# Background

## What is Lifecycle Configuration

* A ***lifecycle configuration*** is a set of rules that define actions that applies to a group of objects.
* Enforcing Lifecycle policies using S3 API is a great way of ensuring your data is managed safely (without experiencing unnecessary costs) and that your data is cleanly deleted once it is no longer required.
* Lifecycle policies allow you to automatically review your buckets and have them moved either to different cloud service provider or in the same cloud but to other storage class or have them deleted . You may want to do this for security, legislative compliance, internal policy compliance, or general housekeeping.
* Object versioning is also supported in lifecycle configuration to customize your data retention approach and control storage costs.

Actions taken in object lifecycle are:

* **Transition actions**—Define when objects transition to anotherstorage class. For example, in AWS S3 you might choose to transition objects to the STANDARD\_IA storage class 30 days after you created them, or archive objects to the GLACIER storage class one year after creating them or in Google cloud where you can choose to transit the objects from Multi-Regional Storage to Coldline Storage. There are costs associated with the lifecycle transition requests.
* **Expiration actions**—Define when objects expire. The lifecycle expiration costs depend on when you choose to expire objects. When an object reaches the end of its lifetime, it is queued for removal and gets removed asynchronously. There may be a delay between the expiration date and the date at which the object is removed.

Note : There are different implementations for Transition and Expiration actions on versioning enabled/suspended and non-versioned buckets.

## **When to use Lifecycle Configuration?**

Define lifecycle configuration rules for objects that have a well-defined lifecycle. For example:

* If you upload periodic logs to a bucket, your application might need them for a week or a month. After that, you might want to delete them.
* Some documents are frequently accessed for a limited period of time. After that, they are infrequently accessed. At some point, you might not need real-time access to them, but your organization or regulations might require you to archive them for a specific period. After that, you can delete them.
* You might upload some types of data to the cloud primarily for archival purposes. For example, you might archive digital media, financial and healthcare records, raw genomics sequence data, long-term database backups, and data that must be retained for regulatory compliance.

## Objectives

\* Goal : Develop OpenSDS object lifecycle management mechanism. Allow tenants to manage lifecycle configuration policies through APIs. There are two scenarios in object lifecycle management.

\* *Scenario 1* - Object lifecycle management across different cloud service providers and support of choosing the storage class while setting the lifecycle rules.

Example: If there is a bucket in STANDARD storage class of AWS S3 and it needs to be transitioned to COOL storage class of Azure Blob we can set a rule which will do it automatically.

\* *Scenario 2* - Object lifecycle management within the cloud storage.

Example : Transitioning a bucket from S3 “STANDARD\_IA” to S3 “GLACIER” using lifecycle rule defined in OpenSDS.

For both the scenarios we must have APIs defined which can read the lifecycle rule and perform lifecycle operations on the OpenSDS bucket.

\* Sub-Goal 1 : Provide lifecycle configuration management for non-versioned bucket.

\* Sub-Goal 2 : Provide lifecycle configuration management for versioning enabled/suspended bucket (This is not supported currently, but in the future).

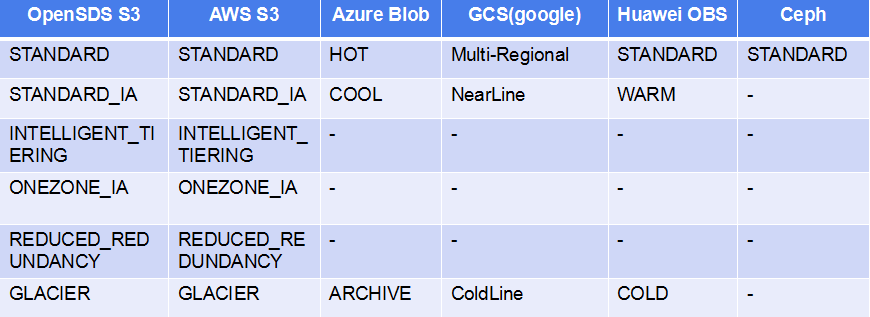
# Solution

## Defining storage classes in OpenSDS

OpenSDS will define the following storage classes :

