Cloudera Runtime 7.2.11

# Securing Apache Hive

**Date published: 2019-08-21**
**Date modified: 2021-09-08**

**https://docs.cloudera.com/**

# Legal Notice

# Contents

# Hive access authorization

As administrator, you need to understand that the Hive default authorization for running Hive queries is insecure and what you need to do to secure your data. You need to set up Apache Ranger.

To limit Apache Hive access to approved users, Cloudera recommends and supports only Ranger. Authorization is the process that checks user permissions to perform select operations, such as creating, reading, and writing data, as well as editing table metadata. Apache Ranger provides centralized authorization for all Cloudera Runtime Services.

You can set up Ranger to protect managed, ACID tables or external tables using a Hadoop SQL policy. You can protect external table data on the file system by using an HDFS policy in Ranger.

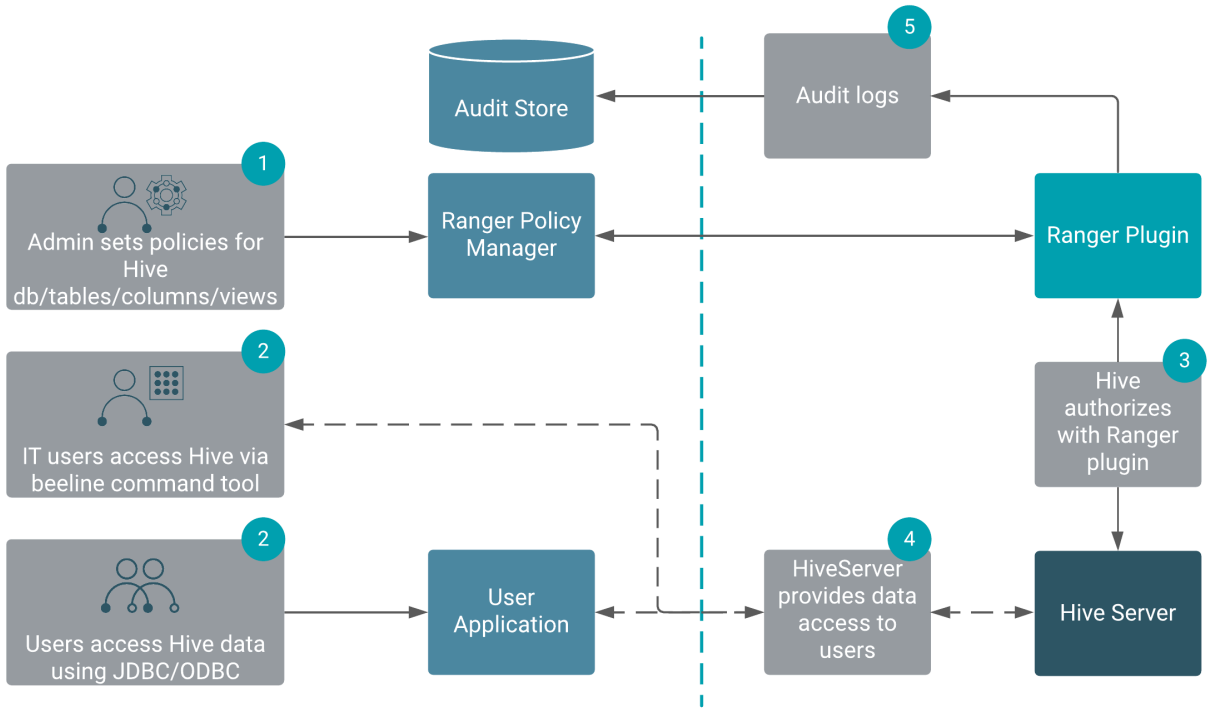
## Preloaded Ranger Policies

In Ranger, preloaded Hive policies are available by default. Users covered by these policies can perform Hive operations. All users need to use the default database, perform basic operations such as listing database names, and query the information schema. To provide this access, preloaded default database tables columns and information_ schema database policies are enabled for group public (all users). Keeping these policies enabled for group public is recommended. For example, if the default database tables columns policy is disabled preventing use of the default database, the following error appears:

```
hive> USE default;
Error: Error while compiling statement: FAILED: HiveAccessControlException
Permission denied: user [hive] does not have [USE] privilege on [default]
```

