Cloudera Runtime 7.3.2

Accessing Cloud Data

Date published: 2020-07-28
Date modified: 2026-03-31

https://docs.cloudera.com/

Legal Notice

Contents

Cloud storage connectors overview

This content provides information and steps required for, using, securing, tuning performance, and troubleshooting access to the cloud storage services using Cloudera cloud storage connectors available for Amazon Web Services (AWS), Microsoft Azure (Azure) and Google Cloud Platform (GCP).

The primary audience of this content are the administrators and users of Cloudera deployed on cloud Infrastructure-as-a-Service (IaaS) such as AWS, Azure and GCP.

You may also use this content if your Cloudera is deployed in your own data center and you plan to access cloud storage through the connectors; however, your experience and performance may vary based on the network bandwidth between your data center and the cloud storage service. In addition, you can optionally configure other features where available.

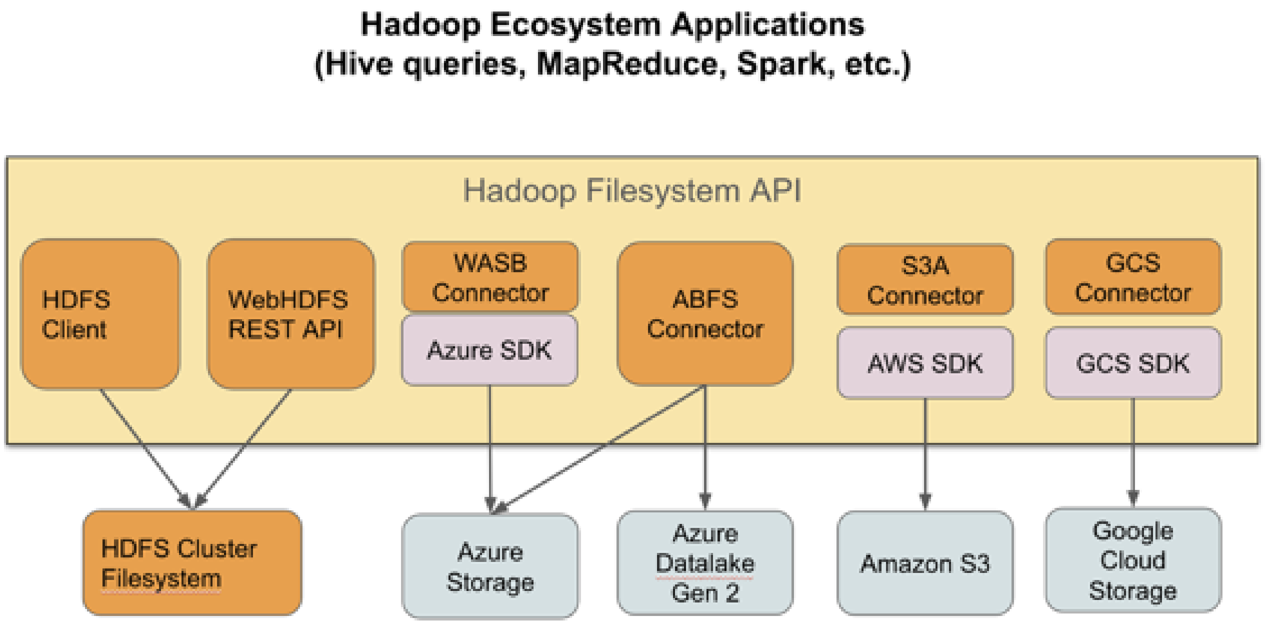
The Cloud Storage Connectors

When deploying Cloudera clusters on cloud Infrastructure-as-a-Service (IaaS), you can take advantage of the native integration with the object storage services available on Amazon Simple Storage Service (S3) in AWS, Azure Data Lake Storage (ADLS) Gen2 in Azure, and Google Cloud Storage (GCS) in Google Cloud. This integration is through cloud storage connectors included with Cloudera. Their primary function is to help you connect to, access, and work with data the cloud storage services.

The Cloud Connectors allow you to access and work with data stored in Amazon S3, ADLS Gen2, and GCS storage services including but not limited to the following use cases:

- Collect data for analysis and then load it into Hadoop ecosystem applications such as Hive or Spark directly from cloud storage services.
- Persist data to cloud storage services for use outside of Cloudera clusters.
- Copy data stored in cloud storage services to HDFS for analysis and then copy back to the cloud when done.
- Share data between multiple Cloudera clusters, and between various external non-Cloudera systems.
- Back up Cloudera clusters using distcp.

The cloud object store connectors are implemented as individual Hadoop modules. The libraries and their dependencies are automatically placed on the classpath.



Amazon S3 is an object storage service. The S3A connector implements the Hadoop filesystem interface using AWS Java SDK to access the web service, and provides Hadoop applications with a filesystem view of the buckets. Applications can manipulate data stored in Amazon S3 buckets with an URL starting with the s3a:// prefix.

ADLS Gen2 is an object storage service that combines the features of Azure Blob Storage and ADLS Gen1. ADLS Gen2 is an object store designed for large scale big-data applications, which can be treated as a hierarchical file system, and has a security model which matches that of HDFS. Applications can manipulate data store in ADLS Gen2 with with URLs starting with the abfs:// prefix.

GCS is an object storage service for unstructured data, which can be accessed through URLs beginning with the gcs:// prefix.

Table 1: Summary of Cloud Storages and Connectors

Cloud Storage Service	Connector Description	URL Prefix
Amazon S3	The S3A connector enables reading and writing files stored in the Amazon S3 object store.	s3a://
ADLS Gen2	The ABFS connector enables reading and writing files stored in the ADLS Gen2 object store.	abfs://
GCS	The GCS connector supports reading and writing data in GCS.	gcs://

Amazon S3 can not be used as a replacement for HDFS as the cluster filesystem in Cloudera. Amazon S3 can be used as a source and destination of work.

Working with Amazon S3

The Amazon S3 object store is the standard mechanism to store, retrieve, and share large quantities of data in AWS.

Cloudera recommends that you use unique S3 bucket names across all regions.

The features of Amazon S3 include:

- Object store model for storing, listing, and retrieving data.
- Support for objects up to 5 terabytes, with many petabytes of data allowed in a single "bucket".
- Data is stored in Amazon S3 in buckets which are stored in different AWS regions.
- Buckets can be restricted to different users or IAM roles.
- Data stored in an Amazon S3 bucket is billed based on the size of data how long it is stored, and on operations accessing this data. In addition, you are billed when you transfer data between regions:
    - Data transfers between an Amazon S3 bucket and a cluster running in the same region are free of download charges (except in the special case of buckets in which data is served on a user-pays basis).
    - Data downloaded from an Amazon S3 bucket located outside the region in which the bucket is hosted is billed per megabyte.
    - Data downloaded from an Amazon S3 bucket to any host over the internet is also billed per-Megabyte.
- Data stored in Amazon S3 can be backed up with Amazon Glacier.

The Hadoop client to S3, called "S3A", makes the contents of a bucket appear like a filesystem, with directories, files in the directories, and operations on directories and files. As a result, applications which can work with data stored in HDFS can also work with data stored in S3. However, since S3 is an object store, it has certain limitations that you should be aware of.

Limitations of Amazon S3

Even though Hadoop's S3A client can make an S3 bucket appear to be a Hadoop-compatible filesystem, it is still an object store, and has some limitations when acting as a Hadoop-compatible filesystem.

The key things to be aware of are:

- Operations on directories are potentially slow and non-atomic.
- Not all file operations are supported. In particular, some file operations needed by Apache HBase are not available — so HBase cannot be run on top of Amazon S3 without additional features provided by a service. HBase Object Store Semantics (HBOSS) adapter bridges the gap between HBase and the S3A filesystem.
- Except for versions of HBase specifically designed to work with S3 storage, HBase must not use s3a:// paths for HBase storage.
- S3 can not be used as a replacement for HDFS as the cluster filesystem in Cloudera. S3 can be used as a source and destination of work.
- Data is not visible in the object store until the entire output stream has been written.
- Neither the per-file and per-directory permissions supported by HDFS nor its more sophisticated ACL mechanism are supported.
- Bandwidth between your workload clusters and Amazon S3 is limited and can vary significantly depending on network and VM load.

For these reasons, while Amazon S3 can be used as the source and store for persistent data, it cannot be used as a direct replacement for a cluster-wide filesystem such as HDFS, or be used as defaultFS.

Configuring Access to S3

IDBroker is a REST API built as part of Apache Knox’s authentication services. It allows an authenticated user to exchange a set of credentials or a token for cloud vendor access tokens. It manages mapping LDAP users to FreeIPA cloud identities for data access. It performs identity mapping for access to object stores.

This section provides information about the options that are enabled by default. For information on how IDBroker works in Cloudera, see Cloud identity federation in the *Cloudera Management Console* documentation.

For Apache Hadoop applications to be able to interact with Amazon S3, they must know the AWS access key and the secret key. This can be achieved in multiple ways, including configuration properties, environment variables, and EC2 instance metadata. While the first two options can be used when accessing S3 from a cluster running in your own data center. IAM roles, which use instance metadata should be used to control access to AWS resources if your cluster is running on EC2.

Deployment Scenario	Authentication Options
Cluster runs on EC2	Use IAM roles to bind your EC2 VMs to a specific role, and so manage its access to AWS resources. IDBroker will provide authenticated users with the AWS credentials they need to access data in S3. There is no need to manually provide any AWS login credentials directly to users
Cluster runs in your own data center	Use configuration properties to authenticate. You can set the configuration properties globally or per-bucket.

Temporary security credentials, also known as "session credentials", can be issued. These consist of a secret key with a limited lifespan, along with a session token, another secret which must be known and used alongside the access key. The secret key is never passed to AWS services directly. Instead it is used tosign the URL and headers of the HTTP request.

By default, the S3A filesystem client follows the following authentication chain:

1. The AWS login details are looked for in the job configuration settings.
2. The AWS environment variables are then looked for.
3. An attempt is made to query the Amazon EC2 Instance Metadata Service to retrieve credentials published to EC2 VMs.

Configuring Access to S3 in Cloudera on cloud

IDBroker is a REST API built as part of Apache Knox's authentication services. It allows an authenticated user to exchange a set of credentials or a token for cloud vendor access tokens. It manages mapping LDAP users to FreeIPA cloud identities for data access. It performs identity mapping for access to object stores.

This section provides information about the options that are enabled by default. For information on how IDBroker works in Cloudera, see Cloud identity federation in the *Cloudera Management Console* documentation.

For Apache Hadoop applications to be able to interact with Amazon S3, they must know the AWS access key and the secret key. This can be achieved in multiple ways, including configuration properties, environment variables, and EC2 instance metadata. While the first two options can be used when accessing S3 from a cluster running in your own data center. IAM roles, which use instance metadata should be used to control access to AWS resources if your cluster is running on EC2.

Deployment Scenario	Authentication Options
Cluster runs on EC2	Use IAM roles to bind your EC2 VMs to a specific role, and so manage its access to AWS resources. IDBroker will provide authenticated users with the AWS credentials they need to access data in S3. There is no need to manually provide any AWS login credentials directly to users
Cluster runs in your own data center	Use configuration properties to authenticate. You can set the configuration properties globally or per-bucket.

Temporary security credentials, also known as "session credentials", can be issued. These consist of a secret key with a limited lifespan, along with a session token, another secret which must be known and used alongside the access key. The secret key is never passed to AWS services directly. Instead it is used tosign the URL and headers of the HTTP request.

By default, the S3A filesystem client follows the following authentication chain:

1. The AWS login details are looked for in the job configuration settings.
2. The AWS environment variables are then looked for.
3. An attempt is made to query the Amazon EC2 Instance Metadata Service to retrieve credentials published to EC2 VMs.

Using EC2 Instance Metadata to Authenticate

If your cluster is running on EC2, the standard way to manage access is via Amazon Identity and Access Management (IAM),which allows you to create users, groups, and roles to control access to services such as Amazon S3 via attached policies.

A role does not have any credentials such as password or access keys associated with it. Instead, if a user is assigned to a role, access keys are generated dynamically and provided to the user when needed. For more information, refer to IAM Roles for Amazon EC2 in Amazon documentation.

When launching your cluster on EC2, specify an IAM role that you want to use; if you are planning to use S3 with your cluster, make sure that the role associated with the cluster includes a policy that grants access to S3. For more information, refer to Using an IAM Role to Grant Permissions to Applications Running on Amazon EC2 Instances in Amazon documentation. No additional configuration is required.

Note: You can use IAM Roles to control access to keys stored in Amazon's KMS Key Management service. For more information, refer to Overview of Managing Access to Your AWS KMS Resources in Amazon documentation.

Referencing S3 Data in Applications

You can reference data in Amazon S3 using a URL starting with the s3a:// prefix followed by bucket name and path to file or directory.

The URL structure is:

```
s3a://<bucket>/<dir>/<file>
```

For example, to access a file called "mytestfile" in a directory called "mytestdir", which is stored in a bucket called "mytestbucket", the URL is:

```
s3a://mytestbucket/mytestdir/mytestfile
```

The following FileSystem shell commands demonstrate access to a bucket named mytestbucket:

```
hadoop fs -ls s3a://mytestbucket/

hadoop fs -mkdir s3a://mytestbucket/testDir

hadoop fs -touch s3a://mytestbucket/testFile

hadoop fs -cat s3a://mytestbucket/testFile
```

Configuring Per-Bucket Settings

You can specify bucket-specific configuration values which override the common configuration values.

1. Authentication mechanisms and credentials
2. Encryption settings
3. The specific S3 endpoints to send HTTP requests to. This is essential when working with S3 regions which only support the "V4 authentication API", in case of which callers must always declare the explicit region.

All fs.s3a options other than a small set of unmodifiable values (currently fs.s3a.impl) can be set on a per-bucket basis.

To set a bucket-specific option:

1. Add a new configuration, replacing the fs.s3a. prefix on an option with fs.s3a.bucket.BUCKETNAME, where BUCKETNAME is the name of the bucket.

   For example, if you are configuring access key for a bucket called "nightly", instead of using fs.s3a.access.key property name, use fs.s3a.bucket.nightly.access.key.
2. When connecting to a bucket, all options explicitly set for that bucket will override the base fs.s3a. values, but they will not be picked up by other buckets.

Example

You may have a base configuration to use the IAM role information available when deployed in Amazon EC2 VM or an AWS container service.

```
<property>
  <name>fs.s3a.aws.credentials.provider</name>
  <value>org.apache.hadoop.fs.s3a.auth.IAMInstanceCredentialsProvider</val
ue>
</property>
```

This will be the default authentication mechanism for S3A buckets.

A bucket s3a://nightly/ used for nightly data uses a session key, so its bucket-specific configuration is:

```
<property>
  <name>fs.s3a.bucket.nightly.access.key</name>
  <value>AKAACCES-SKEY-2</value>
</property>
<property>
  <name>fs.s3a.bucket.nightly.secret.key</name>
  <value>SESSION-SECRET-KEY</value>
</property>
<property>
  <name>fs.s3a.bucket.nightly.session.token</name>
  <value>SHORT-LIVED-SESSION-TOKEN</value>
</property>

<property>
  <name>fs.s3a.bucket.nightly.aws.credentials.provider</name>
  <value>org.apache.hadoop.fs.s3a.TemporaryAWSCredentialsProvider</value>
</property>
```

Finally, the public s3a://landsat-pds/ bucket could be accessed anonymously, so its bucket-specific configuration is:

```
<property>
   <name>fs.s3a.bucket.landsat-pds.aws.credentials.provider</name>
   <value>org.apache.hadoop.fs.s3a.AnonymousAWSCredentialsProvider</value>
</property>
```

For all other buckets, the base configuration is used.

Customizing Per-Bucket Secrets Held in Credential Files

JCEKs credential files support the same per-bucket settings as those in XML files. To provide different credentials for different buckets, simply create par-bucket entries in the JCEKS file with the relevant secrets.

Example

1. Set base properties for fs.s3a.secret.key and fs.s3a.access.key in the JCEKS file. These will be the default.
2. Set all non-security properties per-bucket for a bucket called "frankfurt-1" in the core-site.xml. These will override the base properties when talking to the bucket "frankfurt-1".
3. For the AWS authentication secrets, set the fs.s3a.frankfurt-1.access.key and fs.s3a.frankfurt-1.secret.key properties in the JCEKS file.

Related Information

Credential Provider API Guide

Configuring Per-Bucket Settings to Access Data Around the World

S3 buckets are hosted in different AWS regions, the default being "US-East". The S3A client talks to this region by default, issuing HTTP requests to the server s3.amazonaws.com. This central endpoint can be used for accessing any bucket in any region which supports using the V2 Authentication API, albeit possibly at a reduced performance.

Each region has its own S3 endpoint, documented by Amazon. The S3A client supports these endpoints. While it is generally simpler to use the default endpoint, direct connections to specific regions (i.e. connections via region's own endpoint) may deliver performance and availability improvements, and are mandatory when working with the most recently deployed regions, such as Frankfurt and Seoul. You can use `fs.s3a.endpoint.region` to explicitly set the region for the bucket.

When deciding which endpoint to use, consider the following:

1. Applications running in EC2 infrastructure do not pay for data transfers to or from local S3 buckets. In contrast, they will be billed for access to remote buckets. Therefore, wherever possible, always use local buckets and local copies of data.
2. When the V1 request signing protocol is used, the default S3 endpoint can support data transfer with any bucket.
3. When the V4 request signing protocol is used, AWS requires the explicit region endpoint to be used — hence S3A must be configured to use the specific endpoint. This is done in the configuration option fs.s3a.endpoint.
4. All endpoints other than the default endpoint only support interaction with buckets local to that S3 instance.

If the wrong endpoint is used, the request may fail. This may be reported as a 301 redirect error, or as a 400 Bad Request. Take these failures as cues to check the endpoint setting of a bucket.

The up to date list of endpoints is provided by Amazon: https://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region

Example

The following examples show per-bucket endpoints set for the "landsat-pds" and "eu-dataset" buckets, with the endpoints set to the default central endpoint and EU/Ireland, respectively:

```
<property>
  <name>fs.s3a.bucket.landsat-pds.endpoint</name>
  <value>s3.amazonaws.com</value>
  <description>The endpoint for s3a://landsat-pds URLs</description>
</property>
<property>
  <name>fs.s3a.bucket.eu-dataset.endpoint</name>
  <value>s3-eu-west-1.amazonaws.com</value>
  <description>The endpoint for s3a://eu-dataset URLs</description>
</property>
```

The following example shows how to set the region for a bucket as ap-south-1:

```
<property>
  <name>fs.s3a.endpoint.region</name>
  <value>ap-south-1</value>
  <description>AWS S3 region for a bucket, which bypasses the parsing o
f fs.s3a.endpoint to know the region. This helps in avoiding errors while
 using privateLink URL and explicitly setting the bucket region.
  </description>
</property>
```

Explicitly declaring a bucket bound to the central endpoint ensures that if the default endpoint is changed to a new region, data stored in US-east is still reachable.

