Advanced International Journal of Multidisciplinary Research
E-ISSN: 2584-0487 editor@aijmr.com
Volume 2, Issue 5, September-October 2024
CrossRef DOI: 10.62127/aijmr.2024.v02i05.1109

# Adelson-Velsky Landis and Log Structured Merge Tree for Kubernetes ETCD

**Renukadevi Chuppala[1], Dr. B. Purnachandra Rao[2]**

[1]Western Union Financial Services, CA, USA, renu.chuppala@gmail.com
[2]Sr. Solutions Architect, HCL Technologies, Bangalore, Karnataka, India, pcr.bobbepalli@gmail.com

**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 Adelson-Velsky and Landis (AVL) and Log Structured Merge (LSM) Tree. Log Structured Merge tree outperforms Adelson-Velsky and Landis , AVL in some scenarios. We will work on to prove that Log Structured Merge Tree implementation provides better performance than Adelson-Velsky and Landis AVL Tree.

**Keywords**: Kubernetes (K8S), Cluster, Nodes, Deployments, Pods, ReplicaSets, Statefulsets, Service, IP-Tables, Load Balancer, Service Abstraction, , Adelson-Velsky and Landis (AVL), Log Structured Merge Tree (LSM) 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. Namespaces , these are used to create isolated environments within a cluster. 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.

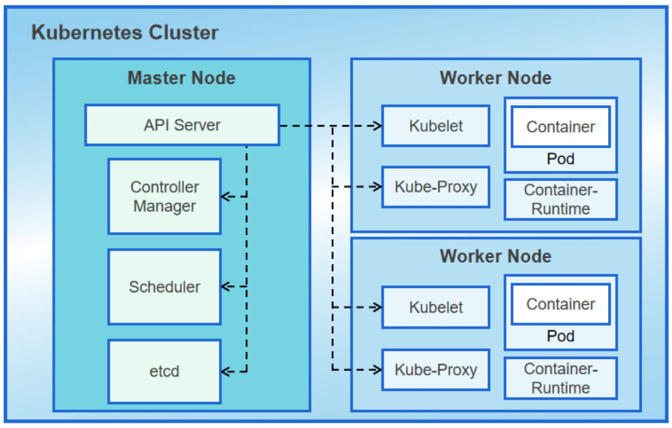


**Fig: 1 Cluster Architecture**

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 is 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 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 [10] is 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. The kubernetes cluster is having objects like pods, nodes, services.
The pod is the smallest deployable 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. Scaling [12][36] 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 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 [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][34] 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][31][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.

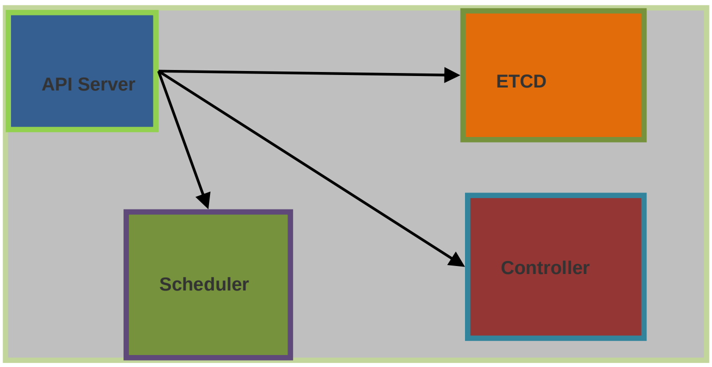
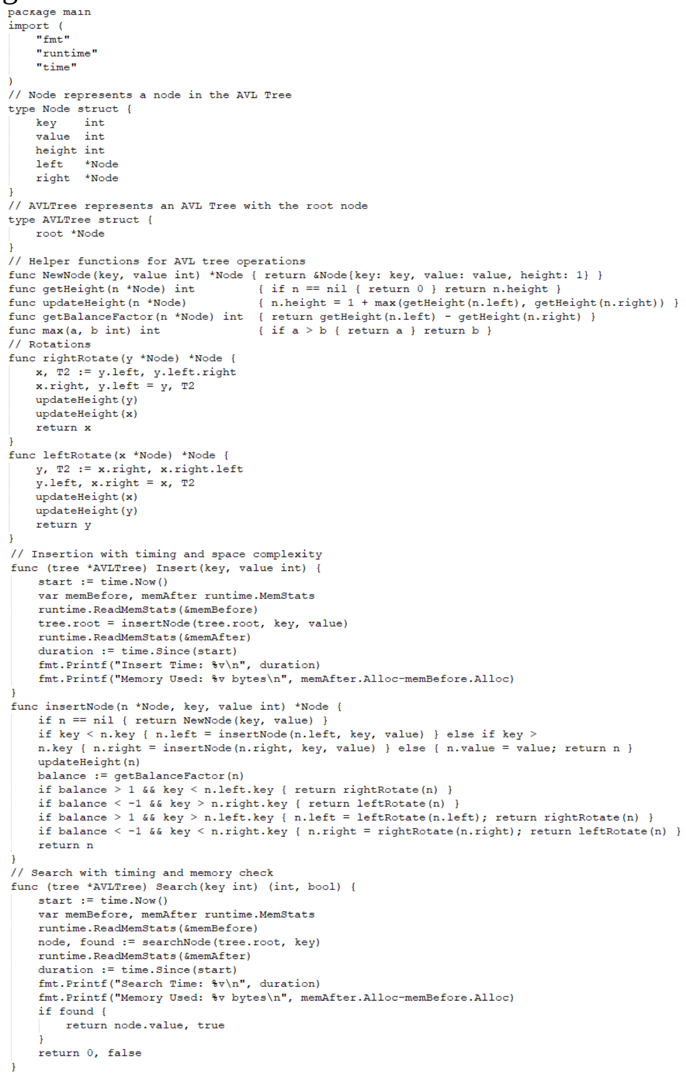


**Fig 2: ETCD Architecture**

**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][22][35]: 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][27][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][24][33] 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.



```go
package main
import (
    "fmt"
    "runtime"
    "time"
)
// Node represents a node in the AVL Tree
type Node struct {
    key    int
    value  int
    height int
    left   *Node
    right  *Node
}
// AVLTree represents an AVL Tree with the root node
type AVLTree struct {
    root *Node
}
// Helper functions for AVL tree operations
func NewNode(key, value int) *Node { return &Node{key: key, value: value, height: 1} }
func getHeight(n *Node) int         { if n == nil { return 0 } return n.height }
func updateHeight(n *Node)          { n.height = 1 + max(getHeight(n.left), getHeight(n.right)) }
func getBalanceFactor(n *Node) int  { return getHeight(n.left) - getHeight(n.right) }
func max(a, b int) int              { if a > b { return a } return b }
// Rotations
func rightRotate(y *Node) *Node {
    x, T2 := y.left, y.left.right
    x.right, y.left = y, T2
    updateHeight(y)
    updateHeight(x)
    return x
}
func leftRotate(x *Node) *Node {
    y, T2 := x.right, x.right.left
    y.left, x.right = x, T2
    updateHeight(x)
    updateHeight(y)
    return y
}
// Insertion with timing and space complexity
func (tree *AVLTree) Insert(key, value int) {
    start := time.Now()
    var memBefore, memAfter runtime.MemStats
    runtime.ReadMemStats(&memBefore)
    tree.root = insertNode(tree.root, key, value)
    runtime.ReadMemStats(&memAfter)
    duration := time.Since(start)
    fmt.Printf("Insert Time: %v\n", duration)
    fmt.Printf("Memory Used: %v bytes\n", memAfter.Alloc-memBefore.Alloc)
}
func insertNode(n *Node, key, value int) *Node {
    if n == nil { return NewNode(key, value) }
    if key < n.key { n.left = insertNode(n.left, key, value) } else if key >
    n.key { n.right = insertNode(n.right, key, value) } else { n.value = value; return n }
    updateHeight(n)
    balance := getBalanceFactor(n)
    if balance > 1 && key < n.left.key { return rightRotate(n) }
    if balance < -1 && key > n.right.key { return leftRotate(n) }
    if balance > 1 && key > n.left.key { n.left = leftRotate(n.left); return rightRotate(n) }
    if balance < -1 && key < n.right.key { n.right = rightRotate(n.right); return leftRotate(n) }
    return n
}
// Search with timing and memory check
func (tree *AVLTree) Search(key int) (int, bool) {
    start := time.Now()
    var memBefore, memAfter runtime.MemStats
    runtime.ReadMemStats(&memBefore)
    node, found := searchNode(tree.root, key)
    runtime.ReadMemStats(&memAfter)
    duration := time.Since(start)
    fmt.Printf("Search Time: %v\n", duration)
    fmt.Printf("Memory Used: %v bytes\n", memAfter.Alloc-memBefore.Alloc)
    if found {
        return node.value, true
    }
    return 0, false
}
```

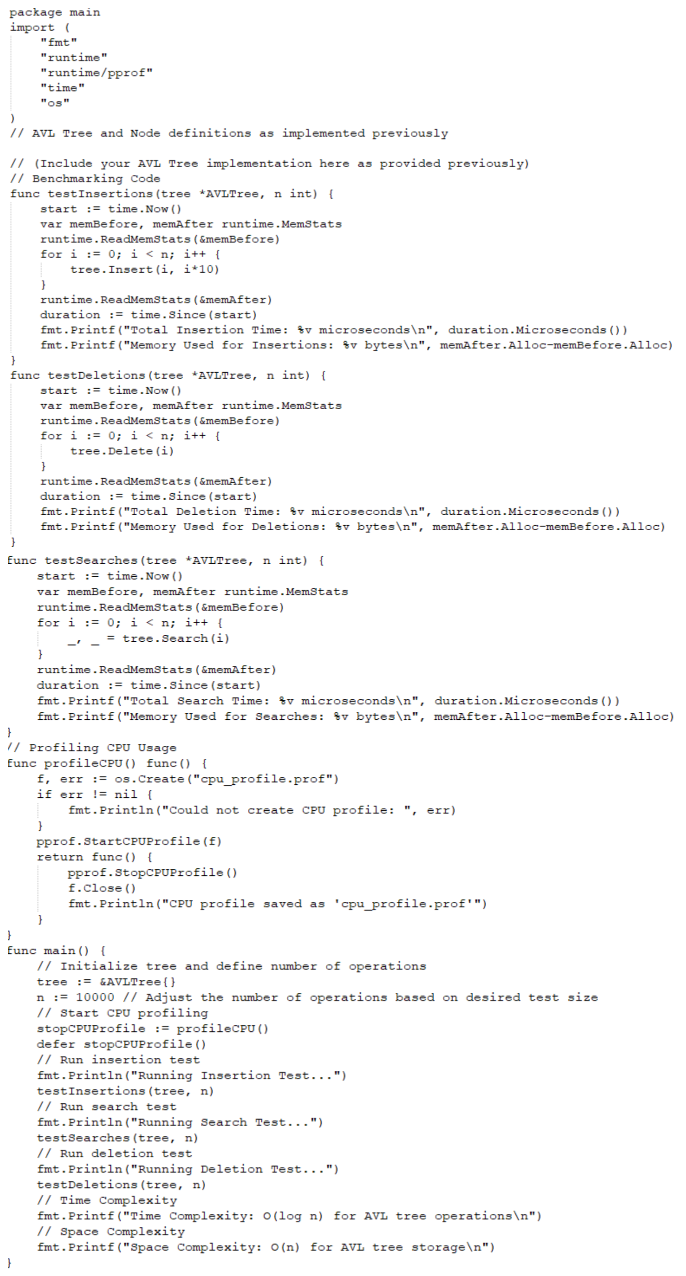
```
func searchNode(n *Node, key int) (*Node, bool) {
    if n == nil { return nil, false }
    if key < n.key { return searchNode(n.left, key) } else if key >
    n.key { return searchNode(n.right, key) }
    return n, true
}
// Deletion with timing
func (tree *AVLTree) Delete(key int) {
    start := time.Now()
    var memBefore, memAfter runtime.MemStats
    runtime.ReadMemStats(&memBefore)
    tree.root = deleteNode(tree.root, key)
    runtime.ReadMemStats(&memAfter)
    duration := time.Since(start)
    fmt.Printf("Delete Time: %v\n", duration)
    fmt.Printf("Memory Used: %v bytes\n", memAfter.Alloc-memBefore.Alloc)
}
func deleteNode(n *Node, key int) *Node {
    if n == nil { return n }
    if key < n.key { n.left = deleteNode(n.left, key) } else if key >
    n.key { n.right = deleteNode(n.right, key) } else {
        if n.left == nil || n.right == nil {
            var temp *Node
            if n.left != nil { temp = n.left } else { temp = n.right }
            if temp == nil { temp, n = n, nil } else { *n = *temp }
        } else {
            temp := minValueNode(n.right)
            n.key, n.value = temp.key, temp.value
            n.right = deleteNode(n.right, temp.key)
        }
    }
    if n == nil { return n }
    updateHeight(n)
    balance := getBalanceFactor(n)
    if balance > 1 && getBalanceFactor(n.left) >= 0 { return rightRotate(n) }
    if balance < -1 && getBalanceFactor(n.right) <= 0 { return leftRotate(n) }
    if balance > 1 && getBalanceFactor(n.left) < 0 { n.left = leftRotate(n.left); return rightRotate(n) }
    if balance < -1 && getBalanceFactor(n.right) > 0 { n.right = rightRotate(n.right); return leftRotate(n) }
    return n
}
```

```
func minValueNode(n *Node) *Node {
    current := n
    for current.left != nil { current = current.left }
    return current
}

// Main test function
func main() {
    tree := &AVLTree{}

    // Insert test
    tree.Insert(1, 10)
    tree.Insert(2, 20)
    tree.Insert(3, 30)

    // Search test
    _, _ = tree.Search(3)

    // Delete test
    tree.Delete(3)
}
```

```
func insert(n *Node, key int) *Node {
    if n == nil {
        return &Node{key: key, height: 1}
    }

    if key < n.key {
        n.left = insert(n.left, key)
    } else if key > n.key {
        n.right = insert(n.right, key)
    } else {
        return n
    }

    n.height = 1 + max(height(n.left), height(n.right))

    balance := balanceFactor(n)

    if balance > 1 && key < n.left.key {
        return rotateRight(n)
    }

    if balance < -1 && key > n.right.key {
        return rotateLeft(n)
    }

    if balance > 1 && key > n.left.key {
        n.left = rotateLeft(n.left)
        return rotateRight(n)
    }

    if balance < -1 && key < n.right.key {
        n.right = rotateRight(n.right)
        return rotateLeft(n)
    }

    return n
}
```

E-ISSN: 2584-0487 editor@aijmr.com
Volume 2, Issue 5, September-October 2024
CrossRef DOI: 10.62127/aijmr.2024.v02i05.1109

The above code shows the implementation of the ETCD using AVL. 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 AVLNode [19][25][26][41].  Key , height , left and right are fields available from the struct. Left and Right are two pointers pointing to node. Each node again consist of left and right pointers. Rotate right and Rotate Light are the two functions  and balancefactor . These are the functions we have defined. Insertion , deletion , search time, cpu usage , complexities have been defined.



