International Journal of Leading Research Publication (IJLRP)

E-ISSN: 2582-8010 ● Website: www.ijlrp.com ● Email: editor@ijlrp.com

Improving Durability In Kubernetes ETCD by Reducing Write Amplification

Satya Ram Tsaliki
Developer III, Vitamix corporation, USA.
satyaram.tsaliki@outlook.com

Dr.B.PurnachandraRao
Sr. Solutions Architect, HCL Technologies , Bangalore, Karnataka, India.
pcr.bobbepalli@gmail.com

Abstract

Kuberntetes is an open source platform for hosting containers and defining application-centric APIs for managing cloud semantics around how these containers are provisioned with storage, networking , security, and other resources. Kubernetes enables continuous reconciliation of the entire state space of your application deployments, including how they are accessed from the outside world. The platform uses a declarative configuration model, enabling users to define desired application states. Kubernetes also provides advanced networking features, persistent storage support, and monitoring tools. Its extensibility allows for custom integrations and add-ons. With its focus on resource optimization, Kubernetes is a go-to choice for modern DevOps practices. 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. 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. If an application writes 1 GB of data, but the storage system writes 3 GB due to internal operations (e.g., compaction, garbage collection), the write amplification factor (WA) would be 3.0. In this paper we will discuss about implementation of ETCD using Ledger DB and Badger DB. Badger DB is showing high performance in write amplification wrt ledger DB implementation.

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

INTRODUCTION

Apps have dependencies that must be fulfilled by the host on which they run. Developers in the pre container era accomplished this task in an ad hoc manner (for example a java app would require a running JVM along with firewall rules to talk to a database). 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 is an open-source, distributed key-value store that provides a reliable way to store and manage data in a distributed system. And the APIs are put to Store a key-value pair[2], 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 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 [3] 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 Kubernetes cluster is a set of machines, called nodes, that work together to manage, deploy, and scale containerized applications. It consists of two main components: the control plane and the worker nodes. The control plane manages the overall cluster, handling scheduling, maintaining desired states, and exposing the API. Worker nodes run the containers and workloads assigned by the control plane. Kubernetes clusters abstract infrastructure details, providing a consistent platform for application deployment. They ensure high availability, scalability, and fault tolerance by balancing workloads and restarting failed containers. Clusters can run on various environments, including on-premises servers, cloud platforms, or hybrid setups.

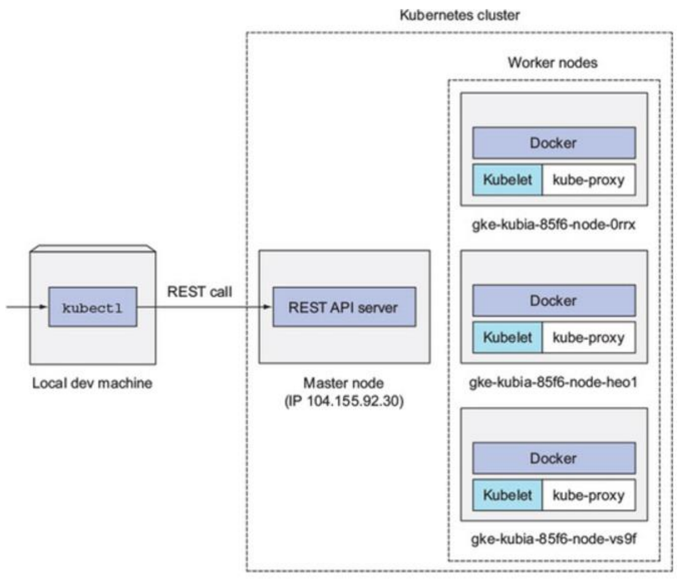


Fig: 1 Cluster Architecture

Fig 1. Shows the Kubernetes cluster architecture. Each node runs Docker, the Kubelet and the kube-proxy. You'll interact with the cluster through the kubectl command line client, which issues REST requests to the Kubernetes API server running on the master node. 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 [4] 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 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 store that holds the state and configuration of the cluster, including information about pods, services, secrets, and configurations. Scheduler [5] 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 [6] is the software responsible for running containers (e.g., Docker, containerd). Kube-proxy manages network traffic between pods and services, handling routing, load balancing, and network rules. The kubernetes cluster is having objects like pods, nodes, services.

The pods run on worker nodes and are managed by the control plane [7]. 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 [8] 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 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 [9] 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. Each of those pods has its own IP address [10]. 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 [11] in Kubernetes provides a way to define a logical set of Pods and a policy by which to access them.

ClusterIP The default type, which exposes the service on a cluster-internal IP. Only accessible from within the cluster. Iptables [12] 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.

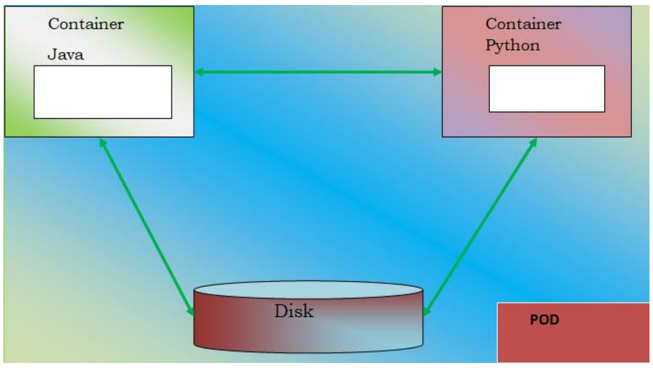


Fig 2: Kubernetes POD

Fig 2. Shows the kubernets pod , it is the atomic unit of scheduling. That means the scheduler tries to find a host that satisfies the requirements of all containers that belong to the pod. If the pod has created with many containers , the scheduler needs to find a host that has enough resources to satisfy all container demands combined.

A pod ensures a collection collocation of containers. The most common ways of communicating include using a shared local file system for exchanging data , using the localhost network interface or us8ing some host interprocess communication IPC mechanism for high performance interactions.

ETCD stores data in a distributed manner, ensuring high availability and reliability Distributed systems overcome various limitations of a centralized system and offer several advantages like high performance, increased availability [13], 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 [14]. 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.

In Kubernetes, everything is represented by an object, which serves as the fundamental unit of its architecture. Objects define the desired state of the system and are abstractions of system components, such as pods, services, deployments [15] , and more. Each object consists of a spec, which describes the desired state, and status, which represents the current state. Kubernetes uses these objects to manage and maintain applications and resources declaratively.

For example, a Pod object represents a group of containers running together, while a Service object defines network access policies. Objects are stored in the ETCD database, ensuring cluster state consistency. Users interact with objects through YAML or JSON configurations and manage them using the kubectl command-line tool. This object-centric design allows Kubernetes to simplify complex orchestration [16] tasks, making workloads portable and infrastructure-agnostic.

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 [17] notices that an instance is assigned to the Kubelet's node. It runs the application instance via 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 [18] monitor the system and keep the applications running.

A request is sent to the service's stable IP address. Kubernetes Networking [19], 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 in Kubernetes ensures efficient distribution of network traffic across multiple pods or services to optimize resource utilization and reliability. It prevents overloading any single component by distributing requests evenly. Kubernetes achieves this using built-in mechanisms like Services [20] with cluster IPs or external load balancers. This process enhances application performance, scalability, and fault tolerance.

Service abstraction in Kubernetes provides a simplified and stable interface for accessing application components, while iptables [21] 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 [22] platform without any hassles.

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.

