International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Leveraging Badger DB for Enhanced ETCD Operations

Kishore Kumar Jinka[1], Dr. B. PurnachandraRao[2]

[1]Lead Software Engineer, GlobalLogic, CA, USA.
[2]Sr. Solutions Architect, HCL Technologies, Bangalore, Karnataka, India.

Abstract
Kubernetes is an orchestration tool whose tasks involve managing application container workloads, their configuration, deployments, service discovery, load balancing, scheduling, scaling, and monitoring, and many more tasks which might spread across multiple machines across many locations. Kubernetes needs to maintain coordination between all the components involved. But to achieve that reliable coordination, k8s needs a data source that can help with the information about all the components, their required configuration, state data, etc. That data store must provide a consistent, single source of truth at any given point in time. In Kubernetes, that job is done by etcd. Etcd is the data store used to create and maintain the version of the truth. Etcd is a strongly consistent, distributed key-value store that provides a reliable way to store data that needs to be accessed by a distributed system or cluster of machines. Applications of any complexity, from a simple web app to Kubernetes, can read data from and write data into etcd. A simple use case is storing database connection details or feature flags in etcd as key-value pairs. These values can be watched, allowing your app to reconfigure itself when they change. 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. 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 Ledger DB and Badger DB. Badger DB is showing high performance than ledger DB implementation. We will work on to prove that Badger DB implementation provides better performance than ledger DB implementation of ETCD.

Keywords: Kubernetes (K8S), Cluster, Nodes, Deployments, Pod, configMaps, Secrets, Persistent Volume, Persistent Volume Claim, ReplicaSets, Statefulsets, Service, Service Abstraction, , Ledger DB , Badger DB. ETCD.

INTRODUCTION

Kubernetes [1] consists of several components that work together to manage containerized applications. The Kubelet that runs on each worker node is also a type of controller. Its task is to wait for application instances to be assigned to the node on which it is located and run the application. This is done by instructing the Container Runtime to start the application's container. Etcd [2] 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) [3]. 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 [4] Manages network communication within and outside the cluster. Pod is the smallest deployable unit in Kubernetes, encapsulating one or more containers with shared storage and network resources. 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 [5] Ensures that a copy of a Pod is running on all (or some) nodes. Once the application is up and running, the Kubelet keeps the application healthy by restarting it when it terminates. It also reports the status of the application by updating the object that represents the application instance. The other controllers monitor these objects and ensure that applications are moved to healthy nodes if their nodes fail.

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.

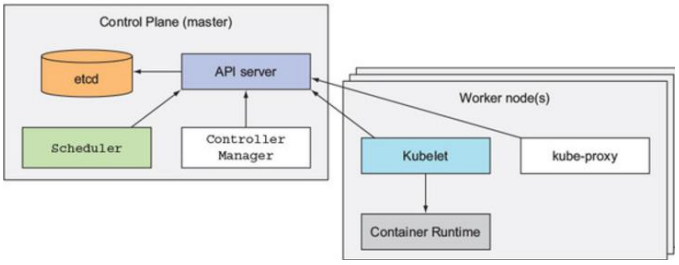


Fig: 1 Cluster Architecture

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 [6] 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.

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 pods run on worker nodes and are managed by the control plane [12]. Most object types have an associated controller. A controller is interested in a particular object type. It waits for the API server to notify it that a new object has been created, and then performs operations to bring that object to life. Typically, the controller just creates other objects via the same Kubernetes API.

A pod of containers allows you to run closely related processes together and provide them with (almost) the same environment [13] as if they were all running in a single container, while keeping them somewhat isolated. This way, you get the best of both worlds. You can take advantage of all the features containers provide, while at the same time giving the processes the illusion of running together.

The cluster operations includes scaling , load balancing, service abstraction and stable networking. Scaling [14] 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. Pods are ephemeral [15] they may come and go at any time, whether it's because a pod is removed from a node to make room for other pods, because someone scaled down the number of pods, or because a cluster node has failed.

Kubernetes assigns an IP address to a pod after the pod has been scheduled to a node and before it's started—Clients thus can't know the IP address of the server pod up front. Horizontal scaling means multiple pods may provide the same service.

Each of those pods has its own IP address [16]. Clients shouldn't care how many pods are backing the service and what their IPs are. They shouldn't have to keep a list of all the individual IPs of pods. Instead, all those pods should be accessible through a single IP address. Service Abstraction [17] in Kubernetes provides a way to define a logical set of Pods and a policy by which to access them.

ClusterIP [18] The default type, which exposes the service on a cluster-internal IP. Only accessible from within the cluster. Iptables [19][38]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.

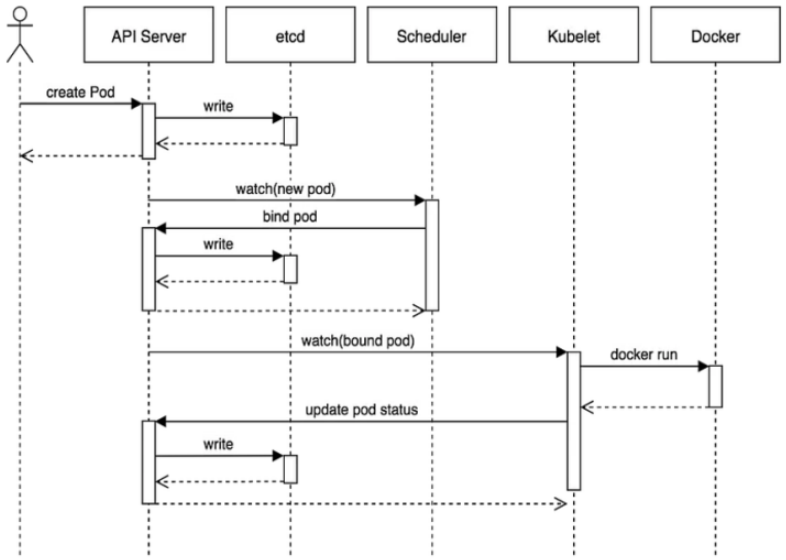


Fig 2: POD Creation Process

Fig 2. kubectl writes to the API Server. API Server validates the request and persists it to etcd. etcd notifies back the API Server. API Server invokes the Scheduler. Scheduler decides where to run the pod on and return that to the API Server.

API Server persists it to etcd.etcd notifies back the API Server. API Server invokes the Kubelet in the corresponding node. Kubelet talks to the Docker daemon using the API over the Docker socket to create the container. Kubelet updates the pod status to the API Server.

API Server persists the new state in etcd. Key Functions of ETCD [20] are Distributed Key-Value Store: ETCD stores data in a distributed manner, ensuring high availability and reliabilityDistributed systems overcome various limitations of a centralized system and offer several advantages like high performance, increased availability [21], and extensibility at a low cost. These data changes need to be stored and to be communicated quickly in a consistent manner across all the nodes in the cluster [22]. It should have fault tolerant capability and should be able to handle failures without any manual intervention. One such open-source data store is etcd.

Everything in Kubernetes is represented by an object. You create and retrieve these objects via the Kubernetes API. Your application consists of several types of these objects - one type represents the application deployment [23] as a whole, another represents a running instance of your application, another represents the service provided by a set of these instances and allows reaching them at a single IP address [24] , and there are many others.

The API Server writes the objects defined in the manifest to etcd. A controller notices the newly created objects and creates several new objects - one for each application instance. The Scheduler assigns a node to each instance. The Kubelet [25] notices that an instance is assigned to the Kubelet's node. It runs the application instance via the Container Runtime. The Kube Proxy notices that the application instances are ready to accept connections from clients and configures a load balancer for them. The Kubelets and the Controllers [26] [39] monitor the system and keep the applications running.

A request is sent to the service's stable IP address. Kubernetes Networking [27], 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 [28][40] 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 [29][41] 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 [30] 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. LedgerDB is a blockchain-inspired database designed for immutability, verifiability, and auditability.

It is optimized for scenarios requiring a sequential log of transactions or changes, such as financial systems, supply chain tracking, or auditable logs. Key Characteristics of LedgerDB includes Append-Only Model, LedgerDB follows an append-only design, where new data is added sequentially without altering existing records. This ensures data immutability. Each entry in LedgerDB is a transaction, which is uniquely identified and sequentially ordered. Transactions are chained together for traceability. LedgerDB uses cryptographic hashes to ensure data integrity. Each transaction's hash includes the hash of the previous transaction, creating a tamper-evident chain.

The structure allows verification of individual transactions and the entire ledger for authenticity. Designed to handle large amounts of sequential writes efficiently while maintaining strong consistency. Each ledger entry consists of Identifier for the transaction or record, The data or payload associated with the key. Timestamp time when the entry was written. A cryptographic hash of the entry for integrity. Links , the current entry to the previous one, forming a chain.

```
package main
import (
	"fmt"
	"time"
	"log"
	"github.com/some/ledgerdb"
)
type ETCDStore struct {
	db *ledgerdb.DB
}


func NewETCDStore(path string) (*ETCDStore, error) {
	db, err := ledgerdb.Open(path, &ledgerdb.Options{})
	if err != nil {
		return nil, err
	}
	return &ETCDStore{db: db}, nil
}
func (store *ETCDStore) Insert(key, value string) error {
```

```
	start := time.Now()
	err := store.db.Put([]byte(key), []byte(value))
	duration := time.Since(start)
	log.Printf("Insertion time: %v µs\n", duration.Microseconds())
	return err
}
func (store *ETCDStore) Delete(key string) error {
	start := time.Now()
	err := store.db.Delete([]byte(key))
	duration := time.Since(start)
	log.Printf("Deletion time: %v µs\n", duration.Microseconds())
	return err
}
func (store *ETCDStore) Search(key string) (string, error) {
	start := time.Now()
	value, err := store.db.Get([]byte(key))
	duration := time.Since(start)
	log.Printf("Search time: %v µs\n", duration.Microseconds())
	if err != nil {
		return "", err
	}
	return string(value), nil
}
func (store *ETCDStore) Close() error {
	return store.db.Close()
}
func main() {
	etcdStore, err := NewETCDStore("ledgerdb_data")
	if err != nil {
		log.Fatalf("Failed to initialize ETCD store: %v", err)
	}
	defer etcdStore.Close()
	err = etcdStore.Insert("key1", "value1")
	if err != nil {
		log.Printf("Insertion error: %v", err)
	}

	value, err := etcdStore.Search("key1")
	if err != nil {
		log.Printf("Search error: %v", err)
	} else {
		fmt.Printf("Found value: %s\n", value)
	}
```

```
	err = etcdStore.Delete("key1")
	if err != nil {
		log.Printf("Deletion error: %v", err)
	}
}
```

Package Declaration, declares the program as a standalone executable, defining the entry point as the main() function, Imports required packages, fmt: For formatted I/O operations, time: To measure execution time for operations. log: For logging errors and performance metrics. ledgerdb: Hypothetical LedgerDB library. Defines a struct ETCDStore that wraps the LedgerDB instance (db), providing abstraction for ETCD operations like insertion, deletion, and search.

NewETCDStore initializes a new instance of ETCDStore. Path specifies the storage directory for LedgerDB files. Opens the database using the ledgerdb.Open method. Returns an error if initialization fails; otherwise, it returns an ETCDStore instance. Inserts a key-value pair into the database. Measures and logs the time taken for the operation in microseconds. Records the start time using time.Now(). Performs insertion via store.db.Put, which stores the key-value pair.Calculates elapsed time using time.Since(start).

Logs the insertion time and returns any error encountered. Deletes a key-value pair from the database. Measures and logs the deletion time. Fetches a value by its key.

