Distributing Data with Zero Migration

Jonathan Yue [1]

[1]DataJaguar

October 31, 2023

Posted on 22 May 2020 — CC-BY-NC-SA 4 — https://doi.org/10.36227/techrxiv.22974845.v1 — e-Prints posted on TechRxiv are preliminary reports that are not peer reviewed. They should not b...

Abstract

ZeroMove hashing is a novel data distribution technique for distributed systems that offers several key benefits. In contrast to the consistent hashing algorithm, which requires data migration when scaling the system, ZeroMove hashing enables the addition of clusters of nodes on demand without the need to move data between nodes. A cluster is located using an encoded unique identifier, while a node is identified with a hash function within a cluster. This approach ensures that data remains in the node where it is hashed, thereby increasing availability and improving system performance. Furthermore, the ZeroMove hashing technique can significantly reduce facility and administrative expenses, making it an excellent option for largescale distributed systems. Our tests on consistent hashing and ZeroMove hashing have shown that scaling from one node to six nodes with 480,000 data records took 6100 seconds in a system based on consistent hashing. In contrast, it took only 1.2 seconds for ZeroMove hashing to achieve similar scaling under the same settings. With consistent hashing, the time taken and amount of data moved increase proportionally with the amount of data stored in the system. However, with ZeroMove hashing, these values does not increase in proportion to the amount of data being stored. This is because ZeroMove hashing only involves the exchange of small amount of metadata between nodes during scaling processes.

Distributing Data with Zero Migration

Jonathan Z. Yue

***Abstract*—ZeroMove hashing is a novel data distribution technique for distributed systems that offers several key benefits. In contrast to the consistent hashing algorithm, which requires data migration when scaling the system, ZeroMove hashing enables the addition of clusters of nodes on demand without the need to move data between nodes. A cluster is located using an encoded unique identifier, while a node is identified with a hash function within a cluster. This approach ensures that data remains in the node where it is hashed, thereby increasing availability and improving system performance. Furthermore, the ZeroMove hashing technique can significantly reduce facility and administrative expenses, making it an excellent option for large-scale distributed systems. Our tests on consistent hashing and ZeroMove hashing have shown that scaling from one node to six nodes with 480,000 data records took 6100 seconds in a system based on consistent hashing. In contrast, it took only 1.2 seconds for ZeroMove hashing to achieve similar scaling under the same settings. With consistent hashing, the time taken and amount of data moved increase proportionally with the amount of data stored in the system. However, with ZeroMove hashing, these values does not increase in proportion to the amount of data being stored. This is because ZeroMove hashing only involves the exchange of small amount of metadata between nodes during scaling processes.**

***Index Terms*—Hashing, distribution, horizontal scaling, cluster, database, storage, distributed systems.**

I. Introduction

THIS paper introduces the ZeroMove hashing protocol, a data distribution technique for distributed systems that eliminates the need for data migration during horizontal scaling. The protocol consists of several algorithms for efficient data distribution and system scaling, which are described in detail in this article.

A. Scaling

Horizontal scaling, also known as scaling out, is a method of increasing the capacity of a system by adding more machines or nodes to the system [1]. This is in contrast to vertical scaling, or scaling up, which involves adding more resources, such as CPU, RAM, or storage, to a single machine.

Horizontal scaling involves distributing the workload across multiple machines, which allows the system to handle more traffic or requests. This approach can improve the system's performance, increase its capacity, improve its reliability, and reduce the risk of a single point of failure. Horizontal scaling is commonly used in cloud computing and distributed systems, where applications can be scaled up or down dynamically based on the workload.

Horizontal scaling has several advantages over vertical scaling. While vertical scaling can be a simple way to increase the capacity of a system, it has several limitations:

1) Cost: As the resource requirements increase, the cost of upgrading the hardware can become prohibitively expensive. In some cases, it may be more cost-effective to add more machines horizontally rather than upgrading a single machine vertically.

2) Limitations of the hardware: The performance of a machine can only be scaled up to a certain limit. Beyond that limit, adding more resources may not result in significant improvements in performance.

3) Single point of failure: When a single machine is responsible for handling all the workload, it becomes a single point of failure. If the machine fails, the entire system can go down, resulting in downtime and lost revenue.

4) Limited scalability: Vertical scaling may not be a practical solution when the workload is highly variable or unpredictable. In such cases, horizontal scaling is a better option since it allows for dynamic scaling up or down based on the workload.

5) Downtime: When upgrading the hardware, the machine needs to be taken offline, resulting in downtime. This can be disruptive to the users and can result in financial losses.

In a storage-computing separation framework, data is stored in a separate storage system that can be scaled up dynamically on demand. However, it is important to note that a storage system is also a distributed system consisting of physical machines, or nodes, and devices that are connected through a network. Cloud storage uses horizontal scaling to increase the capacity of the storage system by adding more nodes to the cluster. When there is a need for more storage capacity, new nodes can be added to the cluster dynamically, allowing the storage system to scale up to meet the demand. The data stored in cloud storage is typically distributed across multiple nodes in the cluster using techniques such as replication, sharding, or erasure coding. In this article, we will adopt the term "scaling" to specifically refer to horizontal scaling.

B. Distribution

The vast majority of distrbuted systems employ hash-based sharding to distribute data across a group of nodes. Other systems utilize a centralized look-up service for finding nodes to store and search data. In a distributed database system that uses a centralized look-up service, the look-up service acts as a directory for storing the mapping between data keys and their corresponding nodes. When a read or write operation needs to access a specific piece of data, it first contacts the look-up service to find out which node currently owns that data.

However, the look-up service can become a single point of failure and a bottleneck for the entire system since each read or write operation needs access to the service. If the look-up service fails, the entire system may become unavailable. Additionally, as the number of nodes and data keys in the system grows, the look-up service may become overwhelmed with requests, leading to poor performance.

To address these issues, some distributed database systems use more decentralized approaches to locate data, such as distributed hash tables or gossip protocols. These approaches distribute the responsibility for maintaining the mapping between data keys and nodes among all the nodes in the system, reducing the reliance on a centralized look-up service.

Classic hashing is a technique in which a hash function is used to map data items to nodes in a fixed set of nodes. The hash function maps each data item to a specific node based on its hash value. This approach is simple and easy to implement, but it suffers from a major drawback: if a new node is added or an existing node is removed from the system, the mapping of data items to nodes may change drastically. This can result in a significant amount of data needing to be moved, which can be expensive in terms of time and network bandwidth.

Consistent hashing is a more sophisticated technique that was developed to address this limitation of classic hashing. In consisent hashing, the hash function is used to map both data items and nodes to a ring-shaped continuum. Each node is assigned a range of hash values on the ring, and data items are mapped to the node whose range of hash values includes the hash value of the data item. When a new node is added or an existing node is removed, only the ranges of the affected nodes need to be adjusted. This means that only a portion of the data needs to be moved, resulting in a more efficient and scalable system. However, the challenge of data migration has not been fully resolved. In successive scaling processes, a significant amount of data still needs to be moved multiple times, causing disruptions and potential performance bottlenecks.