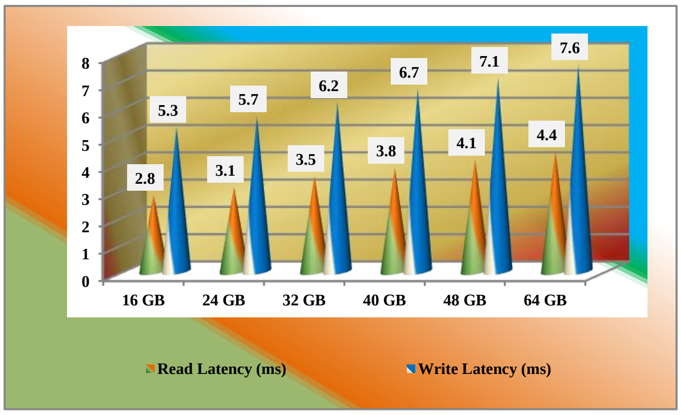
```
package main
import (
	"fmt"
	"log"
	"time"
	"github.com/ledgerdb/ledgerdb-go"
	"runtime"
)
func collectLedgerMetrics(db *ledgerdb.DB, key string, value []byte) {
	start := time.Now()
	err := db.Put([]byte(key), value)
	if err != nil {
		log.Fatalf("Write Error: %v", err)
	}
	writeLatency := time.Since(start)
	start = time.Now()
	_, err = db.Get([]byte(key))
	if err != nil {
		log.Fatalf("Read Error: %v", err)
	}
	readLatency := time.Since(start)
	var memStats runtime.MemStats
	runtime.ReadMemStats(&memStats)
	fmt.Printf("LedgerDB Metrics:\n")
	fmt.Printf("Write Latency: %v\n", writeLatency)
	fmt.Printf("Read Latency: %v\n", readLatency)
	fmt.Printf("Memory Usage: %v MB\n", memStats.Alloc/1024/1024)
	fmt.Printf("Write Amplification: Lower due to append-only logging\n")
}
func main() {
	db, err := ledgerdb.Open("./ledgerdb")
	if err != nil {
		log.Fatalf("Error opening LedgerDB: %v", err)
	}
	defer db.Close()
	collectLedgerMetrics(db, "testKey", []byte("testValue"))
}
```

Write Latency Measures the time taken to write a key-value pair. Read Latency Measures the time taken to read a key-value pair. Memory Usage Uses Go's runtime.MemStats to calculate memory usage. Write Amplification LedgerDB: Write amplification is generally lower because of its append-only log structure[36][37].

Store Size	Read Latency (ms)	Write Latency (ms)	Write Amplification	Memory Usage (MB)
16 GB	2.8	5.3	7	120
24 GB	3.1	5.7	7.5	140
32 GB	3.5	6.2	8	160
40 GB	3.8	6.7	8.5	180
48 GB	4.1	7.1	9	200
64 GB	4.4	7.6	9.5	220

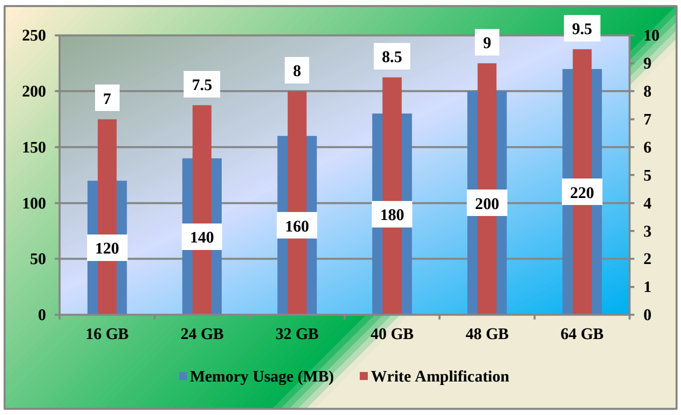
Table 1: Read and write latency: Ledger DB- 1

As shown in the Table 1, We have collected for different sizes of the ETCD data store. We have collected the metrics for read latency, write latency , write amplification and memopry usage. As usual , the values are getting increased while the size of the ETCD data store is growing up.



Graph 1: Read and write latency: Ledger DB- 1

Graph 1 shows the Read latency and write latency for different ETCD sizes like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.



Graph 2: Memory usage and write amplification:

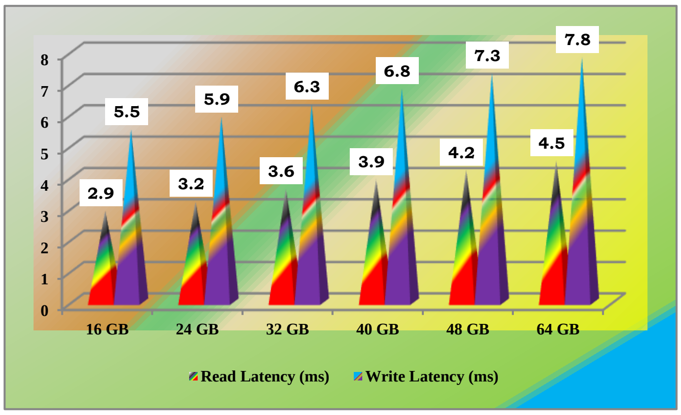
Ledger DB- 1

Graph 2 shows the memory usage and write amplification for the ETCD data store having the Ledger DB implementation. It shows the two scale Y axis plot since we are having two different ranges of data i.e, memory usage from 0 to 250 and write amplification from 0 to 10. Amplification parameter is dimensionless and Write amplification is typically expressed as a ratio, such as WA = Total Data Written to Storage / Total Data Written by Application. If an application writes 1 GB of data, but the storage system writes 3 GB due to internal operations (e.g., compaction, garbage collection), the write amplification factor (WA) would be 3.0.

Store Size	Read Latency (ms)	Write Latency (ms)	Write Amplification	Memory Usage (MB)
16 GB	2.9	5.5	7.2	125
24 GB	3.2	5.9	7.7	145
32 GB	3.6	6.3	8.2	165
40 GB	3.9	6.8	8.7	185
48 GB	4.2	7.3	9.2	205
64 GB	4.5	7.8	9.7	225

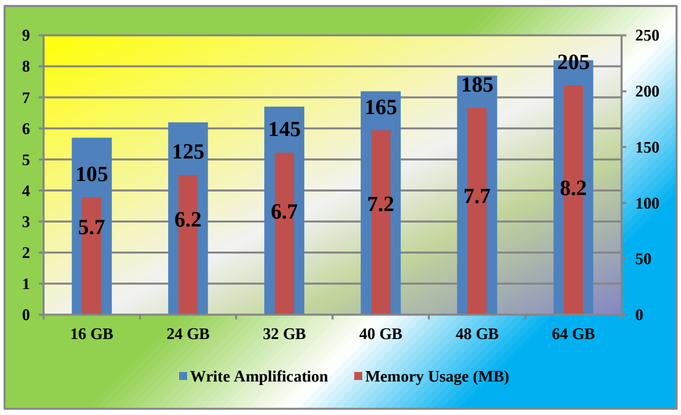
Table 2: Read and write latency: Ledger DB- 2

As shown in the Table 2, We have collected for different sizes of the ETCD data store sizes like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB. We have collected the metrics for read latency, write latency , write amplification and memory usage. As usual , the values are getting increased while the size of the ETCD data store is growing up.



Graph 3: Read and write latency: Ledger DB- 2

Graph 3 shows the Read latency and write latency for different ETCD sizes like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB. Read latency, also known as read access time, is the time it takes for a storage device to retrieve data from its storage media. It's the delay between the moment a read request is sent to the storage device and the moment the requested data is returned.



Graph 4: Memory usage and write amplification:

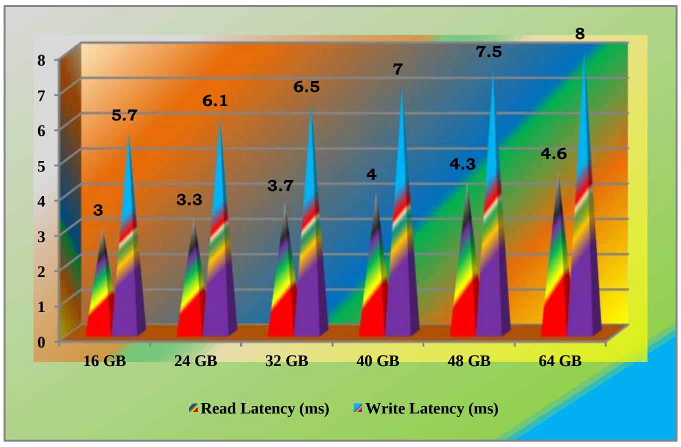
Ledger DB - 2

Graph 4 shows the memory usage and write amplification for the ETCD data store having the Ledger DB implementation. It shows the two scale Y axis plot since we are having two different ranges of data i.e, memory usage from 0 to 250 and write amplification from 0 to 10. Write amplification is a phenomenon where the actual amount of data written to a storage device exceeds the logical amount of data intended to be written, caused by factors such as garbage collection, wear leveling, and metadata management, ultimately leading to decreased storage efficiency and reliability.

