Cloudera Runtime 7.1.9

CDS 3 Powered by Apache Spark

Date published: 2020-07-28
Date modified: 2024-09-09

https://docs.cloudera.com/

Legal Notice

© Cloudera Inc. 2024. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

Contents

CDS 3.3 Powered by Apache Spark Overview.....5

CDS 3.3 Powered by Apache Spark Requirements..... 5

Installing CDS 3.3 Powered by Apache Spark.....7

Enabling CDS 3.3 with GPU Support.....8
Set up a Yarn role group to enable GPU usage..... 8
Configure NVIDIA RAPIDS Shuffle Manager..... 8

Updating Spark 2 applications for Spark 3.....9

Running Applications with CDS 3.3 Powered by Apache Spark.....9
The Spark 3 job commands..... 9
Canary test for pyspark3 command..... 10
Fetching Spark 3 Maven Dependencies.....10
Accessing the Spark 3 History Server..... 10

Running applications using CDS 3.3 with GPU SupportCDS 3.3 with GPU Support.....11

CDS 3.3 Powered by Apache Spark version and download information..... 14

Using the CDS 3.3 Powered by Apache Spark Maven Repository..... 15

CDS 3.3 Powered by Apache Spark Maven Artifacts..... 15

Apache Spark 3 integration with Schema Registry..... 16
Configuration.....17
Fetching Spark schema by name..... 18
Building and deploying your app.....19
Running in a Kerberos-enabled cluster.....20
Unsupported features..... 21

Cumulative hotfixes for CDS..... 21
Cumulative hotfix CDS 3.3.7190.2-1 (CDS 3.3 CHF1 for 7.1.9)..... 21
Cumulative hotfix CDS 3.3.7190.3-1 (CDS 3.3 CHF2 for 7.1.9)..... 21

Cumulative hotfix CDS 3.3.7190.4-1 (CDS 3.3 CHF3 for 7.1.9)....22
Cumulative hotfix CDS 3.3.7190.5-2 (CDS 3.3 CHF4 for 7.1.9)....23
Cumulative hotfix CDS 3.3.7190.7-2 (CDS 3.3 CHF5 for 7.1.9)....24
Cumulative hotfix CDS 3.3.7190.8-2 (CDS 3.3 CHF6 for 7.1.9)....24

Using Apache Iceberg in CDS....25

Prerequisites and limitations for using Iceberg in Spark....25
Accessing Iceberg tables....25
Editing a storage handler policy to access Iceberg files on HDFS or S3....26
Creating a SQL policy to query an Iceberg table....28
Creating a new Iceberg table from Spark 3....30
Configuring Hive Metastore for Iceberg column changes....31
Importing and migrating Iceberg table in Spark 3....31
Importing and migrating Iceberg table format v2....32
Configuring Catalog....33
Loading data into an unpartitioned table....34
Querying data in an Iceberg table....34
Updating Iceberg table data....34
Iceberg library dependencies for Spark applications....35

CDS 3.3 Powered by Apache Spark Overview

Apache Spark is a general framework for distributed computing that offers high performance for both batch and interactive processing. It exposes APIs for Java, Python, and Scala. This document describes CDS 3.3 Powered by Apache Spark. CDS (Cloudera Distribution of Spark) enables you to install and evaluate the features of Apache Spark 3 without upgrading your CDP Private Cloud Base cluster.

For detailed API information, see the Apache Spark project site.

CDS 3.3 Powered by Apache Spark is an add-on service for CDP Private Cloud Base, distributed as a parcel and the Cloudera Service Descriptor file is available in Cloudera Manager for CDP 7.1.9.

On CDP Private Cloud Base, a Spark 3 service can coexist with the existing Spark 2 service. The configurations of the two services do not conflict and both services use the same YARN service. The port of the Spark History Server is 18088 for Spark 2 and 18089 for Spark 3.

Note: Spark 3.3 is the first Spark version with the log4j2 dependency. Previous versions contained the log4j1 dependency. If you are using any custom logging related changes, you must rewrite the original log4j properties' files using log4j2 syntax, that is, XML, JSON, YAML, or properties format.

CDS 3 for GPUs

CDS 3.3 with GPU Support is an add-on service that enables you to take advantage of the RAPIDS Accelerator for Apache Spark to accelerate Apache Spark 3 performance on existing CDP Private Cloud Base clusters.

Unsupported connectors

This release does not support the following connectors:

- SparkR

Unsupported Features

This release does not support the following feature:

- Hudi

Limitations of Spark in CDP

Limitations of Spark (in comparison to Apache Spark 3.3) in CDP are described below:

- spark.sql.orc.compression.codec config doesn't accept zsdt value.

CDS 3.3 Powered by Apache Spark Requirements

The following sections describe software requirements for CDS 3.3 Powered by Apache Spark.

CDP Versions

Important: CDS 3.3 Powered by Apache Spark is an add-on service for CDP Private Cloud Base, and is only supported with Cloudera Runtime 7.1.9. Spark 2 is included in CDP, and does not require a separate parcel.

Supported versions of CDP are described below.

CDS Powered by Apache Spark Version	Supported CDP Versions
3.3.2.3.3.7191000.0-78	CDP Private Cloud Base with Cloudera Runtime 7.1.9

A Spark 2 service (included in CDP) can co-exist on the same cluster as Spark 3 (installed as a separate parcel). The two services are configured to not conflict, and both run on the same YARN service. Spark 3 installs and uses its own external shuffle service.

Although Spark 2 and Spark 3 can coexist in the same CDP Private Cloud Base cluster, you cannot use multiple Spark 3 versions simultaneously. All clusters managed by the same Cloudera Manager Server must use exactly the same version of CDS Powered by Apache Spark.

Software requirements

For CDS 3.3

Each cluster host must have the following software installed:

Java

JDK 8, JDK 11 or JDK 17. Cloudera recommends using JDK 8, as most testing has been done with JDK 8. Remove other JDK versions from all cluster and gateway hosts to ensure proper operation.

Python

Python 3.7 - 3.10

For CDS for GPUs

Each cluster host with a GPU must have the following software installed:

Java

JDK 8, JDK 11 or JDK 17. Cloudera recommends using JDK 8, as most testing has been done with JDK 8. Remove other JDK versions from all cluster and gateway hosts to ensure proper operation.

Python

Python 3.7 - 3.10

GPU drivers and CUDA toolkit

GPU driver v450.80.02 or higher

CUDA version 11.0 or higher

Download and install the CUDA Toolkit for your operating system. The toolkit installer also provides the option to install the GPU driver.

NVIDIA Library

NVIDIA RAPIDS version 23.06.0. For more information, see NVIDIA Release Notes

UCX (Optional)

Clusters with Infiniband or RoCE networking can leverage Unified Communication X (UCX) to enable the RAPIDS Shuffle Manager. For information on UCX native libraries support, see (Optional) Installing UCX native libraries.

Hardware requirements

For CDS 3.3

CDS 3.3 Powered by Apache Spark has no specific hardware requirements on top of what is required for Cloudera Runtime deployments.

For CDS for GPUs

CDS 3.3 with GPU Support requires cluster hosts with NVIDIA Pascal™ or better GPUs, with a compute capability rating of 6.0 or higher.

For more information, see Getting Started at the RAPIDS website.

Cloudera and NVIDIA recommend using NVIDIA-certified systems. For more information, see NVIDIA-Certified Systems in the NVIDIA GPU Cloud documentation.

Installing CDS 3.3 Powered by Apache Spark

CDS 3.3 Powered by Apache Spark (CDS 3.3) is distributed as a parcel. There are no external Custom Service Descriptors (CSD) for Livy for Spark3 or Spark3 using CDS 3.3 for CDP-7.1.9 parcel because they are already part of Cloudera Manager 7.11.3.

