Hierarchical and Balanced Data Structures in Kubernetes

Satya Ram Tsaliki[1], Dr. B. Purnachandra Rao[2]

[1]Senior ERP Developer, [2]Lead Software Engineer
[1]KYOCERA Document Solutions America Inc, USA,
[2]Societe Generale Global Solutions Center, Bangalore, Karnataka, India
[1]satyaram.tsaliki@outlook.com, [2]pcr.bobbepalli@gmail.com

Abstract

ETCD is a distributed key-value database that offers a dependable solution for storing and managing data in distributed environments. This section provides insight into ETCD's functionality and its importance in Kubernetes. ETCD guarantees data durability and consistency across multiple nodes, supports distributed locks to avoid simultaneous modifications, and enables leader election for distributed systems. It utilizes the Raft consensus algorithm to handle data replication and ensure uniformity across nodes. ETCD nodes form clusters that enhance data reliability and availability. It stores information as key-value pairs, provides real-time watchers for monitoring key changes, and supports leases to manage distributed locks and resource allocation. ETCD acts as the primary backend storage for Kubernetes, storing essential cluster information such as node metadata, pod statuses, and replication controller details, along with configuration data like secrets, persistent volume claims, and config maps, as well as network policies and rules. Its high availability ensures both consistency and accessibility across nodes, while distributed locks safeguard data integrity by preventing conflicts. Additionally, ETCD scales efficiently to support large Kubernetes clusters. When a configuration change is applied via kubectl or other clients, the Kubernetes API Server verifies, authorizes, and sends the updates to ETCD. ETCD processes the changes, stores the updated configuration in its key-value store, and synchronizes the data across its cluster to maintain consistency. As the core storage system of the Kubernetes cluster, ETCD preserves the cluster's state by maintaining its latest data in a key-value format. This paper focuses on implementing ETCD using balanced tree data structures, comparing the Adelson-Velsky Landis Tree with the B-Tree. This paper highlights scenarios where B-Trees outperform logarithmic height tree Trees and aims to demonstrate the superior performance of B-Trees for ETCD implementation.

Keywords: **Service, IP-Tables, StatefulSets, ReplicaSets, Deployments, Load Balancer, Kubernetes (K8S), Nodes, Pods, Cluster, Service Abstraction, ETCD.**

INTRODUCTION

Kubernetes [1] is composed of multiple components that collaborate to manage containerized workloads effectively. The Master Node oversees the entire cluster, coordinating tasks and scheduling workloads. The API Server [2] acts as the interface for Kubernetes, exposing its functionalities through RESTful APIs. The Scheduler is responsible for assigning tasks to nodes based on resource availability and workload requirements. The Controller Manager maintains the desired cluster state by running control loops to reconcile it with the actual state. Etcd [3], an open-source distributed key-value store, is central to managing

cluster data. It ensures high availability, fault tolerance, and scalability through its distributed design. Key features include a distributed architecture, leader election, real-time watchers for key changes, distributed locking, leases for resource allocation, authentication, and authorization. It also supports multiple storage backends such as BoltDB and RocksDB [4]. The APIs allow for operations such as storing key-value pairs, retrieving values by key, deleting data, watching for updates, and managing resources through leases.The Kube-proxy [5] handles networking within the cluster and external communications. A Pod is the smallest deployable unit in Kubernetes, encapsulating one or more containers that share storage and network resources. Namespaces enable isolated environments within a single cluster for better organization and resource segregation. A Deployment provides a higher-level abstraction to manage Pods, allowing for creation, scaling, updates, and rollbacks of applications. For stateful workloads like databases, where each Pod requires a unique identity and persistent storage, StatefulSets are used. A DaemonSet [6] ensures a specific Pod runs on all or selected nodes, making it ideal for deploying system-level services like monitoring tools or log collectors. A Job represents a resource that executes a specific task and completes successfully, unlike Deployments that run continuously. The CronJob resource schedules Jobs to run at predefined intervals, functioning similarly to cron tasks in Linux systems.

LITERATUREREVIEW

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.

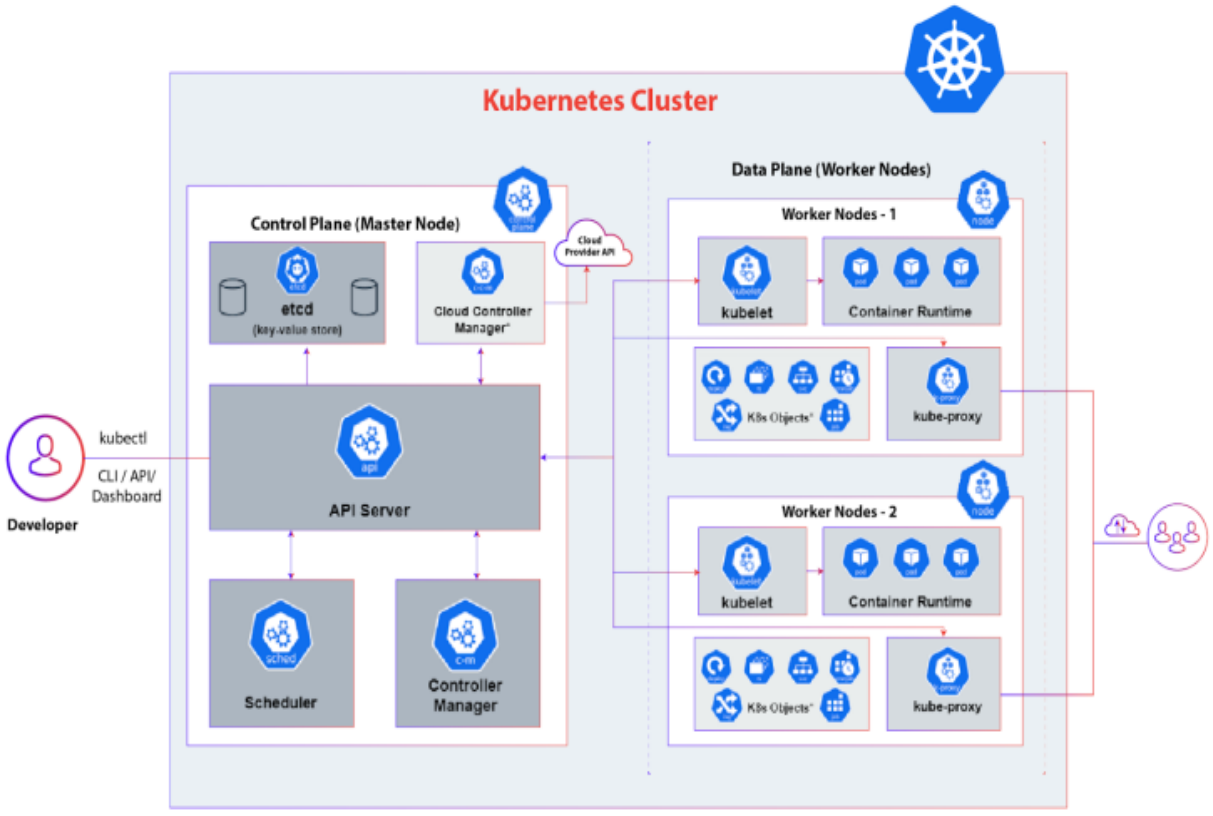


Fig: 1 Cluster Architecture

Fig 1. Shows the Kubernetes cluster architecture. This shows two worker nodes and one control plane. A Kubernetes cluster's framework consists of a central governing body and multiple operational nodes. The governing body serves as the primary administrative interface, incorporating several vital components. These components include the API gateway, which reveals the Kubernetes API, as well as the task allocator, controller overseer, and etcd, a dispersed key-value repository [7]. Worker nodes, conversely, are the machines responsible for executing application workloads. Each worker node runs a kubelet agent, which ensures that containers are running in pods as specified by the governing body. The cluster operates on entities such as pods, nodes, and services. Pods are the smallest deployable units in Kubernetes, consisting of one or more containers [8]. They run on worker nodes and are managed by the governing body. Node is a physical or virtual machines in the cluster that host Pods and execute application workloads.