C. Consistent Hashing

Consistent hashing is a technique used in distributed computing to map keys to nodes in a distributed system in a way that minimizes the amount of reorganization required when nodes are added or removed from the system [2] [3] [4] [5] [6]. The basic idea is to use a hash function to map each key to a point on a ring, and then use the ring to determine which node is responsible for each key.

Here are the mathematical expressions that describe the process of consisent hashing:

- **Hash Function** A hash function H maps a key to a value in the range [0, 1].
- **Node Positions** Each node in the distributed system is represented by a point on a ring, also known as continuum, of circumference 1. The position of the node on the ring is determined by a hash function R, which maps the identifier of a node to a value in the range [0, 1]. The node is responsible for all keys that map to a point on the ring between its own position and the position of its successor node, in clockwise order.
- **Key Mapping** To determine which node is responsible for a given key, we first apply the hash function H to the key to obtain a value in the range [0, 1]. We then find the position of the node on the ring that immediately follows this value (in clockwise order), and that node is responsible for the key.
- **Adding or Removing Nodes** When a new node is added to the system, its position on the ring is determined by the hash function R, and keys that were previously assigned to the node's successor are reassigned to the new node. When a node is removed from the system, keys that were previously assigned to that node are reassigned to its successor.
- **Virtual Nodes** To improve load balancing, a number of virtual nodes may be created out of a node. The virtual nodes are positioned on the ring similarly to the original node. If a key is mapped to a virtual node on the ring, then the key eventually is stored on the original node. The number of virtual nodes for a node acts as a weight in load balancing.

Rendezvous hashing is a technique related to consistent hashing that enables data replication with r copies among a set of N nodes [7]. Unlike consistent hashing, rendezvous hashing does not require precomputing or storage of tokens. However, it requires recomputing N hash values for every read and write operation. As N grows larger, computing all N hash values becomes costly. Consistent hashing is equivalent to rendezvous hashing when the number of sites is equal to one. Both consistent hashing and rendezvous hashing exhibit similar patterns in terms of data migration volume during scaling processes. In both approaches, when the system scales by adding nodes, there is a need to redistribute the data across the nodes.

In this section, we analyze the cost of data migration in consistent hashing. Let us assume that each node in the system has a storage capacity of C, which represents the number of data records it can store. We begin with Z nodes in the system and during each scaling process, m new nodes are added.

Theorem 1: When adding m nodes to an existing set of Z nodes in the consistent hashing method, the amount of data migration required is approximately

$$X = \frac{mZC}{Z+m} \tag{1}$$

Proof: In the consistent hashing method, when a large number of virtual nodes divide each physical node, before scaling, each node is responsible for approximately $\frac{1}{Z}$ portion of the data. However, after scaling by adding m nodes, each node will be responsible for approximately $\frac{Z}{Z+m}$ portion of the data. The migrated data amount from each of the Z nodes is:

$$C - \frac{ZC}{Z+m} = \frac{mC}{Z+m} \tag{2}$$

In total, the migrated data amount from all the Z nodes is:

$$X = \frac{mZC}{Z+m} \tag{3}$$

Another perspective on this problem is to consider dividing the total data amount, denoted as ZC, among $Z+m$ nodes. In this scenario, each node would be responsible for $\frac{ZC}{Z+m}$ data records. With the addition of m new nodes, the total number of records on these nodes would be $\frac{mZC}{Z+m}$, which is equivalent to the number of data records that are being moved during the scaling process.

Approximation arises when the number of virtual nodes is not large enough to achieve perfect data distribution and balance among the physical nodes. Due to the limited number of virtual nodes, there might be slight imbalances in the distribution of data across the physical nodes. However, this approximation is often considered acceptable as it still provides a reasonably balanced data allocation in practice. *Q.E.D*

We now study the behavior of consistent hashing during the incremental scaling processes of a system.

Theorem 2: To scale the system from initially Z_0 nodes to $Z_0 + Nm$ nodes containing $C(Z_0 + Nm)$ data records, the total number of data records that undergo migration after N rounds of scaling operations is approximately mCN.

Proof: At the k-th scaling, the total number of nodes, Z, prior to the k-th scaling is $Z = Z_0 + (k-1)m$, where Z_0 is the number of nodes in the initial cluster. Addingg m new nodes to the value, we obtain the total number of nodes: $Z_0 + (k-1)m + m = Z_0 + km$. Now referring to Theorem 1, data migration amount at the k-th scaling is:

$$\begin{aligned} S(k) &= X \\ &= \frac{mCZ}{Z+m} \\ &= \frac{mC[Z_0+(k-1)m]}{Z_0+km} \\ &= \frac{mC(Z_0+km-m)}{Z_0+km} \\ &= \frac{mC(Z_0+km)}{Z_0+km} - \frac{Cm^2}{Z_0+km} \\ &= mC - \frac{Cm^2}{Z_0+km} \end{aligned} \tag{4}$$

After a total of N times of scaling, the total migration amount is obtained by summing up the amount of data migrated during each scaling process, ranging from k = 1 to N. This cumulative sum captures the overall data movement that occurs during the successive scaling operations. The value of the sum $S(N)$, is expressed as follows:

$$\begin{aligned} S(N) &= \sum_{k=1}^{N} S(k) \\ &= \sum_{k=1}^{N} (mC - \frac{Cm^2}{Z_0+km}) \\ &= mCN - \sum_{k=1}^{N} \frac{Cm^2}{Z_0+km} \\ &= mCN - \sum_{k=1}^{N} \frac{Cm}{(Z_0/m)+k} \end{aligned} \tag{5}$$

To evaluate the term $\sum_{k=1}^{N} \frac{1}{(Z_0/m)+k}$ we use the Harmonic numbers:

$$H(N) = \sum_{k=1}^{N} \frac{1}{k} \tag{6}$$

which is approximately equal to:

$$H(N) \approx \ln N + \gamma - \frac{1}{2N} \tag{7}$$

where $\gamma \approx 0.5772$ is the Euler-Mascheroni constant.

Given the behavior of the Harmonic numbers, we can obtain:

$$\sum_{k=1}^{N} \frac{1}{(Z_0/m)+k} < H(N) \tag{8}$$

$$\begin{aligned} S(N) &> mCN - mCH(N) \\ &> mCN - mC(\ln N + \gamma - \frac{1}{2N}) \end{aligned} \tag{9}$$

Therefore, the asymptotic complexity of $S(N)$ can be expressed as $S(N) = mCN$. To scale the system to have $Z_0 + Nm$ nodes containing $C(Z_0 + Nm)$ data records, the total number of data records that undergo migration after N scaling operations is approximately mCN. *Q.E.D*

Theorem 3: In a distributed system utilizing consistent hashing, the majority of data records created and stored within the system will experience migration from one node to another. This migration occurs as a result of a sequence of incremental scaling processes that the system undergoes.

