Adelson-Velsky Landis Trees and Btrees for Kubernetes Etcd Implementation

Ravi Kiran Jakkilinki[1], Dr.B. Purnachandrarao[2]

[1]Monster Worldwide, Inc., CA, USA
[2]Sr. Solutions Architect, HCL Technologies , Bangalore, Karnataka, Indi

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, and 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. Whenever 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 Landis and BTree. BTree outperforms Adelson Velsky Landis Trees in some scenarios. We will work on to prove that BTree implementation provides better performance than Adelson Velsky Landis Tree.

Keywords: Kubernetes (K8S), Cluster, Nodes, Deployments, Pods, ReplicaSets, Statefulsets, Service, IP-Tables, Load Balancer, Service Abstraction, Adelson-Velsky and Landis (AVL), BTree, 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

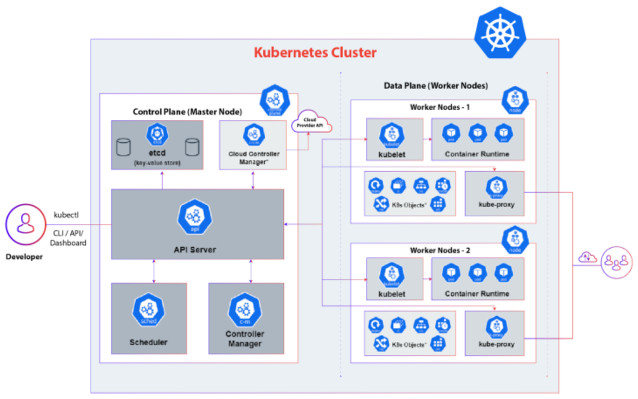


Fig 1. Shows the Kubernetes cluster architecture. This shows two worker nodes and one control plane. Control plane is having four components API Server , Scheduler , Controller and ECTD. Pods are deployed to nodes using scheduler. Client kubectl will connect to API server (part of Master Node) to interact 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.

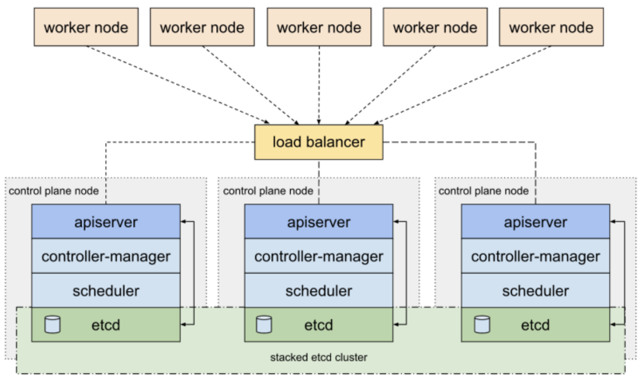


Fig 2: ETCD Architecture

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.
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 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 which is making the deployment [19][41] platform without any hassles. Three node , four node , five node , six node , seven node , eight node , nine node and ten node clusters have been configured with 32 CPU, 64 GB and 500GB for master node and 24 CPU , 32 GB and 350 GB for all worker nodes. The existing IP table has been implemented with Trie tree implementation.
A Trie Tree, also known as a Prefix Tree, is a specialized tree data structure used to store associative data structures, often to represent strings. The key characteristic of a Trie is that all descendants of a node share a common prefix of the string associated with that node. This structure is particularly useful for tasks that involve searching for prefixes, such as auto complete systems, dictionaries, and IP routing tables.

```
package main
import (
"fmt"
"time"
"runtime"
)
type AVLNode struct {
key    int
left   *AVLNode
right  *AVLNode
height int
}
type AVLTree struct {
root *AVLNode
}
func height(node *AVLNode) int {
if node == nil {
return 0
}
return node.height
}
func rightRotate(y *AVLNode) *AVLNode {
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 *AVLNode) *AVLNode {
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 *AVLNode) int {
if node == nil {
return 0
}
return height(node.left) - height(node.right)
}
func (t *AVLTree) insert(key int) {
t.root = insertNode(t.root, key)
}
func insertNode(node *AVLNode, key int) *AVLNode {
if node == nil {
return &AVLNode{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 *AVLTree) search(key int) bool {
return searchNode(t.root, key)
}

func searchNode(node *AVLNode, 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 *AVLTree, 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 := &AVLTree{}
keys := []int{10, 20, 30, 40, 50, 25}

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

An AVL 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 AVL tree.
The Height-Balancing property is after each insertion or deletion operation, the tree automatically performs rotations to maintain its balanced height. This is important because a regular BST can become unbalanced, degrading into a linked list (with time complexity O(n) for operations), And the rotations are when a node becomes unbalanced after insertion or deletion, rotations are applied to restore balance.
Single Rotations: Left Rotation (for right-heavy imbalance) and Right Rotation (for left-heavy imbalance) and the double rotations is Left-Right Rotation and Right-Left Rotation, used when an imbalance cannot be fixed by a single rotation. The performance is due to self-balancing, the AVL tree maintains O(log n) time complexity for insertion, deletion, and search operations, making it efficient for applications where frequent searching and dynamic insertions/deletions are required.
The AVL Node and AVLTree structures have been defined. The height function is defined based on the number of nodes in the tree. A right rotation is applied when the left subtree of a node is heavier (taller) than the right subtree, causing a left-heavy imbalance. The right rotation will lift the left child of the unbalanced node, making it the new root of the subtree.   z   y

```
   / \                       /  \
  y   T3    ----->             x     z
 / \                         / \   / \
x   T2                        T1  T2 T3 T4
/ \
T1 T2
```

Let y be the left child of z. Move the right subtree of y to the left subtree of z (if y has a right subtree). Make y the new root of this subtree. Set z as the right child of y.

A left rotation is used when the right subtree of a node is heavier (taller) than the left subtree, causing a right-heavy imbalance. The left rotation will lift the right child of the unbalanced node, making it the new root of the subtree.

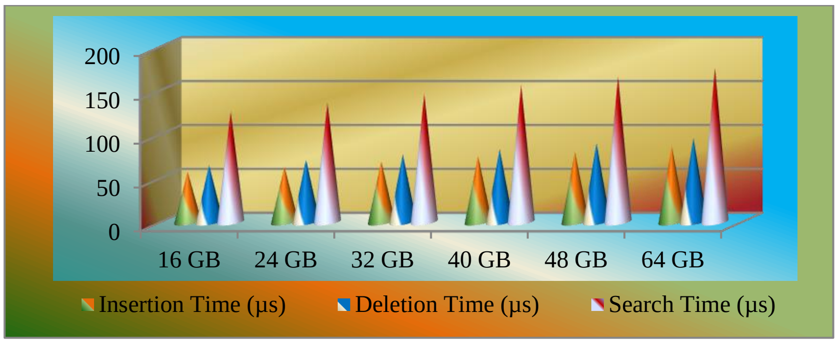
```
    z                        y
   / \                      /  \
  T1  y      ----->            z     x
     / \                   / \   / \
    T2  x                   T1  T2 T3 T4
       / \
      T3 T4
```

Consider a node z that has become unbalanced due to a right-heavy subtree: Let y be the right child of z.
Move the left subtree of y to the right subtree of z (if y has a left subtree).
Make y the new root of this subtree. Set z as the left child of y.
When a single rotation is insufficient to balance the tree, double rotations are used. There are two types of double rotations. Left-Right Rotation: First, a left rotation is applied on the left child of the node, followed by a right rotation on the node itself.
Right-Left Rotation: First, a right rotation is applied on the right child of the node, followed by a left rotation on the node itself.
After inserting or deleting a node in an AVL tree, these rotations are used to adjust the balance factor, ensuring the height difference between the left and right subtrees of any node remains within the allowed range (-1, 0, or +1). This ensures the tree maintains a balanced state and efficient complexity [20][28] O(log n) performance. Insert Node , delete node and search operations have been defined and referenced in main function.

