Cloudera Runtime 7.3.1

Using Apache Iceberg with Spark

Date published: 2020-07-28
Date modified: 2024-12-10

CLOUDERA

Legal Notice

Contents

Using Apache Iceberg in CDS

CDS supports Apache Iceberg which provides a table format for huge analytic datasets. Iceberg enables you to work with large tables, especially on object stores, and supports concurrent reads and writes on all storage media. You can use CDS running Spark 3 to interact with Apache Iceberg tables.

Prerequisites and limitations for using Iceberg in Spark

To use Apache Iceberg with Spark, you must meet the following prerequisite:

• CDS 3 with CDP Private Cloud Base 7.1.9

Limitations

• Iceberg tables with equality deletes do not support partition evolution or schema evolution on Primary Key columns.

  Users should not do partition evolution on tables with Primary Keys or Identifier Fields available, or do Schema Evolution on Primary Key columns, Partition Columns, or Identifier Fields from Spark.
• The use of Iceberg tables as Structured Streaming sources or sinks is not supported.
• PyIceberg is not supported. Using Spark SQL to query Iceberg tables in PySpark is supported.

Iceberg table format version 2

Iceberg table format version 1 and 2 (v1 and v2) is available. Iceberg table format v2 uses row-level UPDATE and DELETE operations that add deleted files to encoded rows that were deleted from existing data files. The DELETE, UPDATE, and MERGE operations function by writing delete files instead of rewriting the affected data files. Additionally, upon reading the data, the encoded deletes are applied to the affected rows that are read. This functionality is called merge-on-read.

With Iceberg table format version 1 (v1), the above-mentioned operations are only supported with copy-on-write where data files are rewritten in their entirety when rows in the files are deleted. Merge-on-read is more efficient for writes, while copy-on-write is more efficient for reads.

> **Note:** Unless otherwise indicated, the operations in the subsequent documentation apply to both v1 and v2 formats.

Accessing Iceberg tables

CDP uses Apache Ranger to provide centralized security administration and management. The Ranger Admin UI is the central interface for security administration. You can use Ranger to create two policies that allow users to query Iceberg tables.

How you open the Ranger Admin UI differs from one CDP service to another. In Management Console, you can select your environment, and then click Environment Details Quick Links Ranger .

You log into the Ranger Admin UI, and the Ranger Service Manager appears.
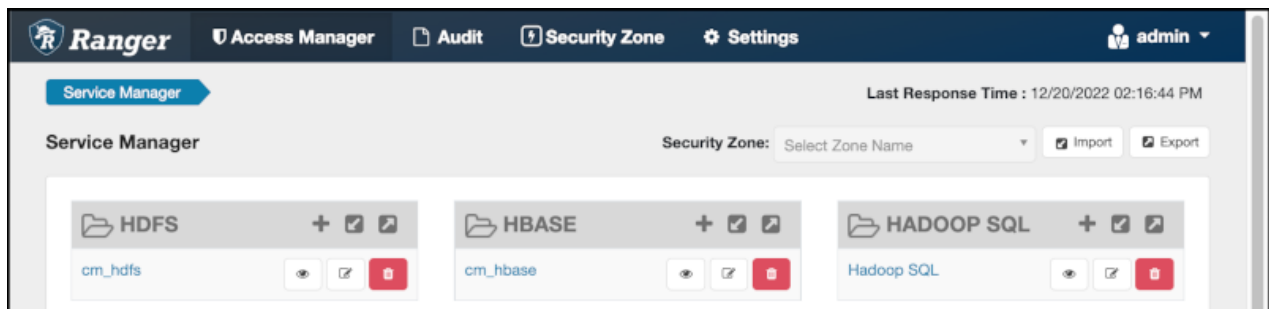
Policies for accessing tables on HDFS

The default policies that appear differ from service to service. You need to set up two Hadoop SQL policies to query Iceberg tables:

- One to authorize users to access the Iceberg files

    Follow steps in "Editing a policy to access Iceberg files" below.
- One to authorize users to query Iceberg tables

    Follow steps in "Creating a policy to query an Iceberg table" below.

Prerequisites

- Obtain the RangerAdmin role.
- Get the user name and password your Administrator set up for logging into the Ranger Admin.

