SkyStore: Unified Storage Across Clouds

Shaopu Song
University of California, Berkeley

Junhao Hu
University of California, Berkeley

Abstract

Applications are frequently spanning across multiple regions and cloud providers to reduce vendor lock-in, drive down cost, and improve resource availabilities, as seen in use cases like geo-distributed model serving, spot applications, and container distribution. However, this multi-cloud, multi-region approach introduces significant challenges in data management, including latency issues, increased operational costs, and the complexity of interfacing with diverse storage APIs from various cloud vendors. To address these challenges, we propose SkyStore, a cross-cloud unified object store built for integrating data management across clouds with minimal effort. SkyStore simplifies user interaction with data, abstracting away the complexities of the underlying storage infrastructure. It optimally manages data operations, including demand-driven data placement, efficient data movement, and robust consistency management. Our system demonstrates runtime optimization and cost efficiencies compared to existing academic and commercial solutions, while also ensuring high availability and fault tolerance on par with standard cloud object stores.

1 Introduction

In the rapidly evolving landscape of cloud computing, the world has become increasingly multi-region and multi-cloud. Geo-distributed and multi-cloud technology stack are utilized to reduce vendor lock-in, increase resource availability at lower cost, and enhance the performance of applications. Additionally, data sovereignty policies mandate organizations to manage customer data in specific regions and clouds.

In this landscape, most of the workloads are data-centric. These applications typically need to manage a variety of data types such as models, containers, and satellite data. For instance, model serving tiers often face the challenge of varying availability of specific instance types across different regions. As the required instance types for efficient training and serving can be readily available in different regions, the model often needs to be trained in one region, and later on moved to other regions for serving. Another example is container sharing. In such scenarios, users are often located in different geographical regions, pushing and pulling container images from different places. In satellite-based systems, it is also common to see large volumes of image data processed and transmitted daily from various global positions for analysis.

The applications mentioned above are characterized by their need to read and write from different regions and clouds. Typically, a mix of data sizes is involved, from small containers to large objects like models and datasets. The workload patterns are predominantly read-heavy, with high concurrency in reads and fewer writes. From the user perspective, these workloads often require fast, repeated reads, quick writes, and a balance of read-after-write and eventual consistency.

To satisfy the workload requirements, organizations today need to manually manage data placement and movement, and reason about them across varied localized namespaces. This is challenging as different cloud provider offers different APIs, storage, and network options, and each has different performance and cost trade-offs. Since different cloud providers have their own APIs, to integrate multiple cloud providers into a single system, a unified API layer that users can interact with needs to be established. Besides, the under-level data management system has to be aware of the multi-region and multi-cloud object storage to make decisions on data storing and fetching.

Another critical aspect is the substantial variability in cloud pricing models, depending on the geographical location of data storage and access patterns. For instance, the cost of storing data in the *azure:eastus2* region is notably lower, approximately half, than storing data in the *aws:sa-east-1* region. On the other hand, performance indicators like data access throughput and latency differ across different cloud regions. For example, a user operating from the *azure:southcentralus* region experiences a significantly faster data access rate – up to 80 times quicker – when retrieving data from *azure:canadacentral* as opposed to *gcp:asia-northeast3-a*. Thus, optimally navigating this complex landscape requires an understanding of both the provider's offerings, as well as the application usage patterns.

Naive data management approaches can be costly and inefficient. The simplest solution is to store data in a centralized cloud region and read and write from it. In this case, only one copy of the data is stored, so the storage cost is minimal. However, repeatedly moving data across regions incurs high egress cost and increase read and write latency. On the other hand, storing data in every region can bring down the data access cost and latency; but it increases storage costs and complexity in maintaining consistency between replicas. Furthermore, as workloads evolve, manual management becomes increasingly impractical.

In this work, we propose SkyStore, a system to address the data management challenges in a multi-region, multi-cloud setting. SkyStore proposes an easy-to-use, efficient, reliable, and cost-effective solution for managing data in a multi-region, multi-cloud context.

Usability: SkyStore offers an S3-compatible API, providing a unified namespace through virtual bucket and object abstraction. This simplifies the user experience by offering a consistent interface across different cloud environments.

Flexibile Data Management Policy: To cope with different workload requirements, we integrate multiple write and read polices to provide flexibility to the client side. This not only ensures data availability and access performance but also minimizes storage cost overheads.

Scalability, Availability, and Fault Tolerance: The architecture of SkyStore is built with scalability in mind. It separates control and data planes, allowing each layer to scale independently based on demand. This separation also contributes to the system's fault tolerance, ensuring that data availability and system operations are not compromised in the event of partial failures.

2 Background

Effective management of data across diverse cloud providers presents several challenges. These challenges are rooted in the inherent heterogeneity of cloud services as well as the variability of workloads.

Storage API Variability Each provider has its unique set of semantics (e.g., put, get, multipart-upload, versioning) that complicates the integration process. For example, for the put operation, we can call *put_object* to achieve this in AWS S3, but there is no *put_object* function in neither the Azure nor the GCS. For the Azure, it uses *put_block_blob* to put the object into the corresponding blob. For GCS, it uses *upload_-streamed_object* to reach the same affect. Similarly, AWS S3 provides *get_object* to fetch the object from buckets. However, in GCS, the *get_object* is used to fetch the object metadata. We need to call *download_stream_object* to obtain the streaming body of the object. The implementation details will be discussed in Section 3.

Cost Heterogeneity Cloud data pricing consists of three components: data storage, network usage, and request (operation) charges. [12]. Different cloud providers and regions have different pricing models for each of these dimensions. Storage cost refers to the expense incurred for the amount of data stored in the cloud. Egress cost, often termed data transfer cost, is the charge from cloud providers for moving or transferring data from the cloud storage where it was uploaded [7]. Request cost represents the fees for operations such as put, get, and delete requests made to the cloud storage service.