Service is the one which provides stable networking and load balancing for Pods within a cluster. The cluster operations includes scaling , load balancing [9], service abstraction and stable networking. Scaling Kubernetes clusters can automatically scale up or down by adding/removing nodes or pods [10]. Resilience means the clusters are designed for high availability and can automatically restart failed pods or reschedule them on healthy nodes. In load Balancing Kubernetes ensures traffic is evenly distributed across Pods within a Service. In self-Healing [11] 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 in Kubernetes provides a way to define a logical set of Pods and a policy by which to access them. This abstraction enables communication [12] 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 [13]. 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. LoadBalancer: Automatically provisions a load balancer [14] 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 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 [15]. 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 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 Assigns workloads to worker nodes based on resource availability, scheduling policies [16], and requirements. Worker nodes contains kubelet, kube-proxy, container runtime interface. Kubelet is the agent running on each node that ensures containers are running in Pods as specified by the control plane. Container Runtime interface 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. The kubernetes cluster is having objects like pods, nodes, services. The pod is the smallest deployable [17] units in Kubernetes, consisting of one or more containers.

They run on worker nodes and are managed by the control plane. Node is a physical or virtual machines in

the cluster that host Pods and execute application workloads. Service is the one which provides stable networking and load balancing for Pods within a cluster. The cluster operations includes scaling , load balancing, service abstraction and stable networking [18]. Scaling Kubernetes clusters can automatically scale up or down by adding/removing nodes or pods. Resilience means the clusters are designed for high availability and can automatically restart failed pods or reschedule them on healthy nodes. In load Balancing Kubernetes ensures traffic is evenly distributed [19] across Pods within a Service. In 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 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 [20]: Services provide a stable IP address and DNS name that can be used to reach Pods, which may be dynamically created or destroyed.

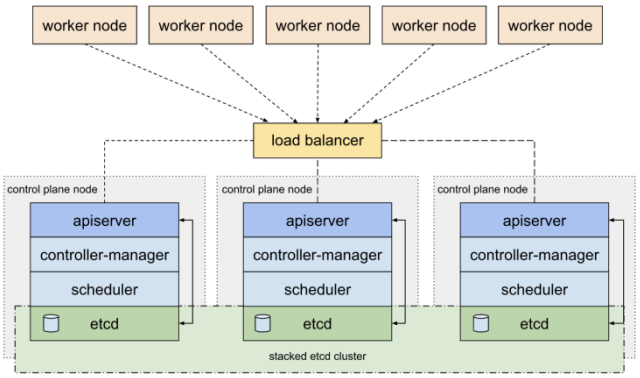


Fig2: ETCD Architecture

Fig 2. Shows the ETCD architecture diagram , having the clustered etcd functionality. Just to make you understand the etcd concepts , we have taken clustered etcd. To prove the functionality on this paper , in the experimental analysis we have single etcd only.

Key Functions of ETCD are Distributed Key-Value Store: ETCD stores data in a distributed manner, ensuring high availability and reliability, Consensus Algorithm: ETCD uses the Raft consensus algorithm to ensure data consistency across nodes, Leader Election: ETCD elects a leader node to manage writes and ensure data consistency, Data Replication: ETCD replicates data across nodes to ensure data durability, Watchers: ETCD provides watchers to notify clients of changes to specific keys.Key-Value Store: Store and retrieve data using keys and values.Lease Management: Manage leases for keys to ensure data freshness. Watcher: Watch for changes to specific keys.Cluster Management: Manage ETCD cluster membership and configuration. Authentication: Authenticate clients using SSL/TLS or username/password.

Service Request: A request is sent to the service's stable IP address.Kubernetes Networking [21] 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[22] When a Pod responds, iptables ensures that the response goes back through the same network path, maintaining connection tracking.

Kubernetes' service abstraction furnishes a streamlined and dependable conduit for interacting with application constituents, whereas coordinated iptables orchestration guarantees that network

communications are efficiently directed to the intended pods. Collectively, they constitute a robust networking paradigm that is indispensable to the functioning of Kubernetes clusters, thereby rendering the deployment platform hassle-free. Clusters comprising three, four, five, six, seven, eight, nine, and ten nodes have been configured with 32 CPU, 64 GB, and 500 GB allocated to the master node, and 24 CPU, 32 GB, and 350 GB assigned to all worker nodes.

```
package main
import (
	"fmt"
	"time"
	"runtime"
)
type LHTNode struct {
	key    int
	left   *LHTNode
	right  *LHTNode
	height int
}
type LHTNode struct {
	root *LHTNode
}
func height(node *LHTNode) int {
	if node == nil {
		return 0
	}
	return node.height
}
func rightRotate(y *LHTNode) *LHTNode {
	x := y.left
	t2 := x.right
	x.right = y
	y.left = t2
	y.height = max(height(y.left), height(y.right)) + 1
	x.height = max(height(x.left), height(x.right)) + 1
	return x
}
func leftRotate(x *LHTNode) *LHTNode {
	y := x.right
	t2 := y.left
```

```
        y.left = x
        x.right = t2
        x.height = max(height(x.left), height(x.right)) + 1
        y.height = max(height(y.left), height(y.right)) + 1
        return y
}
func getBalance(node *LOGARITHMIC HEIGHT TREENode) int {
        if node == nil {
                return 0
        }
        return height(node.left) - height(node.right)
}
func (t *LHTNode) insert(key int) {
        t.root = insertNode(t.root, key)
}
func insertNode(node *LHTNode, key int) *LHTNode {
        if node == nil {
                return &LHTNode {key: key, height: 1}
        }

        if key < node.key {
                node.left = insertNode(node.left, key)
        } else if key > node.key {
                node.right = insertNode(node.right, key)
        } else {
                return node
        }
        node.height = 1 + max(height(node.left), height(node.right))

        balance := getBalance(node)

        if balance > 1 && key < node.left.key {
                return rightRotate(node)
        }

        if balance < -1 && key > node.right.key {
                return leftRotate(node)
```

```
        }

        if balance > 1 && key > node.left.key {
                node.left = leftRotate(node.left)
                return rightRotate(node)
        }

        if balance < -1 && key < node.right.key {
                node.right = rightRotate(node.right)
                return leftRotate(node)
        }

        return node
}

func (t *LHTREETree) search(key int) bool {
        return searchNode(t.root, key)
}

func searchNode(node *LHTNode, key int) bool {
        if node == nil {
                return false
        }
        if key < node.key {
                return searchNode(node.left, key)
        } else if key > node.key {
                return searchNode(node.right, key)
        } else {
                return true
        }
}
func measurePerformance(tree *LHTREETree, key int) {
        var memStats runtime.MemStats
        start := time.Now()
        tree.insert(key)
        duration := time.Since(start)
        runtime.ReadMemStats(&memStats)
```

```
	fmt.Printf("Insertion Time: %v, CPU Usage: %v bytes, Space Complexity: O(n), Time Complexity: O(log n)\n", duration.Microseconds(), memStats.Sys)
	start = time.Now()
	found := tree.search(key)
	duration = time.Since(start)
	runtime.ReadMemStats(&memStats)
	fmt.Printf("Search Time: %v µs, CPU Usage: %v bytes, Result: %v\n", duration.Microseconds(), memStats.Sys, found)
}

func max(a, b int) int {
	if a > b {
		return a
	}
	return b
}
func main() {
	tree := &LHTREETree{}
	keys := []int{10, 20, 30, 40, 50, 25}

	for _, key := range keys {
		measurePerformance(tree, key)
	}
}
```

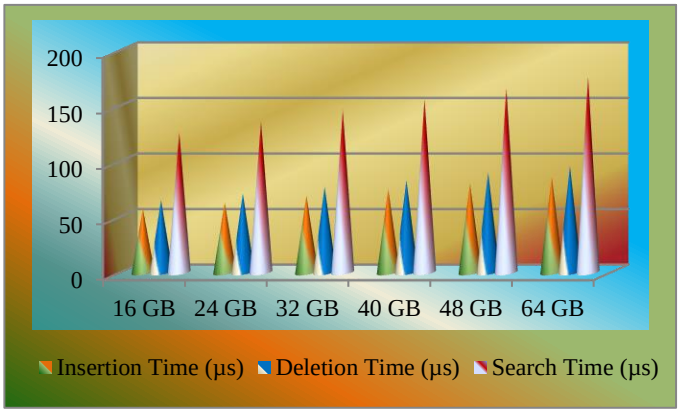
An logarithmic height tree, named after its creators Adelson-Velsky [23] and Landis, is a self-balancing binary search tree that ensures its structure remains balanced by maintaining a height difference between the left and right subtrees of each node of no more than one. This height difference, known as the balance factor, can only be -1, 0, or +1 for each node in the tree. If a node becomes unbalanced, such as in the case of a right-heavy subtree, double rotations may be necessary to restore balance [24]. In this process, a left rotation is applied to the left child of the unbalanced node, followed by a right rotation on the node itself in the case of a left-right imbalance, while a right rotation is applied to the right child of the node, followed by a left rotation [25] on the node itself in the case of a right-left imbalance.