Note: Due to the potential for confusion between CDS Powered by Apache Spark and the initialism CSD, references to the custom service descriptor (CSD) file in this documentation use the term service descriptor.

Install CDS 3.3 Powered by Apache Spark

Note:

Although Spark 2 and Spark 3 can coexist in the same CDP Private Cloud Base cluster, you cannot use multiple Spark 3 versions simultaneously. All clusters managed by the same Cloudera Manager Server must use exactly the same version of CDS 3.3 Powered by Apache Spark.

Follow these steps to install CDS 3.3:

1. Check that all the software prerequisites are satisfied. If not, you might need to upgrade or install other software components first.
2. In the Cloudera Manager Admin Console, add the CDS parcel repository to the Remote Parcel Repository URLs in Parcel Settings as described in Parcel Configuration Settings.

   Note: If your Cloudera Manager Server does not have Internet access, you can use the CDS Powered by Apache Spark parcel files: put them into a new parcel repository, and then configure the Cloudera Manager Server to target this newly created repository.
3. Download the CDS 3.3 parcel, distribute the parcel to the hosts in your cluster, and activate the parcel. For instructions, see Managing Parcels.

   Note: Spark3 RPM and DEB packages are available from CDP Private Cloud version 7.1.9 SP1, but currently do not contain Rapids and UCX.
4. Add the Spark 3 service to your cluster.
   a. In step 1, select any optional dependencies, such as HBase and Hive, or select No Optional Dependencies.
   b. In step 2, when customizing the role assignments, add a gateway role to every host.
   c. On the Review Changes page, you can enable TLS for the Spark History Server.
   d. Note that the History Server port is 18089 instead of the usual 18088.
   e. Complete the remaining steps in the wizard.
5. Return to the Home page by clicking the Cloudera Manager logo in the upper left corner.
6. Click the stale configuration icon to launch the Stale Configuration wizard and restart the necessary services.

Install the Livy for Spark 3 Service

CDS 3 supports Apache Livy, but it cannot use the included Livy service, which is compatible with only Spark 2. To add and manage a Livy service compatible with Spark 3, you must install the Livy for Spark 3 service.

1. In the Cloudera Manager Admin Console, add the CDS 3 parcel repository to the Remote Parcel Repository URLs in Parcel Settings as described in Parcel Configuration Settings.

   Note: If your Cloudera Manager Server does not have Internet access, you can use the Livy parcel files: put them into a new parcel repository, and then configure the Cloudera Manager Server to target this newly created repository.

2. Download the CDS 3.3 parcel, distribute the parcel to the hosts in your cluster, and activate the parcel. For instructions, see Managing Parcels.
3. Add the Livy for Spark 3 service to your cluster.
   a. Note that the Livy port is 28998 instead of the usual 8998.
   b. Complete the remaining steps in the wizard.
4. Return to the Home page by clicking the Cloudera Manager logo in the upper left corner.
5. Click the stale configuration icon to launch the Stale Configuration wizard and restart the necessary services.

If you want to activate the CDS 3.3 with GPU Support feature, Set up a Yarn role group to enable GPU usage on page 8 and optionally Configure NVIDIA RAPIDS Shuffle Manager on page 8

Enabling CDS 3.3 with GPU Support

To activate the CDS 3.3 with GPU Support feature on suitable hardware, you need to create a Yarn role group and optionally make configuration changes to enable the NVIDIA RAPIDS Shuffle Manager.

Set up a Yarn role group to enable GPU usage

Create a Yarn role group so that you can selectively enable GPU usage for nodes with GPUs within your cluster.

Before you begin

GPU scheduling and isolation must be configured.

About this task

Enabling GPU on YARN through the Enable GPU Usage tickbox operates on cluster-level. Role groups in Yarn enable you to apply settings selectively, to a subset of nodes within your cluster.

Role groups are configured on the service level.

Procedure

1. In Cloudera Manager navigate to Yarn Instances .
2. Create a role group where you can add nodes with GPUs.

   For more information, see Creating a Role Group.
3. Move role instances with GPUs to the group you created.

   On the Configuration tab select the source role group with the hosts you want to move, then click Move Selected Instances To Group and select the role group you created.

   You may need to restart the cluster.
4. Enable GPU usage for the role group.
   a) On the Configuration tab select Categories GPU Management .
   b) Under GPU Usage click Edit Individual Values and select the role group you created.
   c) Click Save Changes.

Configure NVIDIA RAPIDS Shuffle Manager

The NVIDIA RAPIDS Shuffle Manager is a custom ShuffleManager for Apache Spark that allows fast shuffle block transfers between GPUs in the same host (over PCIe or NVLink) and over the network to remote hosts (over RoCE or Infiniband).

About this task

NVIDIA RAPIDS Shuffle Manager has been shown to accelerate workloads where shuffle is the bottleneck when using the RAPIDS accelerator for Apache Spark. It accomplishes this by using a GPU shuffle cache for fast shuffle writes when shuffle blocks fit in GPU memory, avoiding the cost of writes to host using the built-in Spark Shuffle, a spill framework that will spill to host memory and disk on demand, and Unified Communication X (UCX) as its transport for fast network and peer-to-peer (GPU-to-GPU) transfers.

CDS 3.3 with GPU Support has built in support for UCX, no separate installation is required.

Cloudera and NVIDIA recommend using the RAPIDS shuffle manager for clusters with Infiniband or RoCE networking.

Procedure

1. Validate your UCX environment following the instructions provided in the NVIDIA spark-rapids documentation.
2. Before running applications with the RAPIDS Shuffle Manager, make the following configuration changes:

```
--conf "spark.shuffle.service.enabled=false" \
--conf "spark.dynamicAllocation.enabled=false"
```

3. You are recommended to make the following UCX settings:

```
spark.executorEnv.UCX_TLS=cuda_copy,cuda_ipc,rc,tcp
spark.executorEnv.UCX_RNDV_SCHEME=put_zcopy
spark.executorEnv.UCX_MAX_RNDV_RAILS=1
spark.executorEnv.UCX_IB_RX_QUEUE_LEN=1024
```

For more information on environment variables, see the NVIDIA spark-rapids documentation.

Note: Running a job with the --rapids-shuffle=true flag does not affect these optional settings. You need to set them manually.

Updating Spark 2 applications for Spark 3

You must update your Apache Spark 2 applications to run on Spark 3. The Apache Spark documentation provides a migration guide.

For instructions on updating your Spark 2 applications for Spark 3, see the migration guide in the Apache Spark documentation.

Running Applications with CDS 3.3 Powered by Apache Spark

With CDS 3.3 Powered by Apache Spark (CDS 3.3), you can run Apache Spark 3 applications locally or distributed across a cluster, either by using an interactive shell or by submitting an application. Running Spark applications interactively is commonly performed during the data-exploration phase and for ad hoc analysis.

The Spark 3 job commands

With Spark 3, you use slightly different command names than with Spark 2, so that you can run both versions of Spark side-by-side without conflicts:

- spark3-submit instead of spark-submit.

- spark3-shell instead of spark-shell.
- pyspark3 instead of pyspark.

For development and test purposes, you can also configure each host so that invoking the Spark 2 command name runs the corresponding Spark 3 executable.