Store Size	Read Latency (ms)	Write Latency (ms)	Write Amplification	Memory Usage (MB)
16 GB	3	5.7	7.4	130
24 GB	3.3	6.1	7.9	150
32 GB	3.7	6.5	8.4	170
40 GB	4	7	8.9	190
48 GB	4.3	7.5	9.4	210
64 GB	4.6	8	9.9	230

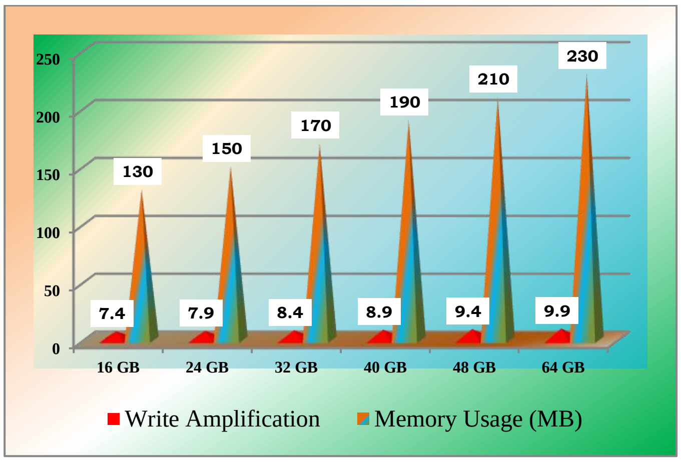
Table 3: Read and write latency: Ledger DB- 3.

As shown in the Table 3, We have collected for different sizes of the ETCD data store sizes like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB. We have collected the metrics for read latency, write latency , write amplification and memory usage. The values are getting increased while the size of the ETCD data store is growing up.



Graph 5 : Read and write latency: Ledger DB- 3

Write latency, also known as write access time, is the time it takes for a storage device to write data to its storage media. It's the delay between the moment a write request is sent to the storage device and the moment the data is successfully written. Graph 5 shows the read latency and write latency.



Graph 6: Memory usage and write amplification:

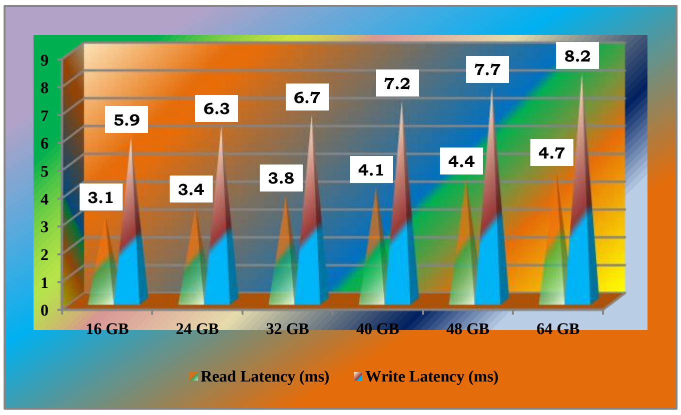
Ledger DB 3

Graph 6 shows the memory usage and write amplification for the ETCD data store having the Ledger DB implementation.

Store Size	Read Latency (ms)	Write Latency (ms)	Write Amplification	Memory Usage (MB)
16 GB	3.1	5.9	7.6	135
24 GB	3.4	6.3	8.1	155
32 GB	3.8	6.7	8.6	175
40 GB	4.1	7.2	9.1	195
48 GB	4.4	7.7	9.6	215
64 GB	4.7	8.2	10.1	235

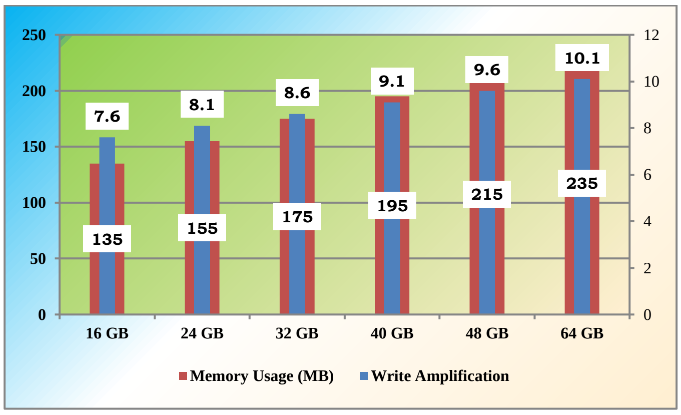
Table 4: Read and write latency: Ledger DB- 4

As shown in the Table 3, We have collected for different sizes of the ETCD data store sizes like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB. We have collected the metrics for read latency, write latency , write amplification and memory usage. The values are getting increased while the size of the ETCD data store is growing up.



Graph 7 : Read and write latency: Ledger DB- 4

Graph 7 shows the read latency and write latency from the fifth sample of the Ledger DB implementation of the key value store (ETCD).



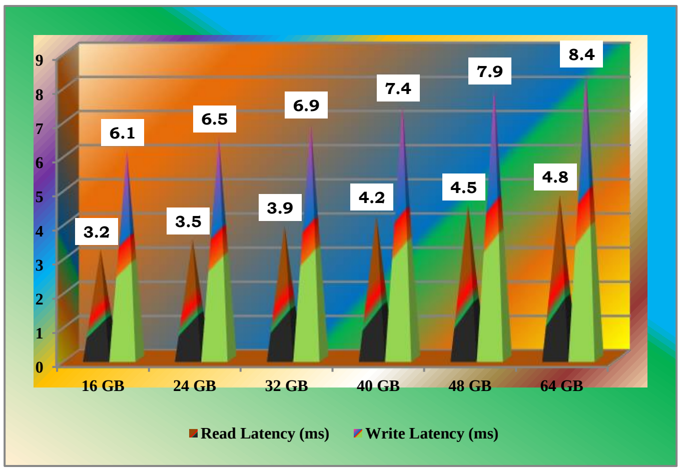
Graph 8: Memory usage and write amplification: Ledger DB 4

Graph 8 shows the memory usage and write amplification for the ETCD data store having the Ledger DB implementation.

Store Size	Read Latency (ms)	Write Latency (ms)	Write Amplification	Memory Usage (MB)
16 GB	3.2	6.1	7.8	140
24 GB	3.5	6.5	8.3	160
32 GB	3.9	6.9	8.8	180
40 GB	4.2	7.4	9.3	200
48 GB	4.5	7.9	9.8	220
64 GB	4.8	8.4	10.3	240

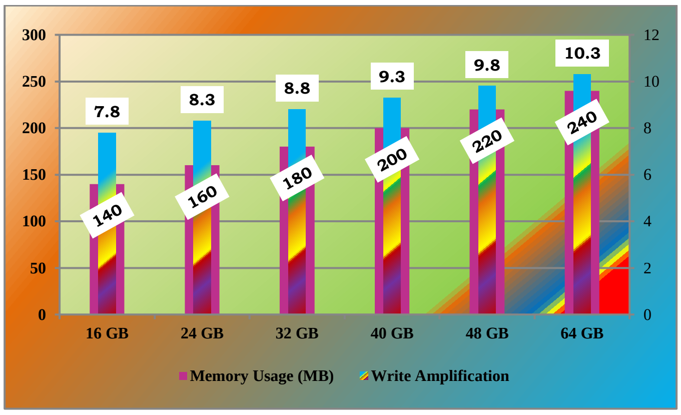
Table 5: Read and write latency: Ledger DB- 5

As shown in the Table 5, We have collected for different sizes of the ETCD data store sizes like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB. We have collected the metrics for read latency, write latency , write amplification and memory usage. The values are getting increased while the size of the ETCD data store is growing up.