The storage class (e.g., standard, infrequent access, archival), the geographic region of the data center, the source and destination of the data transfer, as well as the pricing policies of the cloud provider will all influence the cost.

Bandwidth and Latency Heterogeneity Bandwidth variability highlights the difference in data transfer speeds across different cloud regions and providers. This variability can impact application performance, especially in data-intensive operations.

3 SkyStore Architecture

This section provides a comprehensive overview of the SkyStore architecture, including its structural and operational components. SkyStore integrates seamlessly with existing cloud storage systems, including S3, GCS, and Azure Blob Storage. It offers a familiar interface to both users and developers, ensuring compatibility and ease of use.

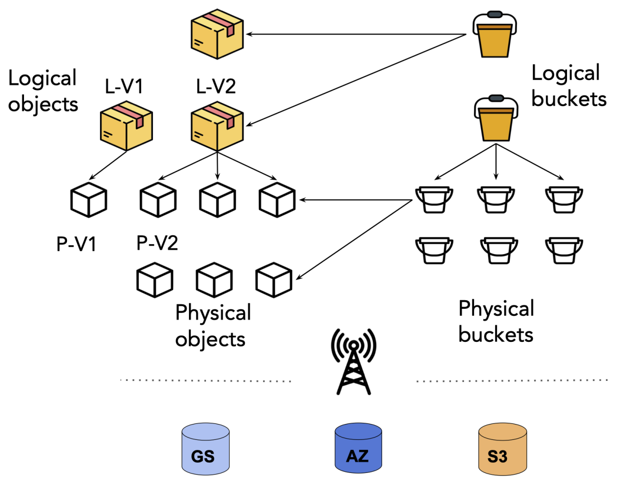


Figure 1: Metadata Architecture

3.1 Architecture Overview

SkyStore deploys as collection of VMs across multiple clouds, a client libraries, and a set of DB deployed in each cloud. The VM can be scaled horizontally to support variable load, DB is partitioned by key (scale horizontally). In each region, multiple metadata servers are deployed. Upon client initialization, a metadata server is dynamically chosen(currently random selection. However, alternative strategies exist, such as selecting the server with the lowest workload to act as the primary.) as the primary server from the available pool within the client's region. This primary server is responsible for overseeing and managing the metadata associated with the client's operations. The client's read and write requests pass through the data proxy, which acts as a stub and initiates requests to the control plane, where the control plane represents the connected metadata server. The control plane, guided by the user-defined policy, selects an appropriate cloud provider's cluster, and relays the outcome back to the data proxy. Subsequently, the data proxy then performs the actual read or write request to the respective cloud provider. The reason for not directly redirecting requests from the control plane to the machines of the corresponding cloud provider is to avoid introducing additional communication overhead between the control plane and the client during the process of reading or writing large data. Therefore, current approach mitigates the overhead on the control plane server.

3.2 API: Virtual Bucket / Object Abstraction

SkyStore utilizes an S3-compatible API, ensuring compatibility with standard operations typical in cloud storage environments. This compatibility extends to a range of operations, divided into object operations and bucket operations.

The system supports a range of object and bucket operations that are compatible with S3 API, including put, get, head, list, copy, and delete objects, and create, delete, list, and change version status on buckets. Multipart upload of large objects (e.g., create, list, upload part copy, complete, abort) is also supported.

3.3 Control Plane: SkyStore Server

The control plane of SkyStore, SkyStore server, serves as the central management hub responsible for handling and routing requests in a multi-region, multi-cloud environment.

Server Configuration and Database Integration The server of SkyStore is developed using FastAPI (integrate with Uvicorn) and validates requests with Pydantic schemas. It uses SQLAlchemy for database communication, following a typical Python web application structure. For database management, SkyStore supports SQLite and Postgres. It is implemented in 3.5K lines of code.

Concurrency Optimization For small workload with few concurrency, SQLite will be a better choice because of its easy deployment; for high-concurrency environments with numerous readers and writers, we employ Postgres for its strong performance. The SkyStore will automatically manage the setup and truncation of the database each time we start the system. Besides, we use connection pool provided by the sqlalchemy to improve the throughput. Based on our tests, the connection pool can bring 100M+ throughput increment to the system.

The SkyStore also support starting multiple workers (processes) of uvicorn to cope with high-concurrency environment. However, this optimization can bring concurrency bugs without careful management. We have several choices to solve the concurrency issues of multi-processes, including file-lock, shared-memory, message queue, or storing the variables into our metadata DB. We choose shared-memory methods to solve this problem finally because of its higher speed compared to other methods that including disk operations.

We used UltraDict [15], which wraps the shared memory mechanism, to solve the concurrency issues in both the database initialization and write/read policy loading when multiple uvicorn workers are used. When initializing the database, the 1st worker that trying to perform this task will allocate a memory region in shared memory to store the log file. By using the same shared memory name(key) of the 1st worker, the next coming workers can attach themselves to the same memory region. Then the 1st worker will exclusively lock the memory handler, write to the log and initialize the database. Other processes will never try to initialize the database again since they will find the log is already written.

The same operation is performed at the policy side, SkyStore will store the policy name set by users in the shared memory so that all the workers can observe the changes.

Logical and Physical Abstraction The control plane effectively manages both logical and physical representations of buckets and objects, as shown in Fig 1, a distinction that is central to SkyStore's architecture.