Canary test for pyspark3 command

The following example shows a simple pyspark3 session that refers to the SparkContext, calls the collect() function which runs a Spark 3 job, and writes data to HDFS. This sequence of operations helps to check if there are obvious configuration issues that prevent Spark 3 jobs from working at all. For the HDFS path for the output directory, substitute a path that exists on your own system.

```
$ hdfs dfs -mkdir /user/jdoe/spark
$ pyspark3
...
SparkSession available as 'spark'.
>>> strings = ["one","two","three"]
>>> s2 = sc.parallelize(strings)
>>> s3 = s2.map(lambda word: word.upper())
>>> s3.collect()
['ONE', 'TWO', 'THREE']
>>> s3.saveAsTextFile('hdfs:///user/jdoe/spark/canary_test')
>>> quit()
$ hdfs dfs -ls /user/jdoe/spark
Found 1 items
drwxr-xr-x   - jdoe spark-users  0 2016-08-26 14:41 /user/jdoe/spark/canary_
test
$ hdfs dfs -ls /user/jdoe/spark/canary_test
Found 3 items
-rw-r--r--   3 jdoe spark-users  0 2016-08-26 14:41 /user/jdoe/spark/cana
ry_test/_SUCCESS
-rw-r--r--   3 jdoe spark-users  4 2016-08-26 14:41 /user/jdoe/spark/canary
_test/part-00000
-rw-r--r--   3 jdoe spark-users 10 2016-08-26 14:41 /user/jdoe/spark/canary
_test/part-00001
$ hdfs dfs -cat /user/jdoe/spark/canary_test/part-00000
ONE
$ hdfs dfs -cat /user/jdoe/spark/canary_test/part-00001
TWO
THREE
```

Fetching Spark 3 Maven Dependencies

The Maven coordinates are a combination of groupId, artifactId and version. The groupId and artifactId are the same as for the upstream Apache Spark project. For example, for spark-core, groupId is org.apache.spark, and artifactId is spark-core_2.12, both the same as the upstream project. The version is different for the Cloudera packaging: see Using the CDS 3 Powered by Apache Spark Maven Repository for the exact name depending on which release you are using.

Accessing the Spark 3 History Server

The Spark 3 history server is available on port 18089, rather than port 18088 as with the Spark 2 history server.

Running applications using CDS 3.3 with GPU SupportCDS 3.3 with GPU Support

1. Log on to the node where you want to run the job.
2. Run the following command to launch spark3-shell:

```
spark3-shell  --conf "spark.rapids.sql.enabled=true" \
              --conf "spark.executor.memoryOverhead=5g"
```

where

--conf spark.rapids.sql.enabled=true

enables the following environment variables for GPUs:

```
"spark.task.resource.gpu.amount" - sets GPU resource amount per
task

"spark.rapids.sql.concurrentGpuTasks" - sets the number of con
current tasks per GPU
"spark.sql.files.maxPartitionBytes" - sets the input partition
 size for DataSource API, The recommended value is "256m".
"spark.locality.wait" - controls how long Spark waits to obtain b
etter locality for tasks.
"spark.sql.adaptive.enabled" - enables Adaptive Query Execution.
"spark.rapids.memory.pinnedPool.size" - sets the amount of pinned
 memory allocated per host.
"spark.sql.adaptive.advisoryPartitionSizeInBytes" - sets the ad
visory size in bytes of the shuffle partition during adaptive op
timization.
```

For example,

```
$SPARK_HOME/bin/spark3-shell \
       --conf spark.task.resource.gpu.amount=2 \
       --conf spark.rapids.sql.concurrentGpuTasks=2 \
       --conf spark.sql.files.maxPartitionBytes=256m \
       --conf spark.locality.wait=0s \
       --conf spark.sql.adaptive.enabled=true \
       --conf spark.rapids.memory.pinnedPool.size=2G \
       --conf spark.sql.adaptive.advisoryPartitionSizeInBytes=1g
```

--conf "spark.executor.memoryOverhead=5g"

sets the amount of additional memory to be allocated per executor process

Note: cuDF uses a Just-In-Time (JIT) compilation approach for some kernels, and the JIT process can add a few seconds to query wall-clock time. You are recommended to set a JIT cache path to speed up subsequent invocations with: --conf spark.executorEnv.LIBCUDF_KERNEL_CACHE_PATH=[local path]. The path should be local to the executor (not HDFS) and not shared between different cluster users in a multi-tenant environment. For example, the path may be in /tmp: (/tmp/cudf-[***USER***]). If the/tmp directory is not writable, consult your system administrator to find a path that is.

You can override these configuration settings both from the command line and from code. For more information on environment variables, see the NVIDIA spark-rapids documentation and the Spark SQL Performance Tuning Guide.

3. Run a job in spark3-shell.

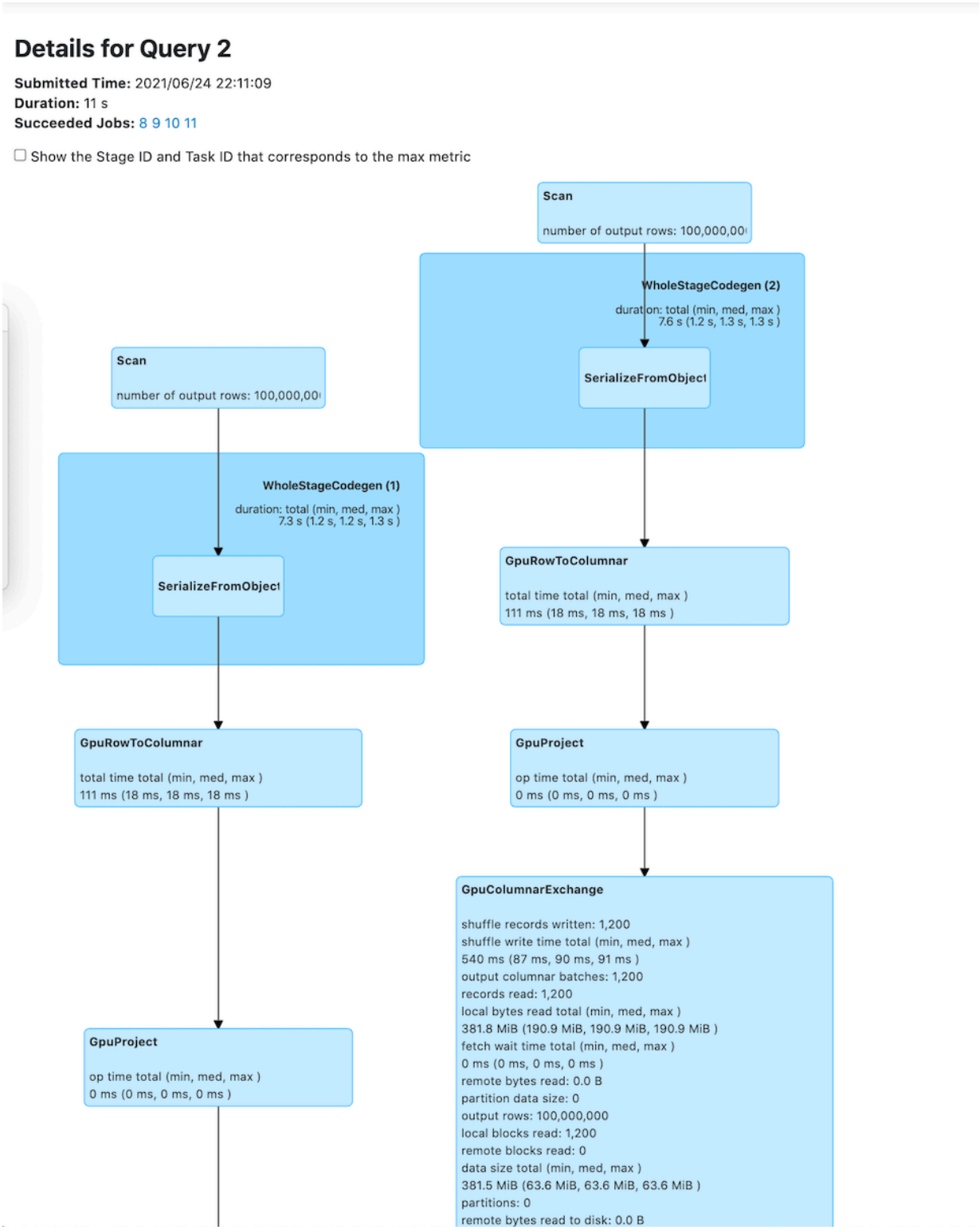
   For example:

```
scala> val df = sc.makeRDD(1 to 100000000, 6).toDF
df: org.apache.spark.sql.DataFrame = [value: int]

scala>val df2 = sc.makeRDD(1 to 100000000, 6).toDF
df2: org.apache.spark.sql.DataFrame = [value: int]
scala> df.select($"value" as "a").join(df2select($"value" as "b"), $"a"
 === $"b").count
res0: Long = 100000000
```

4. You can verify that the job run used GPUs, by logging on to the Yarn UI v2 to review the execution plan and the performance of your spark3-shell application:

   Select the Applications tab then select your [spark3-shell application]. Select ApplicationMaster SQL count at <console>:28 to see the execution plan.



