Kubernetes Etcd Implemantation Using Btree and Fractal Trees

Renukadevi Chuppala[1], Dr. B. PurnachandraRao[2]

Western Union Financial Services, CA, USA
Sr. Solutions Architect, HCL Technologies, Bangalore, Karnataka, India.

Abstract
Etcd is a distributed key-value store that provides a reliable way to store and manage data in a distributed system. Here's an overview of etcd and its role in Kubernetes. Etcd ensures data consistency and durability across multiple nodes, provides distributed locking mechanisms to prevent concurrent modifications, facilitates leader election for distributed systems. Etcd uses a distributed consensus algorithm (Raft) to manage data replication and ensure consistency across nodes. Etcd nodes form a cluster, ensuring data availability and reliability. stores data as key-value pairs., provides watchers for real-time updates on key changes, supports leases for distributed locking and resource management, Etcd serves as the primary data store for Kubernetes, responsible for storing and managing Cluster state i.e, Node information, pod status, and replication controller data, Configuration data like Persistent volume claims, secrets, and config maps, Network policies i.e, Network policies and rules, High availability that ensures data consistency and availability across nodes, Distributed locking i.e, Prevents concurrent modifications and ensures data integrity. Scalability Supports large-scale Kubernetes clusters. When ever we are sending apply command using kubectl or any other client API Server authenticates the request, authorizes the same, and updates to etcd on the new configuration. Etcd receives the updates (API Server sends the updated configuration to etcd), then etcd writes the updated configuration to its key-value store. Etcd replicates the updated data across its nodes and it ensures data consistency across all the nodes. We can say that ETCD is the main storage of the cluster. It carries the cluster state by storing the latest state at key value store. In this paper we will discuss about implementation of ETCD using BTree and Factal Tree. Factal tree outperforms BTree in some scenarios. We will work on to prove that Factal Tree implementation provides better performance than BTree.

Keywords: Kubernetes (K8S), Cluster, Nodes, Deployments, Pods, ReplicaSets, Statefulsets, Service, IP-Tables, Load Balancer, Service Abstraction, , BTree, Factal Tree, ETCD.

INTRODUCTION

Kubernetes [1] 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 [2] 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 [3] is an open-source, distributed key-value store that provides a reliable way to store and manage data in a distributed system. It is designed to be highly available, fault-tolerant, and scalable. Features are Distributed architecture, Key-value store, Leader election, Distributed locking, Watchers for real-time updates, Leases for resource management , Authentication and authorization, Support for multiple storage backends (e.g., BoltDB, RocksDB) [4]. And the APIs are put to Store a key-value pair, get to retrieve a value by key, delete to remove a key-value pair,

watch to watch for changes to a key , and lease to acquire a lease for resource management. Kube-proxy [5]: Manages network communication within and outside the cluster. Pod: The smallest deployable unit in Kubernetes, encapsulating one or more containers 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. Designed to manage stateful applications, where each Pod has a unique identity and persistent storage, such as databases. DaemonSet [6] 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 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.

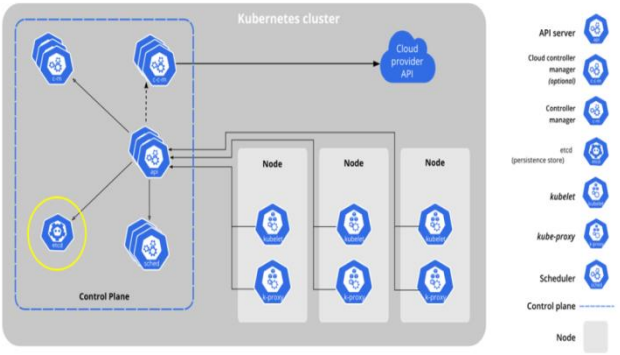


Fig: 1

Fig 1. Shows the Kubernetes cluster architecture. This shows three worker nodes and one control plane. Control plane is having four components API Server , Scheduler , Controller and ECTD. Pods are deployed to nodes using scheduler. Client kubectl will connect to API server (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 and authorization (RBAC plugin) it will connect with corresponding resources to proceed with further operations. Etcd is the storage location for all the kubernetes resources. Scheduler will select the appropriate node for scheduling [7] 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: A distributed key-value [8] store 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 [9]: Assigns workloads to worker nodes based on resource availability, scheduling policies, and requirements.
Worker Nodes:
Kubelet: The agent running on each node that ensures containers are running in Pods as specified by the control plane.
Container Runtime [10]: The software responsible for running containers (e.g., Docker, containerd).
Kube-proxy: Manages network [11] 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:
Scaling [12][22] 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.
Load Balancing: 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 [13][32] 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 [14][23][24] 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.
LoadBalancer: 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 [15][39][40]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.

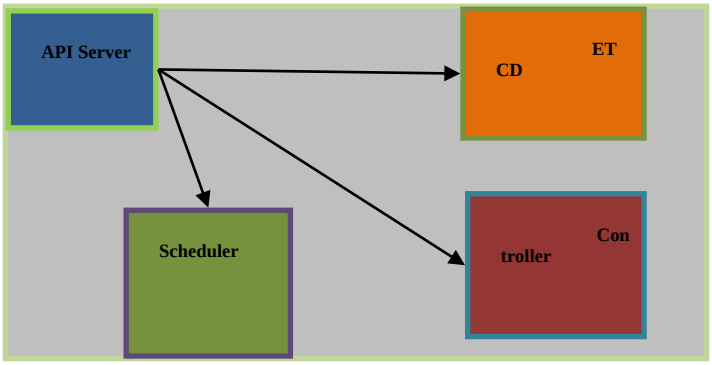


Fig 2: ETCD Architecture

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 [16][35][36]: Kubernetes 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: Iptables distributes incoming traffic among the Pods that match the service's selector, ensuring load balancing.
Return Traffic [17][37][38] 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 [18][33][34] 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, also known as a Prefix Tree, 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.

```
package etcd
import (
    "encoding/json"
    "errors"
    "fmt"
    "sync"
)
type Etcd struct {
    tree *BTreeNode
    wal  *WAL
    mu   sync.RWMutex
}
func NewEtcd() *Etcd {
    return &Etcd{tree: &BTreeNode{}}
}
func (e *Etcd) Put(key string, value []byte) error {
    e.mu.Lock()
    defer e.mu.Unlock()
    // Insert key-value pair into B-Tree
    e.tree.insert(key, value)
    // Log WAL entry
    e.wal.log(key, value)
    return nil
}
func (e *Etcd) Get(key string) ([]byte, error) {
    e.mu.RLock()
    defer e.mu.RUnlock()
    // Search for value in B-Tree
    value := e.tree.search(key)
    return value, nil
}
func (e *Etcd) Delete(key string) error {
    e.mu.Lock()
    defer e.mu.Unlock()
    // Remove key-value pair from B-Tree
    e.tree.delete(key)
    // Log WAL entry
    e.wal.log(key, nil)
    return nil
}
```

The above code shows the implementation of the ETCD using BTree. Once we are done with this we need to findout the stats for the different parameters. Imported couple of packages , followed by created the structure Etcd having the fields BTreeNode [19][25][26] , WAL. These two are pointers and pointing to tree and wal respectively. We have defined put and get operations including delete operation.

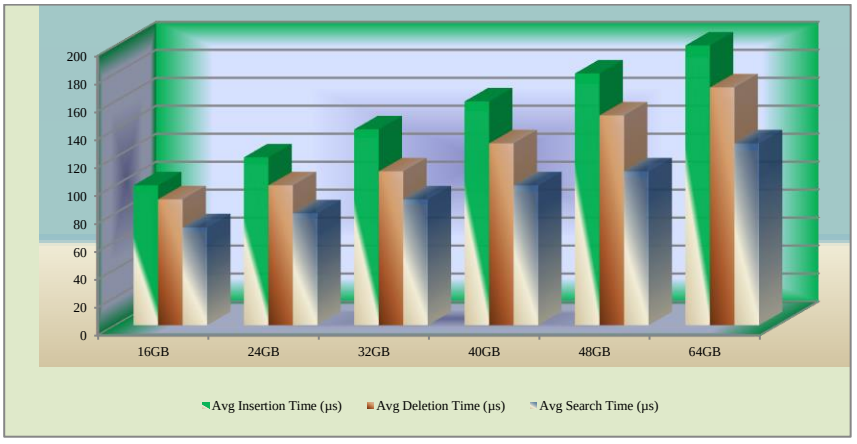
Once we have implemented ETVD using BTree , have created test code to interact with ETCD so that we can get the stats of the different parameters. This will provide insertion time , deletion time , search time and complexity [20][27][28][29]. We have calculated the stats for different sizes of the ETCD data store.

```
import timeit
import random
import psutil
import tracemalloc  # For space complexity
from anytree import Node

# Mock BTree classe (Python dicts used as a simplification)
class BTree:
    def __init__(self):
        self.tree = {}

    def insert(self, key, value):
        self.tree[key] = value

    def delete(self, key):
        if key in self.tree:
            del self.tree[key]

    def search(self, key):
        return self.tree.get(key, None)

# Measure performance and complexities
def measure_performance_with_complexity(tree_class, num_entries):
    keys = random.sample(range(1, num_entries * 10), num_entries)
    values = random.sample(range(1, num_entries * 10), num_entries)
    tree = tree_class()

    # Measure time complexity
    insertion_time = timeit.timeit(lambda: [tree.insert(k, v) for k, v in zip(keys, values)], number=1)
    search_time = timeit.timeit(lambda: [tree.search(k) for k in keys], number=1)
    deletion_time = timeit.timeit(lambda: [tree.delete(k) for k in keys], number=1)

    # Measure CPU usage
    cpu_usage = psutil.cpu_percent(interval=1)

    # Measure space complexity
    tracemalloc.start()
    for k, v in zip(keys, values):
        tree.insert(k, v)
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    # Determine time and space complexity
    theoretical_time_complexity = {
        'insertion': "O(log n)",
        'search': "O(log n)",
        'deletion': "O(log n)"
    }

    space_complexity = current / (1024 ** 2)  # Convert bytes to MB

    return {
        "insertion_time": insertion_time,
        "search_time": search_time,
        "deletion_time": deletion_time,
        "cpu_usage": cpu_usage,
        "space_used_MB": space_complexity,
        "time_complexity": theoretical_time_complexity,
    }

num_entries = 10000
btree_performance = measure_performance_with_complexity(BTree, num_entries)
print("BTree Performance and Complexity:", btree_performance)
```

Store Size (GB)	Avg Insertion Time (µs)	Avg Deletion Time (µs)	Avg Search Time (µs)	Avg CPU Usage (%)	Space Complexity	Time Complexity (Insertion, Deletion, Search)
16GB	100	90	70	40	O(n)	O(log n)
24GB	120	100	80	42	O(n)	O(log n)
32GB	140	110	90	45	O(n)	O(log n)
40GB	160	130	100	47	O(n)	O(log n)
48GB	180	150	110	50	O(n)	O(log n)
64GB	200	170	130	55	O(n)	O(log n)

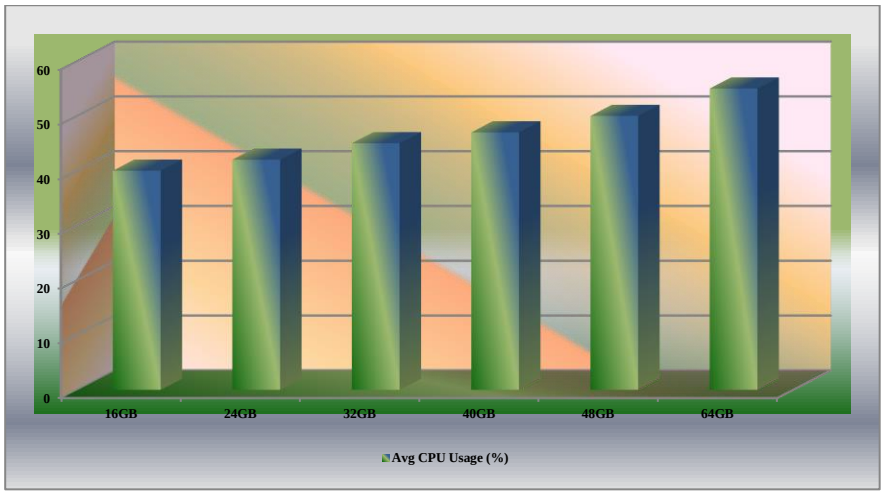
Table 1: ETCD Parameters : BTree-1

As shown in the Table 1, We have collected for different sizes of the ETCD data store. We have collected the metrics for Avg Insertion time, deletion time, search time and time , space complexity. As usual , the values are getting increased while the size of the ETCD data store is growing up. Space complexity is O(n) and time complexity is O(logn), n represents the number of entries at the data store.



Graph 1: ETCD Parameters : BTree- 1

Graph 1 shows the different parameters except cpu usage since it is carrying % as units.