## Apache Ranger policy authorization

Apache Ranger provides centralized policy management for authorization and auditing of all Cloudera Runtime services, including Hive. All Cloudera Runtime services are installed with a Ranger plugin used to intercept authorization requests for that service, as shown in the following illustration.




The following table compares authorization models:

| Authorization model | Secure? | Fine-grained authorization (column, row level) | Privilege management using GRANT/REVOKE statements | Centralized management GUI |
|---|---|---|---|---|
| Apache Ranger | Secure | Yes | Yes | Yes |
| Hive default | Not secure. No restriction on which users can run GRANT statements | Yes | Yes | No |

When you run grant/revoke commands and Apache Ranger is enabled, a Ranger policy is created/removed.

# Spark and Hive permissions on Azure cloud cluster

This topic describes example scenarios about how Ranger authorizes queries coming from Spark to access Hive table.

**Important:** These scenarios are only examples. Environment, database name, table name, and custom user details will vary based on your cluster and requirements. Do not copy any code from this topic.

**Environment details**

Cloudera Data Platform: Azure cloud cluster with CDP 7.2.18.500

Cloudera Manager: CM 7.12.0.500

## Connecting to Spark and running Hive queries

**Connecting to Spark shell**

To connect to the Spark shell, run the following command:

```
spark3-shell
```

**Running Hive queries in Spark shell**

To run Hive queries, you need to call the HiveContext class:

```
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

**Executing a query**

```
sqlContext.sql("select * from customer2").show()
```

## Prerequisites for Ranger Hive security

In DataHub Hive services (HiveServer2 and Hive Metastore), the following configuration needs to be added in ranger-hive-security.xml and the services restarted:

```
ranger.plugin.hive.urlauth.filesystem.schemes = hdfs:,file:,wasb:,adl:,abfs:
```

## Initial setup with Hive user

Run the following commands using the Hive user:

```
CREATE EXTERNAL TABLE customer2 (c1 int, c2 STRING) ROW FORMAT DELIMITED FIE
LDS TERMINATED BY ',';
insert into customer2 values(1,'hivetest1');
select * from customer2;

