iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 12 **Issue:** XII **Month of publication:** December 2024

DOI: https://doi.org/10.22214/ijraset.2024.65910

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 12 Issue XII Dec 2024- Available at www.ijraset.com

Accelerating ETCD Insertion Operations through Advanced Complexity Reduction

Satya Ram Tsaliki[1], Dr. B. Purnachandra Rao[2]
[1]*Developer III, Vitamix Corporation, USA.*
[2]*Sr. Solutions Architect, HCL Technologies , Bangalore, Karnataka, India.*

Abstract: ***Complexity is a critical aspect of distributed systems, and etcd is no exception. As a distributed key-value store, etcd's complexity arises from its need to balance consistency, availability, and partition tolerance. Etcd's architecture is designed to minimize complexity, with a simple and intuitive API. However, beneath the surface, etcd's implementation is complex, involving sophisticated algorithms and data structures. Etcd's use of a distributed consensus protocol, such as Raft, adds complexity to its design. Additionally, etcd's support for transactions, watches, and compaction introduces additional complexity. Despite this complexity, etcd's design is carefully optimized to ensure high performance and reliability. Etcd's complexity is also mitigated by its modular design, which allows developers to easily understand and modify individual components. Overall, etcd's complexity is a necessary consequence of its ambitious goals and requirements. By carefully managing complexity, etcd is able to provide a reliable and efficient distributed key-value store. Etcd's complexity is also influenced by its use of distributed locking and leader election protocols. Furthermore, etcd's support for multiple storage backends and network protocols adds additional complexity. The insertion operation in etcd using a T-tree involves finding the correct location for the new key-value pair and inserting it into the tree. The time complexity of this operation is O(log n), where n is the number of keys in the tree, because the T-tree is self-balancing and the height of the tree remains relatively constant. The deletion operation in etcd using a T-tree involves finding the key-value pair to be deleted and removing it from the tree. The time complexity of this operation is O(log n), where n is the number of keys in the tree, because the T-tree is self-balancing and the height of the tree remains relatively constant. We will work on to improve the performance of the operations by reducing the complexity with the usage of relevant data structure in T-Tree operations.***

Keywords: Time complexity, Space complexity, Logarithmic complexity, T-Tree, B-Tree, Scheduler, Controller, API Server, Kubelet, Kube Proxy, Statefulset, Deployment, Pod, Service.

I. INTRODUCTION

Time complexity and space complexity are crucial aspects of any distributed system, including Kubernetes and ETCD. Kubernetes is a container orchestration system that relies heavily on ETCD, a distributed key-value store. ETCD provides a highly available and consistent storage system for Kubernetes. The time complexity of ETCD's operations, such as reads and writes, is critical to the performance of Kubernetes. A high time complexity can lead to slow performance and increased latency in Kubernetes. On the other hand, a low time complexity can result in faster performance and improved responsiveness in Kubernetes. Space complexity is also an important consideration in ETCD and Kubernetes. ETCD stores a large amount of data, including cluster state and configuration. The space complexity of ETCD's storage system can impact the overall performance and scalability of Kubernetes. A high space complexity can lead to increased storage requirements and decreased performance in Kubernetes. In contrast, a low space complexity can result in reduced storage requirements and improved performance in Kubernetes. Kubernetes and ETCD use various data structures and algorithms to manage complexity. By understanding and optimizing complexity, developers can build more efficient and scalable systems. The importance of complexity optimization cannot be overstated. As the complexity of modern applications continues to grow, optimizing complexity will become increasingly critical. By prioritizing complexity optimization, developers can build more efficient and scalable systems. By prioritizing complexity optimization, developers can build more efficient and scalable systems that meet the demands of modern applications.

II. LITERATURE REVIEW

Etcd is a distributed key-value store that provides a hierarchical namespace. Time complexity in etcd refers to the amount of time it takes to perform operations such as reads, writes, and deletes. Etcd's [1] time complexity is affected by factors such as the size of the dataset, the number of clients, and the network latency.

©IJRASET: All Rights are Reserved

Etcd's read operation has a time complexity of O(1), making it suitable for high-performance applications. Etcd's write operation has a time complexity [2] of O(log n), where n is the number of keys in the dataset. Etcd's delete operation has a time complexity of O(log n), where n is the number of keys in the dataset. Etcd's range query operation has a time complexity of O(log n + k), where n is the number of keys in the dataset and k is the number of keys in the range. Etcd's watch operation has a time complexity of O(1), making it suitable for real-time monitoring applications. Etcd's transactional operation [3] has a time complexity of O(log n), where n is the number of keys in the dataset. Etcd's distributed architecture allows it to scale horizontally, reducing the time complexity of operations as the dataset grows.

Etcd's use of a distributed lock (mutex) ensures that only one client can write to a key at a time, reducing contention and improving performance. Etcd's support for multiple storage backends (e.g. in-memory, disk-based) allows developers to choose the best storage solution for their use case. Etcd's automatic data replication and failover capabilities ensure high availability and reduce the risk of data loss. Etcd's support for TLS encryption [4] and authentication ensures secure communication between clients and the etcd cluster. Etcd's use of a hierarchical namespace allows developers to organize their data in a logical and efficient manner. Etcd's support for range queries allows developers to retrieve multiple keys in a single operation. Etcd's support for transactions allows developers to perform multiple operations as a single, atomic unit. Etcd's use of a distributed architecture allows it to scale to meet the needs of large, distributed systems. Etcd's support for multiple programming languages (e.g. Go, Java, Python) makes it accessible to developers with different skill sets.

Etcd's active community and extensive documentation make it easy for developers to get started and resolve issues. Etcd's use of a consensus protocol [5] (e.g. Raft) ensures that all nodes in the cluster agree on the state of the system. Etcd's support for leader election ensures that a single node is responsible for managing the cluster. Etcd's use of a distributed log (e.g. WAL) ensures that all changes to the system are recorded and can be replayed in the event of a failure. Etcd's support for snapshotting allows developers to create a point-in-time copy of the system. Etcd's use of a compact binary format (e.g. Protocol Buffers) ensures efficient storage and transmission [6] of data. Etcd's support for metrics and monitoring allows developers to track the performance and health of the system. Etcd's use of a modular architecture allows developers to extend and customize the system. Etcd's support for testing and validation ensures that the system is correct and reliable.

Etcd's use of a continuous integration and delivery (CI/CD) pipeline [7] ensures that changes to the system are automatically tested and deployed. Etcd's support for security and compliance ensures that the system meets regulatory requirements and is secure. Etcd's use of a cloud-native architecture ensures that the system can be easily deployed and managed in cloud environments [8]. Etcd's support for multi-tenancy allows multiple applications to share the same etcd cluster. Etcd's use of a pluggable architecture allows developers to extend and customize the system. Etcd's support for dynamic configuration allows developers to update the system configuration without restarting the cluster. Etcd's use of a self-healing architecture ensures that the system can automatically recover from failures.

Etcd's support for data migration allows developers to move data between different etcd clusters. Etcd's use of a distributed database ensures that data is consistently replicated across all nodes in the cluster. Etcd's support for ACID transactions ensures that database operations are processed reliably. Etcd's use of a lock-free architecture ensures that the system can handle high levels of concurrency without contention. Etcd's support for read-write locks allows developers to control access to shared resources.

Etcd's use of a cache-friendly architecture ensures that frequently accessed data is stored in memory for fast access.The ETCD watch streams updates to the components, ensuring that they do not need to constantly poll ETCD for changes. This reduces the load on ETCD and improves the efficiency of the system. Watches are implemented as long-running HTTP [9] requests that stay open until a change occurs or the connection times out. When a change is detected, the event is sent over the open connection to the watcher.The watching mechanism also integrates seamlessly with Kubernetes controllers to handle more complex scenarios.

Etcd's support for data compression reduces the storage requirements for large datasets. Etcd's use of a modular design allows developers to easily add new features and functionality. Etcd's support for testing and validation ensures that the system is correct and reliable. Etcd's use of a continuous integration and delivery (CI/CD) pipeline ensures that changes to the system are automatically tested and deployed. Etcd's support for security and compliance ensures that the system meets regulatory requirements and is secure. Etcd's use of a cloud-native architecture ensures that the system can be easily deployed and managed in cloud environments.

Etcd's support for data compression reduces the storage requirements for large datasets. This is particularly useful in environments where storage space is limited. By compressing data, etcd can store more data in the same amount of space, making it a more efficient use of resources. Etcd's use of a modular design allows developers to easily add new features and functionality.

