[Discuss] Unified Catalog APIs

Xuefu Zhang, Timo Walther, Fabian Hueske, Piotr Nowojski, Bowen Li

Motivation

With its wide adoption in streaming processing, Flink has also shown its potentials in batch processing. Improving Flink's batch processing, especially in terms of SQL, would generate a greater adoption of Flink beyond streaming processing and offer user a complete set of solutions for both their streaming and batch processing needs.

On the other hand, Hive has established its focal point in big data technology and its complete ecosystem. For most of big data users, Hive is not only a SQL engine for big data analytics and ETL, but also a data management platform, on which data are discovered, defined, and evolved. In another words, Hive is a de facto standard for big data on Hadoop.

Therefore, it's imperative for Flink to integrate with Hive ecosystem to further its reach to batch and SQL users. In doing that, integration with Hive metadata and data is necessary.

There are two aspects of Hive metastore integration: 1. Make Hive's meta-object such as tables and views available to Flink and Flink is also able to create such meta-objects for and in Hive; 2. Make Flink's meta-objects (tables, views, and UDFs) persistent using Hive metastore as an persistent storage.

This document is one of the three parts covering Flink and Hive ecosystem integration. It is not only about Hive integration but also reworking the catalog interfaces and unification of the TableEnvironment's catalog and external catalogs, with a long term goal of being able to store both batch and streaming connector information in a catalog (not only Hive but also Kafka, Elasticsearch, etc).

Proposed Changes

In current Flink code base, there is already a set of interfaces defined for external catalog. However, the APIs are not yet stable, needing to adapt for our work.

The proposed changes are best demonstrated in the following class hierarchy:

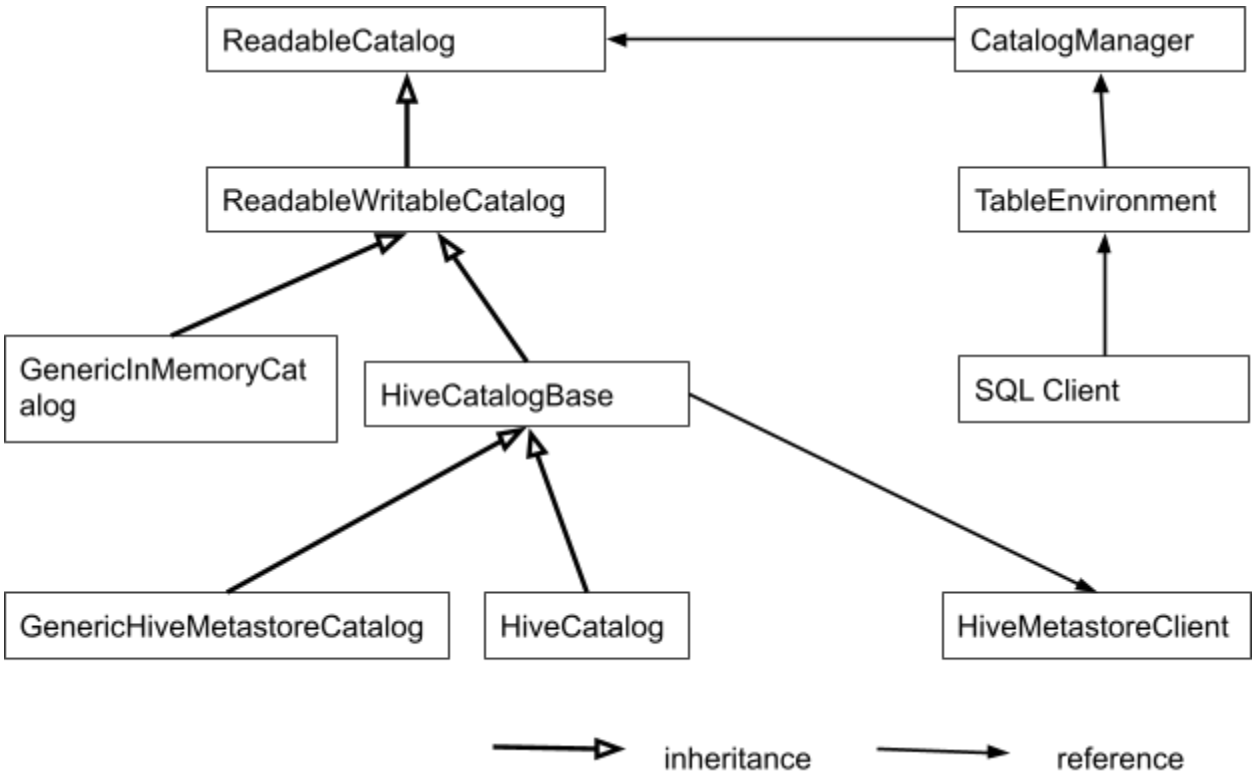

Fig. 1 Overall class diagram for Hive metastore integration

In Fig. 1, `ReadableCatalog`, `ReadableWritableCatalog`, and `CatalogManager` are the main interfaces that we are defining. Others are just implementations or interface callers.

`ReadableCatalog` Interface

This class comes from renaming existing `ExternalCatalog` class. The reason for removing "external" keyword is that there isn't clear distinction between internal and external as a an external catalog can also be used to store Flink's meta objects.

We need to adapt the existing APIs to accommodate other meta-objects like tables and views that are present in Flink and also common in a typical database catalog. We also reinstate schema/database concept rather than non-standard subcatalog term.

```
public interface ReadableCatalog {
    void open();
    void close();
```

```
    // Get a table factory instance that can convert catalog tables in this catalog
    // to sources or sinks. Null can be returned if table factory isn't provided,
    // in which case, the current discovery mechanism will be used.
    Optional<TableFactory> getTableFactory();

    List<String> listDatabases();
    CatalogDatabase getDatabase(String databaseName);

    /**
     * List table names under the given database. Throw an exception if database isn't
     * found.
     */
    List<String> listTables(String databaseName);

    /**
     * List view names under the given database. Throw an exception if database isn't
     * found.
     */
    List<String> listViews(String databaseName);

    // getTable() can return view as well.
    CommonTable getTable(ObjectPath tableOrViewName);

    /**
     * List function names under the given database. Throw an exception if
     * database isn't found.
     */
    List<String> listFunctions(String databaseName);

    CatalogFunction getFunction(ObjectPath functionName);
}
```

The changes are:

1. Add `open()` and `close()`. They are added to `ReadableCatalog` interface to take care of external connections. They might take some runtime context, but I leave it out for now.
2. Added view/UDF related reads
3. Define relationship between table and view (see Fig. 2)