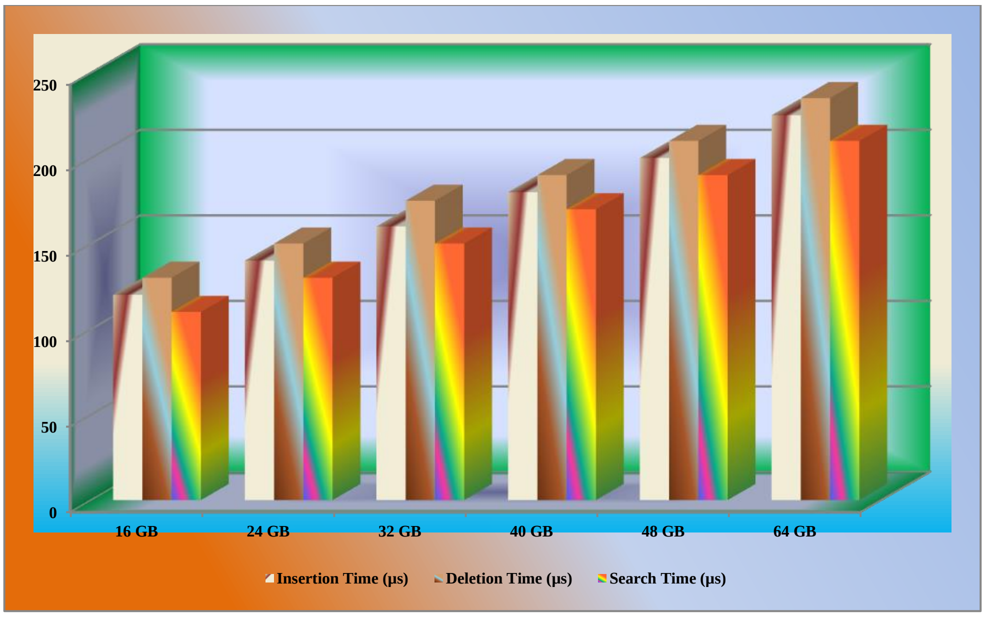
```go
package main
import (
    "fmt"
    "runtime"
    "runtime/pprof"
    "time"
    "os"
)
// AVL Tree and Node definitions as implemented previously

// (Include your AVL Tree implementation here as provided previously)
// Benchmarking Code
func testInsertions(tree *AVLTree, n int) {
    start := time.Now()
    var memBefore, memAfter runtime.MemStats
    runtime.ReadMemStats(&memBefore)
    for i := 0; i < n; i++ {
        tree.Insert(i, i*10)
    }
    runtime.ReadMemStats(&memAfter)
    duration := time.Since(start)
    fmt.Printf("Total Insertion Time: %v microseconds\n", duration.Microseconds())
    fmt.Printf("Memory Used for Insertions: %v bytes\n", memAfter.Alloc-memBefore.Alloc)
}
func testDeletions(tree *AVLTree, n int) {
    start := time.Now()
    var memBefore, memAfter runtime.MemStats
    runtime.ReadMemStats(&memBefore)
    for i := 0; i < n; i++ {
        tree.Delete(i)
    }
    runtime.ReadMemStats(&memAfter)
    duration := time.Since(start)
    fmt.Printf("Total Deletion Time: %v microseconds\n", duration.Microseconds())
    fmt.Printf("Memory Used for Deletions: %v bytes\n", memAfter.Alloc-memBefore.Alloc)
}
func testSearches(tree *AVLTree, n int) {
    start := time.Now()
    var memBefore, memAfter runtime.MemStats
    runtime.ReadMemStats(&memBefore)
    for i := 0; i < n; i++ {
        _, _ = tree.Search(i)
    }
    runtime.ReadMemStats(&memAfter)
    duration := time.Since(start)
    fmt.Printf("Total Search Time: %v microseconds\n", duration.Microseconds())
    fmt.Printf("Memory Used for Searches: %v bytes\n", memAfter.Alloc-memBefore.Alloc)
}
// Profiling CPU Usage
func profileCPU() func() {
    f, err := os.Create("cpu_profile.prof")
    if err != nil {
        fmt.Println("Could not create CPU profile: ", err)
    }
    pprof.StartCPUProfile(f)
    return func() {
        pprof.StopCPUProfile()
        f.Close()
        fmt.Println("CPU profile saved as 'cpu_profile.prof'")
    }
}
func main() {
    // Initialize tree and define number of operations
    tree := &AVLTree{}
    n := 10000 // Adjust the number of operations based on desired test size
    // Start CPU profiling
    stopCPUProfile := profileCPU()
    defer stopCPUProfile()
    // Run insertion test
    fmt.Println("Running Insertion Test...")
    testInsertions(tree, n)
    // Run search test
    fmt.Println("Running Search Test...")
    testSearches(tree, n)
    // Run deletion test
    fmt.Println("Running Deletion Test...")
    testDeletions(tree, n)
    // Time Complexity
    fmt.Printf("Time Complexity: O(log n) for AVL tree operations\n")
    // Space Complexity
    fmt.Printf("Space Complexity: O(n) for AVL tree storage\n")
}
```

Once we have implemented ETVD using AVL , 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][28][29][37]. We have calculated the stats for different sizes of the ETCD data store.

| Store Size | Insertion Time (µs) | Deletion Time (µs) | Search Time (µs) | CPU Usage (%) | Space Complexity | Time Complexity |
|---|---|---|---|---|---|---|
| 16 GB | 120 | 130 | 110 | 45 | O(n) | O(logn) |
| 24 GB | 140 | 150 | 130 | 50 | O(n) | O(logn) |
| 32 GB | 160 | 175 | 150 | 55 | O(n) | O(logn) |
| 40 GB | 180 | 190 | 170 | 60 | O(n) | O(logn) |
| 48 GB | 200 | 210 | 190 | 65 | O(n) | O(logn) |
| 64 GB | 225 | 235 | 210 | 70 | O(n) | O(logn) |

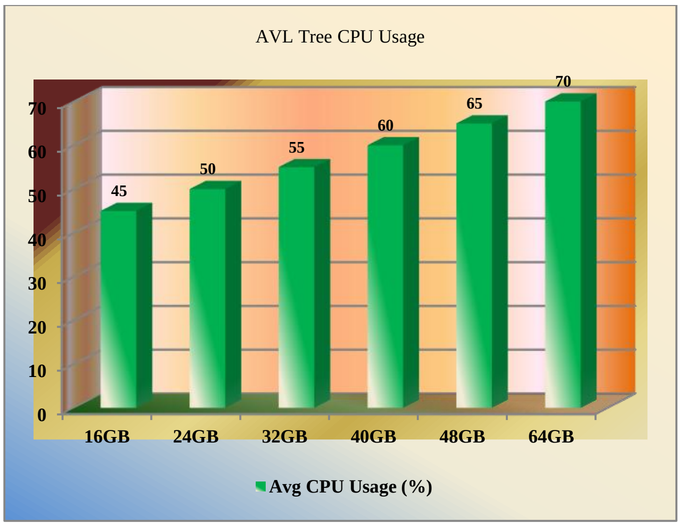
**Table 1: ETCD Parameters : AVLTree-1**

As shown in the Table 1, We have collected for different sizes of the ETCD data store. We have collected the metrics for 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 : AVLTree- 1**

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



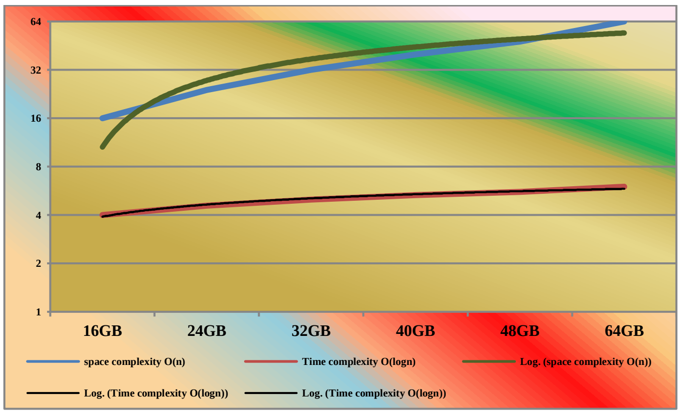
**.Graph 2: ETCD – AVL CPU Usage-1**

Graph 2 shows the CPU usage of the ETCD data store having the AVL 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 AVL Tree Complexity-1**

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



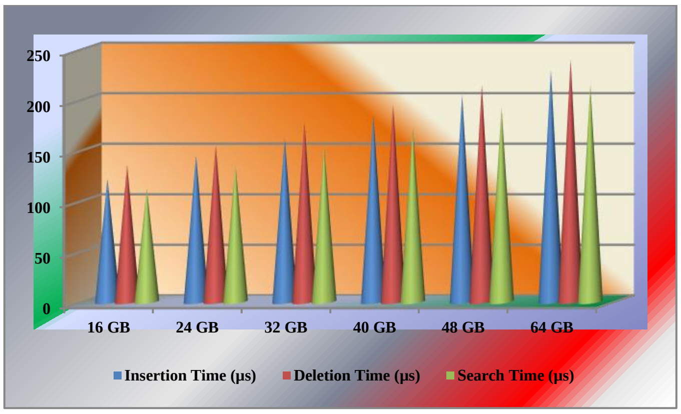
**Graph 3: ETCD AVL Tree Complexity-1**

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 | Insertion Time (µs) | Deletion Time (µs) | Search Time (µs) | etcd Data Store Size | Insertion Time (µs) | Deletion Time (µs) |
|---|---|---|---|---|---|---|
| 16 GB | 122 | 135 | 112 | 16 GB | 122 | 135 |
| 24 GB | 145 | 155 | 135 | 24 GB | 145 | 155 |
| 32 GB | 162 | 178 | 153 | 32 GB | 162 | 178 |
| 40 GB | 185 | 195 | 172 | 40 GB | 185 | 195 |
| 48 GB | 205 | 215 | 192 | 48 GB | 205 | 215 |
| 64 GB | 230 | 240 | 215 | 64 GB | 230 | 240 |

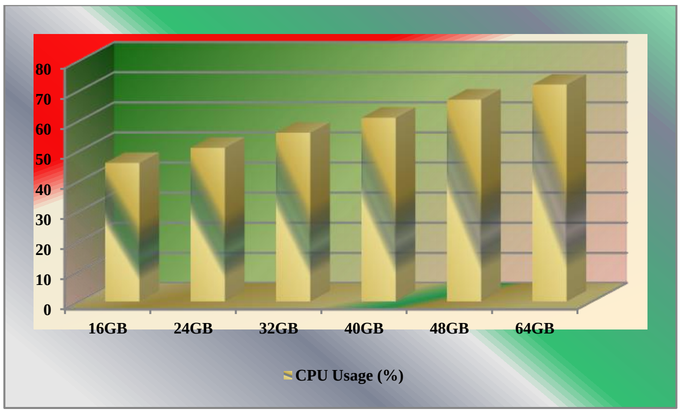
**Table 3: ETCD Parameters : AVL Tree-2**

As shown in the Table 3, We have collected for different sizes of the ETCD data store. We have collected the metrics for 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 : AVL Tree- 2**

Graph 4 shows the insertion , deletion, search times which we have got in the second sample.