Encrypting Data on S3

The S3A filesystem client supports Amazon S3's server-side and client-side encryption mechanisms to better secure the data in S3.

Encryption on S3

- In Server-Side Encryption (SSE), the data is encrypted before it is saved to disk in S3, and decrypted when it is read. This encryption and decryption takes place in the S3 infrastructure, and is transparent to (authenticated) clients.
- In Client-Side Encryption (CSE), the data is encrypted and decrypted on the client, that is, inside the AWS S3 SDK.

In general, the specific configuration mechanism can be set via the property fs.s3a.server-side-encryption-algorithm in core-site.xml. However, some encryption options require extra settings. The encryption method configured in core-site.xml applies cluster wide. Any new file written will be encrypted with this encryption configuration. When the S3A client reads a file, S3 will attempt to decrypt it using the mechanism and keys with which the file was encrypted.

It is also possible to configure encryption for specific buckets and to mandate encryption for a specific S3 bucket.

Note the following:

- It is NOT advised to mix and match encryption types in a bucket.
- It is much simpler and safer to encrypt with just one type and key per bucket.
- You can use AWS bucket policies to mandate encryption rules for a bucket.
- You can use S3A per-bucket configuration to ensure that S3A clients use encryption policies consistent with the mandated rules.
- You can use S3 Default Encryption in the AWS console to encrypt data without needing to set anything in the client.
- Changing the encryption options on the client does not change how existing files were encrypted, except when the files are renamed.
- For all mechanisms other than SSE-C and CSE-KMS, clients do not need any configuration options set in order to read encrypted data: it is all automatically handled in S3 itself.
- Encryption options and secrets are collected by S3A Delegation Tokens and passed to workers during job submission.
- Encryption options and secrets MAY be stored in JCEKS files or any other Hadoop credential provider service. This allows for more secure storage than XML files, including password protection of the secrets.

Server-side encryption

AWS S3 supports server-side encryption inside the storage system itself. When an S3 client uploading data requests data to be encrypted, then an encryption key is used to encrypt the data as it saved to S3. It remains encrypted on S3 until deleted and clients cannot change the encryption attributes of an object once uploaded.

The server-side "SSE" encryption is performed with symmetric AES256 encryption; S3 offers different mechanisms for defining the key to use.

For server-side encryption to work, the S3 servers require secret keys to encrypt data, and the same secret keys to decrypt it. These keys can be managed in three ways:

- SSE-S3: By using Amazon S3-Managed Keys
- SSE-KMS: By using AWS Key Management Service
- SSE-C: By using customer-supplied keys

Client-side encryption

Caution: Client-side encryption (CSE-KMS) is an experimental feature. Do not mix encrypted and unencrypted data in a CSE-enabled configuration; instead maintain separate buckets. CSE will slow down performance due to the linear nature of multi-part uploads.

Client-side encryption encrypts the data on the client, before transmitting to S3, where it is stored encrypted. The data is unencrypted after downloading, when it is being read back.

In CSE-KMS, the ID of an AWS-KMS key is provided to the S3A client; the client communicates with AWS-KMS to request a new encryption key, which KMS returns along with the same key encrypted with the KMS key. The S3 client encrypts the payload and attaches the KMS-encrypted version of the key as a header to the object.

When downloading data, this header is extracted, passed to AWS KMS, and, if the client has the appropriate permissions, the symmetric key is retrieved and returned. This key is then used to decode the data.

Related Information

Troubleshooting S3

Using server-side encryption with Amazon S3 managed keys (SSE-S3) | AWS

Using server-side encryption with AWS KMS keys (SSE-KMS) | AWS

Using server-side encryption with customer-provided keys (SSE-C) | AWS

SSE-S3: Amazon S3-Managed Encryption Keys

In SSE-S3, all keys and secrets are managed inside S3. This is the simplest encryption mechanism.

Related Information

Using server-side encryption with Amazon S3 managed keys (SSE-S3) | AWS

Enabling SSE-S3

To write S3-SSE encrypted files, the value of fs.s3a.server-side-encryption-algorithm must be set to that of the encryption mechanism used in core-site.xml; currently only AES256 is supported.

```
<property>
  <name>fs.s3a.server-side-encryption-algorithm</name>
  <value>AES256</value>
</property>
```

Once set, all new data will be uploaded encrypted. There is no need to set this property when downloading data — the data will be automatically decrypted when read using the Amazon S3-managed key.

To learn more, refer to Protecting Data Using Server-Side Encryption with Amazon S3-Managed Encryption Keys (SSE-S3) in AWS documentation.

Note: When encrypted files are renamed, they are de-encrypted and then re-encrypted with the encryption settings algorithm of the client application performing the rename. If a client with encryption disabled renames an encrypted file, the new file will be unencrypted.

SSE-KMS: Amazon S3-KMS Managed Encryption Keys

Amazon offers a pay-per-use key management service, AWS KMS. This service can be used to encrypt data on S3 using keys which can be centrally managed and assigned to specific roles and IAM accounts.

The AWS KMS can be used by S3 to encrypt uploaded data. When uploading data encrypted with SSE-KMS, the named key that was used to encrypt the data is retrieved from the KMS service, and used to encode the per-object secret which encrypts the uploaded data. To decode the data, the same key must be retrieved from KMS and used to unencrypt the per-object secret key, which is then used to decode the actual file.

KMS keys can be managed by an organization's administrators in AWS, including having access permissions assigned and removed from specific users, groups, and IAM roles. Only those "principals" with granted rights to a key may access it, hence only they may encrypt data with the key, and decrypt data encrypted with it. This allows KMS to be used to provide a cryptographically secure access control mechanism for data stores on S3.

Note:

AWS KMS service is not related to the Key Management Service built into Hadoop (Hadoop KMS). The Hadoop KMS primarily focuses on managing keys for HDFS Transparent Encryption. Similarly, HDFS encryption is unrelated to S3 data encryption.

Related Information

Using server-side encryption with AWS KMS keys (SSE-KMS) | AWS

Enabling SSE-KMS

To enable SSE-KMS, the property fs.s3a.server-side-encryption-algorithm must be set to SSE-KMS in core-site.xml.

```
<property>
  <name>fs.s3a.server-side-encryption-algorithm</name>
  <value>SSE-KMS</value>
</property>
```

The ID of the specific key used to encrypt the data should also be set in the property fs.s3a.server-side-encryptio n.key:

```
<property>
<name>fs.s3a.server-side-encryption.key</name>
<value>arn:aws:kms:us-west-2:360379543683:key/071a86ff-8881-4ba0-9230-95af
6d01ca01</value>
</property>
```

If your account is set up set up with a default KMS key and fs.s3a.server-side-encryption.key is unset, the default key will be used.

Alternatively, organizations may define a default key in the Amazon KMS; if a default key is set, then it will be used whenever SSE-KMS encryption is chosen and the value of fs.s3a.server-side-encryption.key is empty.

Note: AWS Key Management Service (KMS) is pay-per-use, working with data encrypted via KMS keys incurs extra charges during data I/O.

To learn more, refer to Protecting Data Using Server-Side Encryption with AWS KMS-Managed Keys (SSE-KMS) in the AWS documentation.

Note: The AWS KMS key used must be in the same region as the S3 Bucket where the data is being encrypted.

Note: The KMS key specified is only used when writing new data, or when renaming objects. When reading an existing object, whichever KMS key was used to encrypt the data is reused to decrypt it.

IAM Role permissions for working with SSE-KMS

All IAM roles which need to read data encrypted with SSE-KMS must have the permissions to decrypt using the specific key the data was encrypted with: kms:Decrypt

All IAM roles which need to both read and write data need the encrypt and decrypt permissions (encrypt-only permission is not supported).

```
kms:Decrypt
kms:GenerateDatakey
```

If a role does not have the permissions to read data, it will fail with an java.nio.AccessDeniedException.

Note: The renaming files require the permission to decrypt the data, as it is decrypted and then re-encrypted as it is copied. See AWS KMS API Permissions: Actions and Resources Reference for more details on KMS permissions.

SSE-C: Server-Side Encryption with Customer-Provided Encryption Keys

In SSE-C, the client supplies the secret key needed to read and write data.

The same SSE-C key must be used on all clients reading or writing the data; this must be set client-side, in the hadoop configuration.

For hadoop applications to work reliably, SSE-C with a common key *must* be used exclusively within a bucket if it is to be used at all. This is the only way to ensure that path and directory listings do not fail with "Bad Request" errors.

Related Information

Using server-side encryption with customer-provided keys (SSE-C) | AWS

Enabling SSE-C

To use SSE-C, the configuration option fs.s3a.server-side-encryption-algorithm must be set to SSE-C, and a base-64 encoding of the key placed in fs.s3a.server-side-encryption.key.

```
<property>
  <name>fs.s3a.server-side-encryption-algorithm</name>
  <value>SSE-C</value>
</property>

<property>
  <name>fs.s3a.server-side-encryption.key</name>
  <value>RG8gbm90IGV2ZXIgbG9nIHRoaXMga2V5IG9yIG90aGVyd2lzZSBzaGFyZSBpdA==</
value>
</property>
```

This property can be set in a Hadoop JCEKS credential file, which is significantly more secure than embedding secrets in the XML configuration file.

CSE-KMS: Amazon S3-KMS managed encryption keys

Amazon S3 Client Side Encryption (S3-CSE) is used to encrypt data on the client side and then transmit it to S3 storage. The same encrypted data is transmitted to the client while reading and then decrypted on the client side.

Introduction

S3-CSE uses AmazonS3EncryptionClientV2.java as the Amazon S3 client. The encryption and decryption is done by AWS SDK. Currently only the CSE-KMS method of client-side encryption is supported.

Previously, client-side encryption was unavailable due to the AWS S3 client padding uploaded objects with a 16 byte footer. This meant that files were shorter when being read than when are listed them through any of the list API calls/ getFileStatus(). This broke many applications, including anything seeking near the end of a file to read a footer, as ORC and Parquet do.

There is now a workaround: compensate for the footer in listings when CSE is enabled.

- When listing files and directories, 16 bytes are subtracted from the length of all non-empty objects (greater than or equal to 16 bytes).
- Directory markers MAY be longer than 0 bytes long.

The length of files when listed through the S3A client is now going to be shorter than the length of files listed with other clients--including S3A clients where S3-CSE has not been enabled.

Features

- Supports client-side encryption with keys managed in AWS KMS.

- Encryption settings propagated into jobs through any issued delegation tokens.
- Encryption information stored as headers in the uploaded object.

Limitations

- Performance will be reduced. All encrypt/decrypt is now being done on the client.
- Writing files may be slower, as only a single block can be encrypted and uploaded at a time.
- Multipart Uploader API is disabled.
- S3 Select is not supported.
- Multipart uploads would be serial, and partSize must be a multiple of 16 bytes.
- Maximum message size in bytes that can be encrypted under this mode is 2^36-32, or ~64G, due to the security limitation of AES/GCM as recommended by NIST.

Enabling CSE-KMS

To enable CSE-KMS, the property fs.s3a.server-side-encryption-algorithm must be set to CSE-KMS in core-site.xml.

- Generate an AWS KMS Key ID from the AWS console for your bucket, with the same region as the storage bucket.
- If already created, view the KMS key ID following these steps.
- Set fs.s3a.server-side-encryption-algorithm=CSE-KMS.
- Set fs.s3a.server-side-encryption.key=<KMS_KEY_ID>.

Note: If fs.s3a.server-side-encryption-algorithm=CSE-KMS is set, the fs.s3a.server-side-encryption.key=<KMS_KEY_ID> property must be set for S3-CSE to work.

```
<property>
     <name>fs.s3a.server-side-encryption-algorithm</name>
     <value>CSE-KMS</value>
 </property>

 <property>
     <name>fs.s3a.server-side-encryption.key</name>
     <value>${KMS_KEY_ID}</value>
 </property>
```

Configuring Encryption for Specific Buckets

S3A's per-bucket configuration mechanism can be used to configure the encryption mechanism and credentials for specific buckets.

For example, to access the bucket called "production" using SSE-KMS with the key ID arn:aws:kms:us-west-2:360379543683:key/071a86ff-8881-4ba0-9230-95af6d01ca01, the settings are as follows:

```
<property>
  <name>fs.s3a.bucket.production.server-side-encryption-algorithm</name>
  <value>SSE-KMS</value>
</property>
```

```
<property>
  <name>fs.s3a.bucket.production.server-side-encryption.key</name>
  <value>arn:aws:kms:us-west-2:360379543683:key/071a86ff-8881-4ba0-9230-95
af6d01ca01</value>
</property>
```

Encrypting an S3 Bucket with Amazon S3 Default Encryption

To guarantee that all data uploaded to a bucket is encrypted, it is possible to set a default encryption option for a bucket in the AWS management console.

For more information, see Amazon S3 Default Encryption for S3 Buckets.

- This does not encrypt any data already stored in the bucket.
- S3A clients can still be configured to use a different encryption option if desired; this is the default value to use if no other policy was set.

A default encryption across a bucket offers significant benefits:

- It guarantees that all clients uploading data have encryption enabled.
- It guarantees that when a file is renamed, it will be re-encrypted, even if the client does not explicitly request encryption.
- If applied to an empty bucket, it guarantees that all future uploaded data in the bucket is encrypted.

We recommend selecting an encryption policy for a bucket when the bucket is created, and setting it in the bucket policy. This stops misconfigured clients from unintentionally uploading unencrypted data, or decrypting data when renaming files.

Performance Impact of Encryption

Server Side encryption slightly slows down performance when reading data from S3, both in the reading of data during the execution of a query, and in scanning the files prior to the actual scheduling of work.

Amazon throttles reads and writes of S3-SSE data, which results in a significantly lower throughput than normal S3 IO requests. The default rate, 600 requests/minute, means that at most ten objects per second can be read or written using SSE-KMS per second — across an entire hadoop cluster (or the entire customer account). The default limits may be suitable during development — but in large scale production applications the limits may rapidly be reached. Contact Amazon to increase capacity.

Safely Writing to S3 Through the S3A Committers

The S3A committers are three different committers used to commit work directly to Mapreduce and Spark. The committers are enabled by default for Spark in Cloudera.

Introducing the S3A Committers

Amazon's S3 Object Store is not a filesystem: some expected behaviors of a filesystem are missing.

Some of the following aspects of S3 that are different from a filesystem are as follows:

- Directory listings are only eventually consistent.
- File overwrites and deletes are only eventually consistent: readers may get old data.
- There is no rename operation; it is mimicked in the S3A client by listing a directory and copying each file underneath, one-by-one.

Because directory rename is mimicked by listing and then copying files the eventual consistency of both listing and reading fails may result in incorrect data. And, because of the copying: it is slow.

The normal means by which Hadoop MapReduce and Apache Spark commit work from multiple tasks is through renaming the output. Each task attempt writes to a private task attempt directory; when the task is given permission to commit by the MapReduce Application Master or Spark Driver, this task attempt directory is renamed into the job attempt directory. When the job is ready to commit, the output of all the tasks is merged into the final output directory, again by renaming files and directories.

This is fast and safe on HDFS and similar filesystems, and on object stores with rename operations, such as Azure WASB. The time to commit work to S3 is proportional to the amount of data written. An operation which may work well during development can turn out to be unusable in production.

To address this, S3A committers were developed. They allow the output of MapReduce and Spark jobs to be written directly to S3, with a time to commit the job independent of the amount of data created.

What Are the S3A Committers?

The S3A committers are three different committers which can be used to commit work directly to Map-reduce and Spark. They differ in how they deal with conflict and how they upload data to the destination bucket —but underneath they all share much of the same code.

They rely on a specific S3 feature: multipart upload of large files.

When writing a large object to S3, S3A and other S3 clients use a mechanism called “Multipart Upload”.

The caller initiates a “multipart upload request”, listing the destination path and receiving an upload ID to use in the upload operations.

The caller then uploads the data in a series of HTTP POST requests, closing with a final POST listing the blocks uploaded.

The uploaded data is not visible until that final POST request is made, at which point it is published in a single atomic operation.

This mechanism is always used by S3A whenever it writes large files; the size of each part is set to the value of fs.s3a.multipart.size

The S3A Committers use the same multipart upload process, but convert it into a mechanism for committing the work of tasks because of a special feature of the mechanism: The final POST request does not need to be issued by the same process or host which uploaded the data.

The output of each worker task can be uploaded to S3 as a multipart upload, but without issuing the final POST request to complete the upload. Instead, all the information needed to issue that request is saved to a cluster-wide filesystem (HDFS or potentially S3 itself)

When a job is committed, this information is loaded, and the upload completed. If a a task is aborted of fails: the upload is not completed, so the output does not become visible. If the entire job fails, none of its output becomes visible.

For further reading, see:

- HADOOP-13786: Add S3A committers for zero-rename commits to S3 endpoints.
- S3A Committers: Architecture and Implementation.
- A Zero-Rename Committer: Object-storage as a Destination for Apache Hadoop and Spark.

The Three Committers

The three different S3A committers are directory committer, partitioned committer, magic committer.

- Directory Committer: Buffers working data to the local disk, uses HDFS to propagate commit information from tasks to job committer, and manages conflict across the entire destination directory tree.
- Partitioned Committer: Identical to the Directory committer except that conflict is managed on a partition-by-partition basis. This allows it to be used for in-place updates of existing datasets. It is only suitable for use with Spark.
- Magic Committer: Data is written directly to S3, but “magically” re-targeted at the final destination. Conflict is managed across the directory tree. It requires a consistent S3 object store.

We currently recommend use of the “Directory” committer: it is the simplest of the set, and by using HDFS to propagate data from workers to the job committer, this makes it easier to set up.

The rest of the documentation only covers the Directory Committer: to explore the other options, consult the Apache documentation.

Configuring Directories for Intermediate Data

In addition to fs.s3a.committer.name, two other core-site.xml configuration options are used to control where intermediate is stored.