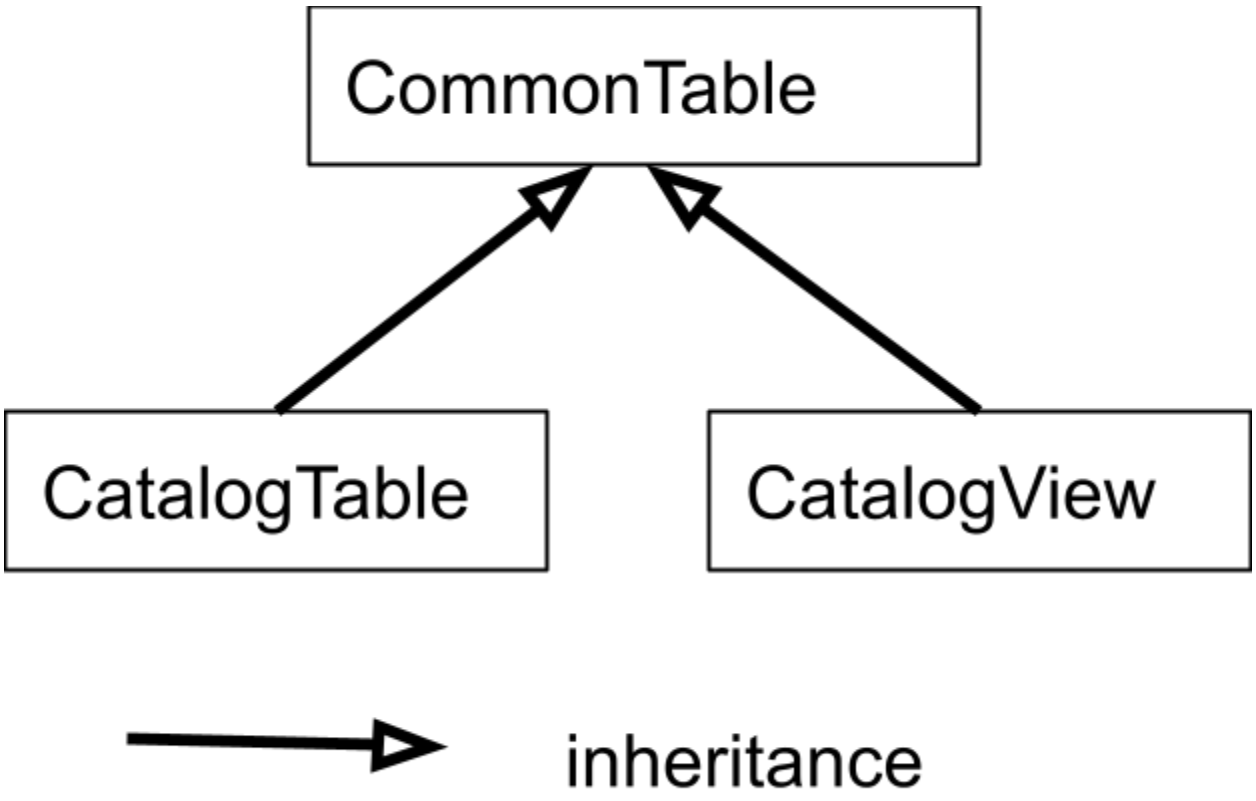

Fig. 2 Table/View hierarchy

View is a specific type of table. To be more specific, view is a virtual table defined over other tables and/or views with a SQL statement.

`CatalogDatabase` Class

This represents a schema/database object. It's currently modelled as a subcatalog, which came from FLINK-6574. More discussions follow in "Additional Notes" section.

Please note that many meta object classes, including `CatalogDatabase`, `CatalogTable`, and `CatalogView`, have a member variable called properties. They come because external catalog might allow user to specify any generic properties such as owner, creation_time, last_access_time, etc.

```
public class CatalogDatabase {
    private Map<String, String> properties;

    public CatalogDatabase(Map<String, String> properties) {
        this.properties = properties;
```

```
    }

    public Map<String, String> getProperties() {
        return properties;
    }
}
```

`ObjectPath` Class

```
/**
 * A database name and object (table/function) name combo in a catalog
 */
public class ObjectPath {
    private final String databaseName;
    private final String objectName;

    public ObjectPath(String databaseName, String objectName) {
        this.databaseName = databaseName;
        this.objectName = objectName;
    }
}
```

`CatalogTable` Interface

Class `CatalogTable` is renamed from `ExternalCatalogTable` with the following changes. It can be implemented by a property map where everything about the table is encoded as key-value pairs (descriptor) that encods table stats and schema, or by just a POJO class that has table schema, table stats, and table properties.

```
public interface CommonTable {

    public Map<String, String> getProperties();

}

public interface CatalogTable extends CommonTable{

    public TableSchema getSchema();

    public TableStatistics getTableStatistics();

}
```

`HiveTable` Class

```
Public class HiveTable implements CatalogTable {
    Public TableSchema getSchema() {
```

```
       // get from Hive megastore
    }

    Public TableStats getStats() {
       // get from Hive megastore
    }

    /**
      * Hive table properties (not contain schema or stats)
      */
    Public TableStats getProperties() {
       // get from Hive megastore
    }
}
```

`GenericCatalogTable` Class

This class represents the tables that are currently defined in Flink, which have no external definition. Such tables are currently stored in memory but can be stored in a permanent storage for persistency across user sessions.

```
public class GenericCatalogTable implements CatalogTable {
    // All table info (schema, stat, and table properties) is encoded as properties
    private Map<String, String> properties;
    private TableSchema tableSchema;
    prviate TableStats tableStats;

    public TableSchema getSchema() {
       return tableSchema
    }

    Public TableStats getStats() {
       return tableStats;
    }

    public Map<String, String> getProperties() {
       return properties;
    }
}
```