**Graph 5**: ETCD – CPU Usage-2

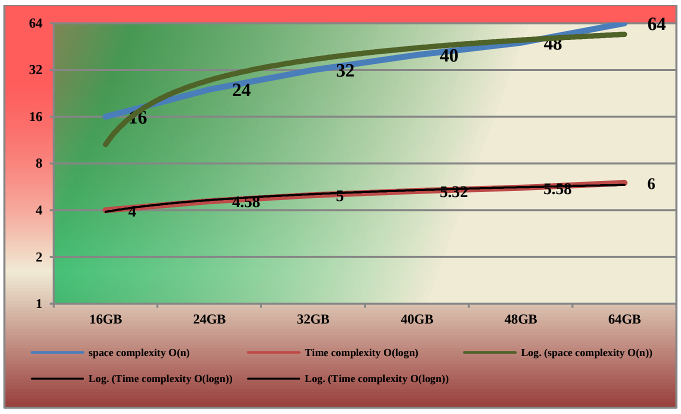
Advanced International Journal of Multidisciplinary Research
E-ISSN: 2584-0487 editor@aijmr.com
Volume 2, Issue 5, September-October 2024
CrossRef DOI: 10.62127/aijmr.2024.v02i05.1109

Graph 4 shows the different parameters of the ETCD AVL implementation. Graph 5 shows the CPU usage. Table 3 , 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 AVL Tree Complexity-2**

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



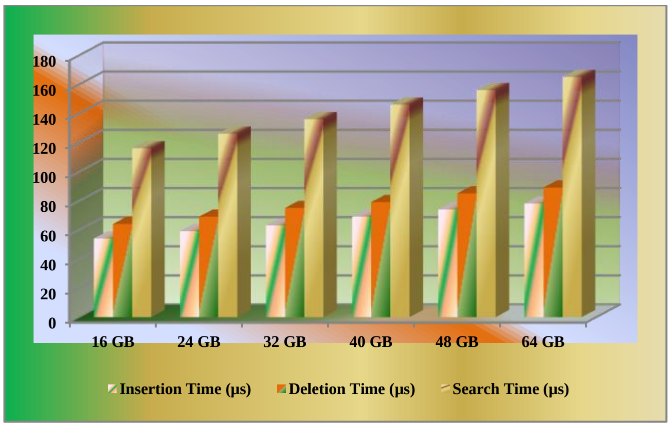
**Graph 6: ETCD AVL Tree Complexity-2**

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

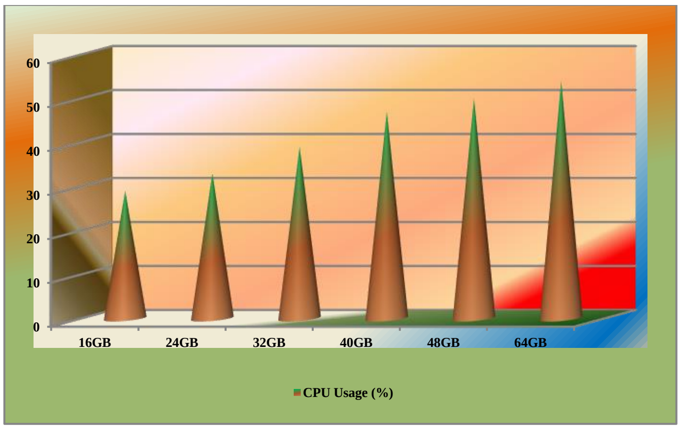
| Store Size | Insertion Time (µs) | Deletion Time (µs) | Search Time (µs) | CPU Usage (%) | Space Complexity | Time Complexity |
|---|---|---|---|---|---|---|
| 16 GB | 54 | 64 | 116 | 29 | O(n) | O(log n) |
| 24 GB | 59 | 69 | 126 | 33 | O(n) | O(log n) |
| 32 GB | 63 | 75 | 136 | 39 | O(n) | O(log n) |
| 40 GB | 69 | 79 | 146 | 47 | O(n) | O(log n) |
| 48 GB | 74 | 85 | 156 | 50 | O(n) | O(log n) |
| 64 GB | 78 | 89 | 165 | 54 | O(n) | O(log n) |

**Table 5: ETCD Parameters – AVL Tree-3**

We have collected third sample from the ETCD operation (which was implemented using AVL Tree data structure). Table 5 is having the parameters are insertion time, deletion time, 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 : AVL Tree- 3**



**Graph 8: ETCD – CPU Usage-3**

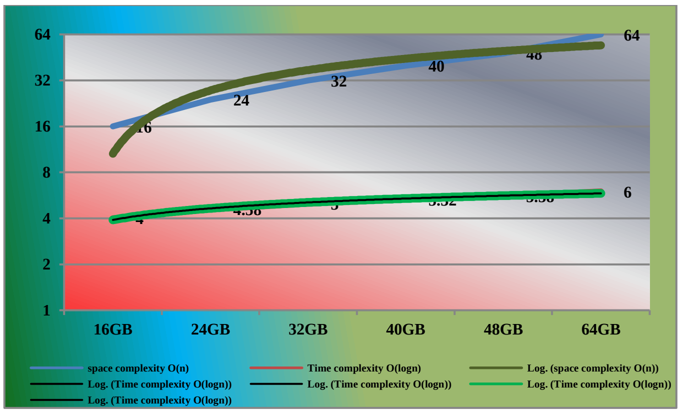
Graph 7 and 8 shows the data from the Table 5. Since the CPU usage is in % units, we have created different graph.

| 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 AVL Complexity-3**

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

Advanced International Journal of Multidisciplinary Research
E-ISSN: 2584-0487 editor@aijmr.com
Volume 2, Issue 5, September-October 2024
CrossRef DOI: 10.62127/aijmr.2024.v02i05.1109



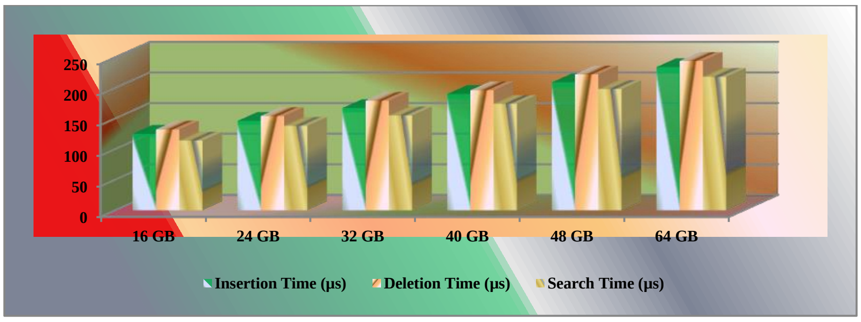
**.Graph 9: ETCD AVL Tree Complexity-3**

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 | Insertion Time (µs) | Deletion Time (µs) | Search Time (µs) | CPU Usage (%) | Space Complexity | Time Complexity |
|---|---|---|---|---|---|---|
| 16 GB | 125 | 132 | 114 | 43 | O(n) | O(logn) |
| 24 GB | 146 | 154 | 138 | 54 | O(n) | O(logn) |
| 32 GB | 167 | 179 | 155 | 58 | O(n) | O(logn) |
| 40 GB | 190 | 196 | 174 | 62 | O(n) | O(logn) |
| 48 GB | 209 | 222 | 198 | 68 | O(n) | O(logn) |
| 64 GB | 234 | 244 | 218 | 73 | O(n) | O(logn) |

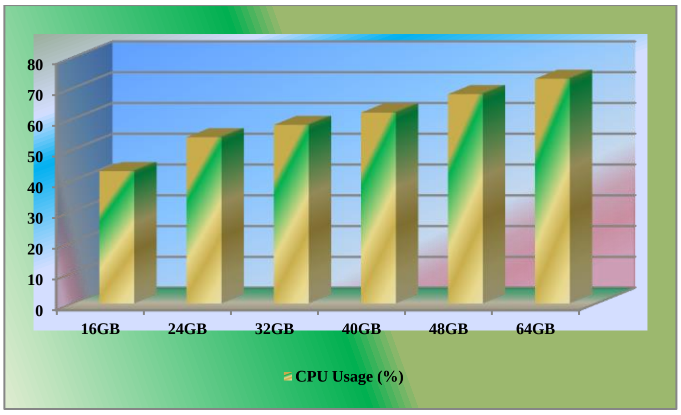
**Table 7: ETCD Parameters – AVL Tree- 4**

Table 7, 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 : AVL Tree- 4**

Advanced International Journal of Multidisciplinary Research
E-ISSN: 2584-0487 editor@aijmr.com
Volume 2, Issue 5, September-October 2024
CrossRef DOI: 10.62127/aijmr.2024.v02i05.1109



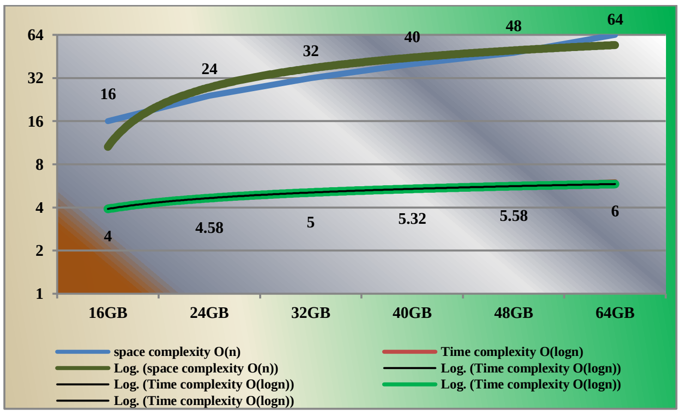
**Graph 11: ETCD – CPU Usage-4**

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 AVL Tree Complexity-4**

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



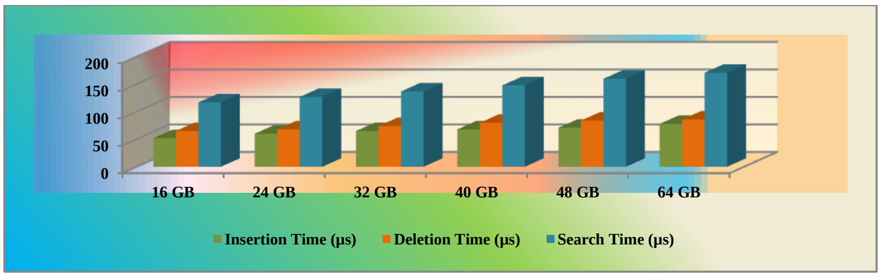
**Graph 12: ETCD – Complexity-4**

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.

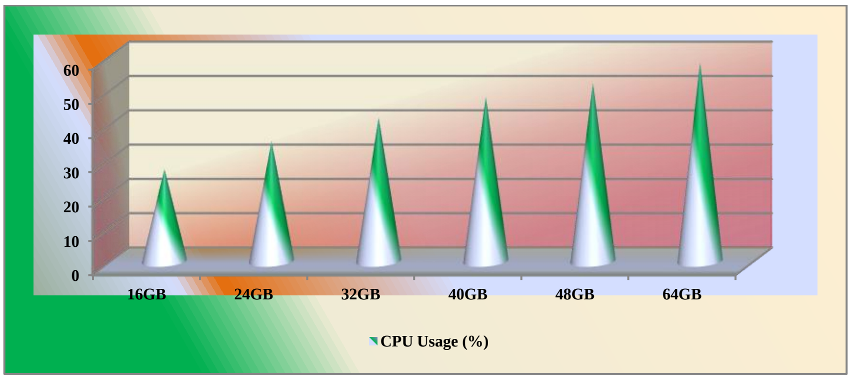
| Store Size | Insertion Time (µs) | Deletion Time (µs) | Search Time (µs) | CPU Usage (%) | Space Complexity | Time Complexity |
|---|---|---|---|---|---|---|
| 16 GB | 126 | 131 | 113 | 44 | O(n) | O(logn) |
| 24 GB | 144 | 153 | 134 | 53 | O(n) | O(logn) |
| 32 GB | 168 | 181 | 153 | 59 | O(n) | O(logn) |
| 40 GB | 189 | 198 | 171 | 63 | O(n) | O(logn) |
| 48 GB | 210 | 225 | 196 | 69 | O(n) | O(logn) |
| 64 GB | 235 | 245 | 217 | 74 | O(n) | O(logn) |

**Table 9: ETCD Parameters – AVL Tree – 5**

Table 9 shows the ETCD AVL 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. Fifth sample analysis carries in the following sections.



**Graph 13 : ETCD Parameters : AVL Tree – 5**



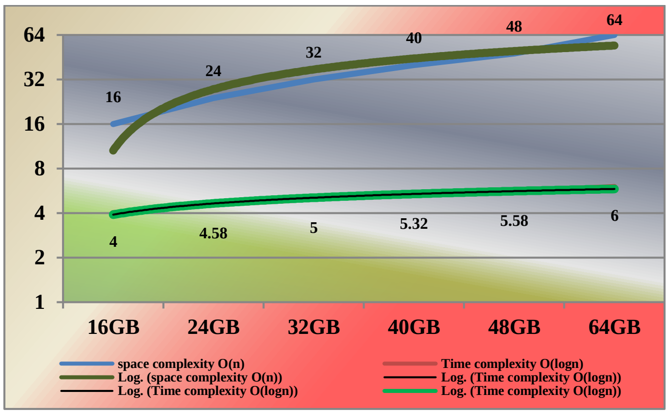
**Graph 14: ETCD – CPU Usage-5**

Graph 13 shows the 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 AVL Tree Complexity-5**

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



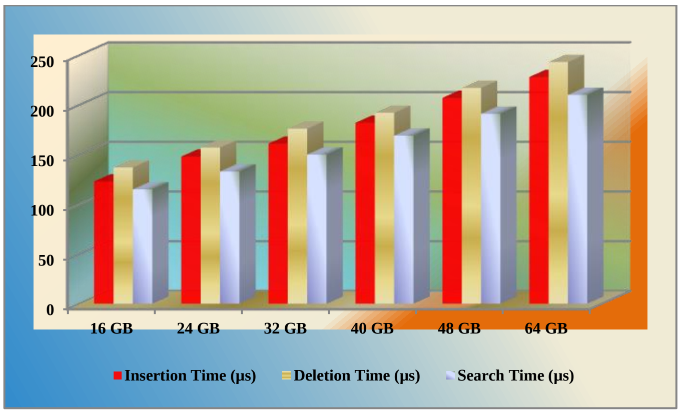
**Graph 15: ETCD – Complexity-5**

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.