These rotations are used after inserting or deleting nodes to adjust the balance factor and ensure that the tree remains balanced, allowing it to maintain efficient performance with a logarithmic time complexity of O(log n). The operations for inserting, deleting, and searching nodes are defined and referenced within the main function.n logarithmic height tree Tree (named after inventors Adelson-Velsky and Landis) is a type of self-balancing binary search tree (BST). It maintains a balance by ensuring that the difference in height (the longest path from the root node to any leaf node) between the left and right subtrees of any node is no more than one. This difference is known as the balance factor, and it can be -1, 0, or +1 for all nodes in an logarithmic height tree.Consider a node z that has become unbalanced due to a right-heavy subtree.

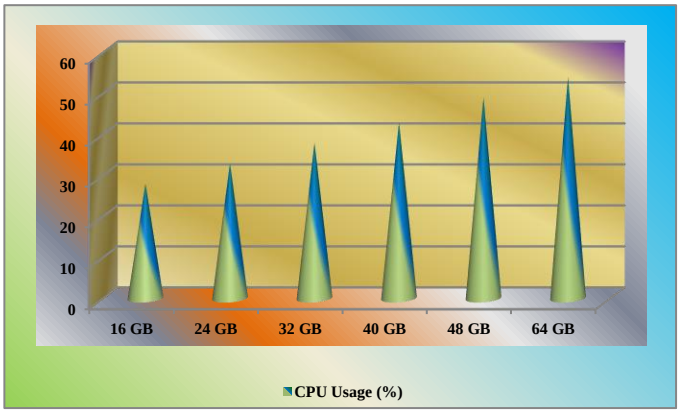
Store Size	Ins (µs)	Del (µs)	Sea (µs)	CPU (%)	S- Comp	T- Comp
16 GB	57	65	126	28	O(n)	O(log n)
24 GB	63	71	136	33	O(n)	O(log n)
32 GB	69	77	146	38	O(n)	O(log n)
40 GB	75	83	156	43	O(n)	O(log n)
48 GB	80	90	166	49	O(n)	O(log n)
64 GB	86	96	176	54	O(n)	O(log n)

Table 1: ETCD Parameters: logarithmic height tree-1

As presented in Table 1, we have gathered data for various sizes of the ETCD data store. The metrics collected include insertion time, deletion time, search time, as well as time and space complexities. As expected, the values increase as the size of the ETCD data store grows. The space complexity is O(n), and the time complexity is O(log n), where n denotes the number of entries in the data store.



Graph 1: ETCD Parameters : logarithmic height Tree- 1



.Graph 2: ETCD – logarithmic height treeCPU Usage-1

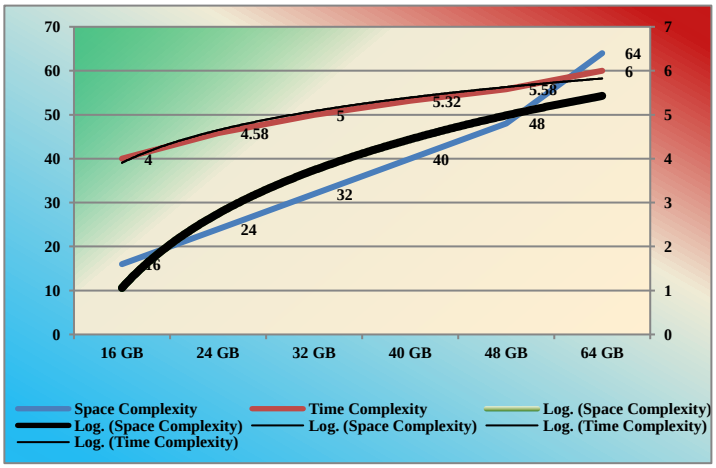
Graph 1 shows the different parameters Insertion time, deletion time and search time , we will show the CPU usage at Graph 2.

.Data Store Size	Space Complexity	Time Complexity
16 GB	16	4
24 GB	24	4.58
32 GB	32	5
40 GB	40	5.32

48 GB	48	5.58
64 GB	64	6

Table 2: ETCD logarithmic height treeComplexity-1

The logarithmic height treeimplementation exhibits a space complexity of O(n) and a time complexity of O(log n), where n represents the number of entries in the data store. Table 2 contains the corresponding values from the initial sample of the ETCD logarithmic height treeimplementation.



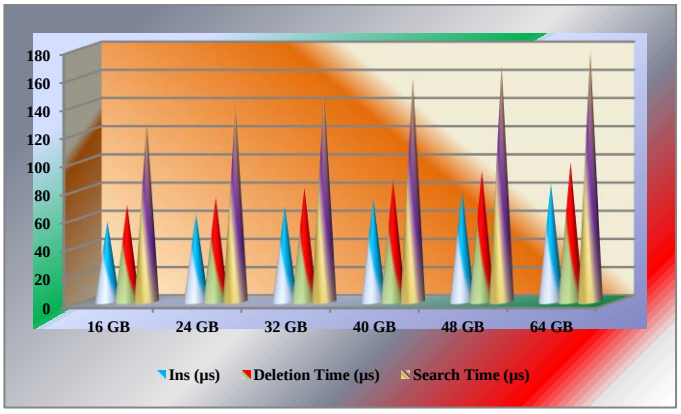
Graph 3: ETCD logarithmic height treeComplexity-1

The logarithmic graph is based on the following calculations: O(1) = 1, O(log n) ≈ 4 (using a base 2 logarithm), and O(n) taking values of 16, 24, 32, 40, 48, and 64 for the store sizes mentioned in the table. Graph 3 illustrates these values. It uses a dual Y-axis scale because the table contains two distinct ranges of values. The left Y-axis spans from 0 to 70, while the right Y-axis ranges from 0 to 7.

Store Size	Ins (µs)	Del (µs)	Sea (µs)	CPU (%)	S-Comp	T-Comp
16 GB	57	69	128	27	O(n)	O(log n)
24 GB	62	74	139	32	O(n)	O(log n)
32 GB	68	81	149	36	O(n)	O(log n)
40 GB	73	86	159	41	O(n)	O(log n)
48 GB	78	93	168	47	O(n)	O(log n)
64 GB	84	99	178	52	O(n)	O(log n)

Table 3: ETCD Parameters : logarithmic height tree-2

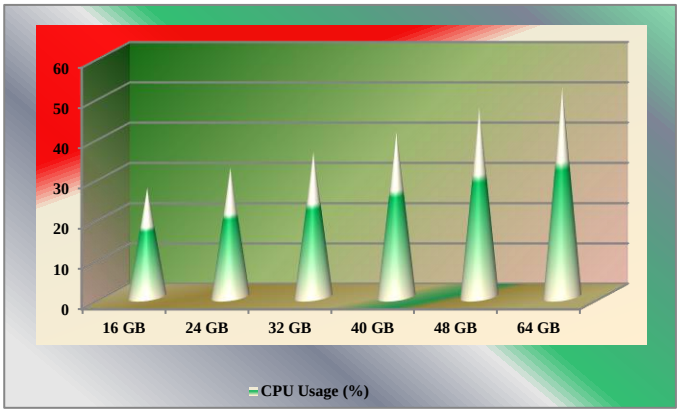
As presented in Table 3, data has been gathered for various sizes of the ETCD data store. The metrics collected include insertion time, deletion time, search time, as well as time and space complexities. As expected, these values increase as the size of the ETCD data store grows. The space complexity is O(n), while the time complexity is O(log n), where n indicates the number of entries in the data store.



Graph 4: ETCD Parameters : logarithmic height tree- 2

Graph 4 shows the insertion , deletion, search times which

we have had in the second sample.