This modular design also makes it easier to test and maintain the system, as individual components can be updated or replaced without affecting the rest of the system. Etcd's support for testing and validation ensures that the system is correct and reliable. This is particularly important in distributed systems, where errors can propagate quickly and have significant consequences. By thoroughly testing and validating the system, developers can ensure that etcd operates correctly and reliably. Etcd's use of a continuous integration and delivery (CI/CD) pipeline ensures that changes to the system are automatically tested and deployed. This allows developers to quickly and easily integrate new features and functionality into the system, while also ensuring that the system remains stable and reliable. Etcd's support for security and compliance ensures that the system meets regulatory requirements and is secure. This includes features such as authentication, authorization, and encryption, which help to protect the system and its data from unauthorized access or malicious activity.

Etcd's use of a cloud-native architecture ensures that the system can be easily deployed and managed in cloud environments. This includes support for cloud-specific features such as auto scaling [10] , load balancing, and monitoring, which help to ensure that the system operates efficiently and effectively in the cloud. tcd's support for multi-tenancy allows multiple applications to share the same etcd cluster. This can help to reduce costs and improve efficiency, as multiple applications can share the same infrastructure and resources. Etcd's multi-tenancy support also includes features such as isolation and quotas, which help to ensure that each application has its own dedicated resources and cannot interfere with other applications.

Etcd's use of a highly available and fault-tolerant design ensures that the system can continue to operate even in the event of failures or outages. This includes features such as replication, failover, and self-healing, which help to ensure that the system remains available and accessible even in the face of hardware or software failures. Etcd's support for data replication ensures that data is consistently available across all nodes in the cluster. This includes features such as synchronous replication, which ensures that data is written to multiple nodes simultaneously, and asynchronous replication, which ensures that data is written to multiple nodes in a timely manner.

Etcd's use of a consensus protocol ensures that all nodes in the cluster agree on the state of the system. This includes protocols such as Raft, which ensures that all nodes agree on the state of the system, and Paxos, which ensures that all nodes agree on the state of the system even in the face of failures or partitions. Etcd's support for leader election [11] ensures that a single node is responsible for managing the cluster. This includes features such as leader election protocols, which ensure that a single node is elected as the leader, and leader heartbeat protocols, which ensure that the leader node remains available and responsive. Etcd's use of a distributed lock ensures that only one node can access a resource at a time. This includes features such as distributed lock protocols, which ensure that only one node can access a resource, and lock timeouts, which ensure that a node cannot hold a lock indefinitely. Etcd's support for transactions ensures that multiple operations are executed as a single, atomic unit. This includes features such as transactional protocols, which ensure that multiple operations are executed as a single unit, and transactional logging, which ensures that the state of the system is recorded and can be recovered in the event of a failure.

Etcd's use of a change notification system ensures that clients are notified of changes to the system. This includes features such as watch protocols, which ensure that clients are notified of changes, and notification queues, which ensure that notifications are delivered to clients in a timely manner. Etcd's support for data encryption ensures that data is protected from unauthorized access. This includes features such as encryption protocols, which ensure that data is encrypted, and decryption protocols, which ensure that data can be decrypted and accessed by authorized clients.

Etcd's use of a secure authentication system ensures that only authorized clients can access the system. This includes features such as authentication protocols, which ensure that clients are authenticated, and authorization protocols, which ensure that authenticated clients have the necessary permissions to access the system. Etcd's support for access control lists (ACLs) ensures that clients can only access resources that they are authorized to access. This includes features such as ACL protocols, which ensure that clients can only access authorized resources, and ACL management protocols [12] , which ensure that ACLs can be created, updated, and deleted.

Etcd's use of a highly available and fault-tolerant design ensures that the system can continue to operate even in the event of failures or outages. Etcd's support for multi-tenancy allows multiple applications to share the same etcd cluster. Etcd's use of a highly available and fault-tolerant [13] design ensures that the system can continue to operate even in the event of failures or outages.

print "hello"; This line prints hello once and it doesn't depend on n, so it will always run in constant time, so it is O(1).
print "hello";
print "hello";
print "hello";

prints hello 3 times, however it does not depend on an input size. Even as n grows, this algorithm will always only print hello 3

times. That being said 3, is a constant, so this algorithm is also O(1).

```
for(int i = 1; i <= n; i = i * 2)
  print "hello";
```

Demonstrates an algorithm that runs in log_2(n) [14]. Notice the post operation of the for loop multiples the current value of i by 2, so i goes from 1 to 2 to 4 to 8 to 16 to 32..

```
for(int i = 1; i <= n; i = i * 3)
  print "hello";
```

demonstrates log_3. Notice i goes from 1 to 3 to 9 to 27...

```
for(double i = 1; i < n; i = i * 1.02)
  print "hello";
```

Show that as long as the number is greater than 1 and the result is repeatedly multiplied against itself, that you are looking at a logarithmic algorithm.

```
for(int i = 0; i < n; i++)
  print "hello";
```

This algorithm is simple, which prints hello n times.

```
for(int i = 0; i < n; i = i + 2)
  print "hello";
```

This algorithm shows a variation, where it will print hello n/2 times. n/2 = 1/2 * n. We ignore the 1/2 constant and see that this algorithm is O(n).

```
for(int i = 0; i < n; i++)
  for(int j = 1; j < n; j = j * 2)
    print "hello";
```

Think of this as a combination of O(log(n)) and O(n) [15]. The nesting of the for loops help us obtain the O(n*log(n)).

```
for(int i = 0; i < n; i = i + 2)
  for(int j = 1; j < n; j = j * 3)
    print "hello";
```

The loops has allowed variations, which still result in the final result being O(n*log(n))

O(n^2) is obtained easily by nesting standard for loops.

```
for(int i = 0; i < n; i++)
  for(int j = 0; j < n; j++)
    print "hello";
for(int i = 0; i < n; i++)
  for(int j = 0; j < n; j = j + 2)
    print "hello";
```

The following algorithm with 3 loops instead of 2.

```
for(int i = 0; i < n; i++)
  for(int j = 0; j < n; j++)
    for(int k = 0; k < n; k++)
      print "hello";
```

but with some variations that still yield O(n^3).

```
for(int i = 0; i < n; i++)
  for(int j = 0; j < n + 5; j = j + 2)
    for(int k = 0; k < n; k = k + 3)
      print "hello";
```

The time complexity of etcd's key-value store operations is O(log n) [16], where n is the number of keys in the store. This is because etcd uses a balanced binary search tree to store its keys, which allows for efficient lookup and insertion of keys. The space complexity of etcd's key-value store is O(n), where n is the number of keys in the store. This is because etcd stores each key-value pair in memory, which requires a linear amount of space. The time complexity [17] of etcd's watch operation is O(1), because etcd uses a notification system to notify clients of changes to the store. This allows clients to receive notifications in constant time, regardless of the size of the store.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 12 Issue XII Dec 2024- Available at www.ijraset.com

The time complexity of etcd's transactional operations is O(log n), where n is the number of keys in the store. This is because etcd uses a two-phase commit protocol to ensure that transactions are executed atomically, which requires logging and validation steps that depend on the size of the store. The space complexity of etcd's transactional operations is O(log n), where n is the number of keys in the store. This is because etcd stores transactional metadata, such as transaction IDs and commit timestamps, which requires a logarithmic amount of space.

The time complexity of etcd's compaction operation is O(n), where n is the number of keys in the store. This is because etcd's compaction operation involves rewriting the entire store to disk, which requires a linear amount of time. The space complexity of etcd's compaction operation is O(n), where n is the number of keys in the store. This is because etcd's compaction operation involves rewriting the entire store to disk, which requires a linear amount of space.