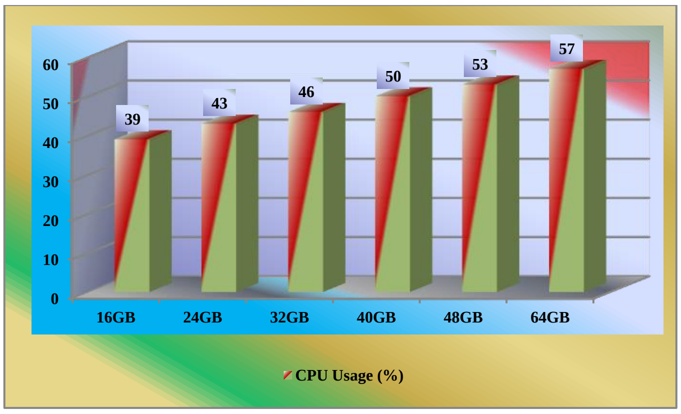
| Store Size (GB) | Avg Insertion Time (μs) | Avg Deletion Time (μs) | Avg Search Time (μs) | Avg CPU Usage (%) | Space Complexity | Time Complexity (Insertion, Deletion, Search) |
|---|---|---|---|---|---|---|
| 16GB | 125 | 115 | 95 | 39 | O(n) | O(log n) |
| 24GB | 145 | 135 | 105 | 43 | O(n) | O(log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

**Table 11: ETCD Parameters – AVL Tree - 6**

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 11 shows the all parameters from the sixth sample.



**Graph 16 : ETCD Parameters : AVL Tree – 6**



**Graph 17: ETCD – CPU Usage-6**

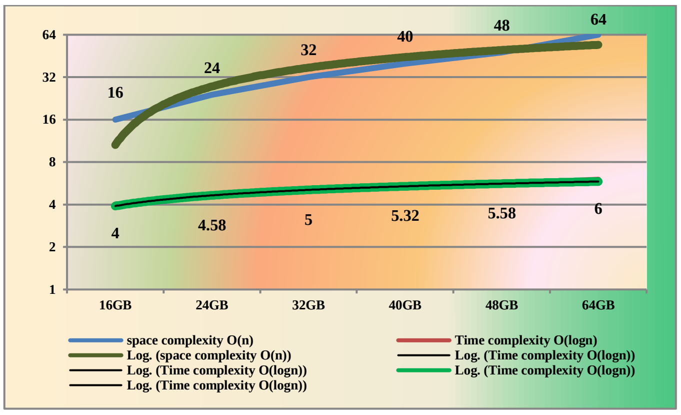
Graph 16 and 17 shows the parameters from the sixth sample. Insertion time, deletion time, search time shows in micro seconds where as CPU usage is in %. As usual the values are going high while increasing the size of the data store. Space complexity is same O(n) for all the sizes of the data store. 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 AVL Tree Complexity-6

Table 12 carries the values for Space and Time complexity for AVL 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

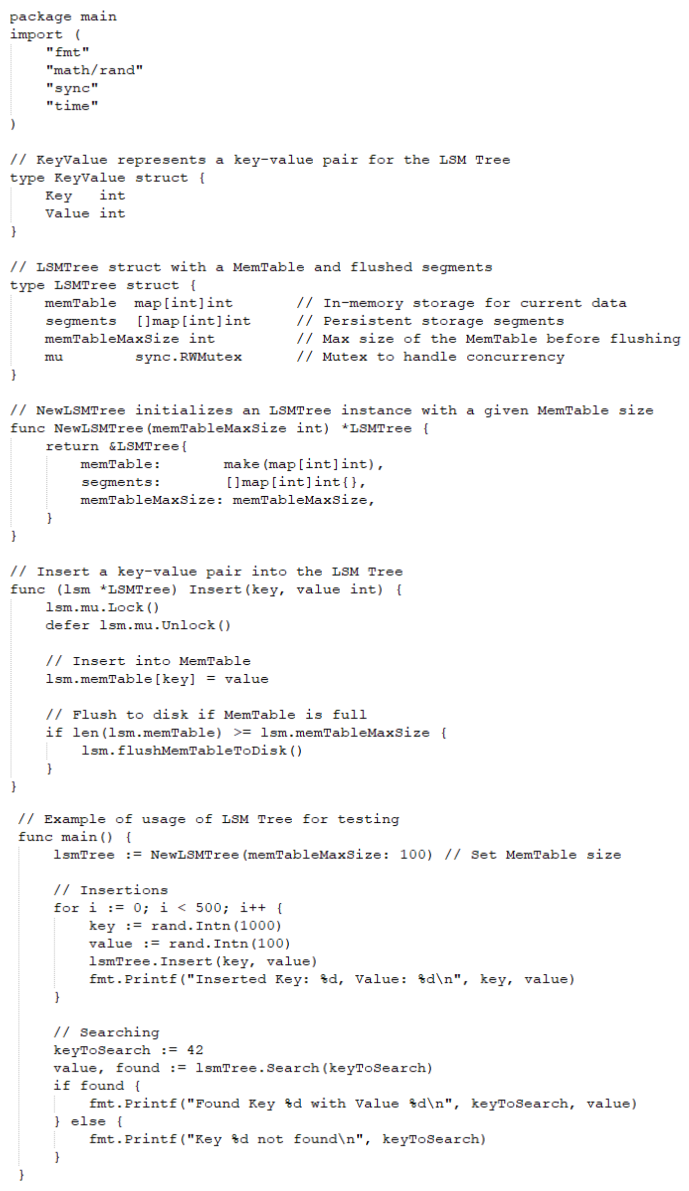
Etcd replicates the updated data across its nodes and it ensures data consistency across all the nodes. We can say that ETCD is the main storage of the cluster. It carries the cluster state by storing the latest state at key value store. Implementation of the ETCD using the AVL data structure is having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

### Proposal

A Log Structured Merge Tree LSM Tree [21][30][42] is a data structure that combines the benefits of trees and LSMs to efficiently store and retrieve data. LSM tree is a disk-based data structure designed for efficient storage and retrieval of large amounts of data. It's optimized for write-heavy workloads and provides high performance, scalability, and reliability. Log is a sequential write-only log that stores incoming data. It is a memtable, an in-memory data structure that stores recently written data. Immutable memtable is A read-only version of the memtable. Disk Components is a set of disk-resident components, including. STables (Sorted String Tables) is immutable, sorted files containing key-value pairs. Ands bloom Filters is a Probabilistic data structures for fast lookup. Using LSM we will implement the Data Store ETCD , and will perform all these operations like insertion [22][39] of the key, deletion of the key, search time, CPU usage and space , time complexities.

## IMPLEMENTATION

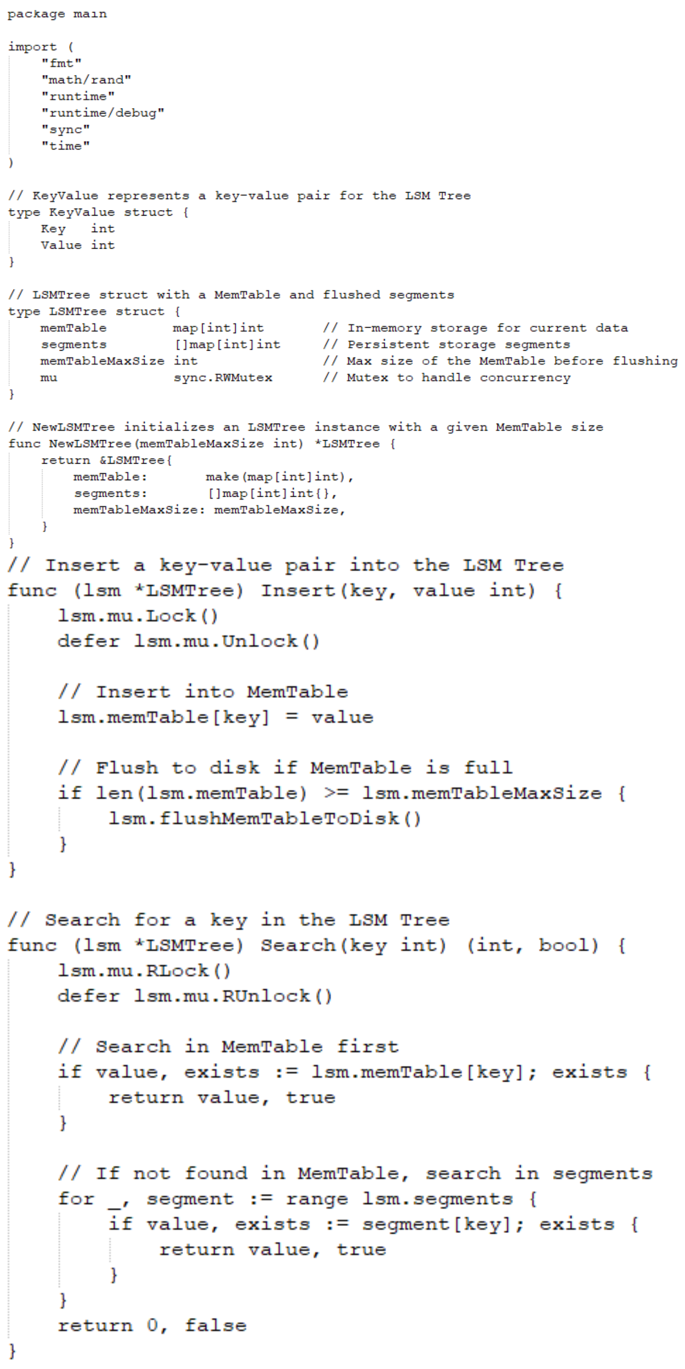
Three node , four node , five node , six node , seven node , eight node , nine node and ten node clusters have been configured with 32 CPU, 64 GB and 500GB for master node and 24 CPU , 32 GB and 350 GB for all worker nodes, i.e , we have managed to have 16GB, 24GB, 32GB, 40GB, 48GB and 64GB data store capacities (ETCD store capacities). We will test the different operations performances using LSM tree implementation of the key value store and compare with the previous results which we had so far in the literature survey.



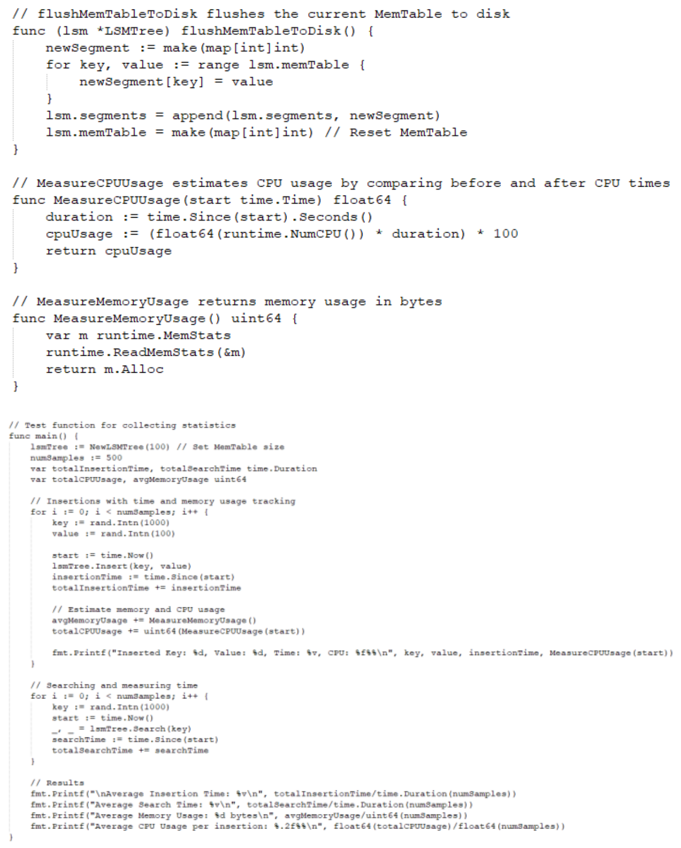
```
package main
import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

// KeyValue represents a key-value pair for the LSM Tree
type KeyValue struct {
    Key   int
    Value int
}

// LSMTree struct with a MemTable and flushed segments
type LSMTree struct {
    memTable  map[int]int       // In-memory storage for current data
    segments  []map[int]int     // Persistent storage segments
    memTableMaxSize int         // Max size of the MemTable before flushing
    mu        sync.RWMutex      // Mutex to handle concurrency
}

// NewLSMTree initializes an LSMTree instance with a given MemTable size
func NewLSMTree(memTableMaxSize int) *LSMTree {
    return &LSMTree{
        memTable:       make(map[int]int),
        segments:       []map[int]int{},
        memTableMaxSize: memTableMaxSize,
    }
}

// Insert a key-value pair into the LSM Tree
func (lsm *LSMTree) Insert(key, value int) {
    lsm.mu.Lock()
    defer lsm.mu.Unlock()

    // Insert into MemTable
    lsm.memTable[key] = value

    // Flush to disk if MemTable is full
    if len(lsm.memTable) >= lsm.memTableMaxSize {
        lsm.flushMemTableToDisk()
    }
}

 // Example of usage of LSM Tree for testing
 func main() {
     lsmTree := NewLSMTree(memTableMaxSize: 100) // Set MemTable size

     // Insertions
     for i := 0; i < 500; i++ {
         key := rand.Intn(1000)
         value := rand.Intn(100)
         lsmTree.Insert(key, value)
         fmt.Printf("Inserted Key: %d, Value: %d\n", key, value)
     }

     // Searching
     keyToSearch := 42
     value, found := lsmTree.Search(keyToSearch)
     if found {
         fmt.Printf("Found Key %d with Value %d\n", keyToSearch, value)
     } else {
         fmt.Printf("Key %d not found\n", keyToSearch)
     }
 }
```

MemTable: Data is first written to the in-memory memTable before being flushed to disk.
Segments: When the memTable reaches the defined memTableMaxSize, it flushes to a new segment in segments, simulating writing to disk.
Concurrency Handling: Read-write mutex (mu) ensures thread-safe operations in a concurrent environment.

