Flink SQL DDL

Author

- Shuyi Chen (suez1224@gmail.com)
- Fabian Hueske (fabian@data-artisans.com)
- Timo Walther (timo@data-artisans.com)
- Rong Rong (walterddr@gmail.com)

Motivation

Current Flink SQL API support only DML (e.g. SELECT and INSERT statements). With only DML, Flink SQL API users still need to define/create table sources and sinks programmatically in Java/Scala. Also, if users want to create View, User-Defined Type or load an external UDF to use within SQL, they need to do so programmatically in Java/Scala as well. These make the Flink SQL API less useful as we should enable the users to define all database schemas/types and etc., all within SQL. This motivates the addition of SQL DDL support in Flink. With DDL support in Flink,

- Users can define/alter/delete table sources and sinks using SQL in Flink
- Users can define/alter/delete view (virtual tables) using SQL in Flink
- Users can define/replace/delete user defined types using SQL in Flink
- Users can use SQL to load external functions and use it as UDF in Flink SQL.

Also, In [FLIP-24], a SQL client is proposed to allow Flink users to use Flink SQL without any IDE or programming effort. However, the current implementation does not allow dynamical creation of table, type or functions within SQL, and users need to put those configurations statically as parameters or configuration files passing into the SQL client before startup. With SQL DDL support, we can extend the SQL client to dynamically create/alter/delete table/view/type/functions/etc. within the command line session.

Usage Example

The following SQL statements use DDL to define a source Kafka table in the default catalog, and Cassandra sink table in the default catalog, a UDAF from a Jar library located in HDFS,

and finally does a SQL query from the Kafka table and streams the output to the Cassandra table.

```
CREATE SOURCE TABLE Kafka10SourceTable (
        intField INTEGER,
        stringField VARCHAR(128) COMMENT 'User IP address',
        longField BIGINT,
        rowTimeField TIMESTAMP
        TIMESTAMPS FROM 'longField'
        WATERMARKS PERIODIC-BOUNDED WITH DELAY '60'
)
COMMENT 'Kafka Source Table of topic user_ip_address'
WITH (
        connector.type='kafka',
        connector.property-version='1',
        connector.version='0.10',
        connector.topic='test-kafka-topic',
        connector.startup-mode = 'latest-offset'
        connector.specific-offset = 'offset'
        format.type='json'
        format.property-version = '1'
        format.version='1'
        format.derive-schema='true'
)

CREATE SINK TABLE CassandraSinkTable (
        newIntField INTEGER,
        stringField VARCHAR(128)
)
WITH (
        connector.type='cassandra',
        connector.property-version='1',
        connector.version='3.0.0',
        connector.contact-points='127.0.0.1',
        connector.cql ='INSERT INTO testNamespace.testTable (newIntField, stringField)
        VALUES(?,?)'
)

CREATE LIBRARY myLib AS 'HDFS:///users/testUser1/flink-udf/myudf-0.1.jar';

CREATE FUNCTION myUDAFunc AS 'com.test.myUDAF' LIBRARY myLib;
```

INSERT INTO CassandraSinkTable **SELECT** myUDAFunc(intField) **AS** newIntField, stringField **FROM** Kafka10SourceTable **GROUP BY TUMBLE**(rowtimeField, **INTERVAL** '5' **MINUTE**), stringField;

Goals

We would like to achieve the following goals in this FLIP.

- Provide a basic design and framework for Flink SQL DDL.
- Add Table DDL support.
- Add View DDL support.
- Add Type DDL support.
- Add Library DDL support.
- Add Function DDL support.
- Add multi-queries SQL support.
- Integrate Flink SQL DDL with SQL client module.

DDL Grammar Design

We will extend the Calcite core parser in Flink to add DDL support. Below are the initial proposed grammar.

Table DDL

createTableStatement:
    CREATE [SOURCE | SINK] TABLE [IF NOT EXISTS] name
    ['(' tableElement [, tableElement] ')']*
    [COMMENT table_comment]
    [WITH '(' name=value [, name=value] ')']*

Name:
   [[catalogName .] schemaName .] tableName

tableElement:
    columnName fieldType
   [COMMENT column_comment]
   [rowTimeDefinition|procTimeDefinition|fromFieldDefinition]
   [WITH COLPROPERTIES '(' name=value [, name=value] ')']*
 | tableConstraint

fromFieldDefinition:
  FROM originalFieldName

procTimeDefinition:
PROCTIME

rowTimeDefinition:
TIMESTAMP
{
FROM-SOURCE
|
FROM '*fieldName*'
|
CUSTOM,
'(' CLASS = *extractorClassName* ')'
}
WATERMARKS
{
PERIODIC-ASCENDING
|
PERIODIC-BOUNDED WITH DELAY '*delay*'
|
FROM-SOURCE
|
CUSTOM,
'(' CLASS=*strategyClassName* ')'
}

fieldType:
{ *simpleType*
| ROW '(' *name* AS *fieldType* [, *name* AS *fieldType*]* ')'
| MAP '(' *simpleType* , *fieldType* ')'
| ARRAY '(' *fieldType* ')'
} [[NOT] NULL]

tableConstraint:
[CONSTRAINT *name*]
{
CHECK '(' *expression* ')'
| PRIMARY KEY '(' *columnName* [, *columnName*]* ')'
| UNIQUE '(' *columnName* [, *columnName*]* ')'
}

dropTableStatement:
DROP TABLE *name* [IF EXISTS]