Running a Spark job using CDS 3.3 with GPU Support with UCX enabled

1. Log on to the node where you want to run the job.
2. Run the following command to launch spark3-shell:

```
spark3-shell  --conf "spark.rapids.sql.enabled=true" \
              --conf "spark.executor.memoryOverhead=5g"
              --rapids-shuffle=true
```

   where
   --rapids-shuffle=true
   makes the following configuration changes for UCX:

```
spark.shuffle.manager=com.nvidia.spark.rapids.spark332cdh.Rapid
sShuffleManager
spark.executor.extraClassPath=/opt/cloudera/parcels/SPARK3/lib/s
park3/rapids-plugin/*
spark.executorEnv.UCX_ERROR_SIGNALS=
spark.executorEnv.UCX_MEMTYPE_CACHE=n
```

   For more information on environment variables, see the NVIDIA spark-rapids documentation.
3. Run a job in spark3-shell.

CDS 3.3 Powered by Apache Spark version and download information

Cloudera Service Descriptors (CSD) file for CDS 3.3 is available in Cloudera Manager for CDP 7.1.9. The following section provides links to the parcel for both CDS 3.3 and CDS 3.3 with GPU Support.

Note:

The CDS version label is constructed in v.v.v.w.w.xxxx.y-z...z format and carries the following information:

v.v.v
: Apache Spark upstream version, for example: 3.2.0

w.w
: Cloudera internal version number, for example: 3.2

xxxx
: CDP version number, for example: 7170 (referring to CDP Private Cloud Base 7.1.7)

y
: maintenance version, for example, 1

z...z
: build number, for example: 279

Table 1: Available CDS Versions

Version	Parcel Repository
3.3.2.3.3.7191000.0-78	https://archive.cloudera.com/p/spark3/3.3.7191000.0/parcels/

Using the CDS 3.3 Powered by Apache Spark Maven Repository

Important: CDS 3.3 Powered by Apache Spark (CDS 3.3) does not include an assembly JAR. When you build an application JAR, do not include Cloudera Runtime or CDS JARs, because they are already provided. If you do, upgrading Cloudera Runtime or CDS can break your application. To avoid this situation, set the Maven dependency scope to provided. If you have already built applications which include the Cloudera Runtime or CDS JARs, update the dependency to set scope to provided and recompile.

If you want to build applications or tools for use with CDS 3.3, and you are using Maven or Ivy for dependency management, you can pull the CDS artifacts from the Cloudera Maven repository. The repository is available at https://repository.cloudera.com/artifactory/cloudera-repos/.

The following is a sample POM (pom.xml) file:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.
org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache
.org/maven-v4_0_0.xsd">
  <repositories>
    <repository>
      <id>cloudera</id>
      <url>https://repository.cloudera.com/artifactory/cloudera-repos/</url>
    </repository>
  </repositories>
</project>
```

CDS 3.3 Powered by Apache Spark Maven Artifacts

The following table lists the groupId, artifactId, and version required to access the artifacts for CDS 3.3 Powered by Apache Spark:

POM fragment

The following pom fragment shows how to access a CDS 3.3 artifact from a Maven POM.

```
<dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.12</artifactId>
      <version>3.3.2.3.3.7191000.0-78</version>
      <scope>provided</scope>
</dependency>
```

The complete artifact list for this release follows.

List of CDS 3.3 Maven artifacts

Project	groupId	artifactId	version
Apache Spark	org.apache.spark	spark-assembly_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-avro_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-catalyst_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-core_2.12	3.3.2.3.3.7191000.0-78

Project	groupId	artifactId	version
	org.apache.spark	spark-graphx_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-hadoop-cloud_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-hive_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-kubernetes_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-kvstore_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-launcher_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-mllib-local_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-mllib_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-network-common_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-network-shuffle_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-network-yarn_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-parent_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-repl_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-sketch_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-sql-kafka-0-10_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-sql_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-streaming-kafka-0-10-assembly_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-streaming-kafka-0-10_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-streaming_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-tags_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-token-provider-kafka-0-10_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-unsafe_2.12	3.3.2.3.3.7191000.0-78
	org.apache.spark	spark-yarn_2.12	3.3.2.3.3.7191000.0-78

Apache Spark 3 integration with Schema Registry

Apache Spark 3 integrated with Schema Registry provides a library to leverage Schema Registry for managing Spark schemas and to serialize and/or de-serialize messages in Spark data sources and sinks.

Note: The Apache Spark 3 integration with Schema Registry is available in CDS 3.3 Powered by Apache Spark from version 3.3.2 for CDP Private Cloud Base 7.1.9 SP1.

Running the example programs

The examples illustrate the API usage and how to integrate with Schema Registry. The examples can be run from IDE (for example, Intellij) by specifying a master URL or by using spark3-submit.

```
spark3-submit --jars /opt/cloudera/parcels/SPARK3/lib/spark3/spark-schema-re
gistry-for-spark3/spark-schema-registry-for-spark3_2.12-jar-with-dependencie
s.jar \
--class com.hortonworks.spark.registry.examples.classname \
/opt/cloudera/parcels/SPARK3/lib/spark3/spark-schema-registry-for-spark
3/examples/spark-schema-registry-for-spark3-examples_2.12.jar [***SCHEMA-
REGISTRY-URL***] \
bootstrap-servers input-topic output-topic checkpoint-location
```

Using the APIs

Typically in a Spark application you define the Spark schema for the data you are going to process:

```
// the schema for truck events
val schema = StructType(Seq(
  StructField("driverId", IntegerType, nullable = false),
  StructField("truckId", IntegerType, nullable = false),
  StructField("miles", LongType, nullable = false),
  StructField("eventType", StringType, nullable = false),
  ...
  ...
)

// read Json string messages from the data source
val messages = spark
      .readStream
      .format(...)
      .option(...)
      .load()

// parse the messages using the above schema and do further operations
val df = messages
      .select(from_json($"value".cast("string"), schema).alias("value"))
      ...
// project (driverId, truckId, miles) for the events where miles > 300
val filtered = df.select($"value.driverId", $"value.truckId", $"value.miles"
)
      .where("value.miles > 300")
```

However, this approach is not practical because the schema information is tightly coupled with the code. The code needs to be changed when the schema changes, and there is no ability to share or reuse the schema between the message producers and the applications that consume the messages.

Using Schema Registry is a better solution because it enables you to manage different versions of the schema and define compatibility policies.

Configuration

The Schema Registry integration comes as a utility method which can be imported into the scope.

```
import com.hortonworks.spark.registry.util._
```

Before invoking the APIs, you need to define an implicit SchemaRegistryConfig which will be passed to the APIs. The main configuration parameter is the schema registry URL.