*STANDARD, STANDARD\_IA, INTELLIGENT\_TIERING, ONEZONE\_IA, REDUCED\_REDUNDANCY, GLACIER.*

Each storage class can be mapped to a specific storage class of all cloud vendor we support in OpenSDS. The mapping is shown in the table below :



**Notes:**

* Hyphen (-) means default storage class. For example when you upload an object or bucket and set storage class as STANDARD\_IA , but the related backend is Ceph S3 which do not support STANDARD\_IA then this object or bucket will use the default storage class set to the bucket (STANDARD in Ceph).
* For GCS, Regional storage is not supported.
* We don't recommend using REDUCED\_REDUNDANCY.

**There are three options described as below, we need to choose one of it.**

## Two ways to define lifecycle rules, one for bucket, the other for backend

### Solution describe

Support to set lifecycle management rules both on bucket and backend.

Rules set on bucket are used in the scenario of expiration and cross-cloud transition. All objects in the bucket will be controlled by those rules. The management of the rules are controlled by OpenSDS only. That means OpenSDS will schedule those rules and delete object or migrate object cross-cloud if necessary.

Rules set on backend are used in the scenario of expiration and in-cloud transition. All objects, no matter which bucket it belongs to, stored in the backend will be controlled by those rules. The management of the rules are controlled by cloud vendor, OpenSDS do nothing except syncing up object meta data with cloud vendor.

### PROS & CONS

PROS

* Less work for OpenSDS.
* Can use all features of lifecycle management of each clouds.
* No limitation for versioning.

CONS

* Need two sets of APIs, that’s different with lifecycle management in S3.
* In-cloud transition is controlled by cloud vendor, it’s difficult to keep metadata of each object completely consistent with cloud all the time.
* Can’t manage lifecycle completely on the bucket level. (Because those rules for backend.)

## Define lifecycle rules on bucket, OpenSDS translate and set rules in cloud if needed

### Solution describe

Only support to set lifecycle management rules on bucket, for each rule:

* with expiration as it’s action, OpenSDS will call API to delete the object, rule will not be set in cloud.
* with cross-cloud transition as it’s action, OpenSDS will migrate objects, rule will not be set in cloud.
* with in-cloud transition as it’s action, OpenSDS will translate and set rule in cloud, the rule will be added a filter that with bucket name as it’s prefix. And OpenSDS need to sync up object metadata with cloud vendor. (Details of sync up?)

### PROS & CONS

PROS

* APIs are compatible with S3.
* Can manage lifecycle on the bucket level like cloud themselves.
* No limitation for versioning.

CONS

* Can’t support to keep the object key in cloud be the same as it in OpenSDS. (As default, object key in cloud is composed of bucket name in OpenSDS and ’/’ and object key in OpenSDS.)
* As an object of a bucket can be stored in any backends, the rule need to be set in all backends, and if a new backend is registered, it also need to add this rule.
* In-cloud transition is controlled by cloud vendor, it’s difficult to keep metadata of each object completely consistent with cloud all the time.
* Not sure if all cloud support filter objects by prefix of object key.

## Define lifecycle rules on bucket, no rules will be set in cloud

### Solution describe

Only support to set lifecycle management rules on bucket, all rules will be scheduled by OpenSDS, and for each rule:

* with expiration as it’s action, OpenSDS will call API to delete the object, rule will not be set in cloud.
* with cross-cloud transition as it’s action, OpenSDS will migrate objects, rule will not be set in cloud.
* with in-cloud transition as it’s action, OpenSDS will call API to change object’s storage class if needed, no rule will be set in cloud. No need to sync up metadata with cloud vendor.