A location is in the local filesystem for buffering data

```
<property>
  <name>fs.s3a.buffer.dir</name>
  <value>${hadoop.tmp.dir}/s3a</value>
  <description>Comma separated list of directories that will be used to b
uffer file
  uploads to.</description>
</property>
```

These directories will store the output created by all active tasks until each task is committed; the more worker processes/spark worker threads a host can support, the more disk space will be needed. Multiple disks can be listed to help spread the load, and recover from disk failure.

A location in the cluster's HDFS filesystem to share summary data about pending uploads.

```
<property>
  <name>fs.s3a.committer.staging.tmp.path</name>
  <value>tmp/staging</value>
</property>
```

These files are generally quite small: a few kilobytes per task.

Using the Directory Committer in MapReduce

Once the propertyfs.s3a.committer.name is set, Hadoop MapReduce jobs writing to paths using the s3a:// schema will automatically switch to the new committers.

Jobs using any other filesystem as a destination will still use the normal file committer.

Verifying That an S3A Committer Was Used

When working correctly, the only sign the new committers are in use is that it should be faster to use S3 as a destination of work.

There is a straightforward way to determine if a new committer was used: examine the _SUCCESS file created in the destination directory of a query. With the original file committer, this is a zero-byte file. The new S3A committers all write a JSON file describing the committer used, the files created and various diagnostics information.

Listing this file is enough to show whether an S3A committer was used:

```
hadoop fs -ls s3a://guarded-bucket/datasets/orc/_SUCCESS
```

If this file is of size 0. then no S3A committer was used. If the file length is greater than zero, then an S3A committer was used.

To see more details about the job commit operation, the file's contents can be printed.

```
hadoop fs -cat s3a://guarded-bucket/datasets/orc/_SUCCESS
```

Cleaning up after failed jobs

The S3A committers upload data in the tasks, completing the uploads when the job is committed.

About this task

Amazon AWS still bill you for all data held in this “pending” state. You can use bucket lifecycle rules to automate the cleanup of these jobs.

Procedure

1. Go to the AWS S3 console: https://s3.console.aws.amazon.com/.

2. Find the bucket you are using as a destination of work.
3. Select the Management tab.
4. Select Add a new lifecycle rule.
5. Create a rule “cleanup uploads” with no filter, and without any “transitions”.

   Configure an “Expiration” action of Clean up incomplete multipart uploads.
6. Select a time limit for outstanding uploads, such as 1 Day.
7. Review and confirm the lifecycle rule

   You need to select a limit of how long uploads can be outstanding. For Hadoop applications, this is the maximum time that either an application can write to the same file and the maximum time which a job may take. If the timeout is shorter than either of these, then programs are likely to fail.

   Once the rule is set, the cleanup is automatic.

Advanced Committer Configuration

The Apache documentation covers the full set of configuration options for the committers.

Enabling Speculative Execution

The S3A committers all support speculative execution.

For MapReduce, enable the following properties in the job:

```
mapreduce.map.speculative true
mapreduce.reduce.speculative true
```

For Spark, set the following configuration option:

```
spark.speculation true
```

Using Unique Filenames to Avoid File Update Inconsistency

When updating existing datasets, if a new file overwrites an existing file of the same name, S3's eventual consistency on file updates means that the old data may still be returned.

To avoid this, unique filenames should be used when creating files. The property fs.s3a.committer.staging.unique-file names enables this.

```
<property>
  <name>fs.s3a.committer.staging.unique-filename</name>
  <value>true</value>
</property>
```

It is set to true by default; you only need to disable it if you explicitly want newly created files to have the same name as any predecessors.

Speeding up Job Commits by Increasing the Number of Threads

When an individual job writes many files to S3, the time to commit the job can increase.

It can be speeded up by increasing the value of fs.s3a.committer.threads>. However, the value of fs.s3a.connectio n.maximum must be at least this size otherwise the limit on the number of parallel uploads is still limited.

```
<property>
  <name>fs.s3a.committer.threads</name>
  <value>8</value>
</property>
<property>
  <name>fs.s3a.connection.maximum</name>
  <value>15</value>
</property>
```

Securing the S3A Committers

There are a few security considerations when using the S3A committers.

S3 Bucket Permissions

To use an S3A committer, the account/role must have the same AWS permissions as needed to write to the destination bucket.

The multipart upload list/abort operations may be a new addition to the permissions for the active role.

Local Filesystem Security

All the committers use the directories listed in fs.s3a.buffer.dir to store staged/buffered output before uploading it to S3.

To ensure that other processes running on the same host do not have access to this data, unique paths should be used per-account.

This requirement holds for all applications working with S3 through the S3A connector.

HDFS Security

The directory and partitioned committers use HDFS to propagate commit information between workers and the job committer.

These intermediate files are saved into a job-specific directory under the path ${fs.s3a.committer.staging.tmp.path}/${user} where ${user} is the name of the user running the job.

The default value of fs.s3a.committer.staging.tmp.path is tmp/staging, Which will be converted at run time to a path under the current user's home directory, essentially /user/${user}/tmp/staging/${user}/.

Provided the user's home directory has access restricted to trusted accounts, this intermediate data will be secure.

If the property fs.s3a.committer.staging.tmp.path is changed to a different location, then this path will need to be secured to protect pending jobs from tampering.

Note: This intermediate data does not contain the output, merely the listings of all pending files and the MD5 checksums of each block.

The S3A Committers and Third-Party Object Stores

The S3A committers will work with any object store which implements the AWS S3 protocols.

The directory committer requires a consistent cluster filesystem to propagate commit information from the worker processes to the job committer. This is normally done in HDFS.

If the third-party object store is consistent, then it may also be used as the cluster filesystem. Set fs.s3a.committer.staging.tmp.path to a chosen path in the store.

Limitations of the S3A Committers

There are limitations of the S3A committers associated with custom file output formats, MapReduce API output format, and non-availability of Hive support.

Custom file output formats and their committers

Output formats which implement their own committers do not automatically switch to the new committers. If such a custom committer relies on renaming files to commit output, then it will depend on a consistent view of the object store, and take time to commit output proportional to the amount of data and the number of files.

To determine if this is the case, find the subclass of org.apache.hadoop.mapreduce.lib.output.FileOutputFormat which implements the custom format, to see if it subclasses the getOutputCommitter() to return its own committer, or has a custom subclass oforg.apache.hadoop.mapreduce.lib.output.FileOutputCommitter.

It may be possible to migrate such a committer to support store-specific committers, as was done for Apache Parquet support in Spark. Here a subclass of Parquet's ParquetOutputCommitter was implemented to delegates all operations to the real committer.

MapReduce V1 API Output Format and Committers

Only the MapReduce V2 APIs underorg.apache.hadoop.mapreduce support the new committer binding mechanism. The V1 APIs under the org.apache.hadoop.mapred package only bind to the file committer and subclasses. The v1 APIs date from Hadoop 1.0 and should be considered obsolete. Please migrate to the v2 APIs, not just for the new committers, but because the V2 APIs are still being actively developed and maintained.

No Hive Support

There is currently no Hive support for the S3A committers.

Troubleshooting the S3A Committers

The Apache documentation contains information about troubleshooting the committers.

The primary issue which surfaces is actually programs not switching to the new committers. There are three common reasons for this.

- The configuration settings to switch to the new committer are not being picked up. This is particularly common in Spark, which is harder to set up.
- The program is using the older V1 MapReduce APIs. Fix: switch to the V2 API.
- The output format the program uses is explicitly creating its own committer. This can only be fixed by modifying the program.

To help debug Spark's configuration, there is an option which can be set to forcibly fail the spark query if the path output committer is used. However, the file committer is being returned.

```
spark.hadoop.pathoutputcommit.reject.fileoutput true
```

There is also the blunt-instrument approach of causing the original output committer to crash with an invalid configuration.

```
spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version 3
```

This (invalid) option ensures that if the original file committer is used, it will raise an exception.

To enable low-level logging of the committers, set the log-level of the package org.apache.hadoop.fs.s3a.commit to DEBUG. With Log4J, this can be one in log4j.properties:

```
log4j.logger.org.apache.hadoop.fs.s3a.commit=DEBUG
```

Security Model and Operations on S3

The security and permissions model of Amazon S3 is very different from this of a UNIX-style filesystem: on Amazon S3, operations which query or manipulate permissions are generally unsupported. Operations to which this applies include: chgrp, chmod, chown, getfacl, and setfacl. The related attribute commands getfattr andsetfattr are also unavailable.

In addition, operations which try to preserve permissions (for example fs -put -p) do not preserve permissions.

Although these operations are unsupported, filesystem commands which list permission and user/group details usually simulate these details. As a consequence, when interacting with read-only object stores, the permissions found in “list” and “stat” commands may indicate that the user has write access — when they do not.

Amazon S3 has a permissions model of its own. This model can be manipulated through store-specific tooling. Be aware that some of the permissions which can be set — such as write-only paths, or various permissions on the root path — may be incompatible with the S3A client. It expects full read and write access to the entire bucket with trying to write data, and may fail if it does not have these permissions.

As an example of how permissions are simulated, here is a listing of Amazon's public, read-only bucket of Landsat images:

```
$ hadoop fs -ls s3a://landsat-pds/
Found 10 items
drwxrwxrwx - mapred 0 2016-09-26 12:16 s3a://landsat-pds/L8
-rw-rw-rw- 1 mapred 23764 2015-01-28 18:13 s3a://landsat-pds/index.html
drwxrwxrwx - mapred 0 2016-09-26 12:16 s3a://landsat-pds/landsat-pds_stats
-rw-rw-rw- 1 mapred 105 2016-08-19 18:12 s3a://landsat-pds/robots.txt
-rw-rw-rw- 1 mapred 38 2016-09-26 12:16 s3a://landsat-pds/run_info.json
drwxrwxrwx - mapred 0 2016-09-26 12:16 s3a://landsat-pds/runs
-rw-rw-rw- 1 mapred 27458808 2016-09-26 12:16 s3a://landsat-pds/scene_list.
gz
drwxrwxrwx - mapred 0 2016-09-26 12:16 s3a://landsat-pds/tarq
drwxrwxrwx - mapred 0 2016-09-26 12:16 s3a://landsat-pds/tarq_corrupt
drwxrwxrwx - mapred 0 2016-09-26 12:16 s3a://landsat-pds/test
```

The following is evident from the examples:

- All files are listed as having full read/write permissions.
- All directories appear to have full rwx permissions.
- The replication count of all files is "1".
- The owner of all files and directories is declared to be the current user (mapred).
- The timestamp of all directories is actually that of the time the -ls operation was executed. This is because these directories are not actual objects in the store; they are simulated directories based on the existence of objects under their paths.

When an attempt is made to delete one of the files, the operation fails — despite the permissions shown by the ls command:

```
$ hadoop fs -rm s3a://landsat-pds/scene_list.gz
rm: s3a://landsat-pds/scene_list.gz: delete on s3a://landsat-pds/scene_list.
gz:
com.amazonaws.services.s3.model.AmazonS3Exception: Access Denied (Service: A
mazon S3;
Status Code: 403; Error Code: AccessDenied; Request ID: 1EF98D5957BCAB3D),
S3 Extended Request ID: wi3veOXFuFqWBUCJgV3Z+NQVj9gWgZVdXlPU4KBbYMsw/gA+hyh
RXcaQ+PogOsDgHh31HlTCebQ=
```

This demonstrates that the listed permissions cannot be taken as evidence of write access. Only object manipulation can determine this.

S3A and Checksums (Advanced Feature)

The S3A connector can be configured to export the HTTP etag of an object as a checksum, by setting the option fs.s3a.etag.checksum.enabled to true. When unset (the defaut), S3A objects have no checksum.

```
$ hadoop fs -touchz s3a://hwdev-bucket/src/something.txt
$ hadoop fs -checksum s3a://hwdev-bucket/src/something.txt
s3a://hwdev-bucket/src/something.txt NONE
```

Once set, S3A objects have a checksum which is created on upload.

```
$ hadoop fs -Dfs.s3a.etag.checksum.enabled=true -checksum s3a://hwdev-bucket
/src/something.txt
s3a://hwdev-bucket/src/something.txt etag 6434316438636439386630306232303
46539383030393938656366383432 3765
```

This checksum is not compatible with that or HDFS, so cannot be used to compare file versions when using the -upd ate option on DistCp between S3 and HDFS. More specifically, unless -skipcrccheck is set, the DistCP operation will fail with a checksum mismatch. However, it can be used for incremental updates within and across S3A buckets.

```
$ hadoop distcp -Dfs.s3a.etag.checksum.enabled=true --update s3a://hwdev-buc
ket/src s3a://hwdev-bucket/dest
$ hadoop fs -Dfs.s3a.etag.checksum.enabled=true -checksum s3a://hwdev-bucke
t/dest/something.txt
s3a://hwdev-bucket/src/something.txt etag 643431643863643938663030623230346
5393830303939386563663834323765
```

As the checksums match small files created as a single block, incremental updates will not copy unchanged files. For large files uploaded using multiple blocks, the checksum values may differ in which case the source file will be copied again.

A List of S3A Configuration Properties

All S3A client options are configured with options with the prefix fs.s3a.

For information about the S3A configuration properties, see General S3A client configuration.

Working with versioned S3 buckets

AWS S3 supports "Versioning" in buckets, where the bucket is configured to save older versions of objects. When an object is overwritten (or even deleted), the old version can be accessed when a request is made for the object using the original version ID.

For more information, see Using Versioning.

The S3A connector supports versioning in the following ways:

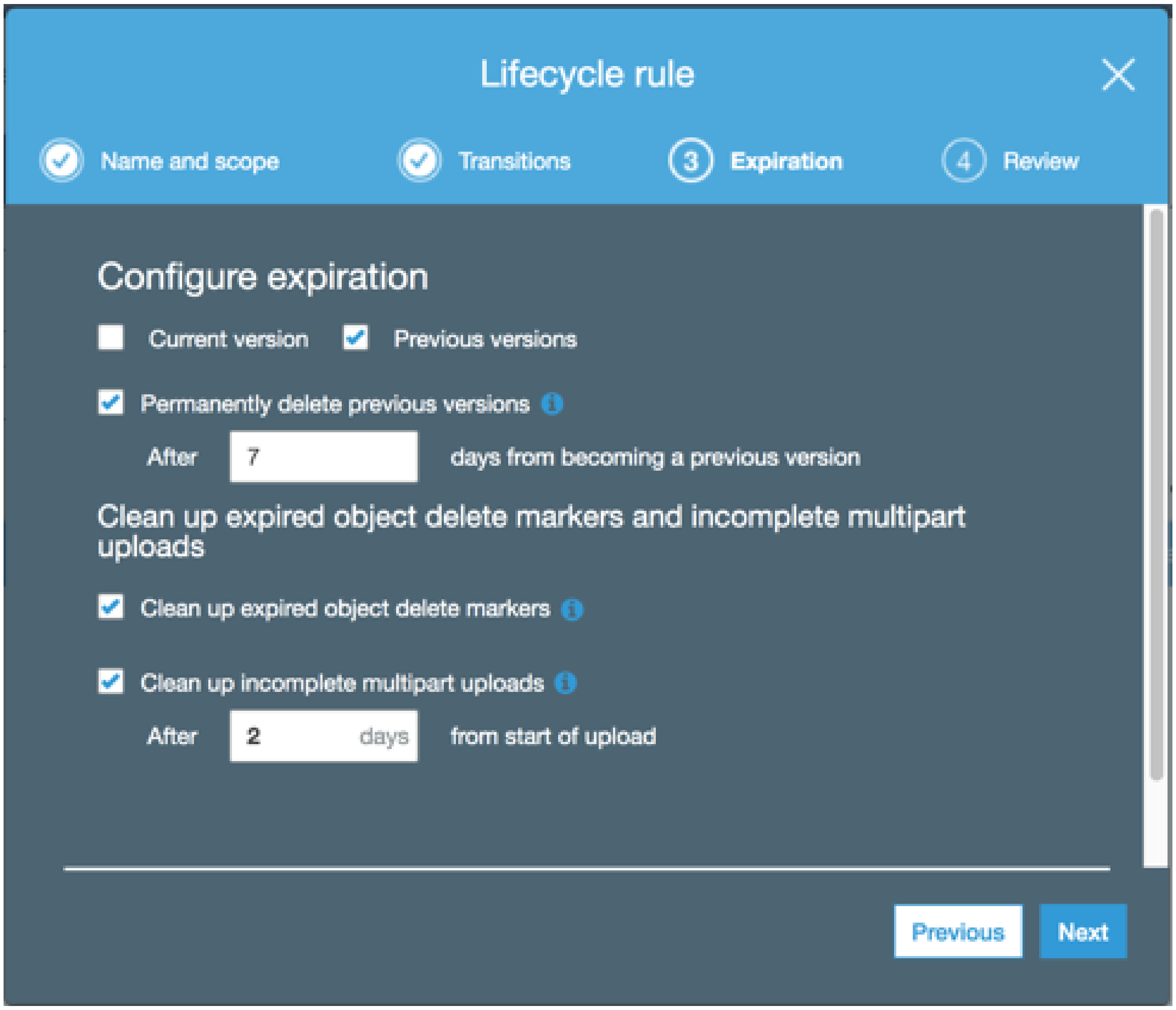
- When a file is opened for reading, the version ID of that file is recorded, and then for the duration of the file read (seconds, minutes, hours...) only that version of the data is read. Even if the file is overwritten, the single ongoing file read will always read the original data.
- When a file is copied (as in a rename operation), the version ID is used to guarantee that even if the source file is overwritten, the copied file will be the original version.
- It can be used as an alternative to moving deleted files to a trash location: simply delete the files and then recover them later. Note: The S3A connector does not provide a recovery tool.

The following are some issues you must be aware of when using versioning:

- Too many S3 Tombstone markers from deleted objects will slow down directory listings, and can result in clients being throttled (https://docs.aws.amazon.com/AmazonS3/latest/dev/troubleshooting.html#troubleshooting-by-symptom-increase-503-reponses)
- Old versions of files are still billed for. That means files overwritten by new versions are still billed by the megabyte until the previous versions are permanently deleted.
- While working with Hive managed or external table directories, staging directories are created for each table or partition every time they are queried. These staging directories are deleted after the query is complete, but copies are kept. Cloudera recommends to add appropriate S3 lifecycle rule to remove the copies (created due to versioning) after a period of time.

To keep costs down and minimize performance problems, we recommend having a lifecycle rule on the bucket which deletes older versions of files after a number of days. This can be done from the Management tab of the AWS S3 console.