    The default credentials for logging into the Ranger Admin Web UI are admin/admin123.

Editing a storage handler policy to access Iceberg files on the file system

You learn how to edit the existing default Hadoop SQL Storage Handler policy to access files. This policy is one of the two Ranger policies required to use Iceberg.

About this task

The Hadoop SQL Storage Handler policy allows references to Iceberg table storage location, which is required for creating or altering a table. You use a storage handler when you create a file stored as Iceberg on the file system or object store.

In this task, you specify Iceberg as the storage-type and allow the broadest access by setting the URL to *.

The Hadoop SQL Storage Handler policy supports only the RW Storage permission. A user having the required RW Storage permission on a resource, such as Iceberg, that you specify in the storage-type properties, is allowed only to reference the table location (for create/alter operations) in Iceberg. The RW Storage permission does not provide access to any table data. You need to create the Hadoop SQL policy described in the next topic in addition to this Hadoop SQL Storage Handler policy to access data in tables.

For more information about these policy settings, see Ranger Storage Handler documentation.

Procedure

1. Log into Ranger Admin Web UI.
   The Ranger Service Manager appears:
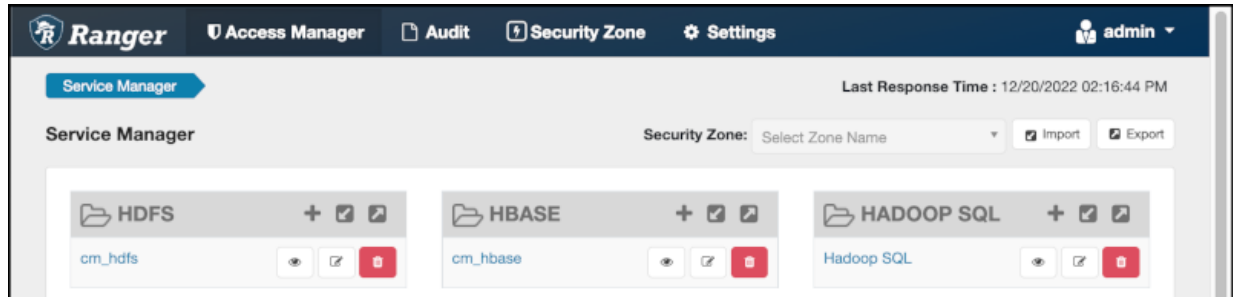
2. In Policy Name, enable the all - storage-type, storage-url policy.

3.
   In Service Manager, in Hadoop SQL, select Edit ✐ and edit the all storage-type, storage-url policy.

4. Below Policy Label, select storage-type, and enter iceberg..

5. In Storage URL, enter the value *, enable Include.

For more information about these policy settings, see Ranger storage handler documentation.

6. In Allow Conditions, specify roles, users, or groups to whom you want to grant RW storage permissions.

You can specify PUBLIC to grant access to Iceberg tables permissions to all users. Alternatively, you can grant access to one user. For example, add the systest user to the list of users who can access Iceberg:

For more information about granting permissions, see Configure a resource-based policy: Hadoop-SQL.

7. Add the RW Storage permission to the policy.

8. Save your changes.

Creating a SQL policy to query an Iceberg table

You learn how to set up the second required policy for using Iceberg. This policy manages SQL query access to Iceberg tables.

About this task

You create a Hadoop SQL policy to allow roles, groups, or users to query an Iceberg table in a database. In this task, you see an example of just one of many ways to configure the policy conditions. You grant (allow) the selected roles, groups, or users the following add or edit permissions on the table: Select, Update, Create, Drop, Alter, and All. You can also deny permissions.

For more information about creating this policy, see Ranger documentation.

Procedure

1. Log into Ranger Admin Web UI.
   The Ranger Service Manager appears.

2. Click Add New Policy.

3. Fill in required fields.
   For example, enter the following required settings:

   - In Policy Name, enter the name of the policy, for example IcebergPolicy1.
   - In database, enter the name of the database controlled by this policy, for example icedb.
   - In table, enter the name of the table controlled by this policy, for example icetable.
   - In columns, enter the name of the column controlled by this policy, for example enter the wildcard asterisk (*) to allow access to all columns of icetable.
   - Accept defaults for other settings.

