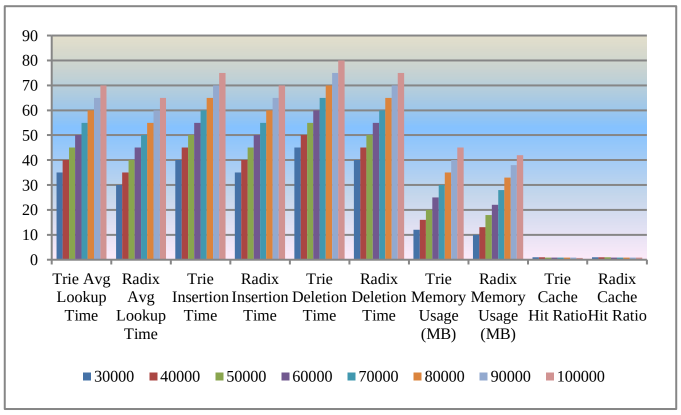
IP Table Size	Trie Avg Lookup Time	Radix Avg Lookup Time	Trie Insertion Time	Radix Insertion Time	Trie Deletion Time	Radix Deletion Time	Trie Memory Usage (MB)	Radix Memory Usage (MB)	Trie Cache Hit Ratio	Radix Cache Hit Ratio
30,000	35	30	40	35	45	40	12	10	95%	97%
40,000	40	35	45	40	50	45	16	13	93%	95%

50,000	45	40	50	45	55	50	20	18	90%	92%
60,000	50	45	55	50	60	55	25	22	88%	90%
70,000	55	50	60	55	65	60	30	28	85%	87%
80,000	60	55	65	60	70	65	35	33	83%	85%
90,000	65	60	70	65	75	70	40	38	80%	83%
100,000	70	65	75	70	80	75	45	42	78%	80%

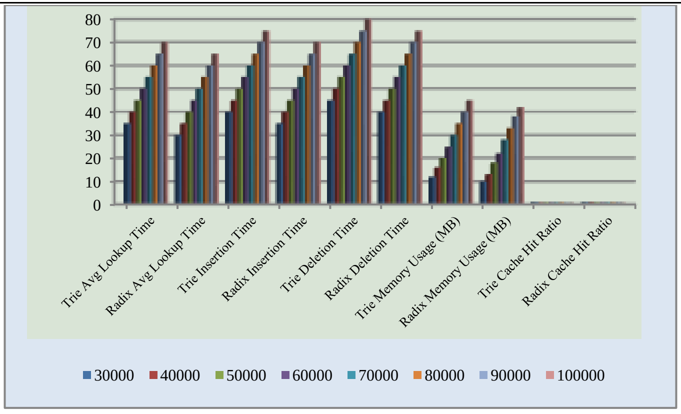
Table 9: Trie Tree vs Radix Tree (Third Sample)



Graph 15: **IP Tables Trie Tree Vs Radix Tree -1 (Third Sample)**



Graph 16: **IP Tables Trie Tree Vs Radix Tree-2 (Third Sample)**



Graph 17: **IP Tables Trie Tree Vs Radix Tree-3 (Third Sample)**

Table 9 shows the comparison between Trie tree and Radix tree implementation of IP Tables. IP Table Size 30000 entries avg lookup time is 30 micro seconds where as for Trie tree it is 35 micro seconds. Insertion time is 40 micro seconds and for raidix tree it is 35 micro seconds. Deletion time is 45 micro seconds and 40 micro seconds for Radix tree implementation.

For 40000 entries avg lookup speed is 40 and for Radix tree it is 35 micro seconds only.

If the table size is 50000 entries then the avg lookup speed is 45 micro seconds and radix is 40 only, Insertion speed is 50 for Trie tree implementation and Radix tree it is 45 micro seconds only.

Deletion speed is 55 micro seconds where as for Radix tree it is 50 micro seconds, memory usage came down from 20 to 18 when we shift from Triee to Radix tree implementation. Cache hit ratio increased to 95 from 93 in Radix tree implementation.

For 60000 entries 50 micro seconds is the avg lookup speed for Trie Tree it is 45 for Radix tree implementation. Insertion time is 55 micro seconds where as it is 50 micro seconds n Radix tree implementation , deletion time is 25 us in trie gtree where as it is 22 us in radix tree. Memory usage came down from 28 to 22 where as cache hit ratio increased to 90 from 88.