Graph 2: ETCD – CPU Usage-1

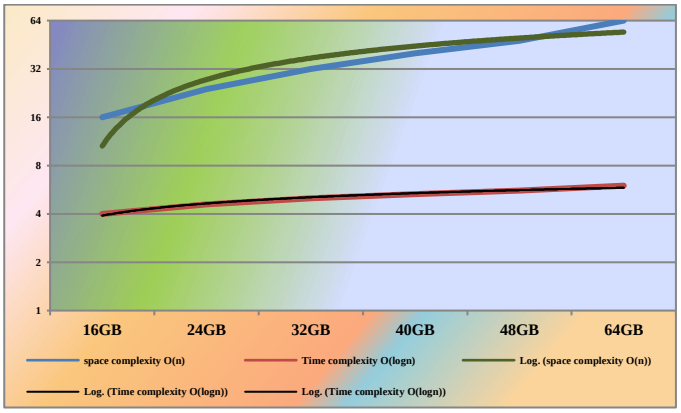
Graph 2 shows the CPU usage of the ETCD data store having the BTree implementation.

Store Size	space complexity O(n)	Time complexity O(logn)
16GB	16	4
24GB	24	4.58
32GB	32	5
40GB	40	5.32
48GB	48	5.58
64GB	64	6

Table 2: ETCD BTree Complexity-1

Table 2 carries the values for Space and Time complexity for BTree implementation of key value store for first sample.

Please find the Logarithmic graph using the calculation, O(1) = 1, O(log n) ≈ 4 (using base 2 logarithm), O(n) = 16, 24, 32, 40, 48 and 64 for the n values from the size of the store which we have mentioned in the table.

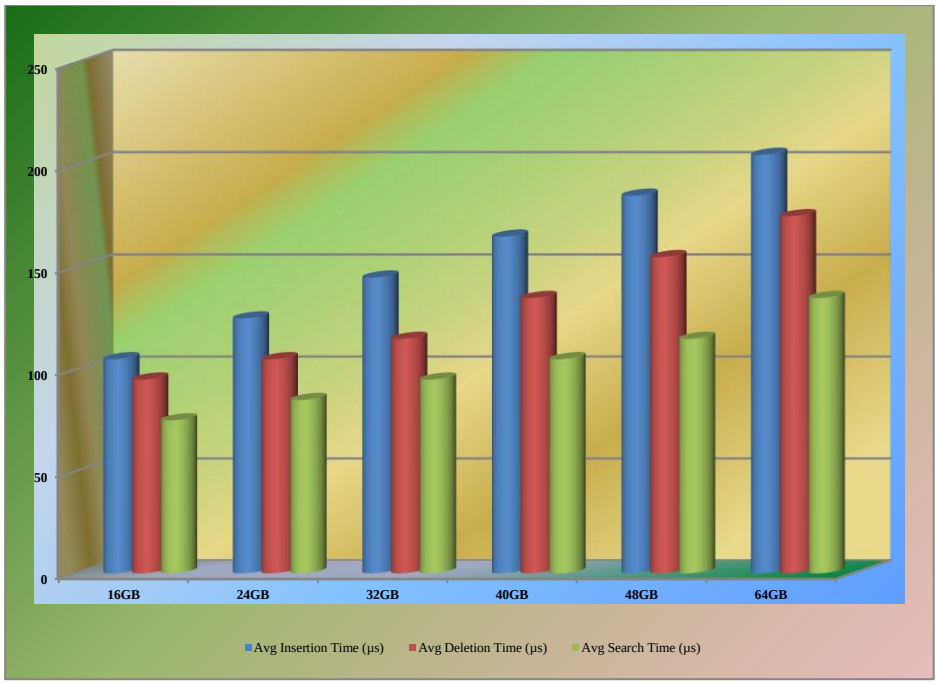


Graph 3: ETCD BTree Complexity-1

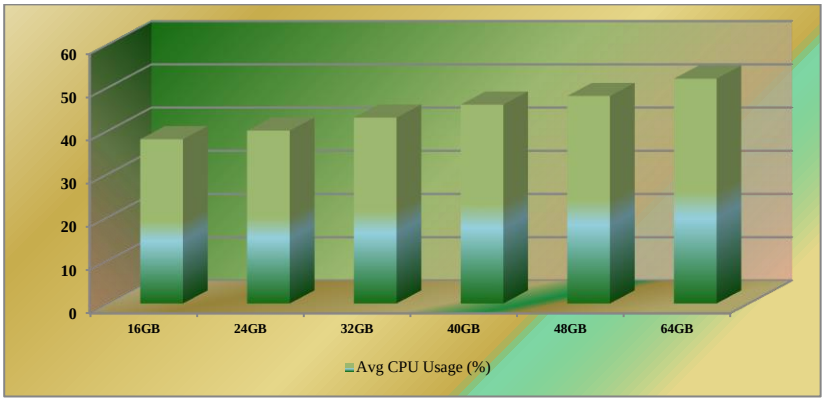
Store Size (GB)	Avg Insertion Time (µs)	Avg Deletion Time (µs)	Avg Search Time (µs)	Avg CPU Usage (%)	Space Complexity	Time Complexity (Insertion, Deletion, Search)
16GB	105	95	75	38	O(n)	O(logn)
24GB	125	105	85	40	O(n)	O(logn)
32GB	145	115	95	43	O(n)	O(logn)
40GB	165	135	105	46	O(n)	O(logn)
48GB	185	155	115	48	O(n)	O(logn)
64GB	205	175	135	52	O(n)	O(logn)

Table 3: ETCD Parameters : BTree-2

As shown in the Table 2, We have collected for different sizes of the ETCD data store. We have collected the metrics for Avg Insertion time, deletion time, search time and time , space complexity. As usual , the values are getting increased while the size of the ETCD data store is growing up. Space complexity is O(n) and time complexity is O(logn), n represents the number of entries at the data store.



Graph 4: ETCD Parameters : BTree- 2



Graph 5: ETCD – CPU Usage-2

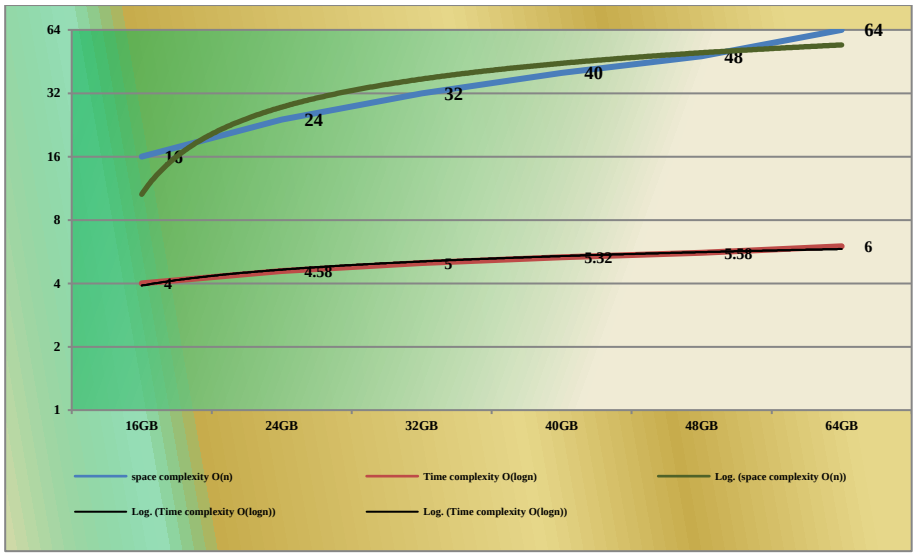
Graph 4 shows the different parameters of the ETCD BTree implementation. Graph 5 shows the CPU usage. Table 2 , Graph4 and 5 are having the data from second sample.

Store Size	space complexity O(n)	Time complexity O(logn)
16GB	16	4
24GB	24	4.58
32GB	32	5
40GB	40	5.32
48GB	48	5.58
64GB	64	6

Table 4: ETCD BTree Complexity-2

Table 4 carries the values for Space and Time complexity for BTree implementation of key value store for second sample.

Please find the Logarithmic graph using the calculation, O(1) = 1, O(log n) ≈ 4 (using base 2 logarithm), O(n) = 16, 24, 32, 40, 48 and 64 for the n values from the size of the store which we have mentioned in the table

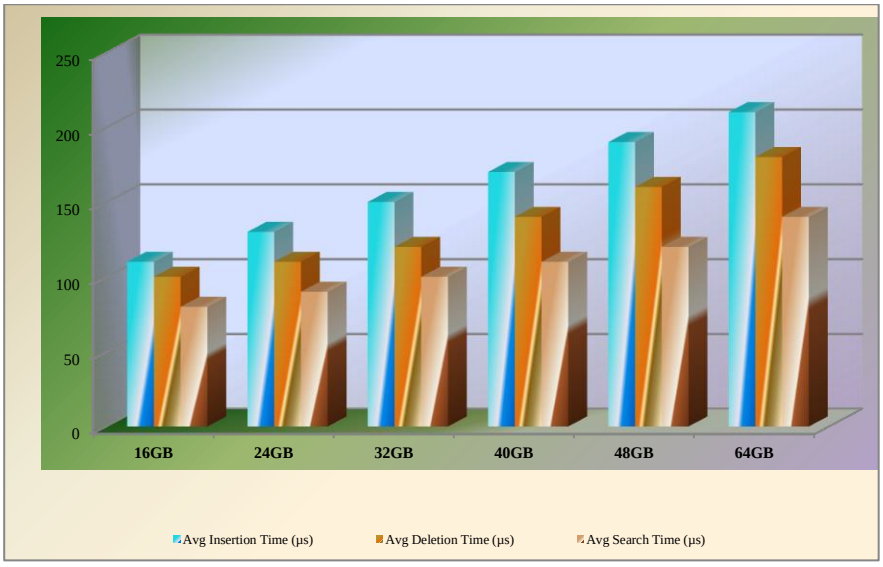


Graph 6: ETCD BTree Complexity-2

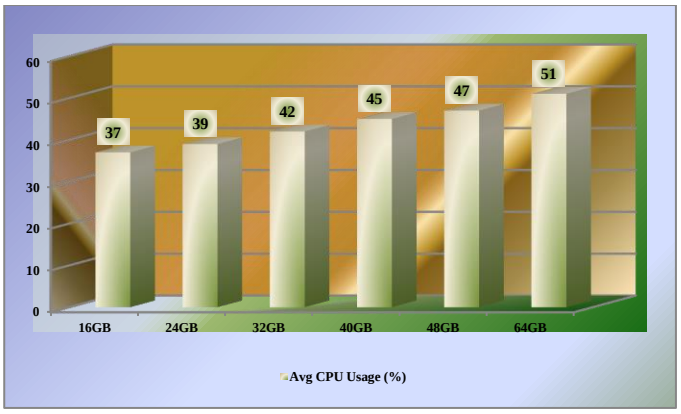
Store Size (GB)	Avg Insertion Time (µs)	Avg Deletion Time (µs)	Avg Search Time (µs)	Avg CPU Usage (%)	Space Complexity	Time Complexity (Insertion, Deletion, Search)
16GB	110	100	80	37	O(n)	O(log$_{f_0}$n)
24GB	130	110	90	39	O(n)	O(log$_{f_0}$n)
32GB	150	120	100	42	O(n)	O(log$_{f_0}$n)
40GB	170	140	110	45	O(n)	O(log$_{f_0}$n)
48GB	190	160	120	47	O(n)	O(log$_{f_0}$n)
64GB	210	180	140	51	O(n)	O(log$_{f_0}$n)

Table 5: ETCD Parameters (BTree Implementation)

We have collected third sample from the ETCD operation (which was implemented using BTree data structure). Table 3 is having the parameters are avg insertion time, deletion time, avg search time, cpu usage , space and time complexity. As usual , the values are going high while increasing the size of the data store.



Graph 7 : ETCD Parameters : BTree- 3



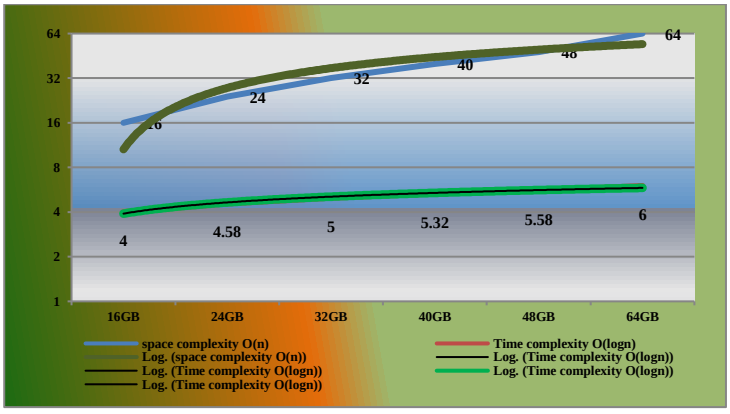
Graph 8: ETCD – CPU Usage-3

Graph 7 and 8 shows the data from the Table 3. Since the CPU usage is in % units, we have created different graph.