`CatalogView` Interface

A catalog view is a specific type of `CommonTable`. A view is defined with a query statement, whose expanded form needs to be stored as well to remember query context such as current database.

```
public interface CatalogView extends CommonTable {
    // Original text of the view definition.
```

```
    String getOriginalQuery();
    // Expanded text of the original view definition
    // This is needed because the context such as current DB is
    // lost after the session, in which view is defined, is gone.
    // Expanded query text takes care of the this, as an example.
    String getExpandedQuery();

}
```

`CatalogFunction` Class/Interface

This class represents a function (UDF) defined in a catalog. It serves as a placeholder for now as much details needs to be flushed out. However, it needs to cover Flink functions as well as Hive functions.

```
/**
  * The detailed definition of this class needs to be further sorted
  * out
  */
public class CatalogFunction {
    private Enum from; // source of the function (can only be "CLASS" for now)
    private String clazz; // fully qualified class name of the function
     ...
    private Map<String, String> properties;

    public CatalogFunction(String from, String clazz, Map<String, String> properties)
{
        this.properties = properties;
    }

    public Map<String, String> getProperties() {
        return properties;
    }
}
```

`ReadableWritableCatalog` Interface

This interface comes from renaming `CrudExternalCatalog` class. We added view and function related methods.

```
public interface ReadableWritableCatalog extends ReadableCatalog {
      void createDatabase(String databaseName, CatalogDatabase database, boolean
ignoreIfExists);
    void alterDatabase(String databaseName, CatalogDatabase database, boolean
ignoreIfNotExists);
    void renameDatabase(String databaseName,String newDatabaseName, boolean
ignoreIfNotExists);
    void dropDatabase(String databaseName, boolean ignoreIfNotExists);
```

```
    void createTable(ObjectPath tableName, CatalogTable table, boolean
ignoreIfExists);

    /**
     * dropTable() also covers views.
     * @param tableName
     * @param ignoreIfNotExists
     */
    void dropTable(ObjectPath tableName, boolean ignoreIfNotExists);
    void renameTable(ObjectPath tableName, String newTableName, boolean
ignoreIfNotExists);
    void alterTable(ObjectPath tableName, CatalogTable table, boolean
ignoreIfNotExists):

    void createView(ObjectPath viewName, CatalogView view, boolean ignoreIfExists);
    void alterView(ObjectPath viewName, CatalogView view, boolean ignoreIfNotExists);

    void createFunction(ObjectPath funcName, CatalogFunction function, boolean
ignoreIfExists);
    void renameFunction(ObjectPath funcName, String newFuncName, boolean
ignoreIfNotExists);
    void dropFunction(ObjectPath funcName, boolean ignoreIfNotExists);
    void alterFunction(ObjectPath funcName, CatalogFunction function, boolean
ignoreIfNotExists);
}
```

`HiveCatalogBase` Class

```
abstract class HiveCatalogBase implements ReadableWritableCatalog {
    Private HiveMetastoreClient hmsClient;

    // implementation for reading metadata from or writing metadata to
    // Hive metastore

    // Any utility methods that are common to both HiveCatalog and
    // FlinkHmsCatalog
}
```

`HiveCatalog` Class

```
class HiveCatalog extends HiveCatalogBase {

    public TableFactory getTableFactory() {
        return new HiveTableFactory();
    }

    // Implementation of other methods that are not implemented yet.
```

```
}
```

`GenericHiveMetastoreCatalog` Class

This is an implementation class for a catalog to hold tables (views/functions) currently defined by Flink. This implementation leverage a Hive metastore as the persistent storage.

```
class GenericHiveMetastoreCatalog extends HiveCatalogBase {
    public TableFactory getTableFactory() {
        return null; // Use table factory discovery mechanism
    }

    // Implementation of other methods that are not implemented yet.
}
```