Here is an example policy which deletes all versions of objects which were overwritten more than a week previously, while also cleaning up tombstone markers and incomplete file uploads:



Configure expiration

☐ Current version ☑ Previous versions

☑ Permanently delete previous versions

After 7 days from becoming a previous version

Clean up expired object delete markers and incomplete multipart uploads

☑ Clean up expired object delete markers

☑ Clean up incomplete multipart uploads

After 2 days from start of upload

Working with Third-party S3-compatible Object Stores

The S3A Connector can work with third-party object stores; some vendors test the connector against their stores — and even actively collaborate in developing the connector in the open source community.

Note: Cloudera tests the S3A Connector only for use with AWS S3. Third-party object store vendors are responsible for the integration testing of the connector with their object store solutions.

You must configure the S3A connector in Cloudera Manager to be able to use it with third-party objects stores.

1. Log in to Cloudera Manager, and select your cluster.
2. Select the Configuration tab.
3. Search for Cluster-wide Advanced Configuration Snippet (Safety Valve) for core-site.xml.

4. Use the + button to add each of the following configuration parameter and provide its value.
5. Click Save, and restart the service.

Option	Change to
fs.s3a.endpoint	(the hostname of the S3 Store)
fs.s3a.endpoint.region	Specify the AWS region for the region.
fs.s3a.path.style.access	true
fs.s3a.signing-algorithm	If the default signing mechanism is rejected, another mechanism may work from: "QueryStringSignerType", "S3SignerType", "AWS3SignerType", "AWS4SignerType", "AWS4UnsignedPayloadSignerType" and "NoOpSignerType".
fs.s3a.connection.ssl.enabled	Set to "false" if HTTP is used instead of HTTPS
fs.s3a.multiobjectdelete.enable	Set to "false" if bulk delete operations are not supported.
fs.s3a.list.version	Set to "1" if the list directories with the default "2" option fails with an error.

The following aspects must be considered when using using S3 connector with third-party object stores:

- Third-party object stores are generally consistent. The S3A Committers will still offer better performance, and should be used for MapReduce and Spark.
- Except for versions of HBase specifically designed to work with S3 storage, HBase must not use s3a:// paths for HBase storage.
- S3 can not be used as a replacement for HDFS as the cluster filesystem in Cloudera. S3 can be used as a source and destination of work.
- Encryption may or may not be supported, refer to the documentation of the third-party object store.
- Security permissions are likely to be implemented differently from the IAM role mode —again, refer to the documentation of the third-party object store to see what is available.

Improving Performance for S3A

You can consider various options for improving performance when working with data stored in Amazon S3.

The bandwidth between the Cloudera cluster and Amazon S3 is the upper limit to how fast data can be copied into S3. The further the Cloudera cluster is from the Amazon S3 installation, or the narrower the network connection is, the longer the operation will take. Even a Cloudera cluster deployed within Amazon's own infrastructure may encounter network delays from throttled VM network connections.

Network bandwidth limits notwithstanding, there are some options which can be used to tune the performance of an upload:

- Working with S3 buckets in the same AWS region on page 26
- Configuring and tuning S3A block upload on page 27

Working with S3 buckets in the same AWS region

A foundational step to getting good performance is working with buckets close to the Hadoop cluster, where "close" is measured in network terms.

Maximum performance is achieved from working with S3 buckets in the same AWS region as the cluster. For example, if your cluster is in North Virginia ("US East"), you will achieve best performance if your S3 bucket is in the same region.

In addition to improving performance, working with local buckets ensures that no bills are incurred for reading from the bucket.

Configuring and tuning S3A block upload

Because of the nature of the S3 object store, data written to an S3A OutputStream is not written incrementally — instead, by default, it is buffered to disk until the stream is closed in its close() method. This can make output slow because the execution time for OutputStream.close() is proportional to the amount of data buffered and inversely proportional to the bandwidth between the host to S3; that is O(data/bandwidth).

Other work in the same process, server, or network at the time of upload may increase the upload time.

In summary, the further the process is from the S3 store, or the smaller the EC2 VM is, the longer it will take complete the work. This can create problems in application code:

- Code often assumes that the close() call is fast; the delays can create bottlenecks in operations.
- Very slow uploads sometimes cause applications to time out - generally, threads blocking during the upload stop reporting progress, triggering timeouts.
- Streaming very large amounts of data may consume all disk space before the upload begins.

Tuning S3A Uploads

When data is written to S3, it is buffered locally and then uploaded in multi-Megabyte blocks, with the final upload taking place as the file is closed.

The following major configuration options are available for the S3A block upload options. These are used whenever data is written to S3.

Parameter	Default Value	Description
fs.s3a.multipart.size	100M	Defines the size (in bytes) of the blocks into which the upload or copy operations will be split up. A suffix from the set {K,M,G,T,P} may be used to scale the numeric value.
fs.s3a.fast.upload.active.blocks	8	Defines the maximum number of blocks a single output stream can have active uploading, or queued to the central FileSystem instance's pool of queued operations. This stops a single stream overloading the shared thread pool.
fs.s3a.buffer.dir	Empty value	A comma separated list of temporary directories use for storing blocks of data prior to their being uploaded to S3. When unset (by default), the Hadoop temporary directory hadoop.tmp.dir is used.
fs.s3a.fast.upload.buffer	disk	The fs.s3a.fast.upload.buffer determines the buffering mechanism to use when uploading data. Allowed values are: disk, array, bytebuffer: • (default) "disk" will use the directories listed in fs.s3a.buffer.dir as the location(s) to save data prior to being uploaded. • "array" uses arrays in the JVM heap. • "bytebuffer" uses off-heap memory within the JVM. Both "array" and "bytebuffer" will consume memory in a single stream up to the number of blocks set by: fs.s3a.multipart.size * fs.s3a.fast.upload.active.blocks. If using either of these mechanisms, keep this value low. The total number of threads performing work across all threads is set by fs.s3a.threads.max, with fs.s3a.max.total.tasks values setting the number of queued work items.

Note that:

- If the amount of data written to a stream is below that set in fs.s3a.multipart.size, the upload takes place after the application has written all its data.
- The maximum size of a single file in S3 is one thousand blocks, which, for uploads means 10000 * fs.s3a.multi part.size. Too A small value of fs.s3a.multipart.size can limit the maximum size of files.
- Incremental writes are not visible; the object can only be listed or read when the multipart operation completes in the close() call, which will block until the upload is completed.

Buffering uploads to disk or RAMs

This is the default buffer mechanism. The amount of data which can be buffered is limited by the amount of available disk space.

When fs.s3a.fast.upload.buffer is set to "disk", all data is buffered to local hard disks prior to upload. This minimizes the amount of memory consumed, and so eliminates heap size as the limiting factor in queued uploads.

Buffering uploads in Byte Buffers

When fs.s3a.fast.upload.buffer is set to "bytebuffer", all data is buffered in "direct" ByteBuffers prior to upload. This may be faster than buffering to disk in cases such as when disk space is small there may not be much disk space to buffer with (for example, when using "tiny" EC2 VMs).

The ByteBuffers are created in the memory of the JVM, but not in the Java Heap itself. The amount of data which can be buffered is limited by the Java runtime, the operating system, and, for YARN applications, the amount of memory requested for each container.

The slower the upload bandwidth to S3, the greater the risk of running out of memory — and so the more care is needed in tuning the upload thread settings to reduce the maximum amount of data which can be buffered awaiting upload.

Buffering Uploads with Array Buffers

When fs.s3a.fast.upload.buffer is set to "array", all data is buffered in byte arrays in the JVM's heap prior to upload. This may be faster than buffering to disk.

The amount of data which can be buffered is limited by the available size of the JVM heap heap. The slower the write bandwidth to S3, the greater the risk of heap overflows. This risk can be mitigated by tuning the upload thread settings.

Thread Tuning for S3A Data Upload

Both the array and bytebuffer buffer mechanisms can consume very large amounts of memory, on-heap or off-heap respectively. The disk buffer mechanism does not use much memory up, but it consumes hard disk capacity.

If there are many output streams being written to in a single process, the amount of memory or disk used is the multiple of all streams' active memory and disk use.

You may need to perform careful tuning to reduce the risk of running out memory, especially if the data is buffered in memory. There are a number parameters which can be tuned:

Parameter	Default Value	Description
fs.s3a.fast.upload.active.blocks	4	Maximum number of blocks a single output stream can have active (uploading, or queued to the central FileSystem instance's pool of queued operations). This stops a single stream overloading the shared thread pool.
fs.s3a.threads.max	10	The total number of threads available in the filesystem for data uploads or any other queued filesystem operation.
fs.s3a.max.total.tasks	5	The number of operations which can be queued for execution

Parameter	Default Value	Description
fs.s3a.threads.keepalivetime	60	The number of seconds a thread can be idle before being terminated.

When the maximum allowed number of active blocks of a single stream is reached, no more blocks can be uploaded from that stream until one or more of those active block uploads completes. That is, a write() call which would trigger an upload of a now full datablock will instead block until there is capacity in the queue.

Consider the following:

- As the pool of threads set in fs.s3a.threads.max is shared (and intended to be used across all threads), a larger number here can allow for more parallel operations. However, as uploads require network bandwidth, adding more threads does not guarantee speedup.
- The extra queue of tasks for the thread pool (fs.s3a.max.total.tasks) covers all ongoing background S3A operations.
- When using memory buffering, a small value of fs.s3a.fast.upload.active.blocks limits the amount of memory which can be consumed per stream.
- When using disk buffering, a larger value of fs.s3a.fast.upload.active.blocks does not consume much memory. But it may result in a large number of blocks to compete with other filesystem operations.

We recommend a low value of fs.s3a.fast.upload.active.blocks — enough to start background upload without overloading other parts of the system. Then experiment to see if higher values deliver more throughput — especially from VMs running on EC2.

Optimizing S3A read performance for different file types

The S3A filesystem client supports the notion of input policies, similar to that of the POSIX fadvise() API call. This tunes the behavior of the S3A client to optimize HTTP GET requests for reading different filetypes. To optimize HTTP GET requests, you can take advantage of the S3A input policy option fs.s3a.experimental.input.fadvise.

Policy	Description
"normal"	This starts off as "sequential": it asks for the whole file. As soon as the application tries to seek backwards in the file it switches into "random" IO mode. This is not quite as efficient for Random IO as the "random" mode, because that first read may have to be aborted. However, because it is adaptive, it is the best choice if you do not know the data formats which will be read.
"sequential" (default)	Read through the file, possibly with some short forward seeks. The whole document is requested in a single HTTP request; forward seeks within the readahead range are supported by skipping over the intermediate data. This leads to maximum read throughput, but with very expensive backward seeks.
"random"	Optimized for random IO, specifically the Hadoop `PositionedReadable` operations — though `seek(offset); read(byte_buffer)` also benefits. Rather than ask for the whole file, the range of the HTTP request is set to that of the length of data desired in the `read` operation - rounded up to the readahead value set in `setReadahead()` if necessary. By reducing the cost of closing existing HTTP requests, this is highly efficient for file IO accessing a binary file through a series of Posi tionedReadable.read() and PositionedReadable.readFully() calls. Sequential reading of a file is expensive, as now many HTTP requests must be made to read through the file.

For operations simply reading through a file (copying, DistCp, reading gzip or other compressed formats, parsing .csv files, and so on) the sequential policy is appropriate. This is the default, so you do not need to configure it.

For the specific case of high-performance random access IO (for example, accessing ORC files), you may consider using the random policy in the following circumstances:

- Data is read using the PositionedReadable API.
- There are long distance (many MB) forward seeks.
- Backward seeks are as likely as forward seeks.
- There is little or no use of single character read() calls or small read(buffer) calls.
- Applications are running close to the Amazon S3 data store; that is, the EC2 VMs on which the applications run are in the same region as the Amazon S3 bucket.

You must set the desired fadvise policy in the configuration option fs.s3a.experimental.input.fadvise when the filesystem instance is created. It can only be set on a per-filesystem basis, not on a per-file-read basis. You can set it in core-site.xml:

```
<property>
  <name>fs.s3a.experimental.input.fadvise</name>
  <value>random</value>
</property>
```

Or, you can set it in the spark-defaults.conf configuration of Spark:

```
spark.hadoop.fs.s3a.experimental.input.fadvise random
```

Be aware that this random access performance comes at the expense of sequential IO — which includes reading files compressed with gzip.

Improving S3A read performance using Vectored IO

The Hadoop Vectored IO API allows file formats like ORC and Parquet to fetch a set of data ranges in a single operation instead of issuing individual read calls for each range.

The Hadoop Vectored IO is an asynchronous API that enables the libraries to perform other tasks while waiting for the data. Different implementations of the Hadoop Vectored IO can support additional optimizations such as merging close-by data ranges and fetching data ranges in parallel from remote cloud storage. This results in a faster and more efficient data retrieval from cloud storage. The S3A connector offers a customized implementation that enables parallel and asynchronous reading of different data blocks.

You can enable Hadoop Vectored IO using hive.exec.orc.use.hadoop-vectored.api=true for Hive on ORC queries i, and parquet.hadoop.vectored.io.enabled=true for Spark on Parquet queries.

The S3A filesystem supports implementation of readVectored API using the client to provide a list of file ranges to read, which returns a future read object associated with each range. For more information about the readVectored API, see the *FSDataInputStream* specification and specification and *Hadoop Vectored IO: your data just got faster!* article.

The following configurations can be used to optimize vectored reads based on the client requirements:

Property	Default value	Description
fs.s3a.vectored.read.min.seek.size	4K	Smallest reasonable seek in bytes to group ranges together during vectored read operation.
fs.s3a.vectored.read.max.merged.size	1M	Largest merged read size in bytes to group ranges together during vectored read. Setting this value to 0 disables merging of ranges.

Property	Default value	Description
fs.s3a.vectored.active.ranged.reads	4	Maximum number of range reads a single input stream can have active (downloading, or queued) to the central FileSystem instance's pool of queued operations. This stops a single stream overloading the shared thread pool

Related Information

FSDataInputStream

Hadoop Vectored IO: your data just got faster!

S3 Performance Checklist

There are various types of checklists that you can use to ensure optimal performance when working with data in S3.

Checklist for Data

- [ ] Amazon S3 bucket is in same region as the EC2-hosted cluster.
- [ ] The directory layout is "shallow". For directory listing performance, the directory layout prefers "shallow" directory trees with many files over deep directory trees with only a few files per directory.
- [ ] The "pseudo" block size set in fs.s3a.block.size is appropriate for the work to be performed on the data.
- [ ] Copy to HDFS any data that needs to be repeatedly read to HDFS.

Checklist for Cluster Configurations

- [ ] Set yarn.scheduler.capacity.node-locality-delay to 0 to improve container launch times.

Checklist for Code

- [ ] Application does not make rename() calls. Where it does, it does not assume the operation is immediate.
- [ ] Application does not assume that delete() is near-instantaneous.
- [ ] Application uses FileSystem.listFiles(path, recursive=true) to list a directory tree.
- [ ] Application prefers forward seeks through files, rather than full random IO.
- [ ] If making "random" IO through seek() and read() sequences or and Hadoop's PositionedReadable API, fs.s3a.experimental.input.fadvise is set to random.

Troubleshooting S3

It can be a bit troublesome to get the S3A connector to work, with classpath and authentication being the usual trouble spots. Use the tips provided in the following links to troubleshoot errors.

S3: https://aws.amazon.com/premiumsupport/knowledge-center/

Related Information

Encrypting Data on S3

Working with Google Cloud Storage

The Cloudera platform allows you to configure access from a cluster to Google Cloud Storage (GCS) in order to access cloud data.

Configuring Access to Google Cloud Storage

Access to Google Cloud Storage can be configured separately for each cluster by providing the service account email address.

Access from a cluster to a Google Cloud Storage is possible through a service account. Configuring access to Google Cloud Storage involves the following steps.

Table 2: Overview of Configuring Access to Google Cloud Storage

Step	Considerations
Creating a service account on Google Cloud Platform and generating a key associated with it.	• You may need to contact your Google Cloud Platform admin in order to complete these steps. • If you already have a service account, you do not need to perform these steps as long as you are able to provide the service account key. If you have a service account but do not know the service account key, you should be able to generate a new key.
Creating a role on Google Cloud Platform with sufficient permissions to access storage buckets.	• You may need to contact your Google Cloud Platform admin in order to complete these steps. • This is a one time operation, and the same role can be used across different service accounts and storage buckets.
Modifying permissions of the Google Cloud Storage bucket so that you can access it by using your service account key.	• You may need to contact your Google Cloud Platform admin in order to complete these steps. • \You should perform these steps for each bucket that you want to access. • You do not need to perform these steps if your service account has project-wide access to all buckets on the account.
Configuring credentials via Cloudera Manager.	• These configuration steps are appropriate for a single-user cluster. • Only one configuration per cluster is recommended; that is, you should use one service account per cluster. If required, it is possible to use multiple service accounts with the same cluster; in that case, each job-specific configuration should be changed to use the desired service account.

Create a GCP Service Account

You must create a Google Cloud Platform service account and generate an access key (in the JSON format). If you are using a corporate GCP account, it is likely that only your GCP admin can perform these steps. Example steps are described below.

