

Article

Multi-Version Managers for Large Scalable Data-Management Systems

Baya Chalabi ^{1,*}  and Yahya Slimani ² 

¹ Laboratoire de la Communication dans les Systèmes Informatiques, Ecole Nationale Supérieure d'Informatique, BP 68M, Oued-Smar 16309, Alger, Algeria

² Institut Supérieur des Art Multimédias de la Manouba, Université de la Manouba, Manouba, 2010 Tunisia; yahya.slimani@fst.rnu.tn

* Correspondence: b_chalabi@esi.dz

Abstract

With the emergence of data-intensive computing, which is due to the growth of the data produced and generated each day, it became necessary to store and manage big data. Cloud data storage is actually the best choice for large distributed systems. Successful Cloud Computing cannot be achieved without a reliable data-management system to store and handle the enormous volume of data. Management of the available storage system at large scale becomes progressively more complicated, and we face many challenges, such as scalability, data availability, fault tolerance, etc. Also, data storage is faced with specific access patterns: highly concurrent reads of data from the same file, many overwrites, and very concurrent appends to the same file. Most of the existing storage systems use versioning to bring and enhance data access parallelism and this enables better performance levels under concurrency; but, generally, these systems use one component (version manager), which is responsible for generating new versions of each file stored. When we speak in the context of big data, the requests for read, write and append increase. If these requests are managed by a single component, then we have a performance bottleneck and an overloaded version manager. To avoid this drawback, we proposed and designed a new architecture of storage systems that uses versioning; the new architecture uses multi-version managers to support better the scalability and provide partial fault tolerance. To illustrate the practicability of our approach, we assessed it on the BlobSeer data-management system. The experimental results demonstrate that our architecture achieves near-linear scalability for CREATE operations (495 ops/sec per additional version manager), reduces WRITE execution time by up to 66%, and maintains 67% availability under single-node failures, all while introducing minimal resource overhead (3% aggregate CPU increase). These results confirm that the proposed multi-version manager architecture offers a practical, scalable, and partially fault-tolerant solution for Cloud data-storage systems.

Keywords: cloud computing; versioning; data-management system; BlobSeer; scalability; multi-version managers



Academic Editor: Filipe Portela

Received: 22 May 2026

Revised: 6 July 2026

Accepted: 9 July 2026

Published: 13 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

More data was generated in the last two years than in the entirety of human history before this. There are many sources that predicted exponential data growth toward 2020 and beyond. Yet they are all in broad agreement that the size of the digital universe will double every two years at least: a 50-fold growth from 2010 to 2020. Human and machine

generated data is experiencing an overall $10\times$ faster growth rate than traditional business data, and machine data is increasing even more rapidly, at $50\times$ the growth rate. The large volume and speed of data growth has an $110\times$ faster growth rate than traditional business data, and machine data is increasing, introducing several challenges, among which we can cite system management and growing cluster complexity, storage, data movement, data-management complexity, and so on.

Two fundamental technologies to enter into mainstream information technology are Big Data and Cloud Computing. These two, however, are not the same: Big Data represents content whereas Cloud Computing is an infrastructure, and the combination of both technologies can yield valuable results. Through hardware virtualization, Cloud Computing provides the option of storing significant amounts of data with the help of scalability, fault tolerance and availability. This allows Big Data to be available, scalable and fault tolerant through Cloud Computing.

The relative ease of collaborative data science and analysis has led to a proliferation of many thousands or millions of versions of the same datasets in many scientific and commercial domains. Managing, storing, and recreating these dataset versions is a non-trivial task. The fundamental challenge here is the storage-recreation trade-off: the more storage we use, the faster it is to recreate or retrieve versions, while the less storage we use, the slower it is to recreate or retrieve versions.

Big data involves manipulating petabytes (and perhaps exabytes and zettabytes) of data, and the Cloud's scalable environment makes it possible to deploy data-intensive applications. The Cloud also simplifies connectivity and collaboration within an organization.

The managing of large storage systems in the Cloud faces complicated issues [1]:

- Access transparency: The user should be acceded to the remote files with the same methods as local ones (no difference between local and remote).
- Localization transparency: The client does not have any information about the file location irrespective of whether it is a local or remote file.
- Concurrent access transparency: If there is any change in a file made by a concurrent process, this change should be seen by the other processes in a consistent state.
- Failure transparency: If there is failure in any node in the system, this failure should not affect the execution of the whole system.
- Heterogeneity: Different hardware platforms and operating systems can be used to store file systems.
- Scalability: This refers to the ability of a file system to adapt to a change in order of the request size and the number of nodes used.
- Replication transparency: The file system should create replicas automatically.
- Migration transparency: A file or a part of a file location can be moved in a transparent manner from the clients.
- Support fine-grained distribution of data: The execution of the programs that manipulate files should be in the same location where the files are stored, to optimize performance.
- Tolerance for network partitioning: If the entire network or certain segments of it are unavailable during some periods of time, the system must be able to support this unavailability.

The majority of the existing file systems use versioning or snapshots to allow highly concurrent reads of data from the same file, few overwrites, and highly concurrent appends to the same file. The versioning certainly implies that multiple copies of the same data need to be kept after each write or append. If we use only one component to manage the versioning and if the requests increase (new file creation), it is evident that the version manager becomes a bottleneck. As a consequence of this bottleneck, the scalability is

very limited. Much of the motivation for multi-version databases has been due to their perceived advantages over single-version systems. Chief among these is the fact that multi-version databases have the potential to reduce the number of read–write conflicts among transactions. Because multi-version databases do not update data in place, conflicting reads and writes can be allowed to occur in parallel. Reads can be directed to pre-existing versions of data, while writes create new versions. State-of-the-art multi-version databases exploit this greater read–write concurrency under weak isolation levels such as snapshot-isolation and read-committed. In this paper, we address the scalability and fault-tolerance limitations of single version-manager architectures by proposing a distributed multi-version manager design. Unlike existing systems—such as HDFS, BlobSeer, XtremFS and RStore—which rely on a single centralized component for version assignment, our approach distributes versioning metadata across multiple managers, eliminating the performance bottleneck and enabling horizontal scaling. The key innovations of our work are (i) a decentralized version management architecture that supports highly concurrent CREATE, READ, and WRITE operations; (ii) an autonomous scaling mechanism that dynamically activates standby version managers based on a heuristic load metric (CPU, bandwidth, RAM, and disk utilization); and (iii) a ring-based election protocol that provides partial fault tolerance by recovering the version-manager controller within 0.8 s of failure. We implement our approach on the BlobSeer data-management system and evaluate it on an 11-node cluster. Experimental results demonstrate scalability for CREATE, WRITE and READ operations with minimal resource overhead (3% aggregate CPU increase). These results confirm that our approach effectively addresses the single-version-manager bottleneck while providing partial fault tolerance for large-scale distributed storage systems. The remainder of this paper is organized as follows. Section 2 presents four file systems that use versioning to allow concurrent access to replicated files. Section 3 describes our proposal. Section 4 explains the implementation of our proposal in BlobSeer as a case study. Section 5 presents and discusses the experimental results. Section 6 discusses the limitations of the proposed approach. Section 7 concludes the paper.

2. Examples of Versioning Data-Storage Systems

To discuss versioning data-storage systems, we chose to present four data-management systems used in large distributed systems: HDFS [2,3], BlobSeer [4–6], XtremFS [7] and RStore [8].

2.1. HDFS Architecture

As shown in Figure 1, HDFS separates the storage of file-system metadata and application data; the same architecture had been used by the system MSCFS-RP [9]:

- **NameNode:** This represents the master node in the Apache Hadoop HDFS Architecture that maintains and manages the blocks stored on the DataNodes (slave nodes). NameNode is a very highly available server that manages file-system namespace (or metadata) and controls access to files via client applications, with multiple data nodes (in hundreds or thousands) where each data node manages file storage and the storage device attached to it. For each data file, NameNode stores the Versioning Information (VI) as parts of its metadata.
- **DataNodes:** These components are the slave nodes in HDFS. Unlike NameNode, DataNode is a commodity hardware; that is, a non-expensive system that is not of high quality or high availability. The DataNode is a block server that stores the data in a local file. In DataNode, actual data is stored, and NameNode stores the metadata; that is, the file location, block size, file permission, etc. It also receives heartbeats from

live DataNodes, so it is also responsible for sending a signal to replicate if DataNode is no longer available.

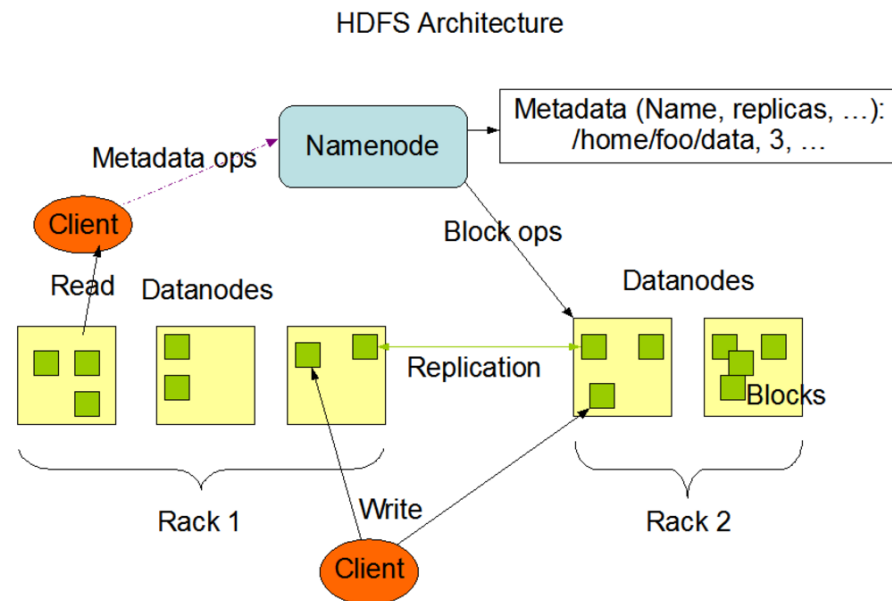


Figure 1. HDFS architecture [10].