Graph 9 : Read and write latency: Ledger DB- 5

Graph 9 shows the read latency and write latency from the fifth sample of the Ledger DB implementation of the key value store (ETCD).



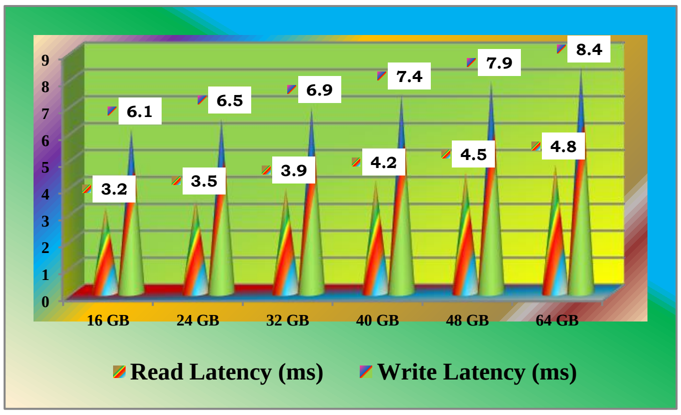
Graph 10: Memory usage and write amplification:
Ledger DB 5

Graph 10 shows the read latency and write latency from the fifth sample of the Ledger DB implementation of the key value store (ETCD).

Store Size	Read Latency (ms)	Write Latency (ms)	Write Amplification	Memory Usage (MB)
16 GB	3.2	6.1	7.8	140
24 GB	3.5	6.5	8.3	160
32 GB	3.9	6.9	8.8	180
40 GB	4.2	7.4	9.3	200
48 GB	4.5	7.9	9.8	220
64 GB	4.8	8.4	10.3	240

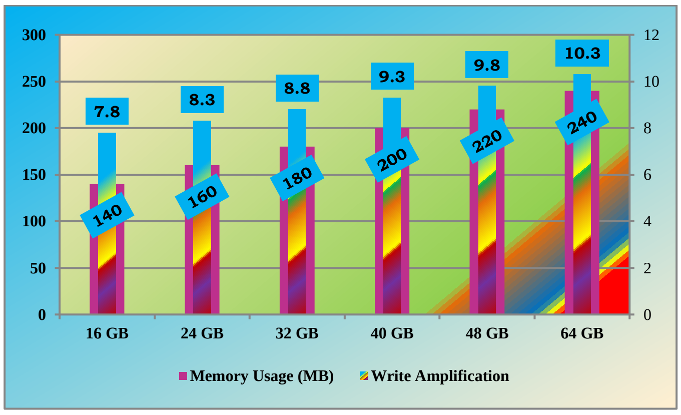
Table 6: Read and write latency: Ledger DB - 6

As shown in the Table 6, We have collected for different sizes of the ETCD data store sizes like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB. We have collected the metrics for read latency, write latency , write amplification and memory usage. The values are getting increased while the size of the ETCD data store is growing up.



Graph 11 : Read and write latency: Ledger DB - 6

Graph 11 shows the read latency and write latency from the sixth sample of the Ledger DB implementation of the key value store (ETCD).



Graph 12: Memory usage and write amplification:

Ledger DB 6

Graph 12 shows the read latency and write latency from the sixth sample of the Ledger DB implementation of the key value store (ETCD).

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 [23] 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 [24] 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 [25]. 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[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 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.

```
package main

import (
	"fmt"
	"log"
	"time"
	"github.com/dgraph-io/badger/v3"
	"runtime"
)

func collectBadgerMetrics(db *badger.DB, key string, value []byte) {
	start := time.Now()
	err := db.Update(func(txn *badger.Txn) error {
		return txn.Set([]byte(key), value)
	})
	if err != nil {
		log.Fatalf("Write Error: %v", err)
	}
	writeLatency := time.Since(start)

	start = time.Now()
	err = db.View(func(txn *badger.Txn) error {
		_, err := txn.Get([]byte(key))
		return err
	})
	if err != nil {
		log.Fatalf("Read Error: %v", err)
	}
	readLatency := time.Since(start)

	var memStats runtime.MemStats
	runtime.ReadMemStats(&memStats)
	fmt.Printf("BadgerDB Metrics:\n")
	fmt.Printf("Write Latency: %v\n", writeLatency)
	fmt.Printf("Read Latency: %v\n", readLatency)
	fmt.Printf("Memory Usage: %v MB\n", memStats.Alloc/1024/1024)
	fmt.Printf("Write Amplification: Approximate factor depends on LSM levels\n")
}
func main() {
	// Open BadgerDB
	opts := badger.DefaultOptions("./badgerdb")
	db, err := badger.Open(opts)
	if err != nil {
		log.Fatalf("Error opening BadgerDB: %v", err)
	}
	defer db.Close()
	collectBadgerMetrics(db, "testKey", []byte("testValue"))
}
```

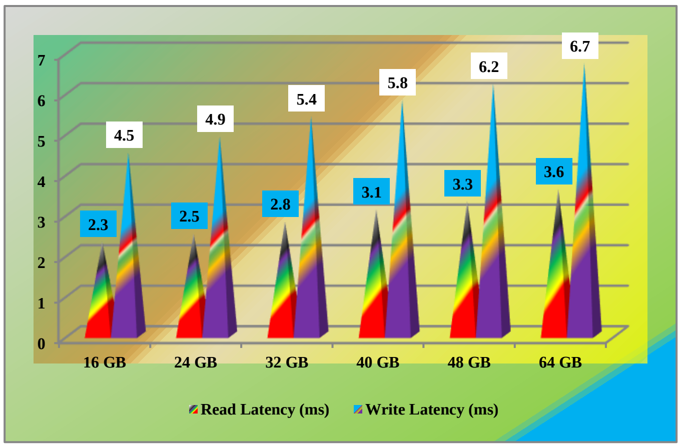
The test code collects performance metrics for the BadgerDB implementation of ETCD [27] ,focusing on insertion time, deletion time, search time, CPU usage, space complexity, and time complexity.

Write Latency Measures the time taken to write a key-value pair. Read Latency Measures the time taken to read a key-value pair. Memory Usage Uses Go's runtime.MemStats to calculate memory usage. Write Amplification BadgerDB Write amplification is influenced by compactions in the LSM structure.

Store Size	Read Latency (ms)	Write Latency (ms)	Write Amplification	Memory Usage (MB)
16 GB	2.3	4.5	5.5	100
24 GB	2.5	4.9	6	120
32 GB	2.8	5.4	6.5	140
40 GB	3.1	5.8	7	160
48 GB	3.3	6.2	7.5	180
64 GB	3.6	6.7	8	200

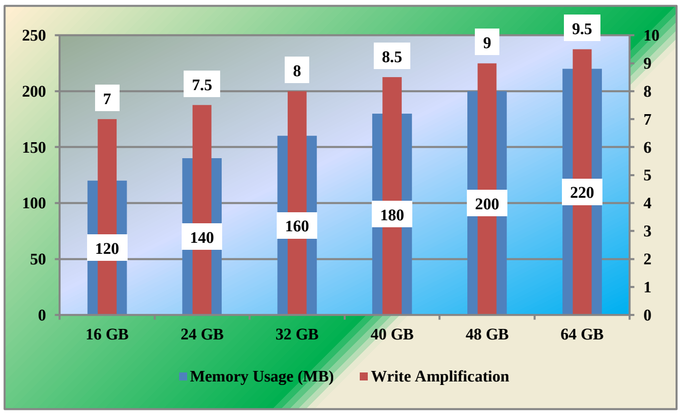
Table 7: Read and write latency: Badger DB – 1

As shown in the Table 7, We have collected for different sizes of the ETCD data store sizes like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB for ETCD Badger DB implementation. We have collected the metrics for read latency, write latency , write amplification and memory usage. The values are getting increased while the size of the ETCD data store is growing up.



