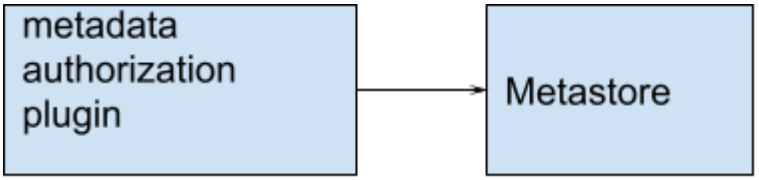
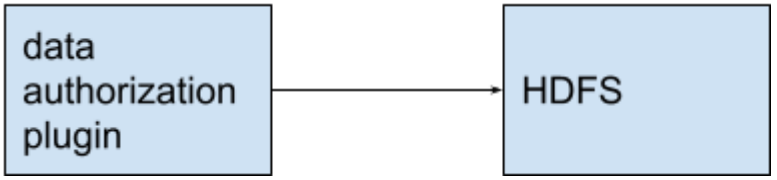
The Support of Relational Chained Authorization Plugin

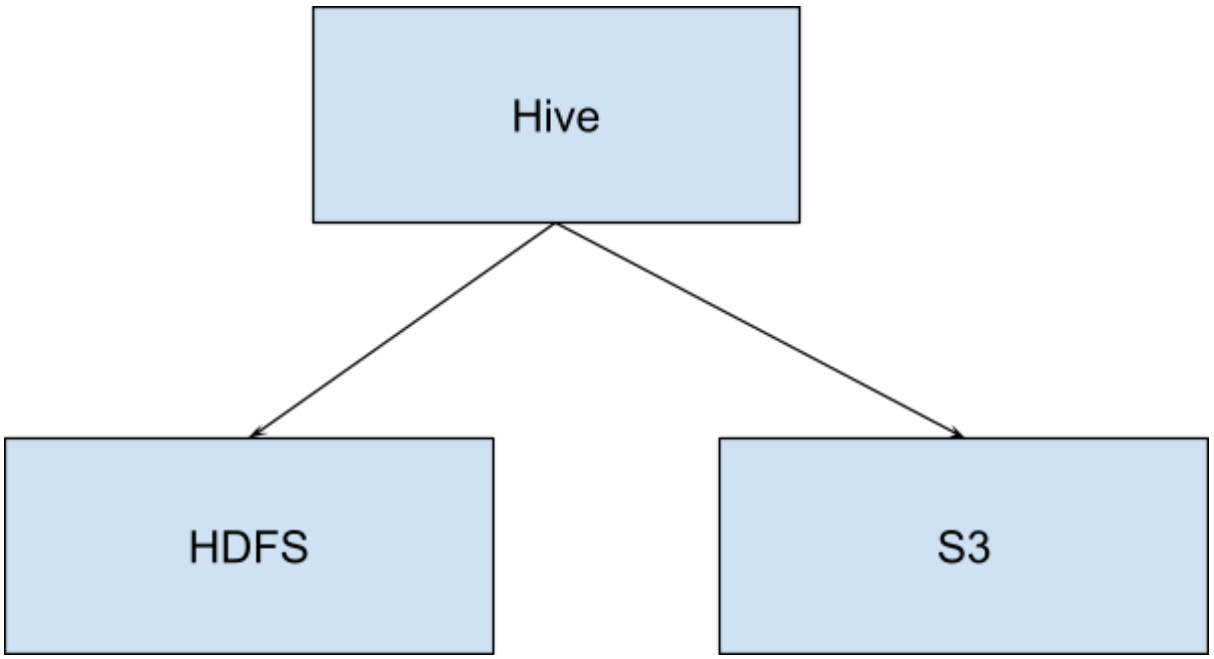
Background

Currently, Ranger plugin for Hive catalogs only restrict access to the metadata in the Hive. It doesn't forbid access to data. For the big data ecosystem, the meta data and data is divided into different systems, sometimes. For example, Hive's metadata is stored in the metastore. The data is stored by HDFS. We should give proper privileges to both of them.
For Ranger, Hive and HDFS will have different Ranger services. We should have a wrapper for them. When privileges are granted to it, we modify them at the same time.





For Hive, a Hive may use different storage cluster paths. Because a cluster, database, table, or partition can have its location.