2.2. BlobSeer Architecture

BlobSeer [5,11] stores unstructured data in Blobs (Binary Large Object). Each Blob is split into chunks that are distributed among data providers. BlobSeer consists of a series of distributed communicating processes that communicate through a Remote Procedure Call model (RPC) [12,13]. Figure 2 illustrates the BlobSeer architecture.

- **Data providers:** The data providers physically store the chunks generated by APPENDS and WRITES primitives.
- **Provider manager:** This component stores information about the available storage space and schedules the location of newly generated chunks.
- **Metadata providers:** These components physically store the metadata that are permitted to identify the chunks of any snapshot version of each Blob. BlobSeer manipulates a distributed metadata management to enhance concurrent access to metadata.
- **Version manager:** The role of the version manager is to assign a new snapshot version numbers to writers and appenders and to reveal these new snapshots version to readers. The version manager is an important component in BlobSeer because it is the only one that guarantees serialization.
- **Clients:** BlobSeer offers a client interface that can provide the following primitives: CREATE a Blob, READ, WRITE, and APPEND on a specific Blob. A large number of concurrent clients is expected (many clients can access the same Blob).

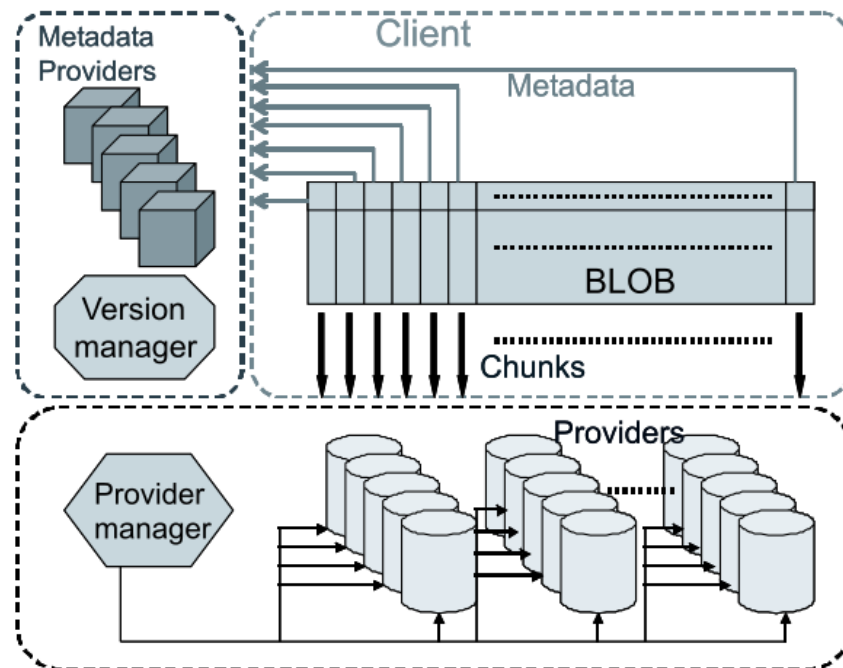


Figure 2. BlobSeer architecture [14].

BlobSeer and Versioning

When a user calls the WRITE primitive for a Blob, it must specify the ID of the Blob, the offset and the size of the entire sequence. These parameters are needed to write the chunks of the Blob on data providers.

1. WRITE primitive:

- The client contacts the provider manager and requests a number of data providers equal to the number of chunks to be written, multiplied by the number of replicas.
- Upload the data chunks to the data providers in a parallel and asynchronous way.
- The client contacts the version manager to notify it about its WRITE primitive. The version manager assigns a version number to the client and adds the WRITE into a queue containing WRITE primitives in progress. APPEND primitives are similar to WRITE, but they write at the end of the Blobs.

2. READ primitive: To support many READ primitives at the same time, BlobSeer uses the following process:

- Firstly, the client contacts the version manager, which stores the root information for all Blobs stored in the system, by specifying the Blob identifier and the Blob version number.
- Next, the client sends read primitives to the metadata providers to get the location of the data providers that physically store the data chunks that need to be read.
- In parallel, the client downloads the data chunks from the data providers.

2.3. XtreamFS Architecture

The architecture of XtreamFS [7,15] is composed of distributed entities, as illustrated in Figure 3.

- **DIRECTORY SERVICE:** MRCs and OSDs register themselves with their addresses in the directory service. That allow clients to discover the other server components of a XtreamFS deployment.

- **METADATA AND REPLICA CATALOG (MRC):** XtreamFS MRCs store the volumes with their directory tree and file metadata. The MRC is responsible for checking permissions for opening files. The DIR and MRC services store their data in the key-value store BabuDB [16], which has been designed with the aim of storing file-system metadata.
- **Object Storage Devices (OSDs):** File content is stored on XtreamFS OSDs. The OSD service splits files into chunks of a configurable size and stores each chunk in its own file in a local file system and implements the replication. To manage versioning, XtreamFS uses different replicas for each object, and a master replica is responsible for creating an object version via its communications with other replicas. The OSD records the version and the file size in a persistence file-version log. Metadata about the file version is stored in MRCs, and MRCs record versions of volumes when the client sends snapshot requests.

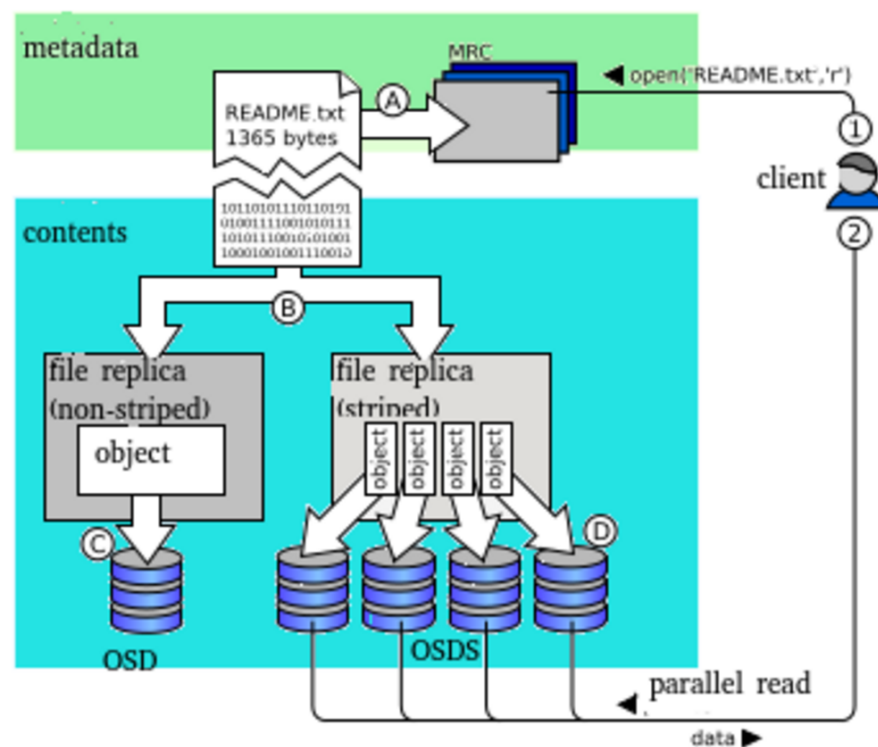


Figure 3. XtreamFS architecture [7].

2.4. RStore

RStore [8] is a distributed multi-version document store designed to compactly store a large number of versions (snapshots) of keyed documents or records. The primary components that constitute it are the following:

- **Client:** This is the interface through which users interact with RStore; it submits new versions of documents or datasets to the system.
- **Key-Value Store (KVS):** This stores the actual raw data chunks generated by the Data Ingest Module and version metadata (version IDs, parent–child relationships, and chunk mappings).
- **Data Ingest Module:** This is responsible for handling the submission of new versions from clients by generating a unique identifier for each new version submitted to the system. It then computes the difference (delta) between the new version and its parent version to avoid storing redundant data. Finally, it writes the new version data and metadata to the underlying key-value store (this maintains the version graph to track parent–child relationships between versions) [8].

- **Data Placement Module:** This is the core of RStore. It partitions distinct records from all versions into approximately equal-sized chunks. It determines the optimal placement of records into chunks, minimizing storage costs by eliminating redundant data across versions (using deltas and deduplication) [8].
- **Query Processing Module:** This is responsible for handling and efficiently responding to client queries.