`CatalogManager` Class

We introduce CatalogManager class to manage all the registered `ReadableCatalog` instances in a table environment. It also has a concept of default catalog and default database, which will be selected when a catalog name isn't given in a meta-object reference.

```
public class CatalogManager {
    // The catalog to hold all registered and translated tables
    // We disable caching here to prevent side effects
    private CalciteSchema internalSchema = CalciteSchema.createRootSchema(true,
false);
    private SchemaPlus rootSchema = internalSchema.plus();

    // A list of named catalogs.
    private Map<String, ReadableCatalog> catalogs;

    // The name of the default catalog
    private String defaultCatalog = null;

    public CatalogManager(Map<String, ReadableCatalog> catalogs, String
defaultCatalog) {
        // make sure that defaultCatalog is in catalogs.keySet().
        this.catalogs = catalogs;
        this.defaultCatalog = defaultCatalog;
    }

    public void registerCatalog(String catalogName, ReadableCatalog catalog) {
        catalogs.put(catalogName, catalog);
    }

    public ReadableCatalog getCatalog(String catalogName) {
        return catalogs.get(catalogName);
```

```
    }

    public Set<String> getCatalogs() {
        return this.catalogs.keySet();
    }

       public void setDefaultCatalog(String catName) {
        // validate
        this.defaultCatalog = catqName;
    }

    public ReadableCatalog getDefaultCatalog() {
        return this.catalogs.get(defaultCatalog);
    }
}
```

Besides a list of `ReadableCatalogs`, `CatalogManger` also encapsulate Calcite's schema framework such that no code outside `CatalogManager` needs to interact with Calcite's schema except the parser which needs all catalogs. (All catalogs will be added to Calcite schema so that all external tables and tables can be resolved by Calcite during query parsing and analysis.)

`TableEnvironment` Class

This is the existing class in table API, which now has a reference to `CatalogManager` instance that is to be added to replace the in-memory meta-objects and registered catalogs.

```
abstract class TableEnvironment(val config: TableConfig) {
…
  private val catalogManager: CatalogManager;

  // This is an existing class with only argument type change
  def registerCatalog(name: String, catalog: ReadableCatalog): Unit

  // Set the default catalog
  def setDefaultCatalog(catName: String);

  // Set the default database
  Def setDefaultDatabase(catName: String, databaseName: String): unit

}
```

`TableEnvironment` class currently has a few `registerTable` methods taking different table definitions, such as `TableSource`, `TableSink`, and non-public classes such as Table `RelTable` and Table `InlineTable`. Those APIs will stay the same. However, their implementation might be changed in order to leverage a persistent catalog. The details will be provided in Part 2 of this design serials.

YAML Configuration for Catalogs

The following is a demonstration of configurations for catalogs in Flink. Available catalog types are: `flink-in-memory`, `generic-hive-metastore`, and `hive`. Details of each of the implementation classes and corresponding factory classes will be provided in subsequent design documents. Here we are only focused on how catalogs will be specified in YAML file.

```
catalogs:
-  name: hive1
    catalog:
     type: hive
     is-default: false
     default-db: default
     connection-params:
       hive.metastore.uris:
“thrift://host1:10000,thrift://host2:10000”
       hive.metastore.username: “flink”
- name: flink1
   catalog:
     type: generic-hive-metastore
     is-default: true
     Default-db: default
     connection-params:
       hive.metastore.uris:
“thrift://host1:10000,thrift://host2:10000”
       hive.metastore.username: “flink”
       Hive.metastore.db: flink
```

`TableFactory` Interfaces

```
// This is the existing interface.
interface TableFactory {
  Map<String, String> requiredContext();

  List<String> supportedProperties();

}

// Utility classes providing implementations for conversions between CatallogTable
// and property map.
public class TableFactoryUtils {

  public static CatalogTable convertToCatalogTable(Map<String,
    String> properties) {
```

```
    // default implementation
  }

  Public static Map<String, String> convertToProperties(CatalogTable
    table) {
    // implementation
  }

}

Interface StreamTableSourceFactory extends TableFactory {
    // this one is existing one, which will be deprecated.
    @Deprecated
    StreamTableSource createStreamTableSource(Map<String, String> properties);

    // This one is new with default implementation.
    Default StreamTableSource createStreamTableSource(CatalogTable table) {
      return createStreamTableSource(
        TableFactoryUtils.convertToProperties(table) );
    }
}

Interface StreamTableSinkFactory extends TableFactory {
    // this one is existing one
    StreamTableSink createStreamSinkSource(Map<String, String> properties);

    // This one is new.
    Default StreamTableSink createStreamSinkSource(CatalogTable table) {
      return createStreamTableSink(
        TableFactoryUtils.convertToProperties(table) );
    }
}

Interface BatchTableSourceFactory extends TableFactory {

    // this one is existing one
    BatchTableSource createBatchTableSource(Map<String, String> properties);

    // This one is new.
    Default BatchTableSource createBatchTableSource(CatalogTable table) {
      return createBatchTableSource(
        TableFactoryUtils.convertToProperties(table) );
    }

}

Interface BatchTableSinkFactory extends TableFactory {
    // this one is existing one
    BatchTableSink createBatchTableSink(Map<String, String> properties);
```

```
    // This one is new.
    BatchTableSink createBatchTableSink(CatalogTable table) {
      return createBatchTableSink(
        TableFactoryUtils.convertToProperties(table) );
    }
}
```