- *DBLogicalBucket*: Represents the user's perspective of a storage bucket. This class includes attributes like bucket name, prefix, and status. Each logical bucket is associated with multiple physical bucket locators (*DBPhysicalBucketLocator*).
- *DBPhysicalBucketLocator*: Details the actual location of bucket data, specifying cloud provider, region, and bucket information. This class reflects the physical presence of a bucket and its data in different clouds. It also includes attributes of the buckets like those in the actual object store.
- *DBLogicalObject*: corresponds to a logical view of an object stored in the cloud. This class captures essential

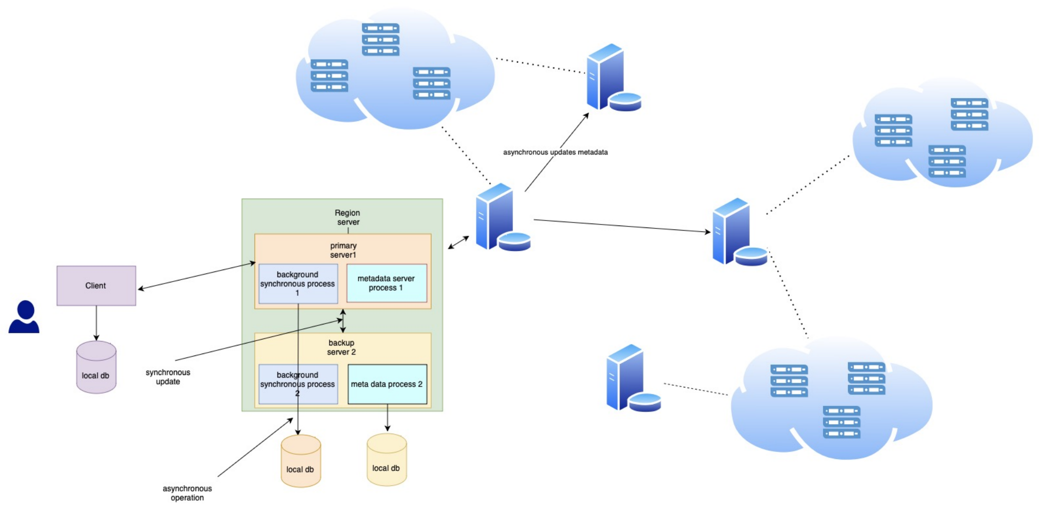


Figure 2: Metadata Server Architecture

metadata like size, modification time, etag, and status. It includes relationships to physical object locators (*DBPhysicalObjectLocator*), linking logical objects to their actual physical locations.

- *DBPhysicalObjectLocator*: Stores the precise location of objects, including cloud provider, region, and specific bucket and key details. It stores a comprehensive view of an object's physical state in the actual object store.

Additional Abstraction

- *DBMetrics*: Identifier of each PUT/GET operation passed to the control plane, containing latency of operations, $< key, size >$ of the objects processed, and the operation type together with request region and issued region. This information will sent by the clients side asynchronously to the control plane side by calling corresponding metrics recording FastAPI method, and collected into DB.

Multi-versioning Support SkyStore supports multi-version in compatible with S3 semantics. In the control plane side, we leverage the primary id of the logical object as its version number, and record the physical *version_id* in its corresponding physical object locators, which is generated by the under-level object storage. The control plane will behave based on the user input of version setting transferred from the data plane side, which includes three types, *SUSPENDED(S)*, *ENABLED(E)*, and *NULL(N)*. System behavior varies when the version setting is different.

When the version status is *N*, only one logical object with the same $\{DBLogical\_bucket, key, region\}$ identifier is allowed to exist in the system. Any attempt of uploading an object with the same identifier will be rejected. After enabling the version, users are allowed to perform the same PUT without upper bound, and each time a new logical object linked with the newly created physical objects of different version numbers will be built. To DELETE an object, user need to explicitly give the version number that ought to be deleted, otherwise a *delete_marker* will be added to the metadata server each time, which makes the DELETE operation performs the same as the PUT. We can suspend the version into *S* after enabling the multi-version setting. This will cause the metadata to be overwritten when a PUT/DELETE operation is attempted with the same identifiers.

There are several advantages of enabling multi-version. First, this allows SkyStore to change consistency level, which will be discussed in Section 5. Second, system can have higher throughput than a single version object storage, since we do not need to exclusively lock the database when the control plane are told by the data plane to try to create logical objects with the same identifiers at the same time.

Functionalities SkyStore Server maintains data catalog through the logical to physical object mappings. All object store requests are first routed to the control plane, which in terms manages policies that dictate how data is moved and stored. SkyStore Server will make decisions about which data should be placed in which cloud region and where to read data from based on the client region, and other factors like cost and performance.

3.4 Data Plane: S3-Proxy

The data plane within SkyStore is a S3-Proxy, a fully compatible web server that adeptly communicates using the S3 protocol with various cloud storage providers such as Azure Blob, AWS S3, and Google Cloud Storage (GCS). This component allows for seamless integration and interaction across different cloud environments.

Design One of the defining characteristics of the S3-Proxy in the SkyStore architecture is its stateless design. The proxy does not retain any internal state between different requests. This architecture allows the data plane to scale horizontally with ease.

The S3-Proxy in the SkyStore architecture handles a variety of operations as detailed in Section 3.2. These operations encompass reading, writing, listing, creating, and deleting objects and buckets in the cloud storage environment. The implementation of these functionalities within S3-Proxy follows a streamlined process. Upon receiving a request, the S3-Proxy consults the SkyStore Server for relevant policies. Based on the guidance received, it then performs the required actions using a client interface specific to each cloud provider. This approach ensures that operations like reading and writing data are executed in accordance with the location and policy directives provided by the server.