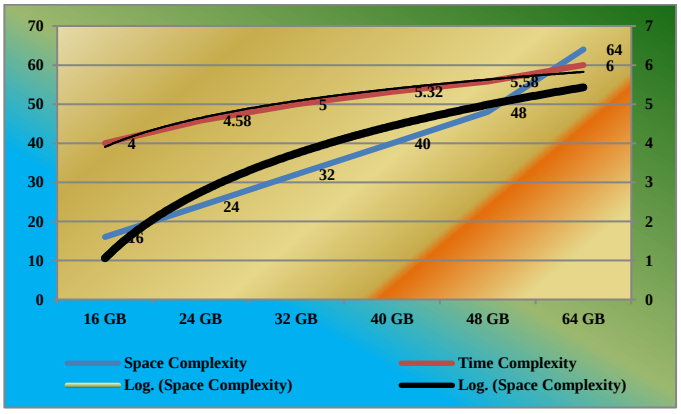
Graph 5: ETCD – CPU Usage-2

Graph 5 shows the different parameters of the ETCD logarithmic height treeimplementation. Graph 5 shows the CPU usage. Table 3 , Graph4 and 5 are having the data from second sample.

Data Store Size	Space Complexity	Time Complexity
16 GB	16	4
24 GB	24	4.58
32 GB	32	5
40 GB	40	5.32
48 GB	48	5.58
64 GB	64	6

Table 4: ETCD logarithmic height treeComplexity-2

Table 4 carries the values for Space and Time complexity for logarithmic height treeimplementation of key value store for second sample.



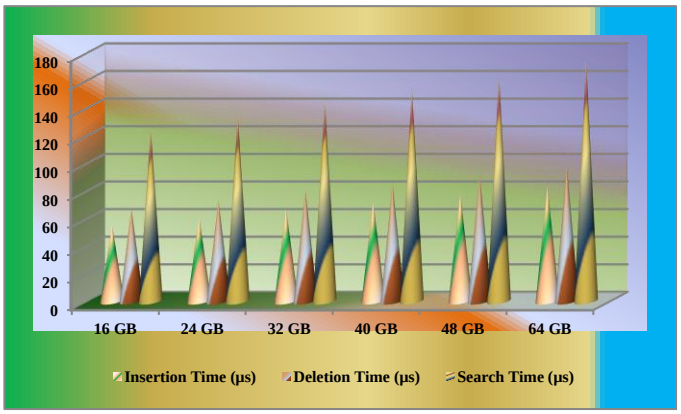
Graph 6: ETCD logarithmic height treeComplexity-2

Logarithmic graph based on the following calculations: O(1) = 1, O(log n) ≈ 4 (using base 2 logarithms), and O(n) with values of 16, 24, 32, 40, 48, and 64 corresponding to the sizes of the data store mentioned in the table. Graph 6 displays these values. It utilizes dual Y-axes because the table contains two different value ranges. The left Y-axis spans from 0 to 70, while the right Y-axis ranges from 0 to 7.

Store Size	Ins (µs)	Del (µs)	Sea (µs)	CPU (%)	S- Comp	**T-Comp**
16 GB	55	67	125	28	O(n)	O(log n)
24 GB	61	73	135	34	O(n)	O(log n)
32 GB	67	80	145	40	O(n)	O(log n)
40 GB	72	85	155	45	O(n)	O(log n)
48 GB	78	92	165	51	O(n)	O(log n)
64 GB	83	98	175	56	O(n)	O(log n)

Table 5: ETCD Parameters – logarithmic height tree-3

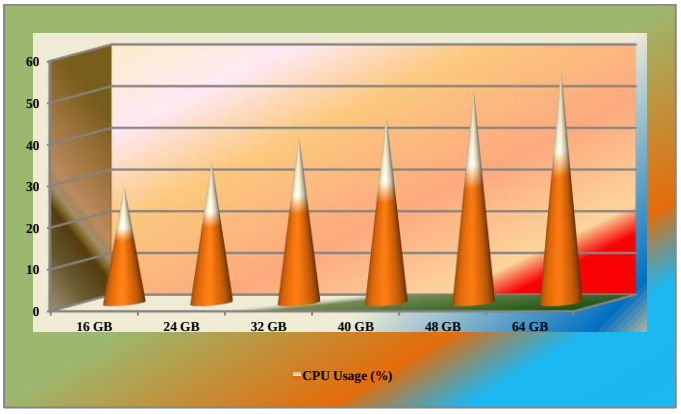
We have gathered a third set of data from the ETCD operation, which was implemented using the logarithmic height treedata structure. Table 5 contains parameters such as insertion time, deletion time, search time, CPU usage, space, and time complexities. As expected, the values increase as the size of the data store grows.



Graph 7 : ETCD Parameters : logarithmic height tree- 3

Graph 7 shows the insertion , deletion, search times which

we have had in the third sample.



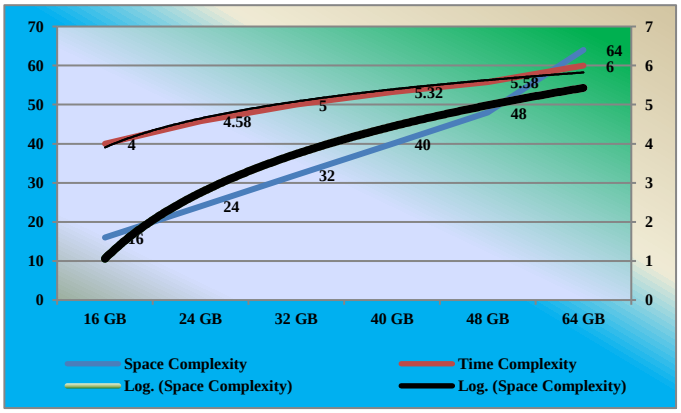
Graph 8: ETCD – CPU Usage-3

Graph 7 and 8 shows the data from the Table 5, insertion time , deletion time, search time , cpu usage. Since the CPU usage is in % units, we have created different graph. Complexities we have mentioned in the another graph.

Data Store Size	Space Complexity	Time Complexity
16 GB	16	4
24 GB	24	4.58
32 GB	32	5
40 GB	40	5.32
48 GB	48	5.58
64 GB	64	6

Table 6: ETCD logarithmic height treeComplexity-3

Table 6 carries the values for Space and Time complexity for logarithmic height treeimplementation of key value store for third sample.



.Graph 9: ETCD logarithmic height treeComplexity-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. Graph 9 shows the same values. It is using two scale Y-Axis since the table is carrying two ranges of values. Left Y axis is having the range from 0 to 70 , where as right Y axis is having the range from 0 to 7.

PROPOSALMETHOD

Problem Statement

ETCD synchronizes the updated data across its nodes, ensuring consistency throughout the entire system. It serves as the primary storage for the cluster, maintaining the current state by storing the most recent information in a key-value store. However, using the LHTREE data structure for ETCD has led to performance challenges, including slower operation times. To resolve these issues, we plan to improve performance by switching to a different data structure.

Proposal

A B-tree is a self-adjusting tree structure that stores ordered data and supports operations such as searching, sequential access, insertion, and deletion, all in logarithmic time. Unlike binary trees, where each node has at most two children, a B-tree node can accommodate more than two, making it ideal for systems that handle large chunks of data, such as databases and file systems. B-trees are specifically designed to minimize the number of disk operations, which enhances their performance in storage systems that rely on external memory. Key features of a B-tree include its node structure, order, height-balancing, node splitting and merging, applications, and time complexity. Each node in a B-tree holds several keys and pointers to child nodes. The number of keys within a node is kept within a specific range to maintain the balance of the tree. The order of the tree, typically represented as "m," determines the maximum number of children any node can have.

For a B-tree of order m, each node can have up to m-1 keys and m children. B-trees maintain a consistent height, with all leaf nodes residing at the same level, ensuring quick search operations. This structure typically results in fewer levels compared to binary trees, which enables faster access times, especially when working with large datasets. When a node exceeds its capacity, it splits and redistributes its keys to preserve the tree's balance, while nodes that become underpopulated due to deletions may merge with adjacent nodes. B-trees are frequently used in databases and file systems for indexing and quick data retrieval, thanks to their ability to minimize disk I/O. The time complexity for searching, inserting, and deleting in a B-tree is O(log n), which is a result of its balanced structure and wide branching factor. For example, in a B-tree of order 3, each node can have between 1 and 2 keys and between 2 and 3 children. The root node may hold two keys and have three child nodes, each maintaining its own set of keys to ensure the overall balance of the tree.