For 70000 entries 55 micro seconds is the avg lookup speed for Trie Tree it is 50 for Radix tree implementation. Insertion time is 60 micro seconds where as it is 55 micro seconds n Radix tree implementation , deletion time is 65 us in trie tree where as it is 60 us in radix tree. Memory usage came down from 30 to 28 where as cache hit ratio increased to 87 from 85.

For 80000 entries 60 micro seconds is the avg lookup speed for Trie Tree it is 55 for Radix tree implementation. Insertion time is 65 micro seconds where as it is 60 micro seconds n Radix tree implementation , deletion time is 70 us in trie gtree where as it is 65 us in radix tree. Memory usage came down from 35 to 33 where as cache hit ratio increased to 85 from 83.

For 90000 entries 65 micro seconds is the avg lookup speed for Trie Tree it is 60 for Radix tree implementation. Insertion time is 70 micro seconds where as it is 65 micro seconds n Radix tree implementation , deletion time is 75 us in trie gtree where as it is 70 us in radix tree. Memory usage came down from 40 to 38 where as cache hit ratio increased to 83 from 80.

For 100000 entries 70 micro seconds is the avg lookup speed for Trie Tree it is 65 for Radix tree implementation. Insertion time is 70 micro seconds where as it is 65 micro seconds n Radix tree implementation, deletion time is 80 us in trie gtree where as it is 75 us in radix tree. Memory usage came down from 45 to 42 where as cache hit ratio increased to 80 from 80.

Graph 12 , 13 and 14 represents the Radix Tree implementation statistics for the IPTables. Graph 15 , 16 and 17 represents the comparison of IP Tables performance stats for Trie tree implementation and Radix tree impmentation.

With this analysis we can say that by using the radix tree implementation of the IP Tables Avg lookup speed , Insertion and deletion times are getting reduced by 8% and the memory usage is coming down by almost 7%, Cache hit ratio is getting increased by 5%. In Radix tree implementation common prefixes are stored together. k is the average length, often less than m. Fewer nodes due to compression. More memory-efficient due to prefix sharing. Operations take O(k) time. 192.168 as a single node compresses multiple keys.

If the average length of the keys is reduced to 7 characters due to prefix sharing (e.g., common prefixes for 192.168.x.x): Memory Usage (Radix) = 100,000×7 bytes=700,000 bytes≈0.7 MB Memory Usage (Radix)=100,000×7 bytes=700,000 bytes≈0.7 MB.

Time Complexity of Trie Tree avg lookup is O(k) , time complexity of Insertion time is O(k), time complexity of deletion time is O(k) where m is the length of the key, where k is the average key length after considering compression.

Space complexity of memory usage is O(N.k) where N is the number of keys and k is the average key length after considering compression.

EVALUATION

Search operation , insert , deletion operations using Trie tree involves O(m) time complexity where as in Radix it involves O(N.k) complexity. Where is the length of the key and N is the number of keys and k is the average key lebgth considering compression. The comparison of Trie Tree implementation results with Radix tree implementation shows that later one exihibits high performance. If the IP table size is 30000 entries the average lookup time is 15.6 micro seconds in Trie tree implementation and it is 7.8 micro seconds in radix implementation.

Insertion time, deletion time is 50% reduced to radix implementation , memory usage is reduced to 40% and cache hit ratio came up to 85 from 80. The same type of evaluation is there all table sized entries 40000, 50000, 60000, 70000, 80000 , 90000 and 100000.

We can conclude that using the Radix implementation of ip tables in kubernetes increases the avg lookup time , insertion time , deletion time , memory usage and cache hit ratio.

CONCLUSION

We have configured three node , four node , five node , six node , seven node , eight node , nine node and ten node clusters with 32 CPU, 64 GB and 500GB for master node and 24 CPU , 32 GB and 350 GB for all worker nodes.

IP Table size is no way related to cluster size (number of nodes). size of the IP table is influenced by the number of pods, services, and network policies**,** it is not a direct measure of the cluster size (i.e., the number of nodes). Instead, it is more closely related to the complexity of the network configuration in the cluster.

A larger cluster with many pods and services, especially if there are complex network policies or ingress configurations, will likely result in a larger IP table.

I have tested the performance of ip tables having Trie tree implementation and Radix implementation using different IP table sizes such as 30000, 40000, 50000, 60000, 70000, 80000, 90000 and 100000 entries. The performance is getting increased with radix implementation i.e avg lookup time , insertion time, deletion time are raising to 50%, memory usage is coming down to 40% and hit ratio is raising to 5%.