Goals

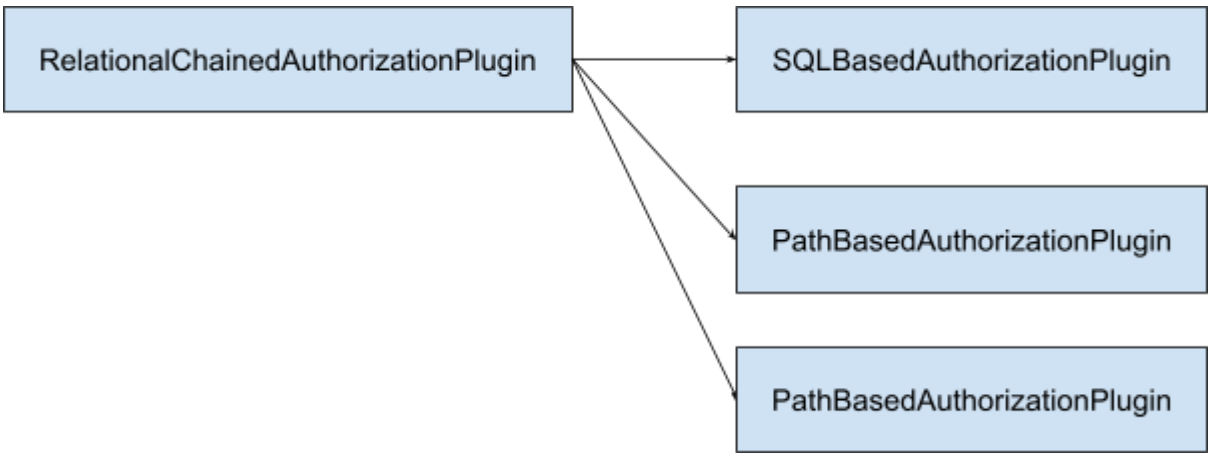
1. Provide a framework to sync the privileges from SQL-based authorization plugin to path-based authorization plugin to guarantee the data protection and metadata protection
2. One SQL-based plugin can work with multiple path-based plugins.
3. Provide the privilege mapping between the gravitino privilege model and plugin privilege model
4. Provide the interface to get the properties that the authorization plugin needs like location

Not Goals

1. Not provide concrete SQL-based plugins or path-based plugins
2. Not considering the location of partitions

Solution

For the metadata, we can use SQLBasedAuthorizationPlugin.
For the data, we can use PathBasedAuthorizationPlugin
The RelaitonalChainedAuthorizationPlugin will call the specific PathBasedAuthorizationPlugin according to the location.



Privilege Mapping

The privilege model in Gravitino is different from SQL-based authorization plugin and path-based authorization plugin. A privilege can be translated into different privileges. One securable object can be translated into different resources for different authorization plugins.

Securable Object Type	privilege	Hive privilege	Hive resource	HDFS privilege	HDFS resource
CATALOG	CREATE TABLE	CREATE	*.*	WRITE	${hive_location}/*/*
CATALOG	CREATE SCHEMA	CREATE	*	WRITE	${hive_location}/*
CATALOG	MODIFY TABLE	UPDATE/ ALTER/W RITE	*.* and *.*.*	WRITE	${hive_location}/*/*/*
CATALOG	SELECT TABLE	SELECT/ READ	*.* and *.*.*	READ	${hive_location}/*/*/*
CATALOG	USE CATALOG	SELECT	*	READ	${hive_location}/*
CATALOG	USE SCHEMA	SELECT	*	READ	${hive_location}/*
SCHEMA	CREATE TABLE	CREATE	{schema}.*	WRITE	${schema_location}/*
SCHEMA	MODIFY TABLE	UPDATE/ ALTER/W RIETE	{schema}.* and {schema}.*.*	WRITE	${schema_location}/*/*

SCHEMA	USE SCHEMA	SELECT	{schema}	READ	${schema_location}
SCHEMA	SELECT TABLE	SELECT/READ	{schema}.* and {schema}.*.*	READ	${schema_location}/*/*
TABLE	MODIFY TABLE	ALTER/UPDATE/WRITE	{table} and {table}.*	WRITE	${table_location}/*
TABLE	SELECT TABLE	SELECT	{table} and {table}.*	READ	${table_location}/*

The Mapping Relation

Option 1:

We configure the mapping relations in catalog properties, For example:

```
authorization-provider=relational-chained
authorization-plugins=hadoop-sql,hdfs1,hdfs2
authorization.hadoop-sql.type=sql-based
authorization.hadoop-sql.provider=ranger
authorization.hadoop-sql.ranger.admin.url=172.0.0.100:6080
authorization.hadoop-sql.ranger.auth.type=simple
authorization.hadoop-sql.ranger.username=Jack
authorization.hadoop-sql.ranger.password=PWD123
authorization.hadoop-sql.ranger.service.name=hiveRepo
authorization.hdfs1.type=path-based
authorization.hdfs1.provider=ranger
authorization.hdfs1.location=hdfs://xxxx:6000
authorization.hdfs1.ranger.admin.url=172.0.0.100:6080
authorization.hdfs1.ranger.auth.type=simple
authorization.hdfs1.ranger.username=Jack
authorization.hdfs1.ranger.password=PWD123
authorization.hdfs1.ranger.service.name=hdfs1Repo
authorization.hdfs2.type=path-based
authorization.hdfs2.provider=ranger
authorization.hdfs2.location=hdfs://xxxx:9000
authorization.hdfs2.ranger.admin.url=172.0.0.100:6080
authorization.hdfs2.ranger.auth.type=simple
authorization.hdfs2.ranger.username=Jack
```

```
authorization.hdfs2.ranger.password=PWD123
authorization.hdfs2.ranger.service.name=hdfs2Repo
```