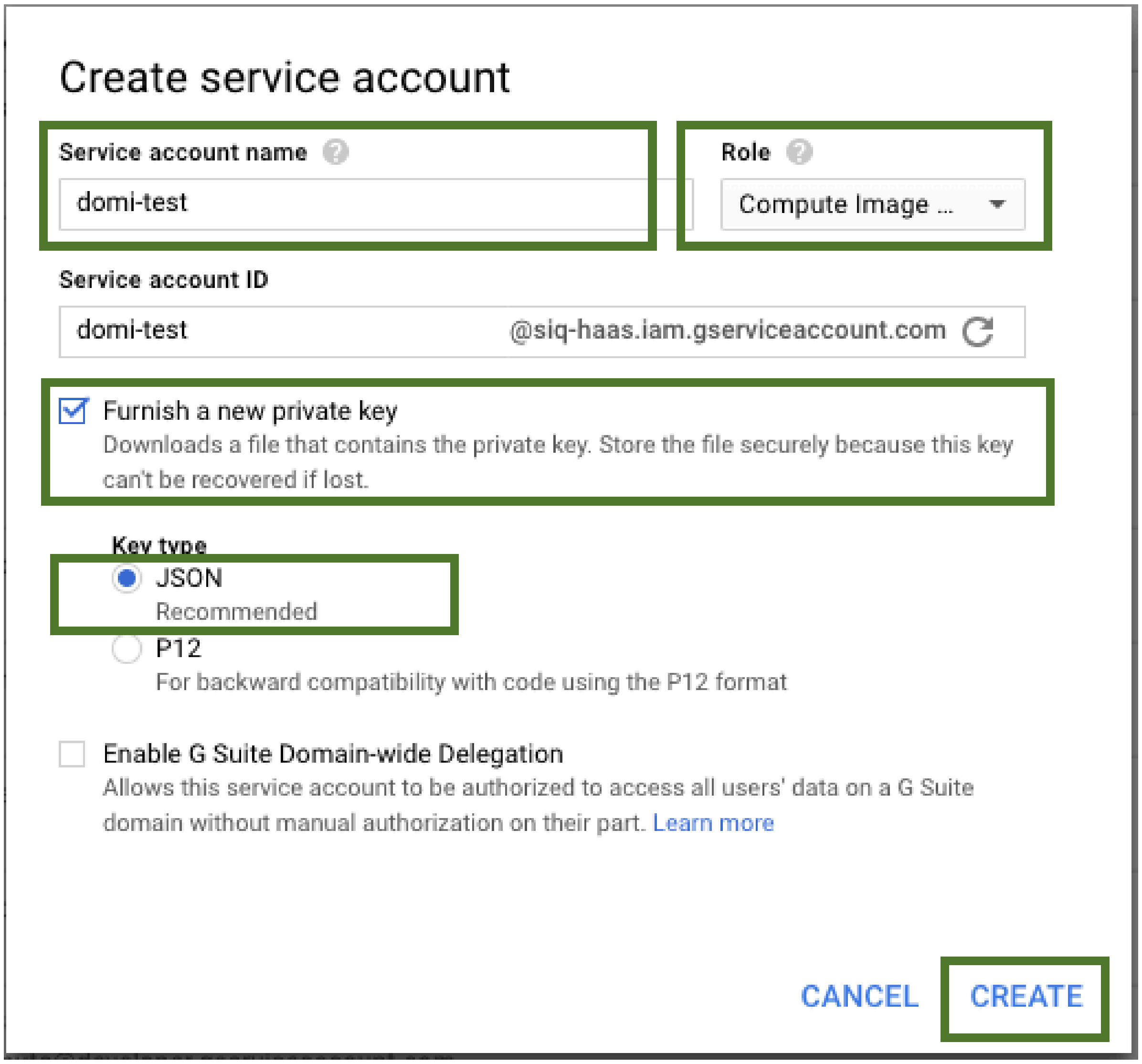
These steps assume that you have a Google Cloud Platform account. If you do not have one, you can create it at https://console.cloud.google.com.

Steps

1. In the Google Cloud Platform web console, navigate to IAM & admin Service accounts:
2. Click +Create Service Account.

3. Provide the following information:

   a. Under Service account name, provide some name for your service account.
   b. Under Role, select the project-level roles that the account should have.
   c. Check Furnish a new private key and select JSON.



Create service account

Service account name: domi-test

Role: Compute Image ...

Service account ID: domi-test @siq-haas.iam.gserviceaccount.com

[✓] Furnish a new private key
Downloads a file that contains the private key. Store the file securely because this key can't be recovered if lost.

Key type
[● JSON] Recommended
○ P12 For backward compatibility with code using the P12 format

[ ] Enable G Suite Domain-wide Delegation
Allows this service account to be authorized to access all users' data on a G Suite domain without manual authorization on their part. Learn more

4. Click Create and the file containing the key will be downloaded onto your machine. The name of the key file is usually long and contains spaces, so you may want to rename the file.

Later you will need to place this key on your cluster nodes.

Create a Custom Role

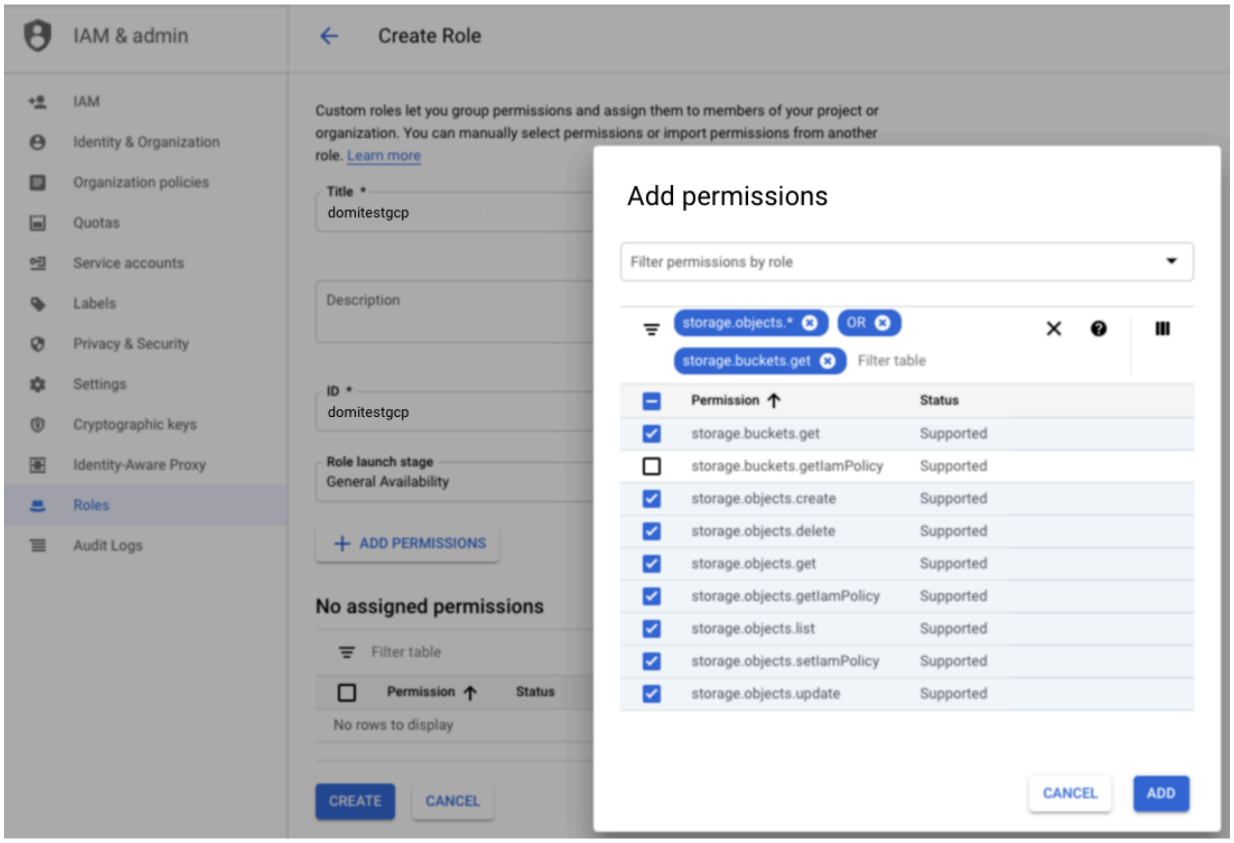
Specific permissions are required for the Google Cloud Storage Connector to access buckets. This set of permissions is the combination of the permissions associated with the existing Google Cloud IAM Role called "Storage Object Admin" and the Google Cloud IAM Permission called "storage.buckets.get".

1. In the Google Cloud Platform web console, navigate to IAM & admin > Roles
2. Click on +Create Role.
3. Provide the following:

   - Enter a title under Title
   - Enter the ID under ID
   - Under Role launch stage select General Availability

4. Click on Add Permissions and add the following permissions:

    - storage.bucket.get
    - storage.objects.create
    - storage.objects.delete
    - storage.objects.get
    - storage.objects.getIamPolicy
    - storage.objects.list
    - storage.objects.setIamPolicy
    - storage.objects.update



Custom roles let you group permissions and assign them to members of your project or organization. You can manually select permissions or import permissions from another role. Learn more

Title *: domitestgcp
Description
ID *: domitestgcp
Role launch stage: General Availability

No assigned permissions

Permission	Status
No rows to display	

Add permissions

	Permission	Status
✓	storage.buckets.get	Supported
☐	storage.buckets.getIamPolicy	Supported
✓	storage.objects.create	Supported
✓	storage.objects.delete	Supported
✓	storage.objects.get	Supported
✓	storage.objects.getIamPolicy	Supported
✓	storage.objects.list	Supported
✓	storage.objects.setIamPolicy	Supported
✓	storage.objects.update	Supported

5. Once done adding permissions, click on Create.

After performing these steps, a new role will be created.

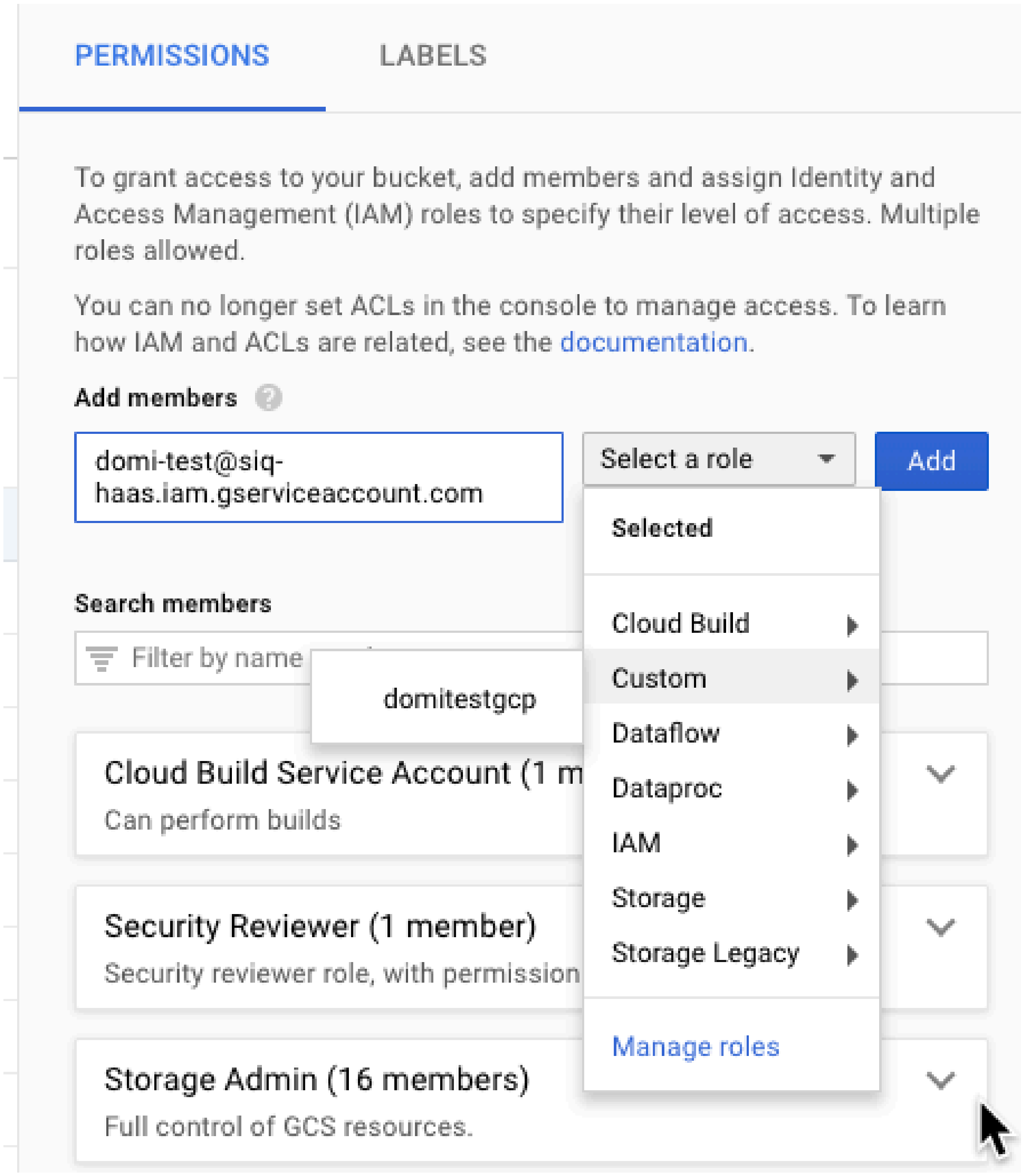
Modify GCS Bucket Permissions

You or your GCP admin must set the bucket permissions so that your service account has access to the bucket that you want to access from the cluster. The IAM role created in the previous step, is the minimum role required to access the cluster. Example steps are described below.

1. In the Google Cloud Platform web console, navigate to Storage > Browser.
2. Find the bucket for which you want to edit permissions.
3. Click the ⋮ and select Edit bucket permissions:
4. In the Permissions tab set the bucket-level permissions:

    - Click on Add members and enter the service account created earlier.
    - Under Roles, select the IAM role created in the previous step. The role should be available under Custom.

5. When done, click Add.


PERMISSIONS
LABELS
To grant access to your bucket, add members and assign Identity and Access Management (IAM) roles to specify their level of access. Multiple roles allowed.
You can no longer set ACLs in the console to manage access. To learn how IAM and ACLs are related, see the documentation.
Add members
domi-test@siq-haas.iam.gserviceaccount.com
Select a role
Add
Selected
Cloud Build
Custom
Dataflow
Dataproc
IAM
Storage
Storage Legacy
Manage roles
Search members
Filter by name
domitestgcp
Cloud Build Service Account (1 m
Can perform builds
Security Reviewer (1 member)
Security reviewer role, with permission
Storage Admin (16 members)
Full control of GCS resources.

After performing these steps, the bucket-level permissions will be updated.

Configure Access to GCS from Your Cluster

After obtaining the service account key, perform these steps on your cluster. The steps below assume that your service account key is called google-access-key.json. If you choose a different name, make sure to update the commands accordingly.

1. In Cloudera Manager UI, set the following three properties under hdfs core-site.xml. Navigate to Clusters > HDFS > Configuration > Advanced > Add property under Cluster-wide Advanced Configuration Snippet (Safety Valve) for core-site.xml.

   - Set the following properties: fs.gs.auth.service.account.email=<VALUE1> fs.gs.auth.service.accou nt.private.key.id=<VALUE2> fs.gs.auth.service.account.private.key=<VALUE3> You can obtain the values for these parameters as follows:

Parameter	Value
fs.gs.auth.service.account.email	The client_email field extracted from the credential's JSON file.
fs.gs.auth.service.account.private.key.id	The private_key_id field extracted from the credential's JSON file.
fs.gs.auth.service.account.private.key	The private_keyfield extracted from the credential's JSON file.

     The JSON key file downloaded in the previous step is in plain text. Open the file in your favorite text editor to extract the relevant values needed in the above configuration.

     Note: For enhanced security, these values can also be configured using Hadoop CredentialProvider.

   - Configure the following properties if they're not already set by Cloudera Manager fs.gs.working.dir=/ fs .gs.path.encoding=uri-path

     Note: Setting fs.gs.working.dir configures the initial working directory of a GHFS instance. This should always be set to "/". Setting fs.gs.path.encoding sets the path encoding to be used, and allows for spaces in the filename. This should always be set to "uri-path".
2. Save the configuration change and restart affected services. Additionally - depending on what services you are using - you must restart other services that access cloud storage such as Spark Thrift Server, HiveServer2, and Hive Metastore; These will not be listed as affected by Cloudera Manager, but require a restart to pick up the configuration changes.
3. Test access to the Google Cloud Storage bucket by running a few commands from any cluster node. For example, you can use the command listed below (replace “mytestbucket” with the name of your bucket): hadoop fs -ls gs:// mytestbucket/

After performing these steps, you should be able to start working with the Google Cloud Storage bucket(s).

Manifest committer for ABFS and GCS

The Intermediate Manifest committer is a high performance committer for Spark work that provides performance on ABFS for real world queries, and performance and correctness on GCS. It also works with other filesystems, including HDFS. However, the design is optimized for object stores where listing operatons are slow and expensive.

The v1/v2 file committers

The only committer of work from Spark to Azure ADLS Gen 2 "abfs://" storage that is safe to use is the "v1 file committer."

This is "correct" in that if a task attempt fails, its output is guaranteed not to be included in the final out. The "v2" commit algorithm cannot meet that guarantee, which is why it is no longer the default.

But the v1 file committer is slow, especially on jobs where deep directory trees of output are used. This is due to lack of any instrumentation in the FileOutputCommitter. Stack traces of running jobs generally show rename(), though list operations do surface too.

On GCS, neither the v1 nor v2 algorithm are safe because the Google filesystem doesn't have the atomic directory rename which the v1 algorithm requires.

A further issue is that both Azure and GCS storage may encounter scale issues with deleting directories with many descendants. This can trigger timeouts because the FileOutputCommitter assumes that cleaning up after the job is a fast call to delete("_temporary", true).

The manifest committer

The Manifest Committer is a higher performance committer for ABFS and GCS storage for jobs that create files across deep directory trees through many tasks. It will also work on hdfs:// and file:// URLs, but it is optimized to address listing and renaming performance and throttling issues in cloud storage.

It will not work correctly with S3, because it relies on an atomic rename-no-overwrite operation to commit the manifest file. It will also have the performance problems of copying rather than moving all the generated data.

Although it will work with MapReduce, there is no handling of multiple job attempts with recovery from previous failed attempts.

This committer uses the extension point which came in for the S3A committers. Users can declare a new committer factory for abfs:// and gcs:// URLs. A suitably configured Spark deployment will pick up the new committer.

Using the manifest committer

Define a factory for the ABFS or GCS schema, then bind to the manifest committer in Spark.

Using the committer

Define a factory for the ABFS schema in mapreduce.outputcommitter.factory.scheme.abfs or mapreduce.outputcomm itter.factory.scheme.gs for GCS.

