Explain Brief of the following in brief

● Hive UDF

User Defined Functions(UDFs) provides us a way to:

Extend the functionality of Hive by writing functions that can be evaluated in Hive QL.

Custom serializers and/or deserializer (“serdes”), which provide a way of either deserializing a custom file format stored on HDFS to a POJO (plain old Java object), or serializing a POJO to a custom file format (or both).

Custom mappers/reducers, which allow you to add a custom map or reduce steps into your Hive query. These map/reduce steps can be written in any programming language, and not just in Java.

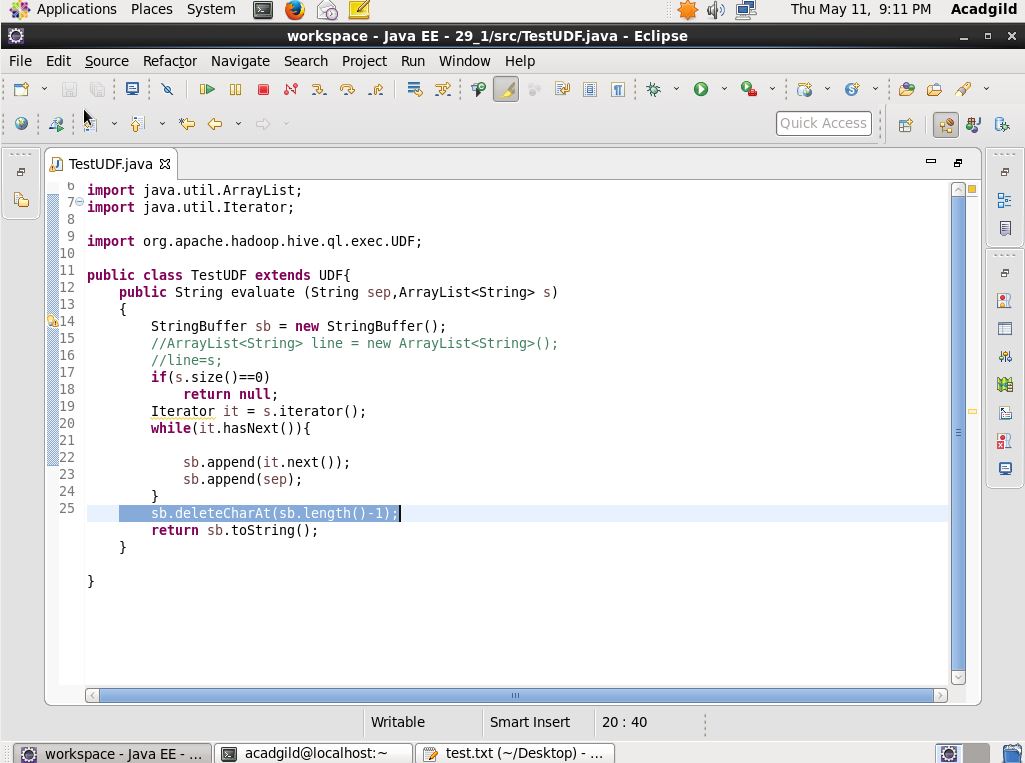
There are two different interfaces you can use for writing UDFs for Apache Hive. One is really simple, the other… not so much.

The simple API (org.apache.hadoop.hive.ql.exec.UDF) can be used so long as your function reads and returns primitive types. By this I mean basic Hadoop & Hive writable types - Text, IntWritable, LongWritable, DoubleWritable, etc.

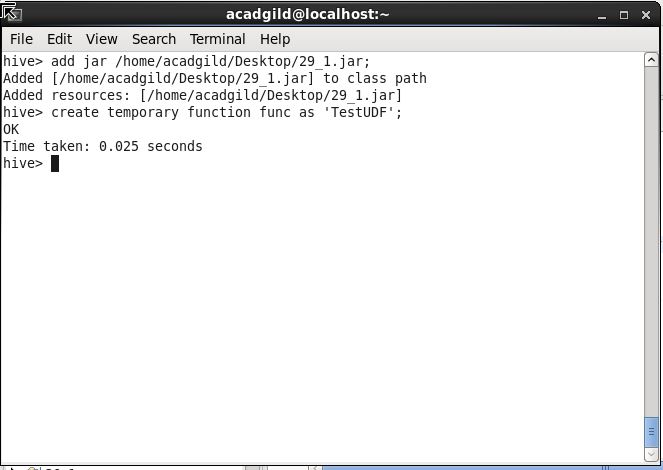
However, if you plan on writing a UDF that can manipulate embedded data structures, such as Map, List, and Set, then you’re stuck using org.apache.hadoop.hive.ql.udf.generic.GenericUDF, which is a little more involved.