Graph 13: Read and write latency: Badger DB – 1

Graph 13 shows the read latency and write latency from the first sample of the Badger DB implementation of the key value store (ETCD).



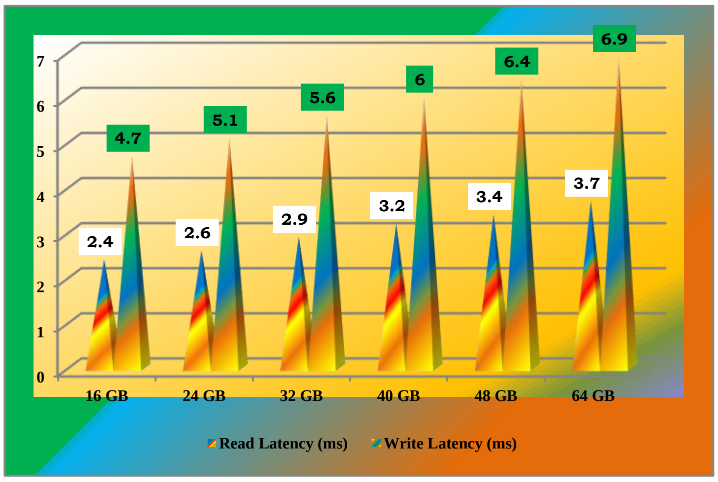
Graph 14: Memory usage and write amplification: Badger DB -1

Graph 14 shows the memory usage and write amplification for the ETCD data store having the Badger DB implementation. It shows the two scale Y axis plot since we are having two different ranges of data i.e, memory usage from 0 to 250 and write amplification from 0 to 10. Write amplification is a phenomenon where the actual amount of data written to a storage device exceeds the logical amount of data intended to be written, caused by factors such as garbage collection, wear leveling, and metadata management, ultimately leading to decreased storage efficiency and reliability.

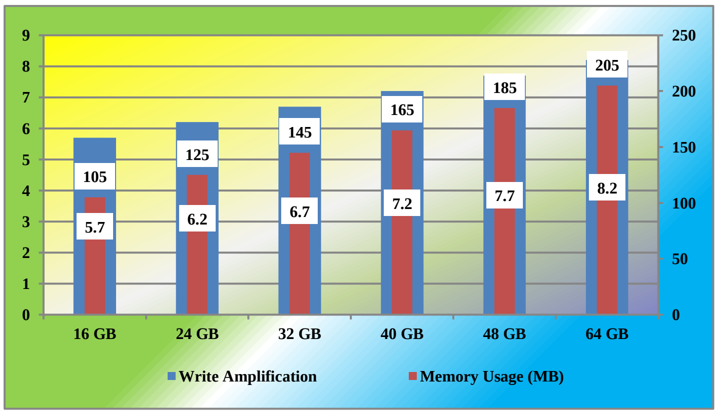
Store Size	Read Latency (ms)	Write Latency (ms)	Write Amplification	Memory Usage (MB)
16 GB	2.4	4.7	5.7	105
24 GB	2.6	5.1	6.2	125
32 GB	2.9	5.6	6.7	145
40 GB	3.2	6	7.2	165
48 GB	3.4	6.4	7.7	185
64 GB	3.7	6.9	8.2	205

Table 8: Read and write latency: Badger DB – 2

As shown in the Table 8, We have collected for different sizes of the ETCD data store sizes like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB for ETCD Badger DB implementation. We have collected the metrics for read latency, write latency , write amplification and memory usage. The values are getting increased while the size of the ETCD data store is growing up.



Graph 15: : Read and write latency: Badger DB – 2



Graph 16: Memory usage and write amplification: Badger DB -2

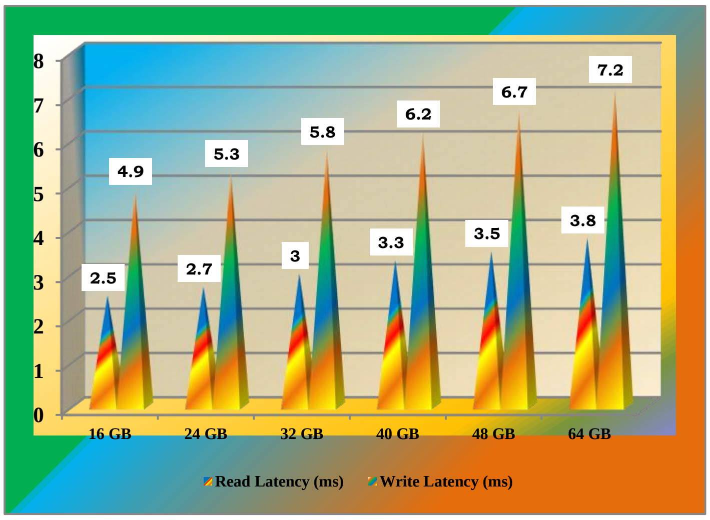
Graph 14 shows the memory usage and write amplification for the ETCD data store having the Badger

DB implementation. It shows the two scale Y axis plot since we are having two different ranges of data i.e, memory usage from 0 to 250 and write amplification from 0 to 10. Write amplification is a phenomenon where the actual amount of data written to a storage device exceeds the logical amount of data intended to be written, caused by factors such as garbage collection, wear leveling, and metadata management, ultimately leading to decreased storage efficiency and reliability.

Store Size	Read Latency (ms)	Write Latency (ms)	Write Amplification	Memory Usage (MB)
16 GB	2.5	4.9	5.9	110
24 GB	2.7	5.3	6.4	130
32 GB	3	5.8	6.9	150
40 GB	3.3	6.2	7.4	170
48 GB	3.5	6.7	7.9	190
64 GB	3.8	7.2	8.4	210

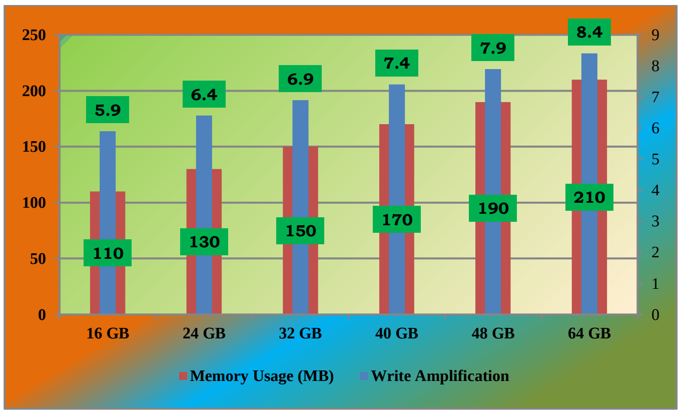
Table 9 : Read and write latency: Badger DB – 3

As shown in the Table 9, We have collected for different sizes of the ETCD data store sizes like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB for ETCD Badger DB implementation. We have collected the metrics for read latency, write latency , write amplification and memory usage. The values are getting increased while the size of the ETCD data store is growing up.



Graph 17: Read and write latency: 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 17 shows the collection of operations metrics for 16GB, 24GB , 32GB , 40GB , 48GB and 64GB ETCD store size.