Implementation Overview S3-Proxy is implemented in 9K lines of code. It relies heavily on the s3s framework, a Rust-based tool that excels in performance, safety, and distribution. This framework takes Amazon's official S3 API schema and generates the entire suite of data types, error codes, and endpoints. Such an approach simplifies the development process, allowing developers to focus on application-specific logic without being bogged down by the intricacies of Amazon's XML outputs.

A key aspect of our implementation is the handling of the complexities inherent in global, distributed object stores. Modern object store APIs are multifaceted, featuring advanced capabilities such as versioning and multipart uploads. We have developed a client interface for each provider that is capable of interacting with their respective S3-compatible APIs. This interface allows for seamless integration with the varied and complex functionalities offered by contemporary cloud storage services. By managing these complexities, the S3-Proxy facilitates efficient and policy-compliant data operations across multiple cloud environments.

Multipart Upload Support SkyStore unified the multipart upload operation in AWS, Azure and GCS. To accelerate the uploading process, clients are allowed to perform parallel part uploading in SkyStore. Besides normal APIs used to perform multipart upload, SkyStore integrates *upload_part_-copy* provided by S3 to improve the speed of data migration by uploading parts from another source bucket rather than uploading from local storage.

When the uploading process is about to finish, user should notify the SkyStore to stop the uploading process. This signal will let object storage integrate the small parts into a large single object. AWS S3 allows composing up to 1,000 parts one time. However, the GCS only supports up to 32 parts being composed together per time. To solve this challenge, when performing multipart upload in GCS buckets, we iterate the composing process by composing 32 parts a time until the total number of objects (either existing before composing or the new large parts generated because of our iteration) is less than 32, and then delete the small parts remained in the bucket.

Multi-versioning Support SkyStore allows users change version setting directly in the proxy side. Like the cloud providers, the version number field will be set and returned back to the client side when versioning is enabled. We expose the logical object version id to the client rather than the real physical version ids. The version setting (*E,S,N*) will be sent to the control plane and record the setting in the database. Still, different behavior on cloud providers have to be coped with.

4 Fault Tolerance

The cost of abstracting away location details of data placement is to have another layer of indirection keeping track of the logical to physical location mappings. The metadata layer needs to be consulted for every single access (read, write).

4.1 Metadata Implementation

In a system where a single machine serves as the control plane, handling all metadata operations, any disruption or crash of this machine would result in a complete service outage for SkyStore. To enhance reliability and system performance, it becomes imperative to introduce multiple machines to manage metadata. Distributing metadata shards for different clients across distinct machines ensures that the system remains resilient, even in the event of a failure. This architectural approach not only bolsters reliability but also contributes to improved overall system performance.

Dynamo [9] utilizes consistent hashing to partition and replicate data, achieving scalability and availability. However, consistent hashing is not conducive to data locality, as clients may need to connect to machines that are very distant from their current location. Therefore, maintaining a server configuration list and selecting one server as the primary based on the client's current region is a more favorable approach. And adopting the primary-backup method for fault tolerance.

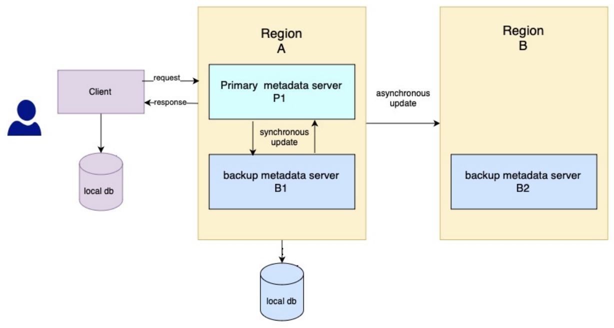


Figure 3: Metadata Backup

Primary-secondary approach is a popular paradigm implemented by various distributed systems like HDFS [14] to ensure fault tolerance and availability. With this approach, the primary server handles all operations, while the secondary server is either updated periodically or always in full sync with the primary server.

The straightforward approach is to update both primary and the secondary server for every operation, making both servers fully synchronized. Although this might not be optimal and affect the overall performance of the system, the benefit is the secondary server is always consistent with the primary. In case the primary server becomes unavailable, the system can switch to secondary right away and there is no data loss.

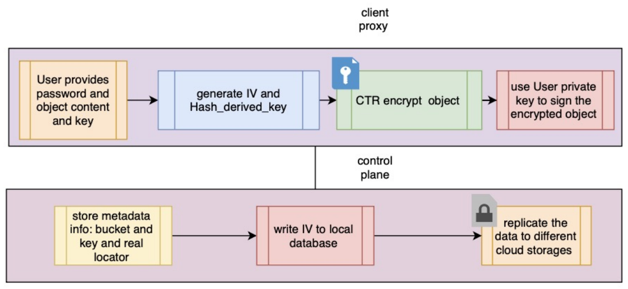
A different approach when the secondary server is periodically updated from the primary. While this reduces overload on the system, the main drawback is that in case of primary server failure, the secondary will have state based on the last sync and even with data recovery, still may result in some data loss. To address the fault tolerance of the metadata server in SkyStore, we decided to follow the hybrid version of the primary-secondary approach as follows. SkyStore will initiate a primary metadata server $P1$ and a backup metadata server $B1$ in the same cloud region. In addition, SkyStore will initiate another metadata backup server $B2$, in a different region. Metadata servers $P1$ and $B1$ will be in full sync inside a single region and thus maintain strong consistency between them. Every client's request to the primary metadata server $P1$, will be synchronously forwarded from $P1$ to the backup metadata server $B1$ in the same region. Moreover, all bucket operations, like create or register bucket, will be synchronously forwarded from $B1$ server into another region with $B2$ server. Once metadata servers fully process the requests, the primary metadata server will generate a response to the user. This ensures that if the primary metadata server is down, has network issues or other faults, SkyStore will continue to operate without disturbance by accessing secondary metadata server $B1$ located in the same region.

