Background

Apache Gravitino(incubating) is a technical data catalog that uses a unified metadata paradigm to manage multiple data sources while still allowing multiple engines like Spark, Trino, and Flink, or Python to connect to these data sources for data processing through Gravitino.

Because each underlying data source will have its own permission system, it can be difficult to plug in data engines with the intent of querying multiple of these sources at once. This is especially important for data governance practitioners who have to worry about data access restrictions and data compliance issues, but this is streamlined through Gravitino. Therefore, in the hopes of solving this big data issue, Gravitino plans to implement a universal set of permissions models and paradigms. With this, users will be able to manage all of their data sources in a single access plane, regardless of whether the data source is a database, or a message queue or an object storage system.

After authorizing these data sources within Gravitino's metadata lake, authentication can then be performed in Spark, Trino, and Flink Engines, as well as our Python client. This abstraction allows users to control access to data and make compliant use of the data without having to obstruct other teams and worrying about the tedious work of individual permission systems.

Key Concepts

Authorization

Authorization is the process of giving someone the ability to access a resource.
Simply put, authorization assigns a specified user or group the means to perform a specific operation on the object.

Authentication

Authentication is the process that an individual, application, or service goes through to prove their identity before gaining access to digital systems.
As such, authentication is used to determine whether a specified user or group can perform a particular method of operation on the object.

Privilege

Privilege is a specific operation method for resources, if you need to control fine-grained privileges on resources in the system, then you need to design many different Privileges, however too many Privileges will cause too complicated settings in the authorization.

If you only need to carry out coarse-grained privilege control on the resources in the system, then you only need to design a small number of Privileges, but it will result in too weak control ability when the authentication. Therefore, the design of Privilege is an important trade-off in the permission system.We know that Privilege is generally divided into two types, one is the management category of Privilege, such as the CREATE, DELETE resource privilege, and the other is the operation category of Privilege, such as the READ and WRITE resource privilege.

In most organizations, the number of data managers is much smaller than the number of data users. Because it is the data users who need fine-grained privilege control, we must provide more Privileges related to usage and more tightly gatekeeper the administrative Privileges. To enforce this, we'll introduce the concept of Ownership as a complete replacement for the administrative category of Privilege.

Ownership

When you create a resource (Gravitino Service, Metalake, Catalog, and any other entity) in Gravitino, each entity has an Owner field that defines the user (or group) to which the resource belongs. The owner of each entity has implicit administrative class privilege, for example to delete that resource. Only the Owner of a resource can fully manage that resource.
If a resource needs to be managed by more than one person at the same time, the owner is assigned to a user group.

Role

The traditional rights system generally uses RBAC (Role Based Access Control) for rights management, where each Role contains a collection of different operating privileges for different resources. When the system adds a new user or user group, you can select the Roles which they are expected to be granted to, so that the user can quickly start using it, without waiting for the administrator to gradually set up the access rights to resources for him.

Roles also employ the concept of ownership – the owner of a Role is by default the creator of the Role, implying the owner has all the permissions to operate the Role, including deleting the Role.

Gravitino introduces the concept of the "Current" Role, where a user can own multiple Roles at the same time, but is restricted to use a Current Role at one moment. When the user in the resource operation is in the Current Role's permissions. At that moment the user will be able to display the Current Role in the list of resources with access rights. Users are able to switch to the Current Role to perform system operations with different roles they have access to. The functional design of the Current Role is to allow users to clearly define and interpret the resources and permissions owned by the Current Role, instead of all their roles put together.

