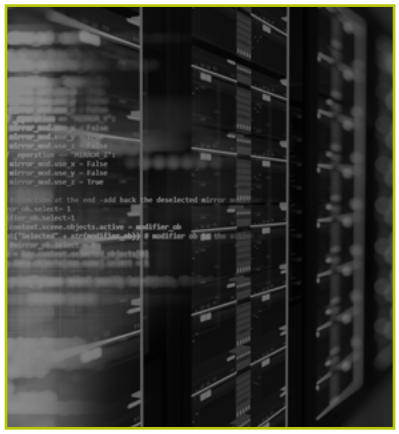
Datadobi



OBJECT MIGRATIONS

The AWS S3 API has become a de-facto standard for accessing data stored on object storage platforms. The appearance of this de-facto standard makes possible for data to be migrated between object platforms provided by different vendors. It is very similar to NAS migrations wherein data can be migrated between disparate vendor platforms via the use of the standard NAS protocols (SMB and/or NFS).

S3 Review

Object platforms that support the S3 API typically store data in buckets which are an abstraction of the underlying physical storage. The buckets provide a flat namespace. This contrasts with NAS platforms that use file systems implementing hierarchical namespaces that employ the concept of directories to store logically related content. In S3, all the data is co-mingled in the same bucket.

To aid with the organization of content, S3 compatible platforms allow object keys (e.g., object names) to include prefixes and delimiters. The combination of prefixes and delimiters allows for the object keys to mimic the hierarchical structures found on traditional NAS systems. For example, on a NAS platform a file named schedule.pdf might be stored in the directory /main/subdir1/subdir2/schedule.pdf along with other logically related files. If this file was copied to an S3 bucket, it would simply exist in the bucket with all other copied items. But with the ability to add prefixes and delimiters to the object name; S3 allows the mimicking of a directory structure since the prefixes "main", "subdir1", and "subdir2" can be added to the object name and delimited with the "/" character such that the resulting object name is "main/subdir1/subdir2/schedule.pdf". When browsing the contents of the bucket, the browsing application can filter the content in a way to present a view comparable to the familiar directory tree found on NAS systems.

In S3, objects are always written and read via HTTP requests. This makes sense because the S3 API is a REST API and HTTP is the mechanism used to interact with the API. In simplest terms: "PUT" requests write objects while "GET" requests read objects. Whenever objects are written they are stored in an immutable fashion – any updates cause a new version of the object to be written. While it is possible to enable versioning on buckets where each new successive update of an object creates a new historical version; there is no requirement for versions to be maintained. If versioning is not enabled, it does not mean objects cannot be updated but simply that a history of prior versions will not be maintained. Updates to an object still create a new object regardless of whether versioning has been enabled on the bucket. And yes, the versioning applies to the ENTIRE bucket versus portions of it.

In addition to the core data associated directly with the object, there is also metadata. Within S3 there are two categories of metadata – User and System-defined. User metadata, as the name implies, can be supplied by the end user and/or application writing the object. The user metadata is stored as key-value pairs in "x-amz-meta-" request headers which uniquely identify them as user metadata versus other HTTP headers. Interestingly, an update of user metadata requires the entire object to be rewritten.

System-defined metadata includes metadata items such as the "Last-Modified" timestamp and the ETag (entity tag) associated with the object. These values are assigned to the object by the server when the object is written and cannot be manipulated by calling applications. Since these values are assigned by the server, there is not the ability of updating system-defined metadata as is possible with user metadata.

S3 API IMPLEMENTATIONS

Since we now know some basics about S3 implementations and behavior, we can move on to the subject of migrating content between S3 systems.

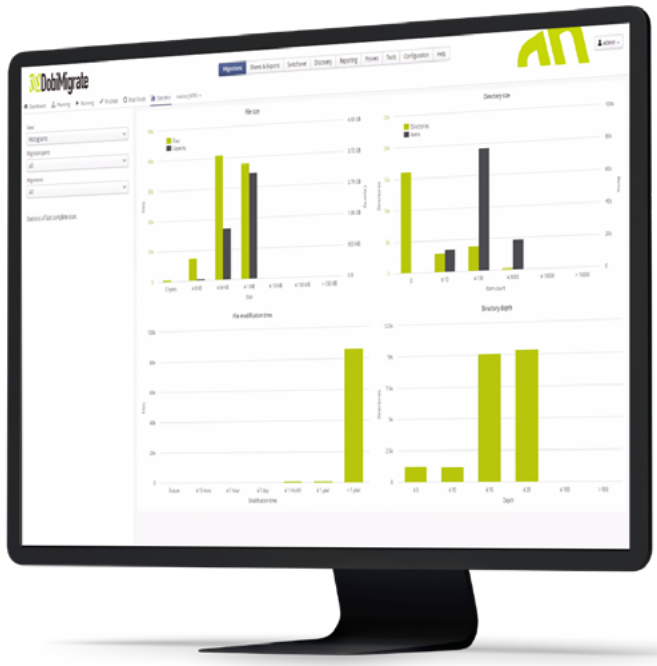
As previously mentioned, the AWS S3 API has become a de-facto standard among storage vendors. It's important to keep in mind that a de-facto standard does not equate to an industry standard as specified by IEEE, for example. This being the case, each vendor offering a storage platform with an S3 compatible interface is providing their own interpretation of the API and there are varying levels of compatibility with the AWS API.