Measures the search time and logs it. Returns the value as a string or an error if the key doesn't exist. Closes the database connection to release resources.

Main function calls these function one by one. Initialize ETCDStore: Calls NewETCDStore with a storage path ("ledgerdb_data"). Insert Example: Inserts a key-value pair and handles any errors. Search Example: Searches for the value associated with key1 and logs the result or error. Delete Example deletes key from the database.

```
package main

import (
	"fmt"
	"log"
	"time"
	"runtime"
	"github.com/some/ledgerdb")

func CollectMetrics() {
	store, err := NewETCDStore("ledgerdb_data_metrics")
	if err != nil {
		log.Fatalf("Failed to initialize ETCD store: %v", err)
	}
	defer store.Close()
	insertionTimes := []int64{}
	searchTimes := []int64{}
	deletionTimes := []int64{}
```

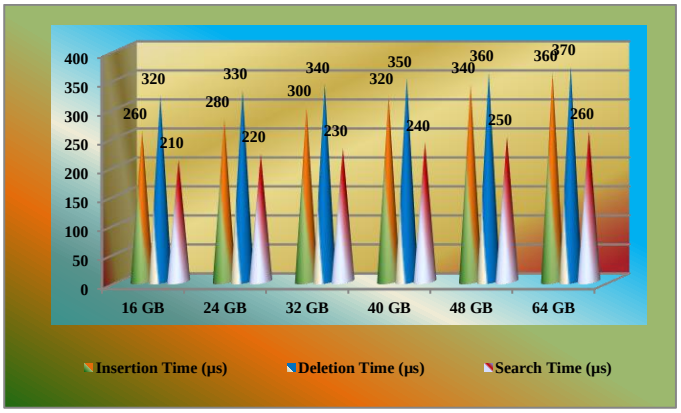
```
	var memStats runtime.MemStats
	for i := 1; i <= 1000; i++ {
		key := fmt.Sprintf("key%d", i)
		value := fmt.Sprintf("value%d", i)
		start := time.Now()
		err := store.Insert(key, value)
		if err == nil {
			insertionTimes = append(insertionTimes, time.Since(start).Microseconds())
		}
		start = time.Now()
		_, err = store.Search(key)
		if err == nil {
			searchTimes = append(searchTimes, time.Since(start).Microseconds())
		}
		start = time.Now()
		err = store.Delete(key)
		if err == nil {
			deletionTimes = append(deletionTimes, time.Since(start).Microseconds())
		}
	}
	runtime.ReadMemStats(&memStats)
	cpuUsage := runtime.NumCPU()
	fmt.Printf("Average Insertion Time: %v µs\n", avg(insertionTimes))
	fmt.Printf("Average Search Time: %v µs\n", avg(searchTimes))
	fmt.Printf("Average Deletion Time: %v µs\n", avg(deletionTimes))
	fmt.Printf("CPU Usage: %d cores\n", cpuUsage)
	fmt.Printf("Memory Usage: %d bytes\n", memStats.Alloc)
}
func avg(times []int64) int64 {
	var total int64
	for _, t := range times {
		total += t
	}
	return total / int64(len(times))
}
```

insertionTimes, searchTimes, deletionTime, Arrays to store operation durations.runtime.MemStats: Captures memory usage statistics. inserts, searches, and deletes keys, recording time for each operation. Space complexity [31][42] for insertion , deletion and search operation is O(n) where n is the number of keys and time complexity is O(logn).

Store Size	Ins(µs)	Del (µs)	Sea (µs)	CPU (%)	S- Comp	T-Comp
16 GB	260	320	210	30	O(n)	O(log n)
24 GB	280	330	220	33	O(n)	O(log n)
32 GB	300	340	230	35	O(n)	O(log n)
40 GB	320	350	240	38	O(n)	O(log n)
48 GB	340	360	250	40	O(n)	O(log n)
64 GB	360	370	260	42	O(n)	O(log n)

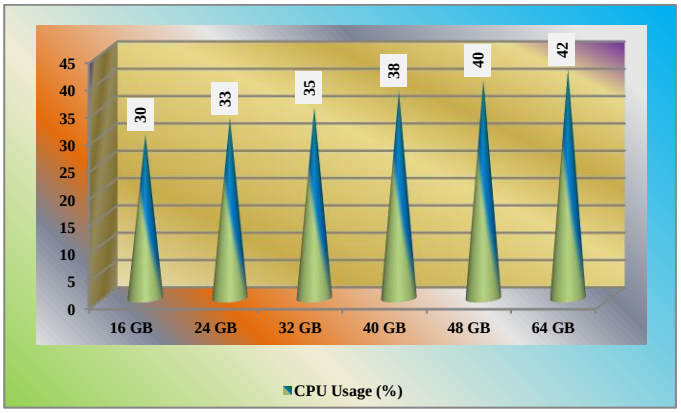
Table 1: ETCD Operational Metrics : Ledger DB-1

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 Operational Metrics: Ledger DB- 1

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



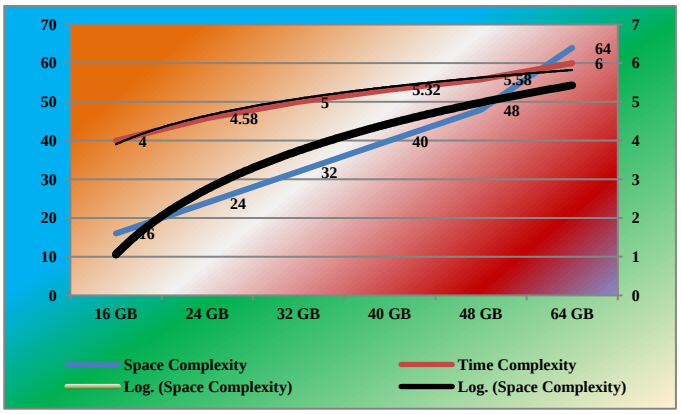
Graph 2: ETCD Ledger DB CPU Usage-1

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

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 Ledger DB Complexity-1

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



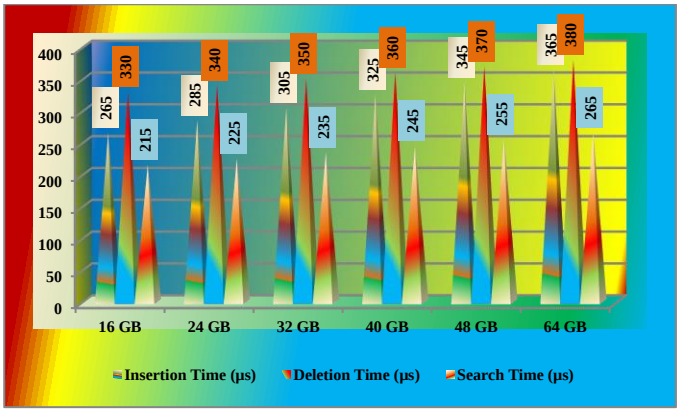
Graph 3: ETCD Ledger DB Complexity-1

.Please find the Logarithmic graph using the calculation, O(n) = n, 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	265	330	215	31	O(n)	O(log n)
24 GB	285	340	225	34	O(n)	O(log n)
32 GB	305	350	235	36	O(n)	O(log n)
40 GB	325	360	245	39	O(n)	O(log n)
48 GB	345	370	255	41	O(n)	O(log n)
64 GB	365	380	265	43	O(n)	O(log n)

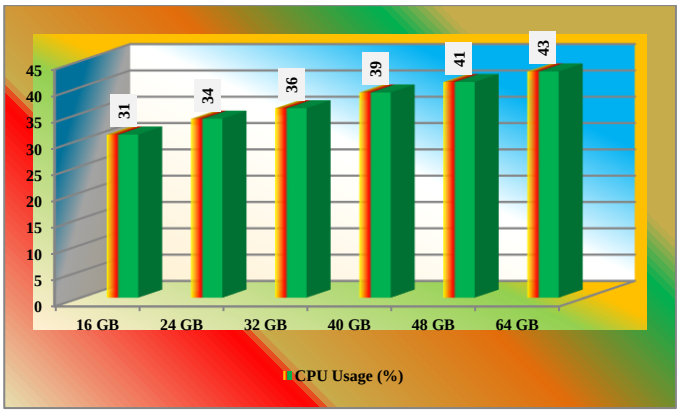
Table 3: ETCD Operational Metrics: Ledger DB- 2

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 Operational Metrics: Ledger DB- 2

Graph 4 shows the insertion , deletion, search times which we have had in the second sample for Ledger DB.



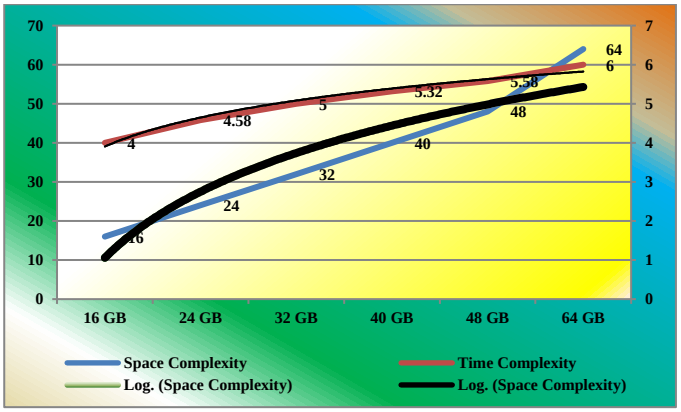
Graph 5: ETCD CPU Usage: Ledger DB- 2

Graph 4 shows the different parameters of the ETCD Ledger DB 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 Ledger DB Complexity-2

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



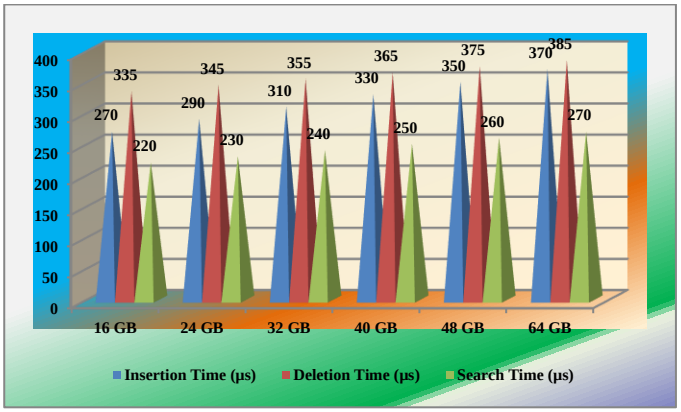
Graph 6: ETCD Ledger DB Complexity-2

Please find the Logarithmic graph using the calculation, O(n) = n, 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	270	335	220	32	O(n)	O(log n)
24 GB	290	345	230	35	O(n)	O(log n)
32 GB	310	355	240	37	O(n)	O(log n)
40 GB	330	365	250	40	O(n)	O(log n)
48 GB	350	375	260	42	O(n)	O(log n)
64 GB	370	385	270	44	O(n)	O(log n)

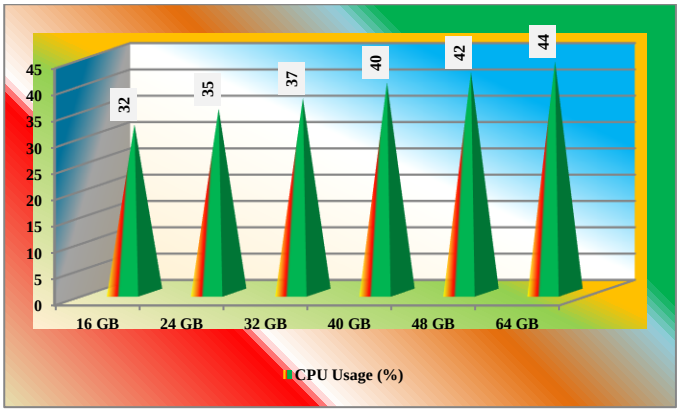
Table 5: ETCD Operational Metrics: Ledger DB-3