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)

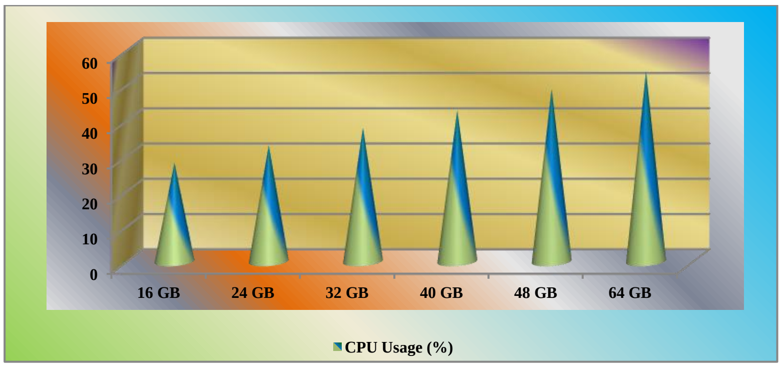
Table 1: ETCD Parameters : AVLTree-1

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 : AVL Tree- 1

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



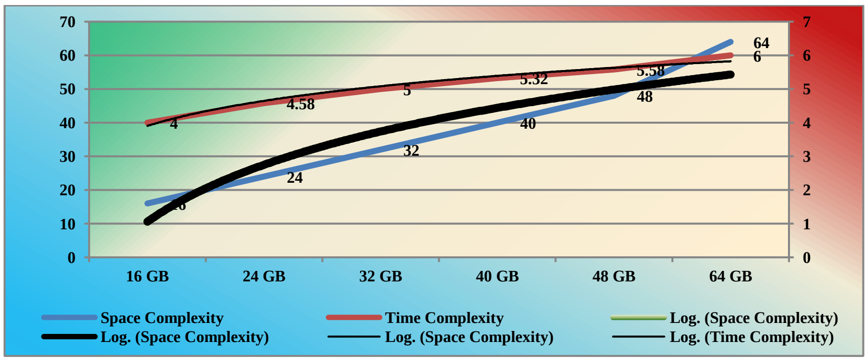
Graph 2: ETCD – AVL CPU Usage-1

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

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 AVL Tree Complexity-1

AVL implementation is having the space and time complexity as O(n) and O(logn) , where ni is the number of entries in the data store. Table 2 carries the same values from the first sample of ETCD AVL implementation.



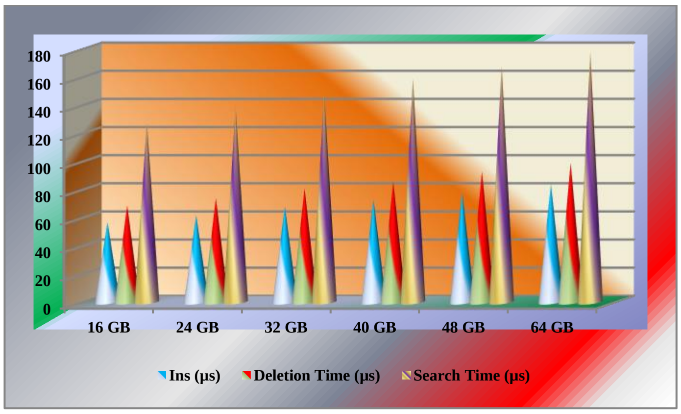
Graph 3: ETCD AVL Tree Complexity-1

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. Graph 3 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	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)

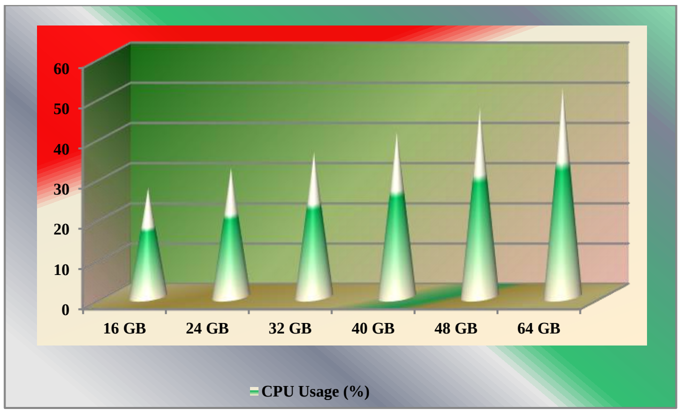
Table 3: ETCD Parameters : AVL Tree-2

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 had in the second sample.



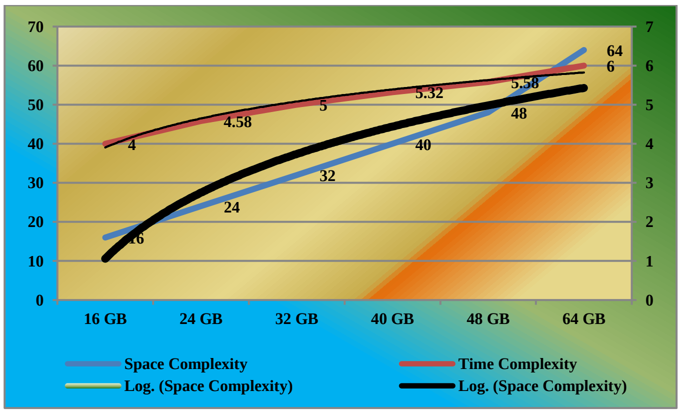
Graph 5: ETCD – CPU Usage-2

Graph 5 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.

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 AVL Tree Complexity-2

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



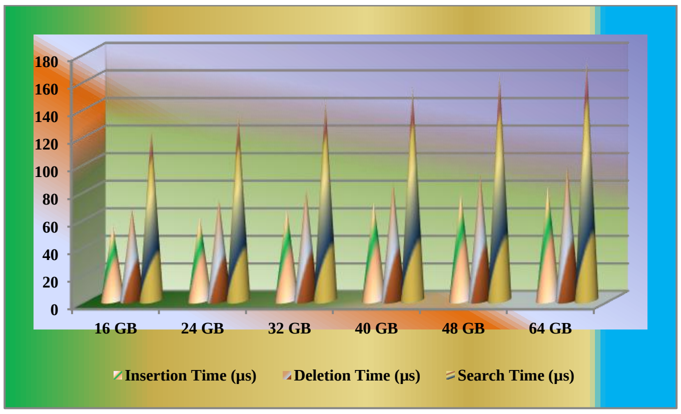
Graph 6: ETCD AVL Tree Complexity-2

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. Graph 6 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	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)

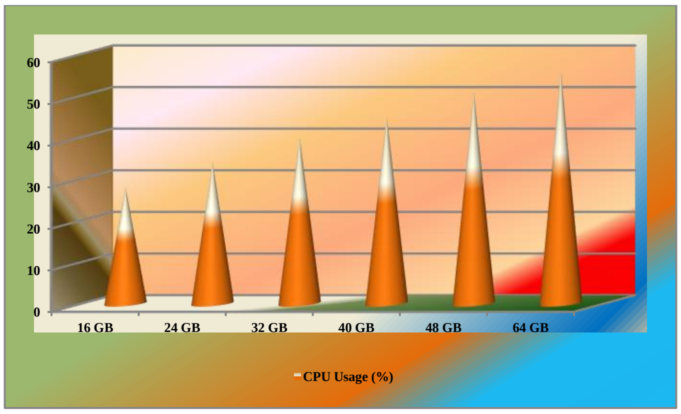
Table 5: ETCD Parameters – AVL Tree-3

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 7 shows the insertion , deletion, search times which we have had in the third sample.



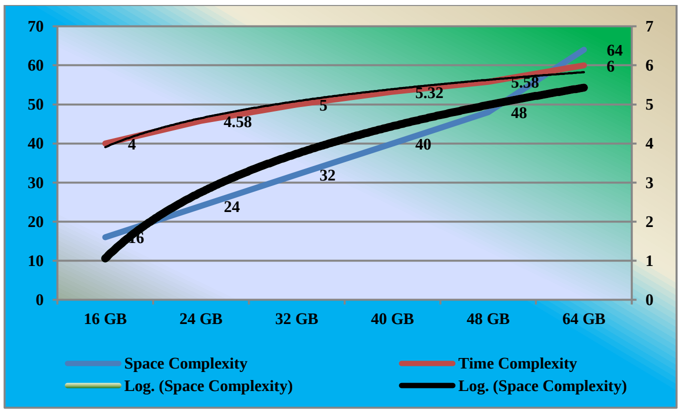
Graph 8: ETCD – CPU Usage-3

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 AVL Complexity-3

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



Graph 9: ETCD AVL Tree Complexity-3

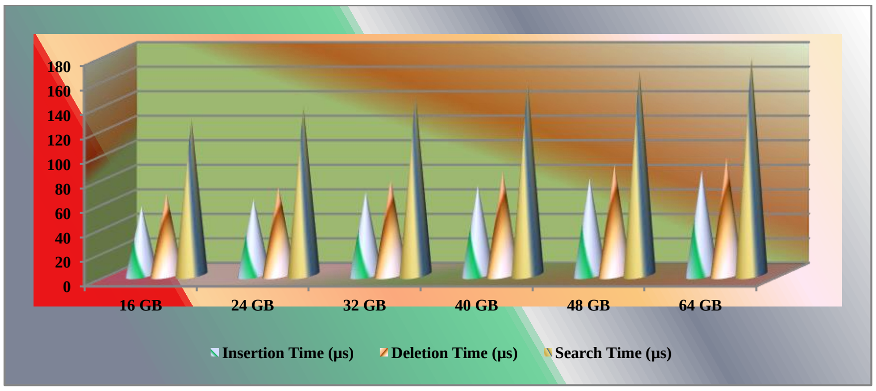
Please find the Logarithmic graph using the calculation, O(1) = 1, O(log n) ≈ 4 (using base 2 logarithm), O(n) = 16, 24, 32, 40, 48 and 64 for the n values from the size of the store which we have mentioned in the table. Graph 9 shows the same values. It is using two scale Y-Axis since the table is carrying 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.