### PROS & CONS

PROS:

* APIs are compatible with S3.
* Can manage lifecycle on the bucket level like cloud themselves.
* All rules are controlled by OpenSDS, there is no sync up problems about lifecycle.
* Because different cloud vendor provide different rules, with this solution we can provide user with the unified rule definition.

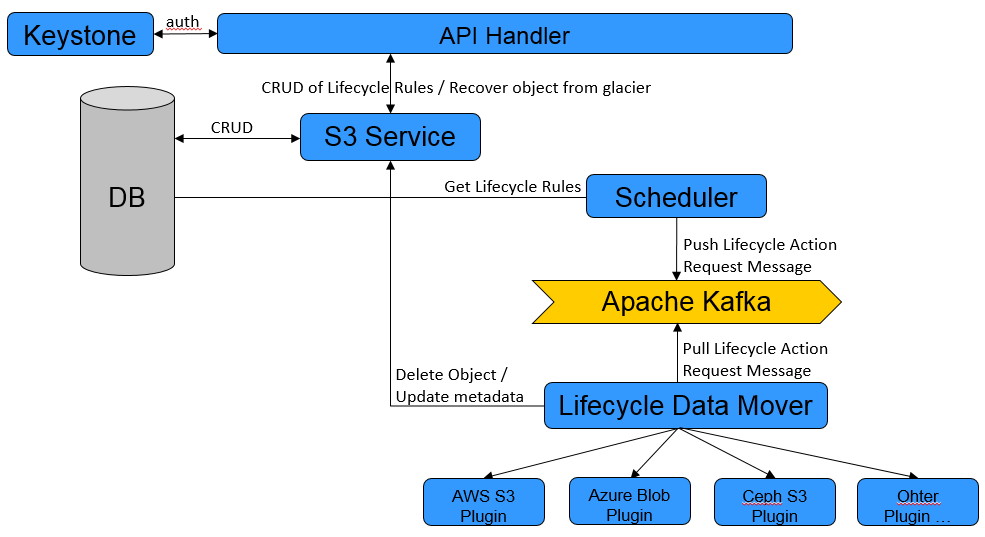
CONS:

* It’s not sure if the cost of using OpenSDS lifecycle and cloud lifecycle are the same for all cloud vendors. (Azure Blob, HW OBS, AWS S3, Google Cloud Storage… )
* Because the limitation of cloud vendor that storage class of object in history (with versioning enabled) can’t be changed, object versioning need to be totally managed by OpenSDS, it can not use the versioning ability of cloud vendor.
* If cloud vendors add some new features to lifecycle management or versioning, there is a risk that we can’t achive the same goal like those new features by ourselves.

## Final choice

**Number 3 is chosen as the final solution.**

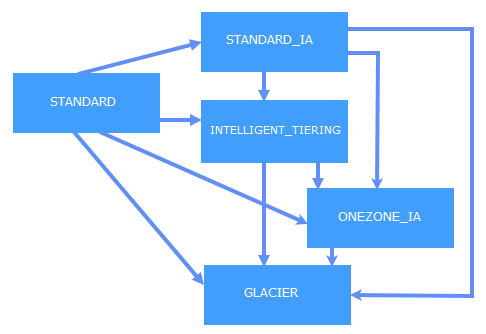
# Framework



# Constraint

## Transition constraint

Like AWS S3, OpenSDS supports a waterfall model for transitioning between storage classes as below:



As showed in the diagram, the transition from one storage class to another is one-way.

Depends on cloud vendor, there may have minimum storage time requirements, that means you need to pay minimum storage charges, for example:

* + - 1. For AWS, an objects must be stored at least 30 days in the storage class of STANDARD, STANDARD\_IA, INTELLIGENT\_TIERING, ONEZONE\_IA, and at least 90 days in GLACIER;
      2. For Azure, a blob must be stored at least 30 days in STANDARD\_IA (cool tier in Azure), and at least 180 days in GLACIER (archive tier in Azure).