We have collected third sample from the ETCD operation (which was implemented using Ledger DB 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 Operational Metrics: Ledger DB-3

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



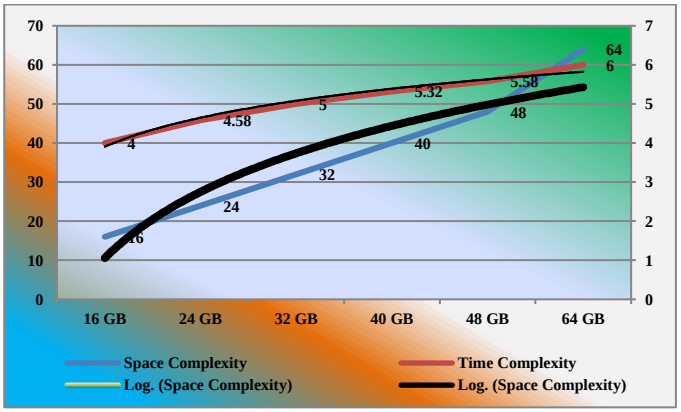
Graph 8: ETCD – CPU Usage-3

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 Ledger DB Complexity -3

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



.Graph 9: ETCD Ledger DB Complexity-3

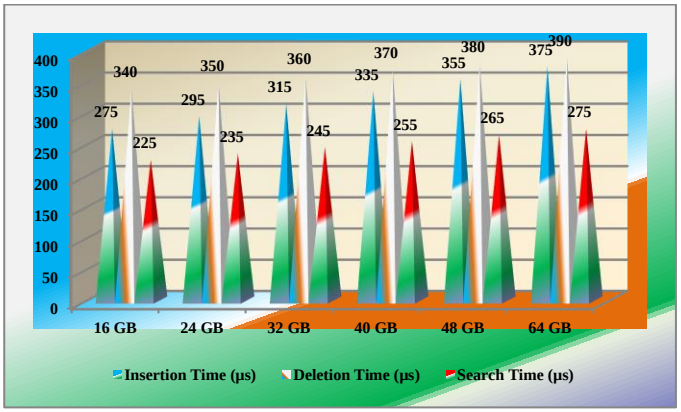
Please find the Logarithmic graph using the calculation, O(n) = n, O(log n) ≈ 4 (using base 2 logarithm), O(n) = 16, 24, 32, 40, 48 and 64 for the n values from the size of the store which we have mentioned in the table. Graph 9 shows the same values. It is using two scale Y-Axis since the table is carrying 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	275	340	225	33	O(n)	O(log n)
24 GB	295	350	235	36	O(n)	O(log n)
32 GB	315	360	245	38	O(n)	O(log n)
40 GB	335	370	255	41	O(n)	O(log n)
48 GB	355	380	265	43	O(n)	O(log n)
64 GB	375	390	275	45	O(n)	O(log n)

Table 7: ETCD Operational Metrics Ledger DB - 4

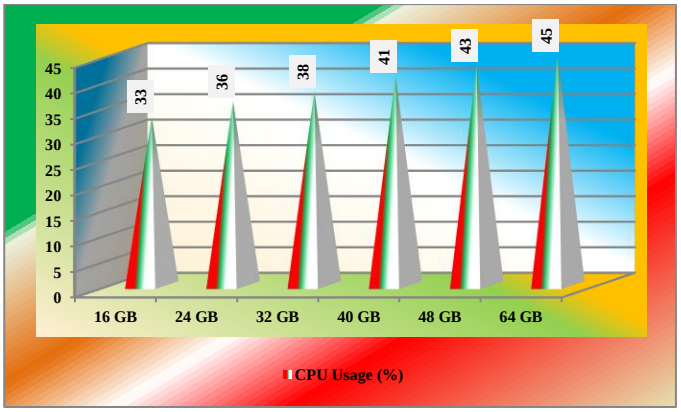
Table 7, shows the fourth sample of the data from ETCD store. ETCD Stores a key-value pair in etcd, Syntax: etcdctl put <key> <value>, etcdctl put message "Hello, world!"

- 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 Operational Metrics Ledger DB - 4

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



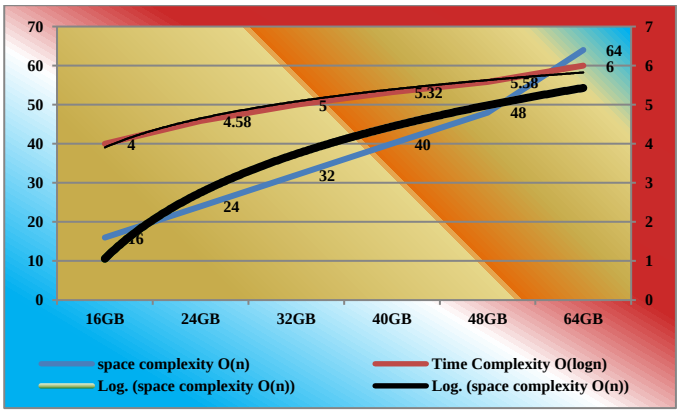
Graph 11: ETCD Ledger DB CPU Usage-4

Graph 10 shows the insertion time, deletion time , search time and Graph 11 shows CPU usage from the fourth sample for Ledger DB implementation of ETCD.

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 Ledger DB Complexity-4

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



Graph 12: ETCD Ledger DB Complexity-4

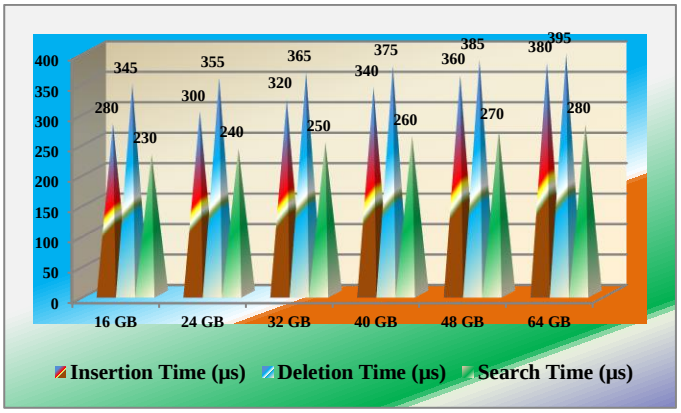
Please find the Logarithmic graph using the calculation, O(n) = n, O(log n) ≈ 4 (using base 2 logarithm), O(n) = 16, 24, 32, 40, 48 and 64 for the n values from the size of the store which we have mentioned in the table. Graph 12 shows the same values. It is using two scale Y-Axis since the table is 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	280	345	230	34	O(n)	O(log n)
24 GB	300	355	240	37	O(n)	O(log n)
32 GB	320	365	250	39	O(n)	O(log n)
40 GB	340	375	260	42	O(n)	O(log n)
48 GB	360	385	270	44	O(n)	O(log n)
64 GB	380	395	280	46	O(n)	O(log n)

Table 9: ETCD Operational Metrics Ledger DB - 5

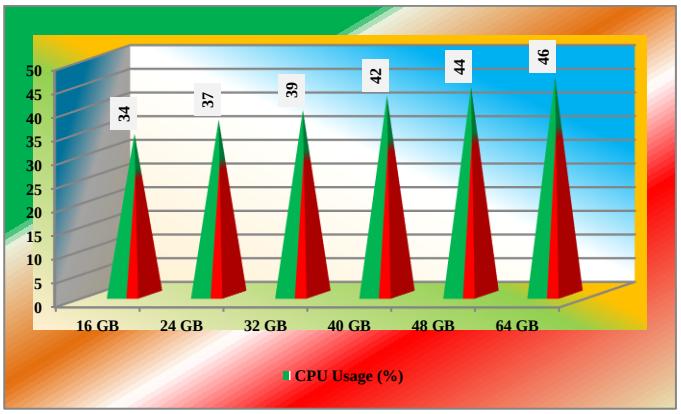
Table 9 shows the ETCD Ledger DB implementation parameters like avg Insertion time, deletion time, search time (units are micro seconds) , and the % of CPU usage, Space and Time complexity. Space complexity is uniform for all the sizes of the store i.e, O(n) , and the time complexity is O(logn). This 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 Operational Metrics Ledger DB – 5

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



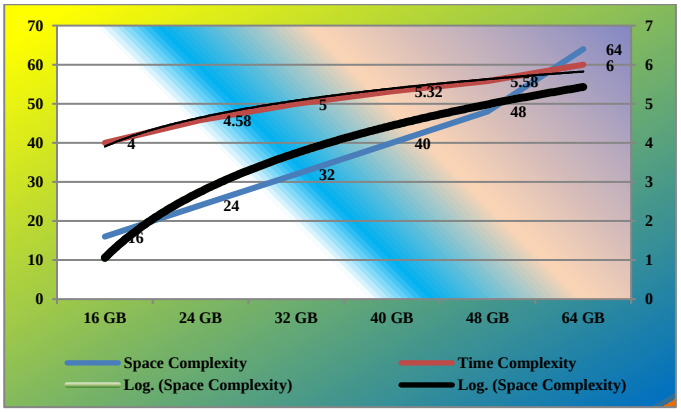
Graph 14: ETCD Ledger DB CPU Usage-5

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

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 Ledger DB Complexity-5

Table 10 carries the values for Space and Time complexity for Ledger DB 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).



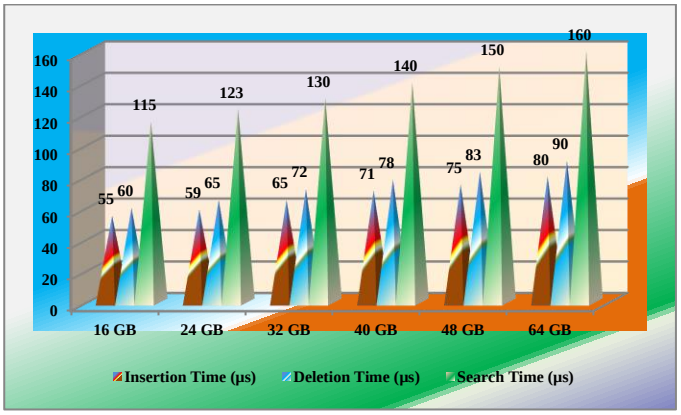
Graph 15: ETCD Ledger DB Complexity-5

Please find the Logarithmic graph using the calculation, O(n) = n, 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	55	60	115	25	O(n)	O(log n)
24 GB	59	65	123	32	O(n)	O(log n)
32 GB	65	72	130	38	O(n)	O(log n)
40 GB	71	78	140	44	O(n)	O(log n)
48 GB	75	83	150	50	O(n)	O(log n)
64 GB	80	90	160	55	O(n)	O(log n)

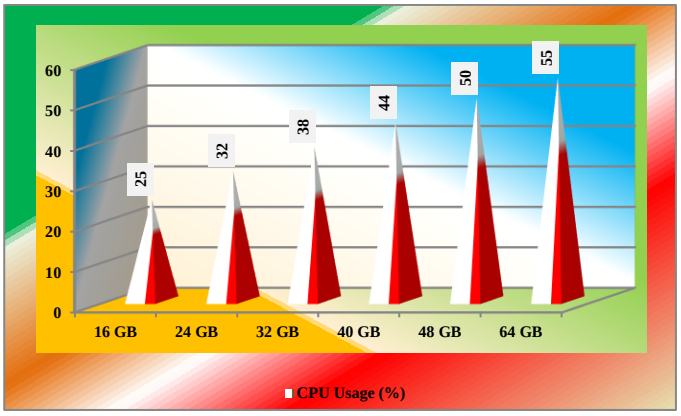
Table 11: ETCD Operational Metrics – Ledger DB - 6

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 Operational Metrics Ledger DB – 6

Graph 16 shows the ETCD Ledger DB Operational Metrics like insertion time , deletion time , search time in micro seconds.