Store Size	Ins(μs)	Del (μs)	Sea (μs)	CPU(%)	S-Comp	T-Comp
16 GB	56	66	127	27	O(n)	O(log n)
24 GB	62	72	137	32	O(n)	O(log n)
32 GB	68	78	147	38	O(n)	O(log n)
40 GB	73	84	157	43	O(n)	O(log n)
48 GB	79	90	167	49	O(n)	O(log n)
64 GB	85	96	177	54	O(n)	O(log n)

Table 7: ETCD Parameters – AVL Tree- 4

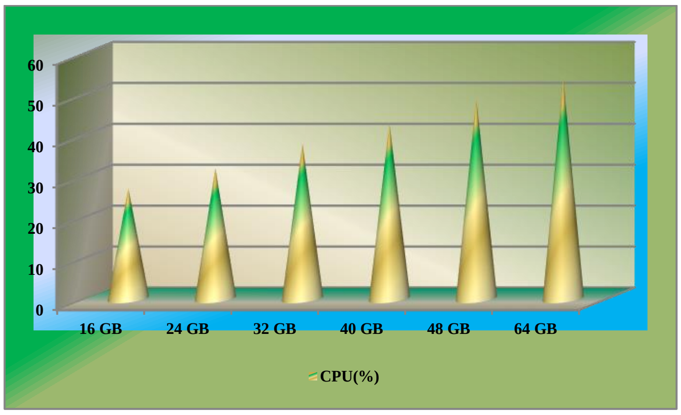
Table 7, shows the fourth sample of the data from ETCD store. ETCD Stores a key-value pair in etcd, Syntax: etcdctl put <key> <value>, etcdctl put message "Hello, world!"

- 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

Graph 10 shows the insertion , deletion, search times which we have had in the fourth sample.



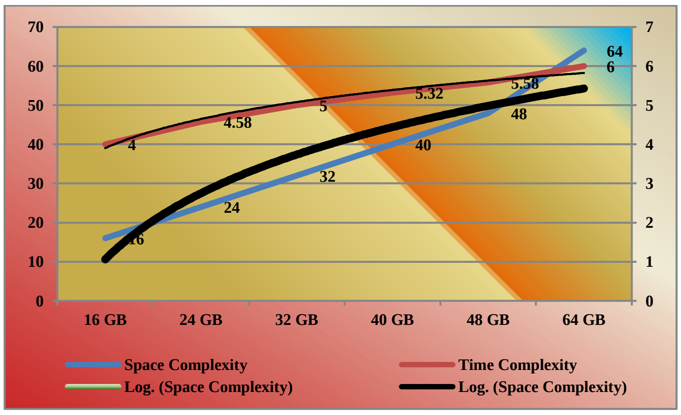
Graph 11: ETCD – CPU Usage-4

Graph 10 shows the 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.



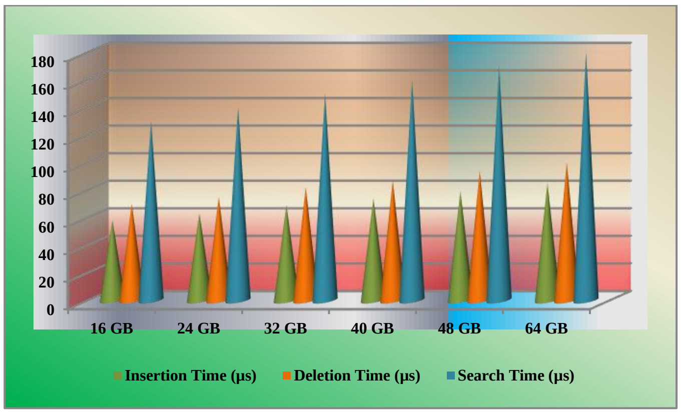
Graph 12: ETCD – Complexity-4

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. Graph 12 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	58	70	130	28	O(n)	O(log n)
24 GB	63	75	140	33	O(n)	O(log n)
32 GB	69	82	150	37	O(n)	O(log n)
40 GB	74	87	160	42	O(n)	O(log n)
48 GB	79	94	170	48	O(n)	O(log n)
64 GB	85	100	180	53	O(n)	O(log n)

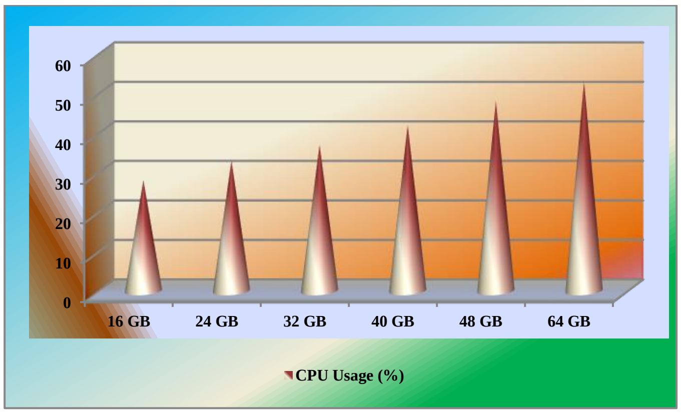
Table 9: ETCD Parameters – AVL Tree – 5

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 13 shows the carries the insertion time, deletion time, search time from the fifth sample of the AVL implementation of the key value store (ETCD).



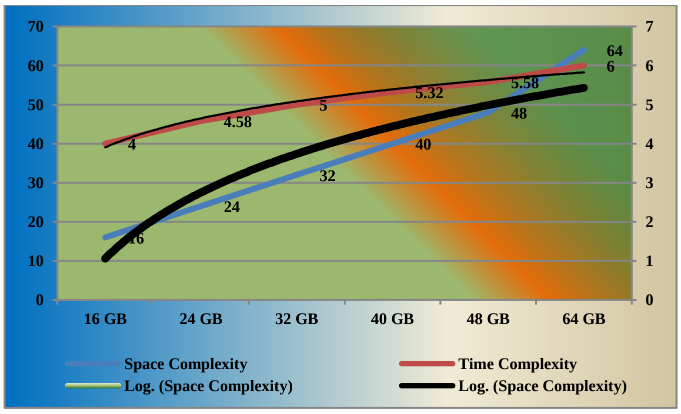
Graph 14: ETCD – CPU Usage-5

Graph 14 shows CPU usage from the fifth sample. It is going high when we start increasing the data store size.

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 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. Since the space complexity is O(n) , the entry size carries at the space complexity, where as at the time complexity values are equal to O(logn).



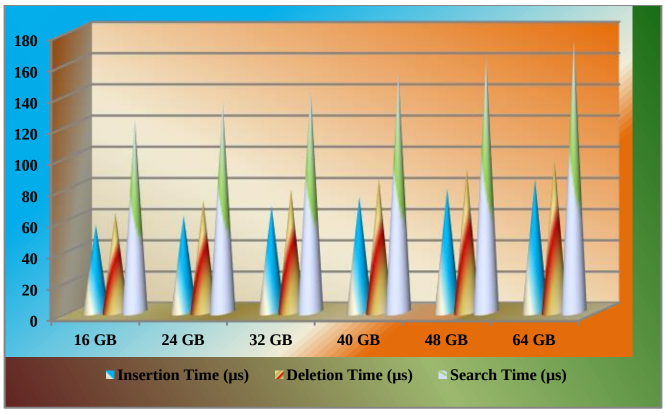
Graph 15: ETCD – Complexity-5

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. 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	56	64	124	27	O(n)	O(log n)
24 GB	62	72	134	33	O(n)	O(log n)
32 GB	68	79	144	39	O(n)	O(log n)
40 GB	74	86	154	45	O(n)	O(log n)
48 GB	79	92	164	51	O(n)	O(log n)
64 GB	85	98	174	56	O(n)	O(log n)

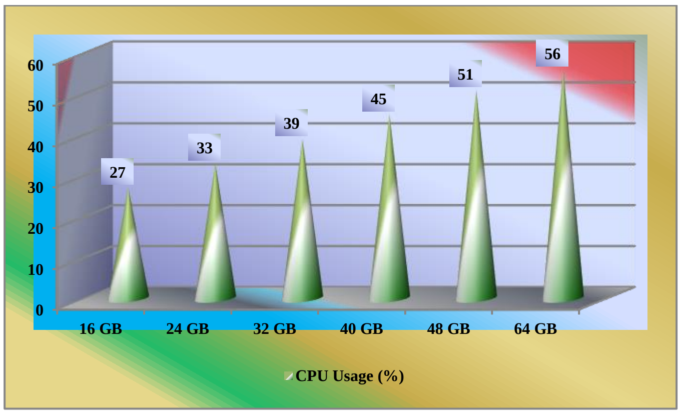
Table 11: ETCD Parameters – AVL Tree – 6

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 16 shows the AVL ETCD operations parameters like insertion time , deletion time , search time in micro seconds.