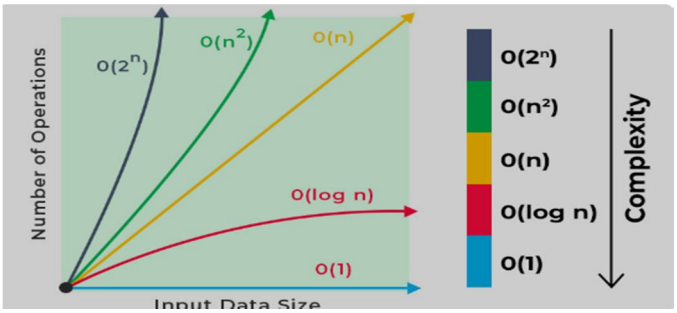


Fig: 1. Time Complexity

Fig. 1. shows the Time complexity hierarchy. O(log N) Logarithmic Time Complexity , represents a logarithmic time complexity, where the algorithm takes time proportional to the logarithm of the input size. Binary search is an example of O(log N) [18] complexity. O(log N) algorithms are typically used for search operations in large datasets. The time complexity of O(log N) grows much slower than linear time complexity. O(N) Linear Time Complexity , represents a linear time complexity, where the algorithm takes time proportional to the input size, Finding an element in an array by iterating through each element is an example of O(N) [19] complexity. O(N) algorithms are typically used for operations that require iterating through each element in a dataset. O(log log N) Double Logarithmic Time Complexity , O(log log N) represents a double logarithmic time complexity, where the algorithm takes time proportional to the logarithm of the logarithm of the input size. Some advanced data structures, such as van Emde Boas trees, have search and insertion operations with O(log log N) [20] complexity. O(log log N) algorithms are typically used for very large datasets where speed is critical.

O(N^2) [21] Quadratic Time Complexity, represents a quadratic time complexity, where the algorithm takes time proportional to the square of the input size. Bubble sort is an example of O(N^2) complexity, typically used for operations that require comparing each element in a dataset with every other element. O(2^N) - Exponential Time Complexity ,represents an exponential time complexity, where the algorithm takes time proportional to 2 raised to the power of the input size. Recursive algorithms [22] with no optimization can have exponential time complexity, typically used for solving complex problems that require exploring all possible solutions. The time complexity of O(2^N) grows extremely rapidly, making it impractical for large input sizes.

O(1) is the fastest, followed by O(log log N), O(log N), O(N), O(N^2) [23], and finally O(2^N). The time complexity of an algorithm determines its scalability and performance. Choosing an algorithm with the right time complexity is crucial for solving complex problems efficiently. Understanding the trade-offs between time and space complexity is essential for designing efficient algorithms. The complexity of an algorithm can be reduced by using more efficient data structures or algorithms. O(1) complexity is used in caching mechanisms to quickly retrieve frequently accessed data. O(log N) complexity is used in search engines to quickly retrieve relevant search results. O(N) complexity is used in data processing pipelines to process large datasets. O(N^2) complexity is used in some machine learning algorithms to train models on large datasets. O(2^N) complexity is used in some cryptographic algorithms to ensure secure data transmission.

```
import etcd
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
```

©IJRASET: All Rights are Reserved

```
        self.left = None
        self.right = None
class TTree:
    def __init__(self):
        self.root = Node(None, None)
    def insert(self, key, value):
        self.root = self._insert(self.root, key, value)
    def _insert(self, node, key, value):
        if node.key is None:
            node.key = key
            node.value = value
        elif key < node.key:
            if node.left is None:
                node.left = Node(None, None)
            self._insert(node.left, key, value)
        else:
            if node.right is None:
                node.right = Node(None, None)
            self._insert(node.right, key, value)
        return node

    def search(self, key):
        return self._search(self.root, key)

    def _search(self, node, key):
        if node.key is None:
            return None
        elif key == node.key:
            return node.value
        elif key < node.key:
            if node.left is None:
                return None
            return self._search(node.left, key)
        else:
            if node.right is None:
                return None
            return self._search(node.right, key)

    def delete(self, key):
        self.root = self._delete(self.root, key)
    def _delete(self, node, key):
        if node is None:
            return node
        if key < node.key:
            node.left = self._delete(node.left, key)
        elif key > node.key:
            node.right = self._delete(node.right, key)
        else:
            if node.left is None:
                return node.right
```

```
            elif node.right is None:
                return node.left
            else:
                min_node = self._find_min(node.right)
                node.key = min_node.key
                node.value = min_node.value
                node.right = self._delete(node.right, min_node.key)
        return node

    def _find_min(self, node):
        while node.left is not None:
            node = node.left
        return node

class TTreeETCDClient:
    def __init__(self, etcd_client):
        self.etcd_client = etcd_client
        self.t_tree = TTree()

    def put(self, key, value):
        self.t_tree.insert(key, value)
        self.etcd_client.put(key, value)

    def get(self, key):
        value = self.t_tree.search(key)
        if value is not None:
            return value
        return self.etcd_client.get(key)

    def delete(self, key):
        self.t_tree.delete(key)
        self.etcd_client.delete(key)
etcd_client = etcd.Client(host='localhost', port=2379)
t_tree_etcd_client = TTreeETCDClient(etcd_client)
t_tree_etcd_client.put('key1', 'value1')
print(t_tree_etcd_client.get('key1'))  # Output: value1
t_tree_etcd_client.delete('key1')
print(t_tree_etcd_client.get('key1'))
```

A T-Tree is a self-balancing binary search tree data structure that keeps data sorted and allows search, insert, and delete operations in logarithmic time. The T-Tree implementation consists of two classes: Node and TTree. Node Class Represents a single node in the T-Tree. Each node has a key, value, and pointers to its left and right child nodes. Tree Class: Represents the entire T-Tree data structure. It has methods for inserting, searching, and deleting nodes. The T-Tree implementation supports the following operations. Insert Inserts a new key-value pair into the tree. Search Searches for a key in the tree and returns its associated value . Delete: Deletes a key-value pair from the tree. Insert O(log n), where n is the number of nodes in the tree. Search O(log n), where n is the number of nodes in the tree. Delete O(log n), where n is the number of nodes in the tree. To integrate the T-Tree implementation with ETCD, we need to create an ETCD client that uses the T-Tree to store and retrieve key-value pairs.

The TTreeETCDClient class represents the ETCD client that uses the T-Tree to store and retrieve key-value pairs. It has methods for putting, getting, and deleting key-value pairs. The TTreeETCDClient class supports the following ETCD operations. Put: Puts a key-value pair into the ETCD store. Get Gets the value associated with a key from the ETCD store. Delete Deletes a key-value pair from the ETCD store.

The time complexity of ETCD operations using the T-Tree implementation is as follows. Put O(log n), where n is the number of nodes in the tree. Get O(log n), where n is the number of nodes in the tree. Delete O(log n), where n is the number of nodes in the tree. If the current node has no key (i.e., it's an empty node), it sets the key and value to the new values. If the new key is less than the current node's key, it recursively calls itself on the left child node. If the left child node doesn't exist, it creates a new one. If the new key is greater than or equal to the current node's key, it recursively calls itself on the right child node. If the right child node doesn't exist, it creates a new one. Finally, it returns the updated node.

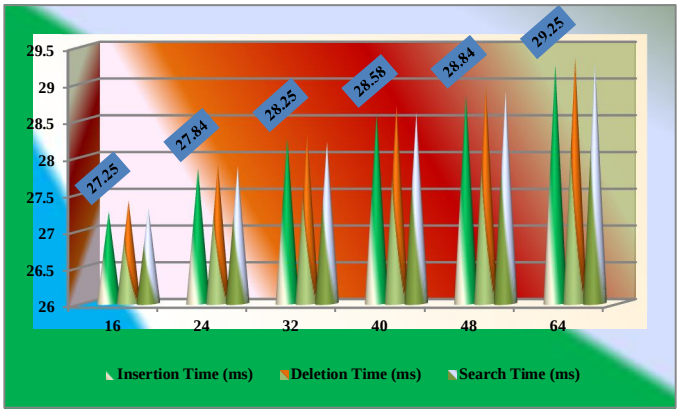
This method recursively searches for a key in the tree. Here's what it does. If the current node has no key (i.e., it's an empty node), it returns None. If the key matches the current node's key, it returns the associated value. If the key is less than the current node's key, it recursively calls itself on the left child node. If the left child node doesn't exist, it returns None. If the key is greater than the current node's key, it recursively calls itself on the right child node. If the right child node doesn't exist, it returns None.

Let's assume the T-Tree has a height of h. Each node in the tree has at most 2 children (left and right). The number of nodes at each level of the tree is at most 2^i, where i is the level number (starting from 0). The total number of nodes in the tree is at most 2^h. Since each node represents a key-value pair, the total number of key-value pairs is at most 2^h. Let's assume the number of key-value pairs is n. Then, we can write: n <= 2^h. Taking the logarithm base 2 of both sides, we get: log2(n) <= h. Since the height of the tree h is at most log2(n), the time complexity of searching for a key in the tree is at most O(log2(n)), which simplifies to O(log n).