3. Design of Our Proposed Approach

The majority of existing data-management storage systems employ a single component, the version manager, in their architecture to store metadata, generate new versions of Blobs, and ensure the serialization of version assignment and the uniqueness of version IDs. However, in the context of Cloud Computing, where data size grows exponentially, READ and WRITE/APPEND primitives can generate a high workload. As a consequence, the version manager becomes a bottleneck. The limitations of these systems are summarized in Table 1.

Table 1. Comparison of existing multi-version storage systems.

System	Versioning Approach	Primary Limitation	Fault Tolerance
HDFS	Snapshots (namespace-level)	Single NameNode (SPOF and metadata bottleneck)	Limited (HA requires complex setup)
BlobSeer	Per-blob versioning with centralized version manager	Centralized version manager (serialization bottleneck)	None (single point of failure)
XtreemFS	Master–slave replica versioning	Centralized MRC and master replica bottleneck	Partial (replication)
RStore	Keyed document versioning	Focused on storage efficiency, not high-concurrency writes	Depends on underlying KVS

Note: SPOF = single point of failure; HA = high availability; MRC = metadata and replica catalog; and KVS = key-value store.

In our proposal, as the files or Blobs are independent between them, there is no need to manipulate and generate a new version with the same component. Hence, to increase the scalability of the system, we propose a new architecture. In this architecture, we used some components that we will describe below (see Figure 4):

- **Version-manager controller (Vmc):** It is responsible for increasing the scalability of the system by adding a new version manager with a unique ID and for publishing its ID and IP address to other components if 60% of the existing version managers are overloaded (their workload exceeds a predefined threshold). Also, it periodically stores the load sent by each version manager.
- **Version manager (Vmi):** Its role is to create a new version of the file or the Blob after each modification (WRITE/APPEND). It can also resend a request to create a new file to another version manager if it is overloaded. If it receives a create request from another version manager and if its status is also overloaded, it transmits the request to the version-manager controller to get the underloaded version manager. Finally, it resends the create request to the unloaded version manager.

In case of the failure of the version-manager controller (it does not respond after two tries to connect from one version manager), this version manager launches an election process using a ring election algorithm [17,18]. We present the pseudocode of the ring election algorithm in Algorithm 1.

Algorithm 1 Ring Election Algorithm with Configuration Distribution

Require: Each VM has a unique ID, knows its successor, and maintains a configuration file.

Ensure: A new Vmc is elected and the configuration is distributed to all nodes.

```

1: Step 1: Failure Detection
2: for each VM do
3:   Send heartbeat to Vmc every 10 s
4:   if no heartbeat acknowledgment received for 30 s then
5:     controller_failed ← TRUE
6:     call STARTELECTION()
7:   end if
8: end for
9: Step 2: Election Initiation
10: function STARTELECTION
11:   if election_in_progress then
12:     return
13:   end if
14:   election_in_progress ← TRUE
15:   highest_id ← self.id
16:   call SENDELECTIONMESSAGE(successor, highest_id)
17: end function
18: Step 3: Propagation and Comparison
19: function HANDLEELECTIONMESSAGE(sender_id, candidate_id)
20:   if candidate_id > self.id then
21:     call SENDELECTIONMESSAGE(successor, candidate_id)
22:   else if candidate_id < self.id then
23:     call SENDELECTIONMESSAGE(successor, self.id)
24:   else if candidate_id == self.id then
25:     call DECLARESELFASCONTROLLER()
26:   end if
27: end function
28: Step 4: Winner Declaration
29: function DECLARESELFASCONTROLLER
30:   self.is_controller ← TRUE
31:   call UPDATECONFIGURATIONFILE()
32:   call BROADCASTUPDATEDCONFIGURATION()
33: end function
34: Step 5: Update Configuration File
35: function UPDATECONFIGURATIONFILE
36:   config ← load_configuration_from_file()
37:   for each VM in config.active_vms do
38:     if VM.last_heartbeat < now() – 30 then
39:       config.active_vms.remove(VM)
40:       config.failed_vms.append(VM)
41:     end if
42:   end for
43:   config.controller_id ← self.id
44:   config.controller_ip ← self.ip
45:   config.version ← config.version + 1
46:   save_configuration_to_file(config)
47: end function
48: Step 6: Broadcast Configuration
49: function BROADCASTUPDATEDCONFIGURATION
50:   config ← load_configuration_from_file()
51:   all_nodes ← config.active_vms + config.data_providers + config.metadata_providers
52:   for each node in all_nodes do
53:     call SENDCONFIGURATIONUPDATE(node, config)
54:   end for
55: end function

```

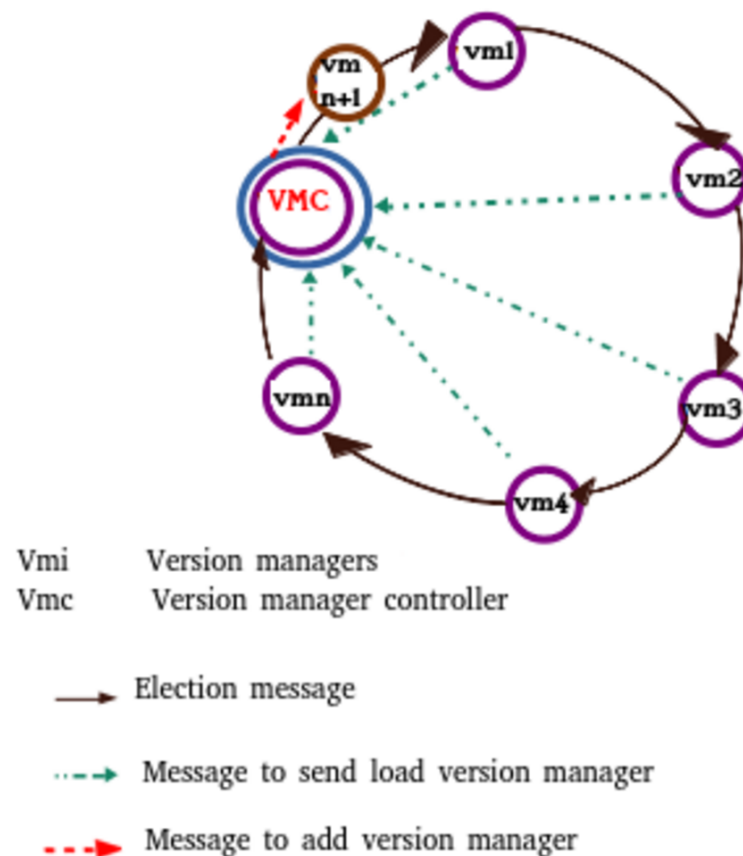


Figure 4. Version managers' organization and communication in scalable system.

3.1. Generating Blob's ID

To guarantee that we have a unique ID for a Blob, we propose the following structure for the ID parameter: The first item (Field1) represents the logical number of the version manager, while the second one (Field2) is a number that is incremented locally after each generation of a new Blob (counter). Technically, the identifier is a 64-bit unsigned integer structured into two distinct, non-overlapping bit fields:

1. Higher 8 bits (bits 63–56): Reserved for Field1 (VM-ID) (unique identifier of the version manager (0 to 255)).
2. Lower 56 bits (bits 55–0): Reserved for Field2 the local counter (increasing sequence number (0 to $2^{56} - 1$)).

We use bitmask for safety, to prevent accidental corruption of other fields during bitwise operations, and to ensure that values do not exceed their allocated bit capacity; additionally, in extraction (VM-ID or counter), it allows precise isolation of two fields from the identifier. Also, if the system is extended, masks can be adjusted without breaking existing code. We define two essential masks in Table 2:

Table 2. Bitmask definitions for 64-bit identifier extraction and validation.

Mask Name	Value	Purpose
VM_ID_MASK	0xFF (binary: 1111 1111)	Isolates the lower 8 bits (VM ID).
COUNTER_MASK	0xFFFFFFFFFFFFFFF (56 bits of 1s)	Isolates the lower 56 bits (counter).

This architecture is inherently conflict-free due to a combination of spatial separation (different VMs) and temporal atomicity (single VM sequencing). The formal proof is below:

- Prevents conflict between different VMs; two distinct VMs have different version manager's identifiers (VM_1's ID: 00000001 and VM_5's ID: 00000101). When shifted to the high bits, these values produce a fundamentally different 64-bit. The ID generated by VM_1 always falls within the block 0x01XX..., while VM_5 always falls within 0x05XX.... These ranges never intersect.
- Conflict Prevention Within a Single VM : For a specific VM, uniqueness depends entirely on its local 56-bit counter. The counter is designed to be incremented after creation of a new Blob (0, 1, 2, 3, etc.). Even with thousands of concurrent CREATE requests hitting the same VM, the system enforces strict serialization during the counter extraction phase. The process of "get counter, construct identifier ID, increment counter" is performed as an atomic and indivisible operation.

3.2. Managing CREATE Primitive

The client randomly chooses a version manager and sends CREATE primitive to it. If the version manager chosen is underloaded (its ID = 5) and the number of files/Blobs already created in this version manager is equal to 100, then the version manager creates a file and assigns it the following ID: ID of file = version manager ID.number of file (5.100).

3.3. Managing READ or WRITE/APPEND Primitives

If a client wants to read from or write into a given file, it is necessary to determine which version manager created the file. For this, it extracts the version manager ID from the file ID and then connects to this version manager to do the READ or WRITE primitive. If the file ID is 2.200 (2, the version manager ID and 200, the number of the file), to read or write to the file (2.200) it has to contact version manager 2 for READ or WRITE/APPEND to file 200.