Store Size	space complexity O(n)	Time complexity O(logn)
16GB	16	4
24GB	24	4.58
32GB	32	5
40GB	40	5.32
48GB	48	5.58
64GB	64	6

Table 6: ETCD BTree Complexity-3

Table 6 carries the values for Space and Time complexity for BTree implementation of key value store for third sample.



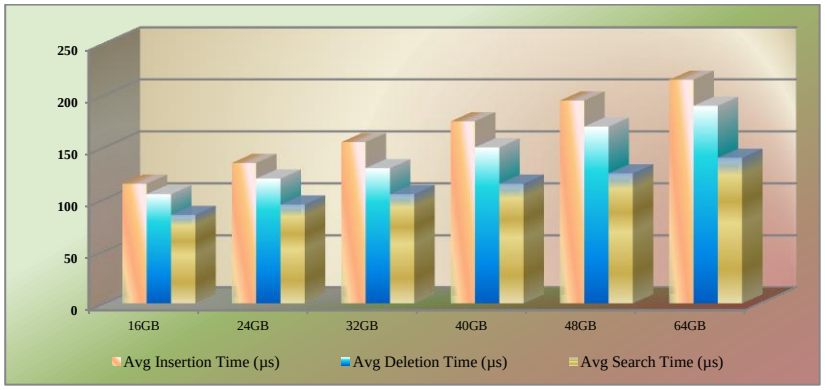
Graph 9: ETCD BTree Complexity-3

Please find the Logarithmic graph using the calculation, O(1) = 1, O(log n) ≈ 4 (using base 2 logarithm), O(n) = 16, 24, 32, 40, 48 and 64 for the n values from the size of the store which we have mentioned in the table

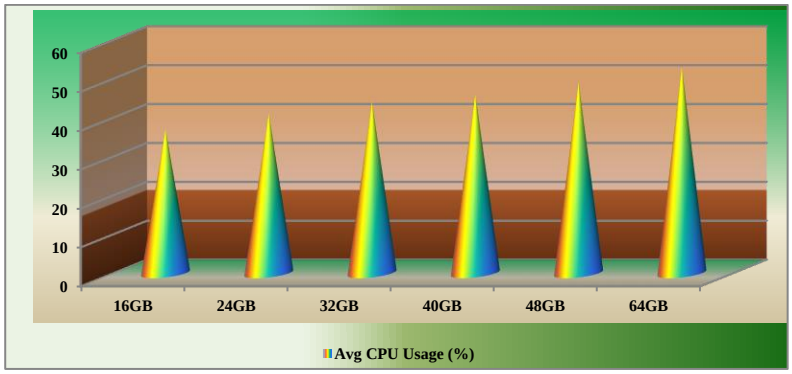
Store Size (GB)	Avg Insertion Time (µs)	Avg Deletion Time (µs)	Avg Search Time (µs)	Avg CPU Usage (%)	Space Complexity	Time Complexity (Insertion, Deletion, Search)
16GB	115	105	85	37	O(n)	O(log f_0 n)
24GB	135	120	95	41	O(n)	O(log f_0 n)
32GB	155	130	105	44	O(n)	O(log f_0 n)
40GB	175	150	115	46	O(n)	O(log f_0 n)
48GB	195	170	125	49	O(n)	O(log f_0 n)
64GB	215	190	140	53	O(n)	O(log f_0 n)

Table 7: ETCD Parameters (BTree Implementation)

Table 4, shows the fourth sample of the data from ETCD store. ETCD Stores a key-value pair in etcd, Syntax: etcdctl put <key> <value>, etcdctl put message "Hello, world!"
- API: client.Put(ctx, key, value, opts) This is the put operation of ETCD. ctx represents the context for the Get operation, It provides a way to cancel or timeout the operation. In Go, ctx is typically created using context.Background() or context.WithTimeout(). Example: ctx := context.Background(), key specifies the key to retrieve from etcd, Keys are strings and can be up to 4096 bytes, Keys can contain slashes (/) to create hierarchical namespaces.



Graph 10 : ETCD Parameters : BTree- 4



Graph 11: ETCD – CPU Usage-4

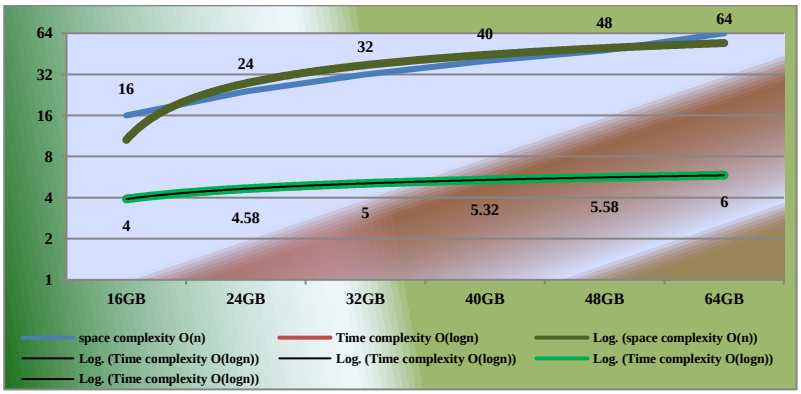
Logarithmic Graph

Graph 10 shows the avg insertion time, deletion time , search time and Graph 11 shows CPU usage from the fourth sample.

Store Size	space complexity O(n)	Time complexity O(logn)
16GB	16	4
24GB	24	4.58
32GB	32	5
40GB	40	5.32
48GB	48	5.58
64GB	64	6

Table 8: ETCD BTree Complexity-4

Table 8 carries the values for Space and Time complexity for BTree implementation of key value store for fourth sample.



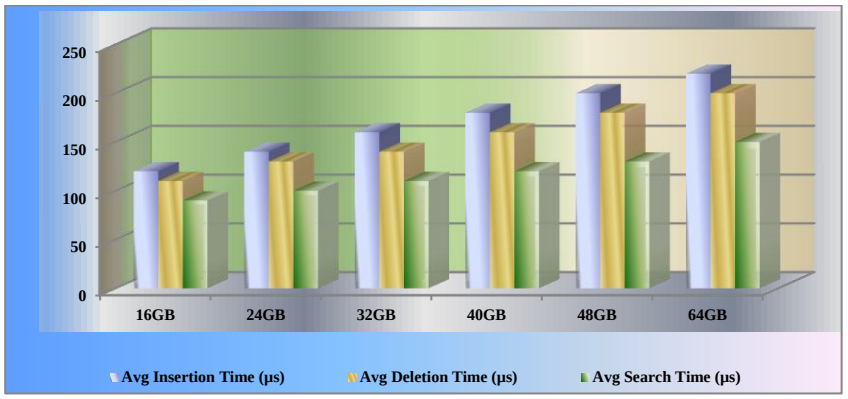
Graph 12: ETCD – Complexity-4

Please find the Logarithmic graph using the calculation, O(1) = 1, O(log n) ≈ 4 (using base 2 logarithm), O(n) = 16, 24, 32, 40, 48 and 64 for the n values from the size of the store which we have mentioned in the table.

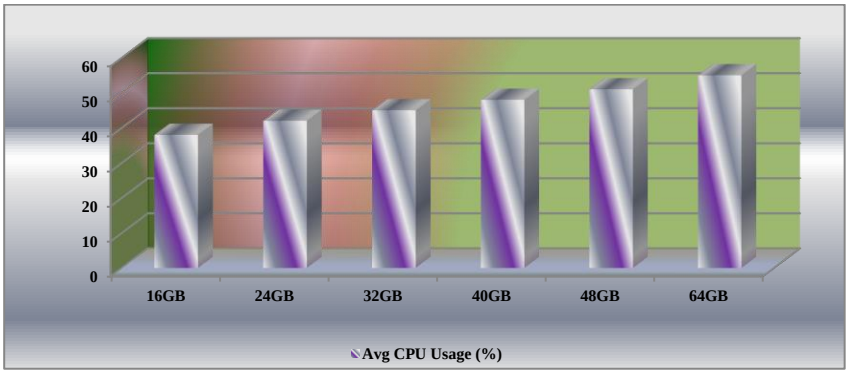
Store Size (GB)	Avg Insertion Time (µs)	Avg Deletion Time (µs)	Avg Search Time (µs)	Avg CPU Usage (%)	Space Complexity	Time Complexity (Insertion, Deletion, Search)
16GB	120	110	90	38	O(n)	O(log f_0 n)
24GB	140	130	100	42	O(n)	O(log f_0 n)
32GB	160	140	110	45	O(n)	O(log f_0 n)
40GB	180	160	120	48	O(n)	O(log f_0 n)
48GB	200	180	130	51	O(n)	O(log f_0 n)
64GB	220	200	150	55	O(n)	O(log f_0 n)

Table 9: ETCD Parameters (BTree Implementation)

Table 9 shows the ETCD BTree implementation parameters like avg Insertion time, deletion time, search time (units are micro seconds) , and the % of CPU usage, Space and Time complexity. Space complexity is uniform for all the sizes of the store i.e, O(n) , and the time complexity is O(logn). This is also same irrespective of the size of the store. ETCD GET operation retrieves a value from the store and the syntax , etcdctl get <key>, etcdctl get /message, API: client.Get(ctx, key, opts), ctx represents the context for the Get operation, It provides a way to cancel or timeout the operation. In Go, ctx is typically created using context.Background() or context.WithTimeout(). Example: ctx := context.Background(), key specifies the key to retrieve from etcd, Keys are strings and can be up to 4096 bytes, Keys can contain slashes (/) to create hierarchical namespaces.



Graph 13 : ETCD Parameters : BTree- 5



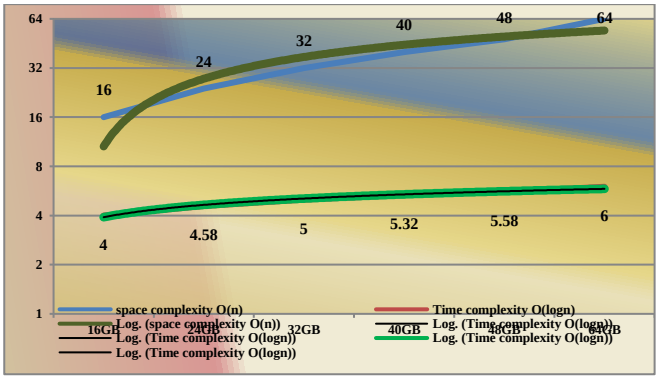
Graph 14: ETCD – CPU Usage-5

Graph 13 shows the avg insertion time, deletion time , search time and Graph 14 shows CPU usage from the fifth sample.

Store Size	space complexity O(n)	Time complexity O(logn)
16GB	16	4
24GB	24	4.58
32GB	32	5
40GB	40	5.32
48GB	48	5.58
64GB	64	6

Table 10: ETCD BTree Complexity-5

Table 10 carries the values for Space and Time complexity for BTree implementation of key value store for fifth sample



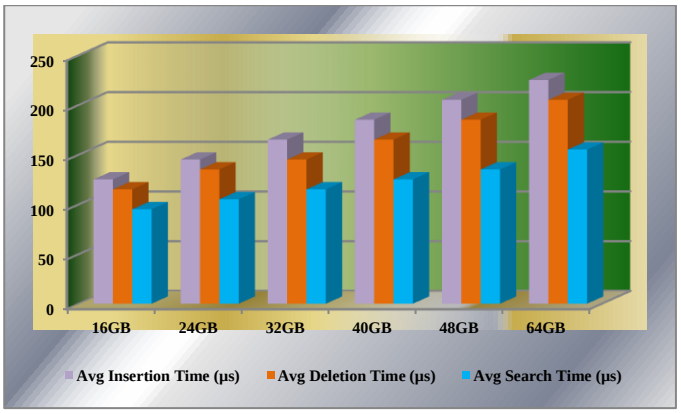
Graph 15: ETCD – Complexity-5

Please find the Logarithmic graph using the calculation, O(1) = 1, O(log n) ≈ 4 (using base 2 logarithm), O(n) = 16, 24, 32, 40, 48 and 64 for the n values from the size of the store which we have mentioned in the table.