This solution however does not cover the case when the entire cloud region is down and both primary and backup metadata servers located in that region become not available. To address this issue, SkyStore also keeps another metadata server $B2$ in a different region that is updated by $B1$ server. As opposed to $P1$ and $B1$ that are updated synchronously for all operations, the $B2$ server is updated asynchronously for metadata related operations. The overall flow would be as shown in Fig. 4.

In case the entire cloud region is down and both $P1$ and $B1$ are not available, SkyStore will continue to operate by accessing the $B2$ metadata server located in the different region.

4.2 Security

In the contemporary landscape, data privacy is of paramount importance, especially concerning the training data used in the context of machine learning training. Users have the option to convey their passwords to SkyProxy, wherein the client-side encryption takes place. Given that user-supplied passwords often exhibit a short length and low entropy, rendering them susceptible to dictionary attacks, it becomes imperative to employ a robust Key Derivation Function (KDF) [10] for the generation of a secure key.



client proxy
User provides password and object content and key
generate IV and Hash_derived_key
CTR encrypt object
use User private key to sign the encrypted object
control plane
store metadata info: bucket and key and real locator
write IV to local database
replicate the data to different cloud storages

Figure 4: Metadata Backup

The data breach encountered by LinkedIn in 2012, wherein 6.5 million user passwords were compromised, serves as a poignant reminder of the vulnerabilities associated with storing unsalted passwords using SHA-1 [11]. This practice exposes them to offline password guessing attacks. Consequently, SkyProxy adopts a stringent security approach by randomly generating a numerical salt and employing PBKDF2 for encryption. PBKDF2, characterized as a slow hash function, significantly elevates the complexity of executing brute force attacks utilizing rainbow tables. The incorporation of a slow hash mechanism serves to augment the overall security of the system. However, this enhancement comes at the cost of increased system latency. Empirical results from local testing reveal that, contingent on the number of iterations employed in the slow hash process, there is a consequential impact on the processing time of client requests. For instance, with an iteration count set at 600,000, the introduction of each byte results in an additional latency of 2001ms. Conversely, for a limited number of iterations, each byte contributes an increased latency of 0.0147168ms.

In the practice of encrypting data using block ciphers, CBC is widely used in commercial applications. However, CTR mode encryption and decryption can be parallelized, leading to higher performance. Additionally, CTR mode provides a crucial feature – the inclusion of a counter enables the encryption or decryption of an arbitrary point in the message without the need to start from the beginning.

we employ RSA [13] signatures to guarantee both integrity and authentication. Despite the computational cost associated with signatures, they offer distinct advantages over HMAC, particularly in data-sharing contexts. Unlike HMAC, which necessitates the sharing of the HMAC key through a secure channel, the verification key for a signature is public, eliminating the need for a secure key-sharing process.

5 SkyStore Policies

In this section, we delve into the specific policies implemented by SkyStore to effectively navigate the challenges of data management in multi-region, multi-cloud environments.

5.1 Write Policy

Currently, we integrate several polices into SkyStore.

- *Single-Region*: User should set a specific region to write into.
- *Write-Local*: Write to the region that client are currently located at.
- *Push*: User will provide a list of regions to SkyStore, all of the regions provided should be in the set of initialization regions.
- *Replicate-All*: Each time every object will be propagated to every region in the initialization region list.
- *Copy-on-Read*: For the first time of GET operation, if the object is located in a remote region, SkyStore will fetch and cache the object in the region the client is currently located at.

5.1.1 First write: write local

SkyStore adopts a *write-local* policy for initial data storage. Upon a write request, data is primarily stored in the request originating region. This approach significantly reduces latency and egress costs associated with the write operation while ensuring immediate availability of data where it's first needed. Each of the write operation generates an object with an updated version. SkyStore tracks versions for each write operation to maintain records of all changes, ensuring data integrity and version control.

5.1.2 Cache: Copy-on-Read

Complementing its write-local strategy, SkyStore employs a reactive *copy-on-read* approach to optimize data management across multiple regions. This method comes into play when a read operation is initiated from a region different from where the data is initially stored. Upon such a request, SkyStore creates a copy of the data in the object store of the requesting region, enhancing read performance for future operations and reducing the costs associated with cross-region data access.

When the *copy-on-read* policy is used, the behavior of multi-versioning support will change: Instead of creating a new logical object for the *copy-on-read* request, SkyStore will link a new physical object locator to the old logical object.

This demand-based caching strategy is particularly effective in lowering data transfer cost when repeated reads are often. It is different from the preemptive replication strategy of systems like SPANStore, which push data to predicted region of access upon write. When workloads vary unpredictably, such replication model can lead to significant egress and unnecessary storage charges, depending on how good the prediction model is.

5.2 Read Policy

Currently, three read policies are supported in SkyStore.

- *Direct Transfer*: Transfer data from region A to region B directly.
- *Cheapest*: Select the node with cheapest transfer cost from the region lists that currently hold the data.
- *Closest*: Read from the region with least latency.

SkyStore's read policy is designed to balance cost-efficiency with service level objectives (SLOs). The system dynamically selects sources to fetch data from based on cost, performance, and consistency requirements. The cost and latency is calculated through a simulation graph established during the 1st call to initialize the policy based on user requirement. For each edge in this graph, it contains the latency (ms) and cost ($/GB) attributes.

Building this graph with more than 80 regions of different cloud providers information is extremely time costing. To reduce this overhead, we attach a singleton class instance that initializing the graph, so that the overhead will only be charged for the 1st time.

Under this policy, network failures do not impede data access within a region, promoting high availability. The policy fundamentally operates on two principles: Read-My-Write Versioning and Eventual Consistency.