Proof: The decimal portion of data records that undergo migration after N scaling operations can be calculated by dividing the total number of migrated records by the total number of records in the system, which is $C(Z_0 + Nm)$.

$$p(N) = \frac{mCN}{C(Z_0+Nm)} = \frac{Nm}{Z_0+Nm} \tag{10}$$

As N grows large, the fraction $p(N)$ approaches 1, indicating that every data record will eventually be moved from one node to another. *Q.E.D*

This excessive data migration results in high costs for the system, including increased power consumption, hardware wear and tear, and degraded performance. While there have been several attempts to reduce data migration, many of these approaches still fall within the scope of consisent hashing [8] [9] [10] [11] [12] [13] [14]. While these methods may differ in their specific implementation details, they do not fundamentally change the underlying framework of consisent hashing. In other words, while there may be some variation in how data migration is handled, the core principles of consisent hashing remain the same.

Next, we introduce a new framework to address the challenge of data migration during the scaling of distributed systems. This new framework is intended to eliminate the need for any data to be moved during the process of scaling, which can be a complex and time-consuming task that may result in downtime or other issues if not managed carefully.

II. ZeroMove Hashing

ZeroMove hashing protocol is a method of distributing data to nodes arranged in clusters. Unlike other hashing protocols, ZeroMove hashing does not require data to be moved from one node to another during a scaling process. Rather, data remains in the node where it was originally hashed indefinitely. The following formulation outlines the process of ZeroMove hashing.

A. State

The state of a distributed system can be defined as the collective knowledge about the system at a particular point in time. It encompasses the configuration and status of all the nodes and clusters that make up the system, as well as the metadata of all user data. The state represents the metadata encompassing the entire system.

- **Cluster** A cluster k, where $k \in \{1, 2, 3, ... N\}$, having k_n nodes in the cluster.
- **Clusters** At any point of time, the system has a total of N clusters.
- **Active Clusters** A set of clusters that are available for storing new data.
- **Nodes** A node in cluster k is denoted by n_{ki}, where $i \in \{0, 1, 2, 3, ...k_n - 1\}$.
- **Topology** The way in which the nodes and clusters are connected and arranged.
- **Metadata** Knowledge of a set of data records.

The metadata in the distributed system provides important information about the data records stored in the system. Typically, each data record has a unique identifier and an associated value, and the metadata describes these attributes along with other relevant information such as the structure, size, location, and access permissions. Compared to the amount of user data stored in the system, the metadata is relatively small and faster to transmit over the network.

In the distributed system, the nodes in the active clusters are designated for storing new data. The system maintains a roster of nodes that can receive and store new data. The decision to designate a cluster as active can be made algorithmically or through a predefined policy. Typically, clusters that have available storage space are chosen as active clusters. However, even clusters that are not designated as active can still perform data reads from the system. This allows for a more efficient use of resources in the distributed system.

B. Protocol

The ZeroMove hashing protocol is applied to data operations and administrative procedures in the following manner:

Protocol 1 ZeroMove Protocol

Input A list of N clusters each containing a set of nodes
Output A ZeroMove distributed system
Main Protocol

1) Initialize the first cluster, N = 1;
2) Insert data with the Insert Algorithm, or
3) Search records with the Search Algorithm, or
4) Update data with the Update Algorithm, or
5) Delete records with the Delete Algorithm, or
6) Add a new cluster, $N \leftarrow N + 1$, with the Scale Algorithm;
7) Proceed to Step 2 to continue

The protocol begins with the initialization of a cluster that can have any number of nodes. Once initialized, the system is ready to perform data operations such as insert, search, update, and delete in any order. However, at some point, the system may need to be scaled out for two main reasons.

Firstly, capacity increase may be required when the storage capacity of the current system is reached, and additional storage is needed. Secondly, performance increase may be necessary when more nodes and resources are required to increase read/write IOPS or both. In storage capacity scale-out, performance may not be an issue, and in performance capacity scale-out, storage capacity may not be an issue. In some cases, scaling out may resolve both storgae and performance issues.

In this article, the term "capacity" refers to either storage capacity or performance capacity. Storage capacity refers to the amount of data that can be stored within a system, while performance capacity relates to the system's ability to handle and process I/O workload within a given time frame. Input capacity represents the capacity of the system to process and store new data or perform write operations efficiently. On the other hand, output capacity represents the capacity of the system to retrieve and deliver data or perform read operations efficiently.

When the system needs to be scaled out, a new cluster must be introduced to the system to handle the increased capacity demand. The scaling method is outlined in the Scale Algorithm, which is described in later sections of the paper.

C. Insert

A distributed system is designed to efficiently store and retrieve data across a network of nodes. One of the key functions of a distributed system is to shard data and allocate nodes for storage in a timely and efficient manner. The algorithm below outlines the process for allocating nodes and inserting data into the system:

The function $getCluster$ is responsible for inspecting the current state of the system and identifying an available active cluster in the system. It is essential for determining the designated cluster for new data and storing the new data to the appropriate cluster.

In the distributed system, there are several methods that can be used to allocate an available active cluster for storing new data. These methods include least-used cluster, round-robin, and performance-based load balancing.

Algorithm 1 Insert Algorithm

Input Data D and state S
Output New state S'
Procedures
$k \leftarrow$ getCluster(S);
$uid \leftarrow$ encoder(D, S, k);
$i \leftarrow$ hash(uid) mod k_n;
Store data D and uid on node n_{ki}

The least-used cluster method involves selecting the cluster with the least amount of data already stored. This technique ensures that the workload is distributed evenly across the active clusters and helps to prevent overloading of any particular cluster.

Round-robin is another method used for allocating clusters in the distributed system. With this method, the system allocates the next available active cluster for storing new data in a circular manner. This ensures that the workload is evenly distributed across all the active clusters, regardless of their capacity.

In addition to these methods, performance-based load balancing can also be used in a distributed system to allocate an available active cluster for storing new data. This method involves selecting a cluster based on its performance capacity, which helps to ensure that the new data is stored in a cluster that can handle the workload effectively.

One technique that can combine or mix the previously mentioned techniques is a hybrid load balancing approach. In this approach, multiple load balancing methods are used together to balance the workload across the active clusters. The hybrid load balancing approache offers greater flexibility and can be tailored to meet the specific requirements and characteristics of a distributed system.

To ensure the uniqueness of each piece of data within the system, the $encoder$ function combines the information of the data D, the system state S, and the active cluster number N to form a unique identifier uid. Timestamps, MAC addresses, process IDs, and IP addresses could be used in the encoder function. More importantly, the available and active cluster number found by the function $getCluster$ must be incorporated in the unique identifier uid. The $encoder$ function plays a critical role in ensuring that each piece of data in the system is uniquely identifiable and can be easily retrieved.

The $encoder$ and $decoder$ functions are two-way mapping functions that can encapsulate and extract information in the identifier uid. The mod function is the modulo operation. The insertion of data into different nodes within the system can be simultaneously carried out in parallel by multiple clients.

D. Search

Efficient data search is a fundamental feature of any distributed system. In general, there are three types of data search: point query, range query, and full scan.

Point query involves searching for a single record in the system based on a given identifier. The system returns the associated value, which is unique to that identifier. This type of search involves only one data record and is very fast by quickly finding the target cluster and node.

Range query, on the other hand, involves searching for a range of records that meet certain criteria. For example, a range query might involve finding all records within a certain time period. If the unique identifier uid is encoded with timestamp as the leading field, then range query by time period is possible and fast. Range queries require the system to search through a larger set of data records.