```
// the schema registry client config
val config = Map[String, Object]("[***SCHEMA.REGISTRY.URL***]" -> schemaR
egistryUrl)
// the schema registry config that will be implicitly passed
implicit val srConfig:SchemaRegistryConfig = SchemaRegistryConfig(config)
```

SSL configuration

SchemaRegistryConfig expects the following SSL configuration properties:

```
"schema.registry.client.ssl.protocol" -> "SSL",
"schema.registry.client.ssl.trustStoreType" -> "JKS",
"schema.registry.client.ssl.trustStorePath" -> "/var/lib/cloudera-scm-agent/
agent-cert/cm-auto-global_truststore.jks",
"schema.registry.client.ssl.trustStorePassword" ->
 "[***CHANGEMECLIENTPWD***]"
```

Fetching Spark schema by name

The API supports fetching the Schema Registry schema as a Spark schema.

- sparkSchema(schemaName: String)

  Returns the spark schema corresponding to the latest version of schema defined in the Schema Registry.
- sparkSchema(schemaName: String, version: Int)

  Returns the spark schema corresponding to the given version of schema defined in the Schema Registry.

Using the Schema Registry integration, the example previously shown can be simplified, as there is no need to explicitly specify the Spark schema in the code:

```
// retrieve the translated "Spark schema" by specifying the schema registry
schema name
val schema = sparkSchema(name)

// parse the messages using the above schema and do further operations
val df = messages
         .select(from_json($"value".cast("string"), schema).alias("value"))
         ...

// project (driverId, truckId, miles) for the events where miles > 300
val filtered = df.select($"value.driverId", $"value.truckId", $"value.miles
")
                  .where("value.miles > 300")
```

Serializing messages using Schema Registry

The following method can be used to serialize the messages from Spark to Schema Registry binary format using schema registry serializers.

- to_sr(data: Column, schemaName: String, topLevelRecordName: String, namespace: String)

  Converts a Spark column data to binary format of Schema Registry. This looks up a Schema Registry schema for the schemaName that matches the input and automatically registers a new schema, if not found. The topoLevelRecordName and namespace are optional and will be mapped to Avro top level record name and record namespace.

De-serializing messages using Schema Registry

The following methods can be used to de-serialize Schema Registry serialized messages into Spark columns.

- from_sr(data: Column, schemaName: String)

  Converts Schema Registry binary format to Spark column, using the latest version of the schema.

- from_sr(data: Column, schemaName: String, version: Int)

  Converts Schema Registry binary format to Spark column using the given Schema Registry schema name and version.

Serialization - deserialization example

The following is an example that uses the from_sr to de-serialize Schema Registry formatted messages into Spark, transforms and serializes it back to Schema Registry format using to_sr, and writes to a data sink.

This example assumes Spark Structured Streaming use cases, but it should work well for the non-streaming use cases as well (read and write).

```
// Read schema registry formatted messages and deserialize to spark columns.
val df = messages
      .select(from_sr($"value", topic).alias("message"))
// project (driverId, truckId, miles) for the events where miles > 300
val filtered = df.select($"message.driverId", $"message.truckId", $"message.
miles")
      .where("message.miles > 300")
// write the output as schema registry serialized bytes to a sink
// should produce events like {"driverId":14,"truckId":25,"miles":373}
val query = filtered
      .select(to_sr(struct($"*"), outSchemaName).alias("value"))
      .writeStream
      .format(..)
      .start()
```

The output schema outSchemaName is automatically published to the Schema Registry if it does not exist.

Building and deploying your app

Add a Maven dependency in your project to make use of the library and build your application JAR file:

```
<dependency>
    <groupId>com.hortonworks</groupId>
    <artifactId>spark-schema-registry-for-spark3_2.12</artifactId>
    <version>version</version>
 </dependency>
```

Once the application JAR file is built, deploy it by adding the dependency in spark3-submit using --packages:

```
spark3-submit --packages com.hortonworks:spark-schema-registry-for-spark3_2.
12:version \
--conf spark.jars.repositories=[***HTTPS://REPOSITORY.EXAMPLE.COM***] \
--class YourApp \
your-application-jar \
args ...
```

Make sure the package is published in a local or online available repository.

If the package is not published to an available repository, or your Spark application cannot access external networks, you can use an uber JAR file instead:

```
spark3-submit --master [***MASTER URL****] \
--jars /opt/cloudera/parcels/SPARK3/lib/spark3/spark-schema-registry-for-s
park3/spark-schema-registry-for-spark3_2.12-jar-with-dependencies.jar \
```

```
--class YourApp \
your-application-jar \
args ...
```

Running in a Kerberos-enabled cluster

The library works in a Kerberos setup, where Spark and Schema Registry has been deployed on a Kerberos-enabled cluster.

To configure, set up the appropriate JAAS config for RegistryClient (and KafkaClient, if the Spark data source or sink is Kafka).

As an example, to run the SchemaRegistryAvroExample in a Kerberos setup, follow these steps:

1. Create a keytab (for example, app.keytab) with the login user and principal you want to run the application.
2. Create an app_jaas.conf file and specify the keytab and principal created in Step 1.

   If deploying to YARN, the keytab and conf files will be distributed as YARN local resources. They will be placed in the current directory of the Spark YARN container, and the location needs to be specified as ./app.keytab.

   ```
   RegistryClient {
       com.sun.security.auth.module.Krb5LoginModule required
       useKeyTab=true
       keyTab="./app.keytab"
       storeKey=true
       useTicketCache=false
       principal="[***PRINCIPAL***]";
   };