For OpenSDS, lifecycle transitions have the following constraints:

1. Objects must be stored at least 30 days in the current storage class before you can transition them to STANDARD\_IA or ONEZONE\_IA. For example, you cannot create a lifecycle rule to transition objects to the STANDARD\_IA storage class one day after you create them.
2. For larger objects, there is a cost benefit for transitioning to INTELLIGENT\_TIERING. OpenSDS does not transition objects that are smaller than 128 KB to the INTELLIGENT\_TIERING storage class because it's not cost effective.
3. For cross-cloud transition, the supported target storage class depends on the type of target backend. The following table shows which storage class of each backend type can be used as target storage class.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | STANDARD | STANDARD\_IA | INTELLIGENT\_TIERING | ONEZONE\_IA | GLACIER |
| **AWS S3** | No | Yes | Yes | Yes | Yes |
| **Azure Blob** | No | Yes | Yes | No | Yes |
| **Huawie OBS** | No | Yes | No | No | Yes |
| **GCS** | No | Yes | No | No | Yes |
| **Ceph S3** | No | No | No | No | No |
| **FusionStorage Object** | No | No | No | No | No |

## About conflicting lifecycle actions

**Conflicting Lifecycle Actions**

For OpenSDS, expiration action is precedence over transition action.

For example, there are two rules as below:

<LifecycleConfiguration>

<Rule>

<ID>Rule 1</ID>

<Filter>

<Prefix>logs/</Prefix>

</Filter>

<Status>Enabled</Status>

<Expiration>

<Days>365</Days>

</Expiration>

</Rule>

<Rule>

<ID>Rule 2</ID>

<Filter>

<Prefix>logs/</Prefix>

</Filter>

<Status>Enabled</Status>

<Transition>

<StorageClass>STANDARD\_IA<StorageClass>

<Days>365</Days>

</Transition>

</Rule>

</LifecycleConfiguration>

In this case, OpenSDS just chooses the expiration action on these objects.

**Overlapping Prefixes Resulting in Conflicting Lifecycle Actions**

For example, there are two rules as below:

<LifecycleConfiguration>

<Rule>

<ID>Rule 1</ID>

<Filter>

<Prefix></Prefix>

</Filter>

<Status>Enabled</Status>

<Transition>

<StorageClass>STANDARD\_IA<StorageClass>

<Days>10</Days>

</Transition>

</Rule>

<Rule>

<ID>Rule 2</ID>

<Filter>

<Prefix>logs/</Prefix>

</Filter>

<Status>Enabled</Status>

<Transition>

<StorageClass>STANDARD\_IA<StorageClass>

<Days>365</Days>

</Transition>

</Rule>

</LifecycleConfiguration>

In this case, OpenSDS will chooses to transition the subsets of objects with the logs/ key name prefix 10 days after creation.

# APIs

## PUT Bucket Lifecycle

The PUT Bucket Lifecycle operation creates a new lifecycle configuration or replaces an existing one.

**Request Syntax**

PUT bucketname/?lifecycle HTTP/1.1

Host: x.x.x.x

Content-Length: {{length}}

Date: {{date}}

Authorization: {{authorizationString}}

Content-MD5: MD5

**Request Body**

The lifecycle configuration can be specified in the request body. The configuration is specified as XML consisting of one or more rules.

<LifecycleConfiguration>

<Rule>

...

</Rule>

<Rule>

...