create database vehicle;
```

```
create external table vehicle.cars(car_id int, car_name string, car_color
 string, car_price int);
insert into vehicle.cars(car_id, car_name, car_color, car_price) VALUES (1,
'car1','color1',100000), (2,'car2','color2',200000), (3,'car3','color3',3000
00), (4,'car4','color4',400000);
select * from vehicle.cars;
```

### Use cases for custom user permissions

**Usecase 1: Running a Select query with a custom user**

- Objective: To run a select query using custom user hrt_qa having policy on a table.
- Workflow:
  1. Created initial Hive policy:
     - Resource:
       - Hive Database = default
       - Hive Table = customer2
       - Hive Column = *
     - User:
       - hrt_qa = select
  2. Attempted command (as hrt_qa after kinit in Spark shell):

     ```
     sqlContext.sql("select * from customer2").show()
     ```
  3. Observed error:

     Denied exception for abfs path. For example,

     ```
     SASTokenProviderException: Failed to acquire a SAS token for
      get-status on /warehouse/tablespace/external/hive/customer2
      due to org.apache.hadoop.security.AccessControlException: P
     ermission denied
     ```

     This indicates a need for an ADLS policy.
  4. ADLS policy creation:
     - Storage account: sdxqeweekly
     - Storage account container: vb-18df-data
     - Relative path: /warehouse/tablespace/external/hive
     - User: hrt_qa = all
  5. Re-attempted command (as hrt_qa after kinit):

     ```
     sqlContext.sql("select * from customer2").show()
     ```
- Result: The command is successful, and table records are visible.

**Usecase 2: Creating a table in the database with a custom user**

- Objective: To allow custom user hrt_qa to create a table in the vehicle database.

- Workflow:
  1. Initial Hive policy creation:
     - Resource:
       - Hive Database = vehicle
       - Hive Table = *
       - Hive Column = *
     - User:
       - hrt_qa = create
  2. Attempted command (as hrt_qa after kinit in a Spark shell):

     ```
     sqlContext.sql("create table vehicle.cars1(car_id int, car_n
     ame string, car_color string, car_price int)").show()
     ```

  3. Observed error: Denied exception for Azure path. For example,

     ```
     SASTokenProviderException: Failed to acquire a SAS token for
      get-status on /warehouse/tablespace/external/hive/vehicle.d
     b/cars1 due to org.apache.hadoop.security.AccessControlExcep
     tion: Permission denied.
     ```

     This indicates a need for an ADLS policy.
  4. ADLS policy creation:
     - Storage Account: sdxqeweekly
     - Storage Account Container: vb-18df-data
     - Relative Path: /warehouse/tablespace/external/hive
     - User: hrt_qa = all
  5. Re-attempted command (as hrt_qa after kinit in a Spark shell):

     ```
     sqlContext.sql("create table vehicle.cars1(car_id int, car_n
     ame string, car_color string, car_price int)").show()
     ```

- Result: Command was successful, and the table was created.

**Usecase 3: Altering a table in the database with a custom user**

- Objective: To allow a custom user hrt_qa to alter the cars table in the vehicle database.

- Workflow:
  1. Initial Hive policy creation:
     - Resource:
       - Hive Database = vehicle
       - Hive Table = cars
       - Hive Column = *
     - User: hrt_qa = alter
  2. Attempted command (as hrt_qa after kinit in a Spark shell):

     ```
     sqlContext.sql("alter table vehicle.cars rename to vehicle.c
     ars_a").show()
     ```
  3. Observed error: Permission denied: user [hrt_qa] does not have [SELECT] privilege on [vehicle/cars].

     This indicates that you need to give SELECT permission to the user.
  4. Updated Hive policy (adding select):
     - Resource:
       - Hive Database = vehicle
       - Hive Table = cars
       - Hive Column = *
     - User: hrt_qa = alter, select
  5. Re-attempted command (as hrt_qa after kinit in a Spark shell):

     ```
     sqlContext.sql("alter table vehicle.cars rename to vehicle.c
     ars_a").show()
     ```
  6. Observed Error: Unable to alter table. Permission denied: user [hrt_qa] does not have [ALTER] privilege on [vehicle/cars_a].

     This indicates that the new table name also needs ALTER permission.
  7. Further updated Hive policy (adding new table name):
     - Resource:
       - Hive Database = vehicle
       - Hive Table = cars, cars_a
       - Hive Column = *
     - User: hrt_qa = alter, select
  8. Re-attempted command (as hrt_qa after kinit in a Spark shell):

     ```
     sqlContext.sql("alter table vehicle.cars rename to vehicle.c
     ars_a").show()
     ```
  9. Deny log generated: Command was successful, and the table was renamed. However, Ranger UI audit access logs showed a deny log for the Azure directory.

     This indicates that you need to perform the alter operation with ADLS policy to avoid the above denied log.
  10. Addressing Azure directory deny log (with ADLS policy):
      - Hive policy:
        - Resource:
          - Hive Database = vehicle
          - Hive Table = cars_a, cars_b

        - Hive Column = *
      - User: hrt_qa = alter, select
    - ADLS policy:
      - Storage Account: sdxqeweekly
      - Storage Account Container: vb-18df-data
      - Relative Path: /warehouse/tablespace/external/hive
      - User: hrt_qa = all

    **11.** Final command (as hrt_qa after kinit in a Spark shell):