Finally, a full scan of all data records might be necessary in certain scenarios. This type of search involves examining every record in the system to find the data that meets specific criteria. Full scans typically require the system to read data records in all the nodes and in all the clusters. In the following, we extend the concept of range queries to encompass full scans that span the entire range from negative infinity to positive infinity.

In range queries, filter conditions may be applied to narrow down the search to only those records that meet specific criteria. These filter conditions might include constraints on the data values themselves, such as a range of values or a specific value, or more complex arithmetic and logical operations involving multiple fields.

The search algorithm for point queries in the system is outlined below:

Algorithm 2 Search Algorithm

Input Unique identifier uid
Output Data D
Procedures
$k \leftarrow$ decoder(uid);
$i \leftarrow$ hash(uid) mod k_n;
Verify or fetch data D on node n_{ki}

The function $decoder$ must be capable of extracting the cluster information from the unique identifier uid, and utilizing it to locate the data linked to the identifier. The primary objective of the search algorithm is to determine the cluster and node that store the required data. In a data search operation, the objective may be to retrieve the value associated with a given identifier or simply to determine whether the value exists or not.

During a range search operation, both the desired range and filtering condition are distributed to all nodes across all clusters. Each node then searches for data records that fall within the given range and/or meet the specified filtering condition.

Just like the insertion operation, the search for data in the system can also be independently and concurrently performed by multiple clients.

E. Update

An update operation changes the data of one or more records in a system. After identifying the cluster and node associated with a unique identifier uid using the $SearchAlgorithm$, an update operation can be carried out on that node. ange updates are executed in a similar manner as range queries.

F. Delete

Once the cluster and node that store the data record associated with a unique identifier uid have been located using the search algorithm 2, it is possible to perform a delete operation on the node.

G. Scale

At some point, a distributed system may need to be scaled out to meet storage capacity or performance capacity requirements. The following algorithm outlines the process for scaling:

Algorithm 3 Scale Algorithm

Input System state S
Output New system state S'
Procedures
Determine the need for scaling;
Identify the appropriate size of a new cluster;
Create a new cluster $N+1$, with a number of nodes;
Add the new cluster to the active cluster set;
Broadcast or gossip state change from S to new state S'

The system should be monitored to identify the need for scaling, which may be triggered by reaching storage capacity limits or experiencing performance issues. The size of the new cluster should be determined based on the requirements of the system, such as storage capacity or performance needs.

Creating a new cluster involves provisioning a set of nodes, allocating network connectivity, and installing the necessary software on each node. The following equations provide insights into determining the appropriate size of the new cluster that needs to be introduced to the system for capacity scaling.

We define I_{ps} as the input per second, representing the data ingestion capacity of each node, and O_{ps} as the output per second, indicating the output capacity of each node. The total number of nodes in all existing clusters that can accept new data is denoted as N_w. In the case of introducing a new cluster, the number of nodes in this cluster is represented by m. It is assumed that the m new nodes have a higher input capacity, denoted as gI_{ps}, and a higher output capacity, denoted as gO_{ps}, where $g \geq 1$.

In this context, the nodes capable of accepting new data and storing it are referred to as input nodes, while the nodes capable of performing data reads are referred to as output nodes. An input node is a member of an active cluster. The input capacity signifies the maximum rate at which a node can process incoming data, while the output capacity indicates the maximum rate at which a node can execute data outputs. An input node in the system has the capability to handle both data read and write operations.

Theorem 4: When a new cluster with m nodes, where m ¿ 1, is added to the system, the read capacity of the system is always increasing.

Proof: The system is designed to enable each node in both the existing clusters and the new cluster to independently perform data reads. Consequently, the input capacity of the existing clusters is determined as $N_w \times I_{ps}$. After the scaling process, the input capacity of the system is augmented to $N_w \times I_{ps} + m \times gI_{ps}$, which is always greater than the input capacity of the existing clusters, $N_w \times I_{ps}$. Therefore, the introduction of m nodes to the system invariably leads to an increase in the overall input capacity of the system. *Q.E.D*

Theorem 5: Upon introducing a cluster consisting of m nodes to the system, where $m > 1$, the input capacity of the system experiences an increase if and only if the condition $gm > N_w - M_w$ is satisfied. Here, N_w represents the total number of input nodes present in all existing clusters before the scaling process, and M_w represents the total number of input nodes in all existing clusters after the scaling.

Proof: Following each scaling operation, the total number of input nodes within the existing clusters may undergo changes. Nodes that have reached their storage capacity limits will be removed from the roster of input nodes. As a result, we have $N_w \geq M_w$. Prior to scaling, the input capacity is determined as $I_{ps} \times N_w$. However, after completing the scaling process, the input capacity becomes $I_{ps} \times M_w + gmI_{ps}$. In order to ensure that the scaling results in an increase in input capacity, the following inequality must hold true:

$$\begin{aligned} I_{ps} \times M_w + gmI_{ps} &> I_{ps} \times N_w \\ M_w + gm &> N_w \\ gm &> N_w - M_w \end{aligned} \tag{11}$$

Q.E.D

Theorem 5 proves valuable in situations where a production system experiences a high rate of data ingestion, leading to the depletion of input capacity. In such scenarios, increasing the input capacity becomes imperative to ensure the system can effectively handle the rapid influx of data.

The process of removing input nodes from the roster of input nodes can be achieved through a distributed approach, wherein each node monitors its own usage level relative to its storage capacity. When the usage level reaches its limit, it initiates a request to all other nodes in the system, requesting its removal from the roster. Subsequently, all nodes in the system respond and take the necessary actions, resulting in the removal of the specified node from the entire system.

There are other cases where the system already possesses sufficient input capacity, but it has reached its storage capacity limit. In these instances, adding new clusters and nodes becomes crucial to scale the system and accommodate additional storage requirements.

Suppose the actual input rate of the system is denoted as t. During the scaling process, it becomes essential to guarantee that the new input capacity is not less than t, as demonstrated below:

$$\begin{aligned} I_{ps} \times M_w + gmI_{ps} &> t \\ I_{ps}(M_w + gm) &> t \end{aligned} \tag{12}$$

The appropriate selection of the new cluster size, denoted as m, is crucial in meeting the input rate requirement based on the values of I_{ps}, t, M_w, and g. It is necessary to carefully

determine the size of the new cluster to ensure that the input rate can be effectively accommodated and sustained by the system.

Once the optimal size for the new cluster has been determined, the new nodes are assembled into a functional cluster. This newly formed cluster is then added to the system through a network command that is simultaneously propagated to all nodes. By executing the command in parallel across the system, the addition of the new cluster can be efficiently and swiftly coordinated. In certain scenarios, partial or complete state information may also be sent to client nodes. This enables client nodes to generate unique identifiers locally, utilizing the received information, without the need to request data from a server node.

In a large distributed system, using a unique identifier for each data record is crucial for quickly finding data associated with an identity. With a vast number of objects such as devices, mobile targets, virtual objects, messages, documents, transactions, events, and parts, quickly locating these objects is a crucial function. The ZeroMove hashing protocol is particularly well-suited for such applications. By ensuring that data remains in the node where it is hashed to forever, the protocol reduces the need for data migration and improves system efficiency. This feature, coupled with the ability to quickly locate data using unique identifiers, makes the ZeroMove hashing protocol an excellent choice for distributed systems handling large amounts of data.