•


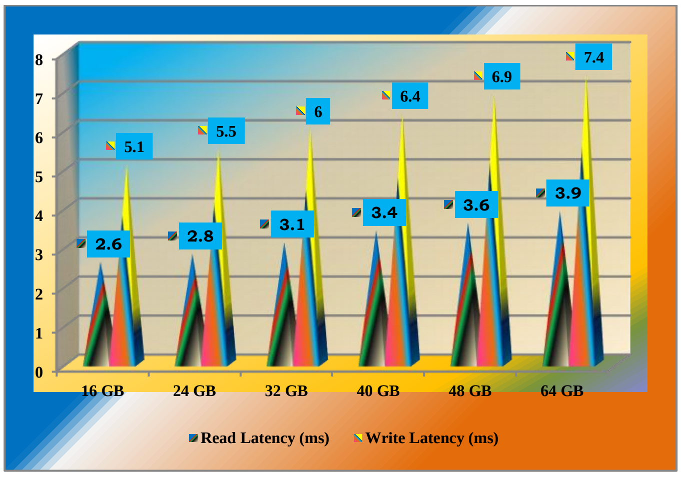
Graph 18: Memory usage and write amplification: Badger DB -3

Graph 18 shows the memory usage and write amplification for the ETCD data store having the Badger DB implementation. It shows the two scale Y axis plot since we are having two different ranges of data i.e, memory usage from 0 to 250 and write amplification from 0 to 10. Write amplification is a phenomenon where the actual amount of data written to a storage device exceeds the logical amount of data intended to be written, caused by factors such as garbage collection, wear leveling, and metadata management, ultimately leading to decreased storage efficiency and reliability.

Store Size	Read Latency (ms)	Write Latency (ms)	Write Amplification	Memory Usage (MB)
16 GB	2.6	5.1	6.1	115
24 GB	2.8	5.5	6.6	135
32 GB	3.1	6	7.1	155
40 GB	3.4	6.4	7.6	175
48 GB	3.6	6.9	8.1	195
64 GB	3.9	7.4	8.6	215

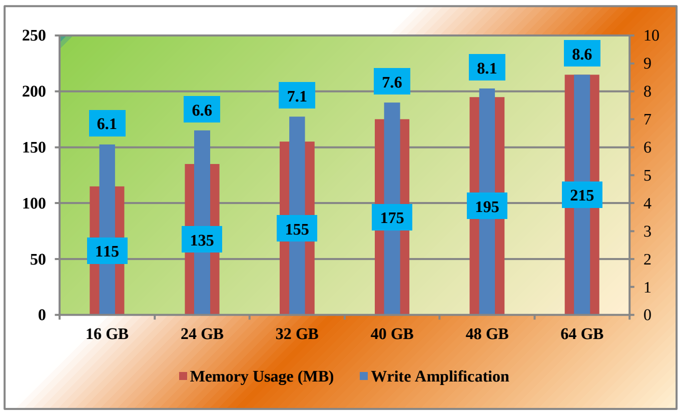
Table 10: Read and write latency: Badger DB -4

As shown in the Table 10, We have collected for different sizes of the ETCD data store sizes like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB for ETCD Badger DB implementation. We have collected the metrics for read latency, write latency , write amplification and memory usage. The values are getting increased while the size of the ETCD data store is growing up.



Graph 19: Read and write latency: Badger DB – 4

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 19 shows the collection of operations metrics for 16GB, 24GB , 32GB , 40GB , 48GB and 64GB ETCD store size.



Graph 20: Memory usage and write amplification:

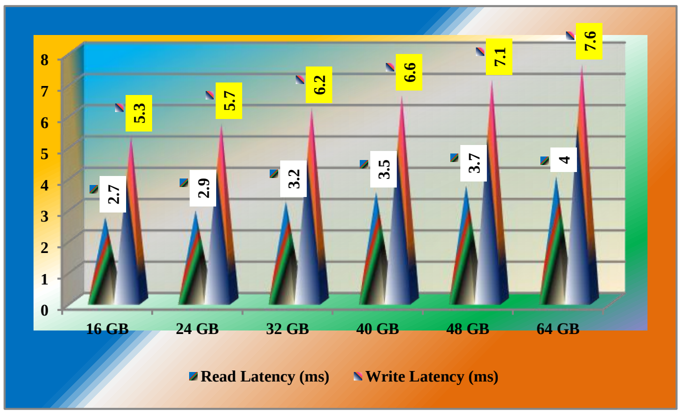
Badger DB -4

Graph 20 shows the memory usage and write amplification for the ETCD data store having the Badger DB implementation. It shows the two scale Y axis plot since we are having two different ranges of data i.e, memory usage from 0 to 250 and write amplification from 0 to 10.

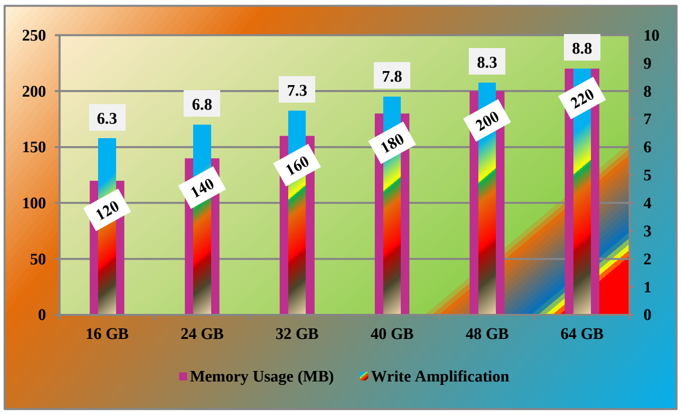
Store Size	Read Latency (ms)	Write Latency (ms)	Write Amplification	Memory Usage (MB)
16 GB	2.7	5.3	6.3	120
24 GB	2.9	5.7	6.8	140
32 GB	3.2	6.2	7.3	160
40 GB	3.5	6.6	7.8	180
48 GB	3.7	7.1	8.3	200
64 GB	4	7.6	8.8	220

Table 11: Read and write latency: Badger DB – 5

Fifth sample metrics have been updated at Table 11.



Graph 21: Read and write latency: Badger DB – 5



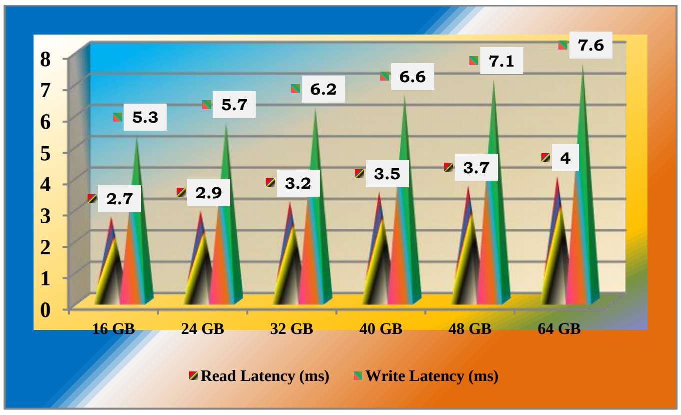
Graph 22: Memory usage and write amplification: Badger DB -5

Graph 22 shows the memory usage and write amplification for the ETCD data store having the Badger DB implementation. It shows the two scale Y axis plot since we are having two different ranges of data i.e, memory usage from 0 to 250 and write amplification from 0 to 10.

Store Size	Read Latency (ms)	Write Latency (ms)	Write Amplification	Memory Usage (MB)
16 GB	2.7	5.3	6.3	120
24 GB	2.9	5.7	6.8	140
32 GB	3.2	6.2	7.3	160
40 GB	3.5	6.6	7.8	180
48 GB	3.7	7.1	8.3	200
64 GB	4	7.6	8.8	220

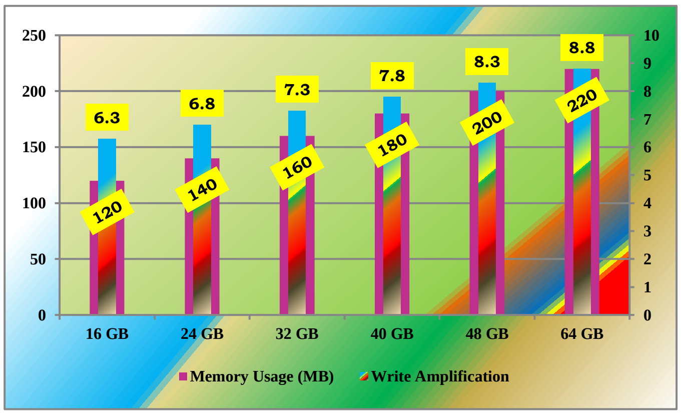
Table 12: Read and write latency: Badger DB -6

Sixth sample metrics have been updated at Table 12.