5.2.1 Read-My-Write Version

For read operations, SkyStore prioritizes accessing the most recent version of the data. Users can choose to read from either the most cost-effective or the fastest region available. The policy ensures that any write operation invalidates other cached copies of the data in different regions, triggering a re-read to maintain version consistency. This approach guarantees that users always interact with the latest data, crucial for applications where data freshness is an important criteria.

5.2.2 Eventual Consistency Version

In scenarios where immediate data recency is less critical, SkyStore offers an eventual consistency model. This model allows reading from local copies, even if they are not the latest version, thus prioritizing access speed and cost over data freshness. The system updates these local copies in subsequent accesses, aligning them with the most current version over time. This bounded staleness approach is particularly effective in scenarios where frequent, cross-region synchronization is not feasible or necessary, providing a balanced trade-off between data currency, access speed, and operational cost.

6 Evaluation

We use several benchmarks to evaluate the performance of the system. Generated traces are used to mimic the real-world workload. The evaluation contains comparison with real object storage, scalability examination, and different policy comparisons. We aim to get small overhead (e.g. 10%) on both the control plane and data plane compared to the under-level object storage. Specifically, for the control plane side, we will prove the high-concurrency support of our optimization design. As for the data management by different polices, we prove the efficiency of fast-write and cache policy, especially on large workload (e.g. 1GB). By simulating on the traces mimic the real-world workload, we compare SkyStore to the naive data management methods and current geo-distributed systems, like SPANStore, to show the cost & latency performance of our system.

6.1 Object Store: Operations

We reference JuiceFS [1] benchmark to measure SkyStore system overhead compared to naively using the provider's tool. We run the performance test on object store and compare the latency for operation with and without SkyStore. The benchmark consists of put, get, list, head, and delete 100 objects of size 128 KiB and measure the average latency per object. The result in Fig. 5 shows that the overhead of using SkyStore compared to using cloud provider's API is minimal: for put and get operation, the overhead is less than 10%. Furthermore, SkyStore can even speedup the list and head operations by up to 5×. The overhead of store server originates from two aspects: first, the control plane side code is currently written in Python, which is comparably slow than C/C++/Rust. Second, the database query takes time to finish. To reduce this overhead, we try to maximize the performance of Postgres, and deploy it on *m5d.8xlarge* VM machine of AWS, which attaches NVMe SSDs. Also, the database queries together with the hyper-parameters of the database are highly optimized to reduce overhead at the server side.

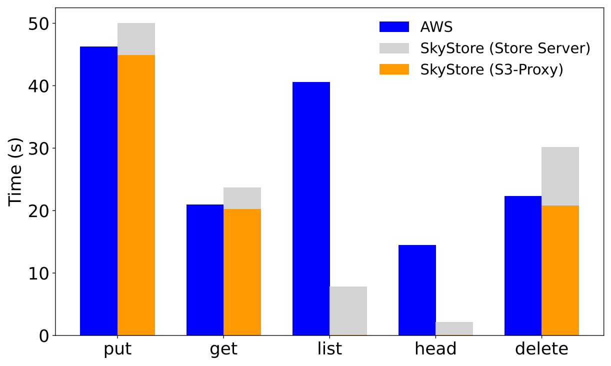


Figure 5: Object store performance: with and without Skyproxy on region *aws:us-west-1*

The PUT operation generally has a 2× overhead on store server side than the GET operation, since the PUT needs to perform two DB transactions (*start_upload* and *complete_upload*). The DELETE operation is not optimized as the PUT and GET, since currently we are deleting objects once a time without parallezing this process. The LIST and HEAD operation is extremely fast since the store server is hosted at the same region of the client, and these operation can query the DB directly without fetching information from the real object storage buckets.

6.2 Object Store: Scalability

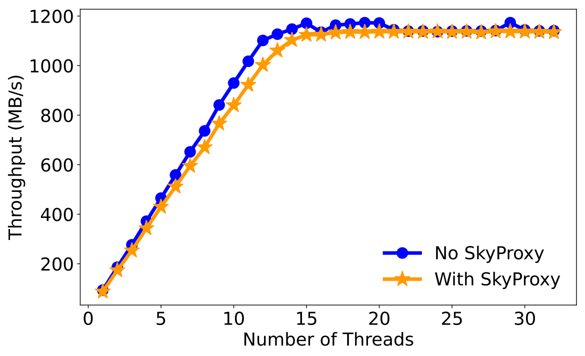


Figure 6: Throughput on Download: *aws:us-west-1* region, running s3 benchmark

To test the scalability of the SkyStore, we use S3-benchmark [2] to plot the relationship between thread count and performance. The object size is set to 16MB, testing from 8 threads to 32 threads on AWS *m5d.8xlarge* EC2 machine, this large machine strong performance allows bench-marking the throughput of the system without worrying about the upper bound of the network bandwidth or disk I/O bandwidth. As shown in Fig. 6, the scalability of SkyStore is quite close to the performance of S3, both of which can reach about 1100MB+ on 32 threads. Since the data plane side is designed as stateless, this result shows the high-concurrency support at the control plane side.

6.3 Policy Efficiency on large objects

To illustrate the affect of using fast-write and Cache on large objects, we compare these two policies against *replicate_all* policy and *single_region* policy respectively.