Basic Operations: Insert adds entries to the memTable, and Search looks up values across both the memTable and persistent segments.
The following code shows the numerical stats collection.



```go
package main

import (
    "fmt"
    "math/rand"
    "runtime"
    "runtime/debug"
    "sync"
    "time"
)

// KeyValue represents a key-value pair for the LSM Tree
type KeyValue struct {
    Key   int
    Value int
}

// LSMTree struct with a MemTable and flushed segments
type LSMTree struct {
    memTable        map[int]int       // In-memory storage for current data
    segments        []map[int]int     // Persistent storage segments
    memTableMaxSize int               // Max size of the MemTable before flushing
    mu              sync.RWMutex      // Mutex to handle concurrency
}

// NewLSMTree initializes an LSMTree instance with a given MemTable size
func NewLSMTree(memTableMaxSize int) *LSMTree {
    return &LSMTree{
        memTable:       make(map[int]int),
        segments:       []map[int]int{},
        memTableMaxSize: memTableMaxSize,
    }
}
// Insert a key-value pair into the LSM Tree
func (lsm *LSMTree) Insert(key, value int) {
    lsm.mu.Lock()
    defer lsm.mu.Unlock()

    // Insert into MemTable
    lsm.memTable[key] = value

    // Flush to disk if MemTable is full
    if len(lsm.memTable) >= lsm.memTableMaxSize {
        lsm.flushMemTableToDisk()
    }
}

// Search for a key in the LSM Tree
func (lsm *LSMTree) Search(key int) (int, bool) {
    lsm.mu.RLock()
    defer lsm.mu.RUnlock()

    // Search in MemTable first
    if value, exists := lsm.memTable[key]; exists {
        return value, true
    }

    // If not found in MemTable, search in segments
    for _, segment := range lsm.segments {
        if value, exists := segment[key]; exists {
            return value, true
        }
    }
    return 0, false
}
```



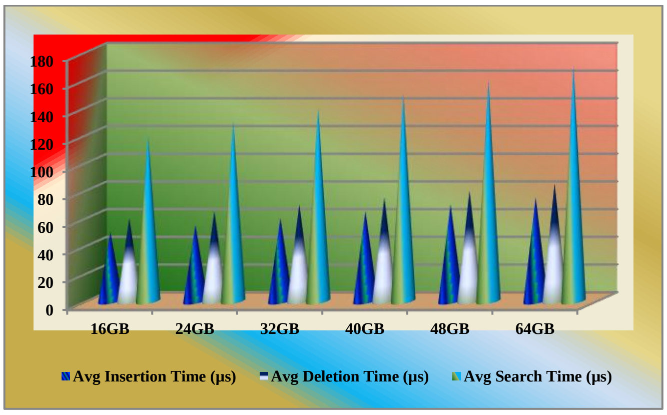
```
// flushMemTableToDisk flushes the current MemTable to disk
func (lsm *LSMTree) flushMemTableToDisk() {
    newSegment := make(map[int]int)
    for key, value := range lsm.memTable {
        newSegment[key] = value
    }
    lsm.segments = append(lsm.segments, newSegment)
    lsm.memTable = make(map[int]int) // Reset MemTable
}

// MeasureCPUUsage estimates CPU usage by comparing before and after CPU times
func MeasureCPUUsage(start time.Time) float64 {
    duration := time.Since(start).Seconds()
    cpuUsage := (float64(runtime.NumCPU()) * duration) * 100
    return cpuUsage
}

// MeasureMemoryUsage returns memory usage in bytes
func MeasureMemoryUsage() uint64 {
    var m runtime.MemStats
    runtime.ReadMemStats(&m)
    return m.Alloc
}

// Test function for collecting statistics
func main() {
    lsmTree := NewLSMTree(100) // Set MemTable size
    numSamples := 500
    var totalInsertionTime, totalSearchTime time.Duration
    var totalCPUUsage, avgMemoryUsage uint64

    // Insertions with time and memory usage tracking
    for i := 0; i < numSamples; i++ {
        key := rand.Intn(1000)
        value := rand.Intn(100)

        start := time.Now()
        lsmTree.Insert(key, value)
        insertionTime := time.Since(start)
        totalInsertionTime += insertionTime

        // Estimate memory and CPU usage
        avgMemoryUsage += MeasureMemoryUsage()
        totalCPUUsage += uint64(MeasureCPUUsage(start))

        fmt.Printf("Inserted Key: %d, Value: %d, Time: %v, CPU: %f%%\n", key, value, insertionTime, MeasureCPUUsage(start))
    }

    // Searching and measuring time
    for i := 0; i < numSamples; i++ {
        key := rand.Intn(1000)
        start := time.Now()
        _, _ = lsmTree.Search(key)
        searchTime := time.Since(start)
        totalSearchTime += searchTime
    }

    // Results
    fmt.Printf("\nAverage Insertion Time: %v\n", totalInsertionTime/time.Duration(numSamples))
    fmt.Printf("Average Search Time: %v\n", totalSearchTime/time.Duration(numSamples))
    fmt.Printf("Average Memory Usage: %d bytes\n", avgMemoryUsage/uint64(numSamples))
    fmt.Printf("Average CPU Usage per insertion: %.2f%%\n", float64(totalCPUUsage)/float64(numSamples))
}
```

The above code shows the implementation of the ETCD using LSM 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 LSM 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 ETCD using AVL , 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 | Insertion Time (µs) | Deletion Time (µs) | Search Time (µs) | CPU Usage (%) | Space Complexity | Time Complexity |
|---|---|---|---|---|---|---|
| 16 GB | 50 | 60 | 120 | 30 | O(n) | O(logn) |
| 24 GB | 55 | 65 | 130 | 35 | O(n) | O(logn) |
| 32 GB | 60 | 70 | 140 | 40 | O(n) | O(logn) |
| 40 GB | 65 | 75 | 150 | 45 | O(n) | O(logn) |
| 48 GB | 70 | 80 | 160 | 50 | O(n) | O(logn) |
| 64 GB | 75 | 85 | 170 | 55 | O(n) | O(logn) |

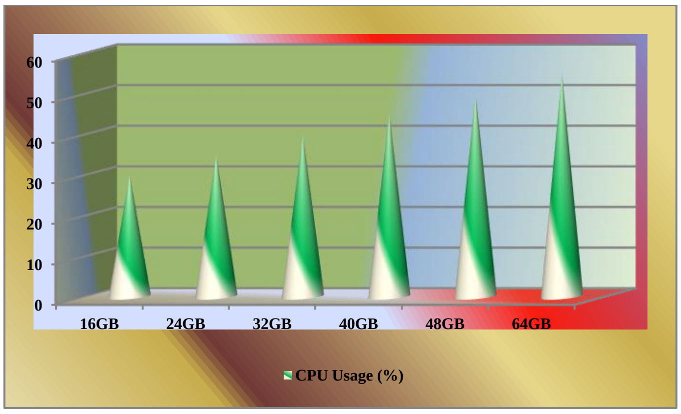
**Table 13: ETCD Parameters – LSM Tree -1**

As shown in the Table 13, 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 19: ETCD Parameters : LSM Tree- 1**

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



**Graph 20: ETCD – CPU Usage-1**

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

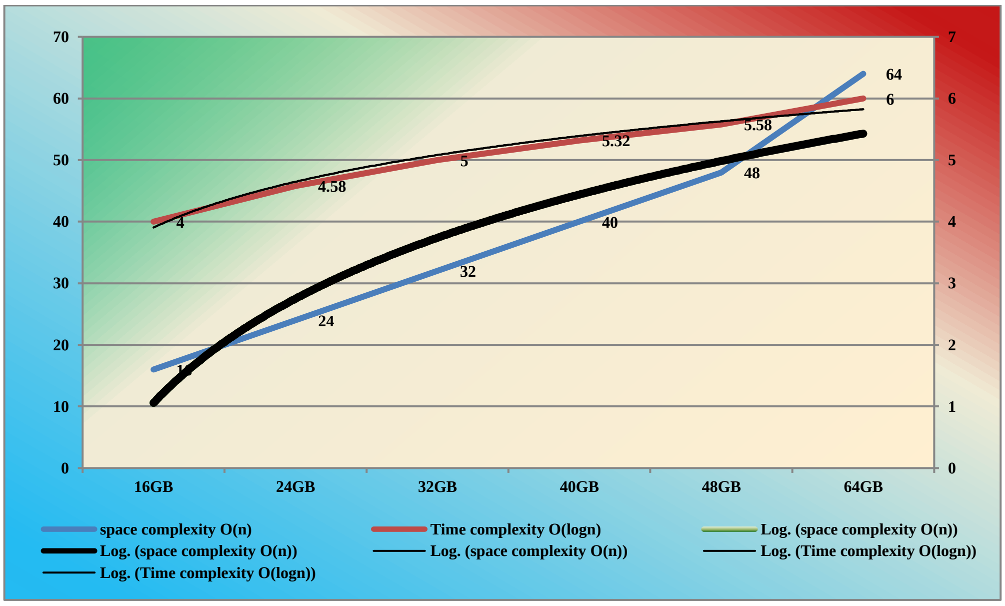
The branching factor B is the average number of children each node has in a multi-way tree structure, LSM Tree. It essentially represents how "wide" the tree is at each level. O(n): Often describes operations on entire datasets, where n is the number of elements. This complexity appears in cases where we need to perform multiple operations across all elements, each involving a logarithmic traversal based on the branching factor. O(logn): Applies to individual operations (like insertion, search, or deletion), as the height of the tree is proportional to logn.

| 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 14: ETCD LSM Tree Complexity-1**

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



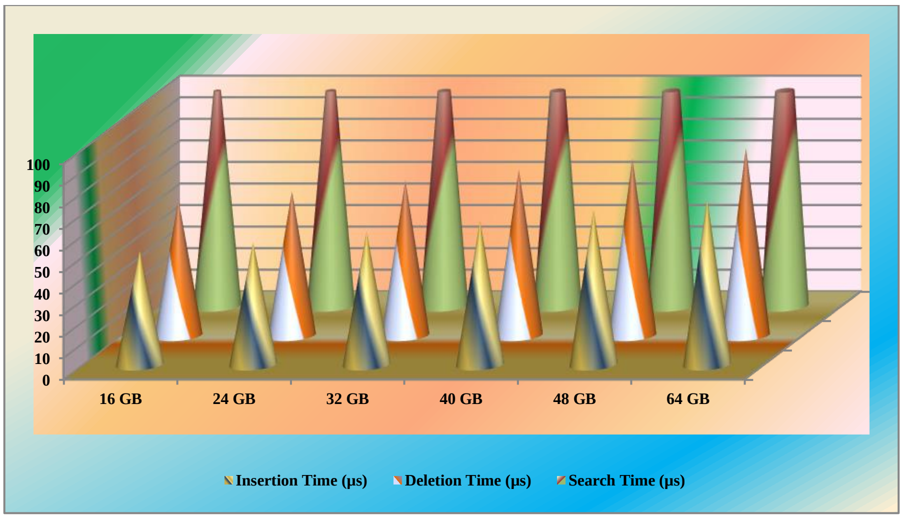
**Graph 21: ETCD – Complexity-1**

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

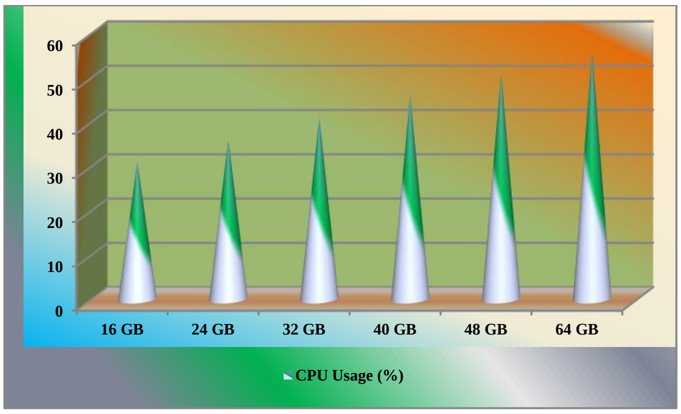
| Store Size | Insertion Time (µs) | Deletion Time (µs) | Search Time (µs) | CPU Usage (%) | Space Complexity | Time Complexity |
|---|---|---|---|---|---|---|
| 16 GB | 53 | 63 | 118 | 31 | O(n) | O(logn) |
| 24 GB | 57 | 67 | 128 | 36 | O(n) | O(logn) |
| 32 GB | 62 | 72 | 138 | 41 | O(n) | O(logn) |
| 40 GB | 67 | 77 | 148 | 46 | O(n) | O(logn) |
| 48 GB | 72 | 82 | 158 | 51 | O(n) | O(logn) |
| 64 GB | 77 | 87 | 168 | 56 | O(n) | O(logn) |

**Table 15: ETCD Parameters – LSM Tree - 2**

As shown in the Table 15, 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 : LSM Tree- 2**