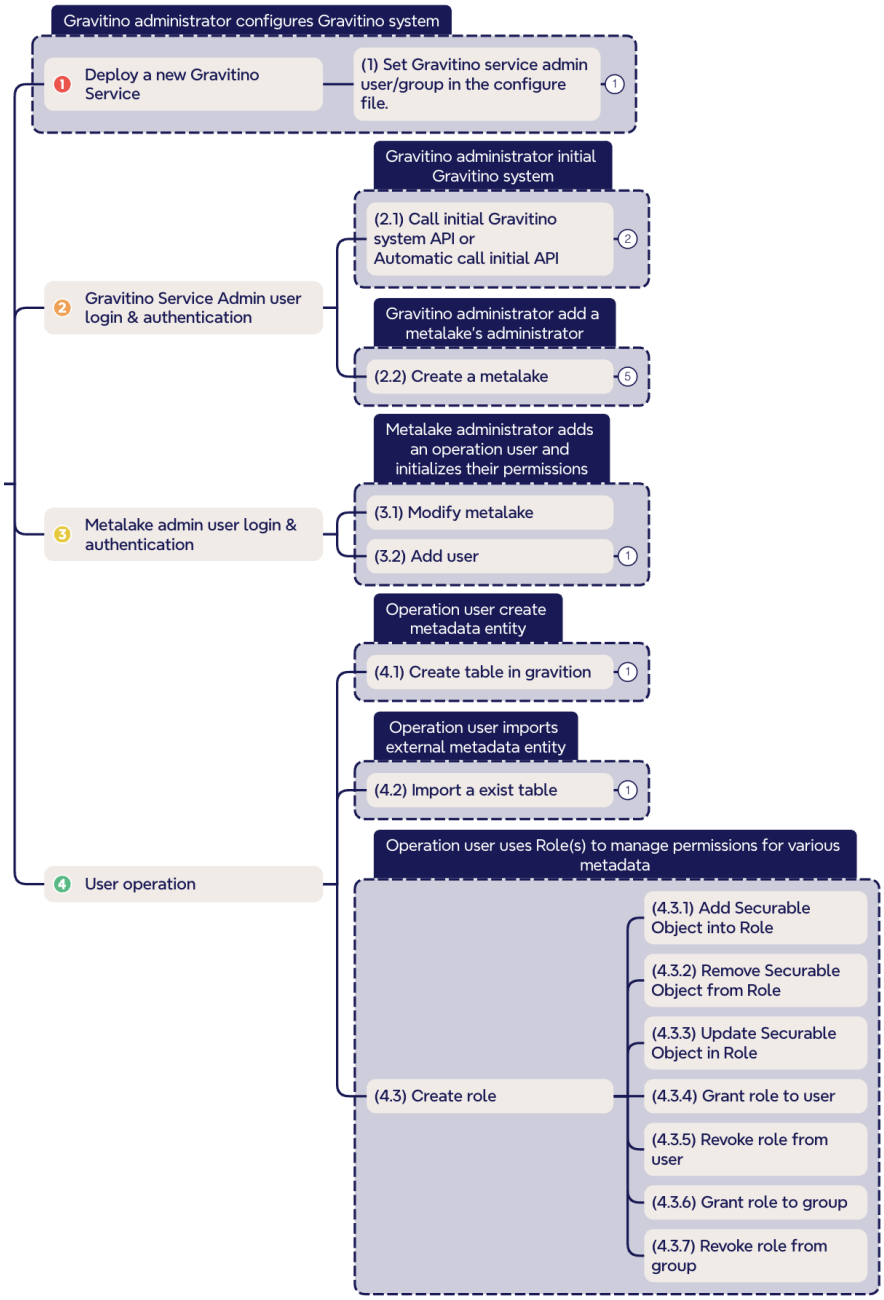
Predefined Roles

There are three typical users in a Gravitino system: the Data Analyst, the Data Administrator, and the Data Auditor. In order to simplify the complexity of setting permissions for them and to allow them to work quickly in Gravitino, predefined templates for these three roles have been created in Gravitino: Asset Viewer, Asset Manager, and Compliance Manager (for more details, please browse the Design Documentation for Predefined Roles Gravitino predefined roles).