Some matching spark configuration changes, especially for parquet binding, will be required. These can be done in core-site.xml, if it is not defined in the mapred-default.xml JAR.

```
<property>
  <name>mapreduce.outputcommitter.factory.scheme.abfs</name>
  <value>org.apache.hadoop.fs.azurebfs.commit.AzureManifestCommitterFactory
</value>
</property>
<property>
  <name>mapreduce.outputcommitter.factory.scheme.gs</name>
  <value>org.apache.hadoop.mapreduce.lib.output.committer.manifest.Manife
stCommitterFactory</value>
</property>
```

Binding to the manifest committer in Spark

In Apache Spark, set the configuration either with command line options (after the '--conf') or by using the spark-de faults.conf file. The following is an example of using spark-defaults.conf, and also including the configuration for Parquet with a subclass of the parquet committer which uses the factory mechansim internally.

```
spark.hadoop.mapreduce.outputcommitter.factory.scheme.abfs org.apache.hadoop
.fs.azurebfs.commit.AzureManifestCommitterFactory
spark.hadoop.mapreduce.outputcommitter.factory.scheme.gs org.apache.hadoop
.mapreduce.lib.output.committer.manifest.ManifestCommitterFactory
spark.sql.parquet.output.committer.class org.apache.spark.internal.io.clou
d.BindingParquetOutputCommitter
spark.sql.sources.commitProtocolClass org.apache.spark.internal.io.cloud.Pat
hOutputCommitProtocol
```

Using the Cloudstore committerinfo command to probe committer bindings

The hadoop committer settings can be validated in a recent build of cloudstore and its committerinfo command. This command instantiates a committer for that path through the same factory mechanism as MR and spark jobs use, then prints its toString value.

```
hadoop jar cloudstore-1.0.jar committerinfo abfs://testing@ukwest.dfs.core.w
indows.net/
2021-09-16 19:42:59,731 [main] INFO  commands.CommitterInfo (StoreDurationI
nfo.java:<init>(53)) - Starting: Create committer
Committer factory for path abfs://testing@ukwest.dfs.core.windows.net/ is
 org.apache.hadoop.mapreduce.lib.output.committer.manifest.ManifestCommitter
Factory@3315d2d7
  (classname org.apache.hadoop.mapreduce.lib.output.committer.manifest.Mani
festCommitterFactory)
2021-09-16 19:43:00,897 [main] INFO  manifest.ManifestCommitter (ManifestC
ommitter.java:<init>(144)) - Created ManifestCommitter with
   JobID job__0000, Task Attempt attempt__0000_r_000000_1 and destination ab
fs://testing@ukwest.dfs.core.windows.net/
Created committer of class org.apache.hadoop.mapreduce.lib.output.committe
r.manifest.ManifestCommitter:
 ManifestCommitter{ManifestCommitterConfig{destinationDir=abfs://testing@u
kwest.dfs.core.windows.net/,
   role='task committer',
   taskAttemptDir=abfs://testing@ukwest.dfs.core.windows.net/_temporary/m
anifest_job__0000/0/_temporary/attempt__0000_r_000000_1,
   createJobMarker=true,
   jobUniqueId='job__0000',
   jobUniqueIdSource='JobID',
   jobAttemptNumber=0,
   jobAttemptId='job__0000_0',
   taskId='task__0000_r_000000',
   taskAttemptId='attempt__0000_r_000000_1'},
   iostatistics=counters=();

gauges=();

minimums=();
maximums=();

means=();
}
```

Verifying that the committer was used

The new committer will write a JSON summary of the operation, including statistics, in the _SUCCESS file.

If this file exists and is zero bytes long: the classic FileOutputCommitter was used.

If this file exists and is greater than zero bytes long, either the manifest committer was used, or in the case of S3A filesystems, one of the S3A committers. They all use the same JSON format.

Configuration options

Option	Meaning	Default value
mapreduce.manifest.committer.delete.target.files	Delete target files?	false
mapreduce.manifest.committer.io.threads	Thread count for parallel operations	64
mapreduce.manifest.committer.summary.report.directory	Directory to save reports	""

Option	Meaning	Default value
mapreduce.manifest.committer.cleanup.parallel.delete	Delete temporary directories in parallel	true
mapreduce.fileoutputcommitter.cleanup.skipped	Skip cleanup of _temporary directory	false
mapreduce.fileoutputcommitter.cleanup-failures.ignored	Ignore errors during cleanup	false
mapreduce.fileoutputcommitter.marksuccessfuljobs	Create a _SUCCESS marker file on successful completion (and delete any existing one in job setup)	true

Scaling jobs mapreduce.manifest.committer.io.threads

The core reason that the manifest committer is faster than the classic FileOutputCommitter is that it tries to parallelize as much file IO as it can during job commit, specifically:

- Task manifest loading
- Deletion of files where directories will be created
- Directory creation
- File-by-file renaming
- Deletion of task attempt directories in job cleanup

These operations are all performed in the same thread pool, whose size is set in the option mapreduce.manifest.committer.io.threads.

Larger values may be used.

XML:

```
<property>
  <name>mapreduce.manifest.committer.io.threads</name>
  <value>200</value>
</property>
```

spark-defaults.conf:

```
spark.hadoop.mapreduce.manifest.committer.io.threads 200
```

A larger value than that of the number of cores allocated to the MapReduce AM or Spark Driver does not directly overload the CPUs, as the threads are normally waiting for (slow) IO against the object store/filesystem to complete.

Caveats:

- In Spark, multiple jobs may be committed in the same process, each of which will create their own thread pool during job commit or cleanup.
- Azure rate throttling may be triggered if too many IO requests are made against the store. The rate throttling option mapreduce.manifest.committer.io.rate can help avoid this.

Optional: deleting target files in Job Commit

The classic FileOutputCommitter deletes files at the destination paths before renaming the job's files into place.

This is optional in the manifest committers, set in the option mapreduce.manifest.committer.delete.target.files with a default value of false. This increases performance and is safe to use when all files created by a job have unique filenames.

Apache Spark does generate unique filenames for ORC and Parquet since SPARK-8406 Adding UUID to output file name to avoid accidental overwriting.

Avoiding checks for/deleting target files saves one delete call per file being committed, so can save a significant amount of store IO.

When appending to existing tables, using formats other than ORC and Parquet, unless confident that unique identifiers are added to each filename, enable deletion of the target files.

```
spark.hadoop.mapreduce.manifest.committer.delete.target.files true
```

Note: The committer will skip deletion operations when it created the directory into which a file is to be renamed. This makes it slightly more efficient, at least if jobs appending data are creating and writing into new partitions.

Note: The committer still requires tasks within a single job to create unique files. This is foundational for any job to generate correct data.

Spark Dynamic Partition overwriting

Spark has a feature called "Dynamic Partition Overwrites" that can be initiated in SQL:

```

INSERT OVERWRITE TABLE ...
```

Or through DataSet writes where the mode is overwrite and the partitioning matches that of the existing table:

```

sparkConf.set("spark.sql.sources.partitionOverwriteMode", "dynamic")
// followed by an overwrite of a Dataset into an existing partitioned table.
eventData2
  .write
  .mode("overwrite")
  .partitionBy("year", "month")
  .format("parquet")
  .save(existingDir)
```

This Spark feature:

- Directs the job to write its new data to a temporary directory
- After job commit completes, scans the output to identify the leaf directories "partitions" into which data was written
- Deletes the content of those directories in the destination table
- Renames the new files into the partitions.

This is all done in Spark, which takes over the tasks of scanning the intermediate output tree, deleting partitions and of renaming the new files.

This feature also adds the ability for a job to write data entirely outside the destination table, which is done by writing new files into the working directory, and then Spark moving them to the final destination in job commit.

The manifest committer is compatible with dynamic partition overwrites on Azure and Google Cloud Storage as together they meet the core requirements of the extension:

- The working directory returned in getWorkPath() is in the same filesystem as the final output.
- rename() is an O(1) operation which is safe and fast to use when committing a job.

None of the S3A committers support this. The first condition is not met by the staging committers, while the second condition is not met by S3 itself.

To use the manifest committer with dynamic partition overwrites, the Spark version must contain SPARK-40034 PathOutputCommitters to work with dynamic partition overwrite.

Be aware that the rename phase of the operation will be slow if many files are renamed--this is done sequentially. Parallel renaming would speed this up, but could trigger the ABFS overload problems the manifest committer is designed to both minimize the risk of and support recovery from.

The Spark side of the commit operation will be listing/treewalking the temporary output directory (some overhead), followed by the file promotion, done with a classic filesystem rename() call. There will be no explicit rate limiting here.

This means that _dynamic partitioning should not be used on Azure Storage for SQL queries/Spark DataSet operations where many thousands of files are created. The fact that these will suffer from performance problems before throttling scale issues surface, should be considered a warning.

Job summaries in _SUCCESS files

You can view and collect job summaries in the _SUCCESS files.

_SUCCESS files

The original Hadoop committer creates a zero byte _SUCCESS file in the root of the output directory unless disabled.

The manifest committer writes a JSON summary which includes:

- The name of the committer.
- Diagnostics information.
- A list of some of the files created (for testing; a full list is excluded as it can get big).
- IO Statistics.

If, after running a query, this _SUCCESS file is zero bytes long, the manifest committer has not been used.

If it is not empty, then it can be examined.

Viewing _SUCCESS file files through the ManifestPrinter tool

The summary files are JSON, and can be viewed in any text editor.

For a more succinct summary, including better display of statistics, use the ManifestPrinter tool.

```
hadoop org.apache.hadoop.mapreduce.lib.output.committer.manifest.files.Manif
estPrinter <path>
```

This works for the files saved at the base of an output directory, and any reports saved to a report directory.

Collecting Job Summaries

The committer can be configured to save the _SUCCESS summary files to a report directory, irrespective of whether the job succeed or failed, by setting a fileystem path in the option mapreduce.manifest.committer.summary.report.dire ctory.

The path does not have to be on the same store/filesystem as the destination of work. For example, a local fileystem could be used.

XML:

```

<property>
  <name>mapreduce.manifest.committer.summary.report.directory</name>
  <value>file:///tmp/reports</value>
</property>
```

spark-defaults.conf:

```
spark.hadoop.mapreduce.manifest.committer.summary.report.directory file:///t
mp/reports
```

This allows for the statistics of jobs to be collected irrespective of their outcome, whether or not saving the _SUC CESS marker is enabled, and without problems caused by a chain of queries overwriting the markers.

Job cleanup

Job cleanup is designed to address a number of issues which may surface in cloud storage:

- Slow performance for deletion of directories.
- Timeout when deleting very deep and wide directory trees.
- General resilience to cleanup issues escalating to job failures.

Option	Meaning	Default Value
mapreduce.fileoutputcommitter.cleanup.skipped	Skip cleanup of _temporary directory	false
mapreduce.fileoutputcommitter.cleanup-failures.ignored	Ignore errors during cleanup	false
mapreduce.manifest.committer.cleanup.parallel.delete	Delete task attempt directories in parallel	true

The algorithm is:

```
if `mapreduce.fileoutputcommitter.cleanup.skipped`:
  return
if `mapreduce.manifest.committer.cleanup.parallel.delete`:
  attempt parallel delete of task directories; catch any exception
if not `mapreduce.fileoutputcommitter.cleanup.skipped`:
  delete(`_temporary`); catch any exception
if caught-exception and not `mapreduce.fileoutputcommitter.cleanup-failures.
ignored`:
  throw caught-exception
```

The goal is to perform a fast/scalable delete and throw a meaningful exception if that didn't work.

When working with ABFS and GCS, these settings should normally be left alone. If somehow errors surface during cleanup, enabling the option to ignore failures will ensure the job still completes. Disabling cleanup even avoids the overhead of cleanup, but requires a workflow or manual operation to clean up all _temporary directories on a regular basis.

Working with Google Cloud Storage

The manifest committer is compatible with and tested against Google Cloud Storage through the gcs-connector library from Google, which provides a Hadoop filesystem client for the schema gs.

Google Cloud Storage has the semantics needed for the commit protocol to work safely.

The Spark settings to switch to this committer are:

```
 spark.hadoop.mapreduce.outputcommitter.factory.scheme.gs org.apache.had
 oop.mapreduce.lib.output.committer.manifest.ManifestCommitterFactory
 spark.sql.parquet.output.committer.class org.apache.spark.internal.io.c
 loud.BindingParquetOutputCommitter
 spark.sql.sources.commitProtocolClass org.apache.spark.internal.io.cloud.
 PathOutputCommitProtocol
```

```
spark.hadoop.mapreduce.manifest.committer.summary.report.directory  (optio
nal: URI of a directory for job summaries)
```

The store's directory delete operations are O(files) so the value of mapreduce.manifest.committer.cleanup.parallel.de lete should be left at the default of true.

For MapReduce, declare the binding in core-site.xmlor mapred-site.xml:

```
<property>
  <name>mapreduce.outputcommitter.factory.scheme.gcs</name>
  <value>org.apache.hadoop.mapreduce.lib.output.committer.manifest.Manife
stCommitterFactory</value>
</property>
```

Advanced topics

Advanced topics and configuration options for the manifest committer.

Advanced configuration

There are some advanced options which are intended for development and testing, rather than production use.

Option	Meaning	Default Value
mapreduce.manifest.committer.store.operations.classname	Classname for Manifest Store Operations	""
mapreduce.manifest.committer.validate.output	Perform output validation?	false

Validating output

The option mapreduce.manifest.committer.validate.output triggers a check of every renamed file to verify it has the expected length.

This adds the overhead of a HEAD request per file, and so is recommended for testing only.

There is no verification of the actual contents.

Controlling storage integration

The manifest committer interacts with filesystems through implementations of the interface ManifestStoreOperations. It is possible to provide custom implementations for store-specific features. There is one of these for ABFS; when the abfs-specific committer factory is used this is automatically set.

It can be explicitly set:

```
<property>
  <name>mapreduce.manifest.committer.store.operations.classname</name>
  <value>org.apache.hadoop.fs.azurebfs.commit.AbfsManifestStoreOperations</
value>
</property>
```

The default implementation may also be configured:

```
<property>
  <name>mapreduce.manifest.committer.store.operations.classname</name>
  <value>org.apache.hadoop.mapreduce.lib.output.committer.manifest.impl.Ma
nifestStoreOperationsThroughFileSystem</value>
</property>
```

There is no need to alter these values, except when writing new implementations for other stores, something which is only needed if the store provides extra integration support for the committer.

Additional Configuration Options for GCS

You may optionally set the following properties related to Google Cloud Storage.

Table 3: Optional Properties Related to Google Cloud Storage

Property	Description
fs.gs.project.id	Allows you to enter Google Cloud's Project ID with access to configured GCS buckets. By default, this option is unset.
fs.gs.block.size	This does not have any effect on storage but allows you to change the way split sizes are computed. The value should be in bytes. We recommend that you set this to the HDFS block size on the cluster, or 134217728 for a block size of 128MB.

Working with Azure Cloud Storage

The Cloudera platform allows you to configure access from a cluster to Azure in order to access cloud data.

Introduction to Azure Storage and the ABFS Connector

The Hadoop-Azure module provides support for Azure Data Lake Storage Gen2 storage layer through the `abfs` connector.

#Azure Data Lake Storage (ADLS) Gen2 combines the features of `Azure Blob storage` and `Azure Data Lake Storage Gen1.` In addition to the existing features of both the services, an important part of Azure Data Lake Storage Gen2 is the addition of hierarchical namespace to Blob storage. The hierarchical namespace feature processes operations such as moving, renaming, and deletings directories by updating a single entry (the parent directory). This ensures a significant improvement in performance of query engines writing data to the store, including MapReduce, Spark, Hive, and DistCp. You can also set permissions on a directory instead of per file basis.

Note: Hierarchical Namespace feature is available only if the container is created with the namespace support. You cannot enable Hierarchical Namespaces on an existing storage account.

Azure Blob Storage

The Azure Storage data model presents three core concepts:

- Storage Account: All access is done through a storage account.
- Container: A container is a grouping of multiple blobs. A storage account may have multiple containers. In Hadoop, an entire file system hierarchy is stored in a single container.
- Blob: A file of any type and size stored with the existing Windows Azure Storage Blob (wasb) connector

Features of the ABFS Connector

The ABFS connector can be used as a replacement for HDFS on Hadoop clusters deployed in Azure infrastructure. Some of the features of the ABFS connector are as follows:

- Supports reading and writing data stored in an Azure Blob Storage account
- Helps to perform hadoop operations on Azure datastores
- Provides a consistent view of the storage across all clients
- Enables using a simple abfs:// url to access containers and directories

- Presents a hierarchical file system view by implementing the standard Hadoop File System interface
- Supports configuration of multiple Azure Blob Storage accounts
- Acts as a source or destination of data in Hadoop MapReduce, Apache Hive, Apache Spark

Feature Comparisons

ADLS Gen 2 is where all future development of Azure Big Data storage is taking place, in the cloud server as well as the client connector. In contrast, ADLS Gen 1 must be considered a `maintenance only` data store, which is not being rolled out across more Azure regions.

Feature	Azure Storage	ADLS Gen 1	ADLS Gen 2
Broadly supported in applications	Yes	No	Yes in non-hierarchical; growing in hierarchical
Scales to many petabytes	No	Yes	Yes
Available in all Azure region	Yes	No	Yes
Interoperability	ADLS Gen 2 in non-hierarchical	none	Azure Storage
Directories with permissions	No	Yes	Yes

Differences between ADLS Gen 1 and ADLS Gen 2

ADLS Gen 2 should be considered as a reimplementation of the ADLS Gen 1 features, but integrated with the original Azure Storage.

Features of ADLS Gen 2

- Supports the Hadoop FileSystem API, with directories, file and directory permissions, and other key features.
- Reads and writes data stores in the original Azure Storage (which has previously used the wasb:// URL)
- No integration with ADLS Gen1. The adls:// connector must be used there.
- Available in all Azure regions.
- If an ADLS Gen 2 account is created "without hierarchical namespaces" then the wasb:// connector can read/write data stored in ADLS Gen 2.
- Capable of storing many Petabytes of data.

Setting up and configuring the ABFS connector

You must have an Azure storage account with hierarchical namespace enabled and Azure container prior to configuring the ABFS connector.

About this task

You can use the following steps to create a new storage account, create a new container, and configure the ABFS connector:

Procedure

1. Create Azure storage account

   Every storage account must belong to an Azure resource group. A resource group is a logical container for grouping your Azure services. When you create a storage account, you have the option to either create a new resource group, or use an existing resource group.

   You can create an Azure storage account using Azure CLI or Azure portal. When creating a Storage Account, you must ensure to enable namespace support by selecting the `Hierarchical Namespace` option. For information on creating an Azure storage account, see Create a storage account.

2. Create Azure Container

   You must create a container to organize the blobs. One Azure storage account can have multiple containers, each with the container name as the user information field of the URI used to reference it.

   For example, the container `container1` in the storage account will have the URL abfs://container1@<storage_account>.dfs.core.windows.net/ .

   For information on creating a container in Azure, see Create a container.

3. Configure ABFS

   You can configure access credentials to authorise access to Azure containers. For information on configuring ABFS, see Configuring ABFS.