   KafkaClient {
      com.sun.security.auth.module.Krb5LoginModule required
      useKeyTab=true
      keyTab="./app.keytab"
      storeKey=true
      useTicketCache=false
      serviceName="kafka"
      principal="[***PRINCIPAL***]";
   };
   ```

3. Provide the required ACLs for the kafka topics (in-topic, out-topic) for the principal.
4. Use spark3-submit to pass the JAAS configuration file with extraJavaOptions. (And also as local resource files in YARN cluster mode.)

   ```
   spark3-submit --master yarn --deploy-mode cluster \
       --keytab app.keytab --principal [***PRINCIPAL***] \
       --files app_jaas.conf#app_jaas.conf,app.keytab#app.keytab \
       --jars /opt/cloudera/parcels/SPARK3/lib/spark3/spark-schema-registry-f
   or-spark3/spark-schema-registry-for-spark3_2.12-jar-with-dependencies.jar
    \
       --conf "spark.executor.extraJavaOptions=-Djava.security.auth.login.c
   onfig=./app_jaas.conf" \
       --conf "spark.driver.extraJavaOptions=-Djava.security.auth.login.co
   nfig=./app_jaas.conf" \
       --class com.hortonworks.spark.registry.examples.SchemaRegistryAvroEx
   ample \
   /opt/cloudera/parcels/SPARK3/lib/spark3/spark-schema-registry-for-spark3/
   examples/spark-schema-registry-for-spark3-examples_2.12.jar \
       [***SCHEMA-REGISTRY-URL***] bootstrap-server in-topic out-topic che
   ckpoint-dir SASL_PLAINTEXT
   ```

Unsupported features

Apache Spark 3 integration with Schema Registry is not supported in pyspark.

Cumulative hotfixes for CDS

You can review the list of cumulative hotfixes that were shipped for CDS.

Cumulative hotfix CDS 3.3.7190.2-1 (CDS 3.3 CHF1 for 7.1.9)

Know more about CDS 3.3 CHF1 for Cloudera Runtime 7.1.9. This cumulative hotfix was released on November 20, 2023.

Important: CDS 3.3 for 7.1.9 Powered by Apache Spark is an add-on service for CDP Private Cloud Base, and is only supported with Cloudera Runtime 7.1.9. Spark 2 is included in CDP, and does not require a separate parcel.

Contact Cloudera Support for questions related to any specific hotfixes.

Following is the list of fixes that were shipped for CDS 3.3.2.3.3.7190.2-1-1.p0.46867244:

- CDPD-61940: [CDS 3.3 for 719] Extend Java options for Spark and Livy to support JDK 17 with Isilon
- CDPD-61888: Livy 3 - Kerberos ticket renewal fails with JDK-8186576
- CDPD-61564: Spark - Caused by: java.lang.NoClassDefFoundError: org/datanucleus/store/query/cache/QueryCompilationCache
- CDPD-61240: Fix Spark Rapids issues
- CDPD-60501: Spark3 - Update pre_commit_hook to use JDK 8 version to u371
- CDPD-59644: [IBM-PPC] YARN node manager service fails to start after adding Spark3 on YARN

Table 2: CDS cumulative hotfix 3.3.7190.2-1 download URL

Parcel Repository Location
`https://[username]:[password]@archive.cloudera.com/p/spark3/3.3.7190.2/parcels/`

Table 3: Supported Versions

CDS Powered by Apache Spark Version	Dependent Stack Version	Supported CDP Versions
3.3.2.3.3.7190.2-1	Cloudera Runtime 7.1.9.2-10	CDP Private Cloud Base with Cloudera Runtime 7.1.9

Cumulative hotfix CDS 3.3.7190.3-1 (CDS 3.3 CHF2 for 7.1.9)

Know more about CDS 3.3 CHF2 for 7.1.9. This cumulative hotfix was released on December 13, 2023.

Important: CDS 3.3 for 7.1.9 Powered by Apache Spark is an add-on service for CDP Private Cloud Base, and is only supported with Cloudera Runtime 7.1.9. Spark 2 is included in CDP, and does not require a separate parcel.

Contact Cloudera Support for questions related to any specific hotfixes.

Following is the list of fixes that were shipped for CDS 3.3.2.3.3.7190.3-1-1.p0.48047943:

- CDPD-64135: [7.1.7 SP2, CDS 3.x] Backport HBASE-27624
- CDPD-63799: [7.1.9, CDS 3.x] Livy - Upgrade snakeyaml to 1.33 due to high CVEs
- CDPD-61742: Test failure: org.apache.spark.sql.hive.execution.HiveTableScanSuite.Spark-4077: timestamp query for null value

Table 4: CDS cumulative hotfix 3.3.7190.3-1 download URL

Parcel Repository Location