Graph 17: ETCD – CPU Usage-6

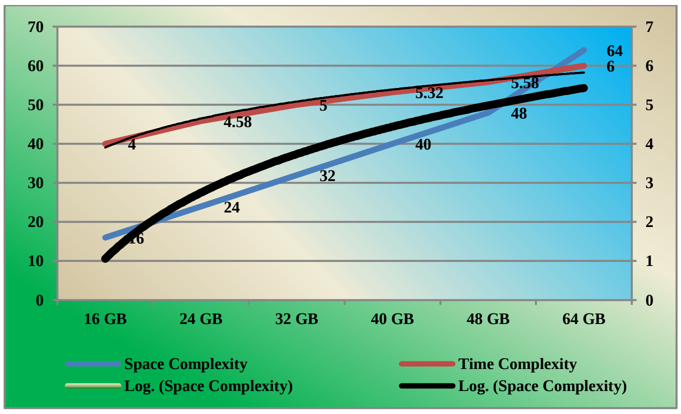
Graph 16 and 17 shows the parameters from the sixth sample. Insertion time, deletion time, search time shows in micro seconds where as CPU usage is in %. As usual the values are going high while increasing the size of the data store. Space complexity is same O(n) for all the sizes of the data store.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

Space complexity is O(n) , so the table size carries at the space complexity, where as time complexity is O(logn), so the logarithmic values are available.



Graph 18: ETCD – Complexity-6

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.

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 B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. Unlike binary trees, a B-tree node can have more than two children, making it well-suited for systems that read and write large blocks of data (like databases and file systems). B-trees are designed to minimize disk reads and writes, making them highly efficient for storage access in external memory.

Key Characteristics of a B-tree are Node Structure, Order, Height-Balancing, Splitting and Merging Nodes, Applications and Time Complexity. Node Structure [21][30][42], Each node in a B-tree contains multiple keys (data elements) and child pointers. The number of keys in a node is within a pre-defined range, which helps maintain balance. Order, The order of a B-tree, often denoted as m, is the maximum number of children each node can have. Each node in a B-tree of order m contains at most m-1 keys and m children. Height-Balancing, B-trees are height-balanced. All leaves are at the same depth, ensuring efficient search operations.

They tend to have fewer levels (height) than binary trees, allowing for faster access times in large datasets. Splitting and Merging Nodes, When a node becomes full, it splits, redistributing keys to maintain the tree’s balance. If nodes become too empty after deletions, they may merge with neighboring nodes to preserve the balance of the tree. Applications, B-trees are widely used in databases and file systems where large amounts of data need to be indexed and accessed quickly. They are suitable for storage systems due to their minimal need for disk I/O operations.

Time Complexity, Search, Insert, Delete: O(log n), The logarithmic time complexity results from the balanced nature and the broad branching factor of the B-tree.

```
        [5 | 10]
       /   |    \
  [1 | 2]  [6 | 7 | 8]  [12 | 15 | 18]
```

A B-tree of order 3 can have, A minimum of 1 and a maximum of 2 keys in each node. A minimum of 2 and a maximum of 3 children per node. The root has two keys (5 and 10) and three children. Each child node also has its own keys, helping to keep the tree balanced.
Advantages of B-trees are Efficient Disk Access, Designed for storage, B-trees minimize the number of disk reads [22][39]. Balanced Structure, Keeps the tree balanced, ensuring logarithmic search, insert, and delete times. Scalability: Allows for efficient scaling by managing large data blocks, unlike binary trees with more frequent rebalancing needs. In summary, B-trees are optimized for high-performance data storage and retrieval in environments where data is too large to fit entirely in memory, making them fundamental in database indexing and filesystem management [23][37][34].
Using BTree we will implement the Data Store ETCD , and will perform all these operations like insertion of the key, deletion of the key, search time, CPU usage[25][26], 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 BTREE 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"
"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 implementation of a BTree focuses on creation of node structure and structure of a tree. Insert key , delete key and search operations have been implemented. The main function is referring all these functions.

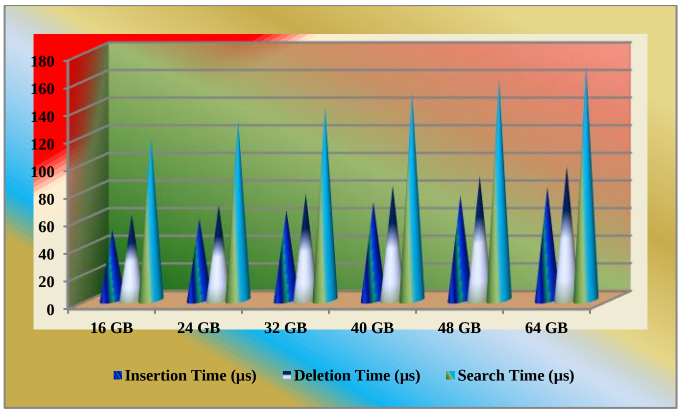
The test code collects performance metrics for the BTRee implementation of ETCD [29][32] ,focusing on insertion time, deletion time, search time, CPU usage, space complexity, and time complexity.

Space Usage: Go's runtime.MemStats structure [43][44] helps retrieve memory allocations specifically related to the BTree 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)

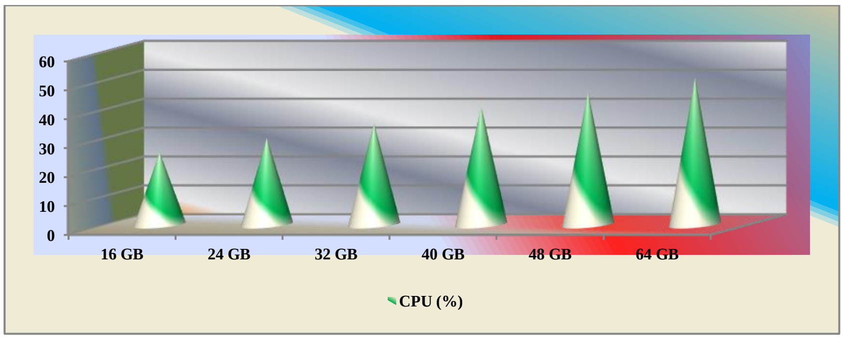
Table 13: ETCD Parameters – BTRee -1

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

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



Graph 20: ETCD – CPU Usage-1

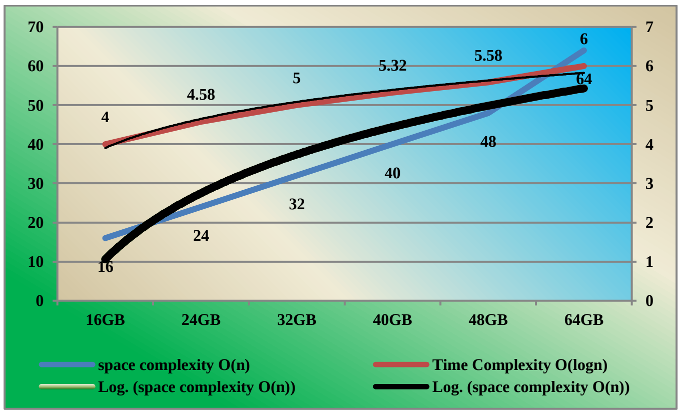
Graph 20 shows the CPU usage of the ETCD data store having the BTree implementation.

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 14: ETCD BTREE Complexity-1

Table 14 carries the values for Space and Time complexity for AVL implementation of key value store for first sample. Space complexity is O(n) , so the table size carries at the space complexity, where as time complexity is O(logn), so the logarithmic values are available.



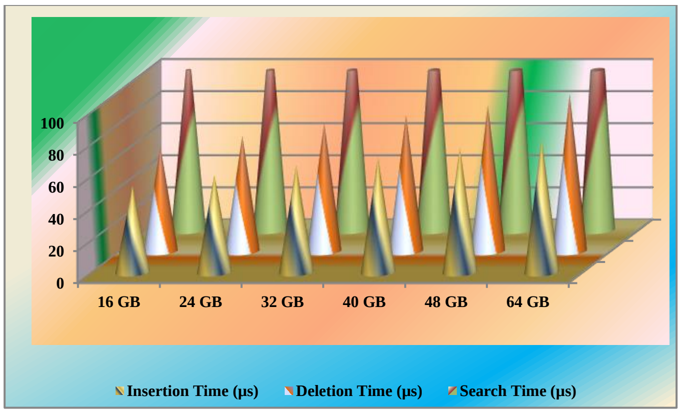
Graph 21: ETCD – Complexity-1

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 21 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	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)

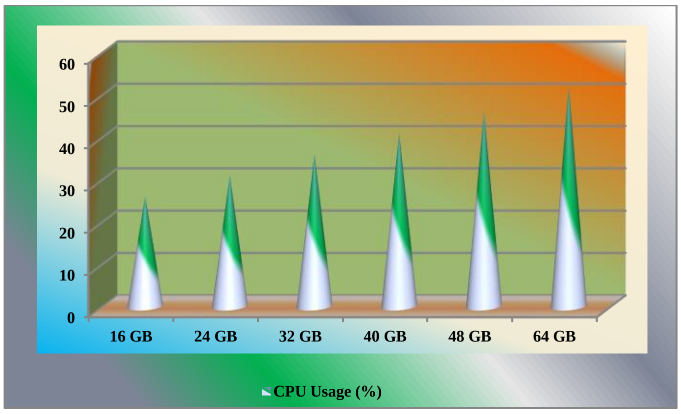
Table 15: ETCD Parameters – BTRee – 2

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 : BTRee – 2

If the node is not a leaf, it finds the appropriate child node to descend into. If that child is full, it splits the child before descending further. splitChild, Splits a full child node. It moves the median key of the full child up to the parent node, divides the child's keys and children, and adjusts pointers to maintain the B-Tree structure. Search, Searches for a key in the B-Tree, moving down through child nodes based on the values in the keys array of each node until it either finds the key or determines that the key is not present.