Predefined Roles Permission Mapping Table

Predefined role \Permission	**Asset viewer**	**Asset manager**	**Compliance manager**
Create catalog	No	Yes	No
Create schema/table	No	Yes(Only the entities he has WRITE permission)	No
edit/delete catalog/schema/table	No	Yes(Only the entities he has WRITE permission)	No
View catalog/schema/table	Yes	Yes	Yes
Create/edit/delete Tag/row-filter/col-mask	No	No	Yes
Add/remove tag to entities	No	Yes(Only the entities has WRITE permission)	No
Create/edit/delete user	No	No	No
Create/edit/delete role for entities	No	Yes(Only the entities has WRITE permission)	No
View asset reports	Yes	Yes	Yes
View compliance reports	No	No	Yes

Permission Processing Flow in Gravitino



Please install xmind to open this mind file:
https://drive.google.com/file/d/1cUTU7H0FJTjWi23i0Hh-g9V-l65Rqr-B/view?usp=drive_link

Deploying a new Gravitino Service

When we are deploying a new Gravitino service, we need to set the Gravitino Service's administrator name to the Gravitino configuration file before starting the Gravitino service.

Administrator initializes Gravitino Service

After an administrator of the Gravitino Service logins to Gravitino, they will need to use Gravitino's system initialization Interface to carry out the following steps:

1. Add the administrator to Gravitino's User information table.
2. Create a ROOT entity object to represent the Gravitino service system, and set the owner of the ROOT entity object to be the administrator, so that the administrator has administrative permission of the Gravitino service and can manage Metalake.

Administrator creates a metalake

A metalake in Gravitino will encompass an entire business organization, users, user groups, and defined roles. Data catalogs or meta stores that belong to different subsets within the business organization may also want their own metalake tenant space. If a given group wants to use Gravitino, they will need to create their own dedicated metalake to fully leverage all of its features. To start this process, the appointed Metalake Administrator will need to submit a request to the Gravitino Service Administrator to create the metalake. From there, the Gravitino Service Administrator will need to perform the following actions:

- The Gravitino Service Administrator creates a metalake entity for the applicant and sets the owner.
- With the metalake entity, the Gravitino Service Administrator can add the Metalake Administrator to the user table information.
- The Gravitino Service Administrator copies a set of predefined roles for each new metalake based on the system's predefined role templates and sets the owner of the role to be the Metalake Administrator, and these predefined roles can only be further modified and adjusted by the Metalake's Administrator.

Metalake Administrator operation flow

After a Gravitino Administrator creates a metalake, the Metalake Administrator can log in to the Gravitino system and update other configurations for the metalake, such as setting a new name. The Metalake Administrator can then also add operational users from their organization to the metalake and set up predefined roles or custom roles for these users.

User operation flow

There are two types of metadata managed in the Gravitino system, the first is the metadata created by Gravitino, and the second is metadata that already exists in an external system and then imported into Gravitino to be managed by Gravitino These two types of metadata are treated in Gravitino using different Ownership methods:

- Metadata created natively withinGravitino will only configure the Owner of this metadata as the creator, and the creator will have full management rights and even the ability to delete this metadata.
- Metadata that is imported into Gravitino for delegated management will not have Owner information set, ensuring that metadata from external systems can only be deleted by the external system that it pulls from, this makes the use of this metadata more secure and reliable.

Regardless of source, both of these 2 types of metadata can still have their permissions managed through Gravitino's Role system.

Permission

Gravitino’s unified management model allows for each data source to have its own authorization features. However, each data source may come with its own dedicated authorization model and methods. We may not be able to properly set permissions to the underlying system, so when a given user tries to access this data, the underlying authorization system may result in permission inconsistencies and cause issues for external access. To mitigate this issue, Gravitino aims to provide a unified authorization model and accompanying methods that sits on top of all the data sources instead, making it much easier to manage access rights.

It is important to note that Gravitino’s authorization model will not merge the permission systems of the underlying data sources to form a large and unwieldy set of permissions. Instead, We will summarize the usage of the permissions currently in use within the data system, and offer a set of Gravitino-native permission models that accurately reflects it.

This is so that when users and data engines use Gravitino for data processing, this permission model is used to address the complexity of managing permissions for different data sources.
This set of permission models is meant to keep everything within the Gravitino system while still managing the permission settings of different data sources separately.