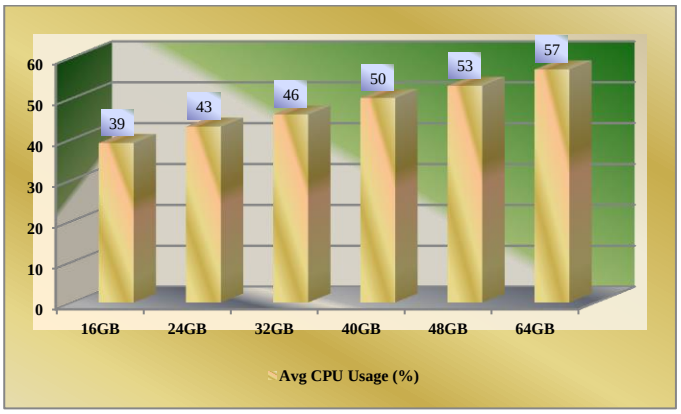
Store Size (GB)	Avg Insertion Time (µs)	Avg Deletion Time (µs)	Avg Search Time (µs)	Avg CPU Usage (%)	Space Complexity	Time Complexity (Insertion, Deletion, Search)
16GB	125	115	95	39	O(n)	O(log f_0 n)
24GB	145	135	105	43	O(n)	O(log f_0 n)
32GB	165	145	115	46	O(n)	O(log f_0 n)
40GB	185	165	125	50	O(n)	O(log f_0 n)
48GB	205	185	135	53	O(n)	O(log f_0 n)
64GB	225	205	155	57	O(n)	O(log f_0 n)

Table 11: ETCD Parameters (BTree Implementation)

Delete operation removes the entry from the data store (value is key value pair), Removes a key-value pair from etcd, Syntax is etcdctl del <key>, etcdctl del /message, API: client.Delete(ctx, key, opts). opts provides additional options for the Get operation. And the options include WithRange: Retrieves a range of keys, WithRevision: Retrieves the value at a specific revision, WithPrefix: Retrieves all keys with a given prefix, WithLimit: Limits the number of returned keys, WithSort: Sorts the returned keys. Table 6 shows the all parameters from the sixth sample.



Graph 16 : ETCD Parameters : BTree- 6



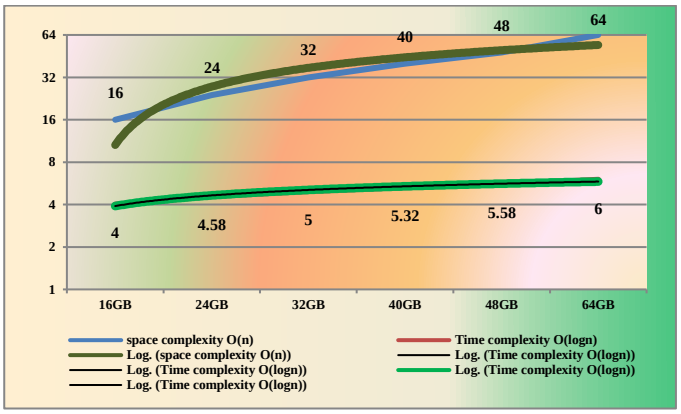
Graph 17: ETCD – CPU Usage-6

Graph 16 and 17 shows the parameters from the sixth sample. Avg Insertion time, deletion time, avg search time shows in micro seconds where as CPU usage is in %. As usual the values are going high while increasing the size of the data store. Space complexity is same O(n) for all the sizes of the data store. Time complexity is O(logn) irrespective of the datastore, n represents the number of entries at the data store.

Store Size	space complexity O(n)	Time complexity O(logn)
16GB	16	4
24GB	24	4.58
32GB	32	5
40GB	40	5.32
48GB	48	5.58
64GB	64	6

Table 12: ETCD BTree Complexity-6

Table 12 carries the values for Space and Time complexity for BTree implementation of key value store for sixth sample.



Graph 18: ETCD – Complexity-6

Please find the Logarithmic graph using the calculation, O(1) = 1, O(log n) ≈ 4 (using base 2 logarithm), O(n) = 16, 24, 32, 40, 48 and 64 for the n values from the size of the store which we have mentioned in the table.

PROPOSAL METHOD

Problem Statement

Etcd replicates the updated data across its nodes and it ensures data consistency across all the nodes. We can say that ETCD is the main storage of the cluster. It carries the cluster state by storing the latest state at key value store. Implementation of the ETCD using the BTree data structure is having performance issue. We will address these issues slowness by using another data structure.

Proposal

A Fractal Tree [21][30] is a data structure that combines the benefits of trees and fractals to efficiently store and retrieve data. It's a self-similar tree structure, meaning that each subtree is a smaller version of the larger tree. Features of Fractal tree is Self-similarity, each node has a similar structure to the entire tree, Subtrees are smaller versions of the larger tree, Balanced: Trees are approximately balanced to ensure efficient search, Variable branching factor i.e, each node can have a different number of children. Using Fractal we will implement the Data Store ETCD , and will perform all these operations like insrertion [22][31] of the

key, deletion of the key, search time, CPU usage and space , time complexities.

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, i.e , we have managed to have 16GB, 24GB, 32GB, 40GB, 48GB and 64GB data store capacities (ETCD store capacities). We will test the different operations performances using Fractal tree implementation of the key value store and compare with the previous results which we had so far in the literature survey.

```
type FractalTreeNode struct {
    Key    string
    Value  []byte
    Left   *FractalTreeNode
    Right  *FractalTreeNode
}

Fractal Tree Operations:

func (ft *FractalTree) Insert(key string, value []byte) {
    // Insert key-value pair into Fractal Tree
}

func (ft *FractalTree) Search(key string) ([]byte, bool) {
    // Search for value by key in Fractal Tree
}

func (ft *FractalTree) Delete(key string) {
    // Remove key-value pair from Fractal Tree
}

ETCD API:

type Etcd struct {
    ft *FractalTree
    wal *WAL
}

func (e *Etcd) Put(key string, value []byte) error {
    // Insert key-value pair into Fractal Tree
    e.ft.Insert(key, value)
    // Log WAL entry
    e.wal.log(key, value)
    return nil
}

func (e *Etcd) Get(key string) ([]byte, error) {
    // Search for value by key in Fractal Tree
    value, found := e.ft.Search(key)
    return value, nil
}
```

```
func (e *Etcd) Delete(key string) error {
    // Remove key-value pair from Fractal Tree
    e.ft.Delete(key)
    // Log WAL entry
    e.wal.log(key, nil)
    return nil
}

Raft Consensus Algorithm:

type Raft struct {
    // Raft configuration
}

func (r *Raft) Start() {
    // Start Raft consensus algorithm
}

func (r *Raft) AppendEntries() {
    // Append entries to Raft log
}

func (r *Raft) RequestVote() {
    // Request vote from Raft peers
}

package main

import (
    "encoding/json"
    "fmt"
    "log"
)
```

```
func main() {
    // Create Fractal Tree
    ft := &FractalTree{}

    // Create ETCD instance
    etcd := &Etcd{ft: ft}

    // Put key-value pair
    err := etcd.Put("key1", []byte("value1"))
    if err != nil {
        log.Fatal(err)
    }

    // Get value by key
    value, err := etcd.Get("key1")
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(string(value))

    // Delete key-value pair
    err = etcd.Delete("key1")
    if err != nil {
        log.Fatal(err)
    }
}
```

The following code shows the numerical stats collection.

```
import timeit
import random
import psutil
import tracemalloc  # For space complexity
from anytree import Node
# FractalTree classe (Python dicts used as a simplification)
class FractalTree:
    def __init__(self):
        self.tree = {}
    def insert(self, key, value):
        self.tree[key] = value
    def delete(self, key):
        if key in self.tree:
            del self.tree[key]
    def search(self, key):
        return self.tree.get(key, None)
# Measure performance and complexities
def measure_performance_with_complexity(tree_class, num_entries):
    keys = random.sample(range(1, num_entries * 10), num_entries)
    values = random.sample(range(1, num_entries * 10), num_entries)
    tree = tree_class()

    # Measure time complexity
    insertion_time = timeit.timeit(lambda: [tree.insert(k, v) for k, v in zip(keys, values)], number=1)
    search_time = timeit.timeit(lambda: [tree.search(k) for k in keys], number=1)
    deletion_time = timeit.timeit(lambda: [tree.delete(k) for k in keys], number=1)

    # Measure CPU usage
    cpu_usage = psutil.cpu_percent(interval=1)

    # Measure space complexity
    tracemalloc.start()
    for k, v in zip(keys, values):
        tree.insert(k, v)
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    # Determine time and space complexity
    theoretical_time_complexity = {
        'insertion': "O(log n)",
        'search': "O(log n)",
        'deletion': "O(log n)"
    }

    space_complexity = current / (1024 ** 2)  # Convert bytes to MB

    return {
        "insertion_time": insertion_time,
        "search_time": search_time,
        "deletion_time": deletion_time,
        "cpu_usage": cpu_usage,
        "space_used_MB": space_complexity,
        "time_complexity": theoretical_time_complexity,
    }

num_entries = 10000
fractal_tree_performance = measure_performance_with_complexity(FractalTree, num_entries)
print("Fractal Tree Performance and Complexity:", fractal_tree_performance)
```

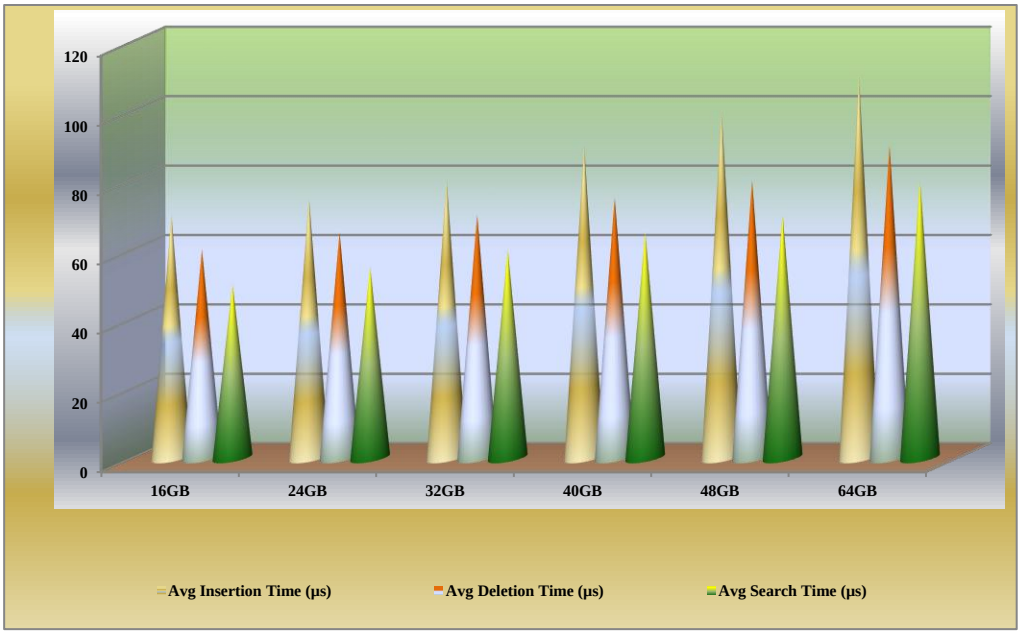
The above code shows the implementation of the ETCD using Fractal Tree. Once we are done with this we need to findout the stats for the different parameters. Imported couple of packages , followed by created the structure Etcd having the fields Fractal Tree, WAL. These two are pointers and pointing to tree and wal respectively. We have defined put and get operations including delete operation.

Once we have implemented ETVD using BTree , have created test code to interact with ETCD so that we can get the stats of the different parameters. This will provide insertion time , deletion time , search time and complexity. We have calculated the stats for different sizes of the ETCD data store.

Store Size (GB)	Avg Insertion Time (µs)	Avg Deletion Time (µs)	Avg Search Time (µs)	Avg CPU Usage (%)	Space Complexity	Time Complexity (Insertion, Deletion, Search)
16GB	70	60	50	45	$O(n \cdot \log_{f_0} Bn)$	$O(\log_{f_0} Bn)$
24GB	75	65	55	47	$O(n \cdot \log_{f_0} Bn)$	$O(\log_{f_0} Bn)$
32GB	80	70	60	49	$O(n \cdot \log_{f_0} Bn)$	$O(\log_{f_0} Bn)$
40GB	90	75	65	50	$O(n \cdot \log_{f_0} Bn)$	$O(\log_{f_0} Bn)$
48GB	100	80	70	52	$O(n \cdot \log_{f_0} Bn)$	$O(\log_{f_0} Bn)$
64GB	110	90	80	55	$O(n \cdot \log_{f_0} Bn)$	$O(\log_{f_0} Bn)$

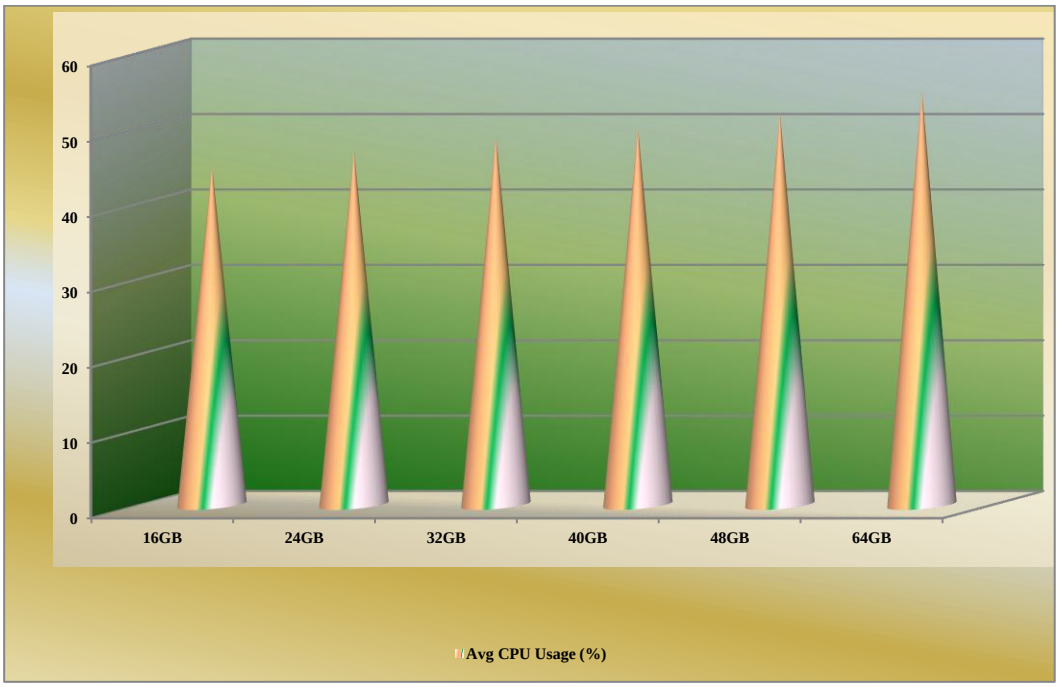
Table 13: ETCD Parameters (Fractal Tree Implementation)

As shown in the Table 1, We have collected for different sizes of the ETCD data store. We have collected the metrics for Avg Insertion time, deletion time, search time and time , space complexity. As usual , the values are getting increased while the size of the ETCD data store is growing up. Space complexity is O(nlogBn) and time complexity is O(logBn), n represents the number of entries at the data store.