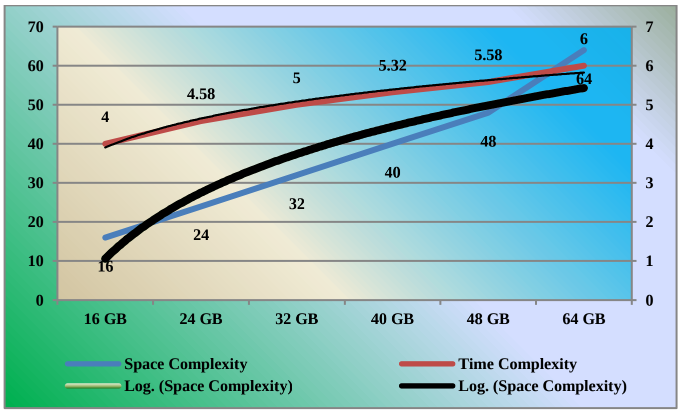
Graph 23: ETCD – CPU Usage-2

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(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 16: ETCD BTREE Complexity-2

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



Graph 24: ETCD – Complexity-2

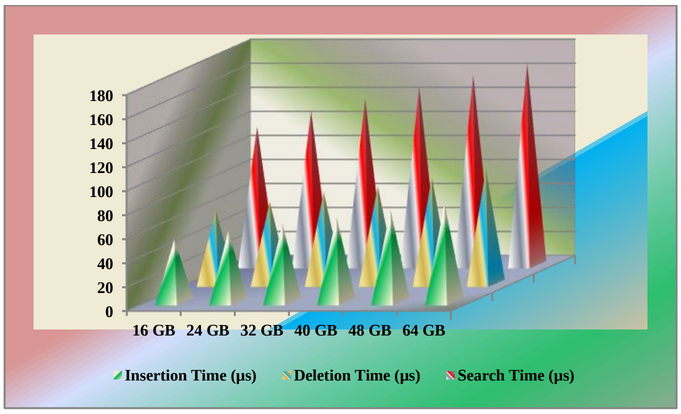
Please find the Logarithmic graph using the calculation, O(log n) ≈ 4 (using base 2 logarithm), O(n) = 16, 24, 32, 40, 48 and 64 for the n values from the size of the store which we have mentioned in the table. Graph 24 shows the same values. It is using two scale Y-Axis since the table is carrying two 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 17 : ETCD Parameters – BTRee – 3

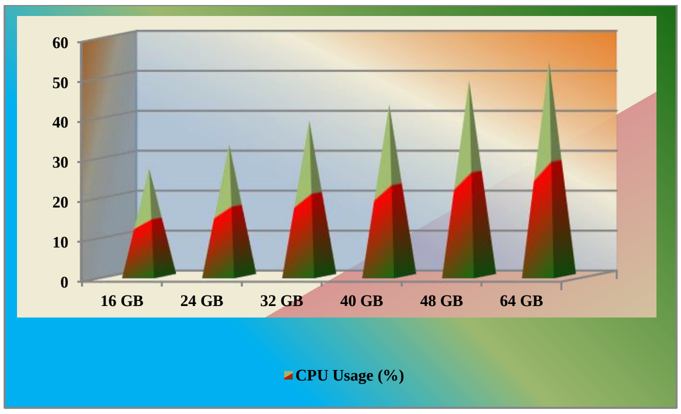
Table 17, shows the fourth sample of the data from ETCD store. ETCD Stores a key-value pair in etcd, Syntax: etcdctl put <key> <value>, etcdctl put message "Hello, world!"

- 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 : 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.

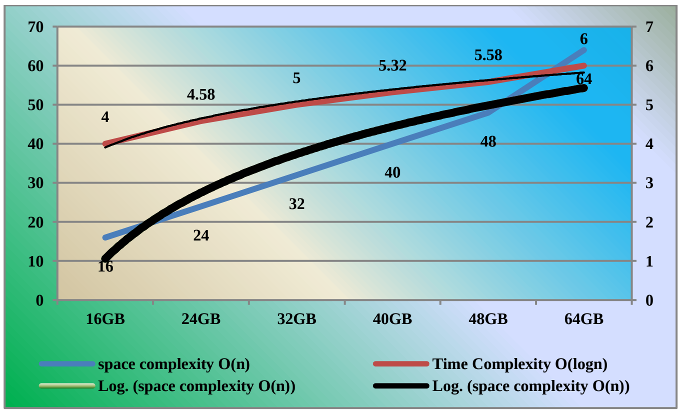


Graph 26: ETCD – CPU Usage-3

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 BTRee Complexity-3

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



Graph 27: 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 27 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.

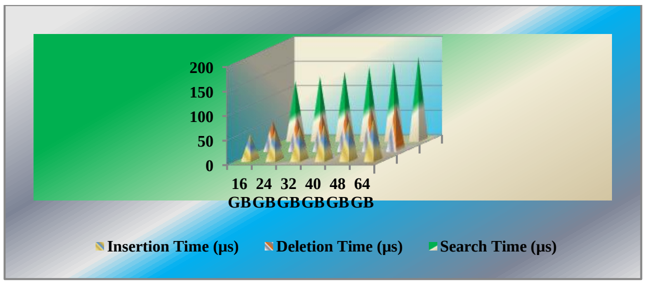
Store Size	Ins(µs)	Del (µs)	Sea (µs)	CPU(%)	S-Comp	T-Comp
16 GB	53	60	119	26	O(n)	O(log n)
24 GB	60	67	129	31	O(n)	O(log n)

32 GB	66	75	139	36	O(n)	O(log n)
40 GB	72	80	149	42	O(n)	O(log n)
48 GB	78	87	159	47	O(n)	O(log n)
64 GB	83	94	169	52	O(n)	O(log n)

Table 19: ETCD Parameters BTRee -4

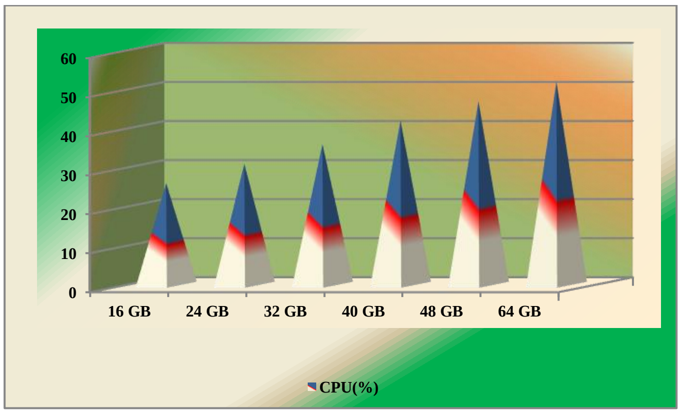
Table 19 shows the ETCD BTree implementation parameters like avg Insertion time, deletion time, search time (units are micro seconds) , and the % of CPU usage, Space and Time complexity. Space complexity is uniform for all the sizes of the store i.e, O(n) , and the time complexity is O(logn). This is also same irrespective of the size of the store.

ETCD GET operation retrieves a value from the store and the syntax , etcdctl get <key>, etcdctl get /message, API: client.Get(ctx, key, opts), ctx represents the context for the Get operation, It provides a way to cancel or timeout the operation. In Go, ctx is typically created using context.Background() or context.WithTimeout(). Example: ctx := context.Background(), key specifies the key to retrieve from etcd, Keys are strings and can be up to 4096 bytes, Keys can contain slashes (/) to create hierarchical



Graph 28: ETCD Parameters : BTREE - 4

Graph 28 shows the insertion time , deletion time and search time in micro seconds. X axis shows the ETCD store entries like 16GB , 24GB, 32GB, 40GB , 48GB and 64GB and the Y axis shows the all operations in micro seconds.