H. Attribute Key

The unique identifier, often denoted as uid, is commonly employed to uniquely identify data records within the system. However, there are situations, such as when indexing numeric values, where one or more attributes of a data record can serve as effective identifiers. These attributes, known as attribute keys, can be either individual attributes or a combination of attributes that possess the ability to uniquely identify each data record. By utilizing attribute keys, alternative approaches to identification can be implemented in specific contexts or scenarios.

Adopting this approach, the Insert Algorithm 1 can be modified as follows:

Algorithm 4 Insert Algorithm 2

Input Attribute key A, Data D and state S
Output New system state S'
Procedures
$k \leftarrow$ getCluster(S);
$i \leftarrow$ hash(A) mod k_n;
Store data D and A on node n_{ki}

Algorithm 4 functions in a similar manner to Algorithm 1, but with a notable difference. In Algorithm 4, the determination of the hashed node involves directly hashing the attribute key A against an available cluster, instead of relying on a unique identifier that may not be accessible or relevant in certain cases.

Similarly, the search algorithm 2 would undergo modifications to accommodate the changes described below.

Algorithm 5 Search Algorithm 2

Input Attribute key A, and cluster number N
Output Data D
Procedures
for k = 1 **to** N **do**
  $i \leftarrow$ hash(A) mod k_n;
  Search A on node $n_k i$;
  if A is found on node $n_k i$ **then**
    Cache k for A;
    return data D of A
  else
    Continue to cluster $k + 1$
  end if
end for
Subsequent searches of A go to cached cluster k and
Find A on its hashed node $i \leftarrow$ hash(A) mod k_n

The search algorithm 5 begins by identifying the cluster that contains the attribute key A and its associated data D. Subsequent searches for A will then only visit the hashed node within the discovered cluster k, using the same hash function. Initially, a search of an item may require visiting all the clusters in the system. However, once the correct cluster has been identified, subsequent searches for other properties of the same item will only need to access the corresponding cluster, resulting in faster search times.

The ZeroMove technique can also be beneficial for analytical tasks that require reading data from all nodes in a well-balanced system. Range searches, similarity studies, or full scans of data records might need to visit all the clusters in the system. However, for these types of tasks, using attribute keys to read a collection of data records from multiple clusters and nodes may be sufficiently efficient. The ZeroMove technique allows for efficient scaling of the system while maintaining a balanced data distribution, which can be particularly advantageous for analytical workloads. By minimizing data migration and maintaining optimal load distribution, ZeroMove can further enhance the efficiency of such tasks in a distributed database environment.

I. Virtual Nodes

In the aforementioned algorithms, we made the assumption that all nodes within a cluster have the same storage capacity and are assigned an equal share of the workload. Consequently, data distribution across the nodes in the cluster would be approximately uniform. However, in practice, it is possible for certain nodes to have greater storage capacity compared to others.

To address this imbalance and achieve better load balancing, we can employ the technique of virtual nodes within Consisent hashing. This approach involves assigning different weights to individual nodes based on their storage capacity. The weight assigned to a node corresponds to the total number of virtual nodes associated with it. Therefore, in algorithms 1, 2, 4, and 5, the hashing function $i \leftarrow$ hash(V) mod k_n should be modified to $j \leftarrow$ hash(V) mod k_v where V is either uid or

A, j is index of the virtual node, and k_v represents the total number of virtual nodes in cluster k. The virtual node index j is then mapped to the real node index i with a map in the state S.

J. Replication

Data replication on multiple nodes is a technique used in distributed systems to improve data availability, fault tolerance, and performance. In this approach, copies of data are maintained on multiple nodes within the system. When data is updated or added, it is replicated across the designated nodes to ensure consistency.

We define a replica node as the designated node that is responsible for storing a copy of the original data. Replica nodes can be selected with a replica selection function $R(i, k)$:

$$R(i, k) = getReplicaNodes(i, k, S) \tag{13}$$

To determine the replica nodes for a given node n_{ki}, the function $getReplicaNodes(i, k, S)$ is designed to consider the network topology of the system. By leveraging information such as the node index i, cluster index k, and the state S of the system, this function returns a list of replica nodes that are appropriate for the specific node n_{ki}.

III. Result

We conducted tests to compare the performance of consisent hashing and ZeroMove hashing techniques in a setup comprising six nodes. Each node had 2TB HDD, 72GB RAM, 2.4GHz CPU, and a gigabit local area network. We used LevelDB as the underlying storage engine, and all the nodes maintained the same network topology. A consistent-hash ring was shared and synchronized among the nodes.

In our test, we first inserted a number of data records into the first node and then scaled the system by adding another node. We continued adding more records by distributing them into the two nodes until there were six nodes in the system. Throughout this process, we measured the time taken for scaling and data migration, as well as the volume of data migrated.

The following steps demonstrate the process:

1) Insert B data records into the first node;
2) Scale the system by adding a second node;
3) Distribute B more data records across the two nodes;
4) Scale the system by adding a third node;
5) Distribute B more data records across the three nodes;
6) Repeat steps 2 and 5 until there are six nodes in the system.

In our tests, each data record comprises a 22-byte key and a 565-byte value.

Figure 1 displays the time taken for scaling in the consistent hashing approach. The experiment began by writing ten thousand data records, $B = 10K$, to the first node in the system. Subsequently, the system was scaled out by adding a second node, which involved migrating data from the first node. Another batch of ten thousand records was then stored in the system by distributing them among the available nodes.

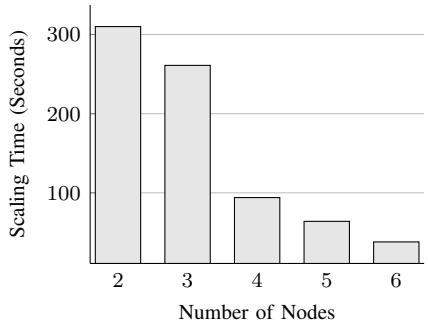

Fig. 1. Consistent Hashing With 60K Records

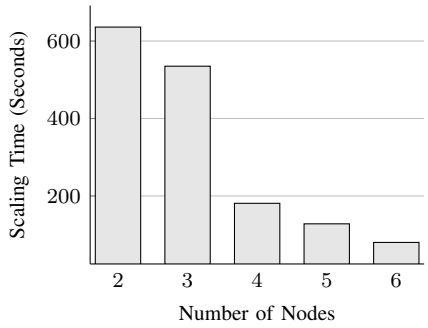

Fig. 2. Consistent Hashing With 120K Records

As a result, the system consisted of sixty thousand records distributed across six nodes.

Figure 2 illustrates the time taken for scaling in the consistent hashing approach when the stored data size is doubled. The experiment began by writing twenty thousand data records, denoted as $B = 20K$, to the first node in the system. Subsequently, the system was scaled out by adding a second node, which required migrating data from the first node. Another batch of twenty thousand records was then stored in the system, with the distribution spread among the available nodes. Consequently, the system consisted of one hundred twenty thousand records distributed across six nodes. It is noteworthy that the scaling time exhibited a significant increase compared to the case with $B = 10K$ as shown in Figure 1.