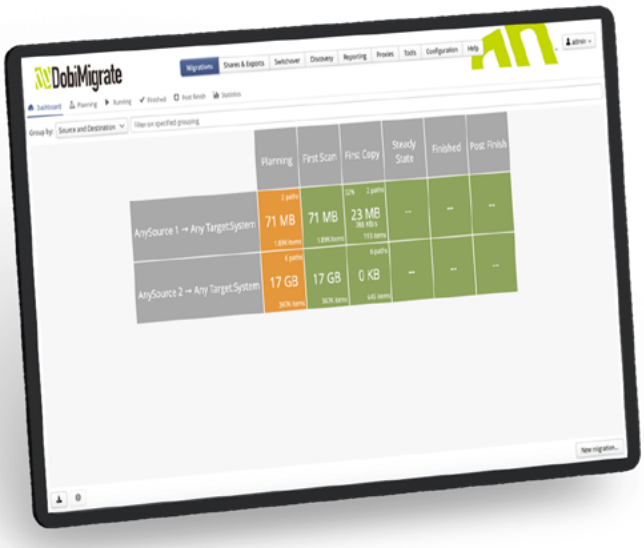
An example of an incompatibility is "Storage Class." AWS provides several storage classes corresponding to varying levels of performance, cost, accessibility, and even durability. While it is possible to write an object (PutObject request) to a specific storage class in AWS, it may not be possible with an object storage platform from another vendor since they may effectively have a single storage class due to homogeneous storage accounting for the entire object storage system. A vendor in this situation will likely not implement support for writing to specific storage classes since it makes no sense to do so. DobiMigrate must navigate this variability in API implementations because a vendor simply claiming "S3 API support" is often a claim of relative versus absolute API support.

HOW DOBIMIGRATE WORKS

DobiMigrate (as of version 5.9) supports migrations between platforms implementing the S3 API. In the same manner that DobiMigrate migrates file data between any vendor NAS platforms, it can migrate S3 object data between any S3 compatible platforms.

The first step of migrating data between source and destination S3 platforms is to ensure a bucket is created on the destination platform. With both source and destination buckets available, DobiMigrate will list the contents of each platform's bucket. In this first listing there is typically a number of objects stored in the source bucket while the destination bucket is empty. DobiMigrate compares the scan results and generates copy operations to copy objects found in the source bucket to the destination bucket. After these copies are completed, the first two phases (First Scan followed by First Copy) of the migration are complete. When copy operations are executed, the objects (including all prefixes and delimiters) are copied to preserve any notion of hierarchical organization present on the source.





After the First Scan and First Copy phases, DobiMigrate enters the Steady State phase. Steady State is an ongoing phase where DobiMigrate will, based on the frequency assigned by the migration administrator, periodically re-scan the source and destination buckets. The scan results from the source and destination will be compared to determine the delta operations required to re-synchronize the two systems. With the delta operations known, DobiMigrate distributes worklists to the proxies who then execute their assigned worklists in a highly multithreaded and parallel manner.

One thing to keep in mind is that there can be an extremely high number of objects stored in the buckets, so it is important to list out the contents of the buckets as quickly as possible. The sort order of the object keys returned is important and DobiMigrate has advanced processes that divide up the keys in a manner that highly parallel scans of the bucket contents can be executed. This approach is several orders of magnitude faster than alternatives that only use a single thread or a simplistic, non-optimized process to list bucket contents.

While it is important to copy data as quickly as possible between the two platforms, it is equally important to verify the accuracy of the copy operations. To that end, DobiMigrate verifies the content of all objects written to the destination against their original source copy. It accomplishes this by calculating a hash digest on each object as it is read from the source system. After copying to the destination, a read-back request is issued along with an additional hash calculation on the object returned from the destination. These hash digests equate to a unique "fingerprint" associated with the data. Any variation in the hash digests associated with the source and destination object indicates a possible corruption of the object. If that occurs, DobiMigrate will automatically re-copy the object identified in the verification failure during its next Steady State iteration. Additionally, DobiMigrate provides chain-of-custody reporting that proves the accuracy of the migration post-switchover by producing a list of objects along with their associated hash digests, ETag values, and timestamps.

At some point during the Steady State phase, a date/time will be determined for executing a Switchover event wherein the destination becomes the primary platform. DobiMigrate provides a dedicated module for scheduling and executing the switchover from the source to the destination system. There are two types of switchover events that can be executed. The first event is called a Dry Run which allows the migration administrator to select any collection of bucket pairings currently in Steady State, and to run the processes involved with the switchover event for the purpose of timing the event. The Dry Run takes the guesswork out of estimating the amount of time required for core switchover event activities related to DobiMigrate. With that estimate in hand the administrator can adjust the time required to coordinate with application owners to redirect applications, perform acceptance testing, and/or other activities prior to the switchover window closing.

The second type of Switchover event is the actual switchover itself - as opposed to the Dry Run previously described. During this event, DobiMigrate allows the administrator to schedule and execute the steps required to proceed from Steady State to the final Finished state where applications and/or end users are redirected to the destination platform. At this point, the chain-of-custody report can be downloaded before declaring the migration complete.

SUMMARY

DobiMigrate executes the fastest, most accurate migrations using purpose-built technology paired with a unique knowledge of various S3 implementations.