IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 27 October 2024, accepted 20 November 2024, date of publication 27 November 2024, date of current version 9 December 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3507194

RESEARCH ARTICLE

PVA: The Persistent Volume Autoscaler for Stateful Applications in Kubernetes

JI-HYUN NA[1], HYEON-JIN YU[1], HYEONGBIN KANG[1], HEEJU KANG[1], HEE-DONG LIM[2], JAE-HYUCK SHIN[1], AND SEO-YOUNG NOH[1]

[1]Department of Computer Science, Chungbuk National University, Cheongju 28644, South Korea
[2]Department of Computer Engineering, Chungbuk National University, Cheongju 28644, South Korea

Corresponding author: Seo-Young Noh (rsyoung@cbnu.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) grant funded by Korean Government [Ministry of Science and ICT (MSIT)] under Grant NRF-2008-00458.

ABSTRACT With the evolution of container technology, there has been increasing interest in the operation of stateful applications in cloud environments. Kubernetes supports the management and scaling of stateful applications using features, such as StatefulSets and autoscaling mechanisms. However, current autoscaling targets CPU and memory without supporting volumes, which are essential resources for stateful applications. This can result in service downtime and volume resource waste. In response to these challenges, this study proposes a Persistent Volume Autoscaler (PVA) that automatically manages the volume resources in Kubernetes. PVA automatically adjusts the various components required for stateful applications to use volumes. It offers autoscaling that supports both the scale-up and scale-down of the volumes. We designed and implemented algorithms to achieve this, and evaluated the performance of the PVA across diverse storage solutions. Our results demonstrated that PVA improves the availability and efficiency of stateful applications. Additionally, we found that PVA performs effectively across various environments, including Ceph, Longhorn, and Synology. This demonstrates its capability to improve the flexibility and scalability of volume resources. By optimizing the operation of stateful applications, PVA contributes to enhanced efficiency and improved service availability.

INDEX TERMS Autoscaling, stateful application, persistent volume, kubernetes, volume management.

I. INTRODUCTION

Recently, various services have transitioned to cloud environments, leveraging the advantages of flexibility and cost-efficiency. Containers are now widely adopted in the cloud, owing to their high portability and flexibility. In particular, Kubernetes has become the de facto standard in cloud environments, offering robust management for the deployment and scaling of containerized services [1].

Containers are inherently portable and flexible, which has led most container orchestrators to primarily focus on stateless applications. Each instance of a stateless application operates independently, and is deployed interchangeably. Because stateless applications do not store state information internally, there is no need to persistently store state information within the application. Instead, they retrieve necessary data from external storage systems or services when needed. Any temporary state data are discarded when an instance is terminated. Therefore, stateless applications do not need to consider the preservation of the state data. This attribute enables container orchestrators to rapidly replace faulty containers by using replication strategies, thereby maintaining continuous service operations. By contrast, stateful applications require each instance to maintain a unique state, making them non-interchangeable. Each instance stores the necessary state data in a volume and operates based on these data. As the types and complexities of services have increased and the use of containers has expanded in recent years, various container orchestrators have begun offering management features for stateful applications.

The associate editor coordinating the review of this manuscript and approving it for publication was Hang Shen.

StatefulSet is a workload API object in Kubernetes that ensures stable deployment and scaling of stateful applications. Kubernetes uses StatefulSets to manage pod replicas, and in the event of pod failure, it redeploys a new replica with the same name to ensure service continuity and data consistency. In addition, Kubernetes supports autoscaling to automatically adjust the number of pods according to resource demand, thereby enhancing service resilience and availability. Because of this support, there has been a growing demand for operating stateful applications in Kubernetes, and most enterprises are striving to integrate various stateful applications, such as databases, messaging, and streaming services, into the Kubernetes environment [2], [3].

However, the default autoscaling features provided by Kubernetes only target CPU and memory, and most existing research on autoscaling follows a similar approach. Stateful applications operate primarily based on volumes, which are persistent storage resources that store state data. These volumes ensure that the application's data are preserved during restarts or failure recovery scenarios. In Kubernetes, abstract components are provided to enable applications to utilize volumes effectively. By employing these components, stateful applications can request storage allocation and utilize volumes to store essential operational data. Consequently, it is essential to manage volume utilization, which reflects the ratio between the size of the allocated volume and the size of the stored data, requiring flexible volume management. If volume management for stateful applications is not handled properly, it can lead to significant problems such as service interruptions, waste of volume resources or state data loss [4].

Therefore, this paper proposes a Persistent Volume Autoscaler (PVA), an autoscaler that supports both scaling up and scaling down of volumes. The proposed PVA monitors the utilization of volumes, which are essential resources in stateful applications, and can dynamically scale volumes without administrator intervention. This capability can significantly enhance the availability of stateful applications, optimize the utilization of volume resources, improve service operation efficiency, and help prevent data loss. In addition, we implement the proposed PVA and conduct experiments to verify the effectiveness of its architecture and algorithms. Through these evaluations, we demonstrate that PVA enables services to operate in Kubernetes environments that are flexible, highly available, and efficient.

The remainder of this paper is organized as follows. Section II provides background information, including existing autoscaling mechanisms, volume usage in Kubernetes, and the deployment structure of stateful applications. In Section III, we highlight the limitations and problems of current volume scaling practices and explain the specific problems that PVA addresses in volume scaling. Section IV describes the cluster architecture and component workflow required for PVA. Section V details the algorithms and implementation methods for scaling volumes using PVA. Sections VI and VII cover the experimental design, implementation, and performance evaluation of PVA. Finally, Section VIII summarizes the content and discusses future research directions.

II. BACKGROUND

A. AUTOSCALING MECHANISMS

As the variety and scale of services have recently increased, the requirements for applications have rapidly evolved. In such environments, resource demands can fluctuate significantly depending on time or conditions, making manual management a time-consuming and challenging task. Moreover, improper management of these resources can lead to waste and increased costs. Autoscaling has become an essential technology for addressing these problems.

Two main strategies are commonly used in autoscaling mechanisms: scaling in/out, and scaling up/down. Scaling in/out refers to horizontal scaling that entails the addition or removal of certain instances. Kubernetes natively supports the Horizontal Pod Autoscaler (HPA) to scale computing resources horizontally. HPA adjusts the number of pod replicas to maintain the desired state. On the other hand, scaling up/down refers to vertical scaling, which involves increasing or decreasing resource allocation for a single instance. Vertical resource scaling in Kubernetes can be performed using a Vertical Pod Autoscaler (VPA). VPA is not included by default in Kubernetes, and must be installed separately [5]. Previously, changing resource sizes with VPA required restarting pods. However, a recent update allows vertical scaling without restarting. This feature is currently in the alpha stage and has not fully developed [6].

The primary target resources for HPA and VPA are CPU and memory. To manage application performance more precisely, Kubernetes allows defining custom metrics based on specific performance indicators, which can then be applied to an autoscaler [7]. Using these custom metrics, the autoscaler can configure metrics related to volume utilization and perform scaling based on them. However, they only serve as triggers for scaling and do not support direct scaling of the volume resources themselves. In other words, the autoscalers provided by Kubernetes do not directly scale volumes.

To increase the volume resources, HPA can be used to increase the number of pods when the volume utilization exceeds a certain threshold, subsequently creating additional volumes to store the extra data. However, if the volume usage drops below a certain threshold, the associated pods may be deleted; because of the nature of HPA and volumes in Kubernetes, the volumes themselves are not deleted or reduced in size. This means that overprovisioned volumes can retain idle resources, leading to waste. Therefore, current autoscalers cannot ensure flexibility in resource volume.

B. KUBERNETES VOLUME

Kubernetes provide an abstraction concept for the use of physical storage systems. Figure 1 illustrates the components involved in deploying stateful applications in Kubernetes, and the process of assigning and using volumes. A physical

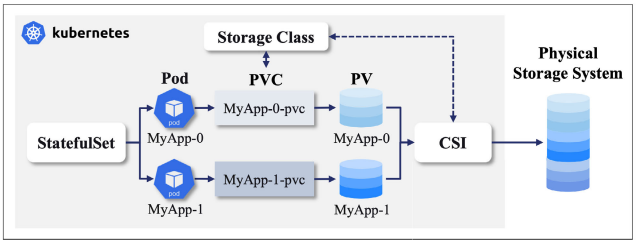


FIGURE 1. Volume allocation process.

storage system that stores data is configured as the backend. Storage Class (SC) is a resource that defines the details of such physical storage, allowing dynamic provisioning of storage according to user requests [8].

Various SCs can exist depending on the storage provider or user requirements, and they request provisioning by connecting to the storage system through Container Storage Interface (CSI) drivers. CSI is a plugin used to expose storage systems to containers, and when deployed in a Kubernetes cluster, it allows the mounting of the exposed storage [9]. Persistent Volume (PV) refers to abstracted storage resource [10]. A Persistent Volume Claim (PVC) defines the SC, requested size, access modes, and other settings and requests a PV based on these. Kubernetes searches among the available PVs for one that meets the defined requirements and binds the PV and PVC. Through the cooperation of these components, pods can use these volumes.