View DDL

createViewStatement:
CREATE VIEW [IF NOT EXISTS] *name*
['(' *columnName [COMMENT column_comment]*, ... ')']
[COMMENT *view_comment*]
[WITH '(' *name=value [, name=value]** ')']

dropViewStatement:
DROP VIEW *name* [IF EXISTS]

Type DDL

createTypeStatement:
CREATE [OR REPLACE] TYPE *name* AS *fieldType*

dropTypeStatement:
DROP TYPE [IF EXISTS] *name*

Library DDL

createLibraryStatement:
CREATE [OR REPLACE] LIBRARY
name
AS '*library-path*'
[DEPENDS '*support-path*' [, '*support-path*']*]

dropLibraryStatement:
DROP LIBRARY [IF EXISTS] *name*

Function DDL

createFunctionStatement:
CREATE [OR REPLACE] FUNCTION *name*
AS '*functionClass*' [LIBRARY *library-name*]

dropFunctionStatement:

DROP FUNCTION [IF EXISTS] name

Proposed Changes

Shared Changes for all DDLs

Extending Calcite Parser

We will extend the Calcite core parser in Flink to parse Flink's DDL commands. The extended parser will be added within *flink-table* module.

For each DDL, a new type of SqlNode will be created and it will implement the *SqlDDLExecutable* interface to allow the DDL commands to be executed directly.

```
/**
* Mix-in interface for {@link SqlNode} that allows DDL commands to be
* executed directly.
*/
public interface SqlDDLExecutable {
    void execute(TableEnvironment tableEnv);
    void execute(SessionEnvironment SessionEnv)
}

public interface SessionEnvironment {
    void addTable(TableDescriptor tableDescriptor);

    void dropTable(String name);

    void addType(TypeDescripter typeDescripter);

    void dropType(String name);

    void addLibrary(LibraryDescripter libraryDescripter);

    void dropLibrary(String name);

    void addFunction(FunctionDescripter functionDescripter);

    void dropFunction(String function);
}
```

MultiQueries Support

We will also add a parser to parse multi-queries SQL separated by semicolon.

Flink Table Environment

We will extend the existing *sqlUpdate()* interface to parse and execute DDL commands in Flink TableEnvironment directly.

Also we will extract the Calcite parser instantiation logic to a auxiliary class *FlinkSqlParser*, so both TableEnvironment and SQL client can reuse to parse query and DDL statements.

Table DDL

The table DDL will make use of the unified table source/sink instantiation mechanism introduced in FLINK-8240 and FLINK-8866, and the unified connector API. We will convert the *SqlCreateTable* SqlNode into the properties map that is required by *TableFactoryService* to create the corresponding table source/sink.

View DDL

FLINK-10163 has already added initial support for View DDL, but without a proper parser. The effort would be to add proper parser for view creation and deletion.

Also, we might need to add proper guards when a SQL statement tries to modify a view because a view might not have corresponding columns into the underlying base tables. Currently, we can make view read-only, and we might add support for updatable view in the future.

Type DDL

CALCITE-2045 already added necessary changes in Calcite to support UDT. In addition to simple type and row type, we will also add map and array type support in Flink's type DDL.

In order for Table DDL to use UDT created through type DDL, we would need to translate the UDT into a concrete type string that *TypeStringUtils* can understand before instantiating the table source/sink.

Library DDL

The library DDL allows users to load external libraries, e.g. in local filesystem, HDFS and etc. and use the library in the Flink SQL application, e.g. load external UDFs from the library. We proposed the following changes.

Calcite Changes

We will add library support in Calcite in CALCITE-2046. With the change, library will become part of Schema like types, function and tables in Calcite.

Loading Libraries in Flink

In order to support loading external libraries and create UDFs from external libraries, there are 2 alternative solutions.

1) Add new interfaces *registerUserJarFile(String jarFile...)* and *getUserJarFiles()* in {Stream}ExecutionEnvironment. It will allow users to register user JAR files to load in their Flink job dynamically. Internally, the JAR files will be shipped using JobGraph.addJar() along with the JobGraph, and loaded into the userCodeClassLoader in RuntimeContext automatically.
2) Add a new interface in {Stream}ExecutionEnvironment.

   registerUserJarFiles(name, jarFiles...)

   The interface register a set of Jar files with key *name*. Internally, it uses similar path as *registerCachedFile()*, which distributes the Jar files to runtime using Flink's Blob Server. Also, add a new interface in RuntimeContext to create and cache a custom userCodeClassLoader using the Jar file set registered under *name*.

   getClassLoaderWithName(name)

   During code generation of the UDF function call, it will load the set of Jar files that are associated with the library into a custom classLoader, and invoke the function reflectively. Also, inside RuntimeContext implementation, we will keep a cached of all loaded custom classLoader so we wont load the same library multiple times.