`https://[username]:[password]@archive.cloudera.com/p/spark3/3.3.7190.3/parcels/`

Table 5: Supported Versions

CDS Powered by Apache Spark Version	Dependent Stack Version	Supported CDP Version
3.3.2.3.3.7190.3-1	Cloudera Runtime 7.1.9.2 (7.1.9 CHF1)	CDP Private Cloud Base with Cloudera Runtime 7.1.9

Cumulative hotfix CDS 3.3.7190.4-1 (CDS 3.3 CHF3 for 7.1.9)

Know more about CDS 3.3 CHF3 for 7.1.9. This cumulative hotfix was released on March 11, 2024.

Important: CDS 3.3 for 7.1.9 Powered by Apache Spark is an add-on service for CDP Private Cloud Base, and is only supported with Cloudera Runtime 7.1.9. Spark 2 is included in CDP, and does not require a separate parcel.

Contact Cloudera Support for questions related to any specific hotfixes.

Following is the list of fixes that were shipped for CDS 3.3.2.3.3.7190.4-1-1.p0.51021169:

- CDPD-66981: [CDS 3.x] Broken lineage on Atlas when calling .cache() or .persist() method on a Spark DataFrame
- CDPD-66940: Timezone value not getting updated in Livy 3
- CDPD-66951: Livy3 server logs are missing on 7.1.9 FIPS clusters
- CDPD-65588: [CDS 3.x] Exclude log4j dependencies from spark-atlas-connector assembly
- CDPD-65584: [CDS 3.x] Spark - Upgraded Apache Derby to 10.17.1.0 due to CVE-2022-46337
- CDPD-65549: [CDS 3.3 for 7.1.9] Iceberg table does not show lineage in Spark for dataframe
- CDPD-63725: HWC - Unhadled VarcharType(100) in Pyspark3 shell
- CDPD-61021: Spark3: Add configuration for disabling fallback to saving in Spark specific format when saving as Hive table results in error
- CDPD-46689: HWC - DIRECT_READER_V2 must handle delete delta files from delete & update queries
- CDPD-44220: Fixed issues with Livy session recovery/HA failover on FIPS clusters

Table 6: CDS cumulative hotfix 3.3.7190.4-1 download URL

Parcel Repository Location
`https://[username]:[password]@archive.cloudera.com/p/spark3/3.3.7190.4/parcels/`

Table 7: Supported Versions

CDS Powered by Apache Spark Version	Dependent Stack Version	Supported CDP Version
3.3.2.3.3.7190.4-1	Cloudera Runtime 7.1.9.4 (7.1.9 CHF3)	CDP Private Cloud Base with Cloudera Runtime 7.1.9

Cumulative hotfix CDS 3.3.7190.5-2 (CDS 3.3 CHF4 for 7.1.9)

Know more about CDS 3.3 CHF4 for 7.1.9. This cumulative hotfix was released on June 17, 2024.

Important: CDS 3.3 for 7.1.9 Powered by Apache Spark is an add-on service for CDP Private Cloud Base, and is only supported with Cloudera Runtime 7.1.9. Spark 2 is included in CDP, and does not require a separate parcel.

Contact Cloudera Support for questions related to any specific hotfixes.

Following is the list of fixes that were shipped for CDS 3.3.2.3.3.7190.5-2-1.p0.54391297:

- CDPD-69879: HWC - java.lang.NoSuchFieldError: DEFAULT_MAX_WAIT
- CDPD-69805: Spark - OpenCSVSerde treats blank value as null
- CDPD-69760: Backport: [SPARK-46680][BUILD] Upgrade Apache commons-pool2 to 2.12.0
- CDPD-69324: Backport SPARK-46779 and SPARK-47955
- CDPD-67703: [CDS 3.3 for 7.1.9] SAC - Missing unknown leaf node: RelationV2
- CDPD-67561: Backport [SPARK-47319][SQL] Improve missingInput calculation
- CDPD-67338: Handle the ClassCastException of CDPD-40874 in the HWC layer
- CDPD-67336: Revert the Spark change done as part of CDPD-40874, to add Identifier field
- CDPD-64801: Livy - Upgrade datatables to 1.10.23+ due to CVE-2020-28458
- CDPD-61470: Spark - Do not call HMS to get list of pruned partitions when translated filter is empty
- CDPD-60979: Spark3 - Upgrade Apache Ivy to 2.5.2 due to CVE-2022-46751
- CDPD-60845: Unable to write data to the non-default database using HWC
- CDPD-58844: Spark - Upgrade Janino to 3.1.10 due to CVE-2023-33546
- CDPD-47129: Spark - Handle empty CSV fields via OpenCSVSerde

Table 8: CDS cumulative hotfix 3.3.7190.5-2 download URL

Parcel Repository Location
`https://[***USERNAME***]:[***PASSWORD***]@archive.cloudera.com/p/spark3/3.3.7190.5/parcels/`

Table 9: Supported Versions

CDS Powered by Apache Spark Version	Dependent Stack Version	Supported CDP Version