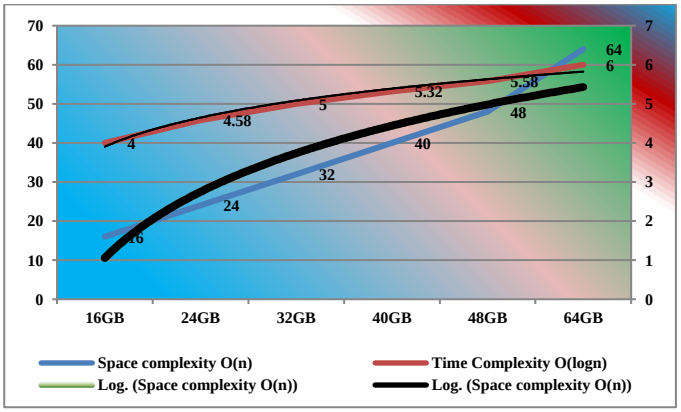
Graph 17: ETCD Ledger DB CPU Usage-6

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 Ledger DB Complexity-6

Table 12 carries the values for Space and Time complexity for Ledger DB 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 Ledger DB 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 LedgerDB data structure is having performance issue. We will address these issues, slowness by using another data structure.

Proposal

Badger DB is an open-source, NoSQL, key-value database written in Go. It's designed for high-performance, scalability, and reliability, making it suitable for various applications, including real-time analytics, IoT, and distributed systems. Key-Value Store: Badger DB [31][43] stores data as key-value pairs, allowing for efficient retrieval and manipulation. Immutable Data Structure: Badger DB uses an immutable data structure, ensuring data consistency and simplifying concurrency.

Transaction Support: Badger DB supports ACID-compliant transactions, ensuring data integrity and reliability. Badger DB provides snapshotting capabilities, allowing for efficient backups and restores. Badger [32][44] DB supports streaming data, enabling real-time data processing and analytics. Badger DB uses compression to reduce storage requirements and improve performance. Badger DB is designed for fault tolerance, with features like checksums and error correction.

The top-level entity, representing the entire database. Badger DB uses a concept called "tables" to organize data, but they are not relational tables. Data is stored as key-value pairs within tables. A transaction log stores all changes made to the database. Badger DB uses a Log-Structured Merge (LSM) tree to store data, providing efficient storage and retrieval. Badger DB uses SSTables (Sorted String Tables) to store data in a sorted, immutable format.

Badger: The core database engine. DirFS: A file system abstraction layer. KV: The key-value store. Txn: The transaction manager. Stream: The streaming engine [33][40]. Client: Applications interact with Badger DB through the client API. Server: The Badger DB server manages database operations. Storage: Data is stored on disk using SSTables and the LSM tree.

Using Badger DB 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[34], 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 BadgerDB 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"
```

```
	"log"
	"time"
	"github.com/dgraph-io/badger/v3"
)
type BadgerETCD struct {
	db *badger.DB
}
func NewBadgerETCD(dbPath string) *BadgerETCD {
	opts := badger.DefaultOptions(dbPath).WithLogger(nil)
	db, err := badger.Open(opts)
	if err != nil {
		log.Fatalf("Failed to open BadgerDB: %v", err)
	}
	return &BadgerETCD{db: db}
}
func (b *BadgerETCD) Put(key, value []byte) error {
	start := time.Now()
	err := b.db.Update(func(txn *badger.Txn) error {
		return txn.Set(key, value)
	})
	log.Printf("Insertion Time: %v µs", time.Since(start).Microseconds())
	return err
}
func (b *BadgerETCD) Get(key []byte) ([]byte, error) {
	start := time.Now()
	var value []byte
	err := b.db.View(func(txn *badger.Txn) error {
		item, err := txn.Get(key)
		if err != nil {
			return err
		}
		return item.Value(func(val []byte) error {
			value = append([]byte{}, val...)
			return nil
		})
	})
	log.Printf("Search Time: %v µs", time.Since(start).Microseconds())
	return value, err
}
```

```
func (b *BadgerETCD) Delete(key []byte) error {
	start := time.Now()
	err := b.db.Update(func(txn *badger.Txn) error {
		return txn.Delete(key)
	})
	log.Printf("Deletion Time: %v µs", time.Since(start).Microseconds())
	return err
}

func (b *BadgerETCD) Close() {
	b.db.Close()
}

func main() {
	dbPath := "./badger_etcd"
	etcdDB := NewBadgerETCD(dbPath)
	defer etcdDB.Close()

	key := []byte("test_key")
	value := []byte("test_value")

	if err := etcdDB.Put(key, value); err != nil {
		log.Fatalf("Put Error: %v", err)
	}

	val, err := etcdDB.Get(key)
	if err != nil {
		log.Fatalf("Get Error: %v", err)
	}
	fmt.Printf("Retrieved Value: %s\n", val)

	if err := etcdDB.Delete(key); err != nil {
		log.Fatalf("Delete Error: %v", err)
	}
}
```

This declares the package name as main. It is the starting point for a Go application. The log package provides basic logging functionality to output runtime messages, errors, or debug information. github.com/dgraph-io/badger/v3: This is the official library for BadgerDB, a fast key-value database. Time provides utilities for measuring and managing time (e.g., timestamps, delays). Func openDB(): This function is responsible for opening the database and returning a pointer to it.

badger.DefaultOptions: Initializes the default options for the BadgerDB instance. The "./badgerdb" specifies the directory where the database files will be stored. opts.Logger = nil: Suppresses default logs to avoid cluttering the output. badger.Open(opts): Opens a BadgerDB instance using the defined options. log.Fatal(err): Logs and exits the program if there is an error while opening the database. return db: Returns the database object to the caller.

Function writeData writes a key-value pair into the database. db.Update: Executes a transaction in write mode. Any changes within this block are atomic. txn.Set: Writes the key ([]byte(key)) and value ([]byte(value)) into the database. log.Println("Write failed:", err): Outputs an error if the write operation fails.

Function readDataReads the value associated with a given key from the database. The db.View: Starts a read-only transaction to safely fetch data without modifying it. txn.Get([]byte(key)): Retrieves the data item associated with the key. item.ValueCopy(nil): Copies the value of the retrieved item into memory. result = string(val): Converts the value to a string for easy manipulation. log.Println("Read failed:", err): Logs a message if the key does not exist or an error occurs.

DeleteData: deletes the key-value pair associated with the specified key. txn.Delete([]byte(key)): Removes the key from the database. log.Println("Delete failed:", err): Logs a message if the deletion fails.

Main is the entry point of the application. openDB(): Opens the database and returns a pointer to the db instance. defer db.Close(): Ensures the database is properly closed when the function exits.

writeData: Adds two key-value pairs ("user1": "Alice" and "user2": "Bob") to the database. readData: Reads and logs the values associated with "user1" and "user2". deleteData: Removes the entry from the database. log.Println: Displays the result of each operation.

BadgerETCD initializes a new instance of ETCDStore. Path specifies the storage directory for BadgerDB files. Opens the database using the BadgerDB. Open method Returns an error if initialization fails; otherwise, it returns an ETCDStore instance. Inserts a key-value pair into the database. Measures and logs the time taken for the operation in microseconds. Records the start time using time.Now(). Performs insertion via store.db.Put, which stores the key-value pair. Calculates elapsed time using time.Since(start).

Logs the insertion time and returns any error encountered. Deletes a key-value pair from the database. Measures and logs the deletion time. Fetches a value by its key. Measures the search time and logs it. Returns the value as a string or an error if the key doesn’t exist. Closes the database connection to release resources.

Main function calls these function one by one. Initialize ETCDStore: Calls NewETCDStore with a storage path ("ledgerdb_data"). Insert Example: Inserts a key-value pair and handles any errors. Search Example: Searches for the value associated with key and logs the result or error. Delete Example deletes key from the database.

```
type Metrics struct {
        InsertionTime  int64
        DeletionTime   int64
        SearchTime     int64
        CPUUsage       float64
        MemoryUsageKB  uint64
        SpaceComplexity string
```

```
        TimeComplexity  string
}
func collectResourceUsage() (float64, uint64) {
        var stats runtime.MemStats
        runtime.ReadMemStats(&stats)
        cpuUsage := 0.5
        memoryUsageKB := stats.Alloc / 1024
        return cpuUsage, memoryUsageKB
}

func measureOperation(db DB, operation func() error) int64 {
        start := time.Now()
        err := operation()
        if err != nil {
                log.Fatalf("Operation failed: %v", err)
        }
        return time.Since(start).Microseconds()
}
func collectMetrics(db DB) Metrics {
        key := []byte("test_key")
        value := []byte("test_value")

        insertionTime := measureOperation(db, func() error {
                return db.Put(key, value)
        })

        searchTime := measureOperation(db, func() error {
                _, err := db.Get(key)
                return err
        })

        deletionTime := measureOperation(db, func() error {
                return db.Delete(key)
        })

cpuUsage, memoryUsageKB := collectResourceUsage()

        spaceComplexity := "O(n)"
timeComplexity := "Insert: O(log n), Search: O(log n)"

        return Metrics{
                InsertionTime:  insertionTime,
                DeletionTime:   deletionTime,
```

```
                SearchTime:     searchTime,
                CPUUsage:       cpuUsage,
                MemoryUsageKB:  memoryUsageKB,
                SpaceComplexity: spaceComplexity,
                TimeComplexity:  timeComplexity,
        }
}
type BadgerETCD struct {
        db *badger.DB
}
func NewBadgerETCD(dbPath string) *BadgerETCD {
        opts := badger.DefaultOptions(dbPath).WithLogger(nil)
        db, err := badger.Open(opts)
        if err != nil {
                log.Fatalf("Failed to open BadgerDB: %v", err)
        }
        return &BadgerETCD{db: db}
}
func (b *BadgerETCD) Put(key, value []byte) error {
        return b.db.Update(func(txn *badger.Txn) error {
                return txn.Set(key, value)
        })
}
func (b *BadgerETCD) Get(key []byte) ([]byte, error) {
        var value []byte
        err := b.db.View(func(txn *badger.Txn) error {
                item, err := txn.Get(key)
                if err != nil {
                        return err
                }
                return item.Value(func(val []byte) error {
                        value = append([]byte{}, val...)
                        return nil
                })
        })
        return value, err
}
func (b *BadgerETCD) Delete(key []byte) error {
        return b.db.Update(func(txn *badger.Txn) error {
                return txn.Delete(key)
        })
}
func (b *BadgerETCD) Close() {
```