```
sqlContext.sql("alter table vehicle.cars_a rename to vehicle
.cars_b").show()
```

- Result: Command was successful, the table got renamed successfully, and no deny logs for the Azure directory were seen.

**Usecase 4: Dropping a table with a custom user**

- Objective: To allow custom user hrt_qa to run a drop table query.
- Workflow:

  **1.** Initial setup (as Hive user):

  **a.** Create database:

```
create database sports;
```

  **b.** Create external table:

```
create external table sports.cricket(id int, name string);
```

  **c.** Insert data:

```
insert into sports.cricket values (11, 'name1');
```

  **d.** Verify data:

```
select * from sports.cricket;
```

  **2.** Initial Hive policy:

  - Resource:
    - Hive Database = sports
    - Hive Table = cricket
    - Hive Column = *
  - User: hrt_qa = drop

  **3.** Attempt to drop table (as hrt_qa after kinit in a Spark shell):

```
sqlContext.sql("drop table sports.cricket").show()
```

  **4.** Observed Error:

  The command failed due to a missing SELECT permission:

```
org.apache.spark.sql.AnalysisException: org.apache.hadoop.hi
ve.ql.metadata.HiveException: Unable to fetch table cricket.
```

```
 Permission denied: user [hrt_qa] does not have [SELECT] pri
vilege on [sports/cricket]
```

This indicates that you need to give SELECT permission to the user.

5. Updated Hive policy:
   - Resource:
     - Hive Database = sports
     - Hive Table = cricket
     - Hive Column = *
   - User: hrt_qa = drop, select
6. Re-attempted command to drop table (as hrt_qa after kinit in a Spark shell):

```
kinit as hrt_qa
sqlContext.sql("drop table sports.cricket").show()
```

- Result: The command executed successfully, and the table was dropped.

**Usecase 5: Inserting values into a table with a custom user**

- Objective: Inserting values into a table with custom user hrt_qa.

- Workflow:
  1. Initial setup (as Hive user):
     a. Create database:

     ```
     create database sports;
     ```

     b. Create external table:

     ```
     create external table sports.football(id int, name strin
     g);
     ```

     c. Verify data:

     ```
     select * from sports.football;
     ```

  2. Initial Hive policy:
     - Resource:
       - Hive Database = sports
       - Hive Table = football
       - Hive Column = *
     - User: hrt_qa = update
  3. Attempt to insert data (as hrt_qa after kinit in a Spark shell):

     ```
     sqlContext.sql("insert into sports.football values (11, 'nam
     e1')").show()
     ```

  4. Observed error (after first attempt):

     The command failed due to a missing SELECT permission:

     ```
     Caused by: org.apache.hadoop.hive.metastore.api.MetaExceptio
     n: Permission denied: user [hrt_qa] does not have [SELECT] p
     rivilege on [sports/football]
     ```

     This indicates that the custom user needs SELECT permission for insertion.
  5. Updated Hive policy:
     - Resource:
       - Hive Database = sports
       - Hive Table = football
       - Hive Column = *
     - User: hrt_qa = update, select
  6. Re-attempted command to insert data (as hrt_qa after kinit in a Spark shell):

     ```
     sqlContext.sql("insert into sports.football values (11, 'nam
     e1')").show()
     ```

  7. Observed error (after second attempt):

     The command failed due to an Azure directory permission issue:

     ```
     org.apache.spark.SparkRuntimeException: Cannot create stagin
     g directory: 'Some(abfs://vb-18df-data@sdxqeweekly.dfs.core.
     windows.net/warehouse/tablespace/external/hive/sports.db/foo
     tball/.hive-staging_hive_ 2024-12-18 _10-12-47_030_641659355
     0262289528-1)': Failed to acquire a SAS token for create-dir
     ectory on /warehouse/tablespace/external/hive/sports.db/foot
     ball/.hive-staging_hive_ 2024-12-18 _10-12-47_030_6416593550
     ```