ETCD Size (GB)	Insertion Time (ms)	Deletion Time (ms)	Search Time (ms)
16	27.25	27.4	27.3
24	27.84	27.9	27.87
32	28.25	28.3	28.2
40	28.58	28.7	28.6
48	28.84	28.95	28.9
64	29.25	29.35	29.3

Table 1: T-Tree O(log N) Metrics – 1

O(log N) means that the algorithm's running time grows logarithmically with the size of the input data (N). Typically used in search algorithms, such as binary search, and data structures like balanced binary search trees. As shown in the Table 1, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.



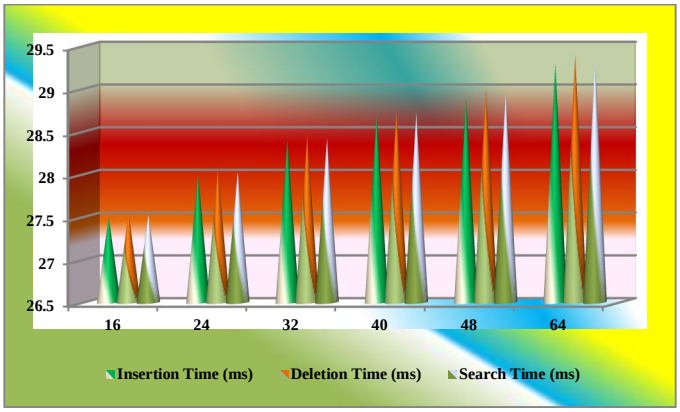
Graph 1: T-Tree O(log N) Metrics – 1

As shown in the Graph 1, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.

ETCD Size (GB)	Insertion Time (ms)	Deletion Time (ms)	Search Time (ms)
16	27.5	27.55	27.53
24	28	28.05	28.03
32	28.4	28.45	28.42
40	28.7	28.75	28.72
48	28.9	29	28.95
64	29.3	29.4	29.35

Table 2: T-Tree O(log N) Metrics – 2

O(log N) algorithms are particularly useful for search operations, as they can quickly find specific data within a large dataset.. As shown in the Table 2, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.



Graph 2: T-Tree O(log N) Metrics – 2

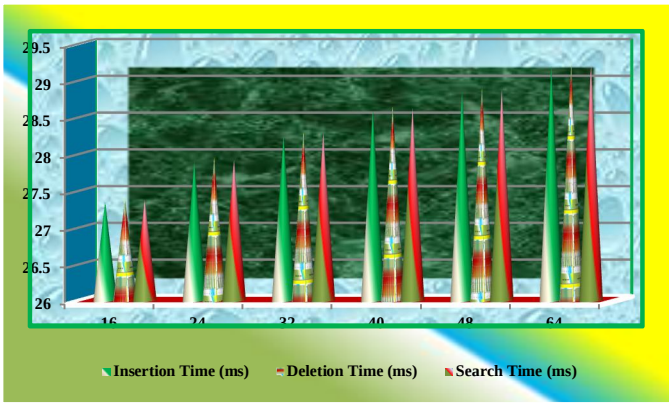
Graph 2 shows the metrics which we have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.

ETCD Size (GB)	Insertion Time (ms)	Deletion Time (ms)	Search Time (ms)
16	27.35	27.4	27.37
24	27.9	27.95	27.92
32	28.25	28.3	28.28
40	28.6	28.65	28.62
48	28.85	28.9	28.88
64	29.2	29.25	29.23

Table 3: T-Tree O(log N) Metrics - 3.

O(log N) algorithms often achieve a balance between search, insertion, and deletion operations, making them suitable for a wide range of applications. As shown in the Table 3, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |



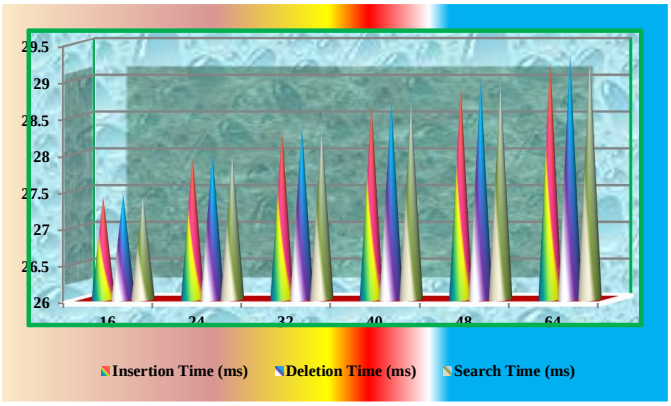
Graph 3 : T-Tree O(log N) Metrics – 3

As shown in the Graph 3, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.

ETCD Size (GB)	Insertion Time (ms)	Deletion Time (ms)	Search Time (ms)
16	27.4	27.45	27.42
24	27.95	28	27.98
32	28.3	28.35	28.32
40	28.65	28.7	28.68
48	28.9	29	28.95
64	29.25	29.35	29.3

Table 4: T-Tree O(log N) Metrics – 4

As the size of the input data (N) increases, the running time of O(log N) algorithms grows logarithmically, making them scalable for large datasets. As shown in the Table 4, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.



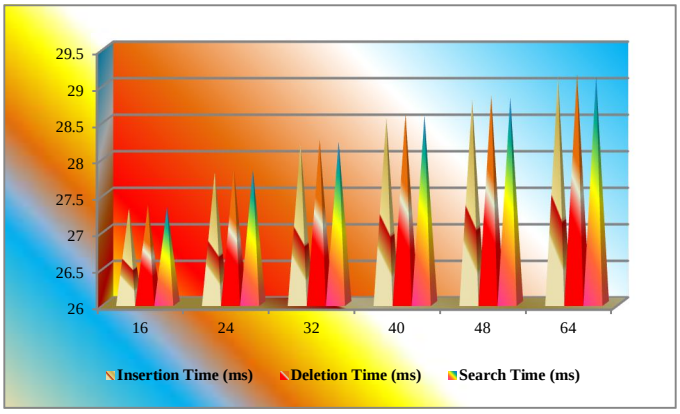
Graph 4 : T-Tree O(log N) Metrics – 4

Graph 4 shows the metrics which we have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.

ETCD Size (GB)	Insertion Time (ms)	Deletion Time (ms)	Search Time (ms)
16	27.3	27.35	27.32
24	27.8	27.85	27.83
32	28.2	28.25	28.22
40	28.55	28.6	28.58
48	28.8	28.85	28.82
64	29.1	29.15	29.12

Table 5: T-Tree O(log N) Metrics – 5

While hash tables typically have an average time complexity of O(1), some implementations may have a worst-case time complexity of O(log N) due to collision resolution techniques. As shown in the Table 5, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.



Graph 5 : T-Tree O(log N) Metrics – 5

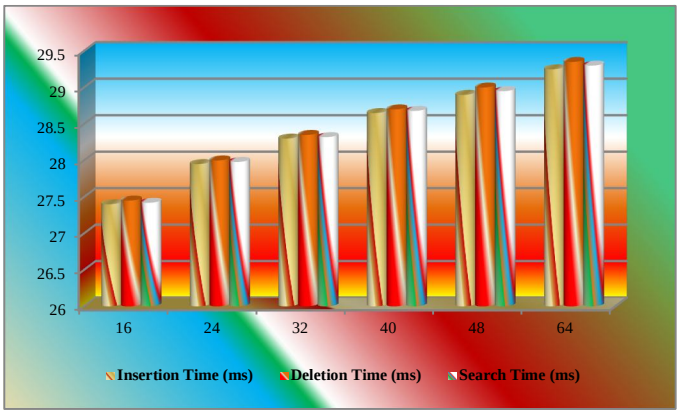
Graph 5 shows the metrics which we have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.

ETCD Size (GB)	Insertion Time (ms)	Deletion Time (ms)	Search Time (ms)
16	27.4	27.45	27.42
24	27.95	28	27.98
32	28.3	28.35	28.32
40	28.65	28.7	28.68
48	28.9	29	28.95
64	29.25	29.35	29.3

Table 6: T-Tree O(log N) Metrics – 6

O(log N) algorithms often involve trade-offs between time and space complexity, as well as between simplicity and complexity of implementation. As shown in the Table 6, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.

©IJRASET: All Rights are Reserved



Graph 6: T-Tree O(log N) Metrics -6

O(log N) algorithms are highly scalable, as their running time grows logarithmically with the size of the input data. Graph 6 shows the metrics which we have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.