3.3.7190.5-2	Cloudera Runtime 7.1.9.14 (7.1.9 CHF7)	CDP Private Cloud Base with Cloudera Runtime 7.1.9

Cumulative hotfix CDS 3.3.7190.7-2 (CDS 3.3 CHF5 for 7.1.9)

Know more about CDS 3.3 CHF5 for 7.1.9. This cumulative hotfix was released on July 30, 2024.

Important: CDS 3.3 for 7.1.9 Powered by Apache Spark is an add-on service for CDP Private Cloud Base, and is only supported with Cloudera Runtime 7.1.9. Spark 2 is included in CDP, and does not require a separate parcel.

Contact Cloudera Support for questions related to any specific hotfixes.

Following is the list of fixes that were shipped for CDS 3.3.2.3.3.7190.7-2-1.p0.55847114:

- CDPD-72347: Backport SPARK-48946
- CDPD-71987: Spark3 UCX build fails due to Centos7 EOL
- CDPD-71817: Spark - Upgrade Aircompressor to 0.27 due to CVE-2024-36114
- CDPD-70400: Corruption of pre-epoch dates and timestamps over HWC operations

Table 10: CDS cumulative hotfix 3.3.7190.7-2 download URL

Parcel Repository Location
`https://[***USERNAME***]:[***PASSWORD***]@archive.cloudera.com/p/spark3/3.3.7190.7/parcels/`

Table 11: Supported Versions

CDS Powered by Apache Spark Version	Dependent Stack Version	Supported CDP Version
3.3.7190.7-2	Cloudera Runtime 7.1.9.14 (7.1.9 CHF7)	CDP Private Cloud Base with Cloudera Runtime 7.1.9

Cumulative hotfix CDS 3.3.7190.8-2 (CDS 3.3 CHF6 for 7.1.9)

Know more about CDS 3.3 CHF6 for 7.1.9. This cumulative hotfix was released on August 26, 2024.

Important: CDS 3.3 for 7.1.9 Powered by Apache Spark is an add-on service for CDP Private Cloud Base, and is only supported with Cloudera Runtime 7.1.9. Spark 2 is included in CDP, and does not require a separate parcel.

Contact Cloudera Support for questions related to any specific hotfixes.

Following is the list of fixes that were shipped for CDS 3.3.2.3.3.7190.8-2-1.p0.56763537:

- CDPD-73191: HWC - MatchError when querying a char/varchar partitioned table
- CDPD-73013: Backport SPARK-41585
- CDPD-72621: HWC - Support default constraints while writing into a table

Table 12: CDS cumulative hotfix 3.3.7190.8-2 download URL

Parcel Repository Location
`https://[***USERNAME***]:[***PASSWORD***]@archive.cloudera.com/p/spark3/3.3.7190.8/parcels/`

Table 13: Supported Versions

CDS Powered by Apache Spark Version	Dependent Stack Version	Supported CDP Version
3.3.7190.8-2	Cloudera Runtime 7.1.9.14 (7.1.9 CHF7)	CDP Private Cloud Base with Cloudera Runtime 7.1.9

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.

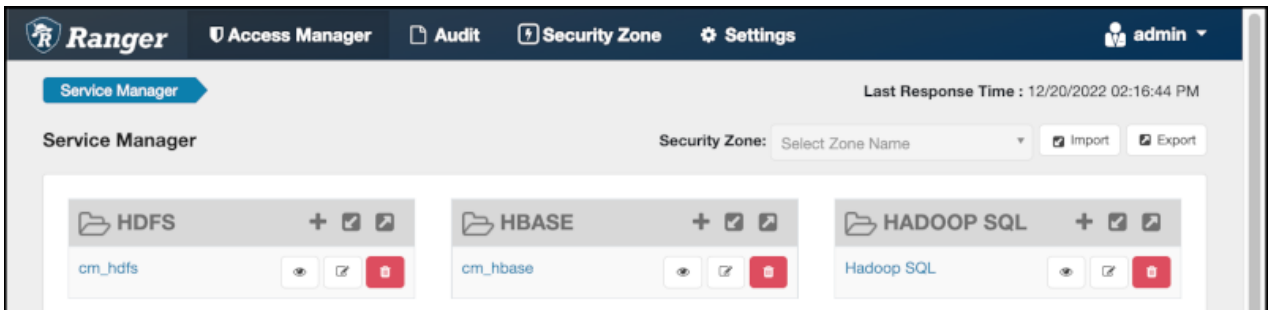
Note: Unless otherwise indicated, the operations in the subsequent documentation apply to both v1 and v2 formats.

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 on HDFS or S3" 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 HDFS or S3

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.

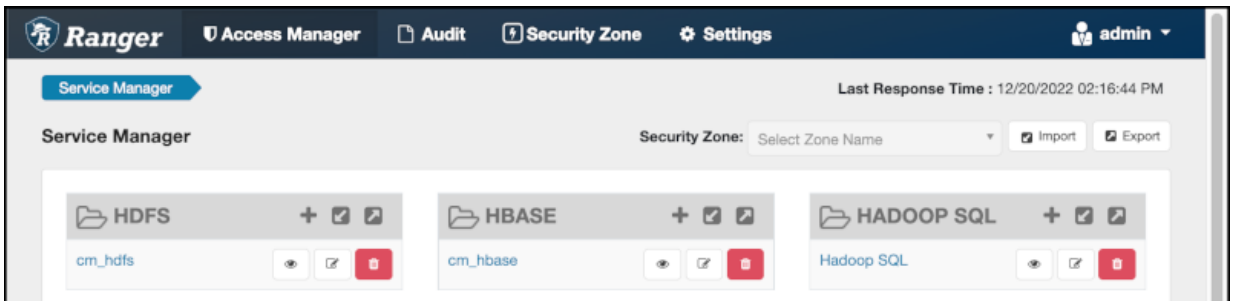
In this task, you specify Iceberg as the storage-type and allow the broadest access by setting the URL to *.

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.

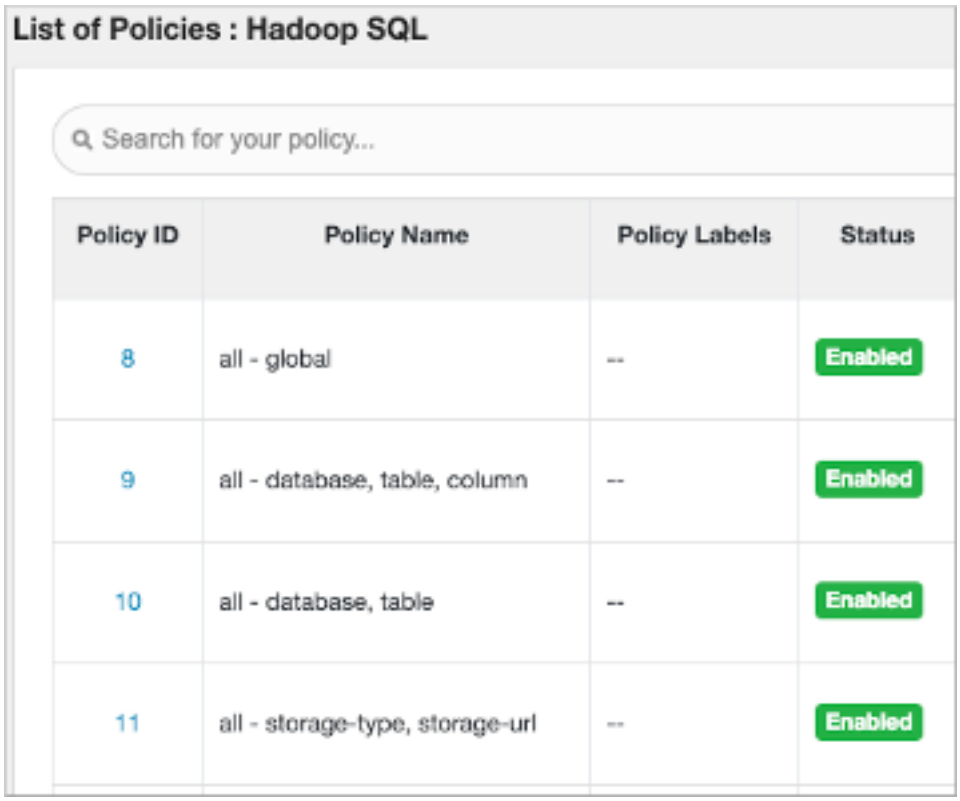
For more information about these policy settings, see Ranger Storage Handler documentation.

Procedure

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

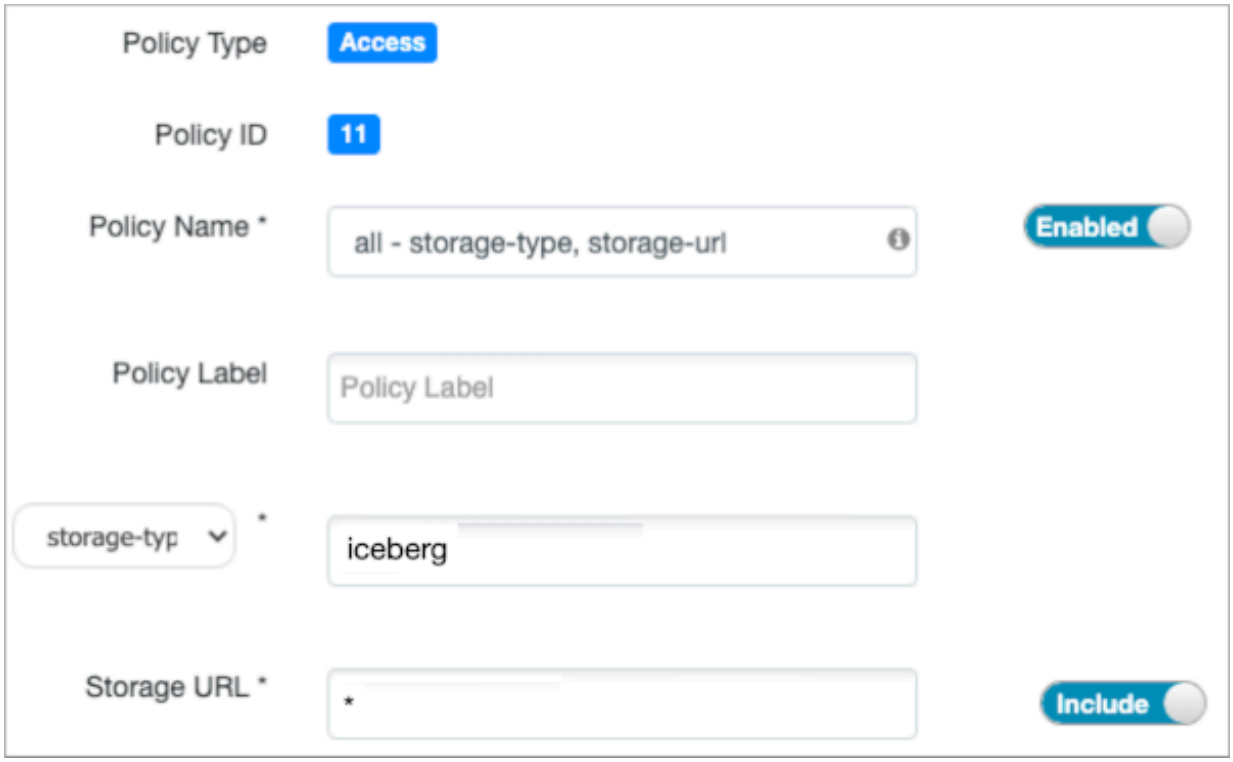


2. In Policy Name, enable the all - storage-type, storage-url policy.



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..

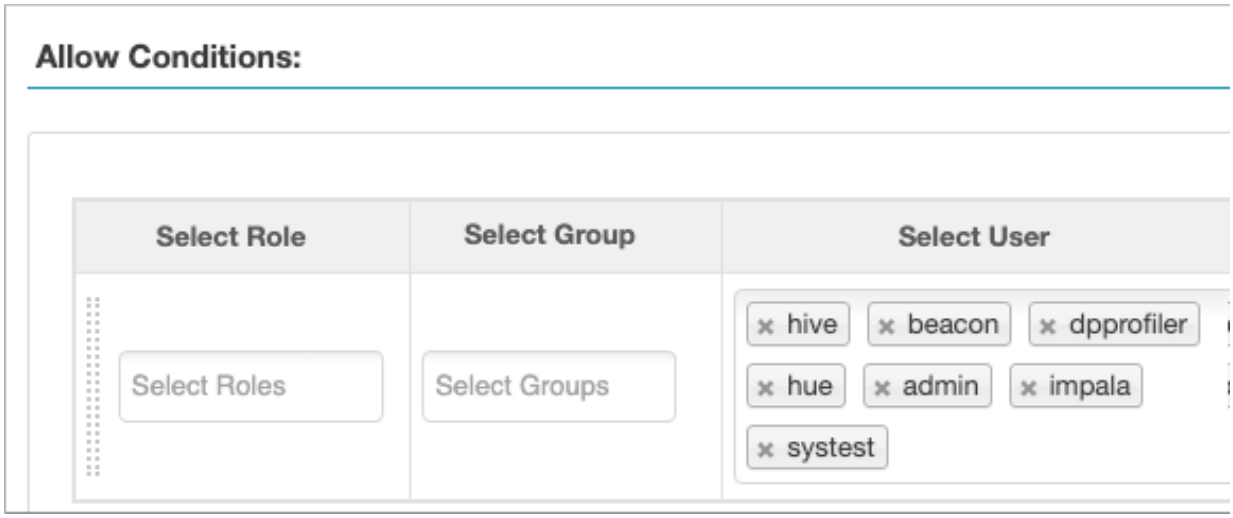
5. In Storage URL, enter the value *, enable Include.



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.

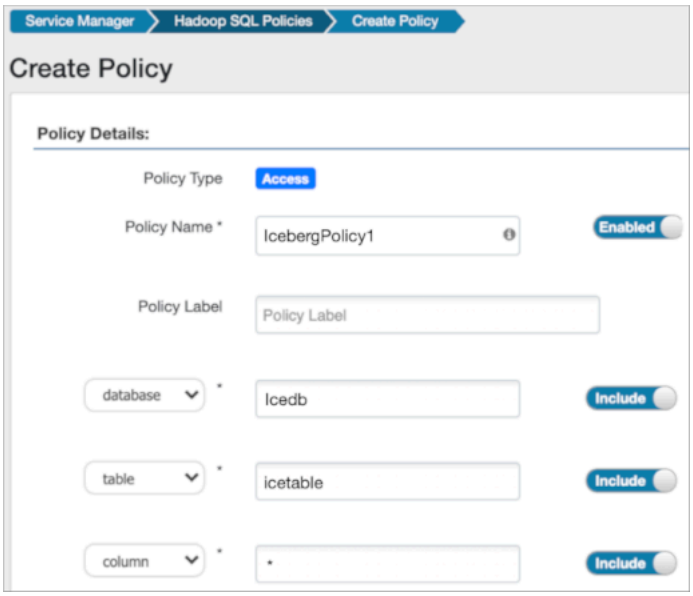
About this task

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.

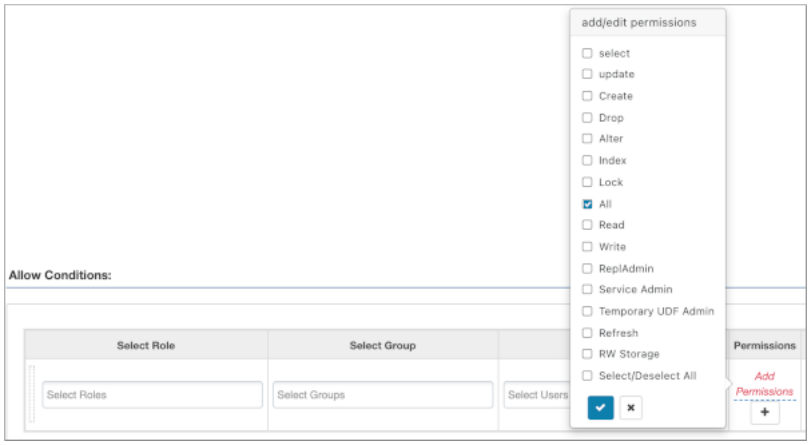
Procedure

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 CDE 1.19 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_catalog, 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.