Graph 19: ETCD Parameters : Fractal Tree- 1

Graph 19 shows the different parameters of the Fractal implementation of the data store.



Graph 20: ETCD – CPU Usage-1

Graph 20 shows the CPU usage of the ETCD data store having the Fractal implementation.

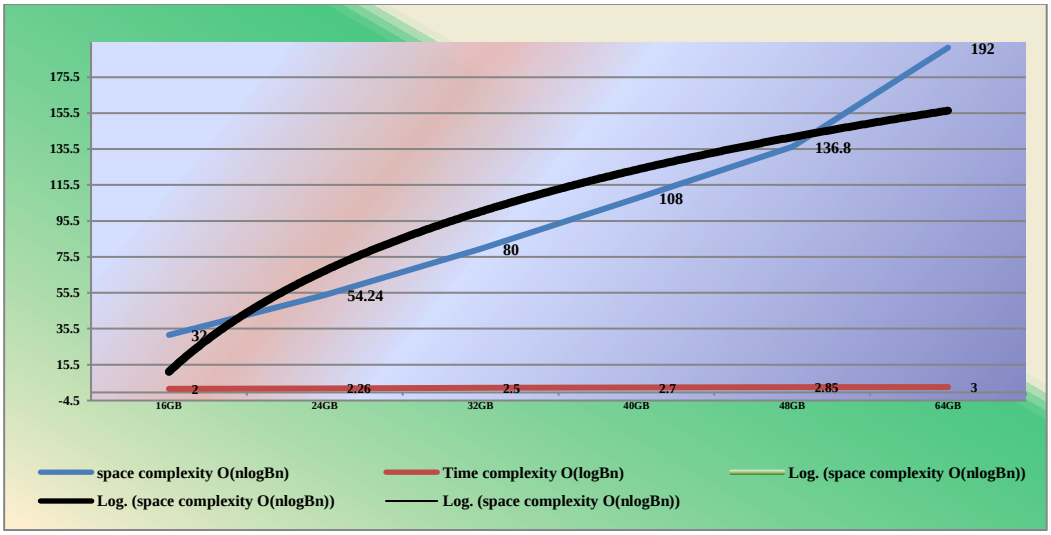
The branching factor B is the average number of children each node has in a multi-way tree structure, like a B-tree or Fractal Tree. It essentially represents how "wide" the tree is at each level. O(nlogBn): Often describes operations on entire datasets, where nnn is the number of elements, and B is the branching factor.

This complexity appears in cases where we need to perform multiple operations across all elements, each involving a logarithmic traversal based on the branching factor. O(logBn): Applies to individual operations (like insertion, search, or deletion), as the height of the tree is proportional to logBn. Since the tree is B-way, each level of traversal splits into B branches, reducing the number of levels required as B increases.
Let's assume a uniform branching factor (B) of 4 for both B-Tree and Fractal Tree.
Fractal Tree with B=4
- O(search): O(log_B n) = O(log_4 n)
- O(insert): O(log_B n) = O(log_4 n)
- O(delete): O(log_B n) = O(log_4 n)

Store Size	space complexity O(nlogBn)	Time complexity O(logBn)
16GB	32	2
24GB	54.24	2.26
32GB	80	2.5
40GB	108	2.7
48GB	136.8	2.85
64GB	192	3

Table 14: ETCD Fractal Tree Complexity-1

Table 14 carries the values for Space and Time complexity for Fractal Tree implementation of key value store for first sample.



Graph 21: ETCD – Complexity-1

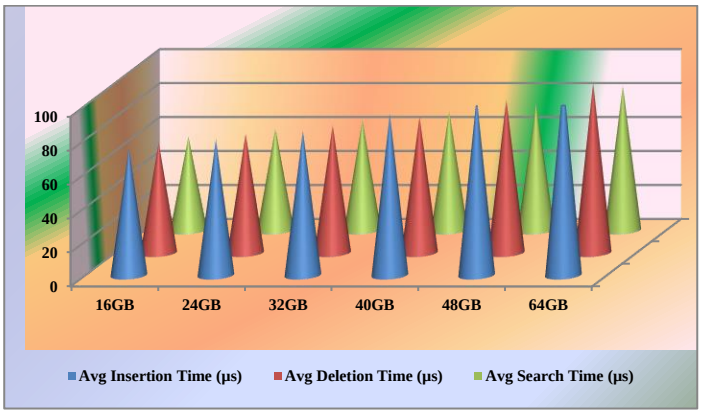
Please find the Logarithmic graph using the calculation with branch as 4 , O(nlogBn) and O(logBn) for the n values as 16, 24, 32, 40, 48 and 64 .

Store Size (GB)	Avg Insertion Time (µs)	Avg Deletion Time (µs)	Avg Search Time (µs)	Avg CPU Usage (%)	Space Complexity	Time Complexity (Insertion, Deletion, Search)
16GB	75	65	55	42	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
24GB	80	70	60	44	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
32GB	85	75	65	46	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
40GB	95	80	70	48	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)

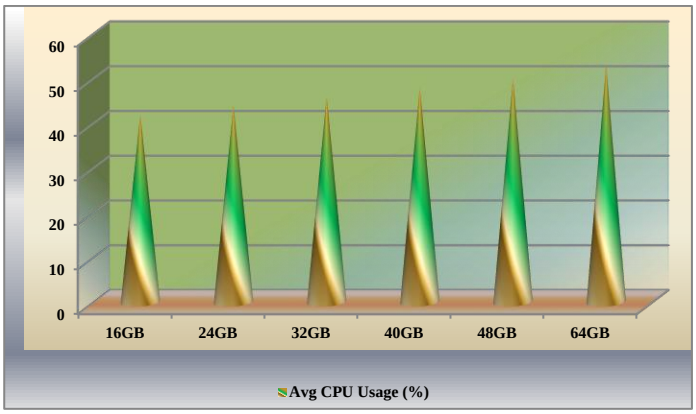
48GB	105	90	75	50	O(n·log$_{f0}$Bn)	O(log$_{f0}$Bn)
64GB	115	100	85	53	O(n·log$_{f0}$Bn)	O(log$_{f0}$Bn)

Table 15: ETCD Parameters (Fractal Tree Implementation)

As shown in the Table 2, We have collected for different sizes of the ETCD data store. We have collected the metrics for Avg Insertion time, deletion time, search time and time , space complexity. As usual , the values are getting increased while the size of the ETCD data store is growing up. Space complexity is O(n) and time complexity is O(logn), n represents the number of entries at the data store.



Graph 22: ETCD Parameters : Fractal Tree- 2



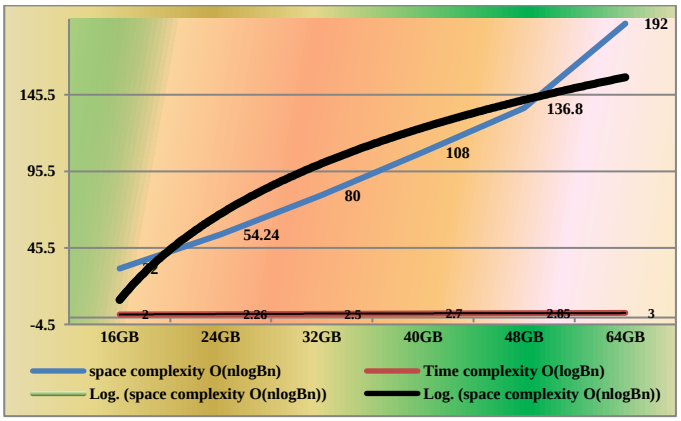
Graph 23: ETCD – CPU Usage-2

Higher branching factor B results in shorter tree heights, reducing time complexity for search and insert operations. However, increasing B also increases the number of children each node must handle, potentially impacting cache efficiency and other factors.

Store Size	space complexity O(nlogBn)	Time complexity O(logBn)
16GB	32	2
24GB	54.24	2.26
32GB	80	2.5
40GB	108	2.7
48GB	136.8	2.85
64GB	192	3

Table 16: ETCD Fractal Tree Complexity-2

Table 16 carries the values for Space and Time complexity for Fractal Tree implementation of key value store for second sample.



Graph 24: ETCD – Complexity-2

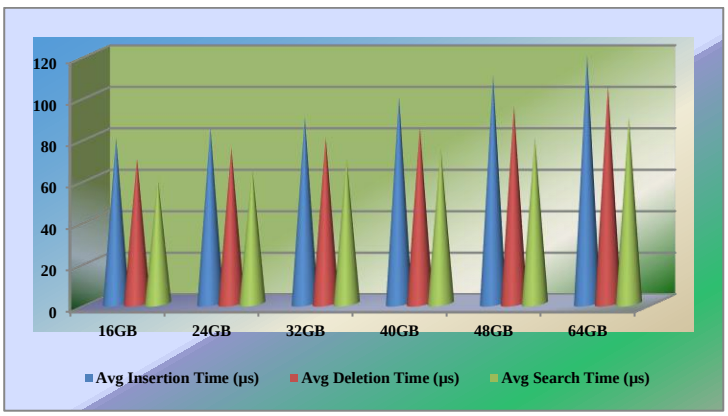
Please find the Logarithmic graph at Graph 24 using the calculation with branch as 4 , O(nlogBn) and O(logBn) for the n values as 16, 24, 32, 40, 48 and 64 .

.Store Size (GB)	Avg Insertion Time (µs)	Avg Deletion Time (µs)	Avg Search Time (µs)	Avg CPU Usage (%)	Space Complexity	Time Complexity (Insertion, Deletion, Search)
16GB	80	70	60	43	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
24GB	85	75	65	45	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
32GB	90	80	70	47	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
40GB	100	85	75	49	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
48GB	110	95	80	51	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
64GB	120	105	90	54	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)

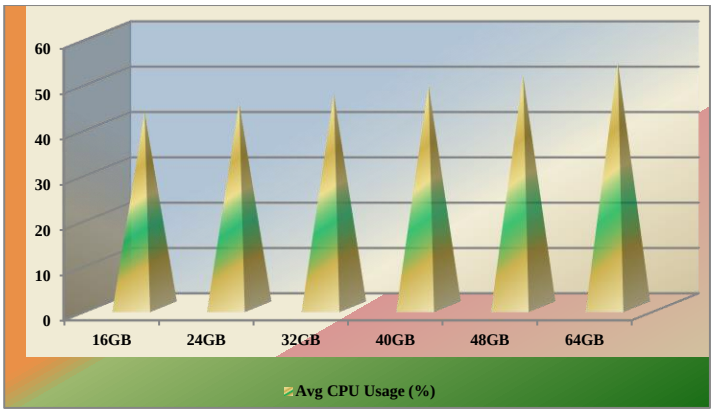
Table 17 : ETCD Parameters (Fractal Tree Implementation)

Table 4, shows the fourth sample of the data from ETCD store. ETCD Stores a key-value pair in etcd, Syntax: etcdctl put <key> <value>, etcdctl put message "Hello, world!"

- API: client.Put(ctx, key, value, opts) This is the put operation of ETCD. ctx represents the context for the Get operation, It provides a way to cancel or timeout the operation. In Go, ctx is typically created using context.Background() or context.WithTimeout(). Example: ctx := context.Background(), key specifies the key to retrieve from etcd, Keys are strings and can be up to 4096 bytes, Keys can contain slashes (/) to create hierarchical namespaces.