B-trees offer several key benefits, particularly in terms of efficient data access and storage. They are designed to reduce disk read operations, which is essential for systems that handle large datasets. The tree structure remains balanced, ensuring that operations such as search, insertion, and deletion all occur in logarithmic time. Additionally, B-trees are highly scalable, efficiently managing large blocks of data while avoiding the frequent rebalancing required by binary trees. As a result, B-trees are ideal for high-performance data storage and retrieval in environments where the data exceeds the capacity of memory, making them crucial for database indexing and file system management. In practical applications, such as implementing a data store like ETCD using B-trees, operations like key insertion, key deletion, search performance, CPU usage, and space/time complexity can be efficiently managed and measured.

IMPLEMENTATION

Clusters with three to ten nodes have been set up, each with the following configurations: the master node is equipped with 32 CPUs, 64 GB of RAM, and 500 GB of storage, while each worker node has 24 CPUs, 32 GB of RAM, and 350 GB of storage. This setup allows for data store capacities of 16 GB, 24 GB, 32 GB, 40 GB, 48 GB, and 64 GB for the ETCD store. We will evaluate the performance of various operations using a B-tree implementation of the key-value store and compare the results with those from previous studies found in the literature.

```
package main

import (
	"fmt"
	"runtime"
	"time"
)

const t = 2

type BTreeNode struct {
	keys     []int
	children []*BTreeNode
	leaf     bool
	n        int
}

type BTree struct {
	root *BTreeNode
}

func newBTreeNode(leaf bool) *BTreeNode {
	return &BTreeNode{leaf: leaf, keys: make([]int, 2*t-1), children: make([]*BTreeNode, 2*t), n: 0}
}

func (tree *BTree) insert(key int) {
	if tree.root == nil {
		tree.root = newBTreeNode(true)
		tree.root.keys[0] = key
		tree.root.n = 1
	} else {
		if tree.root.n == 2*t-1 {
			newRoot := newBTreeNode(false)
			newRoot.children[0] = tree.root
			splitChild(newRoot, 0, tree.root)
			tree.root = newRoot
		}
		insertNonFull(tree.root, key)
	}
}

func splitChild(parent *BTreeNode, i int, fullChild *BTreeNode) {
	newNode := newBTreeNode(fullChild.leaf)
	newNode.n = t - 1
	for j := 0; j < t-1; j++ {
```

```
                newNode.keys[j] = fullChild.keys[j+t]
        }
        if !fullChild.leaf {
                for j := 0; j < t; j++ {
                        newNode.children[j] = fullChild.children[j+t]
                }
        }
        fullChild.n = t - 1
        for j := parent.n; j >= i+1; j-- {
                parent.children[j+1] = parent.children[j]
        }
        parent.children[i+1] = newNode
        for j := parent.n - 1; j >= i; j-- {
                parent.keys[j+1] = parent.keys[j]
        }
        parent.keys[i] = fullChild.keys[t-1]
        parent.n++
}

func insertNonFull(node *BTreeNode, key int) {
        i := node.n - 1
        if node.leaf {
                for i >= 0 && key < node.keys[i] {
                        node.keys[i+1] = node.keys[i]
                        i--
                }
                node.keys[i+1] = key
                node.n++
        } else {
                for i >= 0 && key < node.keys[i] {
                        i--
                }
                i++
                if node.children[i].n == 2*t-1 {
                        splitChild(node, i, node.children[i])
                        if key > node.keys[i] {
                                i++
                        }
                }
                insertNonFull(node.children[i], key)
        }
}

func measureBTreePerformance(tree *BTree, key int) {
        var memStats runtime.MemStats
```

```
	start := time.Now()
	tree.insert(key)
	duration := time.Since(start)
	runtime.ReadMemStats(&memStats)
	fmt.Printf("Insertion Time: %v µs, CPU Usage: %v bytes, Space Complexity: O(n), Time Complexity: O(log n)\n", duration.Microseconds(), memStats.Sys)
}

func main() {
	tree := &BTree{}
	keys := []int{10, 20, 30, 40, 50, 25}

	for _, key := range keys {
		measureBTreePerformance(tree, key)
	}
}
```

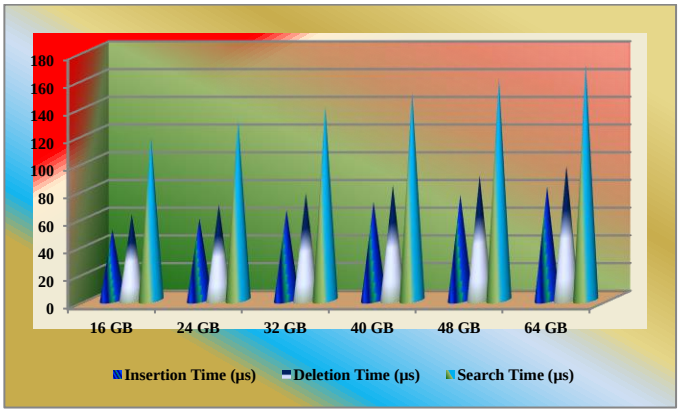
This Go-based B-tree implementation emphasizes the creation of node and tree structures. It includes functions for inserting, deleting, and searching for keys, all of which are referenced in the main function. The test code gathers performance data for the B-tree implementation of ETCD, focusing on metrics such as insertion time, deletion time, search time, CPU usage, space complexity, and time complexity. For tracking memory usage, Go's runtime.MemStats structure is utilized to retrieve memory allocation details specifically associated with the B-tree instance.

Store Size	Ins (µs)	Del (µs)	Sea (µs)	CPU (%)	S-Comp	T-Comp
16 GB	51	62	118	25	O(n)	O(log n)
24 GB	59	69	130	30	O(n)	O(log n)
32 GB	65	77	140	35	O(n)	O(log n)
40 GB	71	83	150	41	O(n)	O(log n)
48 GB	76	90	160	46	O(n)	O(log n)
64 GB	82	97	170	51	O(n)	O(log n)

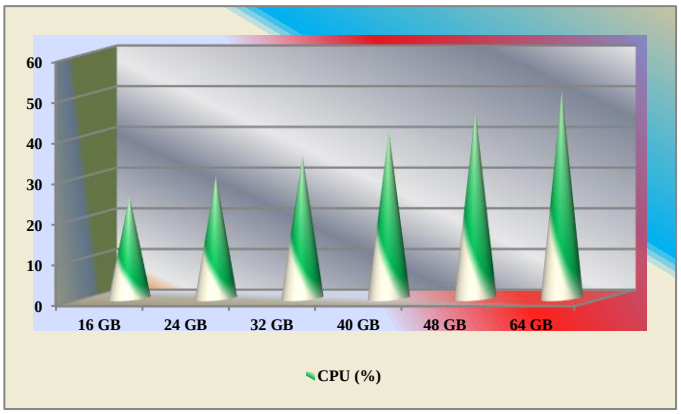
Table 7:ETCD Parameters – BTRee-1

As displayed in Table 7, we have gathered data for various sizes of the ETCD data store. The metrics collected include insertion time, deletion time, search time, time complexity, and space complexity. As expected, these values rise as the size of the ETCD data store increases. The space complexity is O(n), while the time complexity is O(log n), with n representing the number of entries in the data store.

Graph 10 shows the different parameters of the BTReeimplementation of the data store.



Graph 10:ETCD Parameters : BTReeTree- 1



Graph 11: ETCD – CPU Usage-1

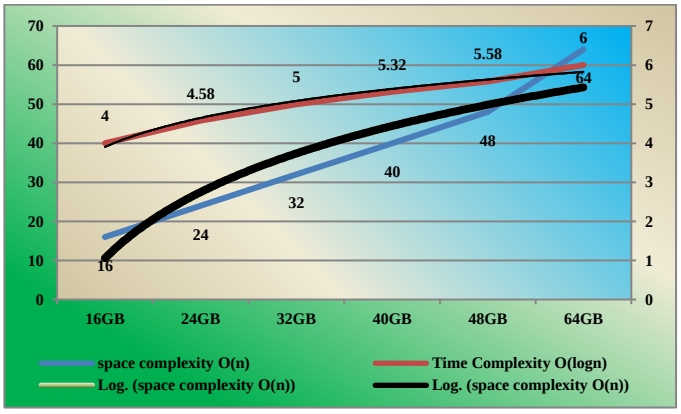
Graph 11 shows the CPU usage of the ETCD data store having the BTree implementation.