C. STATEFUL APPLICATION

Kubernetes supports various controllers for managing application workload. The controllers manage pods to maintain the desired application state. Containers have traditionally evolved based on their stateless characteristics. Kubernetes supports several controllers for managing stateless applications, including Deployment, ReplicaSet, and CronJob. However, as the demand to operate various services in containers has increased, Kubernetes has begun to support stateful applications through a controller called StatefulSet.

Stateful applications refer to workloads in which each pod must maintain its own unique state. Examples include databases (e.g., MySQL, MongoDB, and Cassandra), distributed coordination services (e.g., Zookeeper), and stream processing frameworks (e.g., Kafka Streams) [11], [12], [13]. The state data that must be maintained varies depending on the application. For databases such as MySQL, the state typically includes not only the data stored in the database itself (e.g., tables, records, and indices), but also configuration information and credentials for the MySQL servers [14]. Zookeeper maintains Write-Ahead Logs (WALs) and snapshots of its data, which are essential for fault recovery [15]. On the other hand, Kafka Streams retains the state required for operations such as aggregation, joining, and windowing in stream processing [16]. These state data are unique to each pod, and it is essential to preserve them persistently [17].

Figure 2 shows the difference between deploying stateless and stateful applications. Deployment is a basic controller for deploying stateless applications. When a pod template is defined by containers and volumes to deploy pods, each pod is assigned a random name and shares a single volume. If a pod fails, the deployment creates a new pod with a random name for recovery. In contrast, StatefulSet defines containers in the pod template and the PVCs in the volume claim template. Through this, the deployed pods have sequential names and use different volumes. StatefulSet creates and deletes pods in order based on their sequential names. If a pod fails, it creates a new pod with the same name to access the previous volume.

If a stateful application is deployed using a Deployment, the state data of each pod cannot be guaranteed [18]. For example, if a pod named 'MyApp-xyz' fails, Deployment creates another pod with a random name like 'MyApp-qwe' and connects it to the shared volume. The new pod with the changed name does not have information about the failed pod 'MyApp-xyz' and cannot access its state data, which can cause data consistency problems. Additionally, sharing the same volume makes it difficult for pods to maintain independent state data.

Conversely, StatefulSet creates a new pod with the same name as the failed pod, and connects it to the volume where the previous state data were stored. This ensures data consistency and persistence for stateful applications that must maintain unique state data. Because stateful applications need to maintain the state of each pod and continuously store and access data, the volume storing this data is crucial to operations [19]. Therefore, this study proposes an autoscaler for the volumes of stateful applications, in which stored data is a critical factor.

III. PROBLEM STATEMENT/RELATED WORK

In this section, we describe in detail the challenges associated with volume scaling in Kubernetes that the proposed PVA aims to address. By highlighting the limitations and challenges of existing research and technologies, we emphasize the necessity and significance of our proposed solution.

Volume scaling in Kubernetes requires administrators to monitor usage and manually take action, depending on the situation. Improper management can lead to challenges, such as service disruptions, inefficient costs and data loss. For instance, if volume usage increases rapidly in real-time and the volume is not scaled accordingly, the service relying on that volume may be disrupted, thereby affecting availability. Alternatively, allocating significantly larger volumes than needed to prevent disruption often results in resource wastage and unnecessary cost. Recently, there has been growing interest in managing cloud waste caused by over-provisioning [20]. Therefore, managing unnecessary volume resources to reduce this waste is crucial, as it can improve resource utilization efficiency and contribute to cost savings.

Moreover, the manual process of volume decrease and data migration carries the risk of data loss owing to human

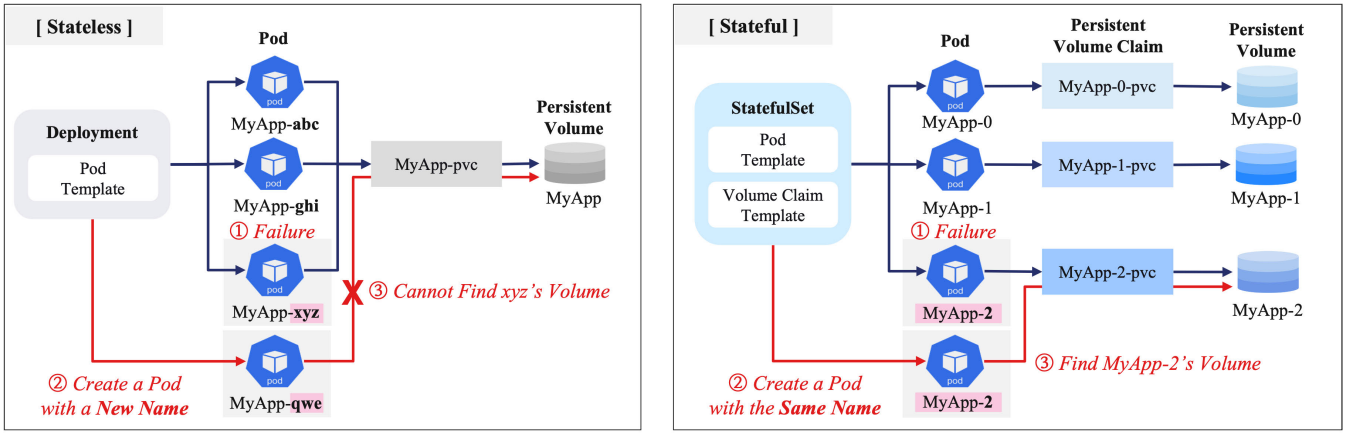


FIGURE 2. Differences in deploying stateless and stateful applications.

error. Automating volume scaling can reduce administrative efforts and make resource use more efficient. This helps ensure service availability and data consistency, ultimately contributing to more stable system operations. Therefore, a volume autoscaler can be an effective solution to enhance resource flexibility and reduce the risk of data loss.

A. TARGET RESOURCES

As mentioned in Section II, Kubernetes autoscalers, such as HPA and VPA, primarily scale CPU and memory resources, whereas volume-related factors function merely as triggers. In addition to these default autoscalers, various other autoscalers have been further researched, and comprehensive surveys have evaluated them. Castillo et al. [21] analyzed several existing autoscaler cases by categorizing the key points of elasticity in cloud environments, such as scaling timing, resource estimation, scaling metrics, scaling methods, and adaptivity.

Tran et al. [22] classified autoscalers in Kubernetes based on indicators, such as application architectures, methods, timing, and indicators, thereby analyzing autoscaling technologies. However, many of the autoscalers presented in these surveys primarily focused on CPU and memory resources. Although some also consider metrics such as network I/O, bandwidth, traffic, request-response latency, and service level agreements (SLAs), they are still used merely as triggers or tools to optimize scaling plans.

Moreover, although several studies have addressed volumes and stateful applications, these studies do not specifically focus on volume scaling directly targeting persistent volumes in stateful applications. For example, Shemyakinskaya et al. [23] discussed disk management automation using CSI, highlighting that although bare-metal CSI offers strong reliability, it lacks support for ephemeral volumes. However, this proposal is limited to a specific CSI and includes only the deletion of ephemeral volumes during pod deletion. Thus, the challenges of comprehensive volume management in Kubernetes remain unresolved. Similarly, Perera et al. [24] highlighted the limitations of autoscaling for database and proposed an algorithm considering database requests. However, this solution is also confined to PostgreSQL and solely suggests a method for pod scaling, without considering volume scaling.

In summary, although several studies have explored scaling mechanisms, volume scaling—crucial for managing stateful applications—has been overlooked. As mentioned in Section II-C, stateful applications operate primarily based on volumes. Accordingly, PVA was designed to perform autoscaling, specifically targeting volumes. This approach helps to manage proper volume allocation, prevent service interruptions, and avoid idle resources.

B. VOLUME SCALE-UP AND SCALE-DOWN

Autoscaling strategies typically include horizontal (scaling in/out) and vertical (scaling up/down) scaling. For stateful applications, maintaining data synchronization and consistency is crucial. When using horizontal scaling to create or delete pod replicas, a separate strategy is required for data replication or merging, owing to the data distribution. Moreover, horizontal expansion involves additional management of multiple volumes and pods, which can be burdensome. Therefore, this study considered vertical scaling, which is simpler and more straightforward from the management perspective.

Currently, several solutions provide automatic volume-expansion functionality, with some variations. These variations reflect different approaches to handling volume scaling, based on the specific needs and environments for which they are designed. Amazon Aurora and Microsoft Azure support horizontal scaling [25], [26]. Amazon RDS and DevOps-Nirvana's autoscaler support vertical scaling [27], [28]. Konev et al. [29] proposed an algorithm that expands volumes automatically in Kubernetes by combining HPA and VPA and using parameters, such as expansion coefficients.