</Rule>

</LifecycleConfiguration>

The following is a rule example:

<LifecycleConfiguration>

<Rule>

<ID>rule\_1</ID>

<Filter>

<Prefix>projectA/</Prefix>

</Filter>

<Status>Enabled</Status>

<Transition>

<Days>30</Days>

<StorageClass>GLACIER</StorageClass>

<Backend>backend\_aws</Backend>

</Transition>

</Rule>

<Rule>

<ID>rule\_2</ID>

<Filter>

<Prefix>projectB/</Prefix>

</Filter>

<Status>Enabled</Status>

<Expiration>

<Days>365</Days>

</Transition>

</Rule>

<Rule>

<ID>rule\_n</ID>

<Filter>

<Prefix> </Prefix>

</Filter>

<Status>Enabled</Status>

<AbortIncompleteMultipartUpload>

<DaysAfterInitiation>7</DaysAfterInitiation>

</AbortIncompleteMultipartUpload>

</Rule>

</LifecycleConfiguration>

Each rule consists of the following:

* A filter identifying a subset of objects to which the rule applies. The filter can be based on a key name prefix.
* A status, indicating whether the rule is in effect.
* One or more lifecycle transition and expiration actions to perform on.

The following table describe the elements in the lifecycle rule.

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Required** |
| **LifecycleConfiguration** | Container for lifecycle rules. You can add as many as 1,000 rules.  Type: Container  Children: Rule  Ancestor: None | yes |
| **Rule** | Container for a lifecycle rule. A lifecycle configuration can contain as many as 1,000 rules.  Type: Container  Ancestor: LifecycleConfiguration | Yes |
| **ID** | Unique identifier for the rule. The value cannot be longer than 255 characters.  Type: String  Ancestor: Rule | No |
| **Filter** | Container for elements that describe the filter identifying a subset of objects to which the lifecycle rule applies. If you specify an empty filter (<Filter></Filter>), the rule applies to all objects in the bucket.  Type: String  Children: Prefix  Ancestor: Rule | Yes |
| **Prefix** | Object key prefix identifying one or more objects to which the rule applies. Empty prefix (<Prefix></Prefix>) indicates there is no filter based on key prefix.  There can be at most one Prefix in a lifecycle rule Filter.  Type: String  Ancestor: Filter | No |
| **Status** | If Enabled, OpenSDS executes the rule as scheduled. If Disabled, OpenSDS ignores the rule.  Type: String  Ancestor: Rule  Valid values: Enabled, Disabled. | Yes |
| **Transition** | This action specifies a period in the objects' lifetime when OpenSDS should transition them to the STANDARD\_IA, ONEZONE\_IA, INTELLIGENT\_TIERING, or the GLACIER storage class.  Type: Container  Children: Days or Date, StorageClass, and Backend  Ancestor: Rule | Yes, if no other action is present in the Rule. |
| **StorageClass** | Specifies the OpenSDS storage class to which you want the object to transition.  Type: String  Ancestor: Transition  Valid values: GLACIER | STANDARD\_IA | ONEZONE\_IA | INTELLIGENT\_TIERING | Yes, if you specify it’s ancestor. |
| **Backend** | Specifies the backend which you want to transition to.  Type: String  Ancestor: Transition | No. If backend is specified, the transition should be cross-cloud transition, otherwise it is in-cloud transition. |
| **AbortIncompleteMultipartUpload** | Container for specifying when an incomplete multipart upload becomes eligible for an abort operation.  Child: DaysAfterInitiation  Type: Container  Ancestor: Rule. | Yes, if no other action is specified for the rule. |
| **DaysAfterInitiation** | Specifies the number of days after initiating a multipart upload when the multipart upload must be completed. If it does not complete by the specified number of days, it becomes eligible for an abort operation and OpenSDS aborts the incomplete multipart upload.  Type: Positive Integer.  Ancestor: AbortIncompleteMultipartUpload. | Yes, if ancestor is specified. |

**Response**

The following is a sample response.

HTTP/1.1 200 OK

x-amz-id-2: r+qR7+nhXtJDDIJ0JJYcd+1j5nM/rUFiiiZ/fNbDOsd3JUE8NWMLNHXmvPfwMpdc

x-amz-request-id: 9E26D08072A8EF9E

Date: Wed, 14 May 2014 02:11:22 GMT

Content-Length: 0

Server: x.x.x.x

## GET Bucket Lifecycle

The GET Bucket Lifecycle operation returns the lifecycle configuration information set on the bucket. This operation does not use request parameters and elements.