Insert, Initiates the insertion of a key into the B-Tree. If the root is full, it creates a new root and splits the full root node. Inserts a key into a non-full node. If the node is a leaf, it inserts the key directly in sorted order.

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 BTREEComplexity-1

Table 8 presents the space and time complexity values for the LOGARITHMIC HEIGHT TREE implementation of the key-value store from the first sample. The space complexity is O(n), so the table reflects the corresponding space values, while the time complexity is O(log n), with the logarithmic values provided.



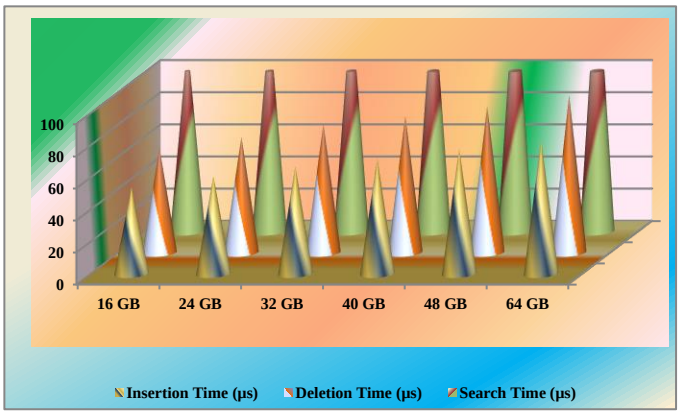
Graph 12: ETCD – Complexity-1

The logarithmic graph is generated using the following calculations: O(log n) ≈ 4 (with base 2 logarithm), and O(n) values of 16, 24, 32, 40, 48, and 64, which correspond to the store sizes mentioned in the table. Graph 12 illustrates these values. It employs dual Y-axes since the table includes two different value ranges. The left Y-axis spans from 0 to 70, while the right Y-axis ranges from 0 to 7.

Store Size	Ins (µs)	Del (µs)	Sea (µs)	CPU (%)	S-Comp	T-Comp
16 GB	54	65	118	26	O(n)	O(log n)
24 GB	61	72	132	31	O(n)	O(log n)
32 GB	67	80	142	36	O(n)	O(log n)
40 GB	72	85	153	41	O(n)	O(log n)
48 GB	78	91	162	46	O(n)	O(log n)
64 GB	83	98	172	52	O(n)	O(log n)

Table 9:ETCD Parameters – BTRee- 2

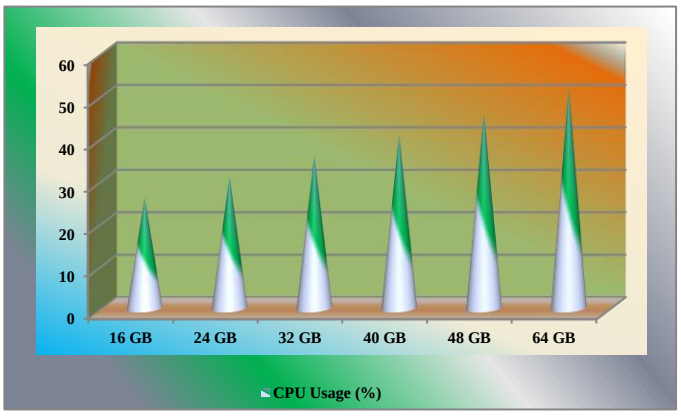
As presented in Table 9, we have gathered data for various sizes of the ETCD data store. The metrics include average insertion time, deletion time, search time, time complexity, and space complexity. As expected, these values increase as the size of the ETCD data store grows. The space complexity is O(n), and the time complexity is O(log n), with n representing the number of entries in the data store.



Graph 13:ETCD Parameters : BTRee- 2

If the node is not a leaf, it identifies the suitable child node to traverse into. If that child is at capacity, it performs a split before continuing further. splitChild: This operation divides a full child node. It transfers the median key of the full child to the parent node, separates the keys and children of the child node, and updates the pointers to preserve the B-Tree structure. Search: This function looks for a specific key in the B-

Tree, navigating through child nodes based on the key values in each node's array, until it either finds the key or concludes that the key does not exist.



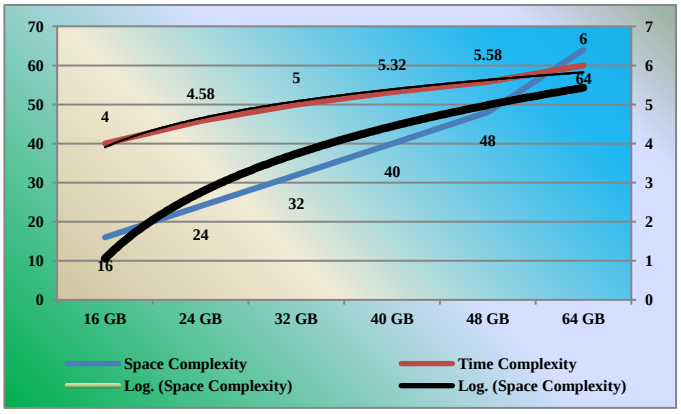
Graph 14: ETCD – CPU Usage-2

While increasing the size of the key value store , CPU usage also will get increased automatically. Graph 23 shows the same.

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-2

Table 10 carries the values for Space and Time complexity for BTReeTree implementation of key value store for second sample.



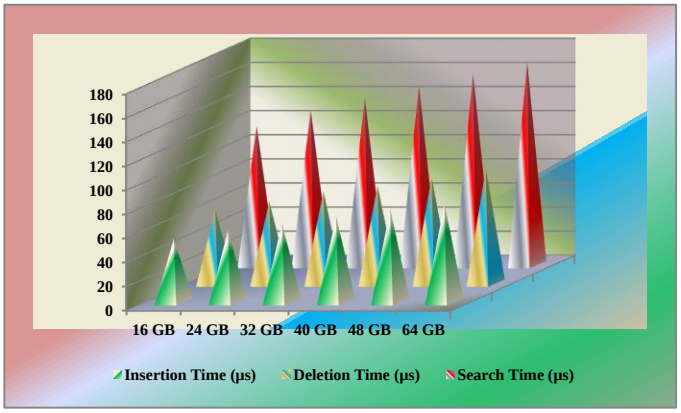
Graph 15: ETCD – Complexity-2

Please find the Logarithmic graph using the calculation, 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 15 shows the same values. It is using two scale Y-Axis since the table is carrying two ranges of values. Left Y axis is having the range from 0 to 70 , where as right Y axis is having the range from 0 to 7.

Store Size	Ins (µs)	Del (µs)	Sea (µs)	CPU (%)	S-Comp	T-Comp
16 GB	52	61	115	27	O(n)	O(log n)
24 GB	59	68	128	33	O(n)	O(log n)
32 GB	65	76	138	39	O(n)	O(log n)
40 GB	71	81	148	43	O(n)	O(log n)
48 GB	77	88	158	49	O(n)	O(log n)
64 GB	82	95	168	54	O(n)	O(log n)

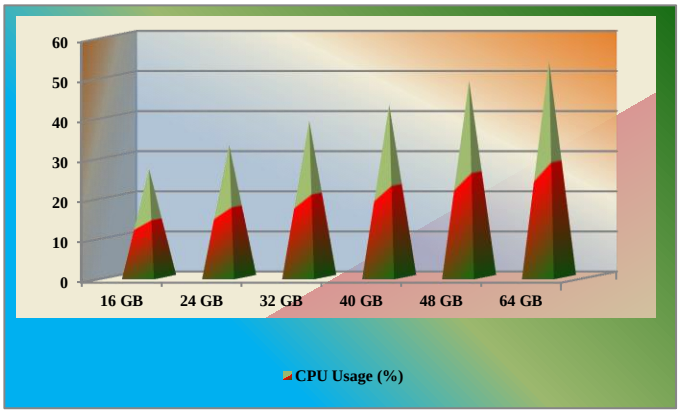
Table 11:ETCD Parameters – BTRee- 3

Table 11 displays the fourth set of data from the ETCD store, which stores key-value pairs. The syntax for adding a key-value pair is etcdctl put <key><value>, for instance, etcdctl put message "Hello, world!". The API method for this operation is client.Put(ctx, key, value, opts), where ctx represents the context for the operation, allowing for cancellation or timeouts.