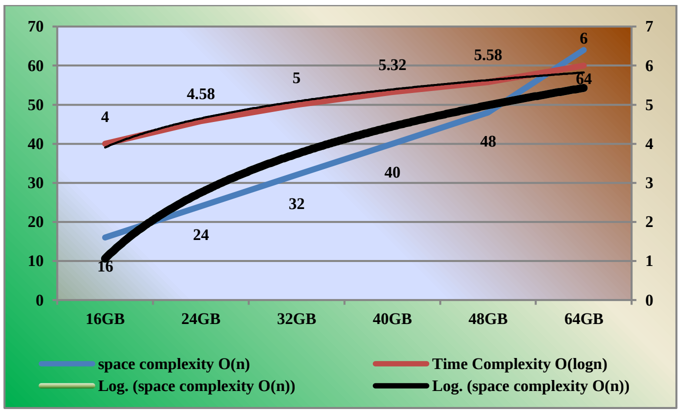
Graph 29: ETCD – CPU Usage-4

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 BTRee Complexity-4

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



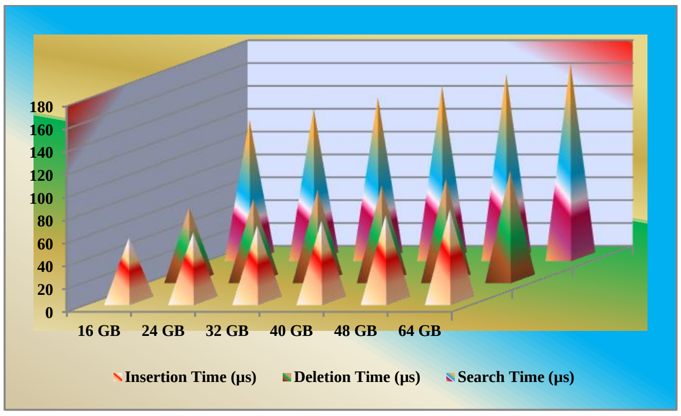
Graph 30: ETCD – Complexity-4

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 30 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.

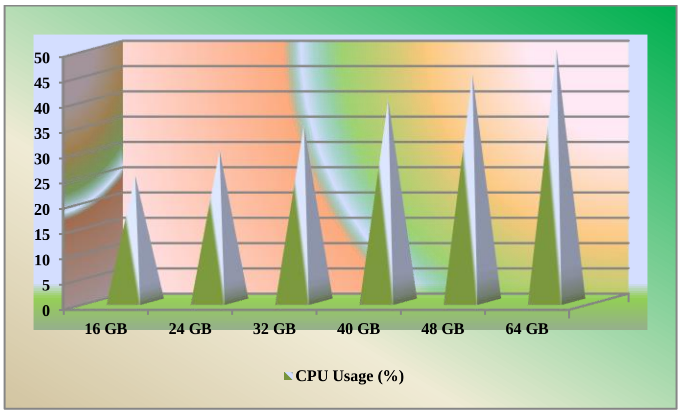
Store Size	Ins (μs)	Del (μs)	Sea (μs)	CPU (%)	S-Comp	T-Comp
16 GB	55	63	120	25	O(n)	O(log n)
24 GB	60	70	130	30	O(n)	O(log n)
32 GB	66	78	140	35	O(n)	O(log n)
40 GB	70	82	150	40	O(n)	O(log n)
48 GB	75	88	160	45	O(n)	O(log n)
64 GB	80	95	170	50	O(n)	O(log n)

Table 21: ETCD Parameters – BTRee – 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 fifth sample.



Graph 31: ETCD Parameters : BTRee – 5

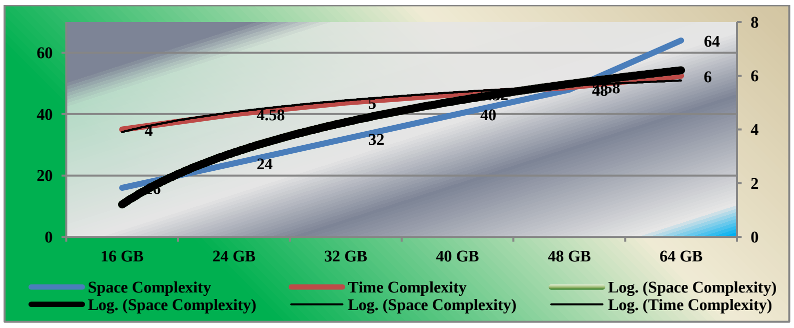


Graph 32: ETCD – CPU Usage-5

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 BTRee Complexity-5

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



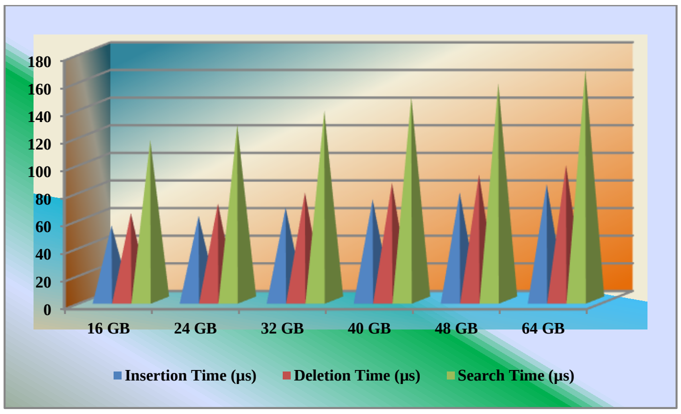
Graph 33: ETCD – Complexity-5

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 33 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	54	63	116	26	O(n)	O(log n)
24 GB	61	70	127	32	O(n)	O(log n)
32 GB	67	78	137	38	O(n)	O(log n)
40 GB	73	85	147	44	O(n)	O(log n)
48 GB	78	91	157	50	O(n)	O(log n)
64 GB	84	98	167	55	O(n)	O(log n)

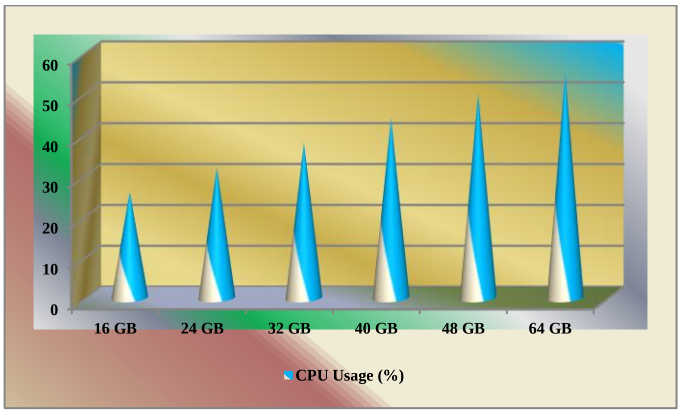
Table 23: ETCD Parameters BTRee Tree -6

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



Graph 34: ETCD Parameters : BTRee – 6

Graph 34 shows the BTRee implementation parameters for ETCD like insertion time, deletion time and search time , all are in micro seconds.



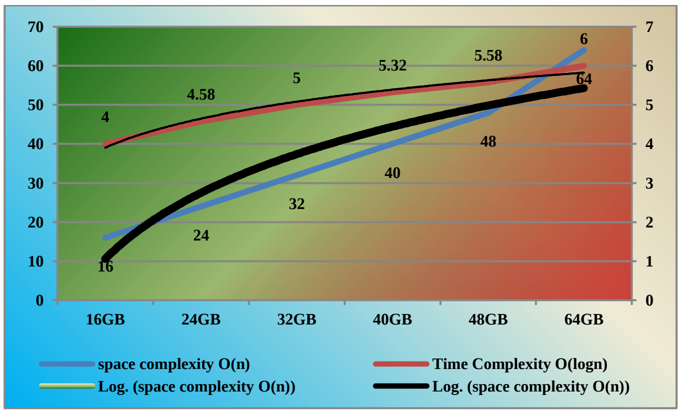
Graph 35: ETCD – CPU Usage-6

Graph 35 shows the cpu usage of ETCD having BTRee implementation. We have tested the performance by using the performance test code which we have mentioned in the previous section.

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

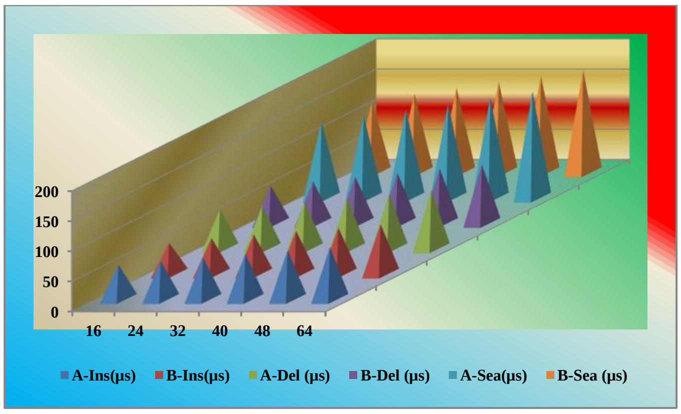
Table 24: ETCD BTree Complexity-6

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



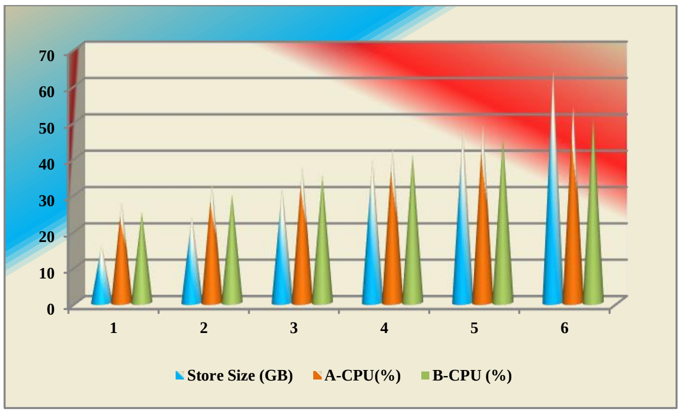
Graph 36: ETCD – Complexity-6

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 36 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.