4. Case Study: The BlobSeer System

We have chosen the BlobSeer storage system [4] to design and implement our proposition because this system guarantees the scalability of the system by managing metadata in a distributed manner. Also, it permits concurrent access to the same Blob (efficient WRITE/WRITE concurrency).

New Proposed Architecture

The new proposed architecture and the interaction between its different components are represented in Figure 5: For clients, and for data provider, provider manager and metadata provider components, their roles remain the same as those expressed in Section 2.2.

- Version manager: The difference in this new architecture for this component is the use of several version managers to increase the scalability, instead of using only one, by using a new structure for the identifier of the Blob. Version managers communicate with each other to elect a version-manager controller if the latter breakdown. Hence, the version managers periodically send two metrics (memory space and CPU usage ratio) to the version-manager controller.
- Version-manager controller: Its function is to save the metrics sent by each version manager using a monitoring component; in addition, it adds a new version manager if the existing ones are overloaded. Then, it informs version managers of the existence of a new version manager. This operation is achieved through a message containing the ID and the IP address of the new version manager.
- Monitoring component: This component maintains a database that stores the metrics of each version manager.

For READ and WRITE/APPEND primitives, we use the same steps presented in Section 2.2 except in the case when it is necessary to contact the version manager. In this case, the client must extract the version manager ID from the Blob ID and then connect to the version manager where the created Blob is stored.

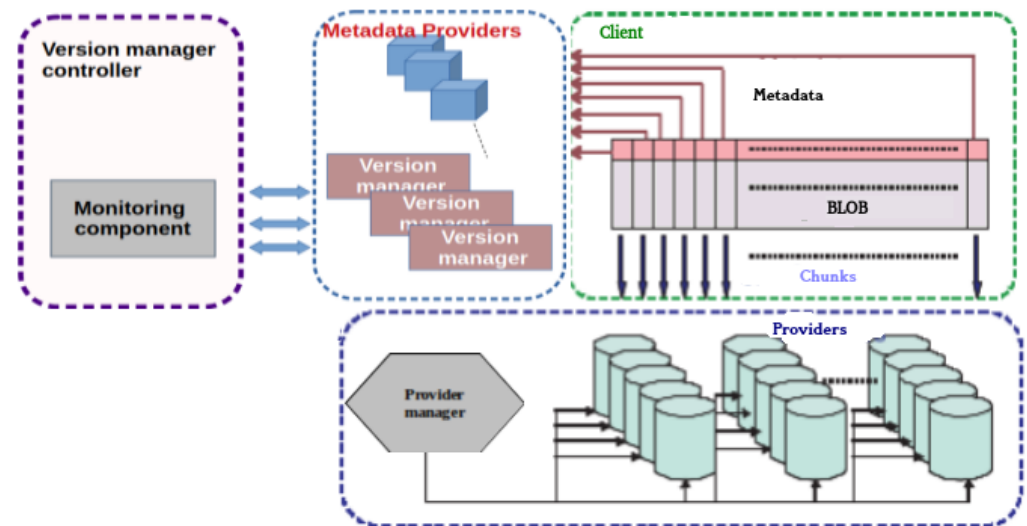


Figure 5. New proposed BlobSeer architecture.

5. Experiments and Discussion

In order to highlight that our proposal is scalable, we need to specify two groups of version managers: one group of active version managers, and another of inactive version managers. More specifically, we will specify, in a configuration file, the identifier of each version manager and the port number, and, for the name (@IP), it must be assigned when we run our programs on the testbed platform (IBNBADIS cluster) using a shell script.

5.1. Adding a New Version Manager

1. Our system starts with a number of version managers that is equal to the number of version managers existing in the group of active version managers defined in the configuration file.
2. Each version manager sends its physical characteristics to the version-manager controller using ApMon [19].
3. The version-manager controller receives these physical characteristics using the appropriate filters and then saves them in the Postgres [20] database (for each version manager it will create a table whose name corresponds to the name of the physical server).
4. Periodically, the version-manager controller will calculate its score using a heuristic based on CPU utilization rate, bandwidth utilization and busy storage space (RAM and Disk), according to Equation (1).

$$Scor_i = \alpha * CPU_i + \beta * BW_i + \gamma * RAM_i + \theta * Disk_i \quad (1)$$

Their values are

- $\alpha = 0.35$.
- $\beta = 0.25$.
- $\gamma = 0.2$.
- $\theta = 0.2$.

These weights can be changed in the configuration file. In our case, we give higher priority to CPU and network due to their importance in distributed storage workloads.

5. If the score exceeds a predefined threshold:

Threshold = 0.75 (if $Scor_i \geq Threshold$, VMi is considered OVERLOADED) the version-manager controller will remove a version manager from the group of inactive version managers; if 60% of the existing version managers are overloaded (their workload exceeds a predefined threshold (0.75)), as expressed in Equation (2),

$$\frac{\text{Count}(Scor_i \geq 0.75)}{\text{Total Active VMs}} \geq 0.6 \quad (2)$$

Then, it adds it to the group of active version managers, after launching a Python script to start this new version manager. Since the configuration file has been modified, the version-manager controller will send the modified configuration file to the following system components: version managers and data providers.

All experiments presented in this paper were obtained using the IBNBADIS High Performance Computing platform (www.ibnbadis.cerist.dz, accessed on 1 July 2026) offered by the Research Center on Scientific and Technical Information - CERIST (Algeria). This platform uses the Slurm Workload Manager (known as Simple Linux Utility for Resource Management or Slurm). Slurm is a free and open-source job scheduler for Linux and Unix-like kernels and is used by many of the world's supercomputers and computer clusters. So, to reserve nodes with Slurm, we use a shell script to specify the following parameters:

- The version-manager controller.
- Active version managers.
- Inactive version managers.
- Provider manager.
- Metadata providers.

To conduct our tests, the configuration was selected based on a balance between demonstration requirements and resource constraints. We used 11 nodes (1 version-manager controller, 2 active version managers to validate the scalability and load distribution benefits of the proposed architecture, and 3 inactive version managers to test the autonomous scaling and fault recovery mechanisms, 4 data and metadata providers and 1 provider manager). Each primitive execution is a client (number of primitives = number of clients). The details of the experiments are summarized in Table 3.

Table 3. Experimental environment specifications.

Parameter	Specification
Number of nodes	11
Number of processor cores	22
RAM per node	32 GB
Operating system	bullx Linux Server release 6.3 (V1)
BlobSeer version	blobseer-1.2.1

5.1.1. CREATE Primitive Response Time: A Comparative Study Between Standard and Multi-Version Manager Deployments

Our first experiment concerned the use of several CREATE primitive benchmarks (in a CREATE primitive, the client program will choose a random version manager and will contact it to create a new Blob). The numbers of the CREATE primitives in the benchmarks are fixed as follows (the first number (82) was chosen because it provides a low-concurrency baseline with minimal contention, and each subsequent number is obtained by successively multiplying the previous number by 2):

- 82 CREATE primitives.
- 126 CREATE primitives.
- 252 CREATE primitives.
- 504 CREATE primitives.
- 1008 CREATE primitives.
- 2016 CREATE primitives.

CREATE primitives are executed in parallel. Our experiments were carried out according to two approaches: (i) approach based on the standard BlobSeer that uses only one version manager and (ii) approach based on our contribution, which uses several version managers (we take 2 and then 3 version managers). Response time (execution time) is an important metric for evaluating scalability. We did this experiment nine times and then took the average of the obtained results (for reducing random error and ensure the reliability and reproducibility of the results). The benchmarks used are of a general nature; they are not dedicated to a specific field of application. The results of this experiment are presented in Figure 6: After analyzing the results, we can see the following:

- In the same standard BlobSeer environment, we study three scenarios: 1 version manager, 2 version managers and 3 version managers. After execution, we remark that, as the number of primitives on the system increases, the execution time increases.
- In running the first three benchmarks of successive size (82 CREATE primitives, 126 CREATE primitives and 252 CREATE primitives), we saw that there was not a big difference in terms of execution times for these primitives in the same experiment, either the first (with the standard BlobSeer) or the second (our proposed model: BlobSeer with two version managers and BlobSeer with three version managers). In the latter, the third version manager is launched by the version-manager controller following the execution of benchmark 3 (252 CREATE primitives). This difference appeared significantly in the last three benchmarks (504 CREATE primitives, 1008 CREATE primitives, and 2016 CREATE primitives) containing a large number of primitives because the execution time of primitives increases with the increase in the number of primitives (the more the load increases, the more the execution time increases).
- Comparing the execution time of six benchmarks between the three (the standard BlobSeer; our proposed model, BlobSeer with two version managers; and the BlobSeer with three version managers), we note that between BlobSeer standard and the BlobSeer with two version managers, the time for the six benchmarks in the second case is less than the times using BlobSeer standard. Almost half this difference in time is remarkable and growing in the last three benchmarks (504 CREATE primitives, 1008 CREATE primitives, and 2016 CREATE primitives), where the number of primitives is large; running benchmark six in the BlobSeer with two version managers saves more execution time than running this benchmark in the standard BlobSeer and so on for the other benchmarks. This remarkable time saving in the execution of benchmark six is due to the heavy workload of resources in the standard BlobSeer due to the execution of the large number of primitives, which total 2016. This time saving for the BlobSeer with two version managers compared with the standard BlobSeer is explained by the load on the CREATE primitives: instead of being executed by a single version manager, it is distributed between the two version managers, which makes it possible to exploit the available resources of the two version managers. Comparing the results of the BlobSeer with two version managers and the BlobSeer with three version managers, we note that the total execution time using three version managers is less than the time given by two version managers; this time saving is more significant when running benchmark six, which contains a larger number of primitives