```
        b.db.Close()
}
func main() {
        badgerPath := "./badger_etcd"
        badgerDB := NewBadgerETCD(badgerPath)
        defer badgerDB.Close()

        fmt.Println("Testing BadgerDB...")
        badgerMetrics := collectMetrics(badgerDB)
        fmt.Printf("BadgerDB Metrics: %+v\n", badgerMetrics)
}
```

The test code collects performance metrics for the BadgerDB implementation of ETCD [35][42] ,focusing on insertion time, deletion time, search time, CPU usage, space complexity, and time complexity.

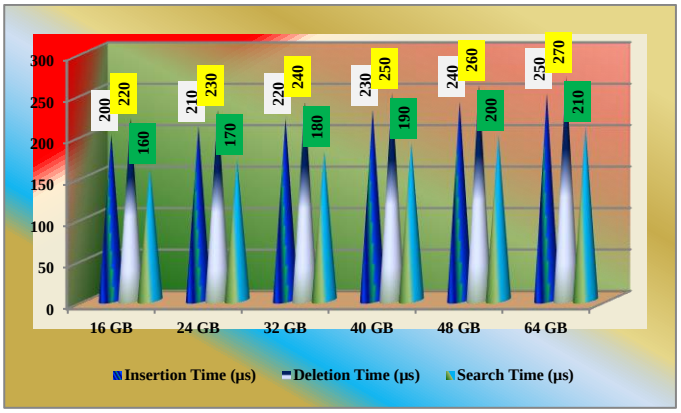
insertionTimes, searchTimes, deletionTimes: Arrays to store operation durations.runtime.MemStats: Captures memory usage statistics. inserts, searches, and deletes keys, recording time for each operation. Insertion/Deletion/Search Times: Captured using time.Now() and computing the elapsed time before and after the operation. CPU Usage: Monitored with libraries like gopsutil. Space complexity [36][37] for insertion , deletion and search operation is O(n) where n is the number of keys and time complexity is O(logn).

Store Size	Ins (µs)	Del (µs)	Sea (µs)	CPU (%)	S-Comp	T-Comp
16 GB	200	220	160	25	O(n)	O(log n)
24 GB	210	230	170	28	O(n)	O(log n)
32 GB	220	240	180	30	O(n)	O(log n)
40 GB	230	250	190	32	O(n)	O(log n)
48 GB	240	260	200	35	O(n)	O(log n)
64 GB	250	270	210	37	O(n)	O(log n)

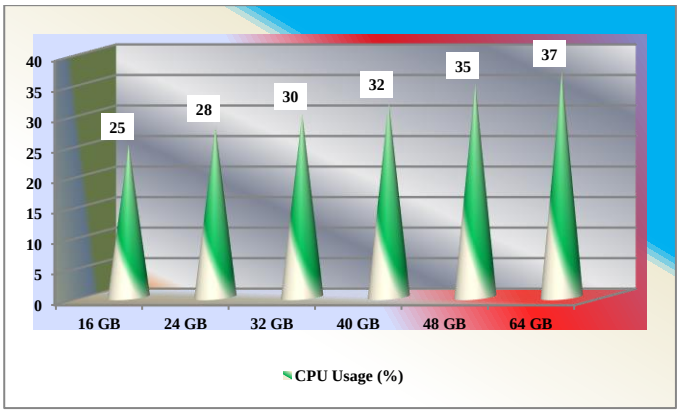
Table 13: ETCD Operational Metrics Badger DB -1

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 shows the different parameters of the Badger DB implementation of the data store.



Graph 19: ETCD Operational Metrics : Badger DB – 1



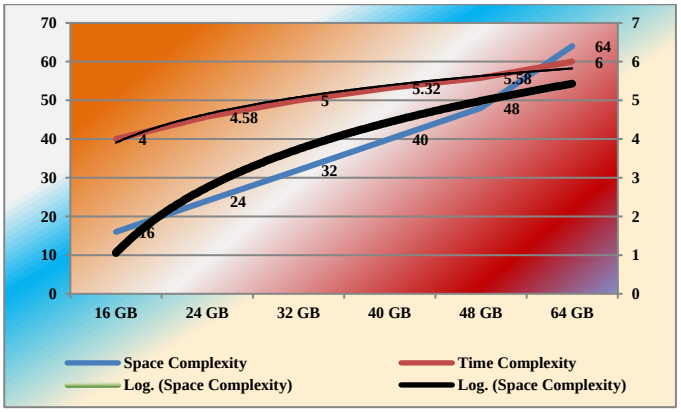
Graph 20: ETCD Badger DB CPU Usage-1

Graph 20 shows the CPU usage of the ETCD data store having the Badger DB 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 14: ETCD Badger DB Complexity-1

Table 14 carries the values for Space and Time complexity for Badger DB 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.



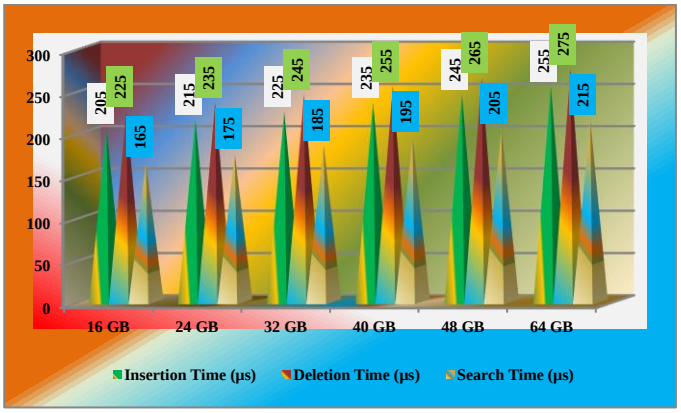
Graph 21: ETCD Badger DB Complexity-1

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.

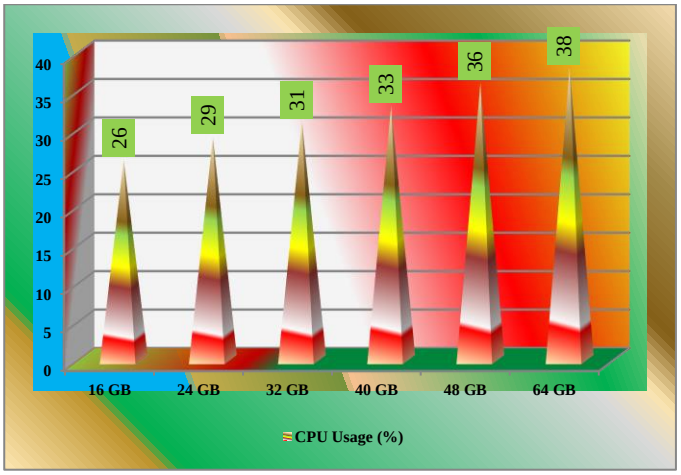
Store Size	Ins (µs)	Del (µs)	Sea (µs)	CPU (%)	S- Comp	T-Comp
16 GB	205	225	165	26	O(n)	O(log n)
24 GB	215	235	175	29	O(n)	O(log n)
32 GB	225	245	185	31	O(n)	O(log n)
40 GB	235	255	195	33	O(n)	O(log n)
48 GB	245	265	205	36	O(n)	O(log n)
64 GB	255	275	215	38	O(n)	O(log n)

Table 15: ETCD Operational Metrics Badger DB - 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 Operational Metrics Badger DB - 2



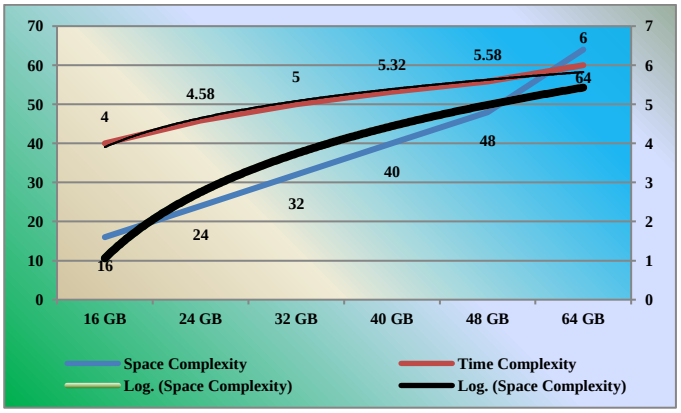
Graph 23: ETCD Badger DB CPU Usage-2

Graph 22 shows the operational metrics for different sizes of the ETCD . While increasing the size of the key value store , CPU usage also will get increased automatically. Graph 23 shows the same.

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 16: ETCD Badger DB Complexity-2

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



Graph 24: ETCD Badger DB Complexity-2

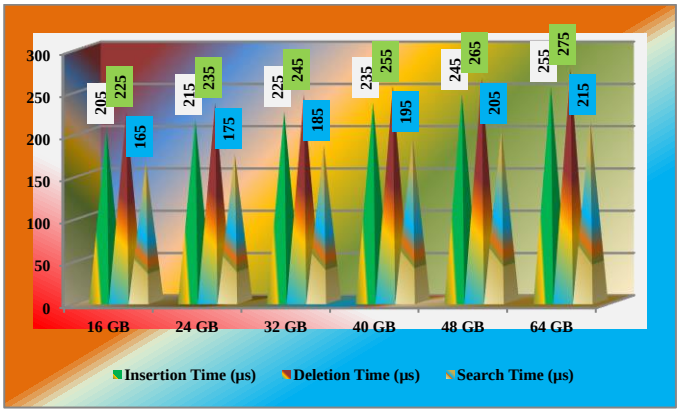
Please find the Logarithmic graph using the calculation, O(log n) ≈ 4 (using base 2 logarithm), O(n) = 16, 24, 32, 40, 48 and 64 for the n values from the size of the store which we have mentioned in the table. Graph 24 shows the same values. It is using two scale Y-Axis since the table is carrying two 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	205	225	165	26	O(n)	O(log n)
24 GB	215	235	175	29	O(n)	O(log n)
32 GB	225	245	185	31	O(n)	O(log n)
40 GB	235	255	195	33	O(n)	O(log n)
48 GB	245	265	205	36	O(n)	O(log n)
64 GB	255	275	215	38	O(n)	O(log n)

Table 17 : ETCD Operational Metrics Badger DB - 3

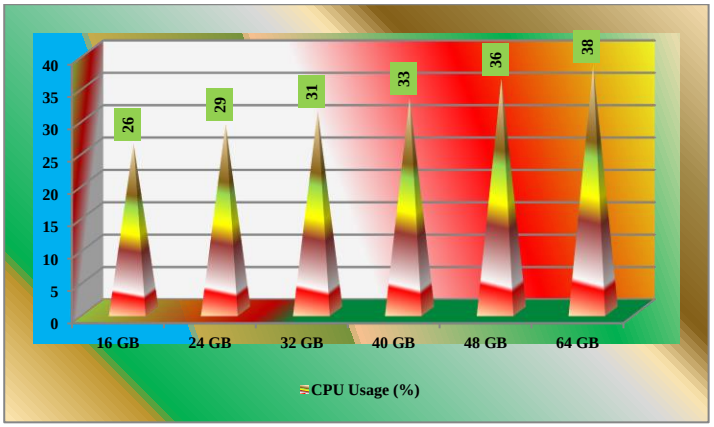
Table 17, shows the fourth sample of the data from ETCD store. ETCD Stores a key-value pair in etcd, Syntax: etcdctl put <key> <value>, etcdctl put message "Hello, world!"