Permission Mode

```
Privilege: {
   "name": "CREATE_CATALOG",
   "condition": "ALLOW"
}
```

```
SecurableObject: {
  "name": "securableObject1",
  "type": "TABLE",
  "parent": "schema1",
  "privileges": [
    {
      "name": "READ_XXX",
      "condition": "ALLOW"
    },
    {
      "name": "WRITE_XXX",
      "condition": "DENY"
    }
  ]
}
```

```
User: {
   "name":"John",
   "roles":["roleId1", "roleId2", "roleId3"]
}
```

```
Group: {
   "name":"admin",
   "roles":["roleId1", "roleId2", "roleId3"]
}
```

```
Role: {
  "name": "role1",
  "securableObjects": [
        {
          "name": "securableObject1",
          "type": "TABLE",
          "parent": "schema1.table1",
          "privileges": [
            {
              "name": "READ_XXX",
              "condition": "ALLOW"
            },
            {
              "name": "WRITE_XXX",
              "condition": "DENY"
            }
          ]
        },
        {
          "name": "securableObject2",
          "type": "TABLE",
          "parent": "schema1.table2",
          "privileges": [
            {
              "name": "READ_XXX",
              "condition": "ALLOW"
            }
          ]
        },
  ],
  "owner":"John"
}
```

Privilege

Privilege defines the types of operations on different metadata entities, and is used to allow or deny a specific type of operation on an entity object.

SecurableObject

SecurableObject binds multiple operation-specific types to a single operation entity object (Privilege)

Role

A Role is a collection of SecurableObjects, and a Role represents multiple operation type permissions on multiple operation entities.

User

Users are generally granted one or multiple Roles, and users have different operating privileges depending on their Role.

Group

To make it easier to grant a single permission to multiple users, we can add users to a user group, and then grant one or multiple Roles to that user group. This process allows all users belonging to that user group to have the permissions in those Roles.

Storage of authority data

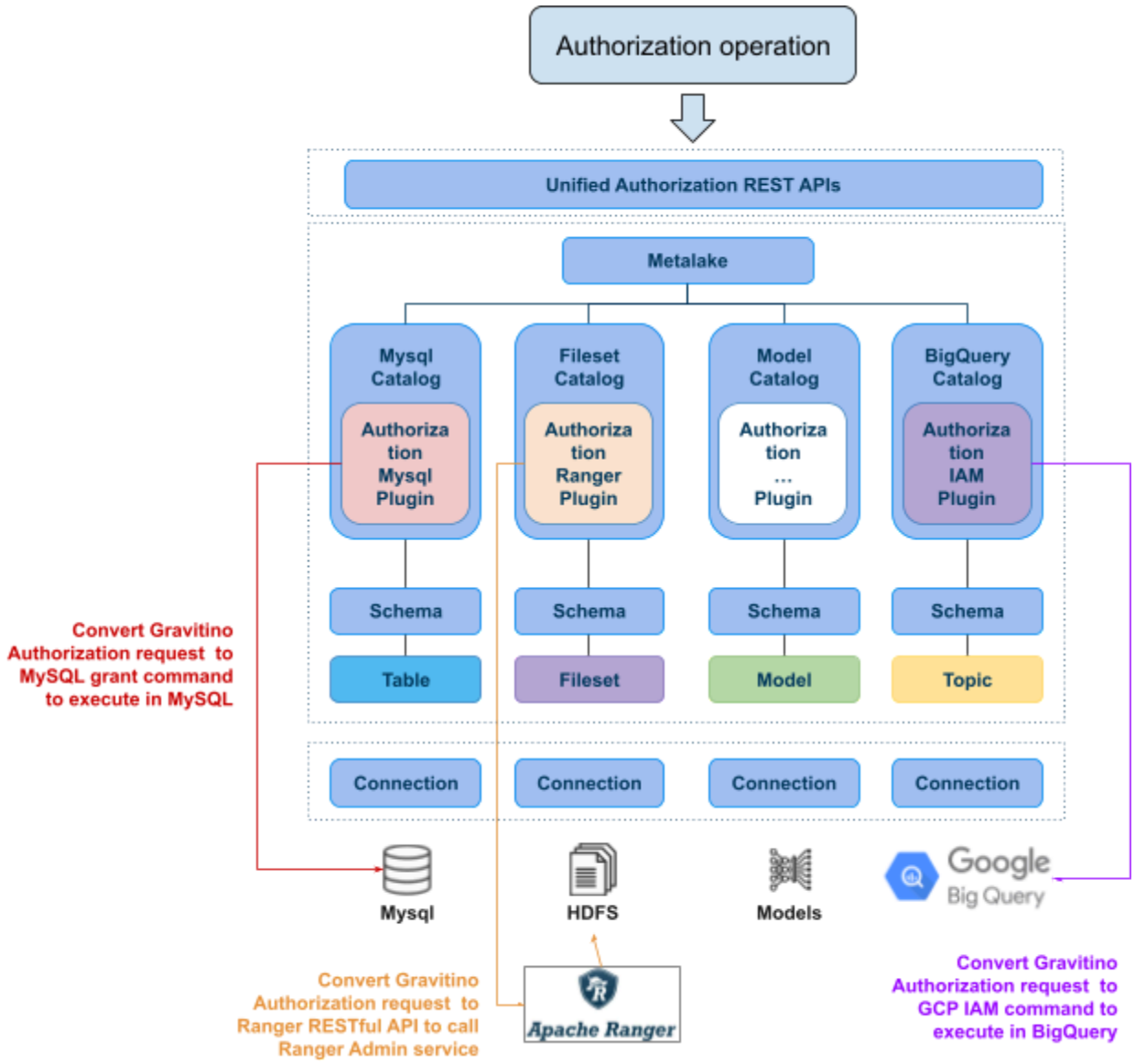
A key requirement of Gravitino should be to display permission information and perform operations such as grant and revoke, along with push-down of the permission information to the underlying data source system. It also needs to be able to translate that information to the corresponding permissions within the underlying system in order to achieve permission control over the data in the data source.