4. Scroll down to Allow Conditions, and select the roles, groups, or users you want to access the table.

   You can use Deny All Other Accesses to deny access to all other roles, groups, or users other than those specified in the allow conditions for the policy.

5. Select permissions to grant.

For example, select Create, Select, and Alter. Alternatively, to provide the broadest permissions, select All.

Ignore RW Storage and other permissions not named after SQL queries. These are for future implementations.

6. Click Add.

Creating a new Iceberg table from Spark 3

You can create an Iceberg table using Spark SQL.

Note: By default, Iceberg tables are created in the v1 format.

An example Spark SQL creation command to create a new Iceberg table is as follows:

```
spark.sql("""CREATE EXTERNAL TABLE ice_t (idx int, name string, state string
)
USING iceberg
PARTITIONED BY (state)""")
```

For information about creating tables, see the Iceberg documentation.

Creating an Iceberg table format v2

To use the Iceberg table format v2, set the format-version property to 2 as shown below:

```
CREATE TABLE logs (app string, lvl string, message string, event_ts timestam
p) USING iceberg TBLPROPERTIES ('format-version' = '2')
```

<delete-mode> <update-mode> and <merge-mode> can be specified during table creation for modes of the respective operation. If unspecified, they default to merge-on-read.

Unsupported Feature: CREATE TABLE … LIKE

The CREATE TABLE ... LIKE feature is not supported in Spark:

```
CREATE TABLE <target> LIKE <source> USING iceberg
```

Here, <source> is an existing Iceberg table. This operation may appear to succeed and does not display errors and only warnings, but the resulting table is not a usable table.

Configuring Hive Metastore for Iceberg column changes

To make schema changes to an existing column of an Iceberg table, you must configure the Hive Metastore of the Data Lake.

Procedure

1. In Cloudera Manager, select the service for the Hive Metastore.
2. Click the Configuration tab.
3. Search for safety valve and find the Hive Metastore Server Advanced Configuration Snippet (Safety Valve) for hive-site.xml safety valve.
4. Add the following property:
   - Name: hive.metastore.disallow.incompatible.col.type.changes
   - Value: false
5. Click Save Changes.
6. Restart the service to apply the configuration change.

Importing and migrating Iceberg table in Spark 3

Importing or migrating tables are supported only on existing external Hive tables. When you import a table to Iceberg, the source and destination remain intact and independent. When you migrate a table, the existing Hive table is converted into an Iceberg table. You can use Spark SQL to import or migrate a Hive table to Iceberg.

Importing

Call the snapshot procedure to import a Hive table into Iceberg using a Spark 3 application.

```
spark.sql("CALL
<catalog>.system.snapshot('<src>', '<dest>')")
```

Definitions:

- <src> is the qualified name of the Hive table
- <dest> is the qualified name of the Iceberg table to be created
- <catalog> is the name of the catalog, which you pass in a configuration file. For more information, see Configuring Catalog linked below.

For example:

```
spark.sql("CALL
spark_catalog.system.snapshot('hive_db.hive_tbl',
'iceberg_db.iceberg_tbl')")
```

For information on compiling Spark 3 application with Iceberg libraries, see Iceberg library dependencies for Spark applications linked below.

Migrating

When you migrate a Hive table to Iceberg, a backup of the table, named <table_name>_backup_, is created.

Ensure that the TRANSLATED_TO_EXTERNAL property, that is located in TBLPROPERTIES, is set to false before migrating the table. This ensures that a table backup is created by renaming the table in Hive metastore (HMS) instead of moving the physical location of the table. Moving the physical location of the table would entail copying files in Amazon s3.

We recommend that you refrain from dropping the backup table, as doing so will invalidate the newly migrated table.

If you want to delete the backup table, set the following:

```
'external.table.purge'='FALSE'
```

Note: For Cloudera Data Engineering Private Cloud 1.5.2 and above, the property will be set automatically.

Deleting the backup table in the manner above will prevent underlying data from being deleted, therefore, only the table will be deleted from the metastore.

To undo the migration, drop the migrated table and restore the Hive table from the backup table by renaming it.