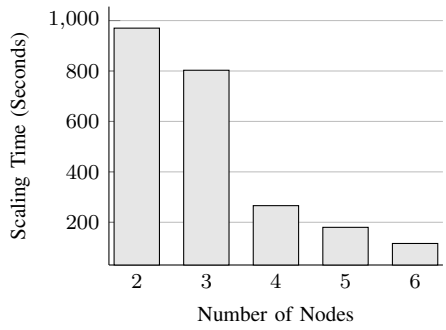

Fig. 3. Consistent Hashing With 180K Records

Figure 3 illustrates the time taken for scaling in the Consistent Hashing approach when the stored data size reaches one

hundred eighty thousand records, denoted as $B = 30K$. As observed from the results, the scaling time is further increased compared to the previous experiments.

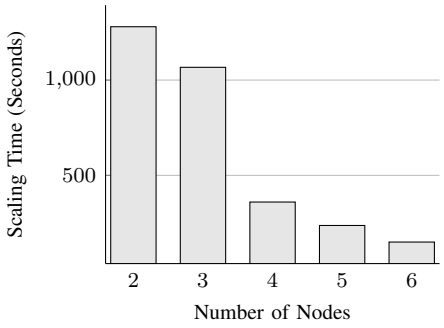

Fig. 4. Consistent Hashing With 240K Records

When the stored data size reaches two hundred forty thousand records, $B = 40K$, the scaling time is further increased compared to the case of $B = 30K$. The result is shown in Figure 4 for scaling in the Consistent Hashing approach.

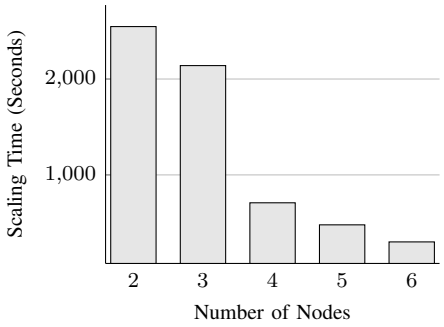

Fig. 5. Consistent Hashing With 480K Records

A series of experiments were conducted in the consistent hashing system with more varying values of $B = 50K, 60K, 70K, 80K$. It was observed that as more data is stored in the system, the scaling time increases. For instance, when there are four hundred thousand eighty data records stored in the system, the scaling time is depicted in Figure 5. The redistribution of the first 80,000 records took 2547 seconds, and the total time taken to scale from one node to six nodes was 6176 seconds, including the subsequent redistribution steps (2140 seconds, 709 seconds, 479 seconds, and 301 seconds).

Figure 6 presents a stacked graph for the results of time taken for scaling in consistent hashing. Each column in the graph represents the total time taken to redistribute data in the system with different values of B. For instance, the rightmost column displays the total time taken (6176 seconds) for scaling when the system stores 480,000 data records with parameter B set to 80,000. Linear regression analysis of the graph estimates that when the system eventually stores 4.8 billion records, the total time spent on scaling processes would be around 61 million seconds.

The scaling time values may vary depending on various factors such as system resources and storage engine utilized

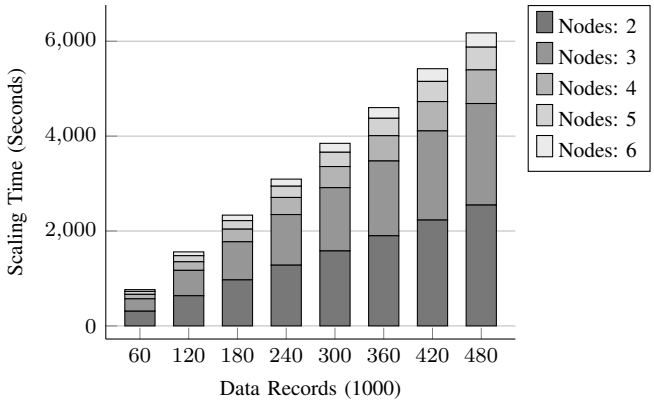

Fig. 6. Scaling Time In Consistent Hashing

in a system that employs consisent hashing. A faster network can result in quicker data migration. However, it is evident that the migration time is directly proportional to the size of the stored data that needs to be redistributed among the nodes.

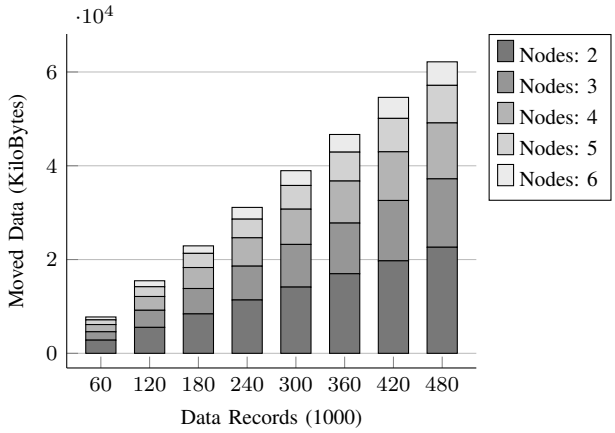

Fig. 7. Migration Size In Consistent Hashing

Figure 7 illustrates the amount of data that is migrated during the scaling process using consistent hashing. Each column on the graph represents the total volume of data that is redistributed in the system. For example, the rightmost column displays the total amount of data moved (61,000 KB) when the system stores 480,000 data records. As shown, the volume of data migration increases proportionally with the amount of data stored in the system.

The study of the scaling behavior of consistent hashing provides valuable insights into the potential cost reductions offered by the ZeroMove hashing method. Unlike consistent hashing, the ZeroMove hashing approach eliminates the need for scaling time and significantly reduces data migration. In ZeroMove hashing, no user data is migrated, and only a small amount of control information is exchanged over the network. As a result, the complexities associated with consistent hashing are drastically reduced in the ZeroMove hashing method. This highlights the significant advantages and efficiency improvements that ZeroMove hashing brings to the scaling process.

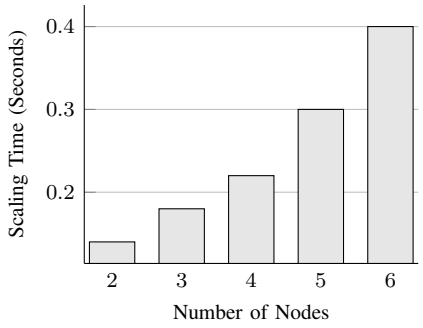

Fig. 8. Scaling Time In ZeroMove Hashing

Figure 8 depicts the time taken to scale a system based on ZeroMove from a single node to multiple nodes. The experiments were conducted on the same machines as the consistent hashing tests, as described in [15]. With the ZeroMove system, adding a new node takes only a matter of sub-seconds. The total time taken to scale from one node to six nodes is 1.24 seconds. Additionally, separate experiments were conducted to add a new cluster of three nodes to an existing cluster of three nodes, which was completed in 1.1 seconds.