**Request Syntax:**

GET bucketname/?lifecycle HTTP/1.1

Host: x.x.x.x

Date: {{date}}

Authorization: {{authorizationString}}

**Sample Request:**

GET bucketname/?lifecycle HTTP/1.1

Host: x.x.x.x

x-amz-date: Thu, 15 Nov 2012 00:17:21 GMT

Authorization: signatureValue

Sample Response:

The following is a sample response that shows a prefix of “projectdocs/” filter and multiple lifecycle configurations for these objects.

* Transition (in-cloud) to STANDARD\_IA after 30 days
* Transition (in-cloud) to GLACIER after 365 days
* Expire after 3,650 days

HTTP/1.1 200 OK

x-amz-id-2: ITnGT1y4RyTmXa3rPi4hklTXouTf0hccUjo0iCPjz6FnfIutBj3M7fPGlWO2SEWp

x-amz-request-id: 51991C342C575321

Date: Thu, 15 Nov 2012 00:17:23 GMT

Server: x.x.x.x

Content-Length: 358

<?xml version="1.0" encoding="UTF-8"?>

<LifecycleConfiguration>

<Rule>

<ID>Archive and then delete rule</ID>

<Filter>

<Prefix>projectdocs/</Prefix>

</Filter>

<Status>Enabled</Status>

<Transition>

<Days>30</Days>

<StorageClass>STANDARD\_IA</StorageClass>

</Transition>

<Transition>

<Days>365</Days>

<StorageClass>GLACIER</StorageClass>

</Transition>

<Expiration>

<Days>3650</Days>

</Expiration>

</Rule>

</LifecycleConfiguration>

## DELETE Bucket Lifecycle

The DELETE Bucket Lifecycle operations removes the lifecycle configurations set on a bucket. To use this operation, you must have permission to perform the PUT Bucket lifecycle action.

The DELETE Bucket Lifecycle operation does not use request parameters and elements.

**Request Syntax**

DELETE bucketname/?lifecycle HTTP/1.1

Host: x.x.x.x

Date: date

Authorization: authorization string

**Request Sample :**

DELETE bucketname/?lifecycle HTTP/1.1

Host: x.x.x.x

Date: Wed, 14 Dec 2011 05:37:16 GMT

Authorization: signatureValue

**Response Sample :**

HTTP/1.1 204 No Content

x-amz-id-2: Uuag1LuByRx9e6j5OnimrSAMPLEtRPfTaOAa==

x-amz-request-id: 656c76696e672SAMPLE5657374

Date: Wed, 14 Dec 2011 05:37:16 GMT

Connection: keep-alive

Server: x.x.x.x

## Recover Object from Glacier Storage(Get the recover progress?)

Objects in the GLACIER storage class are archived. To access an archived object, you must first initiate a restore request. This restores a copy of the archived object. The time it takes restore jobs to finish depends on which data access tier you specify, Expedited, Standard, or Bulk.

**Speed Option**

When restoring an archived object (or using a select request), you can specify one of the following options in the Tier element of the request body:

* Expedited – Lets you quickly access your data when occasional urgent requests for a subset of archives are required. For all but the largest archived object (250 MB+), data accessed using Expedited retrievals are typically made available within 1–5 minutes.
* Standard – Lets you access any of your archived objects within several hours. Standard retrievals typically finish within 3–5 hours. This is the default tier.
* Bulk – The lowest-cost data access option in Glacier. It lets you retrieve large amounts, even petabytes, of data inexpensively in a day. Bulk access typically completes within 5–12 hours.

**Important:** OpenSDS does not guarantee those recover time for each option, because some cloud storage service does not support those options, some other only support a part of those options.

* Azure blob does not support those options.
* Huawei OBS, only support Expedited and Standard. So when you specify Bulk, then Standard will be used.