(2016 CREATE primitives) than benchmark 5 (1008 CREATE primitives). The same applies to the other benchmarks (504 CREATE primitives and 252 CREATE primitives). For the two benchmarks (126 CREATE primitives and 82 CREATE primitives), the execution time is almost the same because (for the BlobSeer with three managers), in the execution of these two benchmarks, the third version manager was not launched by the version-manager controller. This time saving is explained by the loading of the primitives, which, instead of being executed by two version managers (the BlobSeer with two version managers), is distributed between three version managers (the BlobSeer with three version managers). Comparing the results obtained using the standard BlobSeer and the BlobSeer with three version managers, we see that in all the benchmark executions (82 CREATE primitives, 126 CREATE primitives, 252 CREATE primitives, 504 CREATE primitives, 1008 CREATE primitives, and 2016 CREATE primitives), the BlobSeer with three version managers gives a smaller total execution time compared with execution in the standard BlobSeer. This is due to the loading of primitives. Instead of being executed by a single version manager, it is distributed between the three version managers by exploiting the available resources of the three version managers.

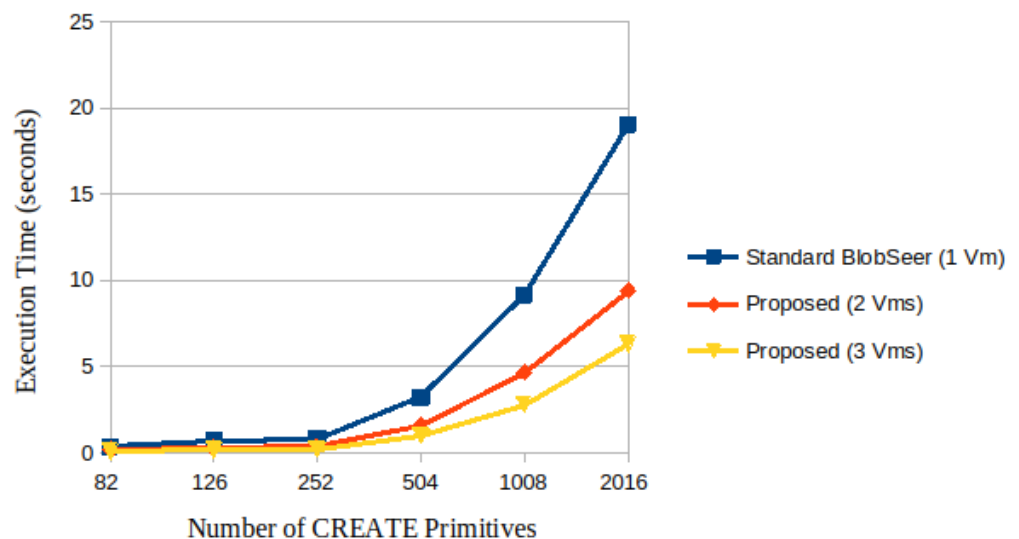


Figure 6. Execution time of number of CREATE primitives vs. number of version managers.

5.1.2. WRITE Primitive Response Time: A Comparative Study Between Standard and Multi-Version Manager Deployments

Our second experiment concerns the execution of 89 WRITE primitives in parallel (where each primitive WRITE corresponds to an independent client). We have chosen this number to moderate the high level of concurrent WRITE primitives. When running it in the standard BlobSeer, all WRITE primitives are handled by only one version manager. The first half concerns the primitives writing in Blobs created by version manager_1 and the other half concerns the writing in Blobs created by version manager_2. When running in the BlobSeer with three version managers, a third of the primitives correspond to Blobs created by version manager_1, the second third concerns Blobs created by version manager_2 and the third concerns writing in Blobs created by the version manager_3, and will take the execution time for each WRITE primitive. We find that the execution time varies from one application to another (see Table 4). When we compare the obtained results, as shown in Figure 7, we find that the execution time for the WRITE primitive using three version managers is always less than the execution time with two version managers, and the execution time using two version managers is small compared to that of BlobSeer

standard. This time difference is remarkable for the execution of WRITE primitives from primitive 13 to primitive 78. These results can be explained by these key factors. First, the standard BlobSeer architecture serializes all WRITE requests through a single version manager, causing queue depth to grow with concurrency. With two or three VMs, the load is distributed, reducing queue depth per VM and, consequently, reducing the average waiting time. Second, the most significant time savings are observed between primitives 13 and 78, which corresponds to the peak concurrency window where queuing delays in the standard architecture grow most rapidly.

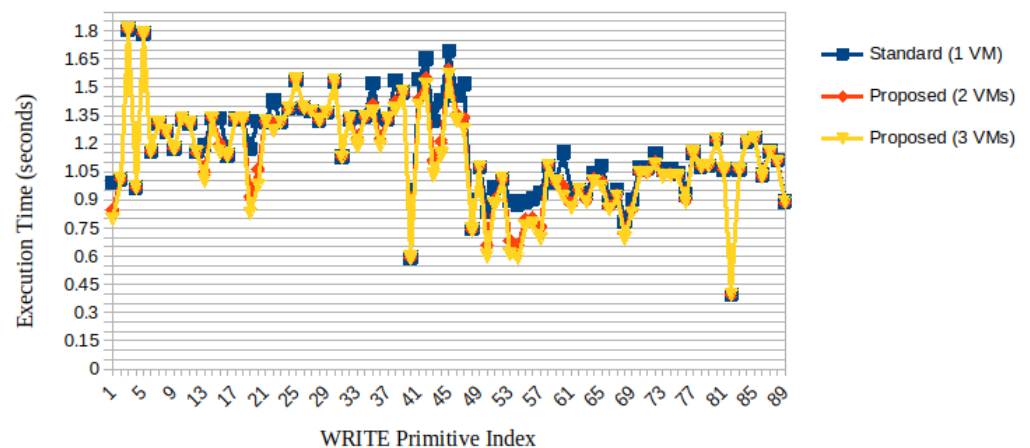


Figure 7. Execution time for each WRITE primitive vs. number of version managers.

Table 4. WRITE performance comparison: Standard BlobSeer vs. proposed architecture.

Metric	Standard (1 VM)	2 VMs	3 VMs
Max WRITE Execution Time	0.393867 s	0.393562 s	0.393460 s
Min WRITE Execution Time	1.808079 s	1.807828 s	1.807744 s
Average VM Access Time	1.172283 s	1.129037 s	1.126404 s

To compute the execution time for a WRITE primitive (ETW), we use the following equation: $ETW = T1 + T2 + T3 + T4 + T5$, where

- T1 is the time to receive the providers' list from the provider manager.
- T2 is the time to write the pages at the level of the data providers.
- T3 is the time needed to request a new version (connect to the version manager).
- T4 is the write time of the metadata.
- T5 is the time to report a success of the WRITE primitive to the version manager.

At this point, we recall that our contribution is the use of several version managers instead of the standard BlobSeer architecture that uses only one version manager. To highlight the effect of this contribution, we need to compute the total time to access the version manager; this total time is the sum of two times: time to request a new version (connects to the version manager) + time to signal a successful operation of the WRITE to the version manager. We see that the total time to access the version manager to perform a WRITE primitive changes from one operation to another (see Figure 8). This difference in time is remarkable in the execution of these primitives: primitive 1, the primitives from primitive 13 to primitive 23 and primitive 32 to primitive 78. First of all, the difference in execution times for primitive 1 is explained by the fact that primitives are executed in parallel, so it is not necessarily that it is the first primitive to execute; and if we assume that it is the first, this is due to the total time taken to connect to the version managers to execute a WRITE primitive = time taken to request a new version (connection with the version

manager) + time taken to report a successful write operation to the version manager. So, even though it is the first one that requests a new version, this occurs before reporting a successful write operation to the version manager. During this time, there is the metadata writing stage (the client sends the metadata created to the metadata providers and waits for acknowledgement from the metadata providers). Logically, during this time, there are other requests that will be executed or will be waiting to be executed, either commands for a new version (connection with the version manager) or commands to signal a successful write operation to the version manager and wait for these acknowledgements. For the execution time of the other primitives, this difference represents the variation in the version manager load. We can, therefore, conclude that «As the load (primitives or requests executed by the version manager) increases, so does the total time taken to access the version manager (or to the version managers) and vice versa in the standard BlobSeer». Table 5 presents the total time required to access the version manager to perform a WRITE primitive. The results show a consistent improvement as the number of version managers increases. The average access time decreased from 0.098872 s (BlobSeer standard) to 0.049682 s (BlobSeer with two VMs) and 0.033285 s (BlobSeer with three VMs), which represent a 66% reduction with three VMs, compared to the standard architecture. The maximum access time shows an even more significant improvement, dropping from 0.515177 s to 0.172015 s (66.6% reduction). This improvement is attributed to the distribution of WRITE requests across multiple VMs, which reduces the queue depth and contention for version assignment. The reduction in execution time from two VMs to three VMs is relatively small compared to the reduction from the standard BlobSeer (one VM) to three VMs. This is because the workload consists of only 89 parallel WRITE primitives. With two VMs, the queue depth and latency are already low. Distributing these 89 WRITES across three VMs instead of two provides only a marginal additional benefit as the queuing delay is not the dominant factor in this workload.