**Graph 23: ETCD – CPU Usage-2**

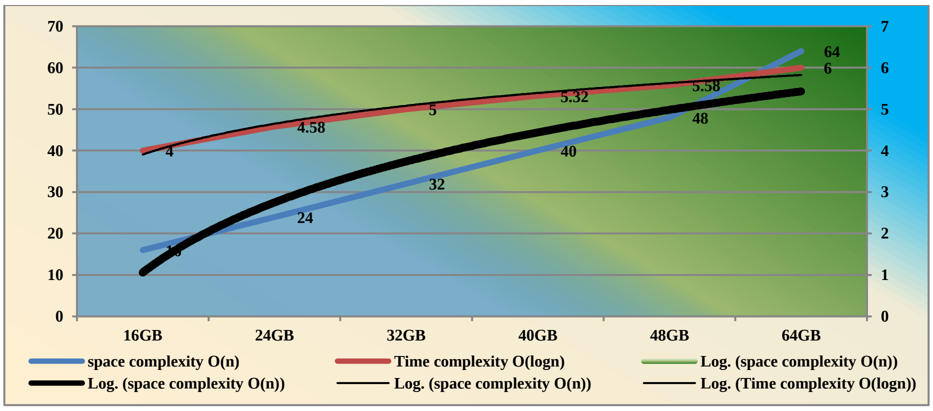
While increasing the size of the key value store , CPU usage also will get increased automatically. Graph 23 shows the same.

| Store Size | space complexity O(nlogBn) | Time complexity O(logBn) |
|---|---|---|
| 16GB | 16 | 4 |
| 24GB | 24 | 4.58 |
| 32GB | 32 | 5 |
| 40GB | 40 | 5.32 |
| 48GB | 48 | 5.58 |
| 64GB | 64 | 6 |

**Table 16: ETCD LSM Tree Complexity-2**

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

Advanced International Journal of Multidisciplinary Research
E-ISSN: 2584-0487 editor@aijmr.com
Volume 2, Issue 5, September-October 2024
CrossRef DOI: 10.62127/aijmr.2024.v02i05.1109



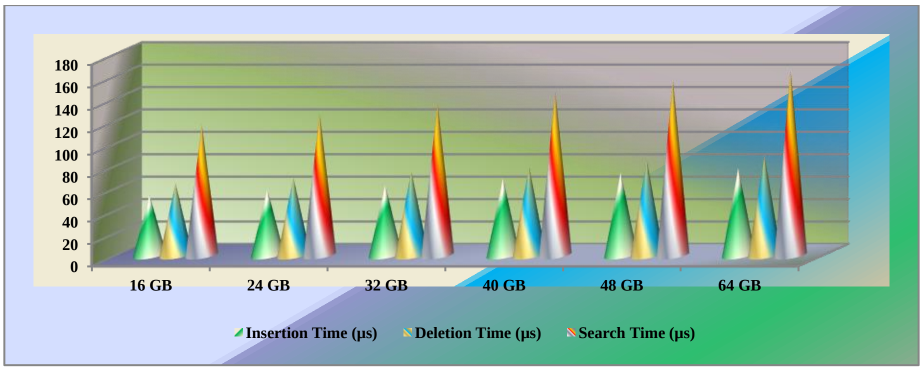
**Graph 24: ETCD – Complexity-2**

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

| Store Size | Insertion Time (µs) | Deletion Time (µs) | Search Time (µs) | CPU Usage (%) | Space Complexity | Time Complexity |
|---|---|---|---|---|---|---|
| 16 GB | 54 | 64 | 116 | 29 | O(n) | O(logn) |
| 24 GB | 59 | 69 | 126 | 33 | O(n) | O(logn) |
| 32 GB | 63 | 75 | 136 | 39 | O(n) | O(logn) |
| 40 GB | 69 | 79 | 146 | 47 | O(n) | O(logn) |
| 48 GB | 74 | 85 | 156 | 50 | O(n) | O(logn) |
| 64 GB | 78 | 89 | 165 | 54 | O(n) | O(logn) |

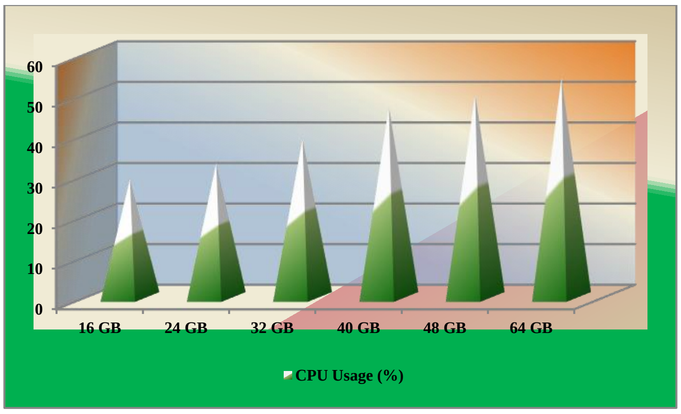
**Table 17 : ETCD Parameters – LSM Tree - 3**

Table 17, 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 : LSM Tree- 3**

Advanced International Journal of Multidisciplinary Research
E-ISSN: 2584-0487 editor@aijmr.com
Volume 2, Issue 5, September-October 2024
CrossRef DOI: 10.62127/aijmr.2024.v02i05.1109

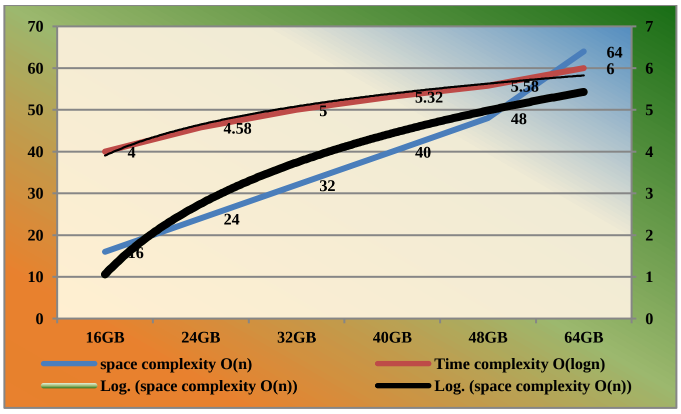


**Graph 26: ETCD – CPU Usage-3**

| 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 18: ETCD LSM Tree Complexity-3**

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



**Graph 27: ETCD – Complexity-3**

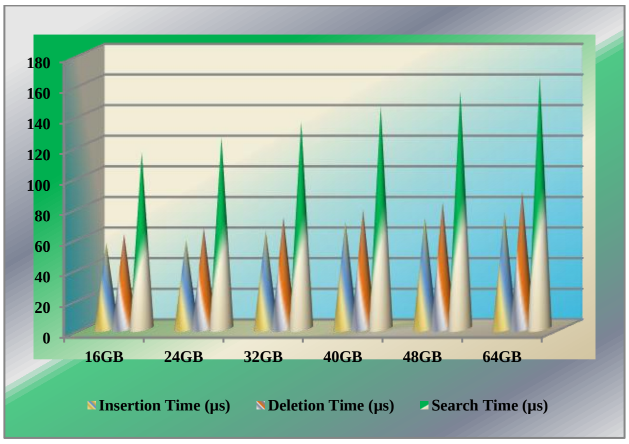
Please find the Logarithmic graph at Graph 27 using the calculation with branch as 4 , O(n) and O(logn) for the n values as 16, 24, 32, 40, 48 and 64.

Advanced International Journal of Multidisciplinary Research
E-ISSN: 2584-0487 editor@aijmr.com
Volume 2, Issue 5, September-October 2024
CrossRef DOI: 10.62127/aijmr.2024.v02i05.1109

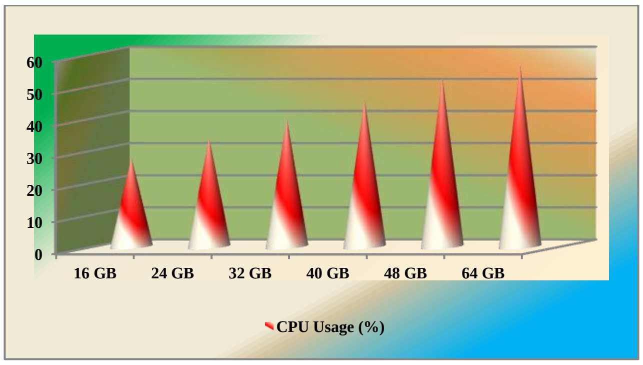
| Store Size | Insertion Time (µs) | Deletion Time (µs) | Search Time (µs) | CPU Usage (%) | Space Complexity | Time Complexity |
|---|---|---|---|---|---|---|
| 16 GB | 56 | 62 | 115 | 28 | O(n) | O(log⁡n) |
| 24 GB | 58 | 66 | 125 | 34 | O(n) | O(log⁡n) |
| 32 GB | 64 | 73 | 135 | 40 | O(n) | O(log⁡n) |
| 40 GB | 70 | 78 | 145 | 46 | O(n) | O(log⁡n) |
| 48 GB | 73 | 83 | 155 | 53 | O(n) | O(log⁡n) |
| 64 GB | 76 | 90 | 165 | 57 | O(n) | O(log⁡n) |

**Table 19: ETCD Parameters LSM – Tree -4**

Table 19 shows the ETCD AVL 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 : LSM Tree- 4**

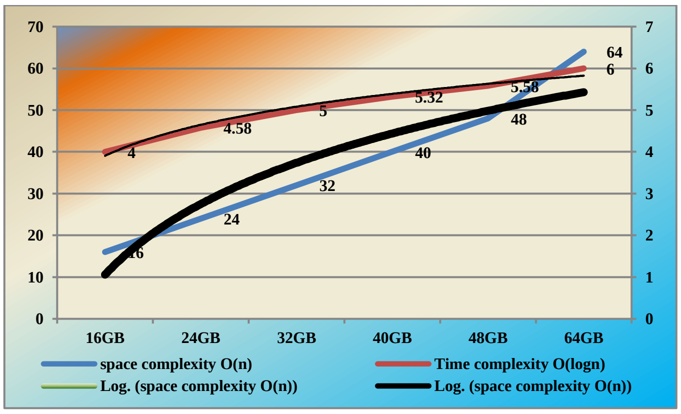


**Graph 29: ETCD – CPU Usage-4**

| 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 20: ETCD LSM Tree Complexity-4**

Table 20 carries the values for Space and Time complexity for LSM Tree implementation of key value store for fourth sample.



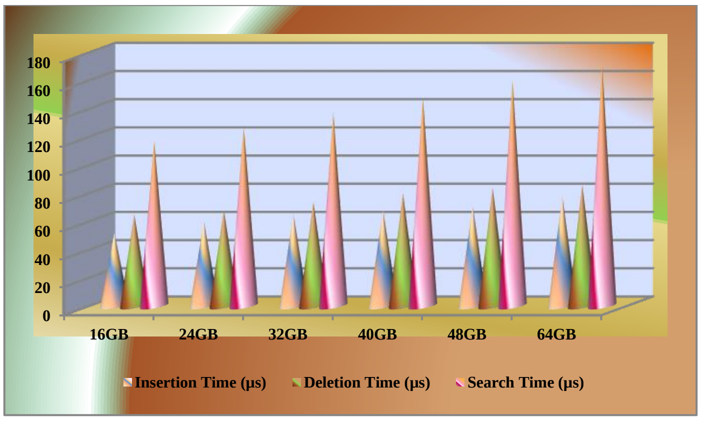
**Graph 30: ETCD – Complexity-4**

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

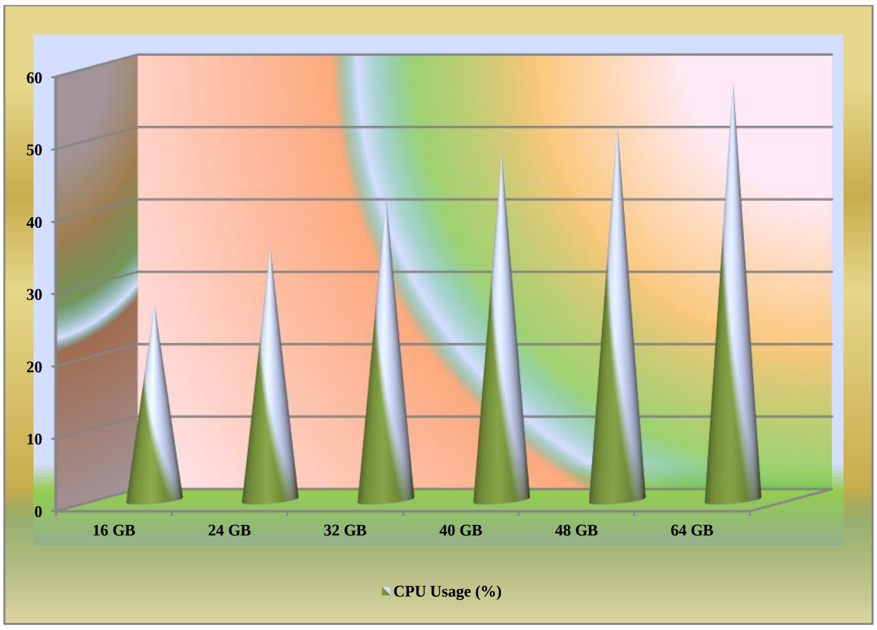
| Store Size | Insertion Time (µs) | Deletion Time (µs) | Search Time (µs) | CPU Usage (%) | Space Complexity | Time Complexity |
|---|---|---|---|---|---|---|
| 16 GB | 52 | 65 | 117 | 27 | O(n) | O(logn) |
| 24 GB | 60 | 68 | 127 | 35 | O(n) | O(logn) |
| 32 GB | 65 | 74 | 137 | 42 | O(n) | O(logn) |
| 40 GB | 68 | 80 | 148 | 48 | O(n) | O(logn) |
| 48 GB | 71 | 84 | 160 | 52 | O(n) | O(logn) |
| 64 GB | 78 | 86 | 171 | 58 | O(n) | O(logn) |

**Table 21: ETCD Parameters – LSM Tree - 5**

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 21 shows the all parameters from the sixth sample.