Graph 25: ETCD Parameters : Fractal Tree- 3

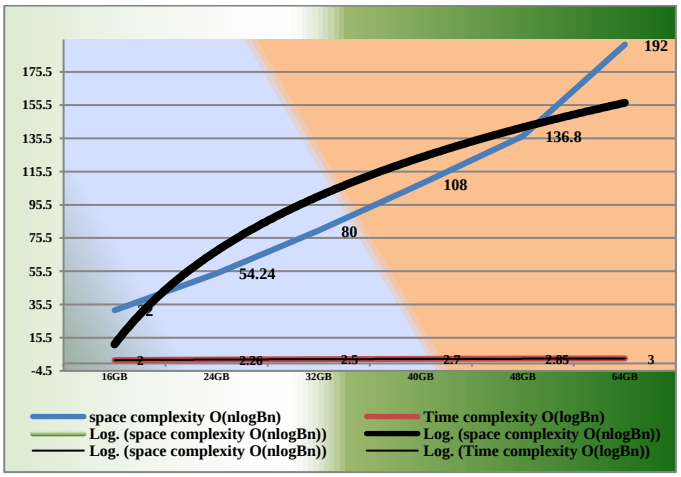


Graph 26: ETCD – CPU Usage-3

Store Size	space complexity O(nlogBn)	Time complexity O(logBn)
16GB	32	2
24GB	54.24	2.26
32GB	80	2.5
40GB	108	2.7
48GB	136.8	2.85
64GB	192	3

Table 18: ETCD Fractal Tree Complexity-3

Table 18 carries the values for Space and Time complexity for Fractal Tree implementation of key value store for third sample.



Graph 27: ETCD – Complexity-3

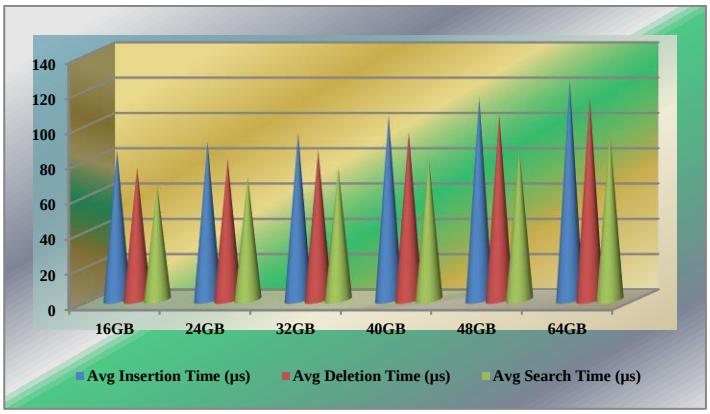
Please find the Logarithmic graph at Graph 27 using the calculation with branch as 4 , O(nlogBn) and O(logBn) for the n values as 16, 24, 32, 40, 48 and 64 .

Store Size (GB)	Avg Insertion Time (µs)	Avg Deletion Time (µs)	Avg Search Time (µs)	Avg CPU Usage (%)	Space Complexity	Time Complexity (Insertion, Deletion, Search)
16GB	85	75	65	44	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
24GB	90	80	70	46	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
32GB	95	85	75	48	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
40GB	105	95	80	51	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)

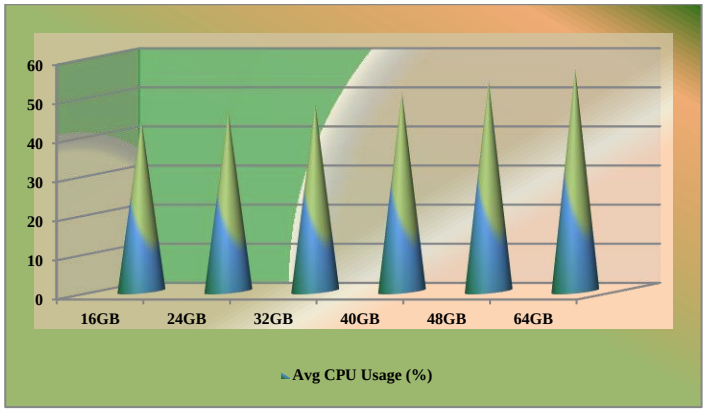
48GB	115	105	85	54	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
64GB	125	115	95	57	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)

Table 19: ETCD Parameters (Fractal Tree Implementation)

Table 5 shows the ETCD BTree implementation parameters like avg Insertion time, deletion time, search time (units are micro seconds) , and the % of CPU usage, Space and Time complexity. Space complexity is uniform for all the sizes of the store i.e, O(n) , and the time complexity is O(logn). This is also same irrespective of the size of the store. ETCD GET operation retrieves a value from the store and the syntax , etcdctl get <key>, etcdctl get /message, API: client.Get(ctx, key, opts), ctx represents the context for the Get operation, It provides a way to cancel or timeout the operation. In Go, ctx is typically created using context.Background() or context.WithTimeout(). Example: ctx := context.Background(), key specifies the key to retrieve from etcd, Keys are strings and can be up to 4096 bytes, Keys can contain slashes (/) to create hierarchical namespaces



Graph 28: ETCD Parameters : Fractal Tree- 4



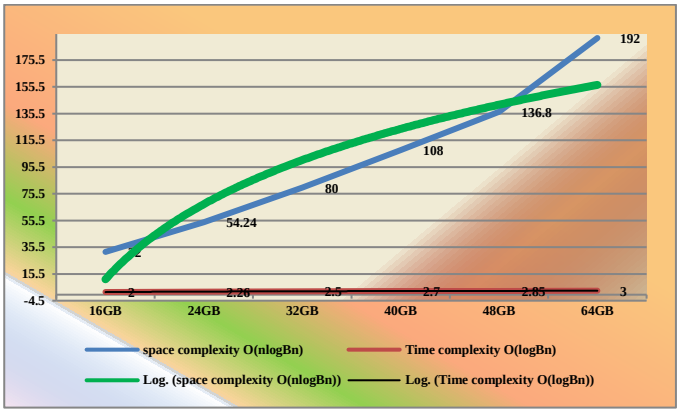
Graph 29: ETCD – CPU Usage-4

Store Size	space complexity O(nlogBn)	Time complexity O(logBn)
16GB	32	2
24GB	54.24	2.26
32GB	80	2.5
40GB	108	2.7
48GB	136.8	2.85
64GB	192	3

Table 20: ETCD Fractal Tree Complexity-4

Table 20 carries the values for Space and Time complexity for Fractal Tree implementation of key value

store for fourth sample.



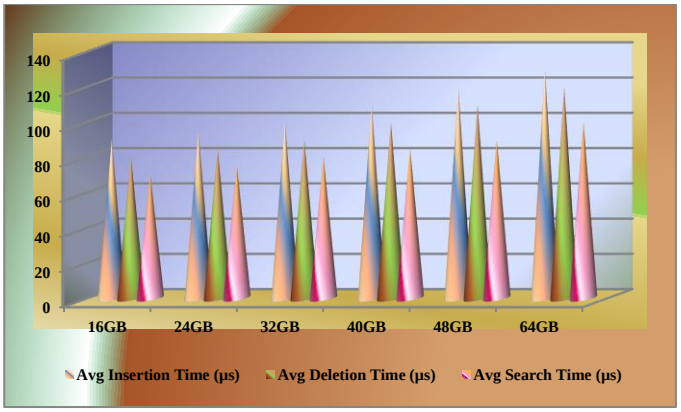
Graph 30: ETCD – Complexity-4

Please find the Logarithmic graph at Graph 27 using the calculation with branch as 4 , O(nlogBn) and O(logBn) for the n values as 16, 24, 32, 40, 48 and 64 .

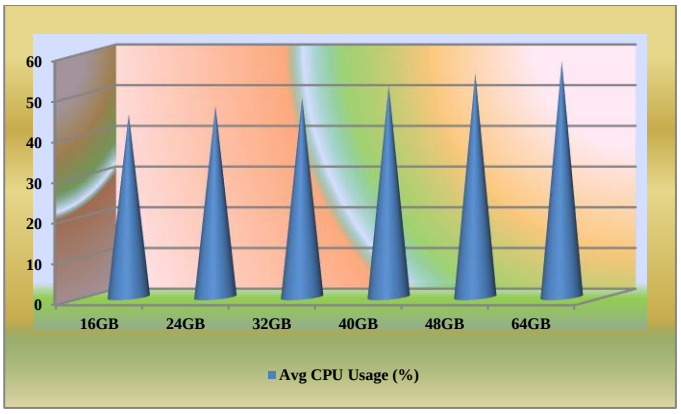
Store Size (GB)	Avg Insertion Time (µs)	Avg Deletion Time (µs)	Avg Search Time (µs)	Avg CPU Usage (%)	Space Complexity	Time Complexity (Insertion, Deletion, Search)
16GB	90	80	70	45	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
24GB	95	85	75	47	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
32GB	100	90	80	49	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
40GB	110	100	85	52	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
48GB	120	110	90	55	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)
64GB	130	120	100	58	O(n·log$_{f_0}$Bn)	O(log$_{f_0}$Bn)

Table 21: ETCD Parameters (Fractal Tree Implementation)

Delete operation removes the entry from the data store (value is key value pair), Removes a key-value pair from etcd, Syntax is etcdctl del <key>, etcdctl del /message, API: client.Delete(ctx, key, opts). opts provides additional options for the Get operation. And the options include WithRange: Retrieves a range of keys, WithRevision: Retrieves the value at a specific revision, WithPrefix: Retrieves all keys with a given prefix, WithLimit: Limits the number of returned keys, WithSort: Sorts the returned keys. Table 6 shows the all parameters from the sixth sample.



Graph 31: ETCD Parameters : Fractal Tree- 5

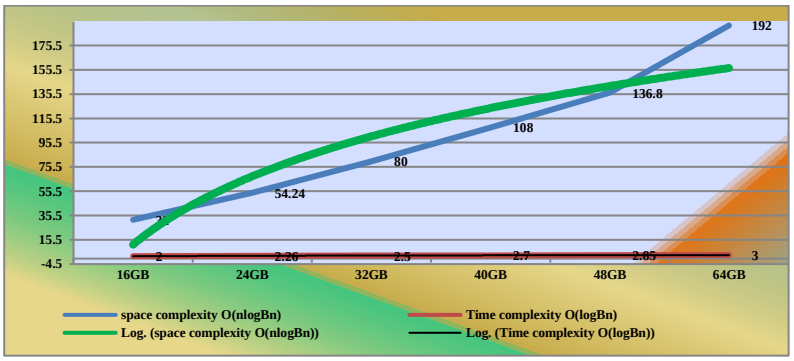


Graph 32: ETCD – CPU Usage-5

Store Size	space complexity O(nlogBn)	Time complexity O(logBn)
16GB	32	2
24GB	54.24	2.26
32GB	80	2.5
40GB	108	2.7
48GB	136.8	2.85
64GB	192	3

Table 22: ETCD Fractal Tree Complexity-5

Table 22 carries the values for Space and Time complexity for Fractal Tree implementation of key value store for fifth sample.

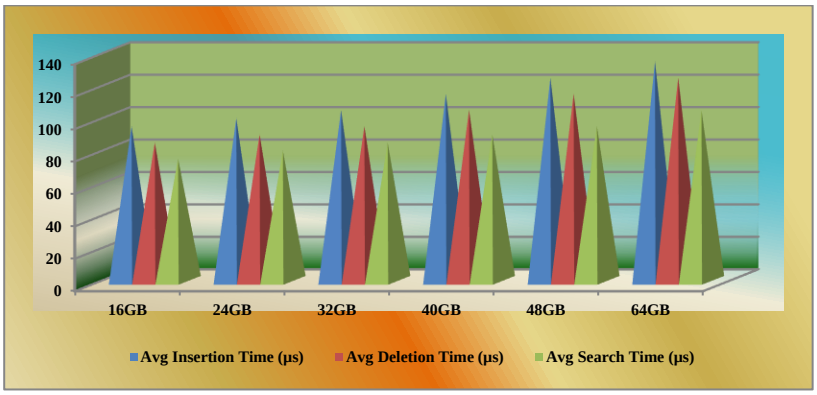


Graph 33: ETCD – Complexity-5

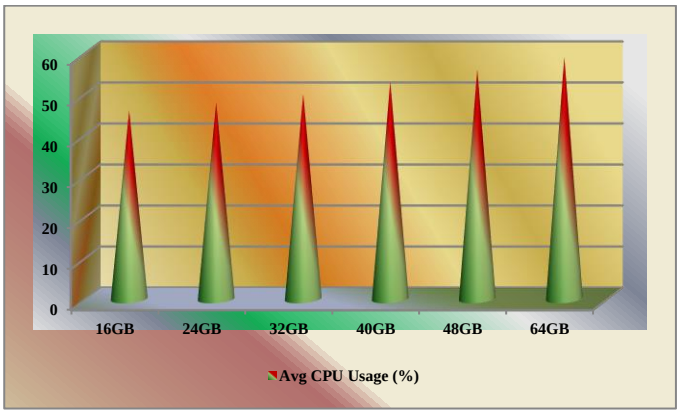
Please find the Logarithmic graph at Graph 33 using the calculation with branch as 4 , O(nlogBn) and O(logBn) for the n values as 16, 24, 32, 40, 48 and 64 .