Call the migrate procedure to migrate a Hive table to Iceberg.

```
spark.sql("CALL
<catalog>.system.migrate('<src>')")
```

Definitions:

- <src> is the qualified name of the Hive table
- <catalog> is the name of the catalog, which you pass in a configuration file. For more information, see Configuring Catalog linked below.

For example:

```
spark.sql("CALL
spark_catalog.system.migrate('hive_db.hive_tbl')")
```

Importing and migrating Iceberg table format v2

Importing or migrating Hive tables Iceberg table formats v2 are supported only on existing external Hive tables. When you import a table to Iceberg, the source and destination remain intact and independent. When you migrate a table, the existing Hive table is converted into an Iceberg table. You can use Spark SQL to import or migrate a Hive table to Iceberg.

Importing

Call the snapshot procedure to import a Hive table into Iceberg table format v2 using a Spark 3 application.

```
spark.sql("CALL
<catalog>.system.snapshot(source_table => '<src>',
table => '<dest>',
properties => map('format-version', '2', 'write.delete.mode', '<delete-mo
de>',
'write.update.mode', '<update-mode>',
'write.merge.mode', '<merge-mode>'))")
```

Definitions:

- <src> is the qualified name of the Hive table
- <dest> is the qualified name of the Iceberg table to be created
- <catalog> is the name of the catalog, which you pass in a configuration file. For more information, see Configuring Catalog linked below.
- <delete-mode> <update-mode> and <merge-mode> are the modes that shall be used to perform the respective operation. If unspecified, they default to 'merge-on-read'

For example:

```
spark.sql("CALL
```

```
spark_catalog.system.snapshot('hive_db.hive_tbl',
 'iceberg_db.iceberg_tbl')")
```

For information on compiling Spark 3 application with Iceberg libraries, see Iceberg library dependencies for Spark applications linked below.

Migrating

Call the migrate procedure to migrate a Hive table to Iceberg.

```
spark.sql("CALL
<catalog>.system.migrate('<src>',
map('format-version', '2',
 'write.delete.mode', '<delete-mode>',
 'write.update.mode', '<update-mode>',
 'write.merge.mode', '<merge-mode>'))")
```

Definitions:

- <src> is the qualified name of the Hive table
- <catalog> is the name of the catalog, which you pass in a configuration file. For more information, see Configuring Catalog linked below.
- <delete-mode> <update-mode> and <merge-mode> are the modes that shall be used to perform the respective operation. If unspecified, they default to 'merge-on-read'

For example:

```
spark.sql("CALL
spark_catalog.system.migrate('hive_db.hive_tbl',
map('format-version', '2',
 'write.delete.mode', 'merge-on-read',
 'write.update.mode', 'merge-on-read',
 'write.merge.mode', 'merge-on-read'))")
```

Upgrading Iceberg table format v1 to v2

To upgrade an Iceberg table format from v1 to v2, run an `ALTER TABLE` command as follows:

```
spark.sql("ALTER TABLE <table_name> SET TBLPROPERTIES('merge-on-read', '
2')")
```

<delete-mode>,<update-mode>, and <merge-mode> can be specified as the modes that shall be used to perform the respective operation. If unspecified, they default to 'merge-on-read'

Configuring Catalog

When using Spark SQL to query an Iceberg table from Spark, you refer to a table using the following dot notation:

```
<catalog_name>.<database_name>.<table_name>
```

The default catalog used by Spark is named spark_catalog. When referring to a table in a database known to spark_ca talog, you can omit <catalog_name>. .

Iceberg provides a SparkCatalog property that understands Iceberg tables, and a SparkSessionCatalog property that understands both Iceberg and non-Iceberg tables. The following are configured by default:

```
spark.sql.catalog.spark_catalog=org.apache.iceberg.spark.SparkSessionCatalog
spark.sql.catalog.spark_catalog.type=hive
```

This replaces Spark's default catalog by Iceberg's SparkSessionCatalog and allows you to use both Iceberg and non-Iceberg tables out of the box.