Graph 23: Read and write latency: Badger DB -6

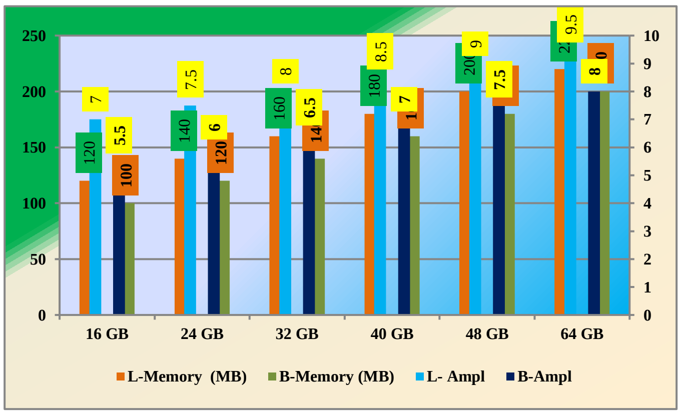
Graph 23 shows the Badger DB implementation parameters for ETCD like Read latency and write latency.



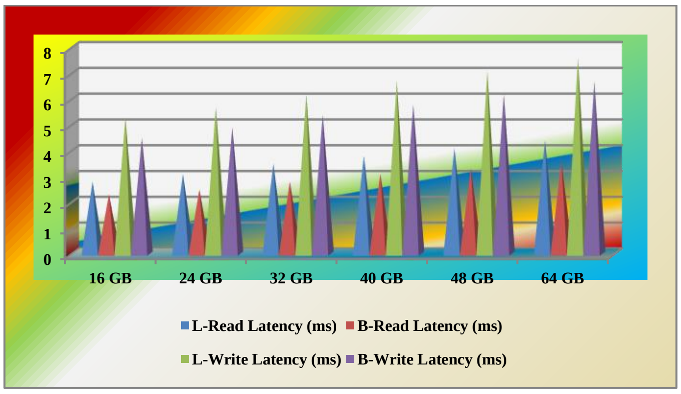
Graph 24: Memory usage and write amplification:

Badger DB -6

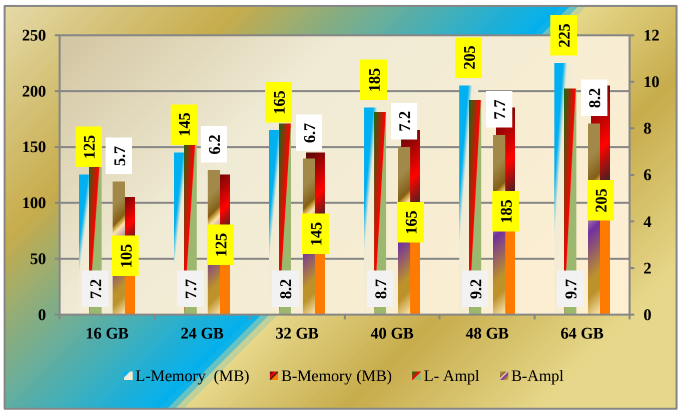
Graph 24 shows the memory usage and write amplification for the ETCD data store having the Badger DB implementation. It shows the two scale Y axis plot since we are having two different ranges of data i.e, memory usage from 0 to 250 and write amplification from 0 to 10.



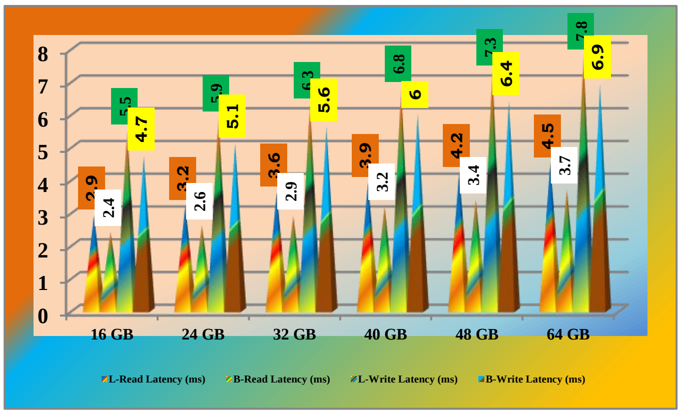
Graph 25: Amplification & Memory Usage Comparison -1.1



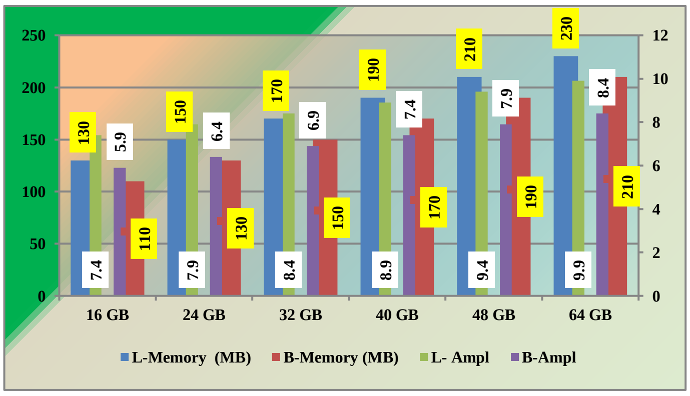
Graph 26: Latency Comparison -2.2



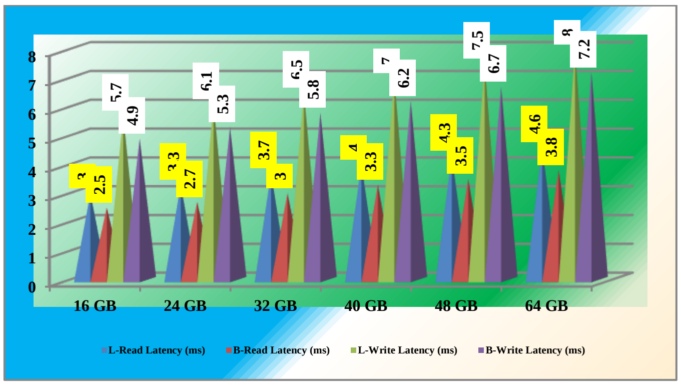
Graph 27: Amplification & Memory Usage Comparison -2.1



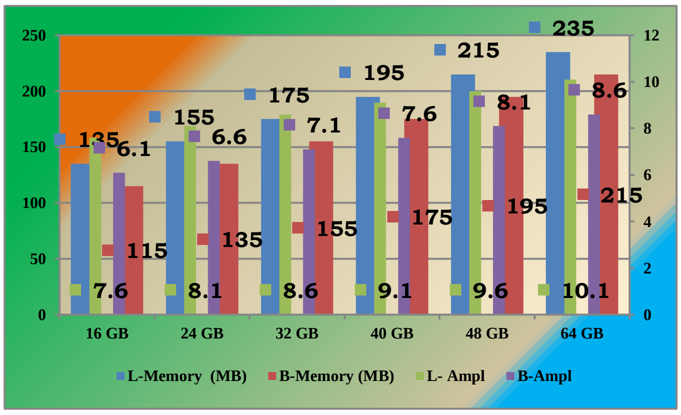
Graph 28: Latency Comparison -2.2



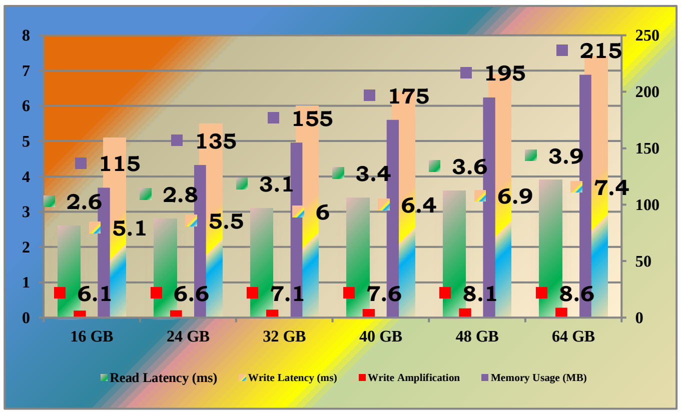
Graph 29: Amplification & Memory Usage Comparison -3.1