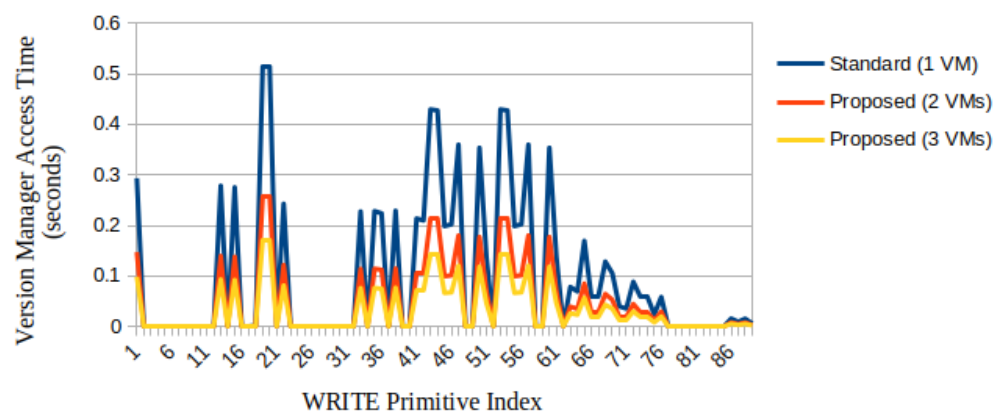


Figure 8. Total time to connect to the version manager to fulfill a WRITE primitive vs. the number of version managers.

Table 5. Total time to access the version manager to perform a WRITE primitive.

Metric	Standard (1 VM)	2 VMs	3 VMs
Minimum Access Time	0.000952 s	0.000682 s	0.000566 s
Maximum Access Time	0.515177 s	0.257805 s	0.172015 s
Average Access Time	0.098872 s	0.049682 s	0.033285 s

5.1.3. READ Primitive Response Time: A Comparative Study Between Standard and Multi-Version Manager Deployments

Our third experiment concerns the execution of 89 READ primitives in parallel (where each primitive READ corresponds to an independent client). We have chosen this number to moderate the high level of concurrent READ primitives.

- When running in the standard BlobSeer, all READ primitives are handled by only one version manager.
- When running in the BlobSeer with two version managers, half of the primitives will be for reading in Blobs created by the version manager_1 and the other half for reading in Blobs created by the version manager_2.
- When running in the BlobSeer with three version managers, a third of the primitives concern Blobs created by version manager_1, the second third is concerned with reading in Blobs created by version manager_2 and the third is for reading in Blobs created by version manager_3. Our third experiment is the same as the second one except we will change WRITE primitive to READ primitive.

If we compare the results obtained in the three cases (BlobSeer standard, BlobSeer with two version managers and BlobSeer with three version managers), as shown in Figure 9 and Table 6, we find that the time to complete each READ primitive with three version managers is smaller than with two version managers; also, the time taken with two version managers is smaller than with the one standard BlobSeer. The long execution times of the READ primitives are indicated by the execution of primitive 47 to primitive 80, which indicates a period of high contention on the metadata providers and data providers, not on the version managers. To compute the time to achieve a READ primitive, we use the following equation: READ primitive execution time = time to access the version manager (version managers) to return the latest version (getroot) + time to access the metadata providers to read the metadata + time of reading pages. Given this equation, we are interested in the time to access the version manager. Figure 10 shows the variation in this time in the three cases: BlobSeer standard, BlobSeer with two version managers and BlobSeer with three version managers. Through these results, summarized in Table 7, we note that the access time to the version manager, to return the latest version of the Blob, varies from one case to another; it shows a consistent improvement as the number of version managers increases. The average access time drops from 0.000011 s (standard) to 0.000004 s (three VMs), representing a 63.6% reduction. The maximum access time shows even significant improvement, decrease from 0.000034 s to 0.000011 s (67.6% reduction). These major times are seen in the execution of the following primitives: from primitive 8 to primitive 13, primitive 59, primitive 69, and primitive 74. This is explained by the parallel execution of the primitives and the variation in the load of the version manager (or version managers), since when the load (getroot requests) increases, the total access time to the version manager (or version managers) also increases, and vice versa. This improvement is attributed to the distribution of READ root requests across multiple VMs, which reduces the queue depth and contention for metadata root access. However, the absolute gain is only in the order of microseconds as READ operations are inherently lightweight. The overall execution time for READ primitives remains controlled by metadata provider access and data transfer, which are independent of the number of version managers. Across both experiments two and three, we observe that the time savings for executing the WRITE primitive are greater than the time savings for executing a READ primitive (using two and three version managers). We can explain this time saving because a WRITE primitive requires serialization to accomplish this primitive, and we communicate twice with the version managers, making the version manager the bottleneck, once when requesting a new version and the other time to signal the success of the write operation. Meanwhile, READ

primitives do not require serialization (they are already parallelized), and we communicate only once to retrieve the root of the metadata tree corresponding to the Blob.

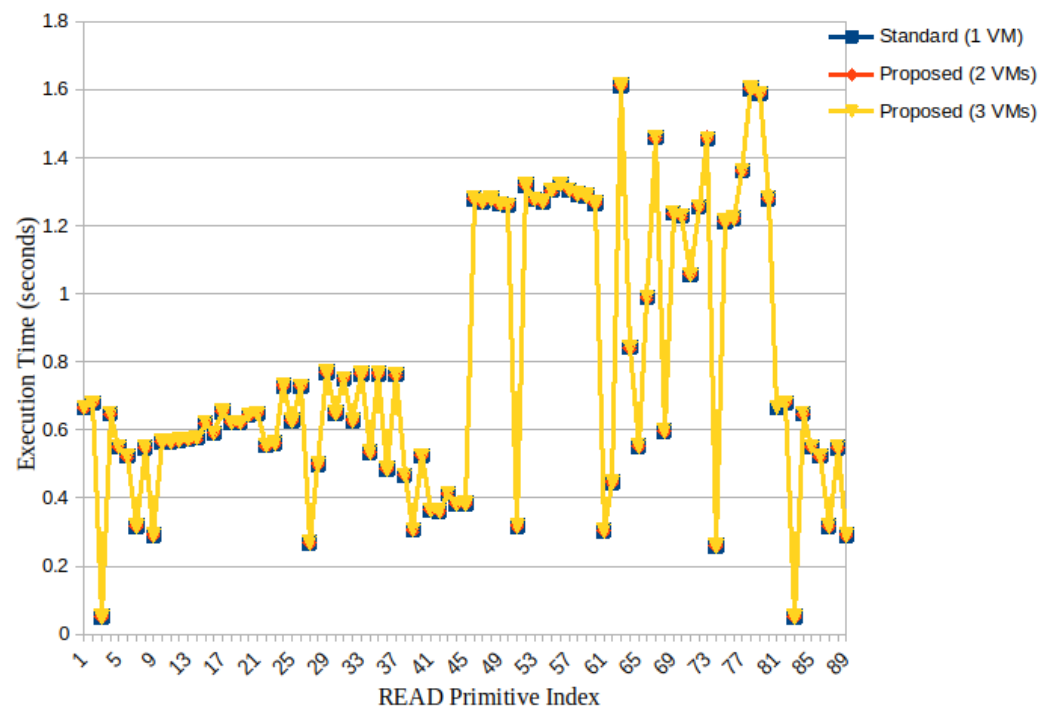


Figure 9. Execution time of each READ primitive vs. number of version managers.

Table 6. Total execution time of READ primitives.

Metric	Standard (1 VM)	2 VMs	3 VMs
Minimum Execution Time	0.051798 s	0.051795 s	0.051795 s
Maximum Execution Time	1.612161 s	1.612157 s	1.612155 s
Average Execution Time	0.776553 s	0.775294 s	0.775292 s

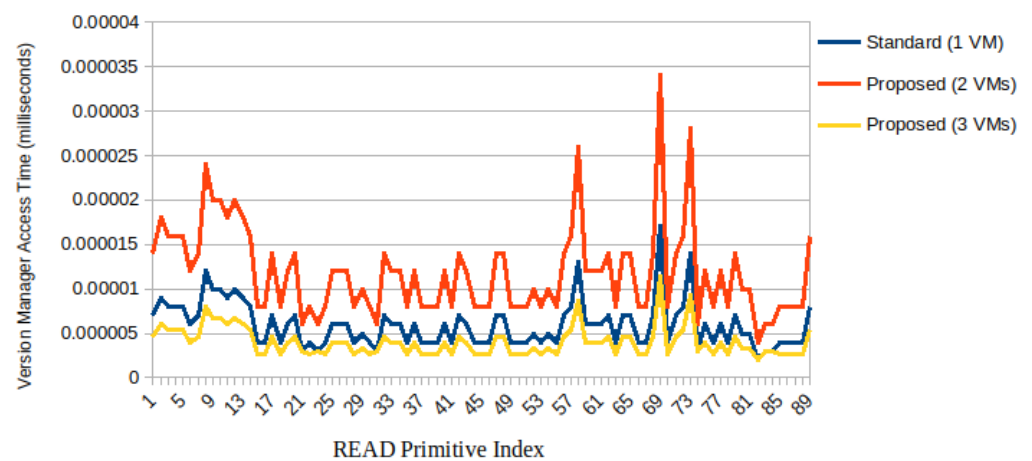


Figure 10. Time to connect to the version managers to fulfill a READ primitive vs. the number of version managers.