The second approach will allow on-demand loading of libraries, and also, support loading the same class of different versions, which might help solve some dependency conflict issues. And, functional-wise speaking, the first approach is a simplified version of the second approach, in which the "*name*" parameter is a fixed value. Therefore, the second approach is preferred.

Function DDL

The function DDL allow users to load a UDF from existing classpath or external libraries and use it in SQL. It uses the *registerFunction* interfaces in [Stream/Batch]TableEnvironment to register UDF/UDAF/UDTF.

SQL client integration

Overview

In SQL client, it allow users to run multiple SQL commands in the same session. Also, SQL client already has ways to statically define table, load external libraries through configuration file.
To support DDL in SQL client, we will do the following:

- Add descriptor for types, libraries similar to TableDescriptor.
- Extend *org.apache.flink.table.client.config.Environment* to add instance variables for types and libraries, so users can configure types, libraries and functions through YAML statically.
- Extend *org.apache.flink.table.client.gateway.SessionContext* to implement *SessionEnvrionment* interface defined above, and add a new instance variable called *sessionEnvironment* in *SessionContext* to store the dynamic tables/types/libraries/functions modified through DDLs.
- Add *executeUpdate* interface in *Executor* interface.
- When *ExecutionContext* is created, it will merge all static environment and dynamically created environment through DDLs, and create/cache the Table/Type/Library/Functions in *ExecutionContext*.
- When *EnvironmentInstance* is created, it will register the Table/Type/Library/Functions with the created TableEnvironment.

Below are the proposed interface change in code.

```
public class Environment {

    private Map<String, TableDescriptor> tables;

    private Map<String, TypeDescriptor> types;

    private Map<String, LibraryDescriptor> libraries;

    private Map<String, FunctionDescriptor> functions;

    private Execution execution;

    private Deployment deployment;

    ...
}
```

```
public class SessionContext implement SessionEnvironment {

        private final String name;
        private final Environment defaultEnvironment;
        private final Environment sessionEnvironment;

        @override void addTable(TableDescriptor tableDescriptor);

        @override void dropTable(String name);

        @override void addType(TypeDescripter typeDescripter);

        @override void dropType(String name);

        @override void addLibrary(LibraryDescripter libraryDescripter);

        @override void dropLibrary(String name);

        @override void addFunction(FunctionDescripter functionDescripter);

        @override void dropFunction(String function);
        ...
}
```

Also, we will add "*create*", "*delete*", "*list types*", "*list libraries*" and "*list functions*" commands and etc. in *CliClient* to support DDL.

DROP Statement

Currently, DROP statement will only support dropping of the entities created dynamically through DDL in SQL client. A DROP statement on a entity defined statically through YAML file in SQL client will be a no-op.

SessionContext Persistence

With DDL, users can dynamically create new tables/functions and etc. And we should allow users to persist their current SessionContext, so they can resume from it later. Therefore, we will also add a '*save*' command to save the SessionContext to external filesystems.

Discussion

Side effect of DDLs

Currently, all CREATE and DROP DDLs will only create a temporary view in Flink, and should not have side effect on the external systems. For example, when a CREATE TABLE command is issued to create a KafkaTableSource, the DDL wont attempt to create the topic in Kafka if it does not exist; when a DROP TABLE command is issued on CassandraTableSink, it will only drop the CassandraTableSink instance in Flink, and the actual external Cassandra table will be intact.

In the future, we might look into adding new DDLs to manipulate the external tables through *ExternalCatalog*.

DROPPING non-DDL created entities

Dropping non-DDL created entities might be complicated. If the entities are defined in the SQL client YAML file, we can use the NULL object pattern, add a NULL descriptor to replace the static entity during environment merging given DDL commands take precedence over static YAML configurations. Currently, we don't plan to support entities that are created through external Catalog, support for this will be our future work.

Test Plan

Normal unit, integration, and end-to-end tests

Implementation Plan

Table DDL

1. Extend Calcite parser to support parsing Table DDL
2. Add support in *sqlUpdate*() in Flink's Table environment to support Table DDL
3. Add support in SQL client for handling Table DDL

View DDL

1. Extend Calcite parser to support parsing View DDL
2. Remove the View DDL parsing logic in SQL client
3. Add guard to prevent modifying View

Type DDL

1. Extend Calcite parser to support parsing Type DDL
2. Add support in *sqlUpdate*() in Flink's Table environment to support type DDL.
3. Add support in SQL client for handling Type DDL
4. Add map and array type support in Flink's type DDL

Library/Function DDL

1. Add library support in Calcite
2. Add support in {Stream}ExecutionEnvironment to allow on-demand loading of user libraries
3. Extend Calcite parser to support parsing Library/Function DDL
4. Add support in *sqlUpdate*() in Flink's Table environment to support Library/Function DDL
5. Add support in SQL client for handling Library/Function DDL

Compatibility, Deprecation, and Migration Plan

No compatibility changes or other deprecation necessary.