However, these studies focused solely on volume expansion, without addressing volume reduction, and no clear

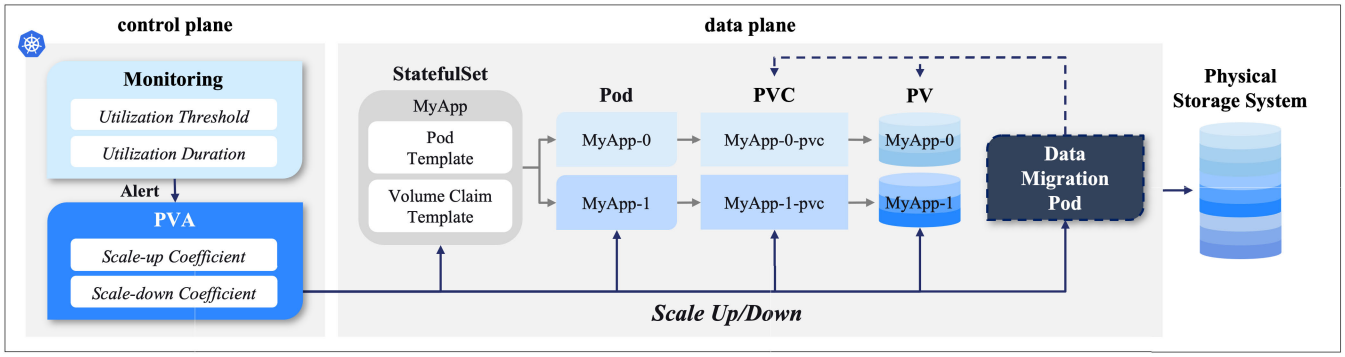


FIGURE 3. Architecture and workflow of PVA.

performance analyses have been conducted. References [25], [26], [27], and [28] did not evaluate the time and potential system load involved in the expansion. Reference [29] only compared the time taken for manual and automatic expansion in two cases, without addressing system load. Consequently, there is no research on automatic volume reduction, and existing volume expansion methods lack comprehensive performance evaluations.

Additionally, these solutions rely on products from specific cloud service providers. Therefore, these functions are often unavailable when building infrastructures for stateful applications in private environments. Thus, there is a need for a method that is independent of the products and environments and can be applied to various stateful applications in real-world operations.

Ultimately, this study proposes a persistent volume autoscaler that integrates both scale-up and scale-down functionalities using a vertical scaling approach, evaluating the performance of each function to assess its system impact. Additionally, by making the autoscaler applicable to all stateful applications in Kubernetes, it ensures broad applicability and flexibility across different environments, without being restricted to specific products.

IV. PVA ARCHITECTURE AND COMPONENTS

Scaling a pod's volume in Kubernetes requires comprehensive consideration of the entire volume management process. This is not merely about physically resizing the storage system, but also requires considering all components for stateful applications, including StatefulSets, PVCs, PVs, SCs, and CSI drivers that enable PVs to access the storage system. To accomplish this, PVA is deployed as a pod within the Kubernetes cluster, enabling it to interact with all the components of stateful applications. This design ensures that volume autoscaling is not limited to specific services or storage providers and can be applied to any stateful application running in Kubernetes.

Figure 3 illustrates the architecture and workflow of PVA. The control plane of the cluster consisted of monitoring the pods and the PVA pod. The data plane included instances of stateful applications and pods for data migration. Initially, StatefulSet, which is a controller managing stateful applications, was deployed. Pods, PVCs, and PVs were generated based on the defined pod and volume claim templates. The data migration pod is temporarily created when the PVA performs a scale-down operation, and is automatically deleted once the operation is completed. This pod interacts with the PVC and PV to perform data synchronization and migration.

A. MONITORING

PVA collaborates with components such as Kubelet, Prometheus, AlertManager and Grafana to monitor the volume states and trigger scaling actions. Kubelet runs on all the nodes within a cluster and periodically reports the status of the nodes and pods to the API server. During this process, Kubelet collects metrics and exposes them to external endpoints [30]. For monitoring, the PVA uses metrics related to volume utilization (e.g., kubelet_volume_stats_capacity_bytes and kubelet_volume_stats_used_bytes).

Kubernetes natively uses Prometheus to periodically collect and monitor metrics from the endpoints [31]. Prometheus queries the collected data, defines alert rules to trigger warnings when specific conditions are met, and visualizes the data in real-time dashboards through Grafana [32], [33]. Using the collected data, PVA monitors and calculates the utilization using the allocated volume size and current usage. If the utilization exceeds a certain threshold (for scale-up) or falls below it (for scale-down), and this state persists for a specific duration, an alert is fired. Subsequently, AlertManager receives these alerts from Prometheus and forwards them to the PVA through webhooks.

B. VOLUME SCALING

When the PVA receives alerts for scale-up or scale-down by monitoring pods, it triggers scaling operations. The PVA uses the scale-up and scale-down coefficients set by the user to calculate the final volume size based on the initial volume size. For example, if the scale-up coefficient is 1.5, and the current volume is 10GB, the new volume

size will be set to 15GB for scale-up. After determining the new volume size, PVA performs a series of scaling operations on all components related to the volume. PVA interacts with StatefulSets, Pods, PVCs, and PVs, modifying the information associated with the current volume to the new desired volume size, and ensuring that the service continues to operate smoothly with the updated volume size in Kubernetes.

V. SCALING ALGORITHMS AND IMPLEMENTATION

In Kubernetes, volumes typically require manual control by administrators. Therefore, to monitor volumes and automate scale-up and scale-down, PVA must be implemented with several tasks related to the instances of stateful applications. We used the Python client library for Kubernetes to implement these functionalities [34]. This library enables PVA to perform the authentication process and a series of commands for collaboration with internal cluster components.

A. SCALE-UP ALGORITHM

Kubernetes provides the ability to expand volumes allocated to a PVC online, without service downtime, by modifying the volume size to a larger size when using a CSI driver [10]. However, this feature is limited to the manual operations performed by an administrator. To automate the expansion process, PVA's scale-up was designed to extend the volume without service disruption.

Algorithm 1 illustrates the scale-up process of PVA. When the volume usage exceeds a certain threshold and persists beyond a specified duration, an alert is triggered based on the alerting rules of an external monitoring module and a webhook sends a notification to PVA. Upon receiving this trigger, PVA begins the scale-up process.

PVA first queries metrics from Prometheus to acquire information, such as names, namespaces, and volume usage of the relevant pods, PVCs, and PVs involved in the expansion. Based on this information, PVA calculates the new volume size by multiplying the current volume capacity by the scale-up coefficient. It then updates the PVC information to request storage with the expanded size, thereby performing volume expansion. Subsequently, the configuration of StatefulSet is retrieved, and the Volume Claim Template is modified to a new size. To apply these changes, StatefulSet is deleted and redeployed using the orphan option. The orphan option allows StatefulSet to be re-executed while keeping the existing pods intact. This approach expands the volume capacity while maintaining the service running with modified size in Kubernetes.

B. SCALE-DOWN ALGORITHM

Unlike Kubernetes and existing storage solutions that support online expansion, reducing PVC to a smaller size is still not supported [10]. Therefore, the scale-down process is more complex than that of scale-up, because PVA automates the creation of a new volume, data migration, and connection to a new StatefulSet to restart the service.

The scale-down of PVA considers the following:

- Data migration takes the most time.
- Data can change while the pod is running.

Data migration typically takes longer than modifying Kubernetes components, such as volumes, pods, and StatefulSets, to reflect changes, particularly when the volume contains a large amount of data. In addition, during data migration while the service is running, data may be added, deleted, or updated at any time. However, unlike scale-up, scale-down does not usually require significant modifications to the stored data while the service is running.

Therefore, PVA automates data migration in two phases: pre-copy and final-copy. In the pre-copy phase, all data are replicated while the service runs, serving as a live data migration. As changes in the stored data may occur during this phase, the final-copy phase stops the service to replicate only the data modified during the pre-copy.

PVA uses rsync for these replication tasks [35]. Rsync is a utility optimized for data synchronization, allowing for copying either the entire dataset or only the modified parts. It is also used for live migration, and its compression features enable faster data transfer and reduce system load [36]. By utilizing Rsync, these two phases minimize service downtime during migration, ensuring data consistency and up-to-date replication while the service remains active.

Algorithm 1 PVA Scale-Up

Define: Pod a $\boldsymbol{P_a}$, PVC of Pod a $\boldsymbol{PVC_a}$, StatefulSet of Pod a $\boldsymbol{STS_a}$, volume size of PVC a $\boldsymbol{VS_a}$, Scale-up coefficient $\boldsymbol{C_{up}}$

while *Scale-up Alert Fired for* $\boldsymbol{P_a}$ **do**
- *// Get Information*
- get_alert_info ($\boldsymbol{P_a}$, $\boldsymbol{PVC_a}$, $\boldsymbol{STS_a}$)
- $\boldsymbol{VS_{new}} \leftarrow \boldsymbol{VS_a} * \boldsymbol{C_{up}}$
- *// Modify PVC*
- patch_pvc($\boldsymbol{PVC_a}$, $\boldsymbol{VS_{new}}$)
- *// Modify StatefulSet*
- old_STS_conf ← get_conf($\boldsymbol{STS_a}$)
- new_STS_conf ← modify(old_STS_conf, $\boldsymbol{VS_{new}}$)
- *// Restart StatefulSet*
- delete_STS_orphan($\boldsymbol{STS_a}$)
- create_STS(new_STS_conf)

end

The algorithm for implementing this scale-down process is presented in Algorithm 2. First, when volume usage falls below the threshold and persists for a specified duration, PVA receives relevant alerts and information from the monitoring module. Based on this, the size of the volume to be decreased is calculated using the scale-down coefficient. Subsequently, based on the configuration of the old PVC, a new volume is created for data migration.