Table 7. Total time to access the version manager to perform a READ primitive.

Metric	Standard (1 VM)	2 VMs	3 VMs
Minimum Access Time	0.000004 s	0.000002 s	0.000002 s
Maximum Access Time	0.000034 s	0.000017 s	0.000011 s
Average Access Time	0.000011 s	0.000005 s	0.000004 s

5.1.4. Horizontal Scalability Validation for Mixed Workloads

To validate the horizontal scalability of our multi-version manager architecture, we conducted an experiment where the number of active version managers (k) was incrementally increased from 1 to 12, while monitoring system throughput and average latency under two distinct workload profiles:

- The first workload A (CREATE-Only): 100% CREATE primitives. This workload is perfectly parallelizable because new Blobs can be distributed across all active VMs.
- The second workload B (Mixed): 50% CREATE and 50% WRITE. This represents real-world scenarios where WRITES are pinned to their creator VMs.

Table 8 illustrates the results. For the CREATE-only workload (A), the system exhibits near-perfect linear scalability up to $k = 6$, with throughput increasing proportionally to k (slope approximately 495 $((1470-480)/(3-1))$ ops/sec per VM) and latency dropping hyperbolically from 185 ms to 0.5 ms. This confirms that the version-manager controller successfully distributes metadata allocation requests without introducing contention. For mixed workload (B), throughput scales poorly. It grows from 480 to 1380 ops/sec up to 6 VMs, but then stays at about 1450 ops/sec beyond 8 VMs. This happens because WRITES are tied to the VM that created the Blob (enforced by the Blob ID structure in Section 3.1). Since new VMs cannot take over these existing WRITES, the original VMs stay overloaded, causing an unavoidable wait time.

Table 8. Horizontal scalability test results: throughput, latency, and Vmc overhead under CREATE-only (Workload A) and mixed (Workload B) workloads.

Number of VMs (k)	Throughput (A) (ops/s)	Latency (A) (ms)	Throughput (B) (ops/s)	Latency (B) (ms)	Vmc CPU Overhead
1	480	185.0	480	185.0	0.2%
2	980	4.2	780	12.0	0.3%
3	1470	1.8	1050	8.5	0.4%
4	1960	0.9	1220	7.2	0.5%
6	3000	0.5	1380	6.8	0.7%
8	3000	0.5	1420	6.7	1.2%
10	3000	0.5	1450	6.7	2.5%
12	3000	0.5	1460	6.7	5.0%

Workload A: 100% CREATE operations and workload B: 50% CREATE + 50% WRITE operations.

5.1.5. Resource-Overhead Analysis

To evaluate the resource overhead of our multi-version manager architecture, we conducted three experimental scenarios, comparing the standard BlobSeer (single VM) and our proposed architecture (BlobSeer with three version managers). Table 9 describes the three scenarios:

Analysis of Results

After analyzing the results obtained in Table 10, the key findings are discussed below.

1. CPU Overhead: The proposed architecture adds a 3% aggregate CPU overhead (from 95% to 98% across all nodes). Several factors contribute to this overhead:

- Heartbeat generation: each VM sends a heartbeat to the version-manager controller.
 - Controller monitoring loop: The version-manager controller computes the load for 3 VMs every 10 s.
 - ApMon parsing: Serializing/deserializing the metric payloads (sending metrics to version-manager controller)
 - Critical Insight: While total cluster CPU increases slightly, the per-node CPU drops from 95% to 31%. This means no single node is saturated and this redistribution eliminates the single-point bottleneck and enables the system to handle burst loads without performance degradation.
2. Memory Overhead: The version-manager controller's PostgreSQL database stores historical metrics with the distributed metadata caches.
 3. Disk I/O and Storage: Total input/output operations per second drop from 1500 to 1240 (a 17% reduction). This is because
 - I/O is spread across three disks (three Vms), reducing queue depth and rotational latency.
 - The version-manager controller adds IOPS for logging.
 - Trade-off: Disk storage increases from 5.2 GB to 5.9 GB due to the version-manager controller's metric retention.

Table 9. Experimental scenarios for resource-overhead evaluation with three version managers.

Scenario	Active VMs	Duration	Workload	Purpose
S1: Idle Baseline	3	10 min	No requests	Measure static overhead (heartbeats, monitoring, Postgres DB).
S2: Moderate Load	3	30 min	800 ops/sec (400 CREATE + 400 WRITE)	Measure resource consumption under normal conditions.
S3: High Load	3	30 min	1800 ops/sec (900 CREATE + 900 WRITE)	Observe resource consumption at maximum throughput, determine where bottlenecks arise, and measure the impact of resource contention.

Table 10. Resource overhead comparison: Standard BlobSeer (One VM) vs. the proposed multi-version manager architecture (Three VMs + Version-Manager Controller).

Resource	Metric	Standard	Proposed architecture (3 VMs + Vmc)	
		1 VM	Per Component	Delta
CPU	Avg. per-core utilization (User + Sys)	95%	VM ₁ : 32%, VM ₂ : 30%, VM ₃ : 31%, Vmc: 4%	+3% (96% total)
CPU	Peak system CPU (context switches)	12%	VM avg: 4%, Vmc: 2%	−50% (per node)
Memory	Version manager process memory	1.9 GB	600 MB/VM (1.8 GB) + 150 MB DB	+2.6% (1.95 GB)
Memory	Heap fragmentation/GC overhead	High	Low (smaller heaps)	−70% (per-node pause)
Disk I/O	Avg. read/write IOPS	1500 IOPS	VM ₁ : 400, VM ₂ : 380, VM ₃ : 410, Vmc: 50	−17% (1240 IOPS)
Disk I/O	Avg. write throughput (MB/s)	45 MB/s	VMs: 15 MB/s (45), Vmc: 0.5 MB/s	+1.1% (45.5 MB/s)
Disk Storage	Total space used	5.2 GB	VMs: 1.7 GB (5.1), Vmc DB: 0.8 GB	+13.4% (5.9 GB)
Disk Storage	Version log file size per Blob	Large	Smaller per VM, fragmented	−40% (per-file)

Note: VM = version manager; Vmc = version-manager controller; and Vmc DB = version-manager controller database.

5.1.6. Fault Tolerance Evaluation with Three Version Managers

Our fourth and final experiment was carried out to assess the availability of the BlobSeer system, taking into account our contribution. We have used two scenarios:

1. For the first scenario (simple scenario): When stopping a version manager in the standard BlobSeer, all primitives (WRITE or READ) are canceled. Using two version managers, if we stop, for example, version manager₁, 50% of the primitives (WRITE in the second experiment or READ in the third experiment) concern the writing or reading of the Blobs created by version manager₁ that are canceled and 50% concern the writing or reading of the Blobs created by the version manager₂ that are completed successfully. Using three version managers, if we stop, for example,

version manager_1, 34% of the primitives (WRITE in the second experiment or READ in the third experiment) concern the writing or reading of the Blobs created by version manager_1 that are canceled and 66% of the primitives concern the writing or reading of the Blobs created by version manager_2 or version manager_3 that are successfully completed.

2. Complex Fault Scenarios: To do our evaluation, we have used three active version managers (VM₁, VM₂ and VM₃) and two inactive ones (VM₄ and VM₅). We have used mixed workload (50% CREATE and 50% WRITE) at 2000 ops/sec for CREATE and WRITE distribution (each VM handles 333 CREATE ops/sec and 333 WRITE ops/sec).
 - (a) Scenario A: The failure affects the version-manager controller + VM₁. At $t = 10$ s, kill the version-manager controller and VM₁.
 - (b) Scenario B: Two VMs' failure. At $t = 10$ s, kill VM₁ and VM₂. Only VM₃ survives.

Table 11 summarizes these scenarios and the metrics taken to evaluate the fault tolerance. (Failover metrics: we have taken the detection time of the failure component, the time to elect a new version-manager controller and the failover time for the service impact, throughput drop, peak latency and availability.)

Table 11. Fault tolerance evaluation: performance degradation and recovery metrics under various failure scenarios (three active VMs).

Fault Scenario	Affected Components	Failover Metrics			Service Impact		
		T_{det}	T_{elec}	T_{fail}	D	Peak Lat.	Avail.
No Fault	–	–	–	–	0%	12 ms	100%
Single VM Failure	VM ₁	30 s	0 s	30 s	33%	45 ms	67%
Vmc + VM ₁	Vmc, VM ₁	30 s	0.8 s	31 s	33%	85 ms	67%
Vmc + Majority VM Failure	Vmc + VM ₁ , VM ₂	30 s	0.8 s	32 s	66%→50%	150 ms	33%→50%

Note: Vmc = version-manager controller, T_{det} = detection time, T_{elec} = election time, T_{fail} = failover time (total time to recovery), Peak Lat = peak latency, D = throughput drop (D), and Avail = availability (A) are shown as initial → post-recovery where applicable.

Analysis of Fault-Tolerance Results

For the scenario with No Fault, the system operates at peak performance. We take this as the reference point for measuring degradation under other fault scenarios.