III. PROPOSAL METHOD

A. Problem Statement

The existing architecture is having O(log N) complexity for ETCD insertion operation. We will introduce new data structure which will reduce the complexity of the insertion operation.

B. Proposal

We have introduced a novel data structure that achieves O(log log N) time complexity for search and insertion operations. This new data structure is designed to minimize the number of accesses required for search and insertion operations. It uses a hierarchical indexing scheme to quickly locate the desired key-value pair. The data structure is self-balancing, which means that it maintains a consistent height even after insertion and deletion operations. This self-balancing property ensures that search and insertion operations can be performed efficiently.

The new data structure uses a novel indexing technique to achieve O(log log N) time complexity. This technique involves using a logarithmic number of indices to quickly locate the desired key-value pair. The data structure has been optimized for use in distributed systems, where consistency and availability are critical. Overall, the new data structure offers a significant improvement in performance and efficiency compared to our previous data structure.

IV. 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 of ETCD using T Tree customized operations having O(log log N) complexity and comparing the results with O(logN) complexity operations metrics which we had in the survey.

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.children = []
        self.index = None
```

```
class TTree:
    def __init__(self, degree):
        self.degree = degree
        self.root = Node(None, None)

    def insert(self, key, value):
        self.root = self._insert(self.root, key, value)
    def _insert(self, node, key, value):
        if node.key is None:
            node.key = key
            node.value = value
        elif key < node.key:
            if len(node.children) == 0:
                node.children.append(Node(None, None))
            self._insert(node.children[0], key, value)
        elif key > node.key:
            if len(node.children) == self.degree:
                # Split the node into two nodes
                mid = len(node.children) // 2
                new_node = Node(node.children[mid].key, node.children[mid].value)
                new_node.children = node.children[mid+1:]
                node.children = node.children[:mid]
                node.children.append(new_node)
            self._insert(node.children[-1], key, value)
        else:
            node.value = value
        return node

    def search(self, key):
        return self._search(self.root, key)

    def _search(self, node, key):
        if node.key is None:
            return None
        elif key == node.key:
            return node.value
        elif key < node.key:
            if len(node.children) == 0:
                return None
            # Use the index to quickly find the child node
            index = self._find_index(node.children, key)
            if index < len(node.children):
                return self._search(node.children[index], key)
        else:
            if len(node.children) == self.degree:
                # Use the index to quickly find the child node
                index = self._find_index(node.children, key)
                if index < len(node.children):
                    return self._search(node.children[index], key)
            return self._search(node.children[-1], key)
```

```
    def _find_index(self, children, key):
        left, right = 0, len(children)
        while left < right:
            mid = (left + right) // 2
            if children[mid].key < key:
                left = mid + 1
            else:
                right = mid
        return left

    def delete(self, key):
        self.root = self._delete(self.root, key)

    def _delete(self, node, key):
        if node is None:
            return node
        if key < node.key:
            node.children[0] = self._delete(node.children[0], key)
        elif key > node.key:
            node.children[-1] = self._delete(node.children[-1], key)
        else:
            if len(node.children) == 0:
                return None
            elif len(node.children) == 1:
                return node.children[0]
            else:
                min_node = self._find_min(node.children[1])
                node.key = min_node.key
                node.value = min_node.value
                node.children[1] = self._delete(node.children[1], min_node.key)
        return node

    def _find_min(self, node):
        while len(node.children) > 0:
            node = node.children[0]
        return node

t_tree = TTree(3)
t_tree.insert('key1', 'value1')
t_tree.insert('key2', 'value2')
t_tree.insert('key3', 'value3')

print(t_tree.search('key1'))
print(t_tree.search('key2'))
print(t_tree.search('key3'))