Algorithm 2 PVA Scale-Down

Define: Pod a P_a, PVC of Pod a PVC_a, StatefulSet of Pod a STS_a, volume size of PVC a VS_a, Scale-down coefficient C_{down}

while *Scale-down Alert Fired for* P_a **do**
- *// Get Information*
- get_alert_info (P_a, PVC_a, STS_a)
- $VS_{new} \leftarrow VS_a * C_{down}$
- *// Create New PVC*
- old_PVC_conf ← get_conf(PVC_a)
- new_PVC_conf ← modify(old_PVC_conf, VS_{new})
- create_PVC(new_PVC_conf, PVC_{new})
- *// Data Migration(Pre-Copy)*
- create_pod($P_{migration}$)
- execute_pre_copy(PVC_a, PVC_{new})
- *// Modify StatefulSet*
- old_STS_conf ← get_conf(STS_a)
- new_STS_conf ← modify(old_STS_conf, VS_{new})
- wait_pre_copy($P_{migration}$)
- *// Data Migration(Final-Copy), Service Down*
- delete_STS_and_pod(STS_a, P_a)
- execute_final_copy(PVC_a, PVC_{new})
- wait_final_copy($P_{migration}$)
- *// Restart Service*
- create_STS(STS_conf, STS_{new}, P_{new})
- *// Clean Up*
- delete_pod($P_{migration}$)
- delete_PVC(PVC_a)

end

Subsequently, a pod containing the rsync container image is created to perform data migration, which is executed to replicate the data of the old volume to the new volume. Concurrently, the volume claim template information of StatefulSet is modified to enable the use of the new PVC size. After the pre-copy is completed, StatefulSet and running pods are deleted to stop the service. A final-copy task is performed to replicate the changed data only. Once all data are updated, the service is restarted based on the modified StatefulSet configuration.

Finally, after confirming that the service has been deployed successfully, the data migration pod, old PVC, and PV are deleted to complete the scale-down process. This process minimizes service downtime and prevents waste from unused volumes as the usage demand decreases.

VI. EXPERIMENTAL DESIGN

Experiments were conducted to evaluate the performance of PVA across various storage solutions in different environments. Ceph, Longhorn, and Synology were selected for their distinct characteristics. Ceph, a distributed file system,

TABLE 1. Server specifications.

Name	Node1	Node2	Node3	NAS
Plane	Control, Data	Data	Data	-
OS	CentOS 7.9	CentOS 7.9	CentOS 7.9	DSM 7.0.1
CPU	8 cores	8 cores	8 cores	4 cores
Memory	32G*4	32G*4	16G*8	4G*1
Disk	890G(SSD) 2.2TB*3(HDD)	890G(SSD) 1.8TB*3(HDD)	890G(SSD) 1.8TB*3(HDD)	8TB*8(HDD)

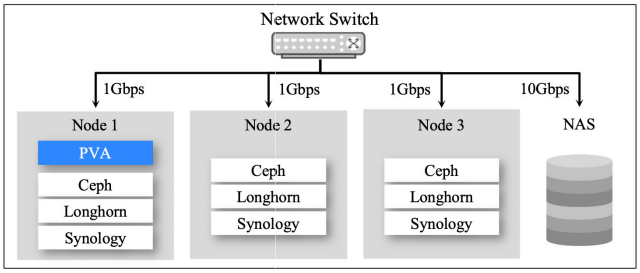


FIGURE 4. Experimental network and node configuration.

emphasizes reliability and scalability [37], while Longhorn is a lightweight, cloud-native distributed block storage solution for Kubernetes [38]. Both were configured as local disks on the nodes. Synology RS1221+, a network-attached storage (NAS) solution, was used to assess network-based remote storage performance. PVA was applied to these storage solutions, and its performance was measured during scale-up and scale-down operations.

The objective was to verify PVA's implementation and assess its scalability and adaptability across diverse environments. Rather than focusing on the performance of specific storage solutions, the study aimed to identify common trends and ensure consistent PVA performance across varying workload scenarios, ranging from light to heavy resource utilization. The experiment also evaluated PVA's flexibility and efficiency under these varying conditions.

Table 1 specifies the server specifications used in the experiments. Figure 4 provides an overview of the network and node configurations in the experimental environment. In the experimental environment, Node 1 served as the control plane, hosting the PVA pod, while also acting as part of the data plane along with Nodes 2 and 3. All three nodes were configured to run CSI pods for the storage solutions, including Ceph, Longhorn, and Synology. Based on the test case scenarios, Kubernetes scheduled application pods on specific nodes, and the CSI pods on those nodes managed the volumes required by the applications. These nodes operated on Kubernetes version 1.28.

The test scenarios for the experiments were as follows. To trigger scaling of PVA, the utilization threshold was set to 70% for scale-up and 30% for scale-down, with a coefficient of 1.5 for scale-up and 0.5 for scale-down. Each test case assessed the impact on performance, including the

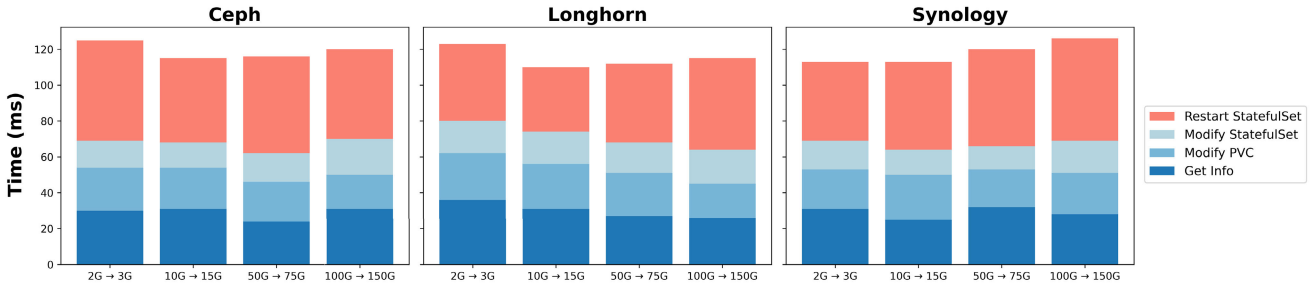


FIGURE 5. Scale-up execution time.

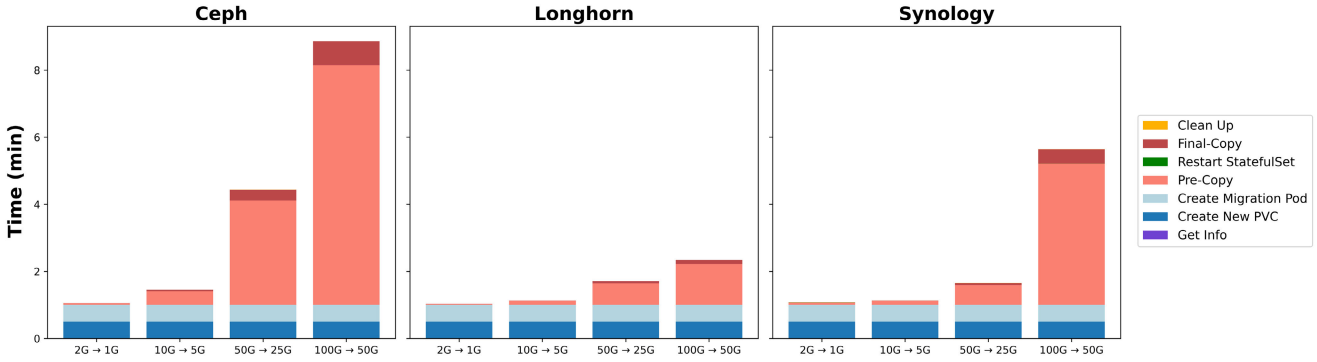


FIGURE 6. Scale-down execution time.

time required for scaling and resource usage of the CPU, memory, and disk.

A. SCALE-UP TEST CASES

- Conducted on stateful applications with initial volume sizes of 2G, 10G, 50G, and 100G.
- Test data of 1.6G, 8G, 40G and 80G were stored in the initial volumes to reach a volume utilization rate of approximately 80%.
- A scale-up coefficient of 1.5 was applied, increasing the initial volume sizes to final sizes of 3G, 15G, 75G and 150G across the storage solutions.

B. SCALE-DOWN TEST CASES

- Conducted on stateful applications with initial volume sizes of 2G, 10G, 50G, and 100G.
- Test data of 0.4G, 2G, 10G and 20G were stored in the initial volumes to reach a volume utilization rate of approximately 20%.
- A scale-down coefficient of 0.5 was applied, decreasing the initial volume sizes to final sizes of 1G, 5G, 25G, and 50G across the storage solutions.
- During the pre-copy phase, an additional 10% of the stored data were inserted to simulate data changes during service execution.

VII. EXPERIMENTS AND PERFORMANCE ANALYSIS

A. AUTOSCALING EXECUTION TIME

Figures 5 and 6 illustrate the execution times to perform scale-up and scale-down operations using PVA on different storage solutions. The graphs represent the time required for scaling with varying volume sizes for each storage solution, which are displayed as stacked bar charts. Each color in the chart indicates the time consumed in different steps of the scaling algorithm.