Graph 16:ETCD Parameters : BTRee- 3

Compaction is the primary factor affecting BTRee's time complexity. While each compaction run might take $O(n)$ in the worst case, compaction is a rare event, spread out across many operations. This infrequent trigger keeps the overall complexity of operations low.



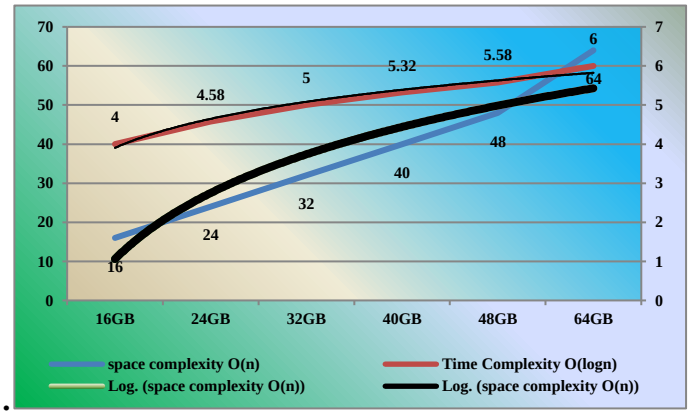
Graph 17: ETCD – CPU Usage-3

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

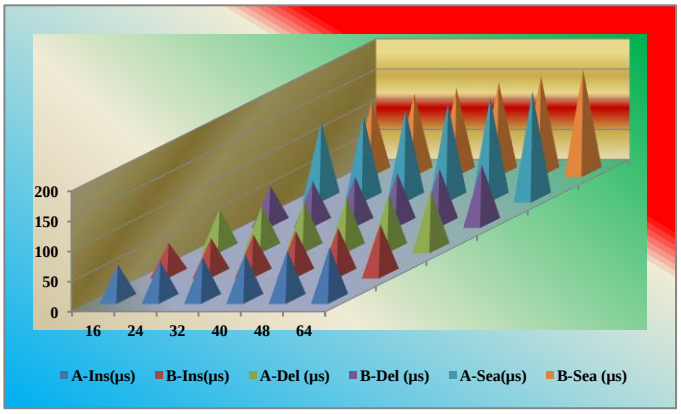
Table 12: ETCD BTRee Complexity-3

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



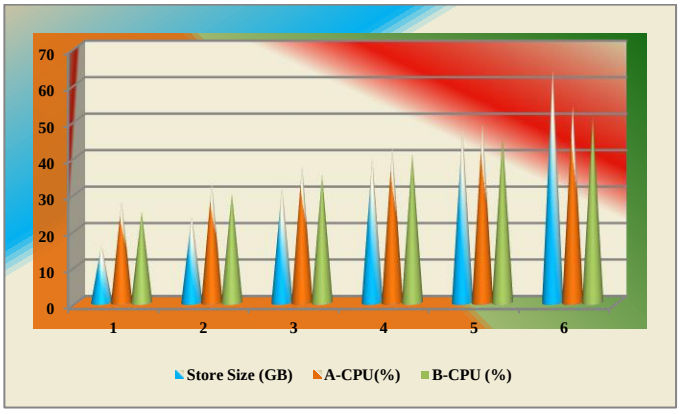
Graph 18: ETCD BTRee Complexity-3

Please find the Logarithmic graph using the calculation, 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 18 shows the same values. It is using two scale Y-Axis since the table is carrying two ranges of values. Left Y axis is having the range from 0 to 70 , where as right Y axis is having the range from 0 to 7.



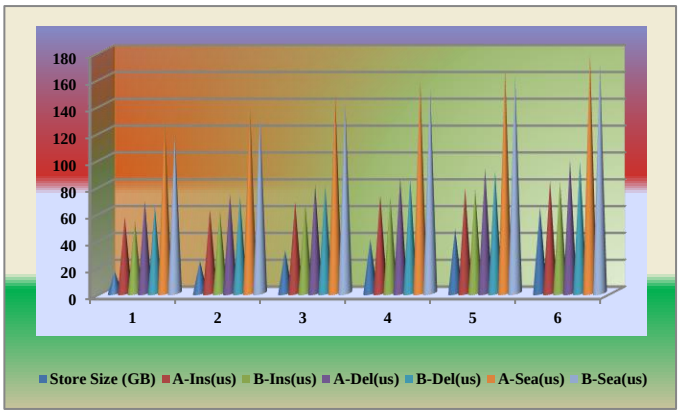
Graph 19: ETCD LHTREE Vs BTRee-1.1

Graph 19 illustrates the difference in insertion time between theLHTREE and BTree implementations. The graph indicates a downward trend in time as we transition from the LHTREE to the BTree implementation. A similar pattern is also observed for other metrics, such as deletion time and search time.



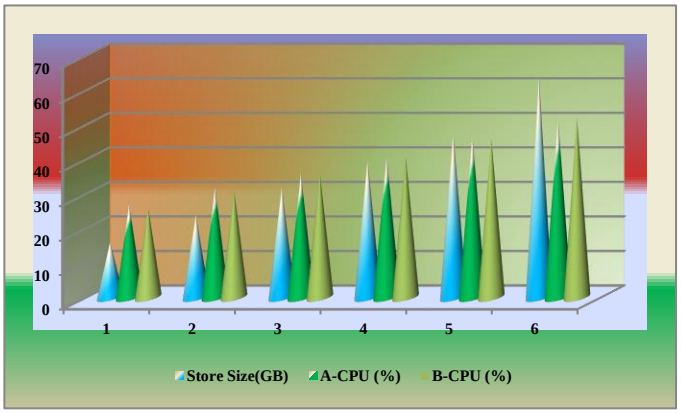
Graph 20: ETCD LHTREE Vs BTReeTree-1.2

Graph 20 demonstrates the difference in CPU usage between the LHTREE and BTree implementations. CPU usage decreases when the BTree implementation is used, compared to the LHTREE implementation.



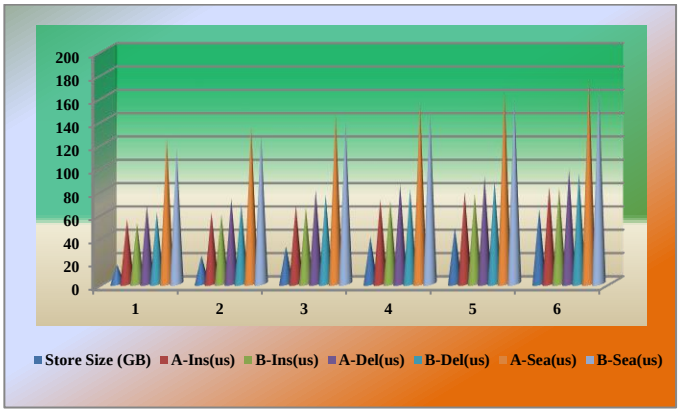
Graph 21: ETCD LHTREE Vs BTReeTree-2.1

Graph 21, is the comparison between LHTREE and BTREETree implementation of the key value store (ETCD). The graph shows the Insertion time difference between LHTREE and BTREETree implementation. As per the graph the time trend is going down as move from LHTREE to BTReeTree implementation. The same observation we can have with other parameters like deletion time and search time.



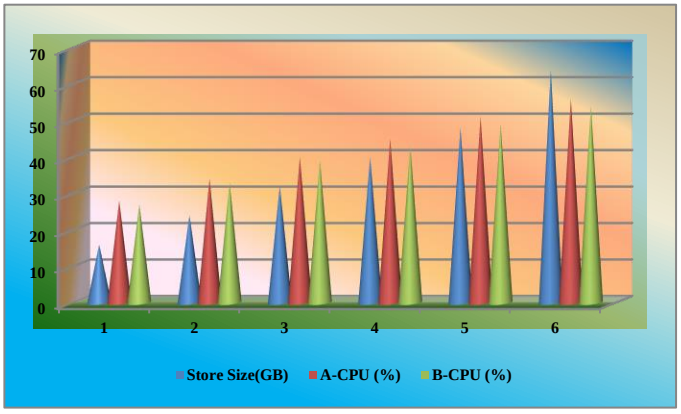
Graph 22: ETCD LHTREE Vs BTRee-2.2

Graph 22 illustrates the variation in CPU usage between the LHTREE implementation and the BtRee implementation. CPU usage decreases once the LHTREE implementation is utilized for the ETCD store.