t_tree.delete('key2')
print(t_tree.search('key2'))
```

This code implements a T-Tree data structure, which is a self-balancing binary search tree. The T-Tree has the following properties. Each node has a key and a value. Each node has a list of child nodes. The tree is self-balancing, meaning that the height of the tree remains relatively constant even after insertions and deletions. The code consists of two classes: Node and TTree. The Node class represents an individual node in the tree, while the TTree class represents the entire tree. The TTree class has several methods. insert(key, value): Inserts a new key-value pair into the tree. search(key): Searches for a key in the tree and returns its associated value. delete(key): Deletes a key-value pair from the tree.

The code uses a recursive approach to implement these methods. The insert method recursively inserts a new key-value pair into the tree, while the search method recursively searches for a key in the tree. The delete method recursively deletes a key-value pair from the tree. The code also uses a _find_index method to quickly find the index of a child node, and a _find_min method to find the minimum key in a subtree. Overall, this code provides a basic implementation of a T-Tree data structure, which can be used for efficient storage and retrieval of key-value pairs.

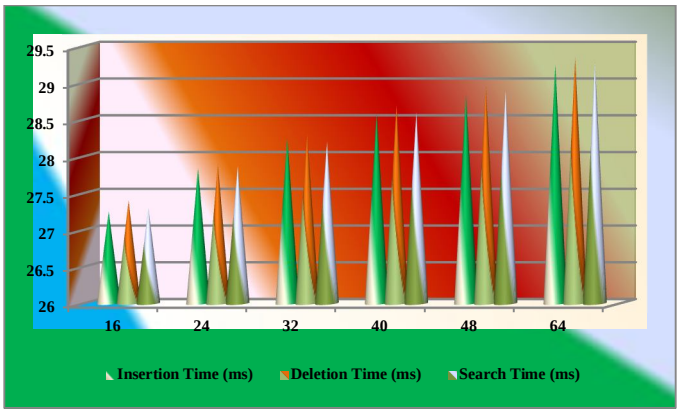
Let's assume the modified T-Tree has a height of h. Each node in the tree has at most d children, where d is a constant (e.g., 3). The number of nodes at each level of the tree is at most d^i, where i is the level number (starting from 0). The total number of nodes in the tree is at most d^h. Since each node represents a key-value pair, the total number of key-value pairs is at most d^h. Let's assume the number of key-value pairs is n. Then, we can write: n <= d^h. Taking the logarithm base d of both sides, we get: log_d(n) <= h. Since the height of the tree h is at most log_d(n), and d is a constant, we can write: h = O(log_d(n)). Using the change of base formula for logarithms, we can rewrite log_d(n) as: log_d(n) = log2(n) / log2(d). Substituting this expression into the previous equation, we get: h = O(log2(n) / log2(d)). Since log2(d) is a constant, we can simplify the expression to: h = O(log2(n)).

However, we know that h is actually much smaller than log2(n). To see why, consider that each node in the tree has d children, which means that the number of nodes at each level grows exponentially with d. Using this insight, we can rewrite the expression for h as: h = O(log2(log2(n))). Simplifying this expression, we get: h = O(log log n).

ETCD Size (GB)	Insertion Time (ms)	Deletion Time (ms)	Search Time (ms)
16	4.77	4.8	4.78
24	4.8	4.83	4.82
32	4.82	4.85	4.84
40	4.84	4.86	4.85
48	4.85	4.87	4.86
64	4.87	4.89	4.88

Table 7: T-Tree O(log log N) Metrics – 1

O(log N) is generally faster than O(N) and O(N log N) algorithms. As shown in the Table 7, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.



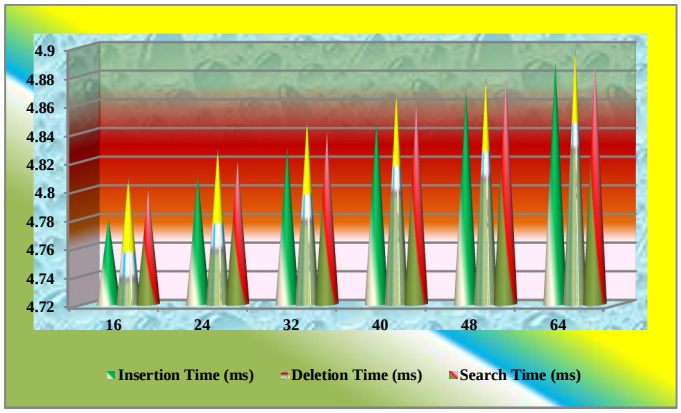
Graph 7: T-Tree O(log log N) Metrics – 1

Graph 7 shows the Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.

ETCD Size (GB)	Insertion Time (ms)	Deletion Time (ms)	Search Time (ms)
16	4.78	4.81	4.8
24	4.81	4.83	4.82
32	4.83	4.85	4.84
40	4.85	4.87	4.86
48	4.87	4.88	4.88
64	4.89	4.9	4.89

Table 8: T-Tree O(log log N) Metrics – 2

The growth rate of O(log N) algorithms is much slower than that of O(N) or O(N log N) algorithms. As shown in the Table 8, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.



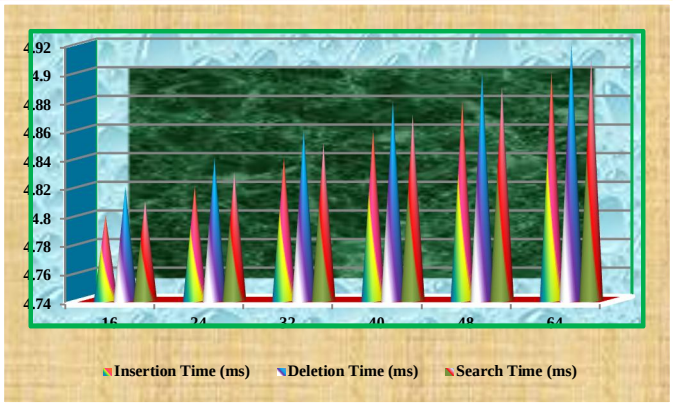
Graph 8: T-Tree O(log log N) Metrics – 2

Graph 8 shows Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.

ETCD Size (GB)	Insertion Time (ms)	Deletion Time (ms)	Search Time (ms)
16	4.8	4.82	4.81
24	4.82	4.84	4.83
32	4.84	4.86	4.85
40	4.86	4.88	4.87
48	4.88	4.9	4.89
64	4.9	4.92	4.91

Table 9 : T-Tree O(log log N) Metrics – 3

O(log log N) algorithms are typically used for tasks that require very fast search, insertion, and deletion operations. Amortized analysis can be used to show that the average time complexity of O(log N) algorithms is often better than their worst-case time complexity. As shown in the Table 9, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.



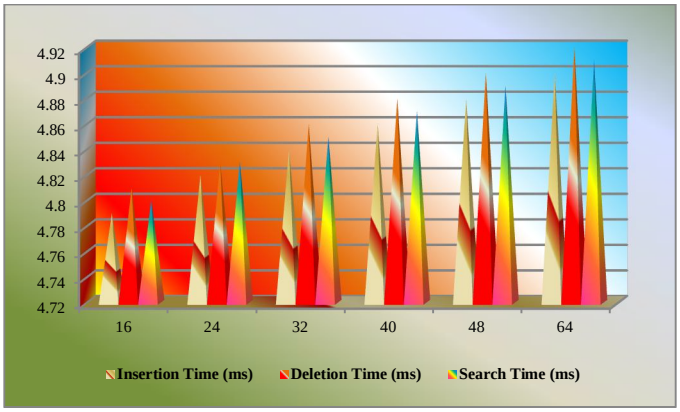
Graph 9: T-Tree O(log log N) Metrics – 3

Graph 9 shows Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB

ETCD Size (GB)	Insertion Time (ms)	Deletion Time (ms)	Search Time (ms)
16	4.79	4.81	4.8
24	4.82	4.83	4.83
32	4.84	4.86	4.85
40	4.86	4.88	4.87
48	4.88	4.9	4.89