   Note: Delete Azure storage account - Deleting a storage account deletes the entire account, including all data in the account, and cannot be undone. For information about deleting an Azure storage account, see Delete a storage account.

Configuring the ABFS Connector

You can configure access credentials to authorise access to Azure containers in multiple ways including IDBroker, Shared Key, Managed Instance, and Shared Access Signature.

Authenticating with ADLS Gen2

Authentication for ABFS is granted by Azure Active Directory. The authentication mechanism is set using the `fs.azure.account.auth.type` (or the account specific variant) property in the core-site.xml file.

Configuring Access to Azure in Cloudera on cloud

IDBroker is a REST API built as part of Apache Knox's authentication services. It allows an authenticated user to exchange a set of credentials or a token for cloud vendor access tokens. IDBroker manages mapping LDAP users to FreeIPA cloud identities for data access. It performs identity mapping for access to object stores.

For information on how IDBroker works in Cloudera, see ADLS Gen2 and managed identities in the *Cloudera Management Console* documentation.

ADLS Proxy Setup

The ADLS connector uses the JVM proxy settings to control its proxy setup.

The Oracle Java documentation covers the options to set.

As the ADLS connector uses HTTPS, the https.proxy options are those which must be configured. See DistCp and Proxy Settings for the specifics of how to set these proxy settings in DistCp (or indeed, any MapReduce job).

Manifest committer for ABFS and GCS

The Intermediate Manifest committer is a high performance committer for Spark work that provides performance on ABFS for real world queries, and performance and correctness on GCS. It also works with other filesystems, including HDFS. However, the design is optimized for object stores where listing operatons are slow and expensive.

The v1/v2 file committers

The only committer of work from Spark to Azure ADLS Gen 2 "abfs://" storage that is safe to use is the "v1 file committer."

This is "correct" in that if a task attempt fails, its output is guaranteed not to be included in the final out. The "v2" commit algorithm cannot meet that guarantee, which is why it is no longer the default.

But the v1 file committer is slow, especially on jobs where deep directory trees of output are used. This is due to lack of any instrumentation in the FileOutputCommitter. Stack traces of running jobs generally show rename(), though list operations do surface too.

On GCS, neither the v1 nor v2 algorithm are safe because the Google filesystem doesn't have the atomic directory rename which the v1 algorithm requires.

A further issue is that both Azure and GCS storage may encounter scale issues with deleting directories with many descendants. This can trigger timeouts because the FileOutputCommitter assumes that cleaning up after the job is a fast call to delete("_temporary", true).

The manifest committer

The Manifest Committer is a higher performance committer for ABFS and GCS storage for jobs that create files across deep directory trees through many tasks. It will also work on hdfs:// and file:// URLs, but it is optimized to address listing and renaming performance and throttling issues in cloud storage.

It will not work correctly with S3, because it relies on an atomic rename-no-overwrite operation to commit the manifest file. It will also have the performance problems of copying rather than moving all the generated data.

Although it will work with MapReduce, there is no handling of multiple job attempts with recovery from previous failed attempts.

This committer uses the extension point which came in for the S3A committers. Users can declare a new committer factory for abfs:// and gcs:// URLs. A suitably configured Spark deployment will pick up the new committer.

Using the manifest committer

Define a factory for the ABFS or GCS schema, then bind to the manifest committer in Spark.

Using the committer

Define a factory for the ABFS schema in mapreduce.outputcommitter.factory.scheme.abfs or mapreduce.outputcomm itter.factory.scheme.gs for GCS.

Some matching spark configuration changes, especially for parquet binding, will be required. These can be done in core-site.xml, if it is not defined in the mapred-default.xml JAR.

```

<property>
  <name>mapreduce.outputcommitter.factory.scheme.abfs</name>
  <value>org.apache.hadoop.fs.azurebfs.commit.AzureManifestCommitterFactory
</value>
</property>
<property>
  <name>mapreduce.outputcommitter.factory.scheme.gs</name>
  <value>org.apache.hadoop.mapreduce.lib.output.committer.manifest.Manife
stCommitterFactory</value>
</property>
```

Binding to the manifest committer in Spark

In Apache Spark, set the configuration either with command line options (after the '--conf') or by using the spark-de faults.conf file. The following is an example of using spark-defaults.conf, and also including the configuration for Parquet with a subclass of the parquet committer which uses the factory mechansim internally.

```
spark.hadoop.mapreduce.outputcommitter.factory.scheme.abfs org.apache.hadoop
.fs.azurebfs.commit.AzureManifestCommitterFactory
spark.hadoop.mapreduce.outputcommitter.factory.scheme.gs org.apache.hadoop
.mapreduce.lib.output.committer.manifest.ManifestCommitterFactory
spark.sql.parquet.output.committer.class org.apache.spark.internal.io.clou
d.BindingParquetOutputCommitter
```

```
spark.sql.sources.commitProtocolClass org.apache.spark.internal.io.cloud.Pat
hOutputCommitProtocol
```

Using the Cloudstore committerinfo command to probe committer bindings

The hadoop committer settings can be validated in a recent build of cloudstore and its committerinfo command. This command instantiates a committer for that path through the same factory mechanism as MR and spark jobs use, then prints its toString value.

```
hadoop jar cloudstore-1.0.jar committerinfo abfs://testing@ukwest.dfs.core.w
indows.net/
2021-09-16 19:42:59,731 [main] INFO  commands.CommitterInfo (StoreDurationI
nfo.java:<init>(53)) - Starting: Create committer
Committer factory for path abfs://testing@ukwest.dfs.core.windows.net/ is
 org.apache.hadoop.mapreduce.lib.output.committer.manifest.ManifestCommitter
Factory@3315d2d7
  (classname org.apache.hadoop.mapreduce.lib.output.committer.manifest.Mani
festCommitterFactory)
2021-09-16 19:43:00,897 [main] INFO  manifest.ManifestCommitter (ManifestC
ommitter.java:<init>(144)) - Created ManifestCommitter with
   JobID job__0000, Task Attempt attempt__0000_r_000000_1 and destination ab
fs://testing@ukwest.dfs.core.windows.net/
Created committer of class org.apache.hadoop.mapreduce.lib.output.committe
r.manifest.ManifestCommitter:
 ManifestCommitter{ManifestCommitterConfig{destinationDir=abfs://testing@u
kwest.dfs.core.windows.net/,
   role='task committer',
   taskAttemptDir=abfs://testing@ukwest.dfs.core.windows.net/_temporary/m
anifest_job__0000/0/_temporary/attempt__0000_r_000000_1,
   createJobMarker=true,
   jobUniqueId='job__0000',
   jobUniqueIdSource='JobID',
   jobAttemptNumber=0,
   jobAttemptId='job__0000_0',
   taskId='task__0000_r_000000',
   taskAttemptId='attempt__0000_r_000000_1'},
   iostatistics=counters=();

gauges=();

minimums=();
maximums=();

means=();
}
```

Verifying that the committer was used

The new committer will write a JSON summary of the operation, including statistics, in the _SUCCESS file.

If this file exists and is zero bytes long: the classic FileOutputCommitter was used.

If this file exists and is greater than zero bytes long, either the manifest committer was used, or in the case of S3A filesystems, one of the S3A committers. They all use the same JSON format.

Configuration options

Option	Meaning	Default value
mapreduce.manifest.committer.delete.target.files	Delete target files?	false

Option	Meaning	Default value
mapreduce.manifest.committer.io.threads	Thread count for parallel operations	64
mapreduce.manifest.committer.summary.report.directory	Directory to save reports	""
mapreduce.manifest.committer.cleanup.parallel.delete	Delete temporary directories in parallel	true
mapreduce.fileoutputcommitter.cleanup.skipped	Skip cleanup of _temporary directory	false
mapreduce.fileoutputcommitter.cleanup-failures.ignored	Ignore errors during cleanup	false
mapreduce.fileoutputcommitter.marksuccessfuljobs	Create a _SUCCESS marker file on successful completion (and delete any existing one in job setup)	true

Scaling jobs mapreduce.manifest.committer.io.threads

The core reason that the manifest committer is faster than the classic FileOutputCommitter is that it tries to parallelize as much file IO as it can during job commit, specifically:

- Task manifest loading
- Deletion of files where directories will be created
- Directory creation
- File-by-file renaming
- Deletion of task attempt directories in job cleanup

These operations are all performed in the same thread pool, whose size is set in the option mapreduce.manifest.committer.io.threads.

Larger values may be used.

XML:

```
<property>
  <name>mapreduce.manifest.committer.io.threads</name>
  <value>200</value>
</property>
```

spark-defaults.conf:

```
spark.hadoop.mapreduce.manifest.committer.io.threads 200
```

A larger value than that of the number of cores allocated to the MapReduce AM or Spark Driver does not directly overload the CPUs, as the threads are normally waiting for (slow) IO against the object store/filesystem to complete.

Caveats:

- In Spark, multiple jobs may be committed in the same process, each of which will create their own thread pool during job commit or cleanup.
- Azure rate throttling may be triggered if too many IO requests are made against the store. The rate throttling option mapreduce.manifest.committer.io.rate can help avoid this.

Optional: deleting target files in Job Commit

The classic FileOutputCommitter deletes files at the destination paths before renaming the job's files into place.

This is optional in the manifest committers, set in the option mapreduce.manifest.committer.delete.target.files with a default value of false. This increases performance and is safe to use when all files created by a job have unique filenames.

Apache Spark does generate unique filenames for ORC and Parquet since SPARK-8406 Adding UUID to output file name to avoid accidental overwriting.

Avoiding checks for/deleting target files saves one delete call per file being committed, so can save a significant amount of store IO.

When appending to existing tables, using formats other than ORC and Parquet, unless confident that unique identifiers are added to each filename, enable deletion of the target files.

```
spark.hadoop.mapreduce.manifest.committer.delete.target.files true
```

Note: The committer will skip deletion operations when it created the directory into which a file is to be renamed. This makes it slightly more efficient, at least if jobs appending data are creating and writing into new partitions.

Note: The committer still requires tasks within a single job to create unique files. This is foundational for any job to generate correct data.

Spark Dynamic Partition overwriting

Spark has a feature called "Dynamic Partition Overwrites" that can be initiated in SQL:

```
INSERT OVERWRITE TABLE ...
```

Or through DataSet writes where the mode is overwrite and the partitioning matches that of the existing table:

```
sparkConf.set("spark.sql.sources.partitionOverwriteMode", "dynamic")
// followed by an overwrite of a Dataset into an existing partitioned table.
eventData2
  .write
  .mode("overwrite")
  .partitionBy("year", "month")
  .format("parquet")
  .save(existingDir)
```

This Spark feature:

- Directs the job to write its new data to a temporary directory
- After job commit completes, scans the output to identify the leaf directories "partitions" into which data was written
- Deletes the content of those directories in the destination table
- Renames the new files into the partitions.

This is all done in Spark, which takes over the tasks of scanning the intermediate output tree, deleting partitions and of renaming the new files.

This feature also adds the ability for a job to write data entirely outside the destination table, which is done by writing new files into the working directory, and then Spark moving them to the final destination in job commit.

The manifest committer is compatible with dynamic partition overwrites on Azure and Google Cloud Storage as together they meet the core requirements of the extension:

- The working directory returned in getWorkPath() is in the same filesystem as the final output.
- rename() is an O(1) operation which is safe and fast to use when committing a job.

None of the S3A committers support this. The first condition is not met by the staging committers, while the second condition is not met by S3 itself.

To use the manifest committer with dynamic partition overwrites, the Spark version must contain SPARK-40034 PathOutputCommitters to work with dynamic partition overwrite.

Be aware that the rename phase of the operation will be slow if many files are renamed--this is done sequentially. Parallel renaming would speed this up, but could trigger the ABFS overload problems the manifest committer is designed to both minimize the risk of and support recovery from.

The Spark side of the commit operation will be listing/treewalking the temporary output directory (some overhead), followed by the file promotion, done with a classic filesystem rename() call. There will be no explicit rate limiting here.

This means that _dynamic partitioning should not be used on Azure Storage for SQL queries/Spark DataSet operations where many thousands of files are created. The fact that these will suffer from performance problems before throttling scale issues surface, should be considered a warning.

Job summaries in _SUCCESS files

You can view and collect job summaries in the _SUCCESS files.

_SUCCESS files

The original Hadoop committer creates a zero byte _SUCCESS file in the root of the output directory unless disabled.

The manifest committer writes a JSON summary which includes:

- The name of the committer.
- Diagnostics information.
- A list of some of the files created (for testing; a full list is excluded as it can get big).
- IO Statistics.

If, after running a query, this _SUCCESS file is zero bytes long, the manifest committer has not been used.

If it is not empty, then it can be examined.

Viewing _SUCCESS file files through the ManifestPrinter tool

The summary files are JSON, and can be viewed in any text editor.

For a more succinct summary, including better display of statistics, use the ManifestPrinter tool.

```
hadoop org.apache.hadoop.mapreduce.lib.output.committer.manifest.files.Manif
estPrinter <path>
```

This works for the files saved at the base of an output directory, and any reports saved to a report directory.

Collecting Job Summaries

The committer can be configured to save the _SUCCESS summary files to a report directory, irrespective of whether the job succeed or failed, by setting a fileystem path in the option mapreduce.manifest.committer.summary.report.dire ctory.

The path does not have to be on the same store/filesystem as the destination of work. For example, a local fileystem could be used.

XML:

```

<property>
  <name>mapreduce.manifest.committer.summary.report.directory</name>
  <value>file:///tmp/reports</value>
</property>
```

spark-defaults.conf:

```
spark.hadoop.mapreduce.manifest.committer.summary.report.directory file:///t
mp/reports
```

This allows for the statistics of jobs to be collected irrespective of their outcome, whether or not saving the _SUC CESS marker is enabled, and without problems caused by a chain of queries overwriting the markers.

Job cleanup

Job cleanup is designed to address a number of issues which may surface in cloud storage:

- Slow performance for deletion of directories.
- Timeout when deleting very deep and wide directory trees.
- General resilience to cleanup issues escalating to job failures.

Option	Meaning	Default Value
mapreduce.fileoutputcommitter.cleanup.skipped	Skip cleanup of _temporary directory	false
mapreduce.fileoutputcommitter.cleanup-failures.ignored	Ignore errors during cleanup	false
mapreduce.manifest.committer.cleanup.parallel.delete	Delete task attempt directories in parallel	true

The algorithm is:

```
if `mapreduce.fileoutputcommitter.cleanup.skipped`:
  return
if `mapreduce.manifest.committer.cleanup.parallel.delete`:
  attempt parallel delete of task directories; catch any exception
if not `mapreduce.fileoutputcommitter.cleanup.skipped`:
  delete(`_temporary`); catch any exception
if caught-exception and not `mapreduce.fileoutputcommitter.cleanup-failures.
ignored`:
  throw caught-exception
```

The goal is to perform a fast/scalable delete and throw a meaningful exception if that didn't work.

When working with ABFS and GCS, these settings should normally be left alone. If somehow errors surface during cleanup, enabling the option to ignore failures will ensure the job still completes. Disabling cleanup even avoids the overhead of cleanup, but requires a workflow or manual operation to clean up all _temporary directories on a regular basis.

Working with Azure ADLS Gen2 storage

Switch the factory for committers with ABFS destiations to the manifest committer factory.

To switch to the manifest committer, the factory for committers for destinations with abfs:// URLs must be switched to the manifest committer factory, either for the application or the entire cluster.

```
<property>
  <name>mapreduce.outputcommitter.factory.scheme.abfs</name>
  <value>org.apache.hadoop.fs.azurebfs.commit.AzureManifestCommitterFactory
</value>
</property>
```

This allows for ADLS Gen2 -specific performance and consistency logic to be used from within the committer. In particular:

- The Etag header can be collected in listings and used in the job commit phase.
- IO rename operations are rate limited
- Recovery is attempted when throttling triggers rename failures.

Warning: This committer is not compatible with older Azure storage services (WASB or ADLS Gen 1).

The core set of Azure-optimized options becomes:

```
<property>
  <name>mapreduce.outputcommitter.factory.scheme.abfs</name>
  <value>org.apache.hadoop.fs.azurebfs.commit.AzureManifestCommitterFactory
</value>
</property>

<property>
  <name>spark.hadoop.fs.azure.io.rate.limit</name>
  <value>10000</value>
</property>
```

Full set of ABFS options for Spark

```
spark.hadoop.mapreduce.outputcommitter.factory.scheme.abfs org.apache.hadoo
p.fs.azurebfs.commit.AzureManifestCommitterFactory
spark.hadoop.fs.azure.io.rate.limit 10000
spark.sql.parquet.output.committer.class org.apache.spark.internal.io.cloud
.BindingParquetOutputCommitter
spark.sql.sources.commitProtocolClass org.apache.spark.internal.io.cloud.
PathOutputCommitProtocol

spark.hadoop.mapreduce.manifest.committer.summary.report.directory  (optio
nal: URI of a directory for job summaries)
```

Experimental: ABFS Rename Rate Limiting

To avoid triggering store throttling and backoff delays, as well as other throttling-related failure conditions file renames during job commit are throttled through a "rate limiter" which limits the number of rename operations per second a single instance of the ABFS FileSystem client may issue.

Option	Meaning
fs.azure.io.rate.limit	Rate limit in operations/second for IO operations.

Set the option to 0 remove all rate limiting.

The default value of this is set to 10000, which is the default IO capacity for an ADLS storage account.

```
<property>
  <name>fs.azure.io.rate.limit</name>
  <value>10000</value>
  <description>maximum number of renames attempted per second</description>
</property>
```

This capacity is set at the level of the filesystem client, and so not shared across all processes within a single application, let alone other applications sharing the same storage account.

It will be shared with all jobs being committed by the same Spark driver, as these do share that filesystem connector.

If rate limiting is imposed, the statistic store_io_rate_limited will report the time to acquire permits for committing files.

If server-side throttling took place, signs of this can be seen in:

- The store service's logs and their throttling status codes (usually 503 or 500).
- The job statistic commit_file_rename_recovered. This statistic indicates that ADLS throttling manifested as failures in renames, failures which were recovered from in the comitter.

If these are seen or other applications running at the same time experience throttling/throttling-triggered problems, consider reducing the value of fs.azure.io.rate.limit, and/or requesting a higher IO capacity from Microsoft.

Important: if you do get extra capacity from Microsoft and you want to use it to speed up job commits, increase the value of fs.azure.io.rate.limit either across the cluster, or specifically for those jobs which you wish to allocate extra priority to.