If Gravitino does not store privilege information itself and only push-down permission to the underlying data source system, it would be very difficult for Gravitino to translate Gravitino's privilege model back from the different privilege systems of the underlying data source.

Gravitino currently stores all the permission information (Privilege, SecurableObject, Role, User and Group) itself, and when there is a discrepancy between the permissions of the underlying data source and the permissions stored in Gravitino, Gravitino’s permission system will be trusted.

Additionally, Gravitino provides a mechanism to ensure fault tolerance in the privilege data storage and push-down to the underlying data source privilege system. As a result of this, Gravitino’s push-downs to the underlying permission system operations are required to be idempotent, meaning that each kind of privilege push-down operation can be repeated without causing conflicts and will always achieve the same output.

Authorization



Gravitino also provides a set of authorization frameworks to interface with different underlying data source authorization systems (e.g., MySQL's permission management and the Apache Ranger permission management system for big data) in accordance with its own authorization model and methodology.

On top of this, Gravitino manages different underlying data sources through Catalogs. When a user performs an authorization operation on the data in a Catalog, Gravitino will call the interface of the Authorization Plugin module in the respective Catalog to encode the Gravitino authorization model into the underlying data source's permission rules of the underlying data source.

Permission is then pushed down to the underlying permission system through the client of the underlying data source (JDBC or Apache Ranger client, etc.).

Authentication

As mentioned above, Gravitino uses Ownership to control the rights of resources in the management category and uses Role to control the permissions of operations in the operation category, so when a user performs a specific operation on a specified resource, Gravitino will perform a composite authentication on the Ownership and Role to which the resource belongs.
When a user has more than one Role, Gravitino will use the user's current Role for authentication, and the user can switch the current Role to access different resources.

Permission fuzzy matching

Gravitino uses the resource's ENTITY ID (long type) to preserve permission relationships.
So Gravitino can't directly support fuzzy matching of resource names (string type), such as wildcards like (*) and (%).
Gravitino uses the resource parent node to express support for all resource (*) wildcards for child resources, for example, if we need to set read permissions for all table resources, we can set it to {catalog1.schema1, READ_TABLE_PRIVILEGE}, which stands for having access to catalog1.schema1.* read access to all tables.