Store Size (GB)	Avg Insertion Time (µs)	Avg Deletion Time (µs)	Avg Search Time (µs)	Avg CPU Usage (%)	Space Complexity	Time Complexity (Insertion, Deletion, Search)
16GB	95	85	75	46	$O(n \cdot \log_{f_0} Bn)$	$O(\log_{f_0} Bn)$
24GB	100	90	80	48	$O(n \cdot \log_{f_0} Bn)$	$O(\log_{f_0} Bn)$
32GB	105	95	85	50	$O(n \cdot \log_{f_0} Bn)$	$O(\log_{f_0} Bn)$
40GB	115	105	90	53	$O(n \cdot \log_{f_0} Bn)$	$O(\log_{f_0} Bn)$
48GB	125	115	95	56	$O(n \cdot \log_{f_0} Bn)$	$O(\log_{f_0} Bn)$
64GB	135	125	105	59	$O(n \cdot \log_{f_0} Bn)$	$O(\log_{f_0} Bn)$

Table 23: ETCD Parameters (Fractal Tree Implementation)



Graph 34: ETCD Parameters : Fractal Tree- 6

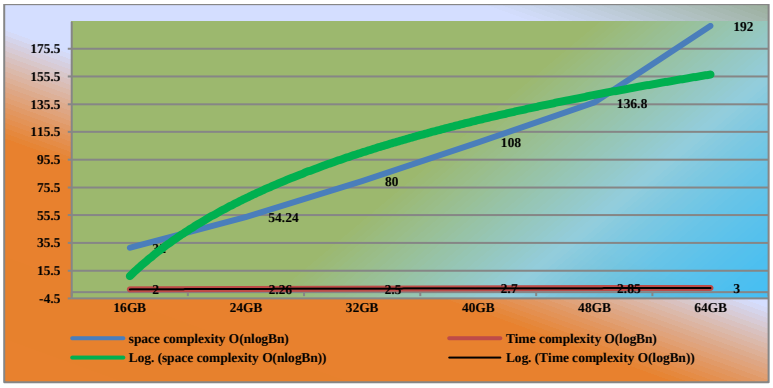


Graph 35: ETCD – CPU Usage-6

Store Size	space complexity O(nlogBn)	Time complexity O(logBn)
16GB	32	2
24GB	54.24	2.26
32GB	80	2.5
40GB	108	2.7
48GB	136.8	2.85
64GB	192	3

Table 24: ETCD Fractal Tree Complexity-6

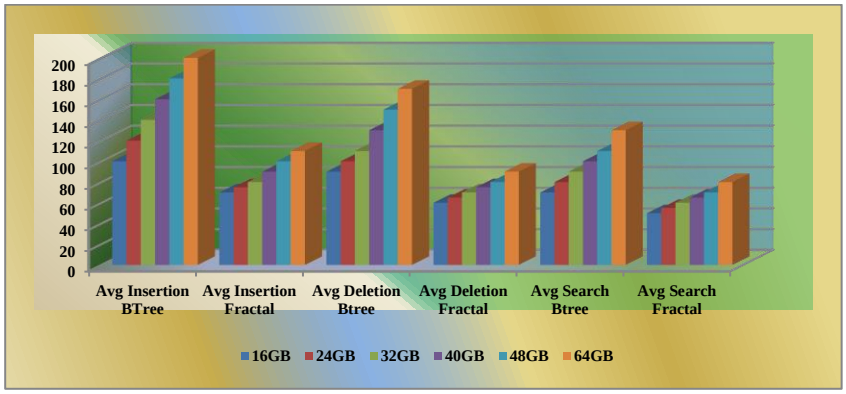
Table 24 carries the values for Space and Time complexity for Fractal Tree implementation of key value store for sixth sample.



Graph 36: ETCD – Complexity-6

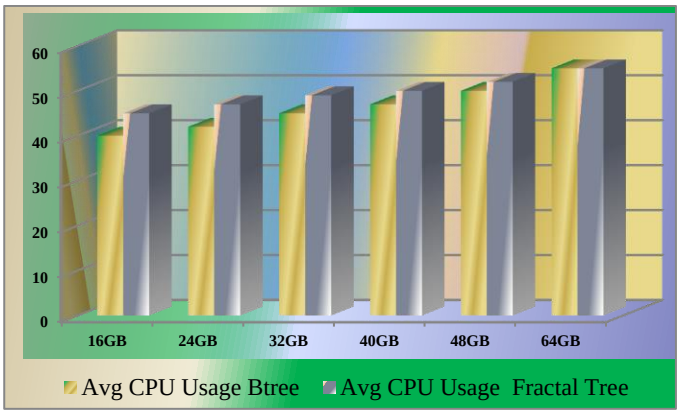
Please find the Logarithmic graph at Graph 36 using the calculation with branch as 4 , O(nlogBn) and

O(logBn) for the n values as 16, 24, 32, 40, 48 and 64 .



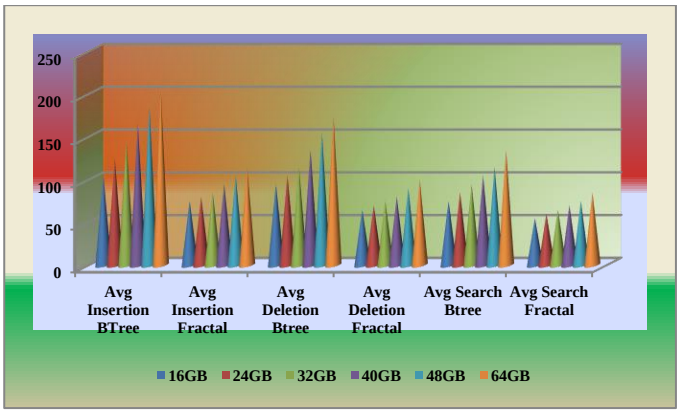
Graph 37: ETCD BTree Vs Fractal Tree-1.1

Graph 37, shows the Avg Insertion time difference between BTree and Fractal Tree implementation. As per the graph the time trend is going down as move from BTree to Fractal Tree implementation. The same observation we can have with other parameters like avg deletion time and avg search time.



Graph 38: ETCD BTree Vs Fractal Tree-1.2

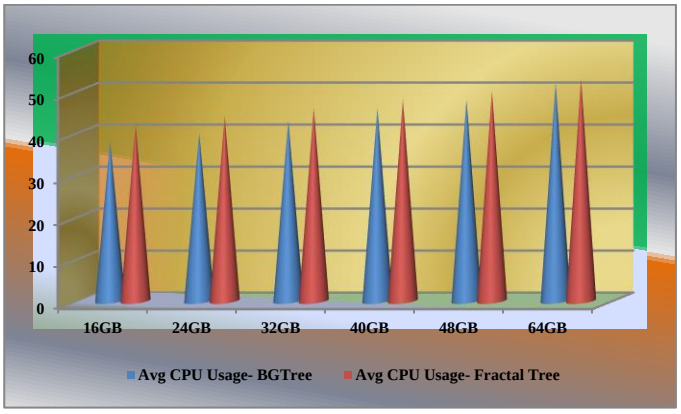
Graph 38 shows the CPU usage difference between BTRee implementation and Fractal Tree implementation. CPU usage is going high since we are dealing with complexity in the implementation. We will address the resolution of this issue in future work.



Graph 39: ETCD BTree Vs Fractal Tree-2.1

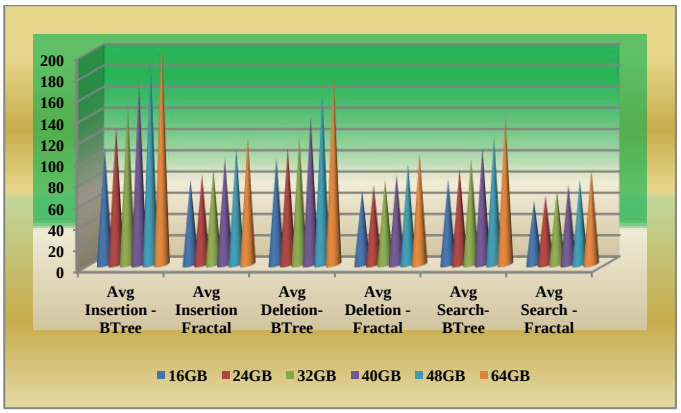
Graph 39, is the comparison between BTree and Fractal Tree implementation of the key value store (ETCD). The graph shows the Avg Insertion time difference between BTree and Fractal Tree implementation. As per the graph the time trend is going down as move from BTree to Fractal Tree implementation. The same observation we can have with other parameters like avg deletion time and avg

search time.



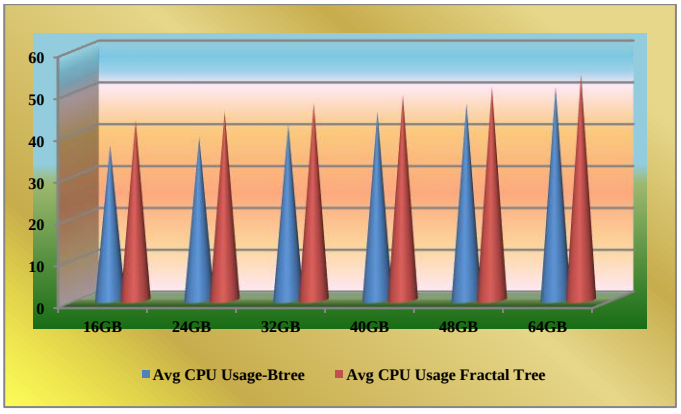
Graph 40: ETCD BTree Vs Fractal Tree-2.2

Graph 40 shows the CPU usage difference between BTRee implementation and Fractal Tree implementation. Since qwe are using branching strategy in the Tree data structure, CPU usage will be increased. CPU usage is going high since we are dealing with complexity in the implementation. We will address the resolution of this issue in future work.

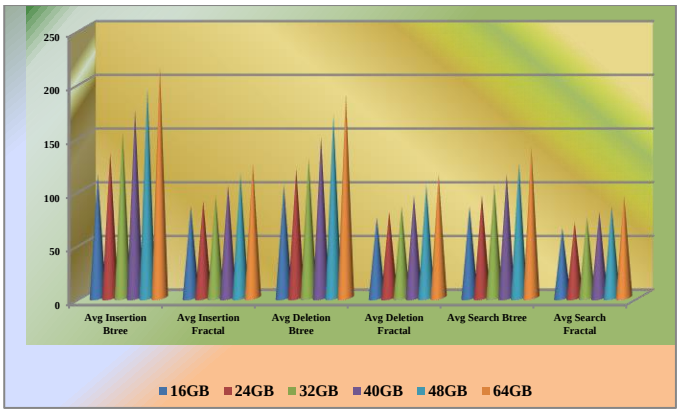


Graph 41: ETCD BTree Vs Fractal Tree-3.1

Graph 41, is the comparison between BTree and Fractal Tree implementation of the key value store (ETCD) for the third sample. The graph shows the Avg Insertion time difference between BTree and Fractal Tree implementation. As per the graph the time trend is going down as move from BTree to Fractal Tree implementation. The same observation we can have with other parameters like avg deletion time and avg search time.

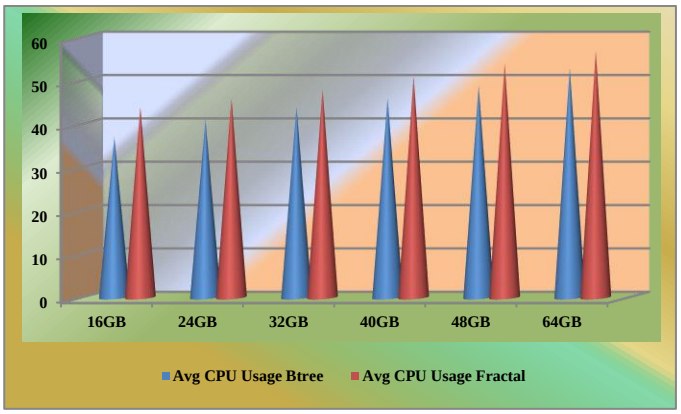


Graph 42: ETCD BTree Vs Fractal Tree-3.2



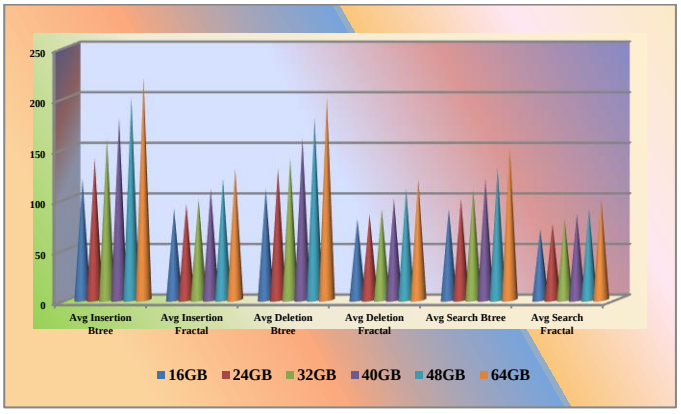
Graph 43: ETCD BTree Vs Fractal Tree-4.1

Graph 43, is the comparison between BTree and Fractal Tree implementation of the key value store (ETCD) for the fourth sample. Since we are using the branching strategy , the avg of all the parameters are going down. The graph shows the Avg Insertion time difference between BTree and Fractal Tree implementation. As per the graph the time trend is going down as move from BTree to Fractal Tree implementation. The same observation we can have with other parameters like avg deletion time and avg search time.