1) SCALE-UP EXECUTION TIME

As shown in Figure 5, the scale-up execution time across all cases was measured in milliseconds. Because the scale-up operation involves only modifying and applying the configurations of PVC and StatefulSet without any data migration, there is no correlation between the stored data size and the time required for scale-up.

Additionally, there was no significant difference in the scale-up times across different storage solutions. All the solutions exhibited similar scale-up times. Ceph tended to take slightly longer than the other two solutions for all volume sizes, whereas Longhorn and Synology showed slightly faster performance compared to Ceph. The scale-up from 10GB to 15GB with Longhorn was the fastest at 110 milliseconds, whereas the scale-up from 100GB to 150GB with Synology was the longest at 126 milliseconds. Across all cases, the difference between the maximum and minimum times was only 16 milliseconds indicating a minimal variance in performance. The average scale-up times for each storage solution were 119 milliseconds for Ceph, 115 milliseconds for Longhorn, and 118 milliseconds for Synology. Since these values were all in milliseconds, the differences are minimal and unlikely to impact real-world service environments.

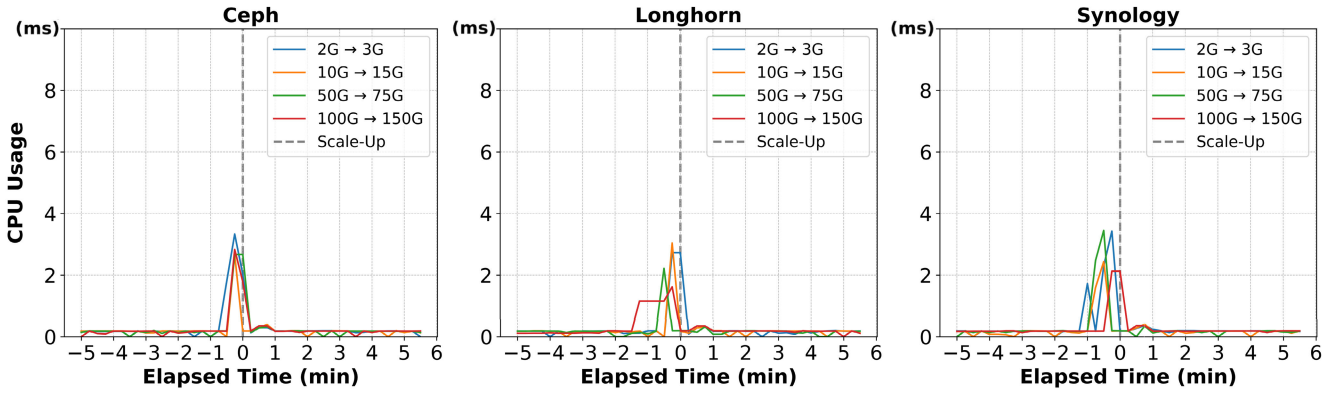


FIGURE 7. CPU usage in scale-up.

For all storage solutions, the "Restart StatefulSet" step, which involves deleting and restarting StatefulSet to apply modifications, accounts for the largest portion of the expansion operation. This step is necessary to ensure that the updated configurations are correctly applied to StatefulSet. However, although this step requires the longest time relative to the others, the difference is measured in milliseconds and is imperceptible during actual service operations. The scale-up process of PVA consistently takes about 0.1 seconds, regardless of the storage solution or data size. Because this operation is performed seamlessly in the background without causing service downtime, it does not have a substantial impact on the overall service performance.

2) SCALE-DOWN EXECUTION TIME

In Figure 6, the scale-down execution time was measured in minutes, showing a tendency to increase as the size of the data increases. Smaller volumes generally required a shorter scaling time. Scaling down from the largest size, 100GB to 50GB, took the longest time. In this scenario, Longhorn completed this in approximately 2 minutes, which is the shortest among the three solutions, whereas Ceph required approximately 9 minutes, which is approximately four times longer.

Ceph employs an architecture and mechanisms that prioritizes data reliability, focusing on fault tolerance, data integrity, and high availability [39], [40]. More complex mechanisms contribute to significant overhead as the data size increases. On the other hand, Longhorn is designed as a Kubernetes-native storage solution with a simple, lightweight engine [41]. Consequently, it demonstrated the smallest increase in time required owing to its optimized performance, ultimately showing the shortest scale-down time. This indicates that the scale-down execution time can vary depending on the storage solution performance.

Excluding the data migration step, all the other steps showed similar execution times across all cases, regardless of the data size. Both the creation of a new PVC and the migration pods each took about 30 seconds on average. Restarting StatefulSet to apply the changed size took an average of 0.03 seconds, and the cleanup took approximately 0.09 seconds. These two steps took such a short time relative to the overall scale-down, which are nearly imperceptible in Figure 6.

The time required for data migration during the pre-copy and final-copy phases showed a proportional increase based on the size of the data. During the scale-down operation, service downtime begins when StatefulSet is deleted and continues until the final-copy is completed, and StatefulSet is redeployed. Therefore, service downtime is determined by the sum of the time taken to restart StatefulSet and the final-copy. The deletion and redeployment of StatefulSet and pods took between 0.040 seconds and 0.054 seconds, which is an exceedingly short duration regardless of the data size or storage solution, with negligible time differences. On the other hand, the time required for the final-copy was approximately 0.1 seconds at a minimum when Longhorn scales down from 2G to 1G, whereas it took about 42 seconds when Ceph scaled down from 100G to 50G. This difference in final-copy led to differences in service downtime. Therefore, the most significant factor affecting service downtime is final-copy, which involves the migration of the modified data.

Across all cases, data migration accounts for the largest portion of the execution time and directly affects downtime. This indicates that migration is the most time-consuming step. In particular, scaling down large volumes results in increased execution time, and frequent data changes can further extend service downtime.

B. AUTOSCALING RESOURCE USAGE

We conducted experiments to measure the CPU and memory usage, as well as disk reads/writes, for each test case. For the scale-up scenario, resource metrics were measured from the PVA pod, whereas for the scale-down scenario, metrics were collected from both the PVA and data migration pods. Unlike scale-up, PVA scale-down requires data migration, which necessitates resource measurements for both components.

The graphs in this section show resource usage and changes during scaling operations. The colored lines represent changes in volume size, and the gray dotted line indicates

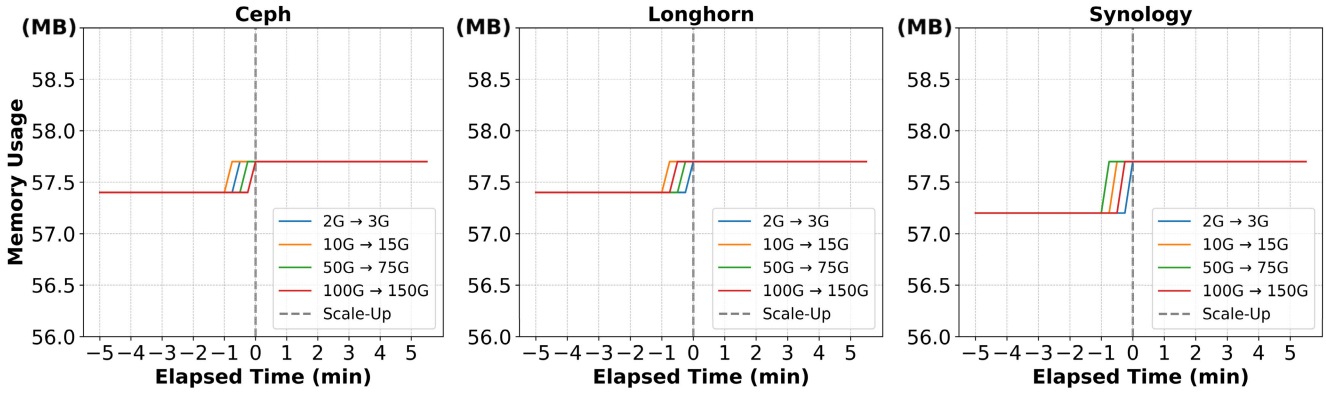


FIGURE 8. Memory usage in scale-up.