`HiveTableFactory` Class

```
/**
  * Batch only for now.
  */
Public class HiveTableFactory implements BatchTableSourceFactory,
BatchTableSinkFactory {
    Map<String, String> requiredContext() {
        // return an empty map to indicate that auto discovery is not needed.
        return new HashMap<>().
    }

    List<String> supportedProperties() {
        // Return an empty list to indicate that no check is needed.
        Return new ArrayList<>();
    }

    BatchTableSource createBatchTableSource(Map<String, String> properties) {
        // convert properties to catalogtable and call the other version of this
method.
        // It’s fine not to support this method.
    }

    BatchTableSource createBatchTableSink(Map<String, String> properties) {
        // convert properties to catalogtable and call the other version of this
method.
        // It’s fine not to support this method.
    }

    BatchTableSource createBatchTableSource(CatalogTable table) {
        Assert (table instanceof HiveTable);
          HiveTable hiveTable = (HiveTable)table;
          // create a table source based on HiveTable
          // This is specific implementation for Hive tables.
    }

    BatchTableSource createBatchTableSink(CatalogTable table) {
        Assert (table instanceof HiveTable);
          HiveTable hiveTable = (HiveTable)table;
        // create a table sink based on HiveTable
        // This is specific implementation for Hive tables.
       }
}
```

Auto Table Factory Discovery

If a catalog (such as `GenericHiveMetastoreCatalog` above) returns `null` from its `getTableFactory()` implementation, then the framework will automatically discover the real table factories utilizing Java Service Provider interfaces (SPI). This is existing mechanism for all current tables defined in Flink.

Additional Notes

With multiple catalog in one system, a table has to be identified by catalog name, schema/database name, and table name. Thus, table reference needs to include catalog name, schema name, and table name, such as `hive1.risk_db.user_events`. When catalog name is missing, it's assumed to mean the default catalog (whatever a catalog that is set as default) and the default database.

We introduced default database concept in Flink SQL. This corresponds to SQL "use xxx" where a schema(database) is set as the current one and any table without database/schema prefix is referring to the default schema. Since Flink has multiple catalogs, the syntax would be "use cat1.db1", with which cat1 will be the default catalog and db1 will be the default database. Given a table name, it has to be resolved by catalog manager to the full name in order to correctly identify the table.

This is in contrast with the changes made in FLINK-6574, where the attempt was to reduced the need of specifying catalog name. In theory this is not feasible because of the fact that multiple catalogs are supported. Preliminary test showed that FLINK-6574 doesn't achieve what it attempts to do. Rather, it created great conceptual confusion. Therefore, we will review and adapt the changes in this effort.

Implementation Plan

Task breakdown

1. Create `ReadableCatalog`, `ReadableWritableCatalog`, and related interfaces. Deprecate existing `ExternalCatalog` and `CrudExternalCatalog` interfaces.
2. Adapt existing `InMemoryExternalCatalog` to `GenericInMemoryCatalog` class.
3. Create `CatalogManager` in `TableEnvironment` to manage registered (possibly multiple) catalogs and encapsulate Calcite schema management.
4. Define catalog entries in SQL client YAML file and handle creation and registration of those entries.
5. Process existing table entries in YAML file to keep backward compatibility.
6. Implement `HiveCatalog` and `HiveTableFactory`.

7. Hook up `HiveTableFactory` with existing table factory discovery.
8. Implement `GenericHiveMetastoreCatalog`.

Prerequisites

1. The design here is completely JAVA-based, so the implementation depends on the completion of the effort that ports current `TableEnvironment` and related classes to JAVA. [FLIP-28], [FLINK-11067]