Option 2:
We set some plugin authorization plugin references. The server can find other plugins according to the location.
We set properties to the Hive catalogs.

```
authorization-provider=relational-chained
authorization-plugins=hadoop-sql,hdfs1,hdfs2
authorization.hadoop-sql.type=sql-based
authorization.hadoop-sql.provider=ranger
authorization.hadoop-sql.ranger.admin.url=172.0.0.100:6080
authorization.hadoop-sql.ranger.auth.type=simple
authorization.hadoop-sql.ranger.username=Jack
authorization.hadoop-sql.ranger.password=PWD123
authorization.hadoop-sql.ranger.service.name=hiveRepo
authorization.hdfs1.location=hdfs://xxxx:6000
authorization.hdfs1.type=path-based
authorization.hdfs2.location=hdfs://xxxx:6000
authorization.hdfs2.type=path-based
```

Server can find a fileset catalog properties according to location

```
authorization.location=hdfs://xxxx:6000
authorization.ranger.admin.url=172.0.0.100:6080
authorization.ranger.auth.type=simple
authorization.ranger.username=Jack
authorization.ranger.password=PWD123
authorization.ranger.service.name=hdfs1Repo
```

Managed Table VS External Table

Hive manages managed tables. We can't grant a specific privilege to a specific file. But external tables could be granted a specific privilege to a specific file.
We could develop a tool to differentiate the privileges between Gravitino and Ranger.

Interface Design

```
/* This class is used for chaining the plugins. Every plugin will
convert the securable object to its own resource and privileges. */
public abstract class ChainedAuthorizationPlugin extends
```

```
AuthorizationPlugin {

List<AuthorizationPlugin> plugins();


@Override
public Boolean onMetadataUpdated(MetadataObjectChange... changes) throws
RuntimeException {
   for (AuthorizationPlugin plugin : plugins()) {
       plugin.onMetadataUpdated(changes);
   }
   return true;
}

AuthorizationPluginException {
   catalog.loadTable();
    table.properties(“location”);
    ExtendedSecurableObject =
   role = new Role (extendedSecurableObject)
   plugin.onRoleCreated(role);

@Override
public Boolean onRoleCreated(Role role, Catalog catalog) throws ;
   for (AuthorizationPlugin plugin : plugins()) {
       plugin.onRoleCreated(role, catalog);
   }
   return true;
}

@Override
public Boolean onRoleAcquired(Role role) throws
AuthorizationPluginException {
   for (AuthorizationPlugin plugin : plugins()) {
       plugin.onRoleAcquired(role);
   }
   return true;
}
...
}
```

```
/* AuthorizationPlugins have different expressions from securable objects. */
public interface translateSecurableObject {
   List<AuthorizationSecurableObject>
convertToAuthorizationObject(SecurableObject object);
}
```

```
/* It’s flexible if we use String*/
public interface AuthorizationSecurableObject {
   Resource resource();

   List<AuthorizationPrivilege> privileges();
}
```

interface Resouce {
   String name();
   void validate();
}

```
interface AuthorizationPrivilege {
   String name();

   Privilege.Condition condition();
}
```

```
public class PathBasedAuthorizationPlugin {
.          List<AuthorizationSecurableObject>
convertToAuthorizationObject(ExtendedSecurableObject object) {
     // If we want to get the HDFS from a file.
     // ExtendedSecurable object = ...
     // String location = object.properties().get("location")
     // If a location matches return new
AuthorizationSecurableObject(location);
     // else return null

  }

}
```

Add a new method for BaseCatalog to acquire the location of the schema, fileset, table

```
class BaseCatalog {
      ...
     // Object could be a schema,fileset or a table
     Map<String, String> extractProperties(Object object);
     ...
}
```

Future Grant

If there is a new schema/table/fileset location, if we ever created a role which can operate all the locations in the catalog or schema, we should grant the privileges of the new location to the role.

Reference

1. SQL Based Authorization Plugin
2. The Support Of Relational Chained Authorization Plugin
3. Gravitino access control 0.8