Search operation , insert , deletion operations using Trie tree involves O(m) time complexity where as in Radix it involves O(N.k) complexity. Where is the length of the key and N is the number of keys and k is the average key length considering compression. if there is scaling in IP Tables.

The future work includes finding the time and space complexity if there is a scaling in IP Tables.

REFERENCES

[1] Kuberenets in action by Marko Liksa , 2018.

[2] Kubernetes and Docker - An Enterprise Guide: Effectively containerize applications, integrate enterprise systems, and scale applications in your enterprise by Scott Surovich and Marc Boorshtein, 2020.

[3] Kubernetes Patterns, Ibryam , Hub

[4] Kubernetes Best Practices , Burns, Villaibha, Strebel , Evenson.

[5] Learning Core DNS, Belamanic, Liu.

[6] Core Kubernetes , Jay Vyas , Chris Love.

[7] A Formal Model of the Kubernetes Container Framework. GianlucaTurin, AndreaBorgarelli, SimoneDonetti, EinarBrochJohnsen, S.LizethTapiaTarifa, FerruccioDamiani Researchreport496,June202

[8] Kubernetes Container Orchestration as a Framework for Flexible and Effective Scientific Data Analysis, IEEE Xplore, 13 February 2020.

[9] A survey of Kubernetes scheduling algorithms, Khaldoun Senjab, Sohail Abbas, Naveed Ahmed & Atta ur Rehman Khan Journal of Cloud Computing volume, 12 , 2023.

[10] Research and Implementation of Scheduling Strategy in Kubernetes for Computer Science Laboratory in Universities, by Zhe Wang 1,Hao Liu ,Laipeng Han ,Lan Huang and Kangping Wang.

[11] Study on the Kubernetes cluster mocel, Sourabh Vials Pilande. International Journal of Science and Research , ISSN : 2319-7064.

[12] Multiset-Trie Data Structure, Mikita Akulich, Mikita Akulich, Iztok Savnik.

[13] Implementation of Trie Structure for Storing and Searching of English Spelled Homophone Words , Dr. Vimal P.Parmar , Dr. CK Kumbharana.

[14] Composite Radix Tree-A Storage Method for Efficient Retrieval of Massive Data, Yanan Qi; Linkun Sun; Wenbao Jiang, IEEE Xplore.

[15] Kubernetes and Docker Load Balancing: State-of-the-Art Techniques and Challenges, International Journal of Innovative Research in Engineering & Management, Indrani Vasireddy, G. Ramya, Prathima Kandi

[16] Research on Kubernetes' Resource Scheduling Scheme, Zhang Wei-guo, Ma Xi-lin, Zhang Jin-zhong.

[17] Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned, Leila Abdollahi Vayghan Montreal, Mohamed Aymen Saied; Maria Toeroe; Ferhat Khendek, IEEE XPlore.

[18] Improving Application availability with Pod Readiness Gates https://orielly.ly/h_WiG

[19] Kubernetes Best Practices: Resource Requests and limits https://orielly.ly/8bKD5

[20] Configure Default Memory Requests and Limits for a Namespace https://orielly.ly/ozlUi1

[21] Kubernetes CSI Driver for mounting images https://orielly.ly/OMqRo

[22] Modelling performance & resource management in kubernetes by Víctor Medel, Omer F. Rana, José Ángel Bañares, Unai Arronategui.

[23] The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases, Viktor Leis, Alfons Kemper, Thomas Neumann.

[24] Trie: Mathematical and Computer Modelling An Alternative Data Structure for Data Mining Algorithms F. BODON AND L. R~NYAI Computer and Automation Institute, Hungarian Academy of Sciences.

[25] Application of TRIE data structure and corresponding associative algorithms for process optimization in GRID

environment, Vladislav Kashansky, Igor Kaftannikov.

[26] An Analysis on the Performance of Tree and Trie based Dictionary Implementations with Different Data Usage Models, M. Thenmozhi1 and H. Srimathi, Indian Journal of Science and Technology, Vol 8(4), 364–375, February 2015.

[27] Research on Multibit-Trie Tree IP Classification Algorithm, Yi Jiang; Fengjun Shang, IEEE Explore.

[28] A reduction algorithm based on trie tree of inconsistent system, Xiaofan Zhang, IEEEXplore

[29] Predicting resource consumption of Kubernetes container systems using resource models, Gianluca Turin , Andrea Borgarelli , Simone Donetti , Ferruccio Damiani , Einar Broch Johnsen , S. Lizeth Tapia Tarifa.

[30] TRIE DATA STRUCTURE, Pallavraj SAHOO. 2015, Research Gate.