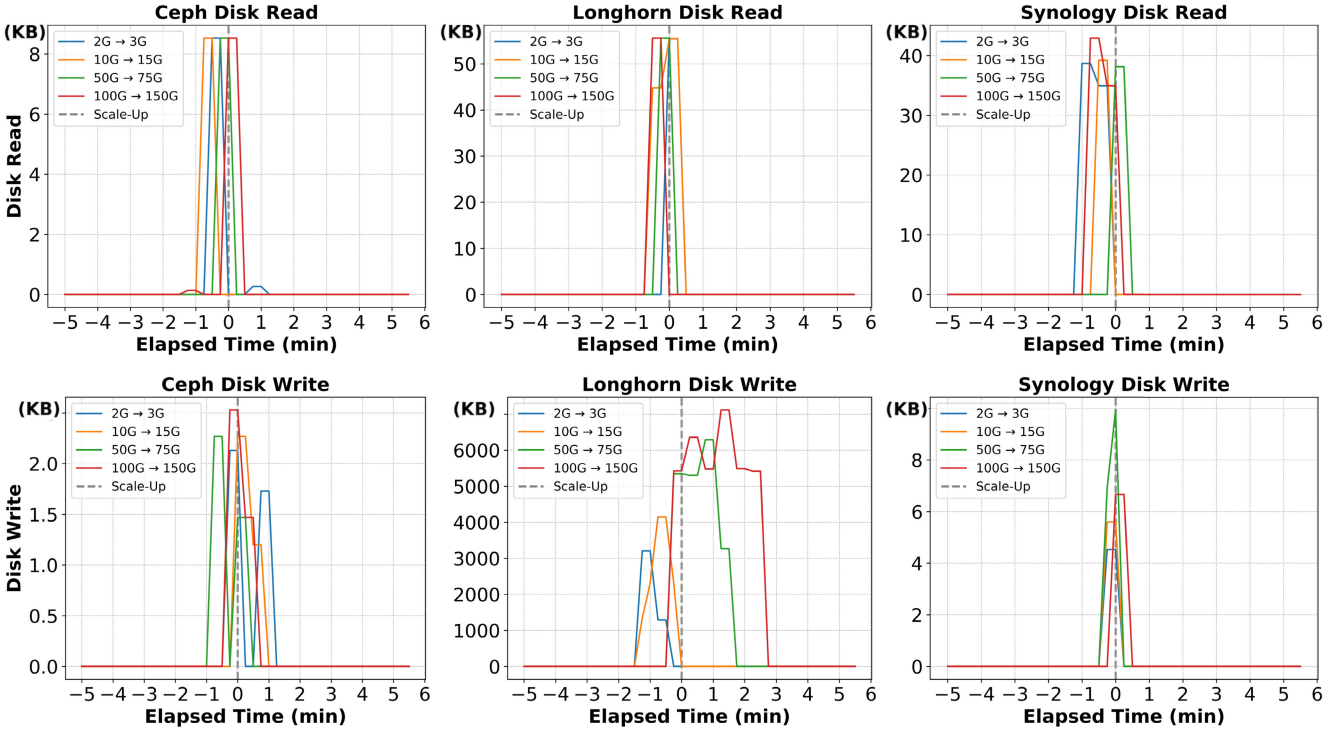


FIGURE 9. Disk read and write in scale-up.

when scaling was completed, marking the key point for observing resource consumption before and after the event. The x-axis in the graphs represents time flow during the scaling process. The center point, marked as 0, represents the completion of the scaling operation. The left side of the center indicates the time before scaling and the right side represents the time after scaling. The y-axis indicates the resource usage.

1) RESOURCE USAGE IN SCALE-UP

Figure 7 illustrates the CPU usage, measured as the CPU time consumed per second, during the volume scale-up operations using the PVA for each test case. A common pattern observed across all three storage solutions was a temporary increase in the CPU usage at the start of the scale-up process.

However, there was no apparent correlation between the CPU usage and volume size. The maximum CPU usage observed was 3.45 milliseconds in the case of Synology when expanding from 50GB to 75GB.

Although there was a brief increase in CPU usage during scale-up operations across all test cases, the usage remained very low, typically not exceeding 3 milliseconds in most cases. This clearly indicates that the scale-up operations using PVA require a negligible load on the CPU.

Figure 8 shows the changes in memory usage relative to the point at which volume expansion was completed. Similar to the CPU usage, there was no correlation between memory usage and data size. All test cases showed a general increase in memory usage during scale-up operations. The increase

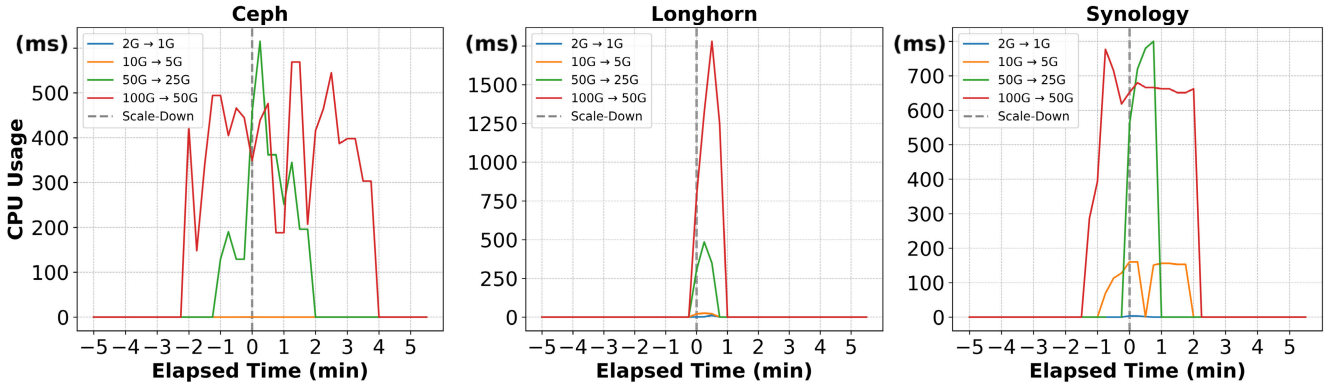


FIGURE 10. CPU usage in scale-down.

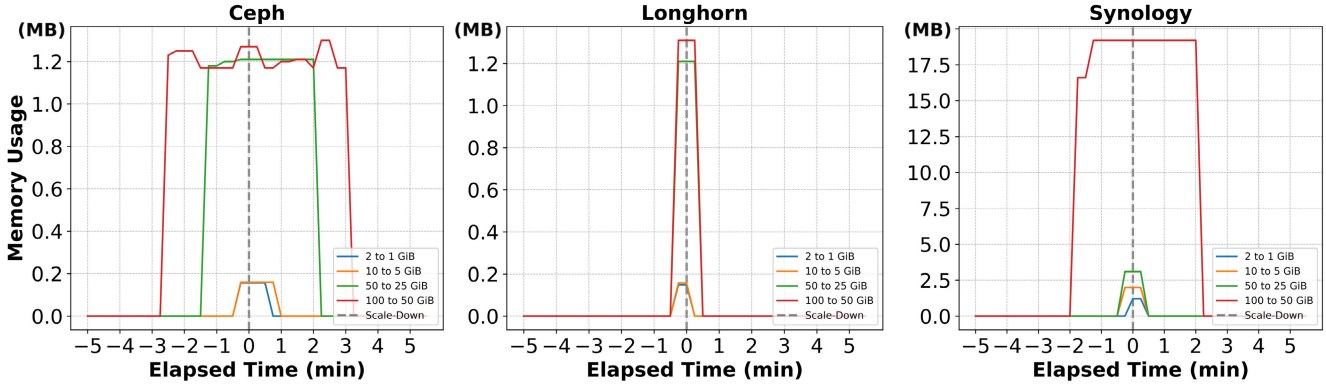


FIGURE 11. Memory usage in scale-down.

is minimal, with Ceph and Longhorn showing an increase of 0.03MB and Synology showing a 0.05MB increase. This suggests that PVA's scale-up operations consistently consume minimal memory resources across different environments, exerting almost no impact on the memory resources of the system.

Figure 9 depicts the disk read and write metrics measured during the scale-up operations using the PVA, showing the data throughput over time. A common trend observed across all storage solutions is an increase in disk read and write activities at the time of operation. Although no actual data were added to the disk during expansion operations with PVA, disk read and write activities occurred because metadata modifications when the volume size changed. Metadata contains information about the data, such as file size, creation time, and modification time [42]. This modification resulted in a temporary increase in disk usage.

According to the data, Longhorn exhibited the most frequent disk read and write operations, followed by Synology and Ceph. Longhorn had the highest usage, with reads peaking at 55KB and writings at approximately 7MB. Unlike other solutions, Longhorn showed a clear increase in disk write activity proportional to the volume size. This is related to Longhorn's metadata writing strategy, which indicates that more disk write operations are required as the volume size increases. However, because this experiment focused on identifying common trends in the performance of PVA across environments rather than analyzing individual storage solutions, a detailed analysis of strategy differences was omitted.

2) RESOURCE USAGE IN SCALE-DOWN

As explained in Section IV, migration in PVA was performed separately by temporarily creating a new pod. Consequently, the resource usage of the PVA pod during scale-down showed similar values and trends to those observed during scale-up. Therefore, resource usage experiments for scale-down primarily focus on the resource consumption of the pod responsible for migration, which is the most critical factor affecting the execution of scale-down.