```
262289528-1 due to org.apache.hadoop.security.AccessControlE
xception: Permission denied.
```

This indicates that an ADLS (Azure Data Lake Storage) policy is required for the custom user to manage the staging directory.

8. Added ADLS policy:
   - Existing Hive Policy:
     - Resource:
       - Hive Database = sports
       - Hive Table = football
       - Hive Column = *
     - User: hrt_qa = update, select
   - ADLS Policy:
     - Storage Account = sdxqeweekly
     - Storage Account Container = vb-18df-data
     - Relative Path = /warehouse/tablespace/external/hive
     - User: hrt_qa = all
9. Re-attempt command to insert data (as hrt_qa after kinit in a Spark shell after adding ADLS policy):

```
sqlContext.sql("insert into sports.football values (11, 'nam
e1')").show()
```

10. Observed error (after third attempt):

    The command failed due to a missing ALTER permission, even though the data was observed to be inserted:

```
org.apache.spark.sql.AnalysisException: Unable to alter tabl
e. Permission denied: user [hrt_qa] does not have [ALTER] pr
ivilege on [sports/football]
```

    This indicates that the custom user also requires ALTER permission.

11. Final updated Hive policy:
    - Existing Hive policy:
      - Resource:
        - Hive Database = sports
        - Hive Table = football
        - Hive Column = *
      - User: hrt_qa = update, select, alter
    - ADLS policy:
      - Storage Account = sdxqeweekly
      - Storage Account Container = vb-18df-data
      - Relative Path = /warehouse/tablespace/external/hive
      - User: hrt_qa = all
12. Final re-attempted command to insert data (as hrt_qa after kinit in a Spark shell):

```
kinit as hrt_qa
sqlContext.sql("insert into sports.football values (11, 'nam
e1')").show()
```

- Result: The command executed successfully, and two entries were observed in the table.

# Transactional table access

As administrator, you must enable the Apache Ranger service to authorize users who want to work with transactional tables. These types of tables are the default, ACID-compliant tables in Hive 3 and later.

ACID tables reside by default in /warehouse/tablespace/managed/hive. Only the Hive service can own and interact with files in this directory. Ranger is the only available authorization mechanism that Cloudera recommends for ACID tables.

# External table access

As administrator, you must set up Apache Ranger to allow users to access external tables.

External tables reside by default in /warehouse/tablespace/external on your object store. To specify some other location of the external table, you need to include the specification in the table creation statement as shown in the following example:

```
CREATE EXTERNAL TABLE my_external_table (a string, b string)
LOCATION '/users/andrena';
```

Hive assigns a default permission of 777 to the hive user, sets a umask to restrict subdirectories, and provides a default ACL to give Hive read and write access to all subdirectories. External tables must be secured using Ranger.

# HWC authorization

The way you configure Hive Warehouse Connector (HWC) affects the query authorization process and your security. There are a number of ways to access Hive through HWC, and not all operations go through HiveServer (HS2). Some operations, such as Spark Direct Reader and Hive Streaming, go to Hive directly through HMS where storage-based permissions generally apply.

As a client user, you must be logged in using kerberos before using HWC. You need appropriate storage permissions to write to destination partition or table location. You need to configure an HWC read option. HWC read configuration options are shown in the following table:

**Table 1:**

| Capabilities | JDBC mode | Spark Direct Reader mode | Secure Access mode |
|---|---|---|---|
| Ranger integration with fine-grained access control | # | N/A | # |
| Hive ACID reads | # | # | # |
| Workloads handled | Non-production workloads, small datasets | Production workloads, ETL without fine-grained access control | Large workloads with fine-grained access control, row filtering, and column masking |

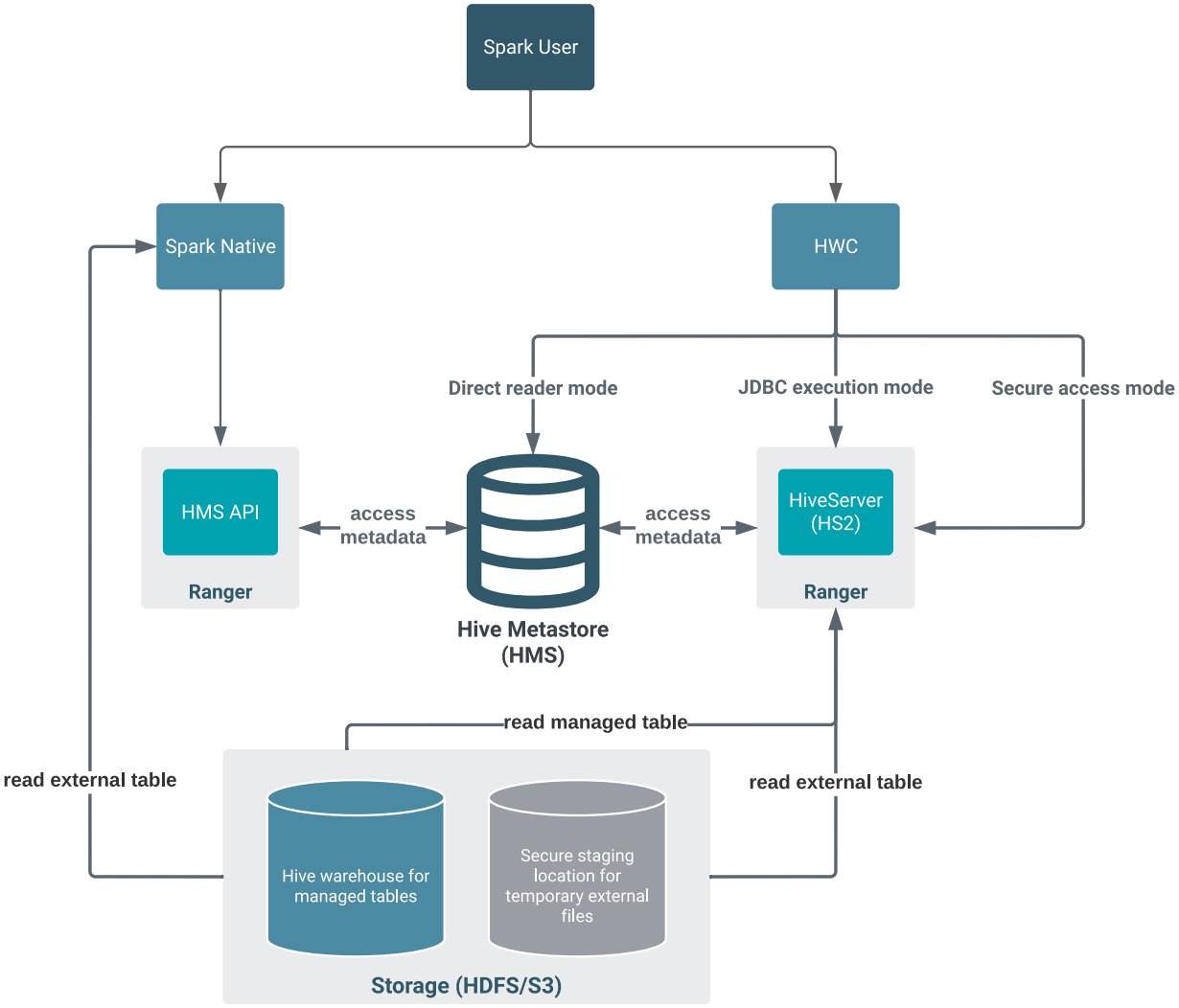
These read configuration options require connections to different Hive components:

- Direct Reader configuration: Connects to Hive Metastore (HMS)
- JDBC configuration: Connects to HiveServer (HS2)
- Secure Access configuration: Connects to HiveServer (HS2)

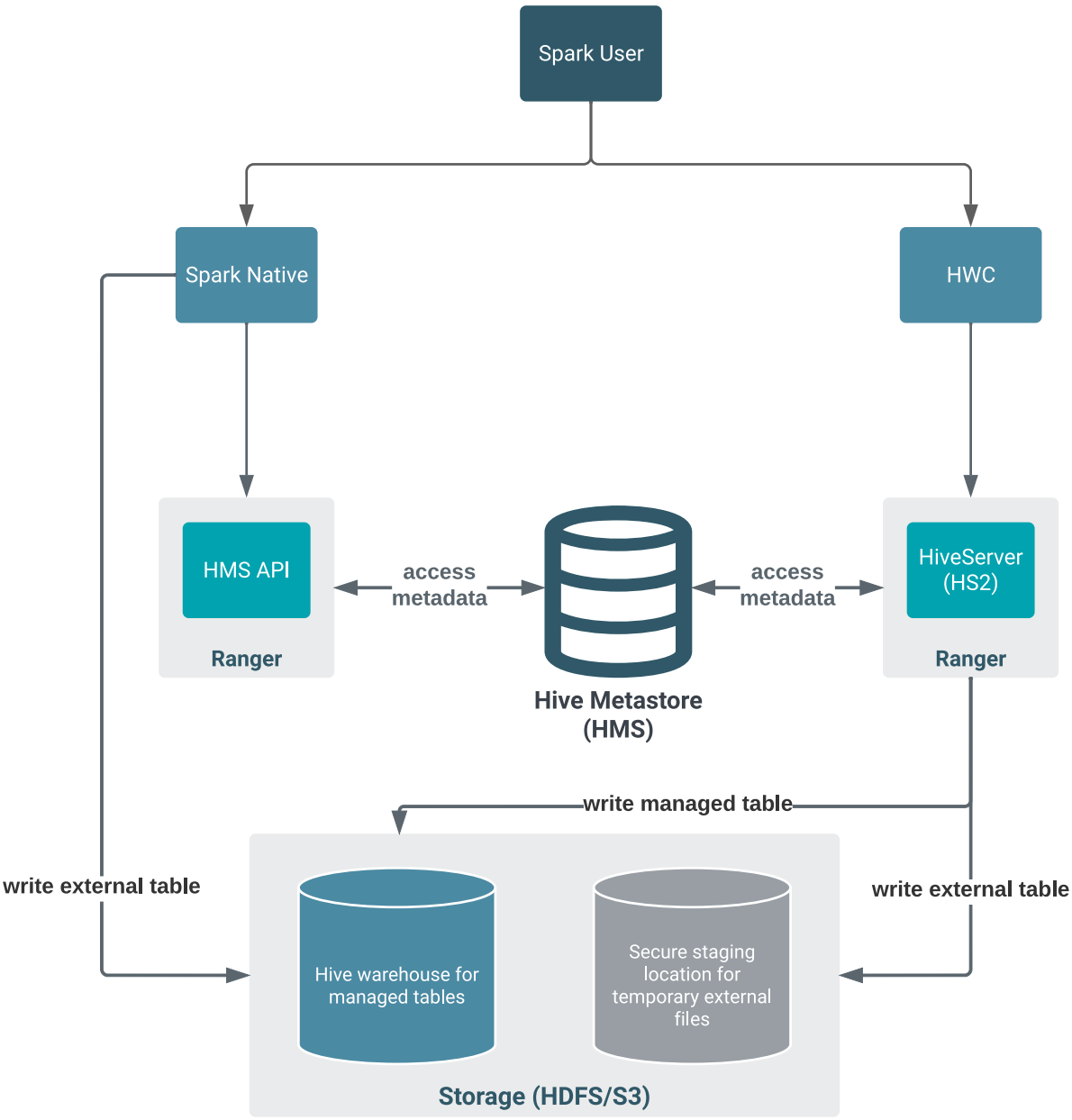
Ranger authorizes access to Hive tables from Spark through HiveServer (HS2) or the Hive metastore API (HMS API).

To write ACID managed tables from Spark to Hive, you must use HWC. To write external tables from Spark to Hive, you can use native Spark or HWC.

The following diagram shows the typical read authorization process:




The following diagram shows the typical write authorization process:




When writing, HWC always enforces authorization through HiveServer (HS2). Reading managed tables in JDBC mode or Secure access mode enforces Ranger authorization, including fine-grained features, such as row masking and column mapping. In Direct Reader mode, the Ranger and HMS integration provides authorization.

External table queries go through the HMS API, which is also integrated with Ranger. If you do not use HWC, the Hive metastore (HMS) API, integrated with Ranger, authorizes external table access. HMS API-Ranger integration enforces the Ranger Hive ACL in this case. When you use HWC, queries such as DROP TABLE affect file system data as well as metadata in HMS.

Using the Direct Reader option, SparkSQL queries read managed table metadata directly from the HMS, but only if you have permission to access files on the file system. You cannot write to managed tables using the Direct Reader option.

Using the Secure access mode, you can enable fine-grained access control (FGAC) column masking and row filtering to secure managed (ACID), or even external, Hive table data that you read from Spark.

Managed table authorization

A Spark job impersonates the end user when attempting to access an Apache Hive managed table. As an end user, you do not have permission to access, managed files in the Hive warehouse. Managed tables have default file system permissions that disallow end user access, including Spark user access.

As Administrator, you set permissions in Ranger to access the managed tables when you configure HWC for JDBC mode or Secure access mode reads. You can fine-tune Ranger to protect specific data. For example, you can mask data in certain columns, or set up tag-based access control.

When you configure HWC for Direct Reader mode, you cannot use Ranger in this way. You must set read access to the file system location for managed tables. You must have Read and Execute permissions on the Hive warehouse location (hive.metastore.warehouse.dir).

External table authorization

Ranger authorization of external table reads and writes is supported. You need to configure a few properties in Cloudera Manager for authorization of external table writes. You must be granted file system permissions on external table files to allow Spark direct access to the actual table data instead of just the table metadata.