There is one caveat when using SparkSessionCatalog. Iceberg supports CREATE TABLE … AS   SELECT (CTAS) and REPLACE TABLE … AS SELECT (RTAS) as atomic operations when using SparkCatalog. Whereas, the CTAS and RTAS are supported but are not atomic when using SparkSessionCatalog. As a workaround, you can configure another catalog that uses SparkCatalog. For example, to create the catalog named iceberg_catalog, set the following:

```
spark.sql.catalog.iceberg_catalog=org.apache.iceberg.spark.SparkCatalog
spark.sql.catalog.iceberg_catalog.type=hive
```

You can configure more than one catalog in the same Spark job. For more information, see the *Iceberg documentation*.

Related Tasks
Iceberg documentation

Loading data into an unpartitioned table

You can insert data into an unpartitioned table. The syntax to load data into an iceberg table:

```
INSERT INTO table_identifier [ ( column_list ) ]
    VALUES ( { value | NULL } [ , ... ] ) [ , ( ... ) ]
```

Or

```
INSERT INTO table_identifier [ ( column_list ) ]
    query
```

Example:

```
INSERT INTO students VALUES
    ('Amy Smith', '123 Park Ave, San Jose', 111111)
INSERT INTO students VALUES
    ('Bob Brown', '456 Taylor St, Cupertino', 222222),
    ('Cathy Johnson', '789 Race Ave, Palo Alto', 333333)
```

Querying data in an Iceberg table

To read the Iceberg table, you can use SparkSQL to query the Iceberg tables.

Example:

```
spark.sql("select * from ice_t").show(1000, false)
```

⚠ **Important:** When querying Iceberg tables in HDFS, CDS disables locality by default. Because enabling locality generally leads to increased Spark planning time for queries on such tables and often the increase is quite significant. If you wish to enable locality, set the spark.cloudera.iceberg.locality.enabled to true . For example, you can do it by passing --conf   spark.cloudera.iceberg.locality.enabled=true to your spark3-submit command.

Updating Iceberg table data

Iceberg table data can be updated using copy-on-write or merge-on-read. The table version you are using will determine how you can update the table data.

v1 format

Iceberg supports bulk updates through MERGE, by defaulting to copy-on-write deletes when using v1 table format.

v2 format

Iceberg table format v2 supports efficient row-level updates and delete operations leveraging merge-on-read.

For more details, refer to *Position Delete Files* linked below.

For updating data examples, see *Spark Writes* linked below.

Iceberg library dependencies for Spark applications

If your Spark application only uses Spark SQL to create, read, or write Iceberg tables, and does not use any Iceberg APIs, you do not need to build it against any Iceberg dependencies. The runtime dependencies needed for Spark to use Iceberg are in the Spark classpath by default. If your code uses Iceberg APIs, then you need to build it against Iceberg dependencies.

Cloudera publishes Iceberg artifacts to a Maven repository with versions matching the Iceberg in CDS.

Note: Use 1.3.0.7.1.9.0-387 iceberg version for compilation. The below iceberg dependencies should only be used for compilation. Including iceberg jars within a Spark application fat jar must be avoided.

```
<dependency>
      <groupId>org.apache.iceberg</groupId>
      <artifactId>iceberg-core</artifactId>
      <version>${iceberg.version}</version>
      <scope>provided</scope>
</dependency>
<!-- for org.apache.iceberg.hive.HiveCatalog -->
<dependency>
      <groupId>org.apache.iceberg</groupId>
      <artifactId>iceberg-hive-metastore</artifactId>
      <version>${iceberg.version}</version>
      <scope>provided</scope>
</dependency>
<!-- for org.apache.iceberg.spark.* classes if used -->
<dependency>
     <groupId>org.apache.iceberg</groupId>
     <artifactId>iceberg-spark</artifactId>
     <version>${iceberg.version}</version>
     <scope>provided</scope>
</dependency>
```

Alternatively, the following dependency can be used:

```
<dependency>
    <groupId>org.apache.iceberg</groupId>
    <artifactId>iceberg-spark-runtime-3.3_2.12</artifactId>
    <version>${iceberg.version}</version>
    <scope>provided</scope>
</dependency>
```

The iceberg-spark3-runtime JAR contains the necessary Iceberg classes for Spark runtime support, and includes the classes from the dependencies above.