Figure 10 illustrates CPU usage trends over time for the data migration pod during the scale-down. In contrast to the maximum CPU usage of 3.45 milliseconds recorded during the scale-up, all cases in the scale-down showed significantly higher CPU consumption. Additionally, as the

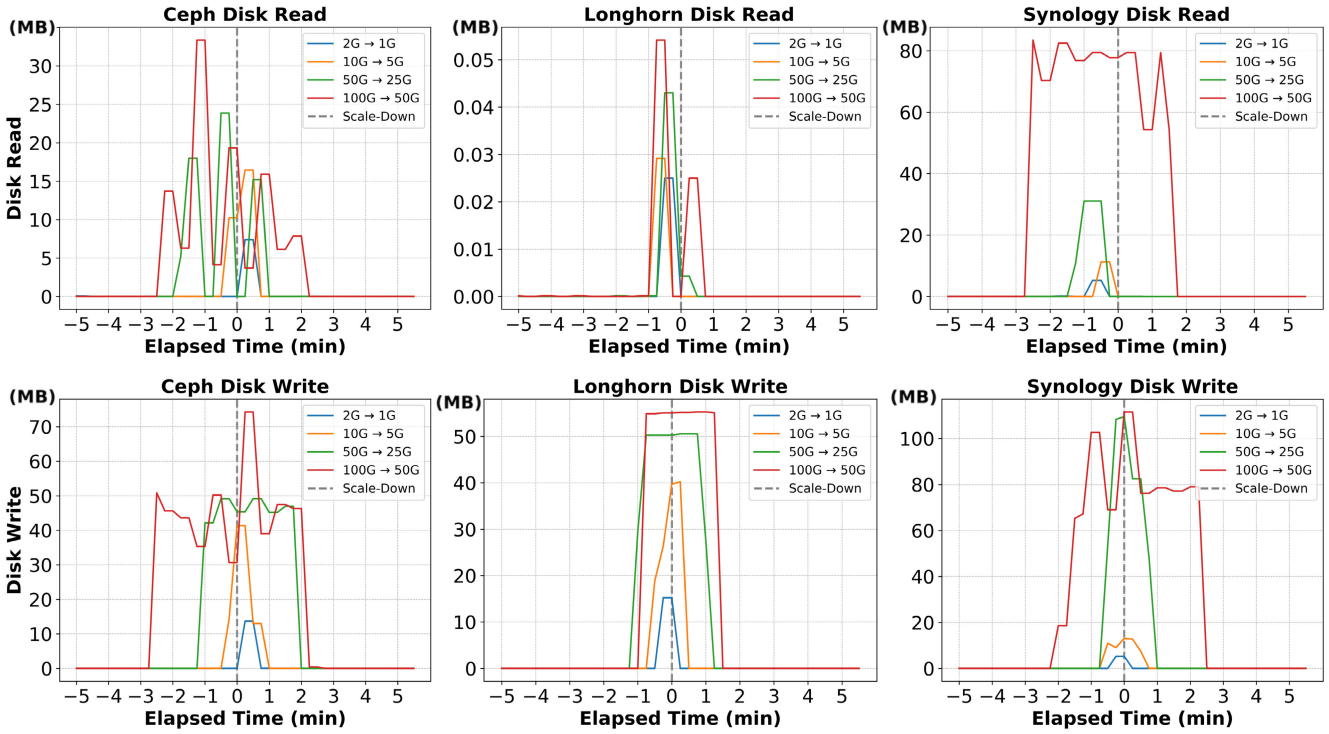


FIGURE 12. Disk read and write in scale-down.

data size increased, the CPU tended to be occupied for a longer duration.

Notably, in the case of scaling down from 100GB to 50GB, the CPU usage of Longhorn reached approximately 1.78, indicating that more than one CPU core was utilized. Compared to other storage solutions, Longhorn demonstrated high CPU usage over a shorter period, suggesting that it leverages CPU resources intensively to complete tasks quickly owing to its shorter scale-down execution time. Although this reduces the time required for large-scale data migration, it also indicates a potential spike in CPU usage.

Figure 11 shows the memory usage over time while performing data migration during the volume scale-down. Memory usage is higher than that during scale-up and increases with larger data sizes. For Ceph, the maximum memory usage was 1.3MB, and Longhorn also showed a similar usage. However, Longhorn exhibited a pattern of rapid increase and decrease in memory usage owing to its shorter scale-down duration.

In contrast, Synology showed a memory usage of approximately 19.2MB, which is approximately 15 times higher than that of other solutions. Given that the Synology operates as a NAS, it transmits data over the network, potentially requiring additional memory for network buffering. This implies that the amount of memory required for remote data migration increases, resulting in a higher memory usage.

Figure 12 presents the results of measuring the amount of disk read and write operations performed by the migration pod during scale-down. All the solutions exhibited a common trend, where the amount and duration of disk read and write usage were proportional to the data size. In cases with the largest size, Ceph required approximately 1.5 times more migration time than Synology; however, Synology performed more read and write operations. As previously mentioned, the additional overhead in Synology is due to the remote transmission of data over the network. In contrast, Longhorn showed the least disk usage, unlike the high disk overhead observed during scale-up owing to metadata processing. This indicates that Longhorn is optimized for handling stored data.

VIII. SUMMARY AND CONCLUSION

In this study, we propose a Persistent Volume Autoscaler to effectively manage volumes, the key resource of stateful applications, and evaluate its performance by testing it in various storage solution environments. The results demonstrate the potential applicability of PVA in diverse environments, offering a method to enhance service availability and resilience in Kubernetes. Furthermore, by minimizing the need for manual involvement in repetitive and time-consuming volume scaling tasks, PVA allows for more efficient utilization of volume resources.

The experimental results showed that although the scale-up operations of PVA exhibited consistent performance, the scale-down operations displayed performance variations depending on the data size and characteristics of the storage solutions. For scale-up, all tested storage solutions completed

the operation in approximately 0.1 seconds, regardless of data size, with generally low CPU, memory, and disk resource usage, indicating minimal impact on the system load. This suggests that PVA can efficiently perform expansion tasks, thereby enhancing service availability.

However, scale-down operations revealed significant differences in performance caused by data migration, which varied according to data size and storage solution. For all storage solutions, larger data sizes tended to increase resource usage. Ceph demonstrated stable overall resource consumption, but had the longest scale-down execution time. In contrast, Longhorn achieved the shortest scale-down execution time but required higher resource usage because of the need to perform intensive operations in a short period. Synology is a network-based storage solution that incurs additional resource overheads associated with network operations. These results indicate that PVA can operate effectively in various environments, but its performance may vary depending on the size of the data and characteristics of the storage solution.

The service downtime during the scale-down also varied based on both storage and data size. It is clear that service downtime occurs during the final-copy phase, which can affect service availability. However, this problem can be minimized by implementing an appropriate service failover strategy. For example, when operating a database service with three replicas, if a specific pod fails, the remaining pods can assume its role. Several stateful applications provide tools to implement such failover strategies [43], [44], [45], [46]. Therefore, by using these strategies in combination with PVA, service downtime can be prevented even during the scale-down process, leading to a more stable system operation.

Thus, because performance optimization based on the characteristics of each storage solution and recovery strategy is essential, future research should focus on these aspects to enhance the effective use of PVA. Additionally, there are plans to explore methods aimed at achieving a zero-downtime scale-down to reduce or eliminate the service downtime that is still present in the process. We also intend to enhance the scalability and stability of PVA by considering its extensive performance and efficiency during volume autoscaling in large-scale cluster environments. Through these efforts, we aim to make the PVA more effective in diverse cloud environments.

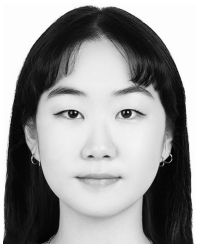
Currently, our work primarily focuses on volume usage-based scaling. However, to provide a more comprehensive approach to resource management, we plan to also consider CPU and memory usage. Future research will explore methods for optimizing resource utilization from multiple perspectives using machine learning. This approach is expected to enhance the scalability and efficiency of PVA, making it more adaptable to dynamic workloads.