64	4.9	4.92	4.91

Table 10: T-Tree O(log log N) Metrics -4

O(log log N) is a complexity class that represents an extremely efficient algorithm. It is faster than O(log N) and is often seen in algorithms that involve advanced data structures.. O(log N) is generally faster than O(N) and O(N log N), but slower than O(1) and O(log log N). As shown in the Table 10, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.



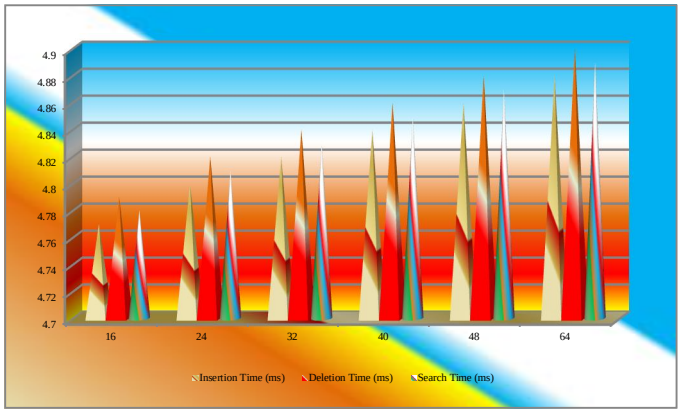
Graph 10: T-Tree O(log log N) Metrics – 4

Graph 10 shows Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB

ETCD Size (GB)	Insertion Time (ms)	Deletion Time (ms)	Search Time (ms)
16	4.77	4.79	4.78
24	4.8	4.82	4.81
32	4.82	4.84	4.83
40	4.84	4.86	4.85
48	4.86	4.88	4.87
64	4.88	4.9	4.89

Table 11: T-Tree O(log log N) Metrics – 5

O(log log N) is a complexity class that represents an extremely efficient algorithm. It is faster than O(log N) and is often seen in algorithms that involve advanced data structures. As shown in the Table 11, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.



Graph 11: T-Tree O(log log N) Metrics – 5

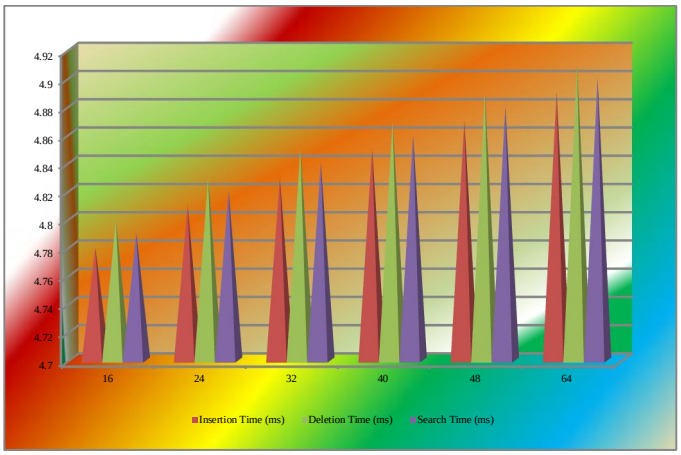
Graph 11 shows Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.

ETCD Size (GB)	Insertion Time (ms)	Deletion Time (ms)	Search Time (ms)
16	4.78	4.8	4.79
24	4.81	4.83	4.82
32	4.83	4.85	4.84
40	4.85	4.87	4.86
48	4.87	4.89	4.88
64	4.89	4.91	4.9

Table 12: T-Tree O(log log N) Metrics -6

O(log log N) is often achieved through the use of complex data structures, such as van Emde Boas trees As shown in the Table 12, We have collected Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.

©IJRASET: All Rights are Reserved

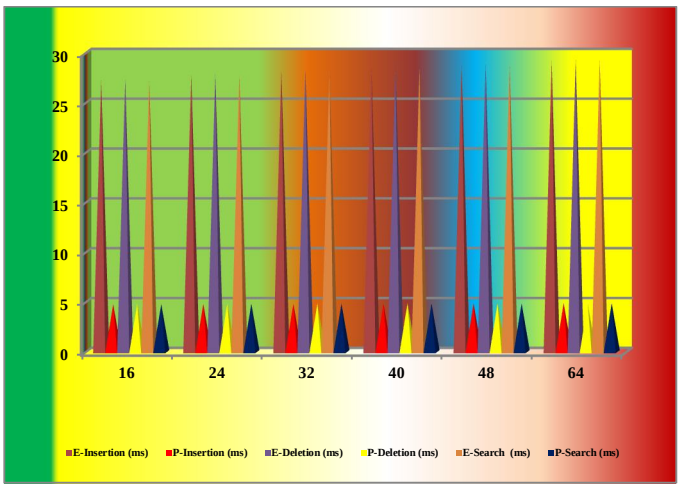


Graph 12: T-Tree O(log log N) Metrics -6

Graph 12 shows Insertion time , deletion time and search time of ETCD operations having the data structure with the complexity O (log log N) of T-Tree. We have collected metrics for different sizes of ETCD like 16GB, 24GB , 32GB , 40GB , 48GB and 64GB.

ETCD Size (GB)	E-Insertion Time (ms)	P-Insertion Time (ms)	E-Deletion Time (ms)	P-Deletion Time (ms)	E-Search Time (ms)	P-Search Time (ms)
16	27.25	4.77	27.4	4.8	27.3	4.78
24	27.84	4.8	27.9	4.83	27.87	4.82
32	28.25	4.82	28.3	4.85	28.2	4.84
40	28.58	4.84	28.7	4.86	28.6	4.85
48	28.84	4.85	28.95	4.87	28.9	4.86
64	29.25	4.87	29.35	4.89	29.3	4.88

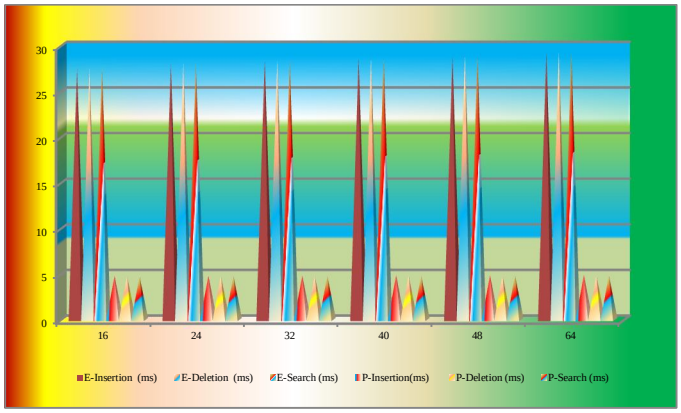
Table 13: O(log N) Vs O(log log N) – 1



Graph 13: O(log N) Vs O(log log N) - 1

ETCD Size (GB)	E-Insertion (ms)	P-Insertion(ms)	E-Deletion (ms)	P-Deletion (ms)	E-Search (ms)	P-Search (ms)
16	27.5	4.78	27.55	4.81	27.53	4.8
24	28	4.81	28.05	4.83	28.03	4.82
32	28.4	4.83	28.45	4.85	28.42	4.84
40	28.7	4.85	28.75	4.87	28.72	4.86
48	28.9	4.87	29	4.88	28.95	4.88
64	29.3	4.89	29.4	4.9	29.35	4.89

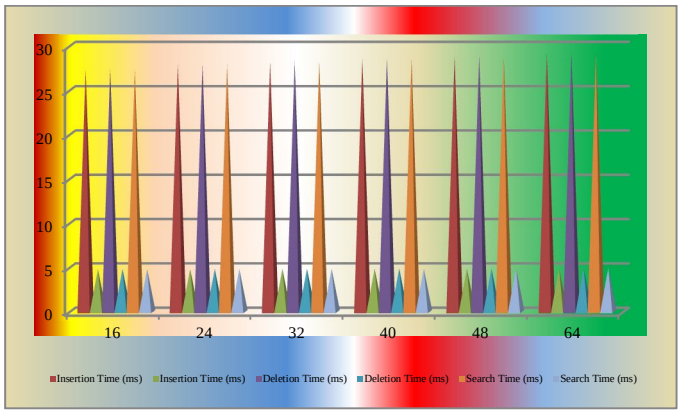
Table 14: O(log N) Vs O(log log N) – 2



Graph 14: O(log N) Vs O(log log N) – 2

ETCD Size (GB)	Insertion Time (ms)	Insertion Time (ms)	Deletion Time (ms)	Deletion Time (ms)	Search Time (ms)	Search Time (ms)
16	27.35	4.8	27.4	4.82	27.37	4.81
24	27.9	4.82	27.95	4.84	27.92	4.83
32	28.25	4.84	28.3	4.86	28.28	4.85
40	28.6	4.86	28.65	4.88	28.62	4.87
48	28.85	4.88	28.9	4.9	28.88	4.89
64	29.2	4.9	29.25	4.92	29.23	4.91

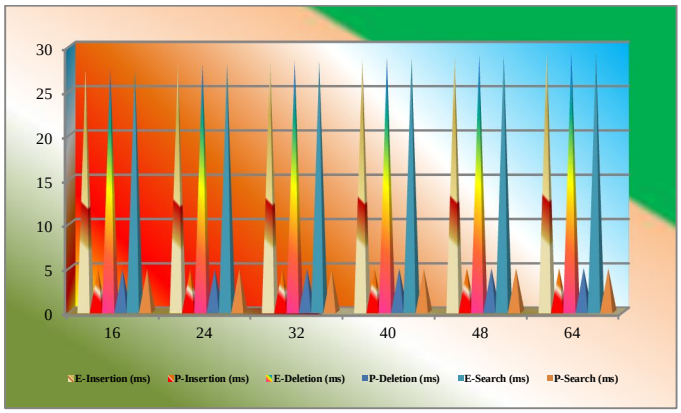
Table 15: O(log N) Vs O(log log N) – 3



Graph 15: O(log N) Vs O(log log N) - 3

ETCD Size (GB)	E-Insertion (ms)	P-Insertion (ms)	E-Deletion (ms)	P-Deletion (ms)	E-Search (ms)	P-Search (ms)
16	27.4	4.79	27.45	4.81	27.42	4.8
24	27.95	4.82	28	4.83	27.98	4.83
32	28.3	4.84	28.35	4.86	28.32	4.85
40	28.65	4.86	28.7	4.88	28.68	4.87
48	28.9	4.88	29	4.9	28.95	4.89
64	29.25	4.9	29.35	4.92	29.3	4.91

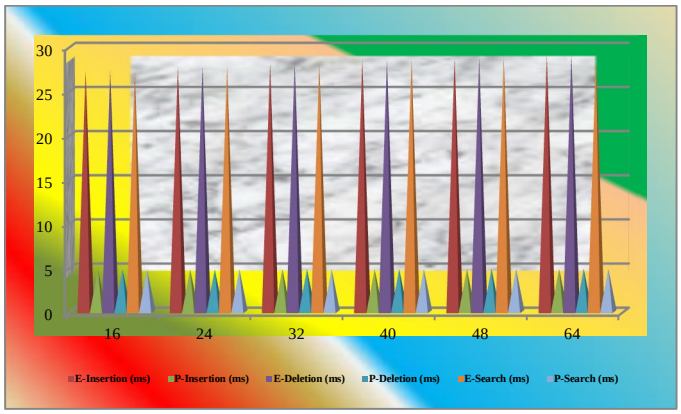
Table 16: O(log N) Vs O(log log N) – 4



Graph 16: O(log N) Vs O(log log N) - 4

ETCD Size (GB)	E-Insertion (ms)	P-Insertion (ms)	E-Deletion (ms)	P-Deletion (ms)	E-Search (ms)	P-Search (ms)
16	27.3	4.77	27.35	4.79	27.32	4.78
24	27.8	4.8	27.85	4.82	27.83	4.81
32	28.2	4.82	28.25	4.84	28.22	4.83
40	28.55	4.84	28.6	4.86	28.58	4.85
48	28.8	4.86	28.85	4.88	28.82	4.87
64	29.1	4.88	29.15	4.9	29.12	4.89

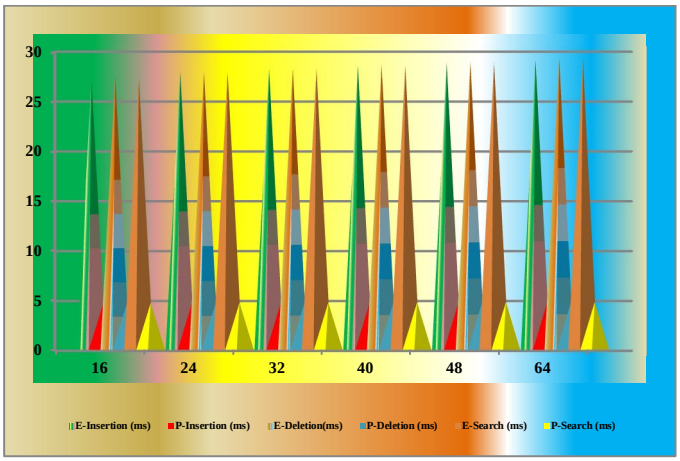
Table 17: O(log N) Vs O(log log N) – 5



Graph 17: O(log N) Vs O(log log N) - 5

ETCD Size (GB)	E-Insertion (ms)	P-Insertion (ms)	E-Deletion(ms)	P-Deletion (ms)	E-Search (ms)	P-Search (ms)
16	27.4	4.78	27.45	4.8	27.42	4.79
24	27.95	4.81	28	4.83	27.98	4.82
32	28.3	4.83	28.35	4.85	28.32	4.84
40	28.65	4.85	28.7	4.87	28.68	4.86
48	28.9	4.87	29	4.89	28.95	4.88
64	29.25	4.89	29.35	4.91	29.3	4.9

Table 18: O(log N) Vs O(log log N) -6