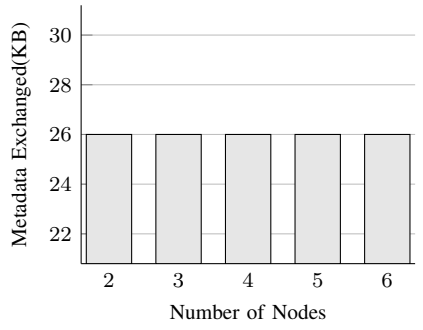

Fig. 9. Metadata Exchange In ZeroMove Hashing

During the scaling process of the ZeroMove hashing method, no user data was migrated. Instead, only metadata was exchanged between the nodes and the client. The size of the metadata was relatively small, approximately 26 kilobytes, in contrast to the considerable amount of user data, which was around 64,000 kilobytes, migrated during the consistent hashing experiments. It is important to note that in real production systems, the amount of user data can reach multiple petabytes, and migrating such a massive volume of data would be an extremely time-consuming task.

The tests conducted clearly demonstrate that the ZeroMove hashing technique provides significant advantages over the consisent hashing hethod. The source code and test scripts for the benchmark can be found on the GitHub repository at github.com/fserv/jaguardb benchmark directory. The benchmark uses LevelDB as the storage engine, and the Ketama package is used to manage the consistent-hashing ring data structure. To test ZeroMove hashing, the package from github.com/fserv/jaguardb was downloaded and included in the benchmark tests.

IV. Discussion

In a single-node system, data migration in consisent hashing and classic hashing algorithms may not incur significant costs. This is primarily because the data migration occurs within the same node, and in-memory data writes are generally very fast. In such scenarios, the data can be easily reorganized within the node by updating the relevant data structures in memory. As a result, the overhead of copying or moving data within the memory is minimal compared to other I/O operations.

However, when it comes to data migration across nodes in a network, the costs can be much higher. Data migration in a distributed system involves transferring data between different nodes, which introduces additional complexities and challenges. Unlike in a single-node system where data can be moved within the memory, migrating data across nodes requires network communication and coordination, which can be resource-intensive.

Network communication introduces various overheads that can impact the efficiency and performance of data migration. These overheads include: 1) Handshaking: Before data transfer can begin, there is often a handshaking process between the sender and receiver nodes to establish a connection and agree on communication parameters. This handshake introduces additional latency and overhead. 2) Flow control: To ensure smooth and efficient data transfer, flow control mechanisms are employed to regulate the rate at which data is sent and received. This involves monitoring the receiver's buffer capacity and adjusting the transmission rate accordingly. Flow control mechanisms can introduce delays and overhead. 3) Congestion control: In network environments where multiple nodes are concurrently transferring data, congestion control mechanisms come into play. These mechanisms detect and manage network congestion to prevent packet loss and ensure fair resource allocation. Congestion control adds complexity and introduces additional latency. 4) Packet sending and acknowledgment: Data is typically divided into packets for transmission over the network. Each packet incurs overhead in terms of packet header information, error checking codes, and other necessary data. Upon receiving packets, the receiver must acknowledge their successful receipt, which adds further overhead.

These overheads are inherent to network communication and can affect the overall performance and efficiency of data migration. They contribute to increased latency, reduced throughput, and potentially longer migration times. The speed of data migration is affected not only by factors such as network throughput but also by the performance of I/O devices. In particular, random I/O operations, which are often involved in data migration, tend to be slower than sequential write operations. Some systems employ techniques to optimize data migration by converting random I/O operations into sequential writes. However, even with such optimizations, every new data write typically requires an actual write to the storage device to ensure data durability, which involves flushing the page cache to the disk drive.

That being said, the ZeroMove hashing method can also offer advantages to applications that rely on in-memory computing, especially when dealing with large amounts of data

that need to be cached or stored in computer memory. In such scenarios, each node can be seen as a bucket, and the clusters of buckets maintain a fixed size. By gradually adding new clusters of buckets to the data structure, it is possible to achieve similar functionality to a distributed system. This approach can provide scalability and efficient data management in memory-centric applications.

The ZeroMove data hashing and distribution technique is evaluated from the following pertinent perspectives.

- **Scalability** Reducing the amount of data migration during the scaling process of a distributed system can significantly improve the system's scalability by minimizing the disruption and potential risks associated with migrating large amounts of data. In a secure system, multiple replicas are stored on the system to prevent data loss. The benefit of zero data migration is more pronounced when data replication strategies are implemented for improved data availability and reliability.
- **Complexity** Data migration can be a complex and time-consuming process, especially in large distributed database systems. By avoiding data migration, the system is simplified, and the potential for errors and downtime associated with data migration is reduced.
- **Consistency** Data consistency refers to the correctness of one data in relation to another data. It can be a challenge in distributed systems, especially during data migration. By avoiding data migration, the system can potentially maintain better data consistency.
- **Cost** Data migration can be expensive, especially if it is big and requires a lot of resources and time. By avoiding data migration, the system can potentially save on equipment and network costs associated with migration.
- **Balance** The active clusters are composed of computer nodes with newer hardware which tends to be more powerful and have higher capacity, allowing them to handle both read and write workloads more efficiently. This means that newer hardware can process more requests in a shorter amount of time, resulting in improved response times and better load balancing.
- **Performance** In a distributed system that requires data migration during the scaling process, a write and a delete operation are required to perform a data move. This can lead to increased network traffic, disk I/O, and CPU utilization, which can impact the performance of normal data operations. In ZeroMove scaling, the two operations of data move are avoided. It is estimated that the throughput rate can be three times higher during extended periods of scaling.
- **Quality** Spikes in workloads on a system can have a significant impact on the service level agreement (SLA) provided to users. When a system experiences a sudden increase in workload, it may struggle to process all the requests in a timely manner, resulting in longer response times, degraded system performance, or even system failures. To mitigate this impact, it is important to implement effective scaling strategies. ZeroMove strategy can improve the quality of service for data platform operators and can help ensure that users can access the system when they need it, and that they can perform their tasks without interruption.

ZeroMove hashing offers flexibility in adapting to the read-to-write ratios of a production system. If there is a higher proportion of reads compared to writes, the system can have fewer input nodes. Conversely, if there is increased write pressure, more input nodes can be added during the scaling process to handle the higher write load.

Additionally, ZeroMove hashing provides load balancing capabilities by assigning a greater workload to newer and more powerful computers. Recently added clusters are responsible for both read and write operations, allowing them to handle a larger share of the system's load.

The growth rate of computing power, as measured by Input/Output Operations Per Second (IOPS), has seen significant advancements over time. With the advent of Solid State Drives (SSDs) that have no moving parts and faster data access speeds compared to Hard Disk Drives (HDDs), the growth in IOPS has been even more pronounced. In the early 2010s, the typical IOPS for SSDs was around 10,000, which then increased to approximately 100,000-200,000 by the mid-2010s. This represents a growth rate of approximately 10-20 times every five years. Presently, SSDs capable of achieving millions of IOPS are being manufactured, showcasing the continuous progress in computing power and storage technology.

V. Conclusion and Future Work

This article introduces a novel technique for horizontally scaling distributed systems with efficient data distribution. Unlike traditional scaling methods, the proposed technique eliminates the need for data migration during the scaling process, resulting in a faster and more efficient scaling operation. This approach significantly improves the availability and performance of distributed systems.