REFERENCES

[1] *USCNCF Annual Survey 2023*. Accessed: Apr. 2024. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2023/
[2] *USCNCF Annual Survey 2022*. Accessed: Jan. 2023. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2022/
[3] *USData on Kubernetes 2022 Report*. Accessed: Oct. 2022. [Online]. Available: https://dok.community/data-on-kubernetes-2022-report
[4] P. Denzler, D. Ramsauer, T. Preindl, W. Kastner, and A. Gschnitzer, "Comparing different persistent storage approaches for containerized stateful applications," in *Proc. IEEE 27th Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2022, pp. 1–8.
[5] *Vertical Pod Autoscaler*. Accessed: Sep. 10, 2024. [Online]. Available: https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler
[6] *Autoscaling Workloads*. Accessed: Sep. 10, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/autoscaling/
[7] Y. X. Chia, C. K. Seow, K. Chen, and Q. Cao, "Exploring resource prediction models based on custom Kubernetes auto-scaling metrics," in *Proc. 9th Int. Conf. Cloud Comput. Big Data Anal. (ICCCBDA)*, Apr. 2024, pp. 47–52.
[8] *Storage Classes*. Accessed: Sep. 10, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/storage/storage-classes/
[9] *CSI*. Accessed: Sep. 10, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/storage/volumes/#csi
[10] *Persistent Volumes*. Accessed: Sep. 10, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/storage/persistent-volumes/
[11] N. D. Nguyen and T. Kim, "Balanced leader distribution algorithm in Kubernetes clusters," *Sensors*, vol. 21, no. 3, p. 869, Jan. 2021.
[12] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Microservice based architecture: Towards high-availability for stateful applications with Kubernetes," in *Proc. IEEE 19th Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2019, pp. 176–185.
[13] P. Choonhaklai and C. Chantrapornchai, "Two autoscaling approaches on Kubernetes clusters against data streaming applications," in *Proc. Int. Tech. Conf. Circuits/Syst., Comput., Commun. (ITC-CSCC)*, Jun. 2023, pp. 1–6.
[14] *Mysql's State*. Accessed: Sep. 10, 2024. [Online]. Available: https://dev.mysql.com/doc/mysql-operator/en/mysql-operator-introduction.html
[15] *Running ZooKeeper, A Distributed System Coordinator*. [Online]. Available: https://kubernetes.io/docs/tutorials/stateful-application/zookeeper/
[16] *Apache Kafka Stream*. Accessed: Sep. 10, 2024. [Online]. Available: https://kafka.apache.org/35/documentation/streams/developer-guide/dsl-api.html#stateful-transformations
[17] N. Nguyen and T. Kim, "Toward highly scalable load balancing in Kubernetes clusters," *IEEE Commun. Mag.*, vol. 58, no. 7, pp. 78–83, Jul. 2020.
[18] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "A Kubernetes controller for managing the availability of elastic microservice based stateful applications," *J. Syst. Softw.*, vol. 175, May 2021, Art. no. 110924.
[19] K. Pakrijauskas and D. Mažeika, "On recent advances on stateful orchestrated container reliability," in *Proc. IEEE Open Conf. Electr., Electron. Inf. Sci. (eStream)*, Apr. 2021, pp. 1–6.
[20] B. Everman, M. Gao, and Z. Zong, "Evaluating and reducing cloud waste and cost—A data-driven case study from Azure workloads," *Sustain. Comput., Informat. Syst.*, vol. 35, Sep. 2022, Art. no. 100708.
[21] U. Villano, M. Rak, and M. Catillo, "A survey on auto-scaling: How to exploit cloud elasticity," *Int. J. Grid Utility Comput.*, vol. 14, no. 1, p. 37, 2022.
[22] M.-N. Tran, D.-D. Vu, and Y. Kim, "A survey of autoscaling in Kubernetes," in *Proc. 13th Int. Conf. Ubiquitous Future Netw. (ICUFN)*, Jul. 2022, pp. 263–265.
[23] A. Shemyakinskaya and I. Nikiforov, "Disk space management automation with CSI and Kubernetes," in *Proc. 7th Int. Congr. Inf. Commun. Technol.*, in Lecture Notes in Networks and Systems, vol. 447, X.-S. Yang, S. Sherratt, N. Dey, and A. Joshi, Eds. Singapore: Springer, 2023, pp. 171–179.
[24] H. C. S. Perera, T. S. D. De Silva, W. M. D. C. Wasala, R. M. P. R. L. Rajapakshe, N. Kodagoda, U. S. S. Samaratunge, and H. H. N. C. Jayanandana, "Database scaling on Kubernetes," in *Proc. 3rd Int. Conf. Advancements Comput. (ICAC)*, Dec. 2021, pp. 258–263.
[25] *Amazon Aurora Auto Scaling*. Accessed: Sep. 10, 2024. [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Integrating.AutoScaling.html
[26] *Overview of Autoscale in Azure*. Accessed: Apr. 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview

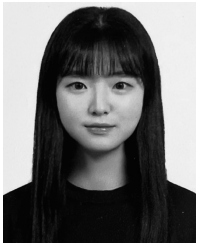
[27] *Working With Storage for Amazon RDS DB Instances*. [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_PIOPS.StorageTypes.html#autoscaling-limitations
[28] *DevOps-Nirvana/Kubernetes-Volume-Autoscaler*. Accessed: Sep. 2024. [Online]. Available: https://github.com/DevOps-Nirvana/Kubernetes-Volume-Autoscaler
[29] I. Konev, I. Nikiforov, and S. Ustinov, "Algorithm for containers' persistent volumes auto-scaling in Kubernetes," in *Proc. 31st Conf. Open Innov. Assoc. (FRUCT)*, Apr. 2022, pp. 89–95. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9770916/
[30] *Kubernetes Components*. Accessed: Sep. 10, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/
[31] O. Mart, C. Negru, F. Pop, and A. Castiglione, "Observability in Kubernetes cluster: Automatic anomalies detection using prometheus," in *Proc. IEEE 22nd Int. Conf. High Perform. Comput. Commun., IEEE 18th Int. Conf. Smart City, IEEE 6th Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, Dec. 2020, pp. 565–570. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9407871/
[32] *What is Prometheus?* Accessed: Sep. 10, 2024. [Online]. Available: https://grafana.com/docs/grafana/latest/fundamentals/intro-to-prometheus/
[33] *Prometheus Data Source*. Accessed: Sep. 10, 2024. [Online]. Available: https://grafana.com/docs/grafana/latest/datasources/prometheus/
[34] *kubernetes-Client/Python*. Accessed: Sep. 2024. [Online]. Available: https://github.com/kubernetes-client/python
[35] *Rsync*. Accessed: Sep. 10, 2024. [Online]. Available: https://rsync.samba.org/
[36] T. Benjaponpitak, M. Karakate, and K. Sripanidkulchai, "Enabling live migration of containerized applications across clouds," in *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, Jul. 2020, pp. 2529–2538.
[37] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Oper. Syst. Des. Implement.*, 2006, pp. 307–320. [Online]. Available: https://www.usenix.org/events/osdi06/tech/full_papers/weil/weil_html
[38] *What is Longhorn?* Accessed: Sep. 10, 2024. [Online]. Available: https://longhorn.io/docs/1.7.2/what-is-longhorn/
[39] K. Jeong, C. Duffy, J.-S. Kim, and J. Lee, "Optimizing the Ceph distributed file system for high performance computing," in *Proc. 27th Euromicro Int. Conf. Parallel, Distrib. Netw. Process. (PDP)*, Feb. 2019, pp. 446–451.
[40] T. Baptista, L. B. Silva, and C. Costa, "Highly scalable medical imaging repository based on Kubernetes," in *Proc. IEEE Int. Conf. Bioinf. Biomed. (BIBM)*, Dec. 2021, pp. 3193–3200.
[41] *Longhorn Architecture and Concepts*. Accessed: Sep. 10, 2024. [Online]. Available: https://longhorn.io/docs/1.7.0/concepts/
[42] A. Merenstein, "Techniques for storage performance measurement and data management in container-native and serverless environments," Ph.D. dissertation, Dept. Comput. Sci., Stony Brook Univ., Stony Brook, NY, USA, 2024.
[43] *MySQL Operator for Kubernetes Manual*. Accessed: Sep. 10, 2024. [Online]. Available: https://dev.mysql.com/doc/mysql-operator/en/
[44] *MongoDB Enterprise Kubernetes Operator*. Accessed: Sep. 10, 2024. [Online]. Available: https://www.mongodb.com/docs/kubernetes-operator/v1.23/multi-cluster-disaster-recovery/
[45] *postgres-Operator*. Accessed: Sep. 10, 2024. [Online]. Available: https://github.com/CrunchyData/postgres-operator
[46] *K8ssandra Operator*. Accessed: Sep. 10, 2024. [Online]. Available: https://docs.k8ssandra.io/components/k8ssandra-operator/

JI-HYUN NA received the B.S. degree in computer science from Chungbuk National University, South Korea, in February 2020, where she is currently pursuing the M.S. degree with the Computer Science Department. She previously worked with the System Infra Department, Korea Electric Power Corporation Knowledge, Data and Network. Her research interests include cloud computing, container, and system infrastructure engineering.



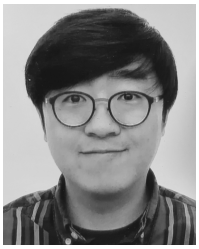
HYEON-JIN YU received the B.S. degree in computer science from Chungbuk National University, South Korea, in February 2023, where she is currently pursuing the Ph.D. degree with the Computer Science Department. Her research interests include scientific and high performance computing and grid software optimization.

HYEONGBIN KANG received the B.S. degree in computer science from Chungbuk National University, South Korea, in February 2023, where he is currently pursuing the M.S. degree with the Computer Science Department. His research interests include cloud computing, performance engineering, and IT governance.



HEEJU KANG is currently pursuing the B.S. degree in computer science with Chungbuk National University, South Korea. Her research interests include cloud computing and container.

HEE-DONG LIM received the B.S. and M.S. degrees in computer engineering from Chungbuk National University, South Korea, where he is currently pursuing the Ph.D. degree with the Department of Computer Engineering. His research interests include artificial intelligence, especially natural language processing.



JAE-HYUCK SHIN received the B.S. and M.S. degrees from Chungbuk National University, South Korea, where he is currently pursuing the Ph.D. degree with the Department of Computer Science. He previously worked with Samsung Research. His research interests include distributed computing and embedded systems.

SEO-YOUNG NOH received the B.E. and M.E. degrees in computer engineering from Chungbuk National University, South Korea, and the M.S. and Ph.D. degrees in computer science from Iowa State University. He is currently an Associate Professor with the School of Computer Science, Chungbuk National University. Prior to joining Chungbuk National University, he was a Principal Researcher with the National Supercomputing Division, Korea Institute of Science and Technology Information, and as a Chief Research Engineer with LG Electronics. His research interests include scientific data management, cloud and scientific computing, Linux platforms, and database management systems.