**Lifetime of the restored (active) copy**

In a restore request, you specify the number of days that you want the restored copy to exist.

But it only used for those backend which support restore period. For:

* Amazon S3 and Huawei OBS, after the specified period, the restored copy will be deleted, while the object remains archived.
* Azure, it is not used, after recovered, the object will be stored in the new storage class, not in Glacier any more.
* Google Cloud Storage, it actually no need to recover before access the object stored in Glacier, so this option is meaningless.

**Request Syntax**

POST bucketname/objectname?restore HTTP/1.1

Host: x.x.x.x

Date: date

Authorization: authorization string

Content-MD5: MD5

request body

**Request Body**

The following is an XML example of a request body for restoring an archive.

<RestoreRequest>

<Days>2</Days>

<GlacierJobParameters>

<Tier>Bulk</Tier>

</GlacierJobParameters>

</RestoreRequest>

The following table explains the XML for archive restoration in the request body.

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Required** |
| RestoreRequest | Container for restore information.  Type: Container | Yes |
| Days | Lifetime of the restored (active) copy. The minimum number of days that you can restore an object from Glacier is 1.  Type: Positive integer  Ancestors: RestoreRequest | Yes |
| GlacierJobParameters | Container for Glacier job parameters.  Do not use this element with a SELECT type of request.  Type: Container  Ancestors: RestoreRequest | No, use Standard as default. |
| Tier | The data access tier to use when restoring the archive. Standard is the default.  Type: Enum  Valid values: Expedited | Standard | Bulk  Ancestors: GlacierJobParameters | No, use Standard as default. |

**Rrequest Example:**

POST bucketname/objectname?restore HTTP/1.1

Host: x.x.x.x

Date: Mon, 22 Oct 2012 01:49:52 GMT

Authorization: authorization string

Content-Length: content length

<RestoreRequest>

<Days>2</Days>

<GlacierJobParameters>

<Tier>Expedited</Tier>

</GlacierJobParameters>

</RestoreRequest>

**Response Example:**

HTTP/1.1 202 Accepted

x-amz-id-2: GFihv3y6+kE7KG11GEkQhU7/2/cHR3Yb2fCb2S04nxI423Dqwg2XiQ0B/UZlzYQvPiBlZNRcovw=

x-amz-request-id: 9F341CD3C4BA79E0

Date: Sat, 20 Oct 2012 23:54:05 GMT

Content-Length: 0

Server: x.x.x.x

# Changes on the current implementation

## S3 module

1. Provide CRUD(Create/Read/Update/Delete) APIs of lifecycle rules on bucket.
2. Provide the API of recover object from glacier storage.
3. Provide the ability of filter objects by prefix in the list API.
4. Provide the ability of filter objects by storage class in the list API.
5. Add an API to update object metadata.
6. Add the HEAD API to get object metadata.

## scheduler module

1. Periodically get lifecycle rules of each bucket, based on the Read API of S3.
2. For each rule, if it is active, then get object list by rule filter based on the List API of S3.
3. For each object get in #2, send a LifecycleActionRequest to datamover through kafka.
4. If rules of a specific bucket are conflict (see [4.2 About conflicting lifecycle actions](#_About_conflicting_lifecycle)), scheduler need to handle it.

LifecycleActionRequest have the following parameters:

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Required** |
| **ObjKey** | Key of object. | Yes. |
| **Action** | Action need to be done. | Yes. Valid value: Delete, Transition. |
| **StorageClass** | The target storage class. | Yes, when actions is transition. |
| **SourceBackend** | Which backend object stored. | Yes, when actions is transition. |
| **TargetBackend** | Which backend object will be transition to. | Yes, when actions is transition. |

## datamover module

1. Add a LifecycleDriver which is responsible for handling the LifecycleActionRequest and .
2. For each cloud, there is a plugin.
3. For each LifecycleActionRequest, LifecycleDriver will:
   1. For delete action, call Delete API of S3 to delete object.
   2. For in-cloud transition action, call cloud plugin to change storage class. And Call API of S3 to update metadata.
   3. For cross-cloud transition action, call cloud plugin to download and upload object. And Call API of S3 to update metadata.