This is still a work in progress; it may be expanded to support all IO operations performed by a single filesystem instance.

Advanced topics

Advanced topics and configuration options for the manifest committer.

Advanced configuration

There are some advanced options which are intended for development and testing, rather than production use.

Option	Meaning	Default Value
mapreduce.manifest.committer.store.operations.classname	Classname for Manifest Store Operations	""
mapreduce.manifest.committer.validate.output	Perform output validation?	false

Validating output

The option mapreduce.manifest.committer.validate.output triggers a check of every renamed file to verify it has the expected length.

This adds the overhead of a HEAD request per file, and so is recommended for testing only.

There is no verification of the actual contents.

Controlling storage integration

The manifest committer interacts with filesystems through implementations of the interface ManifestStoreOperations. It is possible to provide custom implementations for store-specific features. There is one of these for ABFS; when the abfs-specific committer factory is used this is automatically set.

It can be explicitly set:

```
<property>
  <name>mapreduce.manifest.committer.store.operations.classname</name>
  <value>org.apache.hadoop.fs.azurebfs.commit.AbfsManifestStoreOperations</
value>
</property>
```

The default implementation may also be configured:

```
<property>
  <name>mapreduce.manifest.committer.store.operations.classname</name>
  <value>org.apache.hadoop.mapreduce.lib.output.committer.manifest.impl.Ma
nifestStoreOperationsThroughFileSystem</value>
</property>
```

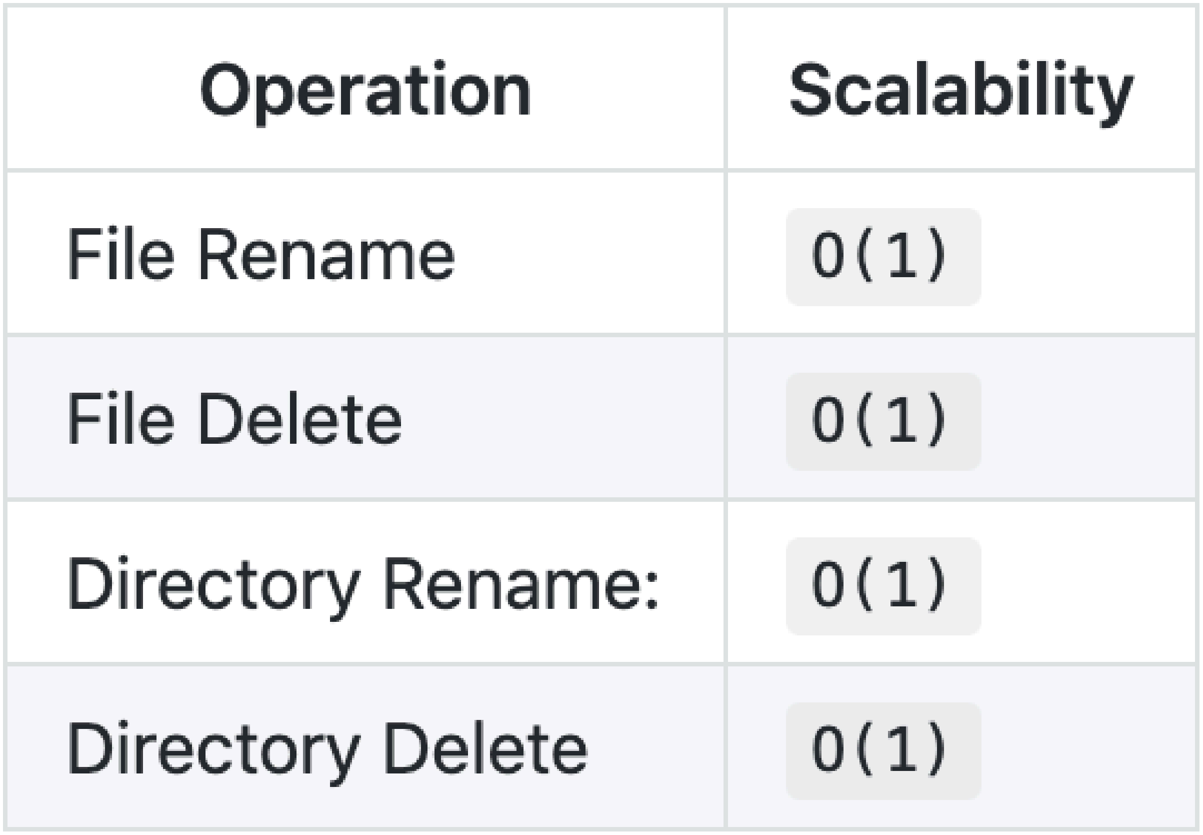
There is no need to alter these values, except when writing new implementations for other stores, something which is only needed if the store provides extra integration support for the committer.

Performance and Scalability

Like all Azure storage services, the Azure Datalake Gen 2 store offers a fully consistent view of the store, with a complete Create, Read, Update, and Delete operation consistency for data and metadata.

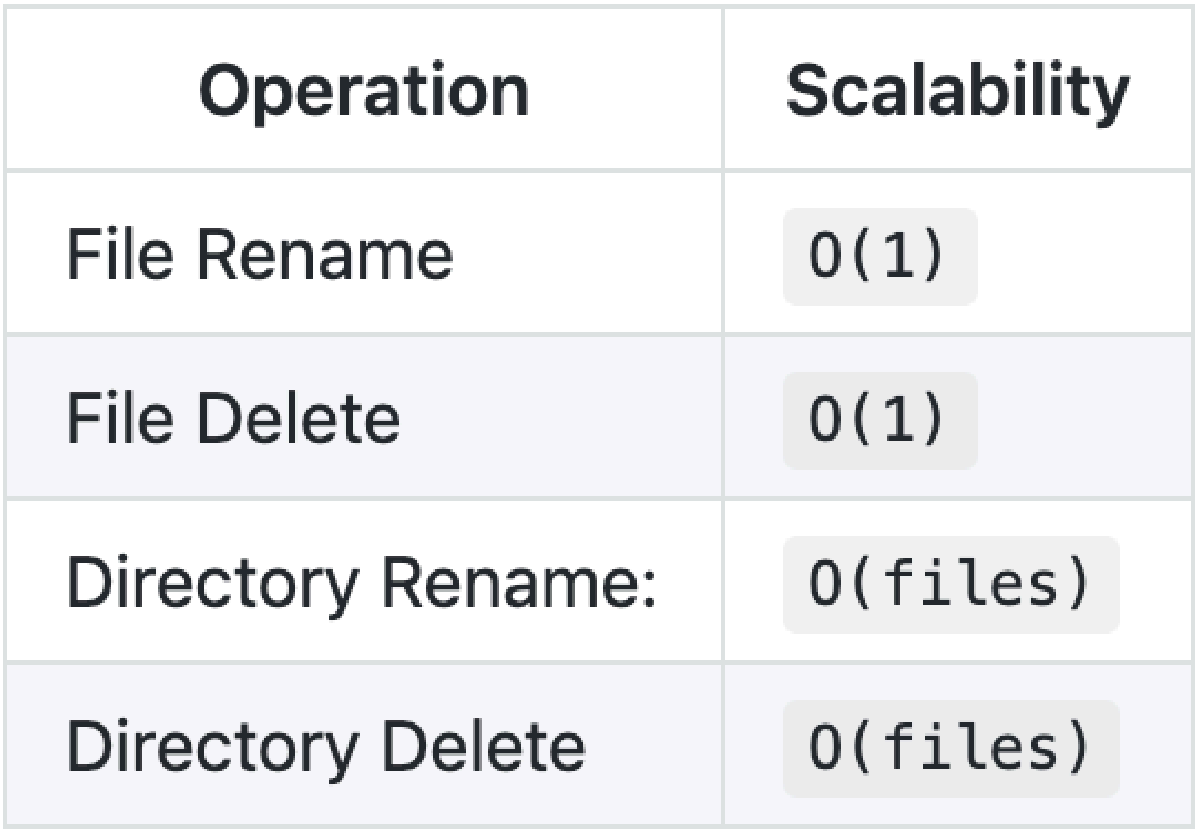
Hierarchical namespaces vs. non-namespaces

For containers with hierarchical namespaces, the scalability numbers, in Big-O Notation, are as follows:



Operation	Scalability
File Rename	`O(1)`
File Delete	`O(1)`
Directory Rename:	`O(1)`
Directory Delete	`O(1)`

For non-namespace stores, the scalability numbers are as follows:



Operation	Scalability
File Rename	`O(1)`
File Delete	`O(1)`
Directory Rename:	`O(files)`
Directory Delete	`O(files)`

Source: Performance_and_Scalability

Flush options

The Azure Blob File System and OutputStream flush options are enabled by default. These flush options have an impact on performance. Those applications which call flush() frequently, for example, with every write of a line, might show poor performance. Hence, you must disable the flush option if your application does not need it.

- Azure Blob File System Flush Options (`fs.azure.enable.flush`) - renders ABFS flush APIs - HFlush() and HSync(), to be no-op. Both the APIs ensure that data persists.
- OutputStream Flush Options (`fs.azure.disable.outputstream.flush`) - renders OutputStream Flush() API to be a no-op in AbfsOutputStream. Hflush() being the only documented API that can provide persistent data transfer, flush() also attempting to persist buffered data will lead to performance issues.

Using ABFS using CLI

After you configure access in the core-site.xml file, you can access your cluster using the CLI. You can run Hadoop file system commands, DistCP commands, create Hive tables, and so on using the CLI.

Hadoop File System commands

You can execute Hadoop file system commands using CLI on your cluster.

- Access your cluster by using the URL of the container.

```
abfs://<NAME OF CONTAINER>@<NAME OF ACCOUNT>.dfs.core.windows.net/
```

- Create a directory using the mkdir command. For example,

  ```
  hadoop fs -mkdir abfs://abfscontainer@abfstorageacc.dfs.core.windows.net/
  myDir/
  hadoop fs -ls abfs://abfscontainer@abfstorageacc.dfs.core.windows.net/

  Found 2 items

  drwxr-xr-x - root root 0 2020-05-20 15:14
  abfs://abfscontainer@abfstorageacc.dfs.core.windows.net/myDir
  drwxr-x--- - root root 0 2020-05-20 12:50
  abfs://abfscontainer@abfstorageacc.dfs.core.windows.net/test
  ```

  Similarly, you can use the `put` command to add a file to one of the directories, `ls command` to list the items in your cluster, and `cat` command to read the contents of the file.

Note: If you name a file with the same file name that is already present in the directory, the existing file `is not overwritten`.

Create a table in Hive

You can create, modify, update, and remove tables in Hive using beeline or any other tool to access Hive.

1. Enter the beeline command shell by beeline command in your cluster:

   ```
   ~ beelinex
   ```

2. Enter the database you want to access.

   ```
   ~ use <DATABASE_NAME>;
   ```

   Or create and use a new database. In this following example, abfsdb is the name of the database.

   ```
   create database abfsdb;
   ~ use abfsdb;
   ```

3. Create a table inside the container in a directory named table_1. If the directory does not exist, it is automatically created:

   ```
   ~ create external table myTable(key STRING, value INT) location
   'abfs://abfscontainer@abfstorageacc.dfs.core.windows.net/table_1/';
   ```

4. View the table structure using the `show` command:

   ```
   ~ show create table myTable;
   INFO  : Compiling command(queryId=hive_20200520153116_301af680-9630-49d1-
   af40-3dfa5349e52a): show create table myTable
   INFO  : Semantic Analysis Completed (retrial = false)
   INFO  : Created Hive schema: Schema(fieldSchemas:[FieldSchema(name:createt
   ab_stmt, type:string, comment:from deserializer)], properties:null)
   INFO  : Completed compiling command(queryId=hive_20200520153116_301af680
   -9630-49d1-af40-3dfa5349e52a); Time taken: 0.052 seconds
   INFO  : Executing command(queryId=hive_20200520153116_301af680-9630-49d1-
   af40-3dfa5349e52a): show create table myTable
   INFO  : Starting task [Stage-0:DDL] in serial mode
   INFO  : Completed executing command(queryId=hive_20200520153116_301af680-
   9630-49d1-af40-3dfa5349e52a); Time taken: 0.072 seconds
   INFO  : OK

   +----------------------------------------------------+
   |                    createtab_stmt                  |
   ```

```
+----------------------------------------------------+
| CREATE EXTERNAL TABLE `myTable`(                   |
|   `key` string,                                    |
|   `value` int)                                     |
| ROW FORMAT SERDE                                   |
|   'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'  |
| STORED AS INPUTFORMAT                              |
|   'org.apache.hadoop.mapred.TextInputFormat'       |
| OUTPUTFORMAT                                       |
|   'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat' |
| LOCATION                                           |
|   'abfs://abfscontainer@abfstorageacc.dfs.core.windows.net/table_1' |
| TBLPROPERTIES (                                    |
|   'bucketing_version'='2',                         |
|   'transient_lastDdlTime'='1589988617')            |
+----------------------------------------------------+
14 rows selected (0.162 seconds)
```

After the table is created inside the Azure storage, it behaves like a regular table and you can use all the `hql` commands. For example, to insert values in the table, run ~ insert into myTable values("myKey", 1);.

Accessing Azure Storage account container from spark-shell

You can use spark-shell to query the files that are stored in the Azure Storage account. You should be able to access spark as an `hdfs` user using the `~ sudo -u hdfs -s` command.

```
~ spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use set
LogLevel(newLevel).
20/05/20 16:35:32 WARN cluster.YarnSchedulerBackend$YarnSchedulerEndpoint: A
ttempted to request executors before the AM has registered!
Spark context available as 'sc' (master = yarn, app id = application_1589987
399184_0004).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.0.7.1.1.0-413
      /_/
Using Scala version 2.11.12 (OpenJDK 64-Bit Server VM, Java 1.8.0_232)
Type in expressions to have them evaluated.
Type :help for more information.

scala> val sampleRDD = sc.textFile("abfs://abfscontainer@abfstorageacc.dfs
.core.windows.net/myDir/testingFile.txt")
sampleRDD: org.apache.spark.rdd.RDD[String] = abfs://abfscontainer@abfstor
ageacc.dfs.core.windows.net/myDir/testingFile.txt MapPartitionsRDD[1] at tex
tFile at <console>:24

scala> sampleRDD.collect().foreach(println)
testing the hadoop fs commands on azure.
```

Hence, after accessing the azure container, the data in `sampleRDD` would work like any other text file. Now, you can use any spark operation on these flies. For example,

- Find the word count of a particular word:

  ```
  scala> sampleRDD.filter(line => line.contains("azure")).count()
  res4: Long = 1
  ```

- Use wordcount by MapReduce, as popularized by Hadoop. Spark can implement MapReduce flows easily:

```
scala> val wordcount = sampleRDD.flatMap(line => line.split(" ")).map(word
 => (word,1)).reduceByKey(_+_)
wordcount: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[7] at
reduceByKey at <console>:26
scala> wordcount.collect()
res7: Array[(String, Int)] = Array((commands,1), (testing,1), (hadoop,1),
 (on,1), (fs,1), (azure.,1), (the,1))
```

Copying data with Hadoop DistCp

DistCp (distributed copy) is a tool used to copy files in large inter-cluster and intra-cluster environments. It uses MapReduce to affect its distribution, error handling and recovery, and reporting. It expands a list of files and directories into input to map tasks, each of which copy a partition of the files specified in the source list.

The following are some of the examples of distcp commands with object stores:

- Copying between directories in an object store

```
~ hadoop distcp
abfs://abfscontainer@abfstorageacc.dfs.core.windows.net/myDir/testingFi
le.txt \ abfs://abfscontainer@abfstorageacc.dfs.core.windows.net/test/

20/05/21 08:48:09 INFO mapreduce.Job: Job job_1589987399184_0005
completed successfully

~hadoop fs -ls abfs://abfscontainer@abfstorageacc.dfs.core.windows.net/tes
t/

Found 1 items

-rw-r--r--   1 hdfs hdfs         41 2020-05-21 08:48
abfs://abfscontainer@abfstorageacc.dfs.core.windows.net/test/testingFil
e.txt
```

- Copying between two different object stores

```
~ hadoop distcp
abfs://abfscontainer@abfstorageacc.dfs.core.windows.net/myDir/testingFi
le.txt \ abfs://mycontainer@mystoragehastoexist.dfs.core.windows.net/new
test/

20/05/21 08:53:26 INFO mapreduce.Job: Job job_1589987399184_0007
completed successfully
~ hadoop fs -ls
abfs://mycontainer@mystoragehastoexist.dfs.core.windows.net/newtest/

Found 1 items

-rw-r--r--   1 hdfs hdfs         41 2020-05-21 08:53
abfs://mycontainer@mystoragehastoexist.dfs.core.windows.net/newtest/tes
tingFile.txt
```

For more information about the DistCp commands, see DistCP documentation.

DistCp and Proxy Settings

When using DistCp to back up data from an on-site Hadoop cluster, proxy settings may need to be set so as to reach the cloud store. For most of the stores, these proxy settings are hadoop configuration options which must be set in core-site.xml, or as options to the DistCp command.

ADLS uses the JVM proxy settings, which need to be set in DistCp's map and reduce processes. This can be done through the `mapreduce.map.java.opts` and `mapreduce.reduce.java.opts` options respectively.

```
export DISTCP_PROXY_OPTS="-Dhttps.proxyHost=web-proxy.example.com -Dhttps.pr
oxyPort=80"
hadoop distcp \
  -D mapreduce.map.java.opts="$DISTCP_PROXY_OPTS" \
  -D mapreduce.reduce.java.opts="$DISTCP_PROXY_OPTS" \
  -update -skipcrccheck -numListstatusThreads 40 \
  hdfs://namenode:8020/users/alice adl://backups.azuredatalakestore.net/us
ers/alice
```

Without these settings, even though access to ADLS may work from the command line, distcp access can fail with Error fetching access token.

ADLS Trash Folder Behavior

If the fs.trash.interval property is set to a value other than zero on your cluster and you do not specify the -skipTrash flag with your rm command when you remove files, the deleted files are moved to the trash folder in your ADLS account. The trash folder in your ADLS account is located at adl://your_account.azuredatalakestore.net/user/user_name/.Trash/current/.

For more information about HDFS trash, see Configuring HDFS Trash.

Troubleshooting ABFS

You might encounter issues when configuring the ABFS connector. The issues are usually related to classpath, network, and authentication. For tips to troubleshoot these issues, see https://hadoop.apache.org/docs/current/hadoop-azure/abfs.html#Troubleshooting.