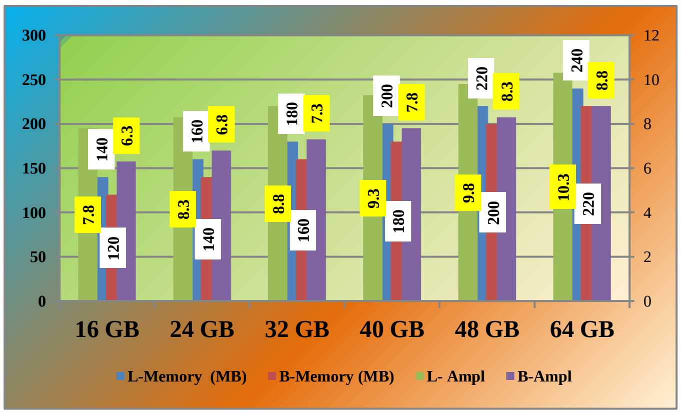
Graph 30: Latency Comparison -3.2



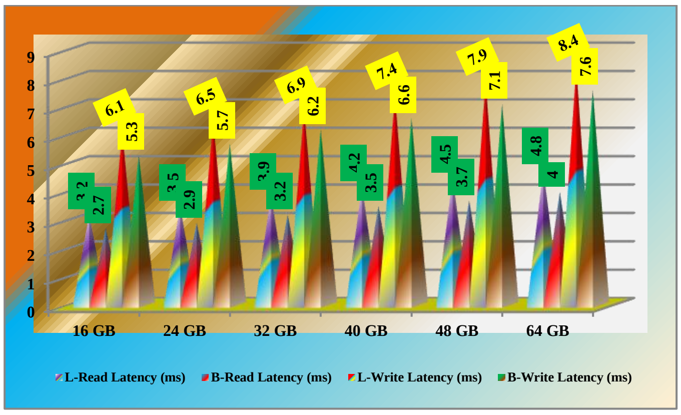
Graph 31: Amplification & Memory Usage Comparison -4.1



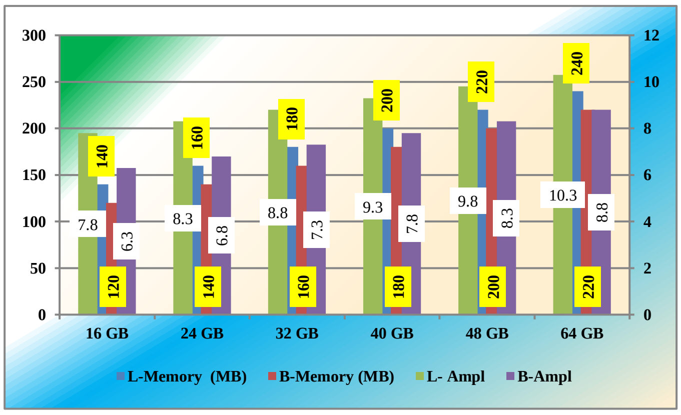
Graph 32: Latency Comparison -4.2



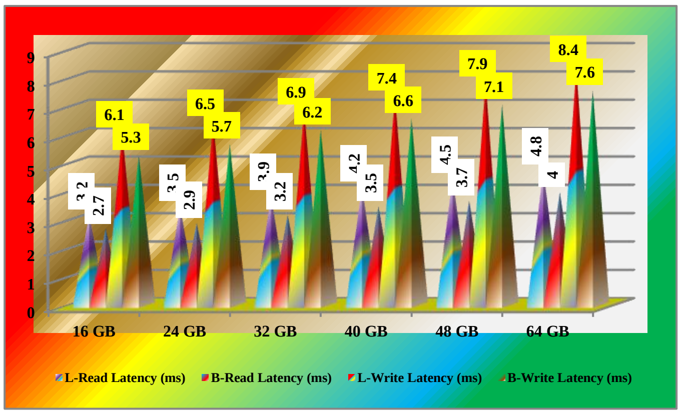
Graph 33: Amplification & Memory Usage Comparison -5.1



Graph 34: Latency Comparison -5.2



Graph 35: Amplification & Memory Usage Comparison -6.1



Graph 36: Latency Comparison -6.2

Graph 25, 27, 29, 31, 33 and 35 shows the amplification and memory usage comparison for six samples , Graph 26, 28, 30, 32 , 34 and 36shows the read latency and write latency comparison for six samples which we have collected based on the existing method and proposal method. According to the analysis of metrics which we have collected we can conclude that amplification , read and write latencies are going down while changing from larger DB to Badger DB implementation of ETCD key value store.

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. As per the analysis carried out so far in this research work , amplification is going down while using Badger DB implementation in place of Ledger DB implementation for ETCD key value store. Along with our observations carries that read and write latencies are also going down but the memory usage is going high.

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. As per the analysis carried out so far in this research work , amplification is going down while using Badger DB implementation in place of Ledger DB implementation for ETCD key value store. Along with our observations carries that read and write latencies are also going down but the memory usage is going high..

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] Kubernetes Container Orchestration as a Framework for Flexible and Effective Scientific Data Analysis, IEEE Xplore, 13 February 2020.
[3] Networking Analysis and Performance Comparison of Kubernetes CNI Plugins, 28 October 2020, pp 99–109, Ritik Kumar & Munesh Chandra Trivedi.
[4] "etcd: A Highly-Available, Distributed Key-Value Store" by Brandon Philips et al. (2014), Proceedings of the 2014 ACM SIGOPS Symposium on Cloud Computing.
[5] "Optimizing Kubernetes for Low-Latency Applications" by IBM (2020).
[6] "Performance Analysis of Kubernetes Clusters" by Microsoft (2018).
[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] Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned, Leila Abdollahi Vayghan Montreal, Mohamed Aymen Saied; Maria Toeroe; Ferhat Khendek, IEEE XPlore.
[11] High Availability Storage Server with Kubernetes, Ali Akbar Khatami; Yudha Purwanto; Muhammad Faris Ruriawan, 2020, IEEE Xplore.
[12] Improving Application availability with Pod Readiness Gates https://orielly.ly/h_WiG
[13] Kuberenets in action by Marko Liksa , 2018.
[14] 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.
[15] "Kubernetes Network Policies" by Calico (2019).
[16] Kubernetes Best Practices , Burns, Villaibha, Strebel , Evenson.

[17] Kubernetes Best Practices: Resource Requests and limits https://orielly.ly/8bKD5
[18] "Kubernetes Network Security" by Cisco (2018).
[19] Core Kubernetes , Jay Vyas , Chris Love.
[20] Configure Default Memory Requests and Limits for a Namespace https://orielly.ly/ozlUi1
[21] Kubernetes Persistent Storage by Google (2018).
[22] Kubernetes Scalability and Performance" by Red Hat (2019).
[23] Kubernetes Storage Performance by Red Hat (2019).
[24] Learning Core DNS, Belamanic, Liu.
[25] "Secure Kubernetes Deployment" by Palo Alto Networks (2019)".
[26] Modelling performance & resource management in kubernetes by Víctor Medel, Omer F. Rana, José Ángel Bañares, Unai Arronategui.
[27] The log-structured merge-tree (LEDGER DB-tree),June 1996, Patrick O’Neil, Edward Cheng, Dieter Gawlick & Elizabeth O’Neil.