- 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 Operational Metrics Badger DB - 3

BadgerDB uses a log-structured merge (LSM) tree for efficient storage. BadgerDB supports compression to reduce storage requirements. BadgerDB has a built-in cache for faster data retrieval. BadgerDB supports concurrent access for multiple readers and writers. Graph 25 shows the collection of operations metrics for 16GB, 24GB , 32GB , 40GB , 48GB and 64GB ETCD store size.



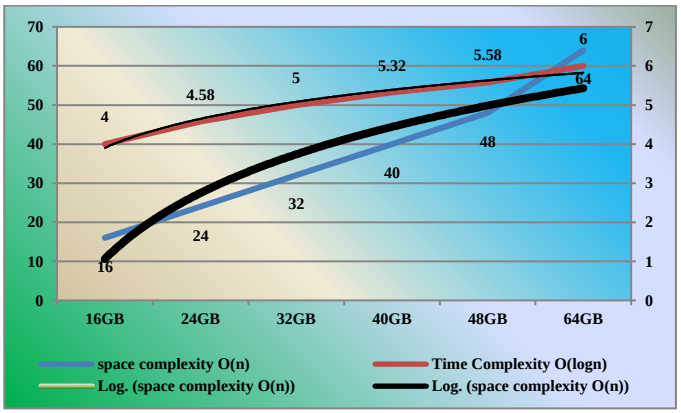
Graph 26: ETCD Badger DB CPU Usage-3

Graph 26 shows the CPU usage of third sample for different operational activities like insertion , deletion and searching the key.

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 Badger DB Complexity-3

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



Graph 27: ETCD Badger DB 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.

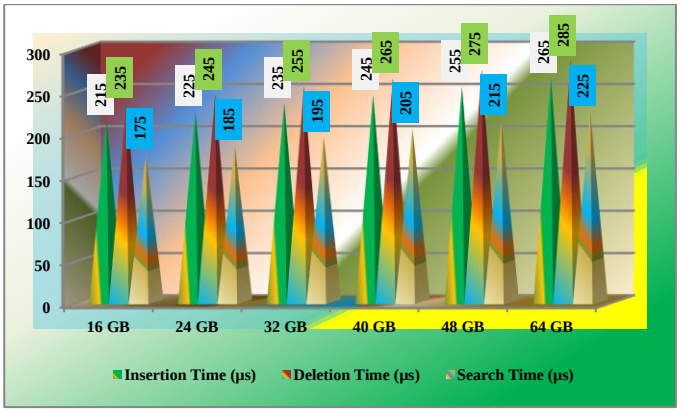
Store Size	Ins (µs)	Del(µs)	Sea(µs)	CPU (%)	S-Comp	T-Comp
16 GB	215	235	175	28	O(n)	O(log n)
24 GB	225	245	185	31	O(n)	O(log n)
32 GB	235	255	195	33	O(n)	O(log n)
40 GB	245	265	205	35	O(n)	O(log n)
48 GB	255	275	215	38	O(n)	O(log n)
64 GB	265	285	225	40	O(n)	O(log n)

Table 19: ETCD Operational Metrics Badger DB -4

Table 19 shows the ETCD Badger DB 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).

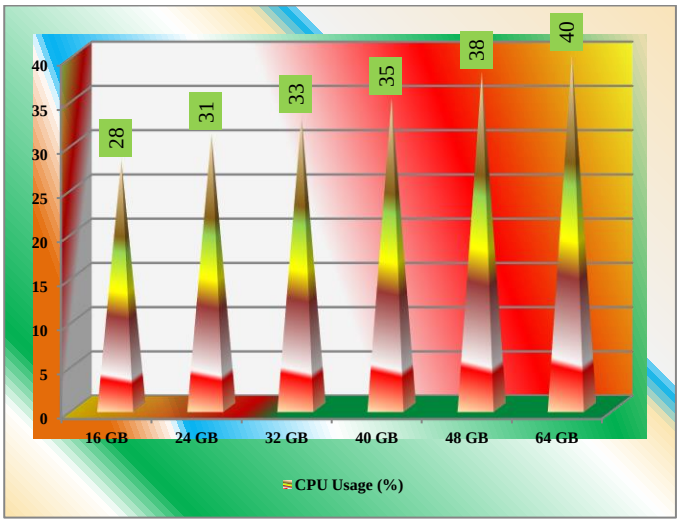
ETCD GET operation retrieves a value from the store and the syntax , etcdctl get <key>, etcdctl get

/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 Operational Metrics Badger DB - 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.

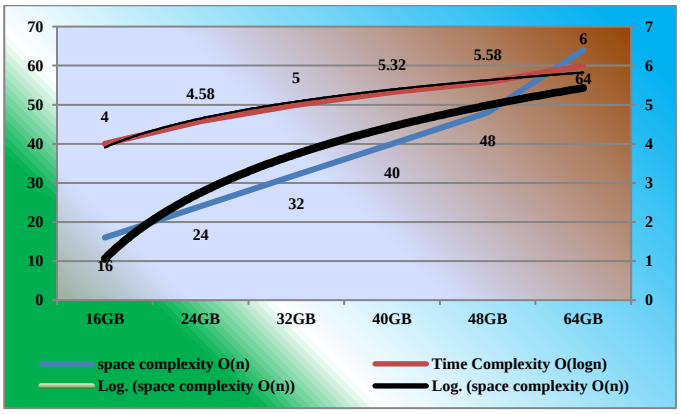


Graph 29: ETCD Badger DB CPU Usage-4

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 Badger DB Complexity-4

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



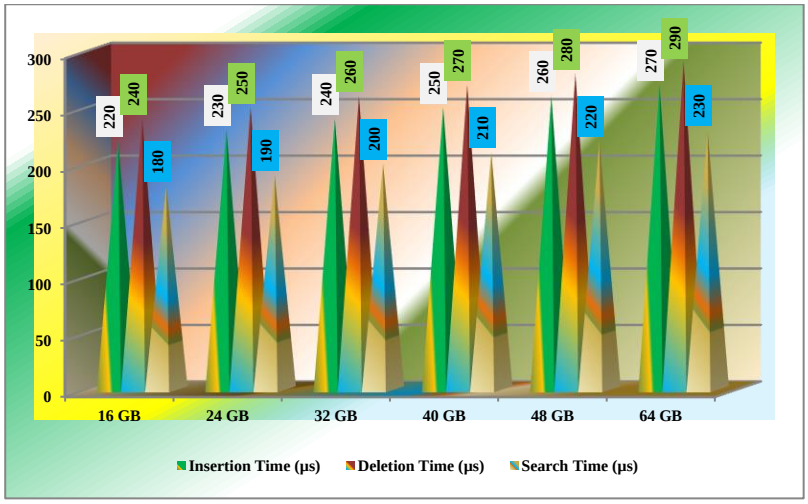
Graph 30: ETCD – Badger DB Complexity-4

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.

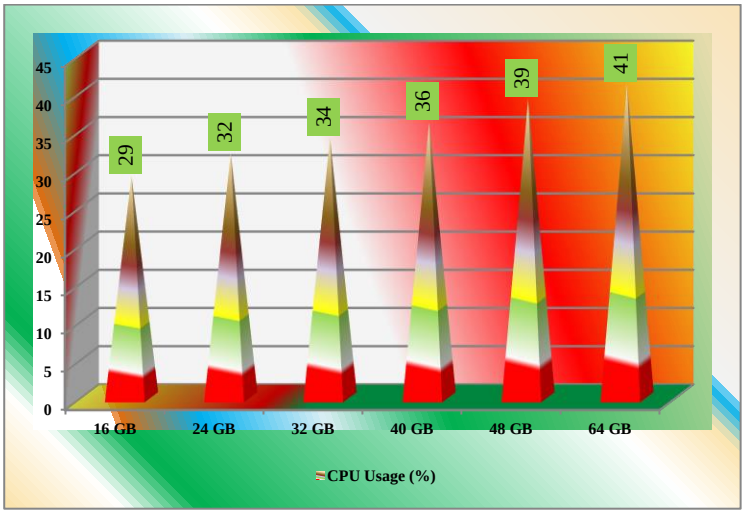
Store Size	Ins (µs)	Del (µs)	Sea (µs)	CPU (%)	S-Comp	T-Comp
16 GB	220	240	180	29	O(n)	O(log n)
24 GB	230	250	190	32	O(n)	O(log n)
32 GB	240	260	200	34	O(n)	O(log n)
40 GB	250	270	210	36	O(n)	O(log n)
48 GB	260	280	220	39	O(n)	O(log n)
64 GB	270	290	230	41	O(n)	O(log n)

Table 21: ETCD Operational Metrics Badger DB - 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 all parameters from the fifth sample.



Graph 31: ETCD Parameters : Badger DB – 5

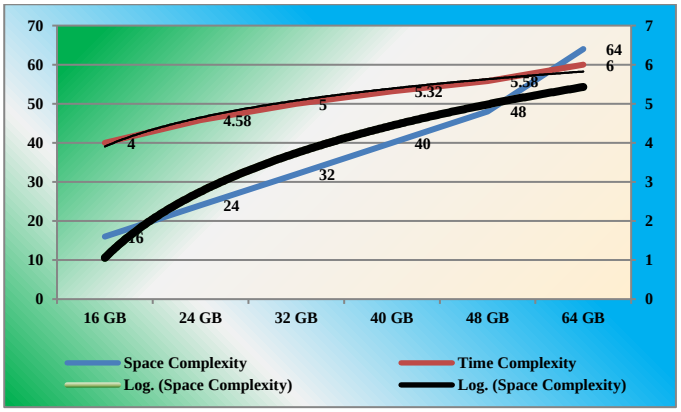


Graph 32: ETCD Badger DB CPU Usage-5

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 22: ETCD Badger DB Complexity-5

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



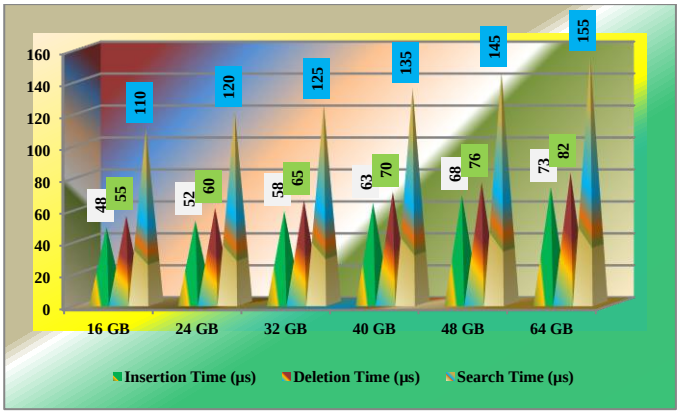
Graph 33: ETCD Badger DB Complexity-5

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	48	55	110	20	O(n)	O(log n)
24 GB	52	60	120	27	O(n)	O(log n)
32 GB	58	65	125	34	O(n)	O(log n)
40 GB	63	70	135	40	O(n)	O(log n)
48 GB	68	76	145	47	O(n)	O(log n)
64 GB	73	82	155	52	O(n)	O(log n)

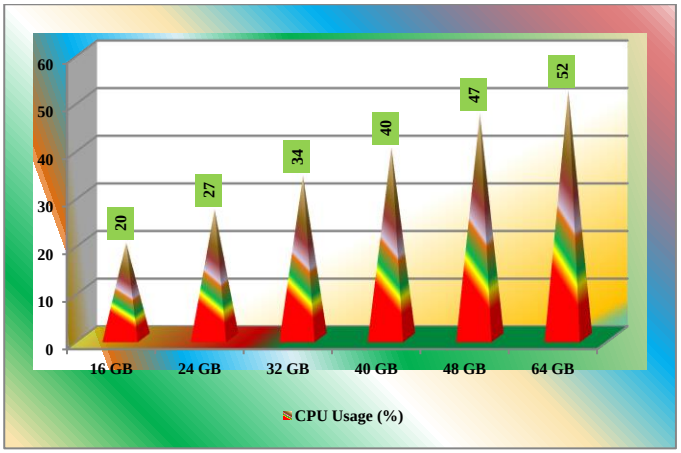
Table 23: ETCD Operational Metrics Badger DB -6

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



Graph 34: ETCD Operational Metrics Badger DB - 6

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



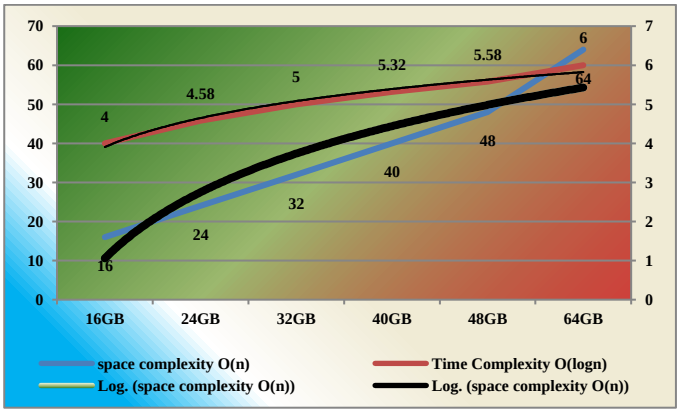
Graph 35: ETCD – Badger DB CPU Usage-6

Graph 35 shows the cpu usage of ETCD having Badger DB 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

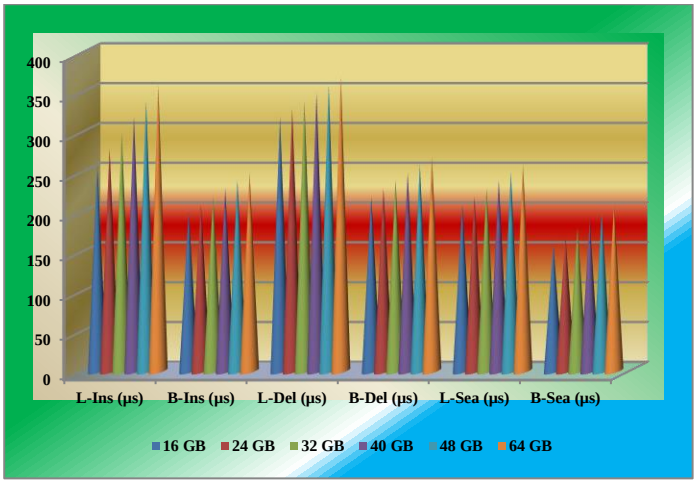
Table 24: ETCD Badger DB Complexity-6

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



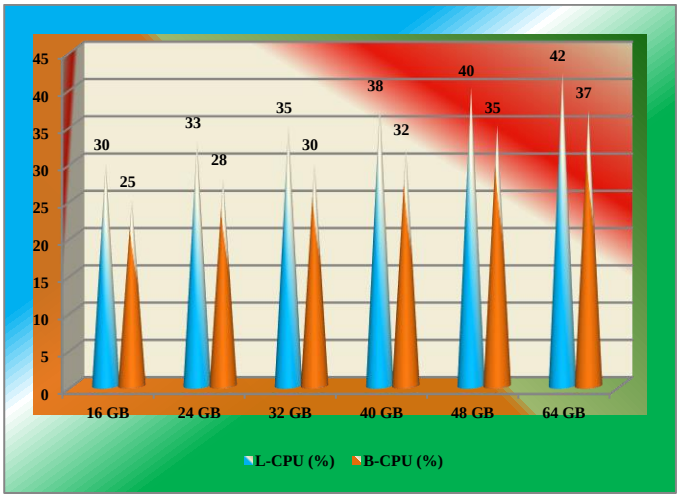
Graph 36: ETCD Badger DB Complexity-6

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.



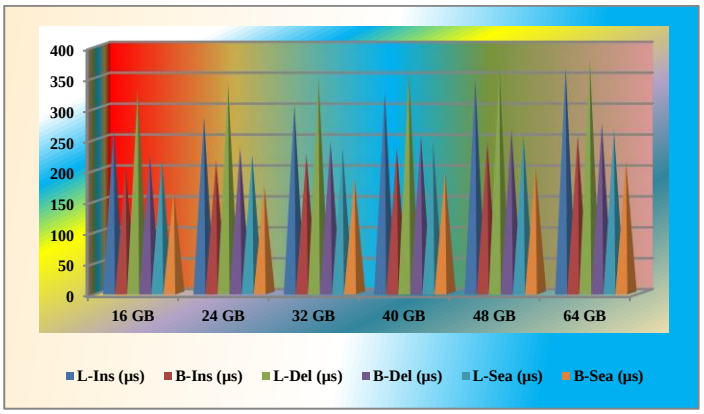