### Direct Reader Authorization Limitation

As Spark allows users to run arbitrary code, Ranger fine grained access control, such as row level filtering or column level masking, is not possible within Spark itself. This limitation extends to data read using Direct Reader.

To restrict data access at a fine-grained level, use a read option that supports Ranger. Only consider using the Direct Reader option to read Hive data from Spark if you do not require fine-grained access. For example, use Direct Reader for ETL use cases.

# Configuring ZooKeeper SASL authentication for Hive clients

Learn about how to configure Hive clients to authenticate with a ZooKeeper ensemble that enforces Simple Authentication and Security Layer (SASL).

### ZooKeeper client authentication

When a ZooKeeper ensemble is configured to enforce SASL authentication, it ensures that only authenticated clients can establish a session. Without this enforcement, the ZooKeeper server remains open to anonymous client connections.

In Kerberized environments, Hive components such as HiveServer2 (HS2) and the Metastore use ZooKeeper for service discovery, high availability, and locking. While server-side components like HS2 can easily authenticate using a predefined principal and keytab, client-side discovery (such as JDBC or Metastore clients) requires specific configurations to handle different Kerberos login methods, including keytabs or the local ticket cache.

### Hive ZooKeeper authentication properties

The following table describes the properties used to manage ZooKeeper authentication for Hive components:

| Property | Description |
|---|---|
| hive.zookeeper.kerberos.enabled | Specifies whether hive beeline/JDBC HS2 uses Kerberos to secure the ZooKeeper connection. This property is evaluated as true only when HS2 is Kerberized. |

### Configuring Hive clients for SASL-enforced ZooKeeper

To enable successful authentication when a client connects to a SASL-enforced ZooKeeper ensemble, perform the following:

1. Authenticate by using `kinit` if you are using the ticket cache before running the Hive client.