Our study focuses on the behavior of consistent hashing, a widely used hashing algorithm, and identifies that data migration occurs for almost every piece of data during incremental scaling operations. To address this challenge, we present the protocol and algorithms of the ZeroMove hashing technique, which enables efficient storage, searching, updating, and deleting of data without the need for data migration.

In addition, we provide comprehensive guidelines and mathematical expressions for introducing new clusters to the system during scaling processes. These guidelines ensure that capacity and performance requirements are met while minimizing disruption to the system.

To validate the effectiveness of ZeroMove hashing, we conducted comparative experiments between consistent hashing and ZeroMove hashing methods. The results clearly demonstrate the advantages of ZeroMove hashing in terms of efficiency and cost-effectiveness.

Overall, this paper contributes a valuable technique for scaling distributed systems, offering significant improvements in efficiency and cost-effectiveness compared to traditional methods. The proposed ZeroMove hashing technique has the potential to enhance the performance and scalability of various distributed systems in real-world applications.

In today's world, there is a growing awareness among companies about the importance of reducing their carbon footprint and embracing sustainable business practices. The use of energy contributes to various environmental issues, including carbon dioxide emissions, depletion of the ozone layer, and damage to the Earth's ecosystem. To address these concerns, innovative technologies like ZeroMove can play a crucial role in helping enterprises achieve their carbon neutrality and net zero goals.

ZeroMove technology stands out with its focus on reducing power consumption, making it an ideal solution for environmentally conscious companies. By minimizing energy usage, ZeroMove can significantly contribute to lowering carbon emissions and mitigating the negative impact on the environment. By adopting ZeroMove, businesses can actively contribute to sustainability efforts and align their operations with a greener and more eco-friendly approach.

The implementation of ZeroMove technology empowers enterprises to make substantial strides towards their sustainability goals. By optimizing energy consumption and reducing their carbon footprint, companies can demonstrate their commitment to environmental responsibility and contribute to the global effort to combat climate change. By embracing ZeroMove, businesses not only reap the benefits of improved efficiency and cost savings but also make a positive impact on the environment.

Further research can explore the performance of the system in larger-scale scenarios, encompassing higher data volumes that necessitate migration. Specifically, investigations should focus on sustained write and read operations, as well as multiple scaling processes. Experimental studies can be designed to evaluate the system's efficacy in meeting storage capacity requirements and handling performance demands efficiently.

Conducting a comprehensive analysis of the operational costs and benefits related to the implementation of ZeroMove hashing would be a valuable endeavor. This analysis could encompass various factors, including the impact of wear and tear of computer hardware, power consumption, maintenance requirements, and management overhead. By quantifying these aspects, a cost-effectiveness assessment can be performed to evaluate the economic advantages of adopting ZeroMove hashing compared to alternative scaling approaches. This analysis will provide valuable insights into the financial implications and help organizations make informed decisions regarding the adoption of ZeroMove hashing in their distributed systems.

ACKOWLEDGMENTS

I would like to express my sincere appreciation to Wenguang Wang, VMWare, Inc., for taking the time to read and review my paper. I am truly grateful for the insightful questions and valuable suggestions provided. His engagement and willingness to delve into the problem at hand were truly inspiring and shed light on important aspects that may have been overlooked. His input has been instrumental in enhancing the quality and depth of the work.

REFERENCES

[1] C. Roy, et al., "Horizontal Scaling Enhancement for Optimized Big Data Processing", in Proc. of IEMIS 2018, vol. 1, Jan. 2019, pp. 639-649

[2] D. Karger, et al., "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Horizontal Scaling Enhancement for Optimized Big Data Processing", in Proc. of Twenty-Ninth Ann. ACM Symp. on Theory of Computing, pp. 654-663

[3] J. P. Carzolio, "A Guide to Consistent Hashing", [Online], Available: *https://www.toptal.com/big-data/consistent-hashing*

[4] D. Featherston, "Cassandra: Principles and Application", [Online], Available: *http://disi.unitn.it/ montreso/ds/papers/Cassandra.pdf*,

[5] S. Sankarapandi, et al., "Storing of Unstructured data into MongoDB using Consistent Hashing Algorithm", *Int. J. of Emerg. Technol. in Eng. Research (IJETER) December 2015, Volume 3, Issue 3*

[6] G. DeCandia, et al., "Dynamo: Amazon Highly Available Key-value Store", [Online], Available: *https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf*

[7] D. Thaler, et al., "Using Name-Based Mapping Schemes to Increase Hit Rates", *in IEEE/ACM Transactions on Networking, Februray 1998, Volume 6, Issue 1*

[8] J. Lamping, E. Veach, "A Fast, Minimal Memory, Consistent Hash Algorithm", [Online], Available: *https://arxiv.org/pdf/1406.2294*

[9] Y. Xiong, et al., "ECCH: Erasure Coded Consistent Hashing for Distributed Storage Systems", [Online], Available: *http://www.cloud-conf.net/264600a177.pdf*

[10] W. Xie and Y. Chen, "Elastic Consistent Hashing for Distributed Storage Systems", [Online], Available: *https://ieeexplore.ieee.org/document/7967178*

[11] D. Earp, "Heuristics in Distributing Data and Parity with Distributed Hash Tables", [Online], Available: *https://etd.auburn.edu/handle/10415/8001*

[12] J. Noor, et al. "Portkey: Adaptive Key-Value Placement over Dynamic Edge Networks", in ACM Symposium on Cloud Computing, November 1-4, 2021, Seattle, WA, USA.

[13] A. Haeberlen, et al. "Consistent Key Mapping in Structured Overlays", Dept. of Computer Science, Rice Univ., Tech. Rep. TR05-456, Auguest 2005.

[14] B. Dageville, et al. "The Snowflake Elastic Data Warehouse", [Online], Available: *https://event.cwi.nl/lsde/papers/p215-dageville-snowflake.pdf*

[15] JaguarDB, NoSQL Database Software, [Online], Available: *https://github.com/fserv/jaguardb*,

BIOGRAPHY

Jonathan Z. Yue is interested in distributed systems, distributed databases, decentralized systems, and distributed storage systems. He founded the open-source project JaguarDB with the aim of revolutionizing the way distributed databases are scaled. Recognizing the challenges associated with traditional data migration methods, he pioneered the concept of zero data migration, enabling seamless horizontal scaling without the need for costly and time-consuming migrations. He is also the founder of the Omicro project, a decentralized database that ensures fault tolerance in the face of malicious nodes or data corruption. Leveraging his expertise in consensus algorithms, he devised a unique approach where only three messages are exchanged per node, providing robust Byzantine fault tolerance while minimizing network overhead. He holds PhD degree in computer science of EECS. He worked as a software engineer at technology companies such as Yahoo and VMWare, where he tackled big data challenges and delivered cutting-edge cloud solutions. His experience at these industry-leading companies equipped him with invaluable insights into the complexities of distributed systems and the scalability demands of modern data-driven applications. This hands-on experience in real-world environments served as a catalyst for his passion to create open-source projects aimed at revolutionizing the way databases are scaled and decentralized.