**Graph 31: ETCD Parameters : LSM Tree- 5**

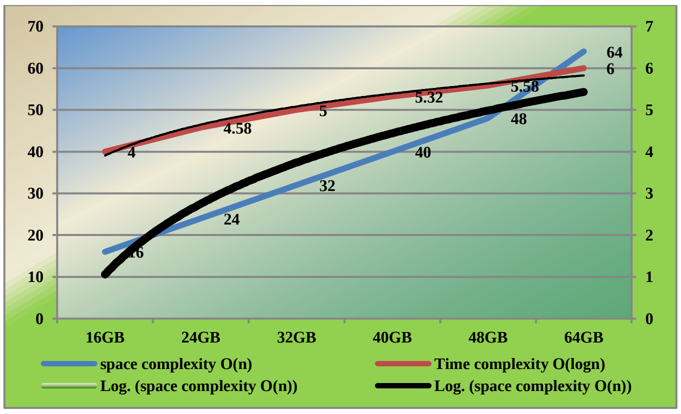


**Graph 32: ETCD – CPU Usage-5**

| 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 22: ETCD LSM Tree Complexity-5**

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



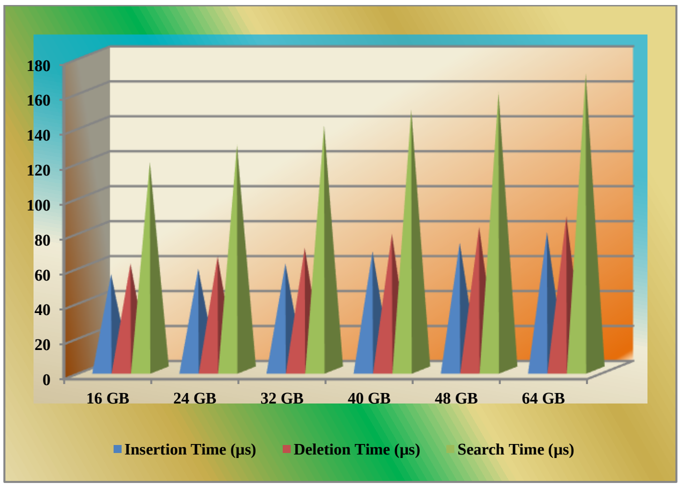
**Graph 33: ETCD – Complexity-5**

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

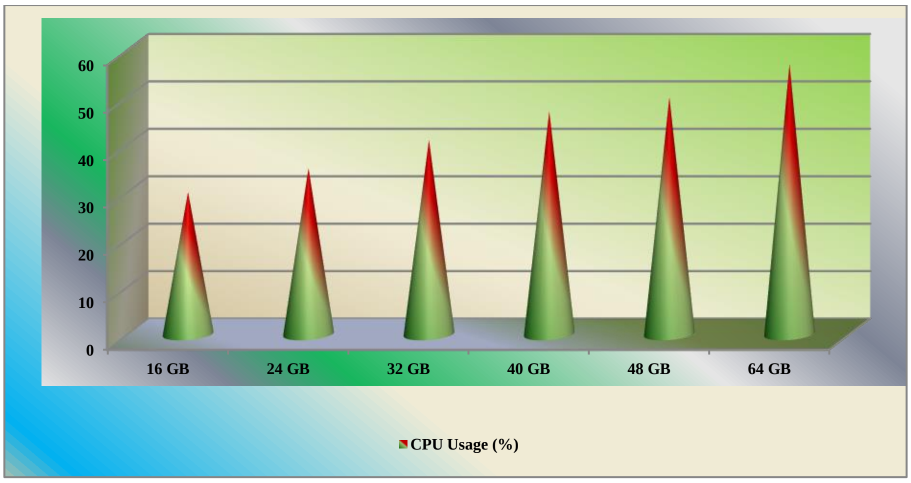
| Store Size | Insertion Time (µs) | Deletion Time (µs) | Search Time (µs) | CPU Usage (%) | Space Complexity | Time Complexity |
|---|---|---|---|---|---|---|
| 16 GB | 55 | 61 | 119 | 30 | O(n) | O(logn) |
| 24 GB | 58 | 65 | 129 | 35 | O(n) | O(logn) |
| 32 GB | 61 | 70 | 140 | 41 | O(n) | O(logn) |
| 40 GB | 68 | 78 | 149 | 47 | O(n) | O(logn) |
| 48 GB | 73 | 82 | 159 | 50 | O(n) | O(logn) |
| 64 GB | 79 | 88 | 170 | 57 | O(n) | O(logn) |

**Table 23: ETCD Parameters LSM Tree -6**

Table 23 carries the values for LSM implementation of ETCD parameters like insertion time, deletion time, search time.



**Graph 34: ETCD Parameters : LSM Tree- 6**

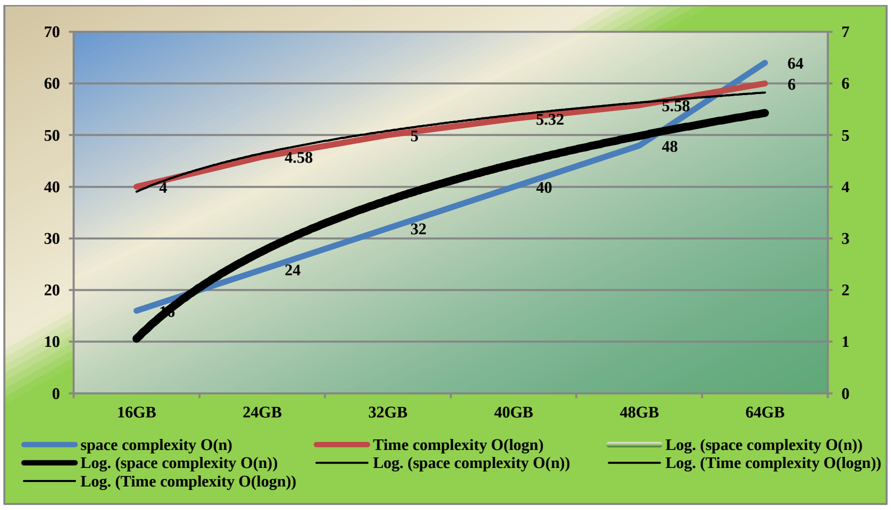


Graph 35: ETCD – CPU Usage-6

| 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 |

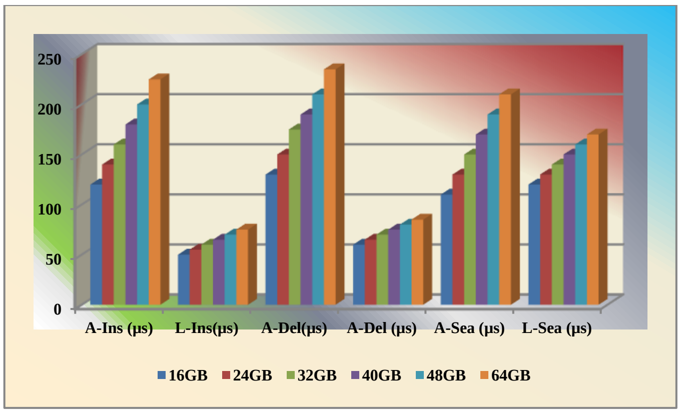
Table 24: ETCD LSM Tree Complexity-6

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



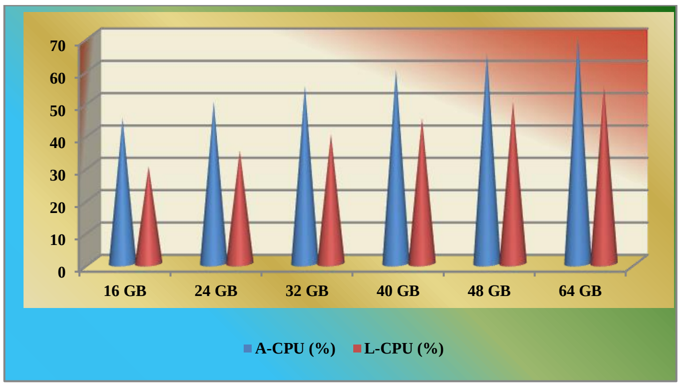
Graph 36: ETCD – Complexity-6

Please find the Logarithmic graph at Graph 36 using the calculation with branch as 4 , O(n) and O(logn) for the n values as 16, 24, 32, 40, 48 and 64 .



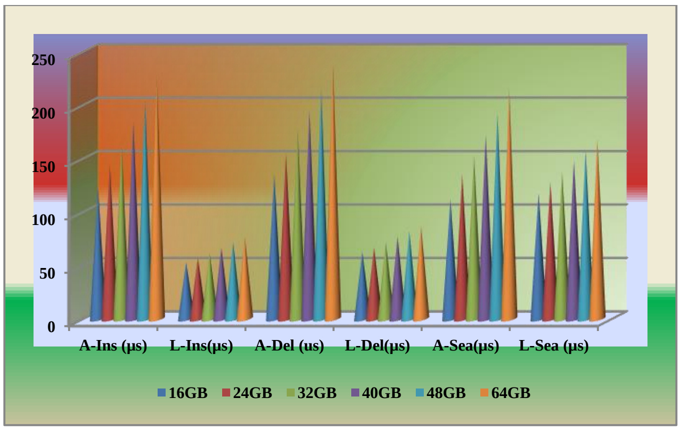
**Graph 37: ETCD AVL Vs LSM Tree-1.1**

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



**Graph 38: ETCD AVL Vs LSM Tree-1.2**

Graph 38 shows the CPU usage difference between AVL implementation and LSM Tree implementation. CPU usage is going low once we are dealing with LSM in the implementation.

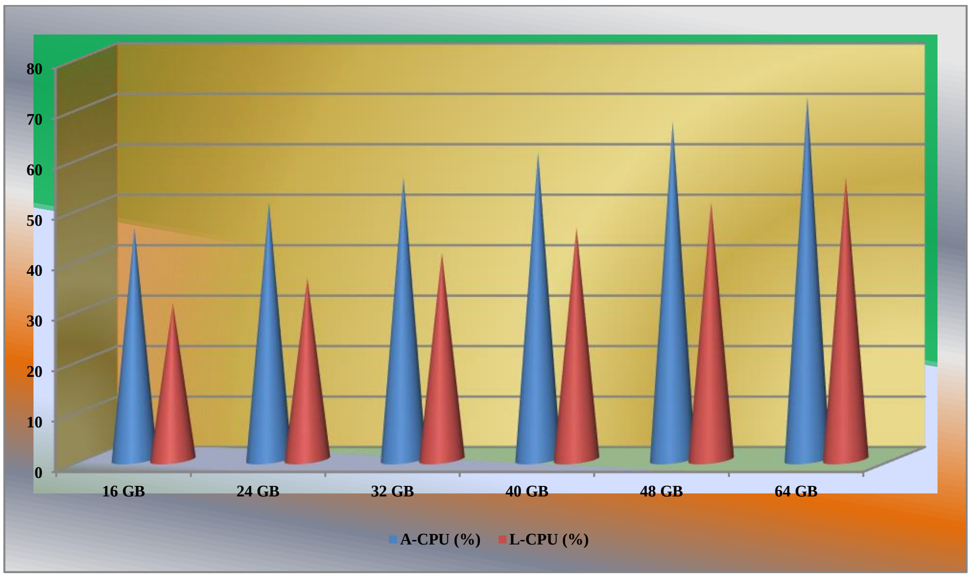


**Graph 39: ETCD AVL Vs LSM Tree-2.1**

Graph 39, is the comparison between AVL and LSM Tree implementation of the key value store (ETCD). The graph shows the Avg Insertion time difference between AVL and LSM Tree implementation. As per the graph the time trend is going down as move from AVL to LSM Tree

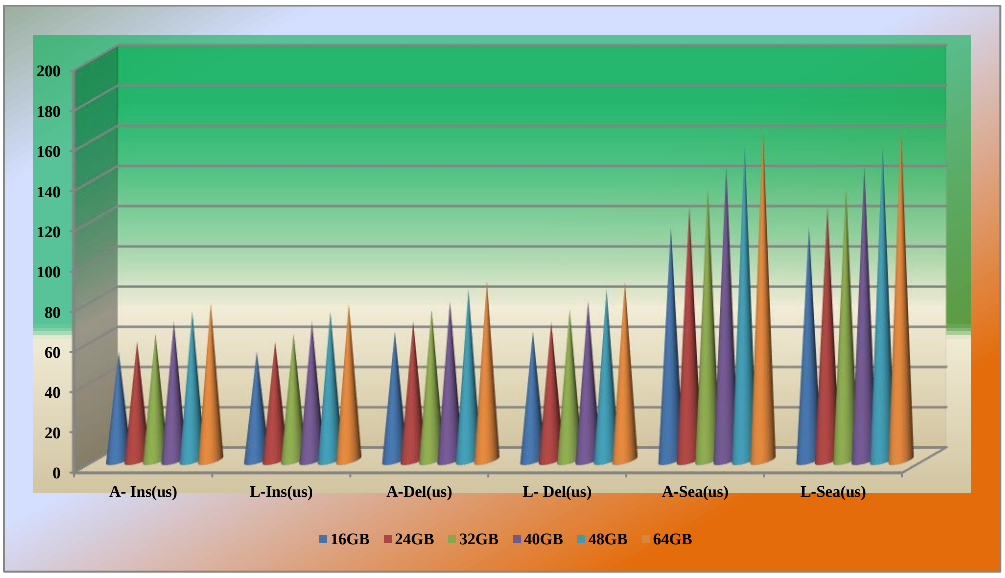
Advanced International Journal of Multidisciplinary Research
E-ISSN: 2584-0487 editor@aijmr.com
Volume 2, Issue 5, September-October 2024
CrossRef DOI: 10.62127/aijmr.2024.v02i05.1109

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



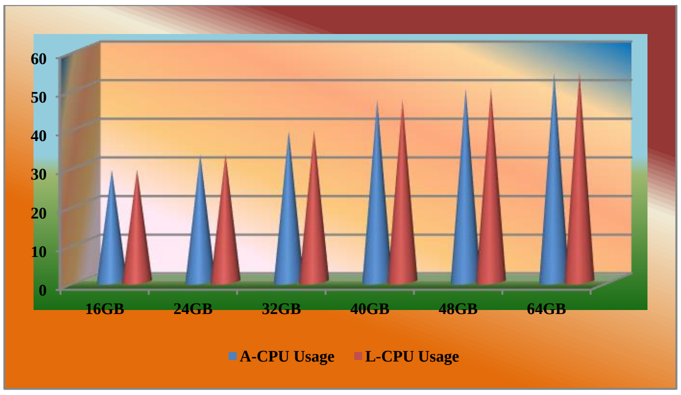
**Graph 40: ETCD AVL Vs LSM Tree-2.2**

Graph 40 shows the CPU usage difference between AVL implementation and LSM Tree implementation. The CPU usage also going down once we started using the LSM implementation of the ETCD store.