Graph 37: ETCD LedgerDB Vs Badger DB -1.1

Graph 37, shows the Insertion time difference between LEDGER DB and Badger DB implementation. As per the graph the insertion time trend is going down while moving from LedgerDB to Badger DB implementation. The same observation we can have with other parameters like deletion time and search time.



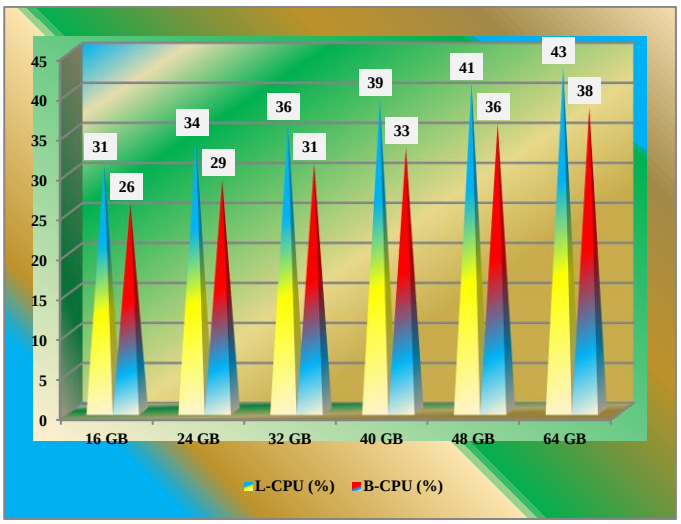
Graph 38: ETCD LedgerDB Vs Badger DB -1.2

Graph 38 shows the CPU usage difference between LEDGER DB implementation and Badger DB implementation. CPU usage is going low once we are dealing with Badger DB in the implementation.



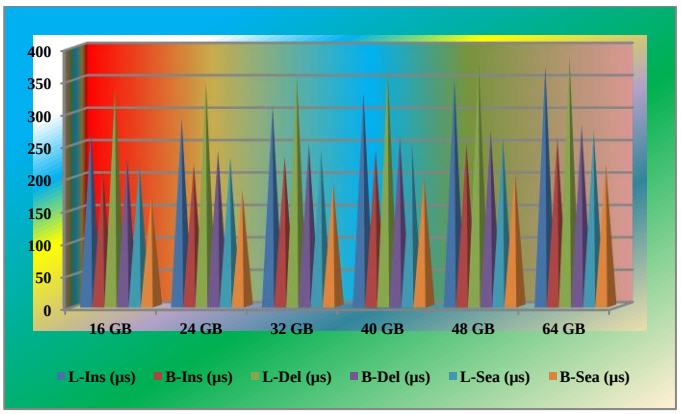
Graph 39: ETCD LedgerDB Vs Badger DB -2.1

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



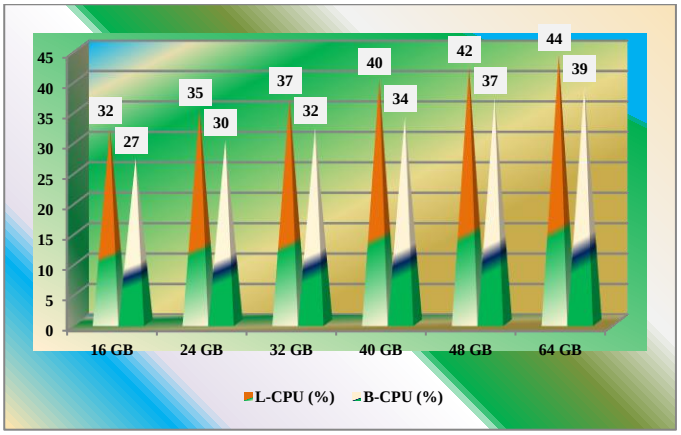
Graph 40: ETCD LedgerDB Vs Badger DB -2.2

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



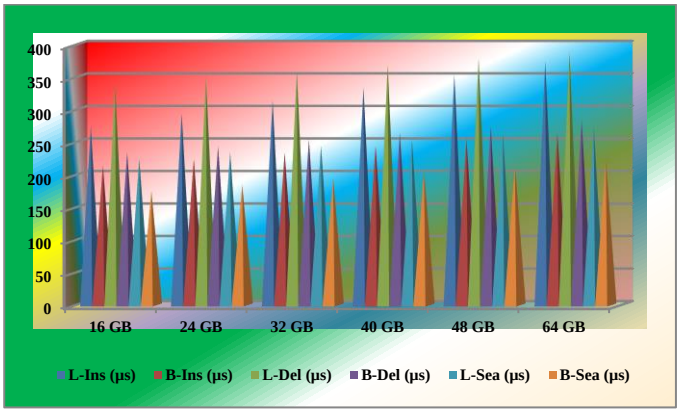
Graph 41: ETCD LedgerDB Vs Badger DB -3.1

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



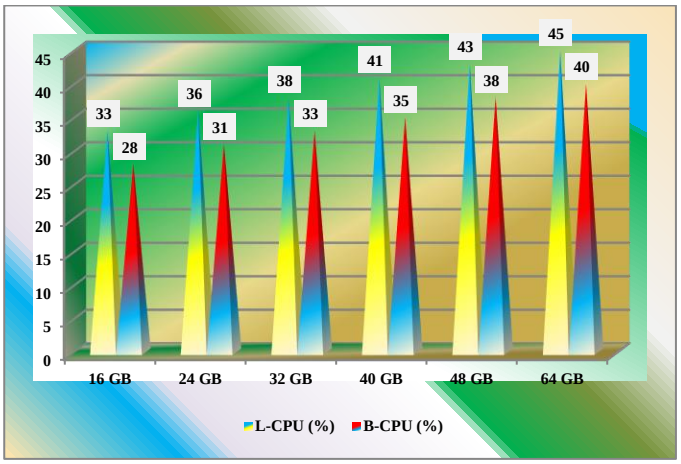
Graph 42: ETCD LedgerDB Vs Badger DB -3.2

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



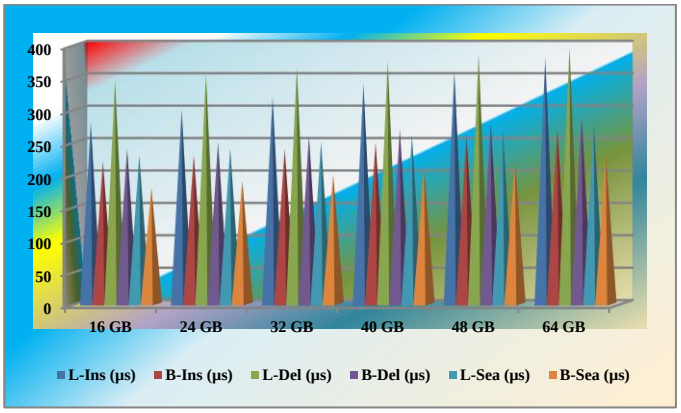
Graph 43: ETCD LedgerDB Vs Badger DB -4.1

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



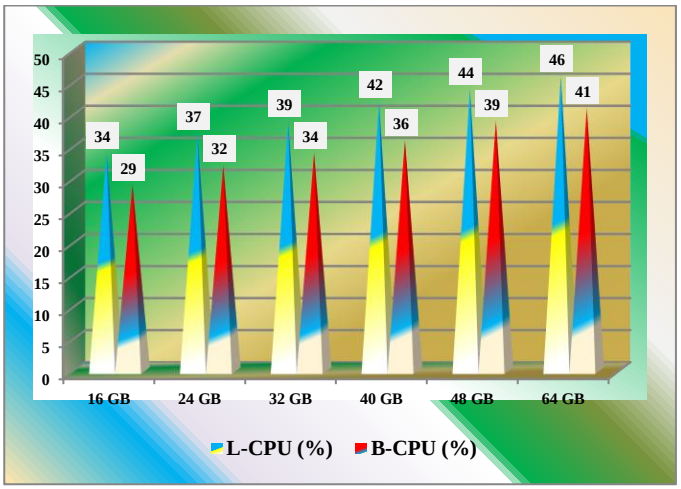
Graph 44: ETCD LedgerDB Vs Badger DB-4.2

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



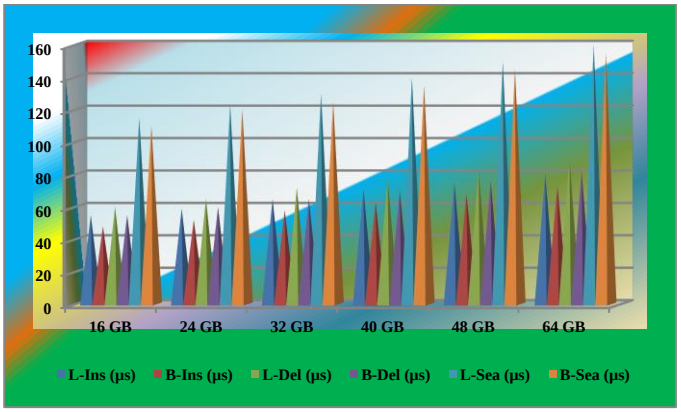
Graph 45: ETCD LedgerDB Vs Badger DB -5.1

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



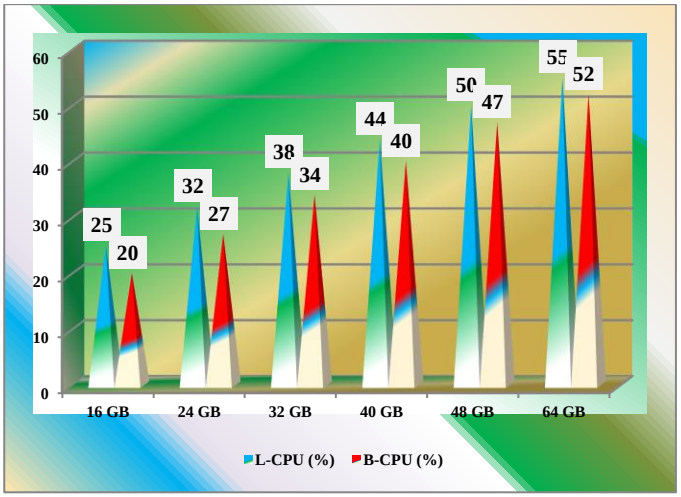
Graph 46: ETCD LedgerDB Vs Badger DB -5.2

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



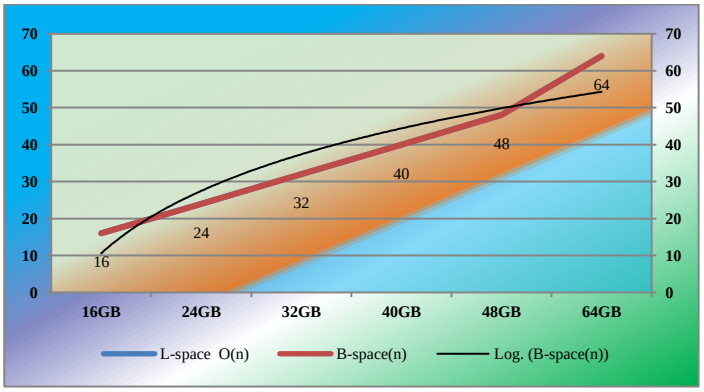
Graph 47: ETCD LedgerDB Vs Badger DB -6.1

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



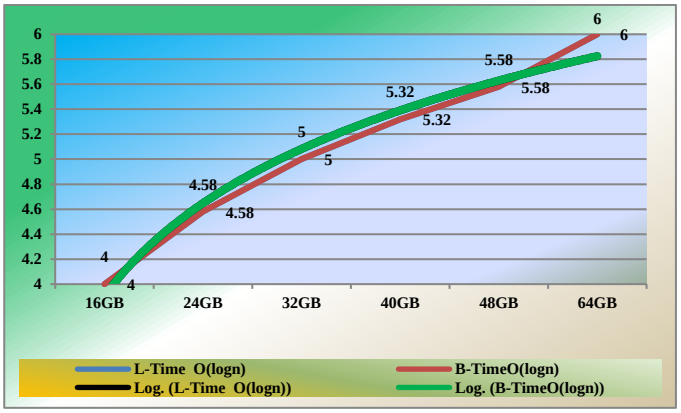
Graph 48: ETCD Ledger DB Vs Badger DB -6.2

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



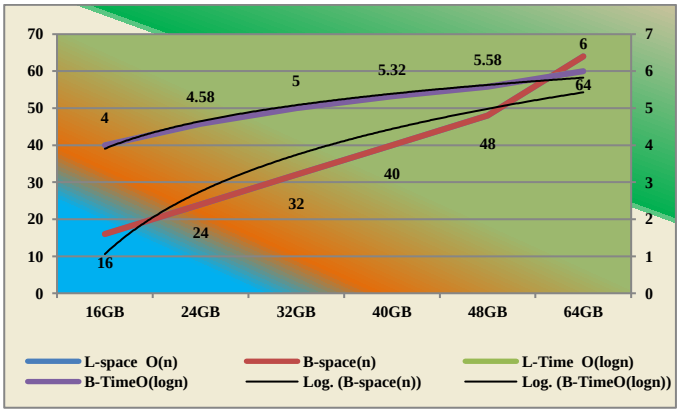
Graph 49: ETCD LedgerDB Vs Badger DB - Space Complexities

Graph 49 shows the space complexities comparison for the LedgerDB and Badger DB implementation of the key value store.



Graph 50: ETCD LedgerDB Vs Badger DB - Time Complexities

Graph 50 shows the comparison of time complexities between LedgerDB and Badger DB implementation of the ETCD.



Graph 51: ETCD LedgerDB Vs Badger DB Time and Space complexities

Graph 49 , 50 and 51 shows the comparison of complexities between LedgerDB and Badger DB implementation.

EVALUATION

The comparison of LedgerDB implementation results with Badger DB 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) with Badger DB instead of Ledger DB.

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 Badger DB implementation compared to Ledger DB

implementation. Space complexity and time complexity are also compared, along with CPU usage . Complexities are almost same , while CPU usage values are going down.

LedgerDB is Best for sequential workloads in lightweight or edge Kubernetes clusters. Badger DB Ideal for high-throughput Kubernetes ETCD storage with minimal write amplification.

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

Future work : BadgerDB often requires more memory due to its reliance on bloom filters and in-memory tables to optimize reads. In contrast, LedgerDB might use a more memory-efficient approach, especially in constrained environments. Need to work on minimizing the memory consumption.

REFERENCES

1. "etcd: A Distributed, Reliable Key-Value Store for the Edge" by Corey Olsen et al. (2018)
2. Networking Analysis and Performance Comparison of Kubernetes CNI Plugins, 28 October 2020, pp 99–109, Ritik Kumar & Munesh Chandra Trivedi.