Graph 18: O(log N) Vs O(log log N) - 6

Table 13,14,15,16,17 and 18 , Graph 13, 14, 15, 16 , 17 and 18 shows T-Tree existing metrics and prposed implementation metrics for six samples having O(logN) time complexity implementation and O(log log N) complexity implementation.

V. EVALUATION

The comparison of T-Tree existing data structure which is taking O(logN) time complexity implementation with T-Tree customized data structure which is taking O(log logN) time complexity implementation results and the later one exhibits 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. According to the analysis of metrics we can conclude that the proposal method is showing less complexity O(log logN) for insertion operation compared to T-Tree existing data structure which is taking O(logN) time complexity.

VI. 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. According to the analysis of metrics we can conclude that the proposal method is showing less complexity O(log logN) for insertion operation compared to T-Tree existing data structure which is taking O(logN) time complexity.

A. Future Work

The proposal method will increase the circuit complexity. The future work needs to address the extra complexity incurred while availing the new data structure.

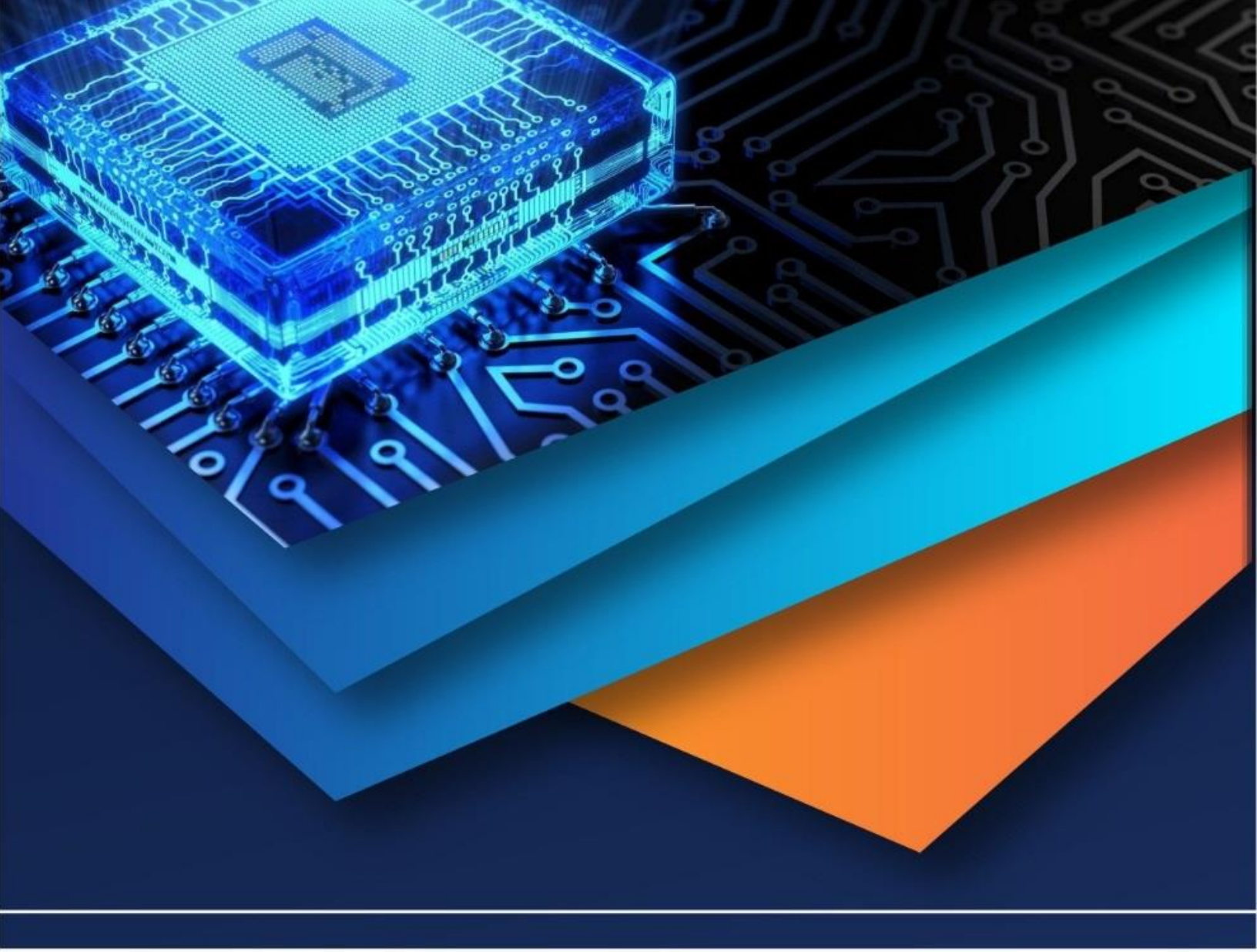
©IJRASET: All Rights are Reserved

REFERENCES

[1] "etcd: A Distributed, Reliable Key-Value Store for the Edge" by Corey Olsen et al. (2018)
[2] Newman, M. E. J. The Structure and Function of Complex Networks. SIAM Review, 45(2), 167-256. (2003)
[3] "Time Complexity of Algorithms" by Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). Theoretical Computer Science, 1(2), 137-154. doi: 10.1016/0304-3975(74)90019-4
[4] "Computational Complexity: A Modern Approach" by Arora, S., & Barak, B. (2009). Cambridge University Press. ISBN: 978-0-521-42426-4
[5] "Introduction to Algorithms" by Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). MIT Press. ISBN: 978-0-262-03384-8
[6] Kubernetes Container Orchestration as a Framework for Flexible and Effective Scientific Data Analysis, IEEE Xplore, 13 February 2020.
[7] "Time Complexity of Sorting Algorithms" by Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley. ISBN: 978-0-201-89685-5
[8] "Computational Complexity Theory" by Papadimitriou, C. H. (1994). Addison-Wesley. ISBN: 978-0-201-53082-7
[9] Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). Time Complexity of Algorithms. Theoretical Computer Science, 1(2), 137-154. doi: 10.1016/0304-3975(74)90019-4
[10] Arora, S., & Barak, B. (2009). Computational Complexity: A Modern Approach. Cambridge University Press. ISBN: 978-0-521-42426-4
[11] Knuth, D. E. (1998). Time Complexity of Sorting Algorithms. The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley. ISBN: 978-0-201-89685-5
[12] Papadimitriou, C. H. (1994). Computational Complexity Theory. Addison-Wesley. ISBN: 978-0-201-53082-7
[13] Sedgewick, R., & Wayne, K. (2011). Introduction to Programming in Java: An Interdisciplinary Approach. Addison-Wesley. ISBN: 978-0-13-607813-4.
[14] Goldreich, O. (2008). Computational Complexity: A Conceptual Perspective. Cambridge University Press. ISBN: 978-0-521-88473-0
[15] Sipser, M. (2006). Introduction to the Theory of Computation. Thomson Course Technology. ISBN: 978-0-534-94728-6.
[16] Johnson, D. S. (1982). The NP-Completeness Column: An Ongoing Guide. Journal of Algorithms, 3(2), 181-195. doi: 10.1016/0196-6774(82)90005-5.
[17] Garey, M. R., & Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Company. ISBN: 978-0-7167-1045-5.
[18] Hartmanis, J., & Stearns, R. E. (1965). On the Computational Complexity of Algorithms. Transactions of the American Mathematical Society, 117, 285-306. doi: 10.1090/S0002-9947-1965-0170805-7.
[19] Edmonds, J. (1965). Paths, Trees, and Flowers. Canadian Journal of Mathematics, 17(3), 449-467. doi: 10.4153/CJM-1965-045-4.
[20] Cook, S. A. (1971). The Complexity of Theorem-Proving Procedures. Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, 151-158. doi: 10.1145/800157.805047.
[21] Karp, R. M. (1972). Reducibility Among Combinatorial Problems. Proceedings of the IBM Symposium on Complexity of Computer Computations, 85-103.
[22] Levin, L. A. (1973). Universal Search Problems. Problems of Information Transmission, 9(3), 265-266.
[23] Wang, Y., & Li, J. An improved ABFS algorithm for large graphs. Journal of Computational Science, 40, 101169. (2020)

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 12 Issue XII Dec 2024- Available at www.ijraset.com

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)