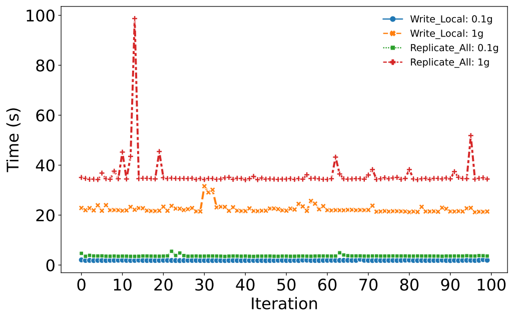


Figure 7: Write_Local & Replicate_All

To evaluate the performance of fast write, we generate two files of 0.1GB and 1GB respectively. the client is located at region *aws:us-west-1*. The *write_local* policy will always write to the *aws:us-west-1* region. On the other hand, the *replicate_all* policy will write to both the *aws:us-west-1* and *aws:us-east-1*. The result is shown in Fig 7. For objects with smaller sizes, the advantage of using fast write is not obvious, while for those large writes (e.g. 1GB), this can allow users to finish the PUT operation as soon as possible. The improvement compared to naive data management depends on several other factors, such as the region numbers and distances, and object sizes.

In Fig 8, the user located at *aws:us-west-1* with *single_-region* policy will try to write to the *aws:us-east-1*. And For the *copy_on_read* policy, the GET request will perform an additional PUT operation that caches the objects fetching from *aws:us-east-1* into *aws:us-west-1*, that reasons the time cost of the 1st time is much higher than the next requests.

As the object sizes we dealt with are becoming larger, the performance effectiveness of fast write and Cache become more obvious than the small sizes cases. This allows SkyStore to take advantage of its flexible write polices to get better performance on higher workloads.

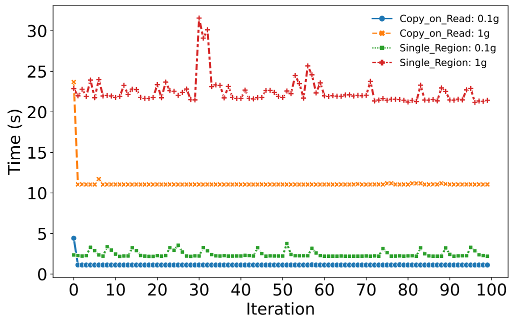


Figure 8: Copy_on_Read & Single_Region (Remote)

6.4 Large-Scale System Comparisons

To comprehensively examine Object Storage behavior on real-world workload, we generated traces based on the Storage Networking Industry Association's (SNIA) I/O Traces, Tools, and Analysis (IOTTA) repository [3].

Each row in the trace represents a REST access with the following columns: <TIMESTEMP> the time elapsed in milliseconds from the beginning of the week. <OP> the operation type (GET, PUT, HEAD, DELETE, COPY). <OBJ_KEY> an anonymized Object key. <SIZE> the Object Size. <ISSUE-REGION> represents the requests that is performed at.

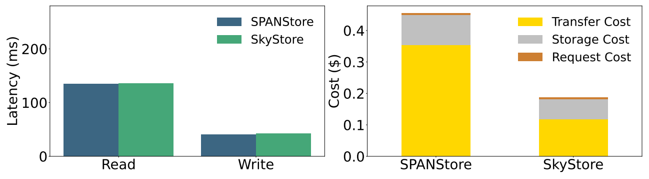


Figure 9: SkyStore latency & cost performance against SPANStore

To simulate the workload of real world, where read-heavy access pattern will take place as we have mentioned in section 1, SkyStore is initialized on three regions (*aws:me-south-1, aws:eu-south-1, aws:us-east-1*), with 1700+ requests in traces, with a GET/PUT ratio of 3.25. Object sizes we use range from 416 Bytes to 400MB. We first compare SkyStore against SPANStore. From Fig 9, the SPANStore policy, which will query and predict every 1h based on historical workload measurements for the follow-up behavior of the requests, add extra cost on moving objects around regions and buckets. For SkyStore, it uses the cache (*copy-on-read*) policy here to solve the challenge of read-heavy, this helps the system reaches read/write latency similar to the single storage case, while keeping the aggregated costs lower than SPANStore.

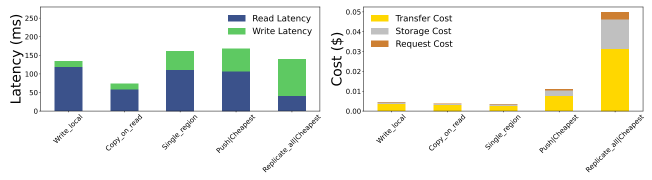


Figure 10: SkyStore latency & cost performance on different policies

To evaluate the performance of our system more thoroughly, we generate another trace, which contains 100 requests with a GET/PUT ratio of 7:3. The object sizes is smaller compared with the last one, for 1MB average. The 100 requests are distributed across 22 regions, including Asia, America, Europe, etc. For the *push* policy used here, we let the SkyStore to push to three regions, including *aws:us-west-2, aws:us-east-1, aws:us-west-1*, which can be regarded as a *replicate-all* operation with smaller overhead.

As in Fig 10, the *write-local* policy takes the smallest write latency. When replicating objects to all the regions, as expected, it helps reach a small read latency while bringing high transfer cost and storage cost. This will be useful when activating global client collaboration and users will perform read and write from all locations frequently. Storing all objects in a single region (including *write-local*) leads to a high read latency, since for repeated reads we always need to read remotely, but that helps on reducing storage cost. For *copy-on-read* policy, it avoids the high read latency by replicating to the local client region, while the write latency is basically the same since we perform this COPY operation asynchronously.

The cost of *write-local, copy-on-read* and *single-region* is similar, since they are all performed write operation to a specific region per time. And since the number of requests and the size of the objects is comparably small, the transfer costs of them do not have too much difference.

7 Related Works

Multi-Region Buckets Amazon S3 Multi-Region Access Points [5] enable applications to access data from S3 buckets located across multiple AWS Regions through a single global endpoint. These access points use AWS Global Accelerator, which automatically routes application requests to the closest S3 bucket, optimizing for the shortest distance and offering built-in network resilience.