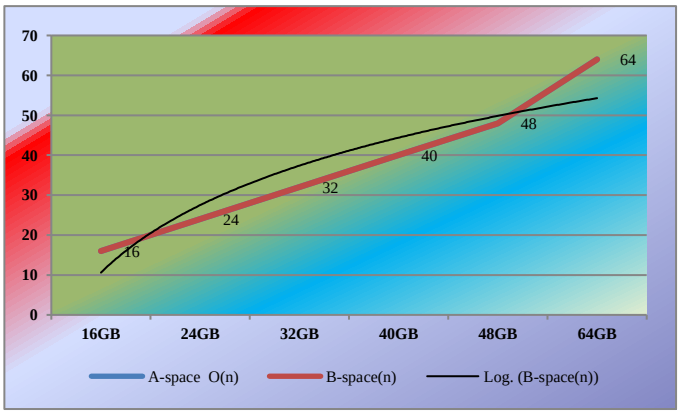
Graph 23: ETCD LHTREE Vs BTRee-3.1

Graph 23 presents a comparison between the LHTREE and BTree implementations of the key-value store (ETCD) for the third sample. The graph highlights the difference in insertion times between the LHTREE and BTree implementations, showing a downward trend as we transition from LHTREE to BTree. A similar pattern can be observed for other metrics, such as deletion time and search time.



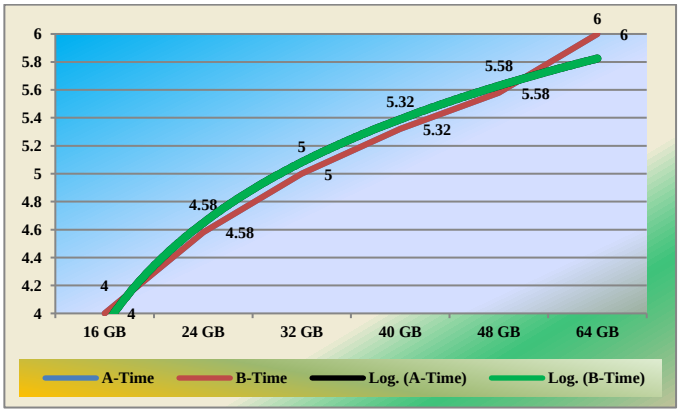
Graph 24: ETCD LHTREE Vs BTRee-3.2

Graph 24 shows that the CPU utilization is going down form high to low when we are moving from LHTREE implementation to BTReeimplementation of Key value store.



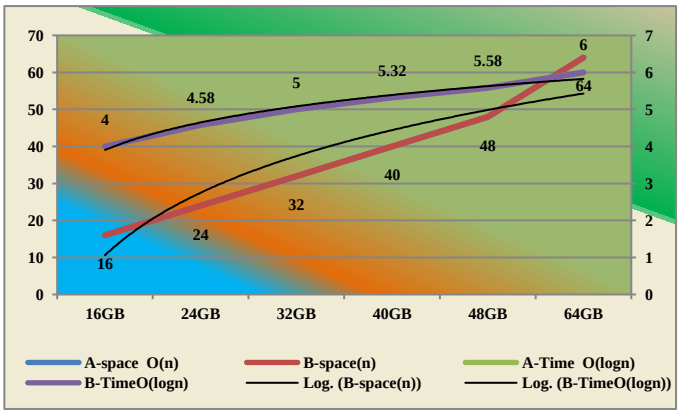
Graph 25: ETCD LHTREE Vs BTRee- Space Complexities

Graph 25 shows the space complexities comparison for the LHTREE and BTReeimplementation of the key value store.



Graph 26: ETCD LHTREE Vs BTRee-Time Complexities

Graph 26 shows the comparison of time complexities between LHTREE and BTReeimplementation of the ETCD.



Graph 27: ETCD LHTREE Vs BTReeTime and Space complexities

Graph 25, 26 and 27shows the comparison of complexities between LHTREE and BTReeTree implementation. We can conclude that by using the BTReeimplementation of the ETCD is better than using the LHTREE implementation.In summary, the time complexity of BTReeis generally O(n) for insertion, deletion, and search operations on average, with occasional $O(n)$ overheads for compaction, amortized over time. This makes BTReehighly efficient for applications requiring fast sequential writes and moderate lookup performance.

EVALUATION

The comparison between the implementation results of LHTREE trees and B-Trees demonstrates that the latter delivers superior performance. We analyzed various data store sizes, including 16GB, 24GB, 32GB, 40GB, 42GB, and 64GB, to gather statistics for evaluation. For each of these data capacities, the same performance parameters were measured and compared. Based on the analysis conducted, it is evident that insertion time, deletion time, and search time significantly improve when the data store (ETCD) is implemented using B-Trees instead of LHTREE trees.

CONCLUSION

We set up clusters with three, four, five, six, seven, eight, nine, and ten nodes, where the master node was configured with 32 CPUs, 64 GB RAM, and 500 GB storage, while worker nodes had 24 CPUs, 32 GB RAM, and 350 GB storage each. Performance testing of ETCD operations, including insertion, deletion, and search, was conducted using metrics collection tools. Results show that B-Tree implementation

outperformed LHTREE Tree implementation in all operations. Space and time complexities for both were found to be nearly identical, but CPU usage was notably lower with B-Trees.

In the LHTREE -based ETCD implementation, LHTREE Trees maintain strict balance, ensuring that search operations remain efficient with a time complexity of O(log n). This makes them particularly suitable for workloads dominated by read operations. On the other hand, B-Trees are better optimized for disk storage, especially when handling large datasets, as their broad branching factor reduces disk reads by storing more keys per node.

From the analysis conducted throughout this study, it is evident that insertion, deletion, search times, and CPU usage are reduced when employing the B-Tree implementation, while complexities remain consistent.

Future Work: Although B-Trees excel with large datasets due to their wide nodes that lower tree height, they may cause inefficient memory usage when working with smaller datasets. Future research will focus on optimizing B-Tree implementations in ETCD to handle small datasets more efficiently.

REFERENCES

1. Scalable and Reliable Kubernetes Clusters" by Google (2018).
2. Kuberenets in action by Marko Liksa , 2018.
3. Kubernetes Best Practices , Burns, Villaibha, Strebel , Evenson.
4. Learning Core DNS, Belamanic, Liu.
5. Core Kubernetes , Jay Vyas , Chris Love.
6. Kubernetes Scalability and Performance" by Red Hat (2019).
7. 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.
8. Study on the Kubernetes cluster mocel, Sourabh Vials Pilande. International Journal of Science and Research , ISSN : 2319-7064.
9. Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability, Shixiong Qi; Sameer G. Kulkarni; K. K. Ramakrishnan, 25 December 2020 , IEEEXplore.
10. Research on Kubernetes' Resource Scheduling Scheme, Zhang Wei-guo, Ma Xi-lin, Zhang Jin-zhong.
11. Improving Application availability with Pod Readiness Gates https://orielly.ly/h_WiG
12. Configure Default Memory Requests and Limits for a Namespace https://orielly.ly/ozlUi1
13. Kubernetes CSI Driver for mounting images https://orielly.ly/OMqRo
14. Modelling performance & resource management in kubernetes by Víctor Medel, Omer F. Rana, José Ángel Bañares, Unai Arronategui.
15. "etcd: A Distributed, Reliable Key-Value Store for the Edge" by Corey Olsen et al. (2018)
16. 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.
17. "Kubernetes Network Security" by Cisco (2018).
18. A Portable Load Balancer for Kubernetes Cluster, 28 January 2018, Kimitoshi Takahashi, Kento Aida, Tomoya Tanjo, Jingtao SunAuthors Info & Claims.
19. "etcd: A Highly-Available, Distributed Key-Value Store" by Brandon Philips et al. (2014), Proceedings of the 2014 ACM SIGOPS Symposium on Cloud Computing.
20. Kubernetes Storage Performance by Red Hat (2019).
21. Kubernetes Persistent Storage by Google (2018).
22. "Performance Analysis of Kubernetes Clusters" by Microsoft (2018).
23. "Secure Kubernetes Deployment" by Palo Alto Networks (2019)".

24. AVL and Red Black tree as a single balanced tree, March 2016, Zegour Djamel Eddine, Lynda Bounif
25. The log-structured merge-tree (AVL-tree),June 1996, Patrick O'Neil, Edward Cheng, Dieter Gawlick & Elizabeth O'Neil.