3. "etcd: A Highly-Available, Distributed Key-Value Store" by Brandon Philips et al. (2014), Proceedings of the 2014 ACM SIGOPS Symposium on Cloud Computing.
4. "Optimizing Kubernetes for Low-Latency Applications" by IBM (2020).
5. "Performance Analysis of Kubernetes Clusters" by Microsoft (2018).
6. "Secure Kubernetes Deployment" by Palo Alto Networks (2019)".
7. LedgerDB: A Centralized Ledger Database for Universal Audit and Verification, Xinying Yang, Yuan Zhang, Sheng Wang, Benquan Yu, Feifei Li , Yize Li , Wenyuan Yan., August 2020.
8. Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability, Shixiong Qi; Sameer G. Kulkarni; K. K. Ramakrishnan, 25 December 2020 , IEEEXplore.
9. LEDGER DB and Red Black tree as a single balanced tree, March 2016, Zegour Djamel Eddine, Lynda Bounif
10. Configure Default Memory Requests and Limits for a Namespace https://orielly.ly/ozlUi1
11. Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned, Leila Abdollahi Vayghan Montreal, Mohamed Aymen Saied; Maria Toeroe; Ferhat Khendek, IEEE XPlore.
12. High Availability Storage Server with Kubernetes, Ali Akbar Khatami; Yudha Purwanto; Muhammad Faris Ruriawan, 2020, IEEE Xplore.
13. Improving Application availability with Pod Readiness Gates https://orielly.ly/h_WiG
14. Kuberenets in action by Marko Liksa , 2018.
15. 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.
16. 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.
17. "Kubernetes Network Policies" by Calico (2019).
18. Kubernetes Best Practices , Burns, Villaibha, Strebel , Evenson.
19. Kubernetes Best Practices: Resource Requests and limits https://orielly.ly/8bKD5

20. "Kubernetes Network Security" by Cisco (2018).
21. Kubernetes Container Orchestration as a Framework for Flexible and Effective Scientific Data Analysis, IEEE Xplore, 13 February 2020.
22. Kubernetes CSI Driver for mounting images https://orielly.ly/OMqRo
23. Kubernetes Patterns, Ibryam , Hub
24. Core Kubernetes , Jay Vyas , Chris Love.
25. Kubernetes Persistent Storage by Google (2018).
26. Kubernetes Scalability and Performance" by Red Hat (2019).
27. Kubernetes Storage Performance by Red Hat (2019).
28. Learning Core DNS, Belamanic, Liu.
29. "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).
30. Modelling performance & resource management in kubernetes by Víctor Medel, Omer F. Rana, José Ángel Bañares, Unai Arronategui.
31. 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.
32. A Portable Load Balancer for Kubernetes Cluster, 28 January 2018, Kimitoshi Takahashi, Kento Aida, Tomoya Tanjo, Jingtao SunAuthors Info & Claims.
33. Impact of etcd deployment on Kubernetes, Istio, and application performance,William Tärneberg, Cristian Klein, Erik Elmroth, Maria Kihl, 07 August 2020.
34. On the Performance of etcd in Containerized Environments" by Luca Zanetti et al. (2020), IEEE International Conference on Cloud Computing (CLOUD).
35. Performance Evaluation of etcd in Distributed Systems" by Jiahao Chen et al. (2020), 2020 IEEE International Conference on Cloud Computing (CLOUD).
36. Rearchitecting Kubernetes for the Edge, Andrew Jeffery, Heidi Howard, Richard MortierAuthors Info & Claims, 26 April 2021.
37. Reliability Analysis of Kubernetes Distributed Systems" by University of California (2020).
38. 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.
39. Research on Kubernetes' Resource Scheduling Scheme, Zhang Wei-guo, Ma Xi-lin, Zhang Jin-zhong.
40. The log-structured merge-tree (LEDGER DB-tree),June 1996, Patrick O'Neil, Edward Cheng, Dieter Gawlick & Elizabeth O'Neil.
41. Scalable and Reliable Kubernetes Clusters" by Google (2018).
42. Study on the Kubernetes cluster mocel, Sourabh Vials Pilande. International Journal of Science and Research , ISSN : 2319-7064.
43. 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.
44. 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