# Q & A

*[https://docs.google.com/document/d/1XbA5lPLlJCBKf-Kk9BCwPy3VEj1bqEF7tvlriU83tAM/edit#](https://docs.google.com/document/d/1XbA5lPLlJCBKf-Kk9BCwPy3VEj1bqEF7tvlriU83tAM/edit)*

1. **All the different cloud vendor have their Request body in different format , for example AWS S3 has it in XML format while Microsoft Azure has it in JSON format How will OpenSDS implement one format which can support all cloud vendors request body ?**

That’s where plug-in comes in , OpenSDS will implement different plug-ins for different cloud vendors which will convert the OpenSDS API request body (XML) into suitable format which is supported by mapped cloud vendor.

**2. How to decide upon costing. Do we need to advise user for what action need to be taken for transition, for example the cost comparison of all the actions ?** TBD

**3. How to control insignificant transitions in lifecycle management ?**

**For example : If user decides to transit from STANDARD\_IA of aws s3 to HOT in Azure blob, the transition is insignificant as cloud vendor will not allow this action locally(in cloud) . We need to restrict the user from doing it.**

One way is to restrict in GUI or Dashboard , while creating the transition actions user will only be able to see the valid and significant storage classes where the object can be moved.

For CLI , first we will retrieve the storage class where the bucket sits currently. And before any transition action happen we can check the destination storage class specified in rule and check if the transition is valid or not.

Other suggestion - RULE ENGINE, supports a waterfall model for transitioning between storage classes, as shown in chapter 4.1.

**4. While transitioning to other storage class across cloud or within cloud , how will user know if there is any space left in the destination Storage class.**

Actually, there is no space limitation for cloud object storage. Cloud vendor promise the storage space is enough.

**5. Several cloud vendors allow Cross-Region-Replication in their space. If a bucket or object is moved in the cloud manually by some user , how will OpenSDS lifecycle management keep track of this movement. For example if ‘bucket1’ in OpenSDS has lifecycle policy enforced on it and it is moved in cloud by user from one region to other , how will OpenSDS know about it and how will it manage the lifecycle rule even after the movement ?**

Right now, we do not support this scenario, that is a limitation. In the future we will support replication by ourselves, that is a question about sync up meta data with cloud vendor.

**6. Do we need to enforce lifecycle configuration on buckets or on objects or both?**

For bucket all cloud vendors allow lifecycle management , but AWS S3 allows to have lifecycle configured on objects in the buckets too. This is helpful while managing the versions of the objects. For example if user wants to delete/expire the previous versions of the object and to keep just the current version it should have object level lifecycle management.

Lifecycle Rules will be enforced at bucket level only , but the rules will manage objects as well.

TODO:

1. Include IBM cloud in list of supported cloud vendors.

# Appendix

## How to access object with each storage class?

For AWS S3 :

1. Object stored in each storage class except GLACIER can be accessed using the common API.
2. Object stored in GLACIER needs to be restored before using the common API to access the object.

[Restoring the object from GLACIER storage](https://aws.amazon.com/premiumsupport/knowledge-center/restore-s3-object-glacier-storage-class/)

For Azure Blob :

1. Object stored in each storage class except ARCHIVE can be accessed using the common API.
2. Object stored in ARCHIVE needs to be restored before it is used by common API for accessing.

For Google GCS :

1. Object stored in each storage class can be accessed using the common API . There is no need to restore the object before accessing it.

For Huawei OBS :

1. Object stored in each storage class except COLD can be accessed using a common API.
2. Object stored in COLD needs to be restored before using the common API to access.

For Ceph :

1. Only support one level of storage class - STANDARD