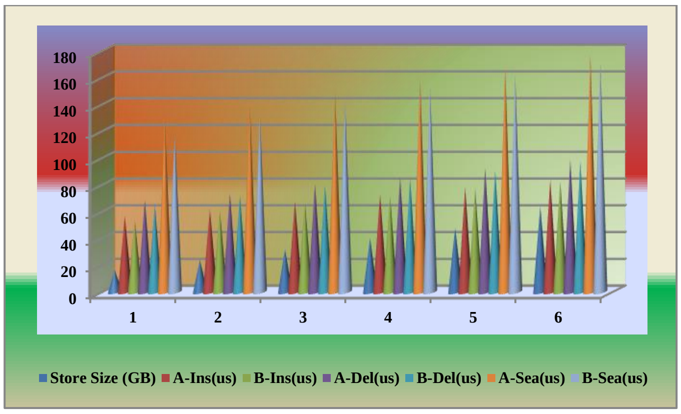
Graph 37: ETCD AVL Vs BTRee -1.1

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



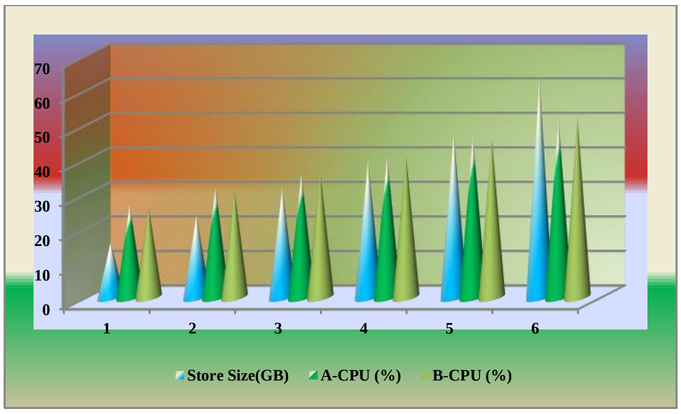
Graph 38: ETCD AVL Vs BTRee Tree-1.2

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



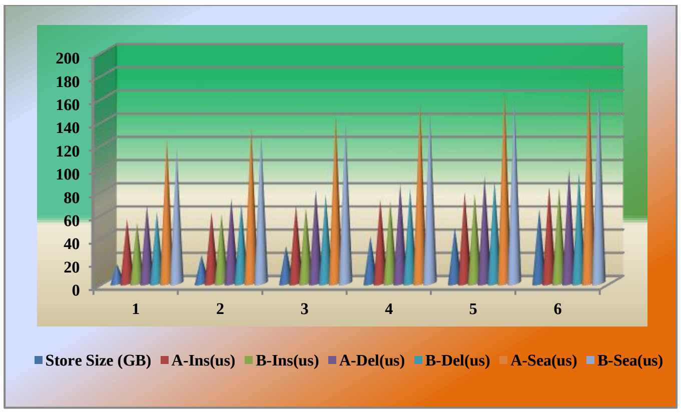
Graph 39: ETCD AVL Vs BTRee Tree-2.1

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



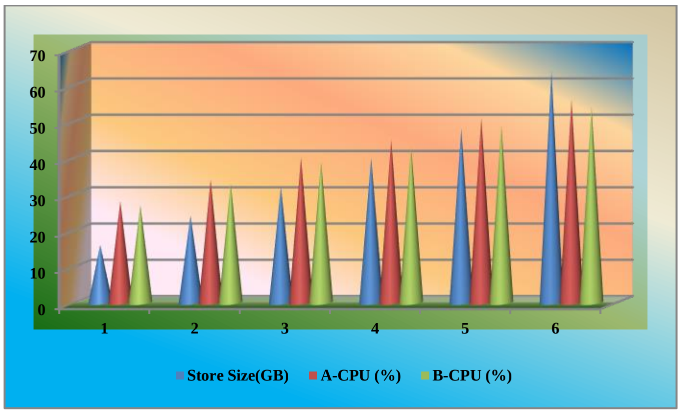
Graph 40: ETCD AVL Vs BTRee -2.2

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



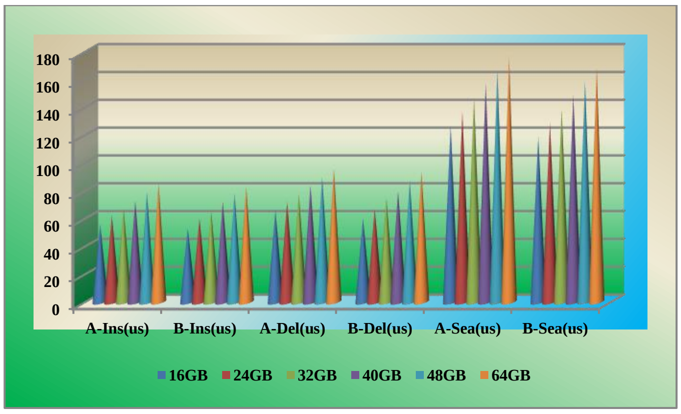
Graph 41: ETCD AVL Vs BTRee -3.1

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



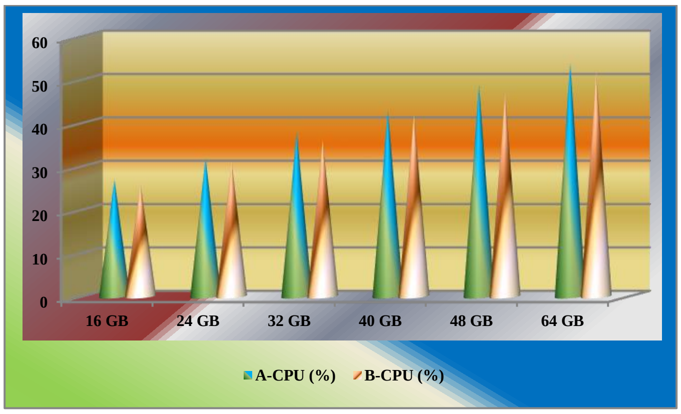
Graph 42: ETCD AVL Vs BTRee -3.2

Graph 42 shows that the CPU utilization is going down form high to low when we are moving from AVL implementation to BTRee implementation of Key value store.



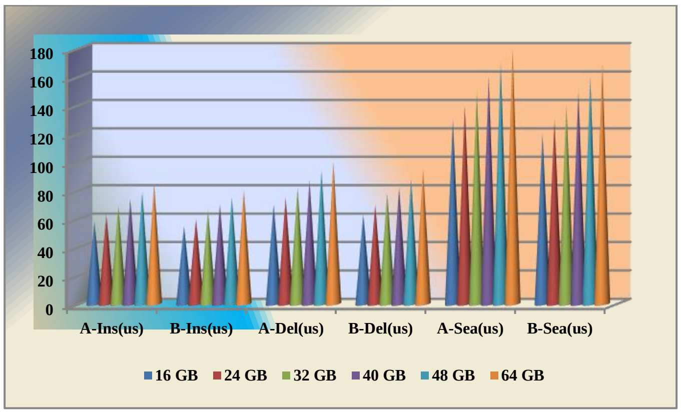
Graph 43: ETCD AVL Vs BTRee -4.1

Graph 43, is the comparison between AVL and BTREE Tree implementation of the key value store (ETCD) for the fourth sample. The graph shows the Insertion time difference between AVL and BTRee Tree implementation. As per the graph the time trend is going down as move from AVL to BTRee Tree implementation. The same observation we can have with other parameters like deletion time and search time.