In Google Cloud, multi-region buckets [8] are designed to store data across multiple geographic regions, allowing users to access the data from various locations. This setup can enhance data redundancy and availability, ensuring that data remains accessible even in the event of a regional outage or failure.

Geo-distributed Storage SPANStore [16] is the first system of its kind that combines object storage across multiple public cloud providers to deliver cost-effective and efficient data access for geo-distributed applications. These applications span a variety of areas, including shared web services, social networking platforms, and collaborative editing tools. The primary aim of this project is to streamline the process of data replication to offer a cost-optimal application services present a unified and consistent view of global storage services.

Volley [4] is one of the earliest works we identified that optimizes data placement in geo-distributed applications. It focuses on the problem of migrating fine-grained data items across multiple data centers to minimize clients' access latency. The workload it considers is end-user web applications, specifically Microsoft Live Mesh (a predecessor to OneDrive), Microsoft Live Messenger, and Facebook NewsFeed.

Nomad [6] is a geo-distributed, demand-based, replicated key-value store embodying the concept of "distributed storage overlays". The distributed storage overlays are an abstraction that represents data as stacked layers in different places.

8 Conclusion

There are many multi-regions and multi-cloud solutions today, such as the multi-region buckets provided by the cloud providers and geo-distributed storage system like SPANStore [16]. For the multi-region buckets, it only supports users to perform operations inside one specific cloud providers. SkyStore provides unified interface based on S3 semantics to support multiple cloud providers simultaneously. And the under-level control plane will make decisions based on the read/write policies automatically.

For the previous geo-distributed systems, they only provides few APIs (e.g. PUT/GET). It is difficult for those systems to cope with complex scenarios which requires multiple operations to be engaged in. Also, SkyStore reaches a smaller latency and lower cost compared to the commercial object storage and academic geo-distributed systems. Although integrating the data plane and control plane, it controls the maximum overhead of GET/PUT under 10%, and obtain better performance on metadata fetching operations like LIST/-HEAD.

Faced with high-concurrency scenarios, SkyStore optimize its performance by providing highly-optimized control plane, which enables SkyStore's high throughput. The control plane is optimized not only for high-concurrency support, low latency queries, but also fault tolerance in case of single machine/region failures.

For the security insurance, we pass the user's password through PBKDF2 to derive the hash-derived key. This key is then utilized for CTR encryption, and the user's private signing key is employed for digital signatures, ensuring a robust combination of data integrity and confidentiality.

9 Future Work

Currently, SkyStore provides several polices to replicate the data, including *copy-on-read*, *push* and *replicate-all*/ However, these approaches also bring other challenges regarding storage expenses. In scenarios where the replicated data has low subsequent access, the system may accumulate multiple underutilized copies and increase overheads in maintaining consistency across cached copies. To solve this problem, we need to design eviction policies to optimize the storage cost and query latency.

Besides, the system needs more thorough support on the APIs it integrated. For example, the Azure Rust SDK has not added the feature of changing current version setting. Since our system relies heavily on the SDKs provided by the cloud providers, SkyStore does not support change version setting at Azure side.

Third, the data plane and control plane can be further optimized. This includes changing the store server side code from Python to C/C++/Rust, and support parallel deletion operations in the S3-proxy side.

In the event of a client-initiated region change, the pre-existing metadata remains housed in the original metadata server. As a result, the client is still required to establish a connection with the metadata server of the initial region, leading to higher latency in user operations. Hence, there is a need to implement a data movement mechanism to handle such situations.

We require a more efficient mechanism for data recovery. Utilizing periodic checkpoints, in the event of a primary node failure, the backup can promptly transmit the checkpoint to the newly initiated node. Subsequently, during periods of low workload, the data following the checkpoint can be asynchronously sent to the recovery node. The implementation of the checkpoint mechanism serves to optimize the backup's load during the data recovery process.

References

[1] https://juicefs.com/en/.

[2] https://github.com/dvassallo/s3-benchmark.

[3] SNIA: IOTTA repository. http://iotta.snia.org/traces/key-value/36305, 2023.

[4] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Habinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, 2010.

[5] Amazon s3 multi-region access points. https://aws.amazon.com/s3/features/multi-region-access-points/. Accessed on 12/15/2022.

[6] Ignatius De Villiers. *Nomad: a pithos based P2P distributed storage network implementation*. PhD thesis, Stellenbosch: Stellenbosch University, 2021.

[7] Egress Fee. https://www.cloudflare.com/learning/cloud/what-are-data-egress-fees/.

[8] Gcp multi-region bucket. https://cloud.google.com/storage/docs/locations#location-mr. Accessed on 12/15/2022.

[9] Madan Jampani Gunavardhan Kakulapati Avinash Lakshman Alex Pilchin Swaminathan Sivasubramanian Peter Vosshall Giuseppe DeCandia, Deniz Hastorun and Werner Vogels. Dynamo: Amazon's highly available key-value store. 2007.

[10] H Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme cryptology eprint archive report 2010/264 (2010). 2010.

[11] Simon Marechal. Advances in password cracking. journal in computer virology 4, 1 (2008), 73–81. 2008.

[12] Google Cloud Storage Price. https://cloud.google.com/storage/pricing.

[13] L. Adleman R. L. Rivest, A. Shamir. A method for obtaining digital signatures and public key cryptosystems r. l. rivest, a. shamir, and l. adleman. communications of the acm, vol. 21, no. 2, february 1978. 1978.

[14] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[15] UltraDict. https://github.com/ronny-rentner/UltraDict.

[16] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 292–308, New York, NY, USA, 2013. Association for Computing Machinery.