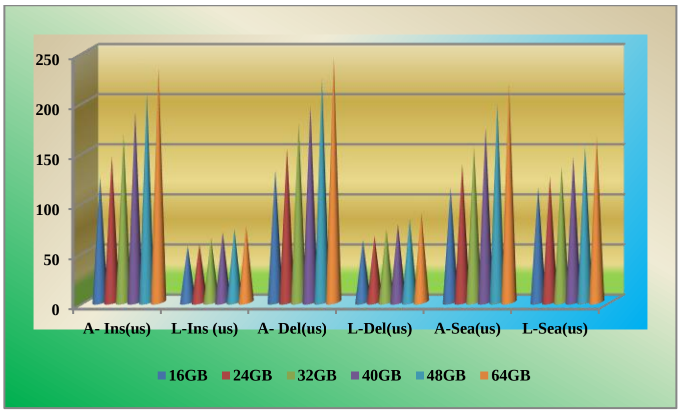


**Graph 41: ETCD AVL Vs LSM Tree-3.1**

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

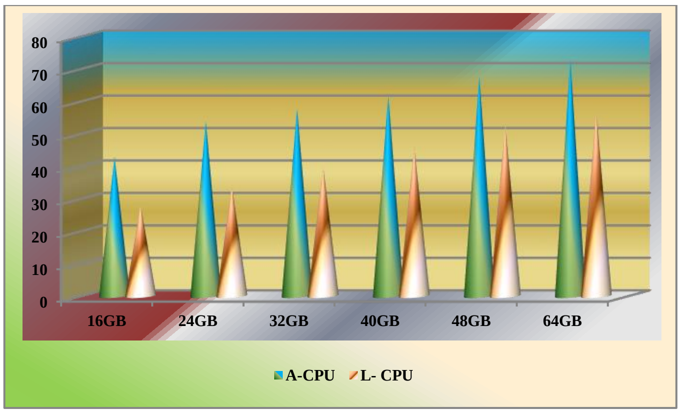


**Graph 42:** ETCD AVL Vs LSM Tree-3.2



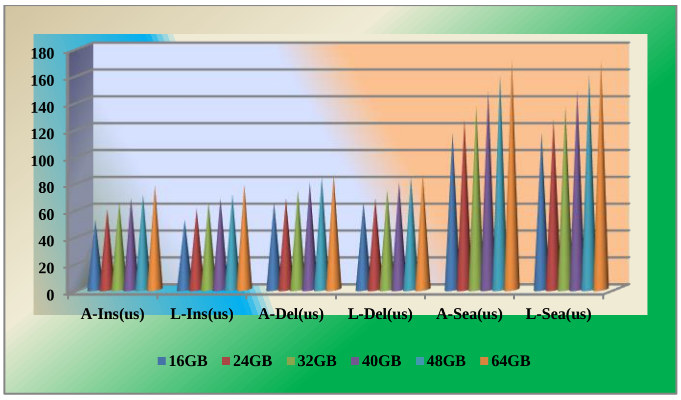
**Graph 43: ETCD AVL Vs LSM Tree-4.1**

Graph 43, is the comparison between AVL and LSM 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 AVL and LSM Tree implementation. As per the graph the time trend is going down as move from AVL to LSM Tree implementation. The same observation we can have with other parameters like deletion time and search time.



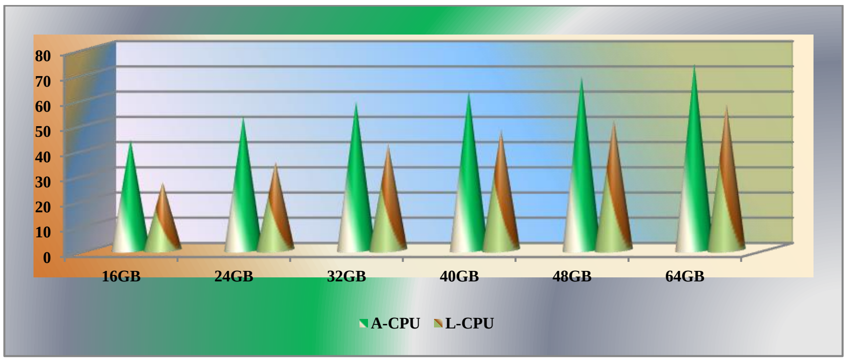
**Graph 44: ETCD AVL Vs LSM Tree-4.2**

Graph 44 shows the CPU usage difference between AVL implementation and LSM Tree implementation. The cpu usage is going down once we start using the LSM implementation of the key value store.



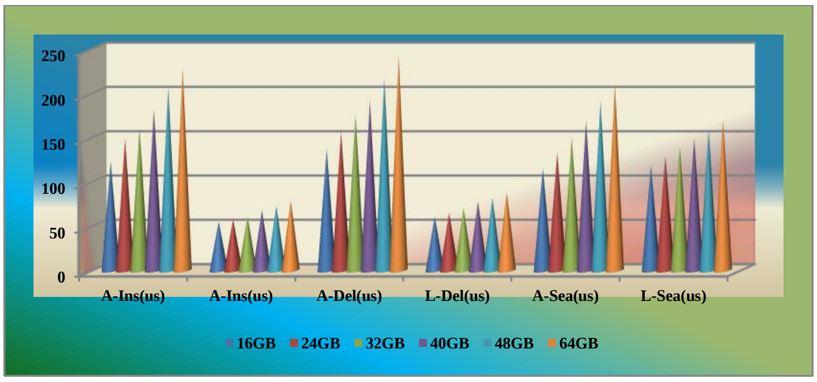
**Graph 45: ETCD AVL Vs LSM Tree-5.1**

Graph 45, is the comparison between AVL and LSM Tree implementation of the key value store (ETCD) for the third fifth. The graph shows the  Insertion time difference between AVL and LSM Tree implementation. As per the graph the time trend is going  down as move from AVL to LSM Tree implementation. The same observation we can have with other parameters like deletion time and search time.



**Graph 46: ETCD AVL Vs LSM Tree-5.2**

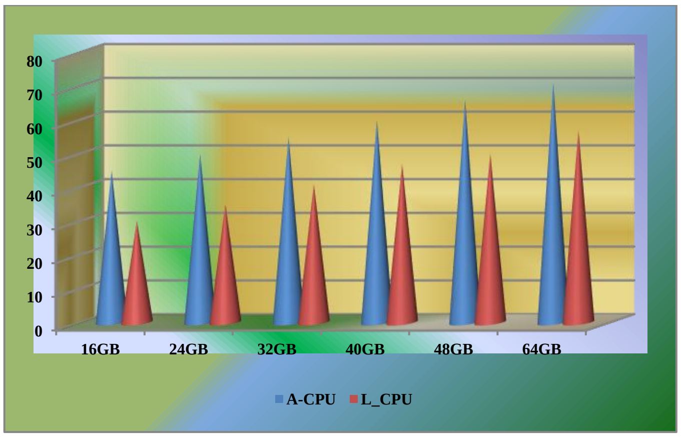
Graph 46 shows the CPU usage difference between AVL implementation and LSM Tree implementation. LSM implementation is using less cpu compared to AVL implementation. So this analysis is positive to proceed further with LSM implementation of key value store (ETCD).



**Graph 47: ETCD AVL Vs LSM Tree-6.1**

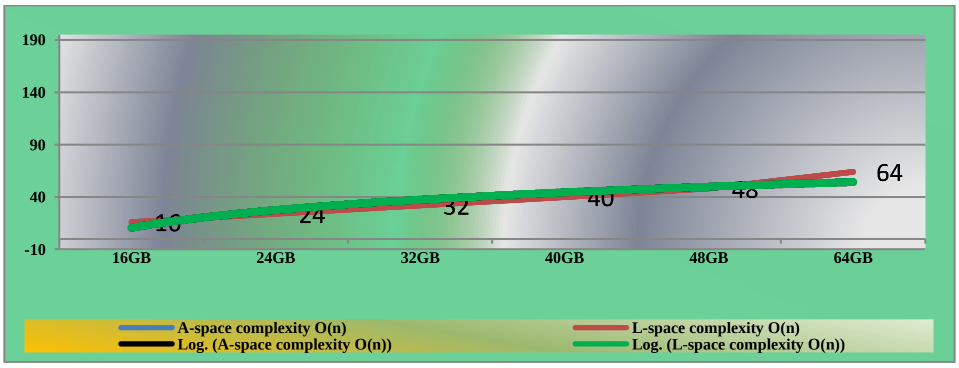
Advanced International Journal of Multidisciplinary Research
E-ISSN: 2584-0487 editor@aijmr.com
Volume 2, Issue 5, September-October 2024
CrossRef DOI: 10.62127/aijmr.2024.v02i05.1109

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

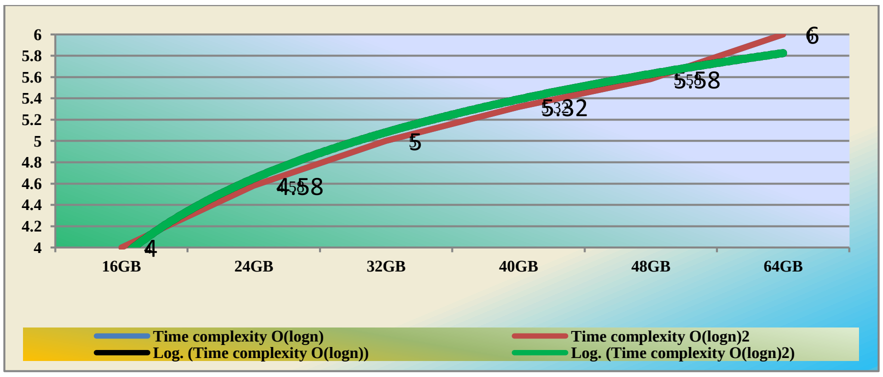


**Graph 48: ETCD AVL Vs LSM Tree-6.2**

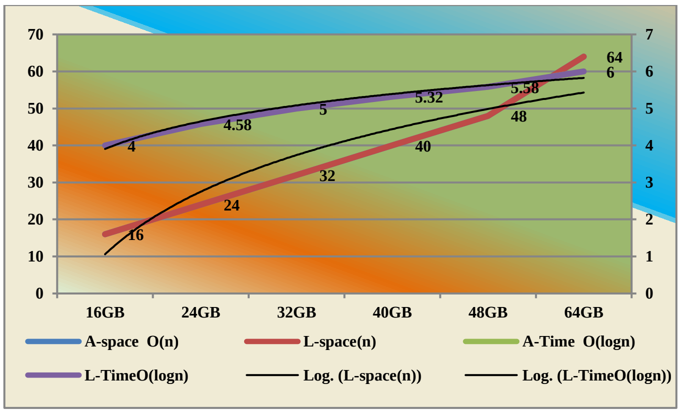
Graph 48 shows the CPU usage difference between AVL implementation and LSM Tree implementation. ETCD is consuming less CPU once we have LSM implementation of the same. AVL implementation is consuming bit high compared to LSM implementation.



**Graph 49: ETCD AVL Vs LSM Tree- Space Complexities**



**Graph 50: ETCD AVL Vs LSM Tree- Time Complexities**



**Graph 51: ETCD AVL Vs LSM Time and Space complexities**

Graph 49 , 50 and 51 shows the comparison of complexities between AVL and LSM Tree implementation. We can conclude that by using the LSM implementation of the ETCD is better than using the AVL implementation.

## EVALUATION

The comparison of AVL implementation results with LSM 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 have been observed. As per the analysis carried out so far in this states that insertion time , deletion time, and search time are going down if u start using the implementation of the Data Store (ETCD) using the LSM Tree instead of AVL.

## 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 operations using the metrics collection code. We have collected six samples on etcd operations like insertion , deletion , search . All these activities are performing better in the LSM Tree implementation compared to AVL implementation. Space complexity and time complexity are also compared, along with this CPU usage . Complexities are almost same , while CPU usage values are going down.
Future work includes working on Log Structured Hash Table (LSHT) which will have less complexity than Log Structured Merge Tree (LSM).

## 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, Einar Broch Johnsen, S. LizethT apiaTarifa, 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.
41. AVL and Red Black tree as a single balanced tree, March 2016, Zegour Djamel Eddine, Lynda Bounif
42. The log-structured merge-tree (LSM-tree),June 1996, Patrick O'Neil, Edward Cheng, Dieter Gawlick & Elizabeth O'Neil.