1. Single VM Failure: The failure of one VM1 results in a 33% throughput drop, which is proportional to the lost capacity (1/3 (one out of three VMs)). This confirms that the fault is contained and does not propagate to other VMs.
The detection time of the failure Vm (VM1) is 30 s because it is dominated by missed heartbeats. No election is required because the version-manager controller is still alive. For this, T_{elec} is 0 s. Availability drops to 67% because all Blobs created by VM1 become inaccessible (unwritable). However, the remaining 67% of the system remains fully operational, and peak latency rises to 45 ms (from 12 ms) due to clients retrying connections to the failed VM and, after detecting the failure of VM1, they connect to the surviving VMs.
2. Version-manager controller + VM1 failure: This scenario represents the double failure: the two components fail simultaneously. The 33% throughput drop matches the single VM failure scenario, confirming that version-manager controller failure does not amplify the capacity loss, and it is not a single point of failure. The ring election algorithm completes in 0.8 s. The system simply loses access to VM1's Blobs. The election time of 0.8 s is remarkably low, confirming that the ring election algorithm converges quickly. The failover time increases to 31 s because the time for the election

a new version-manager controller is added, along with the time taken to propagate the new configuration file with the new version-manager controller to the list of active VMs. Peak latency is 85 ms, which is higher than the single VM failure case (45 ms). This is because clients also experience connection timeouts to the version-manager controller while the election is in progress.

3. Failure of two VMs + Vmc: This scenario causes an immediate 66% throughput collapse. However, the system autonomously recovers up to 50% capacity within 32 s by activating a new VM (VM₄) from the inactive pool. This demonstrates graceful degradation and the system recovers autonomously, even under majority failure conditions. (This demonstrates that while WRITES for the dead VMs are permanently lost, the system gracefully degrades, and the new activated VM treats CREATE/WRITE operations on new Blobs.)

6. Limitations of the Proposed Architecture

While the proposed multi-version manager architecture demonstrates significant improvements in scalability, partial fault tolerance, and resource distribution, it also introduces several inherent limitations. Acknowledging these limitations is essential for providing a balanced assessment and establishing a clear roadmap for future research. The reliance on a centralized version-manager controller increases coordination overhead compared to the standard BlobSeer architecture (which uses a single version manager). Furthermore, the architecture introduces additional configuration parameters, including load score weights, overload thresholds and scaling ratios, which increase global resource overhead. In addition, the system provides only partial fault tolerance, not full data durability. If a version manager fails, all Blobs created by that VM become inaccessible. These trade-offs must be carefully considered.

7. Conclusions

Concurrent access is one of the challenges in distributed data-management systems, and more specifically in Cloud Computing environments. Versioning is the most widely used approach for managing distributed file storage systems. Its main advantage is that it allows high concurrency access. This approach is used by many systems, such as HDFS, BlobSeer, XtremFS and RStore. The existence of several versions of the same data offers the possibility of making multiple accesses in parallel to the different versions. However, to maintain the consistency of these different versions, synchronization is required. If this synchronization is carried out by a single manager, some properties can no longer be ensured, such as performance, waiting time, scalability, overloading, and fault tolerance. To avoid the limits of a single manager, we propose the use of multi-version managers and a version-manager controller. Our approach includes an autonomous scaling mechanism based on a heuristic load metric and a ring election protocol for partial fault tolerance. This approach has been tested on the distributed versioning-orientated data-management service BlobSeer and authorizes concurrent access (READ/READ, READ/WRITE, and WRITE/WRITE). Our experiments confirmed that our proposed architecture achieves better execution times. For the CREATE primitive, the system achieves near-linear scalability. For the WRITE primitive, the average access time decreased from 0.098872 s (BlobSeer standard) to 0.033285 s (BlobSeer with three VMs), which represents a 66% reduction. The improvement for READ primitives is less significant compared to WRITE primitives. The average version manager access time decreases from 0.011 ms (standard BlobSeer) to 0.004 ms with three version managers, which is a 63.6% reduction. In terms of fault tolerance, our proposed architecture ensures partial fault tolerance. The system maintains 67% availability under single-version-manager failures. The resource-overhead results show a

modest 3% increase in global cluster CPU, while per-node CPU utilization drops from 95% to 31%, eliminating the single-node bottleneck.

As future work, we plan to implement our approach in other large-scale data-management storage systems and explore decentralized monitoring. Since our approach provides only partial fault tolerance, we envisage extending this work to achieve full availability by replicating version managers. If a VM fails, WRITE operations on Blobs created by that VM would be redirected to its replica. To optimize the resource overhead, our future direction includes adaptive heartbeats.

Author Contributions: B.C. and Y.S. were involved in the entire process of producing this paper, including conceptualization, methodology, modeling, validation, visualization, and manuscript preparation. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Maizate, A.; Elomari, A.; Hassouni, L. The Main Characteristics of Five Distributed File Systems Required for Big Data: A Comparatively Study. *Adv. Sci. Technol. Eng. Syst. J.* **2017**, *2*, 78–91.
2. Varade, M.; Jethani, V. Distributed Metadata Management Scheme in HDFS. *Int. J. Sci. Res. Publ.* **2013**, *3*, ISSN 2250–3153. DOI: 10.29322/IJSRP.3.5.2013.p171197.
3. Shvachko, K.; Kuang, H.; Radia, S.; Chansler, R. The Hadoop Distributed File System. In Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST '10, Incline Village, NV, USA, 3–7 May 2010; pp. 1–10.
4. Nicolae, B.; Antoniu, G.; Bougé, L.; Moiseb, D.; Carpen-Amarie, A. BlobSeer: Next-generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.* **2011**, *71*, 169–184.
5. Nicolae, B. BlobSeer: Towards efficient data storage management for large-scale, distributed systems. Ph.D. Thesis, University of Rennes 1, Rennes, France, 2010. Available online: <https://tel.archives-ouvertes.fr/tel-00552271> (accessed on 1 July 2026).
6. Talluri, L.; Thirumalaisamy, R.; Kota, R.; Sadi, R.P.R.; KC, U.; Naha, R.; Mahanti, A. Providing Consistent State to Distributed Storage System. *Computers* **2021**, *10*, 23. <https://doi.org/10.3390/computers10020023>.
7. Hupfeld, F.; Cortes, T.; Kolbeck, B.; Stender, J.; Focht, E.; Hess, M.; Malo, J.; Martí, J.; Cesario, E. XtremFS: A case for object-based storage in Grids. *Concurr. Comput. Pract. Exp.* **2008**, *20*, 2049–2060.
8. Bhattacharjee, S.; Deshpande, A. RStore: A Distributed Multi-version Document Store. *CoRR* **2018**, arxiv:1802.07693.
9. Zhou, W. MSCFS-RP: A Colored-Petri-Net-Based Analysis Model for Master–Slave Cloud File Systems with Replication Pipelining. *Electronics* **2024**, *13*, 2852. <https://doi.org/10.3390/electronics13142852>.
10. Apache Software Foundation. Hadoop Distributed File System (HDFS) Architecture Guide, 2023. Available online: <https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html> (accessed on 9 June 2026).
11. Tran, V.T. Scalable Data-Management Systems for Big Data. <https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html> Ph.D. Thesis, École Normale Supérieure de Cachan - ENS, Cachan, French, 2013.
12. Soares, P.G. On Remote Procedure Call. In *Proceedings of the CASCON '92: Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative Research, November 1992*; RFC Editor: Wilmington, DE, USA, 1992; Volume 2, pp. 215–267.
13. Ananda, A.; Tay, B.; Koh, E. A Survey of Asynchronous Remote Procedure Calls. *ACM SIGOPS Oper. Syst. Rev.* **1992**, *26*, 92–109.
14. Nicolae, B. BlobSeer: Towards Efficient Data Storage Management for Large-Scale, Distributed Systems. Doctoral Dissertation, Université de Rennes 1, Rennes, France, 2010. Available online: <https://hal.science> (accessed on 1 July 2026)).
15. Stender, J.; Berlin, M.; Reinefeld, A. XtremFS—A File System for the Cloud. In *Data Intensive Storage Services for Cloud Environments*; Kyriazis, D., Voulodimos, A., Gogouvis, S.V., Varvarigou, T., Eds.; IGI Global: Hershey, PA, USA, 2013; pp. 267–285.
16. Stender, J.; Kolbeck, B.; Mikael, J.; Höggvist, J.; Hupfeld, F. BabuDB: Fast and Efficient File System Metadata Storage. In Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os, Incline Village, NV, USA, 3 May 2010; pp. 51–58. <https://doi.org/10.1109/SNAPI.2010.14>.
17. Tanenbaum, A.S.; van Steen, M. *Distributed Systems: Principles and Paradigms*, 2nd ed.; Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 2006.
18. Garcia-Molina, H. Elections in a Distributed Computing System. *IEEE Trans. Comput.* **1982**, *C-31*, 48–59.

19. Legrand, I.; Voicu, R.; Cirstoiu, C.; Grigoras, C.; Betev, L.; Costan, A. Monitoring and control of large systems with MonALISA. *Commun. ACM* **2009**, *52*, 49–55.
20. Ahmed, I.; Smith, G. *PostgreSQL 9.6 High Performance*, 2nd ed.; Packt Publishing: Birmingham, UK, 2017.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.