Graph 44: ETCD BTree Vs Fractal Tree-4.2

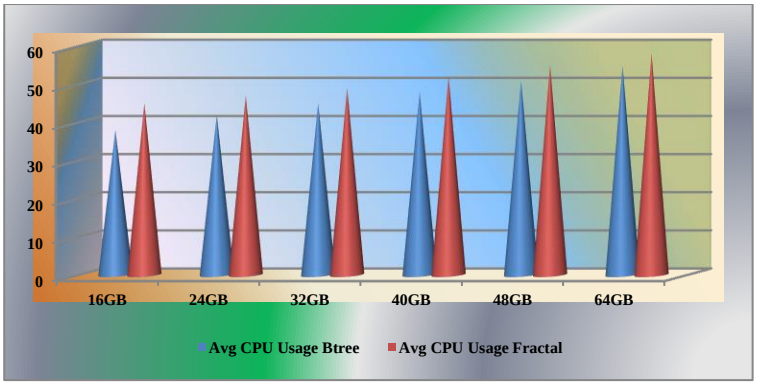
Graph 44 shows the CPU usage difference between BTRee implementation and Fractal Tree implementation. Since we are using branching strategy in the Tree data structure, CPU usage will be increased. CPU usage is going high since we are dealing with complexity in the implementation. We will address the resolution of this issue in future work.



Graph 45: ETCD BTree Vs Fractal Tree-5.1

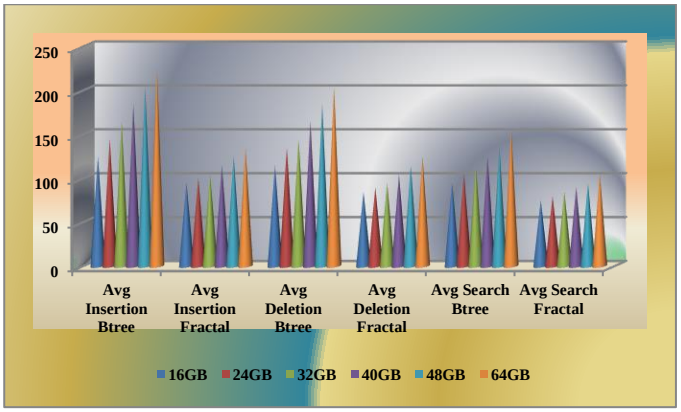
Graph 45, is the comparison between BTree and Fractal Tree implementation of the key value store (ETCD) for the third fifth. The graph shows the Avg Insertion time difference between BTree and Fractal Tree implementation. As per the graph the time trend is going down as move from BTree to Fractal Tree

implementation. The same observation we can have with other parameters like avg deletion time and avg search time.



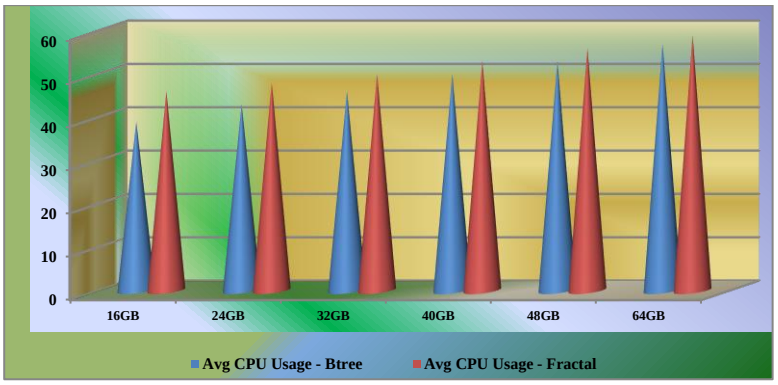
Graph 46: ETCD BTree Vs Fractal Tree-5.2

Graph 46 shows the CPU usage difference between BTRee implementation and Fractal Tree implementation. Since we are using branching strategy in the Tree data structure, CPU usage will be increased. CPU usage is going high since we are dealing with complexity in the implementation.



Graph 47: ETCD BTree Vs Fractal Tree-6.1

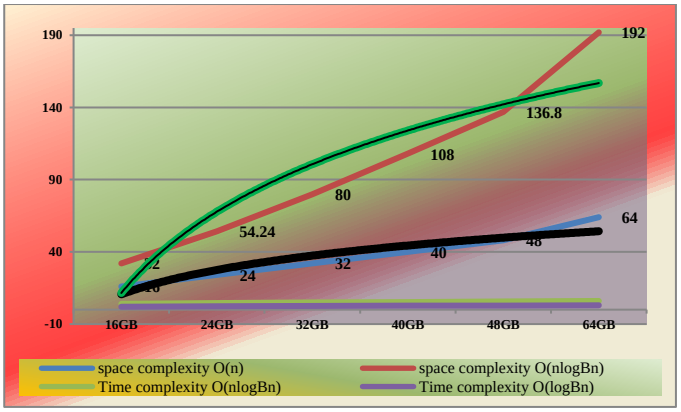
Graph 47, is the comparison between BTree and Fractal Tree implementation of the key value store (ETCD) for the sixth sample. The graph shows the Avg Insertion time difference between BTree and Fractal Tree implementation. As per the graph the time trend is going down as move from BTree to Fractal Tree implementation. The same observation we can have with other parameters like avg deletion time and avg search time.



Graph 48: ETCD BTree Vs Fractal Tree-6.2

Graph 48 shows the CPU usage difference between BTRee implementation and Fractal Tree implementation. Since we are using branching strategy in the Tree data structure, CPU usage will be

increased. CPU usage is going high since we are dealing with complexity in the implementation. We will address the resolution of this issue in future work.



Graph 49: ETCD BTree Vs Fractal Tree-Complexities

Graph 49 shows the comparison of complexities between BTree and Fractal Tree implementation. Fractal Tree implementation complexities are going little bit high compared to BTree Implementation. This is acceptable since we are increasing the complexity in the architecture.

EVALUATION

The comparison of BTree implementation results with Fractal Tree implementation shows that later one exihibits high performance. We have collected the stats for different sizes of the Data Store size. The Data Sore capacities are 16GB, 24GB, 32GB , 40GB , 42GB and 64GB. For all these events the comparison of the same parameters are have been observed. As per the analysis carried out so far in this states that avg insertion time , abg deletion time, and search time are going down if u start using the implementation of the Data Store (ETCD) using the Fractal Tree instead of BTree.

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 and tested the performance of ETCD operartions using the metrics collection code. We have collected six samples on etcd operations like insetion , deletion , search . All these activities are performing better in the Fractal Tree implementation compared to BTree implementation. Space complexity and time complexity are also compared, since we are increasing the complexity in the architecture , space and time complexity will it increased automatically. Along with this CPU usage also will get increased.

Future work includes working on the CPU usage to make it control while we are availing the facilities of the Fractal Tree implementation of the ETCD.

REFERENCES

1. A Comprehensive Study of “etcd”—An Open-Source Distributed Key-Value Store with Relevant Distributed Databases, April 2022,Emerging Technologies for Computing, Communication and Smart Cities (pp.481-489),Husen Saifibhai Nalawala, Jaymin Shah, Smita Agrawal, Parita Oza.
2. Impact of etcd deployment on Kubernetes, Istio, and application performance,William Tärneberg, Cristian Klein, Erik Elmroth, Maria Kihl, 07 August 2020.
3. Kuberenets in action by Marko Liksa , 2018.
4. Kubernetes Patterns, Ibryam , Hub
5. 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.
6. Kubernetes Best Practices , Burns, Villaibha, Strebel , Evenson.
7. Learning Core DNS, Belamanic, Liu.
8. Core Kubernetes , Jay Vyas , Chris Love.
9. A Formal Model of the Kubernetes Container Framework. GianlucaTurin, AndreaBorgarelli, SimoneDonetti, EinarBrochJohnsen, S.LizethTapiaTarifa, FerruccioDamiani Researchreport496,June202
10. Kubernetes Container Orchestration as a Framework for Flexible and Effective Scientific Data Analysis, IEEE Xplore, 13 February 2020.
11. On the Performance of etcd in Containerized Environments" by Luca Zanetti et al. (2020), IEEE International Conference on Cloud Computing (CLOUD).
12. 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.
13. Study on the Kubernetes cluster mocel, Sourabh Vials Pilande. International Journal of Science and Research , ISSN : 2319-7064.
14. Network Policies in Kubernetes: Performance Evaluation and Security Analysis, Gerald Budigiri; Christoph Baumann; Jan Tobias Mühlberg; Eddy Truyen; Wouter Joosen, IEEE Xplore 28 July 2021.
15. Networking Analysis and Performance Comparison of Kubernetes CNI Plugins, 28 October 2020, pp 99–109, Ritik Kumar & Munesh Chandra Trivedi.
16. Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability, Shixiong Qi; Sameer G. Kulkarni; K. K. Ramakrishnan, 25 December 2020 , IEEEXplore.
17. 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
18. Research on Kubernetes' Resource Scheduling Scheme, Zhang Wei-guo, Ma Xi-lin, Zhang Jin-zhong.
19. Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned, Leila Abdollahi Vayghan Montreal, Mohamed Aymen Saied; Maria Toeroe; Ferhat Khendek, IEEE XPlore.
20. Improving Application availability with Pod Readiness Gates https://orielly.ly/h_WiG
21. Kubernetes Best Practices: Resource Requests and limits https://orielly.ly/8bKD5
22. Configure Default Memory Requests and Limits for a Namespace https://orielly.ly/ozlUi1
23. Kubernetes CSI Driver for mounting images https://orielly.ly/OMqRo
24. Modelling performance & resource management in kubernetes by Víctor Medel, Omer F. Rana, José Ángel Bañares, Unai Arronategui.
25. "etcd: A Distributed, Reliable Key-Value Store for the Edge" by Corey Olsen et al. (2018)
26. "An Empirical Study of etcd's Performance and Scalability" by Zhen Xiao et al. (2019) 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS).
27. Distributed Kubernetes Metrics Aggregation, 23 September 2022, pp 695–703, Mrinal Kothari, Parth Rastogi, Utkarsh Srivastava, Akanksha Kochhar & Moolchand Sharma, Springer.
28. 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.
29. A Portable Load Balancer for Kubernetes Cluster, 28 January 2018, Kimitoshi Takahashi, Kento Aida, Tomoya Tanjo, Jingtao SunAuthors Info & Claims.
30. "etcd: A Highly-Available, Distributed Key-Value Store" by Brandon Philips et al. (2014), Proceedings of the 2014 ACM SIGOPS Symposium on Cloud Computing.
31. 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.

32. Performance Evaluation of etcd in Distributed Systems" by Jiahao Chen et al. (2020), 2020 IEEE International Conference on Cloud Computing (CLOUD).
33. Rearchitecting Kubernetes for the Edge, Andrew Jeffery, Heidi Howard, Richard MortierAuthors Info & Claims, 26 April 2021.
34. A Two-Tier Storage Interface for Low-Latency Kubernetes Deployments, Ionita, Teodor Alexandru, 2022-05-11.
35. Scalable Data Plane Caching for Kubernetes, Stefanos Sagkriotis; Dimitrios Pezaros, 2022, IEEE Xplore.
36. High Availability Storage Server with Kubernetes, Ali Akbar Khatami; Yudha Purwanto; Muhammad Faris Ruriawan, 2020, IEEE Xplore.
37. Management of Life Cycle of Computing Agents with Non-deterministic Lifetime in a Kubernetes Cluster, Mykola Alieksieiev; Volodymyr Smahliuk, 2023 , IEEE Xplore.
38. SECURITY IN THE KUBERNETES PLATFORM: SECURITY CONSIDERATIONS AND ANALYSIS, Ghadir Darwesh, Jafar Hammoud, Alisa Andreevna VOROBYOVA, 2022.
39. Security Challenges and Solutions in Kubernetes Container Orchestration, Oluebube Princess Egbuna, 2022.
40. The Implementation of a Cloud-Edge Computing Architecture Using OpenStack and Kubernetes for Air Quality Monitoring Application, Endah Kristiani, Chao-Tung Yang, Chin-Yin Huang, Yuan-Ting Wang & Po-Cheng Ko , 16 July 2020.