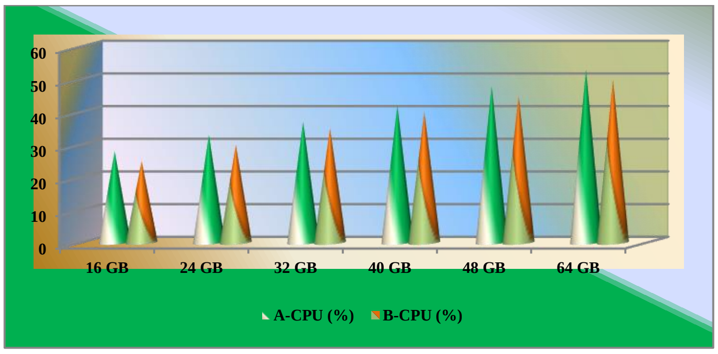
Graph 44: ETCD AVL Vs BTRee -4.2

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



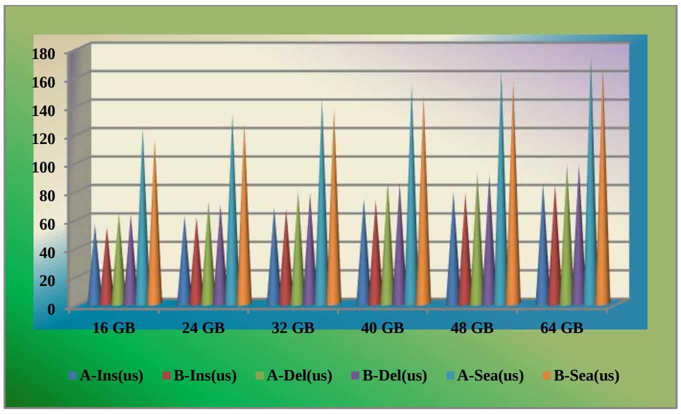
Graph 45: ETCD AVL Vs BTREE Tree-5.1

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



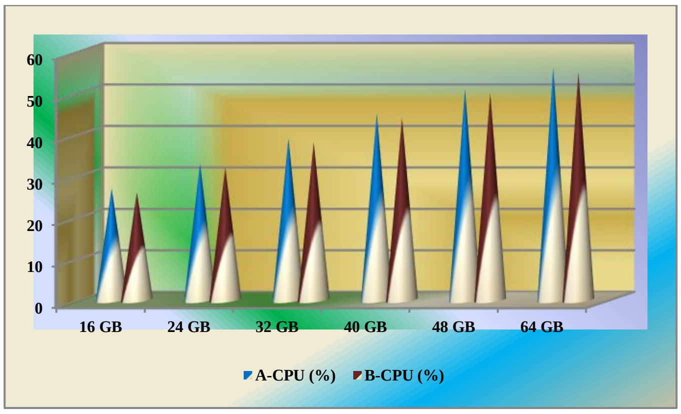
Graph 46: ETCD AVL Vs BTRee -5.2

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



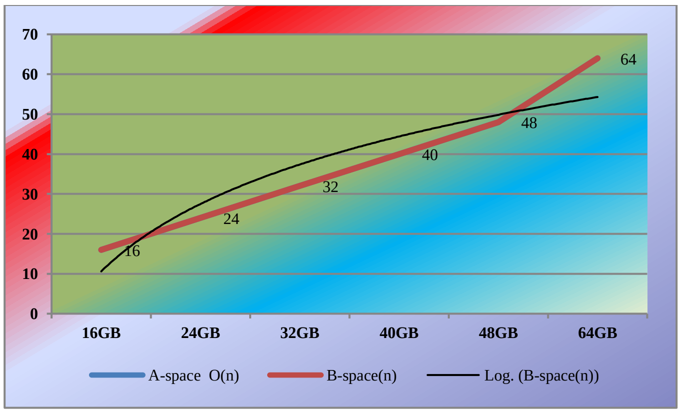
Graph 47: ETCD AVL Vs BTRee -6.1

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



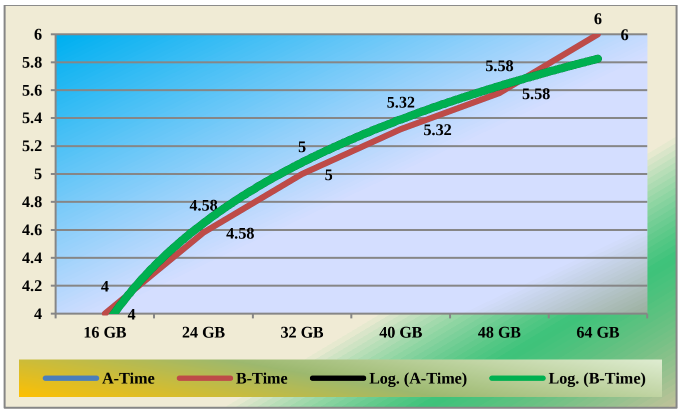
Graph 48: ETCD AVL Vs BTRee -6.2

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



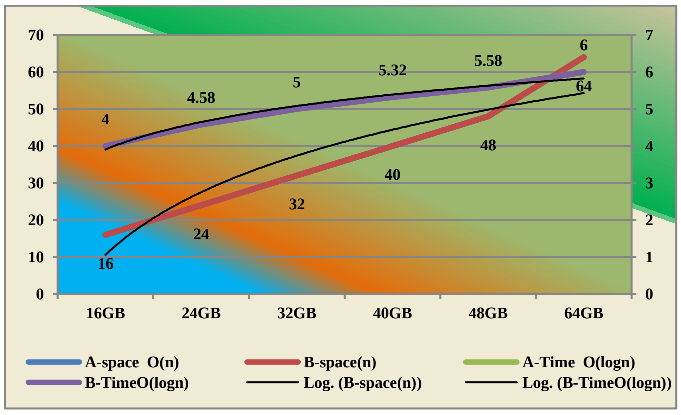
Graph 49: ETCD AVL Vs BTRee - Space Complexities

Graph 49 shows the space complexities comparison for the AVL and BTRee implementation of the key value store.



Graph 50: ETCD AVL Vs BTRee - Time Complexities

Graph 50 shows the comparison of time complexities between AVL and BTRee implementation of the ETCD.



Graph 51: ETCD AVL Vs BTRee Time and Space complexities

Graph 49 , 50 and 51 shows the comparison of complexities between AVL and BTRee Tree implementation. We can conclude that by using the BTRee implementation of the ETCD is better than using the AVL implementation. In summary, the time complexity of BTRee is generally O(n) for insertion, deletion, and search operations on average, with occasional $O(n)$ overheads for compaction, amortized over time. This makes BTRee highly efficient for applications requiring fast sequential writes and moderate lookup performance.

EVALUATION

The comparison of AVL implementation results with BTRee 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 you start using the implementation of the Data Store (ETCD) using the BTRee 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 BTRee implementation compared to AVL implementation. Space complexity and time complexity are also compared, along with CPU usage . Complexities are almost same , while CPU usage values are going down.

Please use AVL implementation of ETCD AVL Trees are strongly balanced, so search operations are fast (O(log n)). If the workload consists mostly of read operations, AVL Trees perform well because balance is maintained rigorously.

If there are large Datasets and Disk Storage, B-Trees are optimized for storage on disk rather than in-memory. They minimize the number of disk reads due to their wide branching factor, which allows more keys per node.

By having the analysis which we had through out the paper , we can conclude that insertion time, deletion time, search time , CPU usage are getting decreased automatically while complexities remains the same.

Future work: B-Trees use wide nodes to reduce the height of the tree, which is beneficial for large datasets but can lead to inefficient memory usage when handling small datasets We need to work on how to deal with small data sets while using BTree implementation of
ETCD.

REFERENCES:

1. Scalable and Reliable Kubernetes Clusters" by Google (2018).
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. Kubernetes Scalability and Performance" by Red Hat (2019).
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 Namespahttps://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. Reliability Analysis of Kubernetes Distributed Systems" by University of California (2020).
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. "Kubernetes Network Security" by Cisco (2018).
30. A Portable Load Balancer for Kubernetes Cluster, 28 January 2018, Kimitoshi Takahashi, Kento Aida, Tomoya Tanjo, Jingtao SunAuthors Info & Claims.
31. "etcd: A Highly-Available, Distributed Key-Value Store" by Brandon Philips et al. (2014), Proceedings of the 2014 ACM SIGOPS Symposium on Cloud Computing.
32. 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.
33. Performance Evaluation of etcd in Distributed Systems" by Jiahao Chen et al. (2020), 2020 IEEE International Conference on Cloud Computing (CLOUD).
34. Rearchitecting Kubernetes for the Edge, Andrew Jeffery, Heidi Howard, Richard MortierAuthors Info & Claims, 26 April 2021.
35. Kubernetes Storage Performance by Red Hat (2019).
36. Kubernetes Persistent Storage by Google (2018).
37. High Availability Storage Server with Kubernetes, Ali Akbar Khatami; Yudha Purwanto; Muhammad Faris Ruriawan, 2020, IEEE Xplore.
38. "Optimizing Kubernetes for Low-Latency Applications" by IBM (2020).
39. "Performance Analysis of Kubernetes Clusters" by Microsoft (2018).

40. "Secure Kubernetes Deployment" by Palo Alto Networks (2019)".
41. 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.
42. AVL and Red Black tree as a single balanced tree, March 2016, Zegour Djamel Eddine, Lynda Bounif
43. The log-structured merge-tree (AVL-tree),June 1996, Patrick O'Neil, Edward Cheng, Dieter Gawlick & Elizabeth O'Neil.
44. "Kubernetes Network Policies" by Calico (2019).