* Simple API - org.apache.hadoop.hive.ql.exec.UDF
* Complex API - org.apache.hadoop.hive.ql.udf.generic.GenericUDF

The Simple API Building a UDF with the simpler UDF API involves little more than writing a class with one function (evaluate). Here is an example:









* UDAF:

1. Create a Java class which extends org.apache.hadoop.hive.ql.exec.hive.UDAF;
2. Create an inner class which implements UDAFEvaluator;
3. Implement five methods ()
   1. init() – The init() method initializes the evaluator and resets its internal state. We are using new Column() in the code below to indicate that no values have been aggregated yet.
   2. iterate() – this method is called every time there is a new value to be aggregated. The evaulator should update its internal state with the result of performing the aggregation (we are doing sum – see below). We return true to indicate that the input was valid.
   3. terminatePartial() – this method is called when Hive wants a result for the partial aggregation. The method must return an object that encapsulates the state of the aggregation.
   4. merge() – this method is called when Hive decides to combine one partial aggregation with another.
   5. terminate() – this method is called when the final result of the aggregation is needed.
4. Compile and package the JAR
5. CREATE TEMPORARY FUNCTION in hive CLI
6. Run Aggregation Query and Verify Output!!!

UDAF to find the largest Integer in the table.

package com.hive.udaf;

import org.apache.hadoop.hive.ql.exec.UDAF;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;

public class Max extends UDAF

{

public static class MaxIntUDAFEvaluator implements UDAFEvaluator

{

private IntWritable output;

public void init()

{

output=null;

}

public boolean iterate(IntWritable maxvalue) // Process input table

{

if(maxvalue==null)

{

return true;

}

if(output == null)

{

output = new IntWritable(maxvalue.get());

}

else

{

output.set(Math.max(output.get(), maxvalue.get()));

}

return true;

}

public IntWritable terminatePartial()

{

return output;

}

public boolean merge(IntWritable other)

{

return iterate(other);

}

public IntWritable terminate() //final result

{

return output;

}

}

}

* **UDTF:**

UDTF is a User Defined Table Generating Function that operates on a single row and produces multiple rows a table as output.

Lets suppose that we would like to create a cleaner table of peoples’ names. The new table will have:

Separate columns for First Name and Surname.

No records that do not contain both first and last names (have no separating white space).

Separate rows for each person in a couple (eg Nick and Nicole Smith).

To accomplish this goal, we will implement the org.apache.hadoop.hive.ql.udf.generic.GenericUDTF API.

We have to override 3 methods:

// in this method we specify input and output parameters: input ObjectInspector and an output struct

abstract StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException;

// here we process an input record and write out any resulting records

abstract void process(Object[] record) throws HiveException;

// this function is Called to notify the UDTF that there are no more rows to process. Clean up code or additional output can be produced here.

abstract void close() throws HiveException;

The UDTF takes string as a parameter and returns a struct with two fields. Similarly to the GenericUDF, we have to manually configure all of the input and output object inspectors Hive needs in order to understand the inputs and outputs.

We identify a PrimitiveObjectInspector for the input string.

stringOI = (PrimitiveObjectInspector) args[0]

Defining the output object inspectors requires us to define both field names, and the object inspectors required to read each field (in our case, both fields are strings).

List<String> fieldNames = new ArrayList<String>(2);

fieldNames.add("name");

fieldNames.add("surname");

List<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>(2);

fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);

fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);

return ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames, fieldOIs);

The bulk of our logic resides in the processInputRecord function which is fairly straightforward. Separating our logic allows easier testing without having to struggle with object inspectors.

Finally, once we have the result we can forward it, this registers that object as an output record for Hive to process.

while (it.hasNext()){

Object[] r = it.next();

forward(r);

}

}

Using our function

We can build our function and use it in Hive

mvn package

cp target/hive-extensions-1.0-SNAPSHOT-jar-with-dependencies.jar ./ext.jar

Then use it from hive

ADD JAR ./ext.jar;

CREATE TEMPORARY FUNCTION process\_names as 'com.matthewrathbone.example.NameParserGenericUDTF';

SELECT

adTable.name,

adTable.surname

FROM people

lateral view process\_names(name) adTable as name, surname;

● Thrift server

Apache Thrift is a software framework for scalable cross-language services development, which combines a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java, Python, PHP, Ruby, Perl, C#, JavaScript, Node.js and other languages.

Thrift can be used when developing a web service that uses a service developed in one language access that is in another language.

HiveServer is a service that allows a remote client to submit requests to Hive, using a variety of programming languages, and retrieve results. It is built on Apache Thrift, therefore it is sometimes called as the Thrift server.

In the context of Hive, Java language can be used to access Hive server. The Thrift interface acts as a bridge, allowing other languages to access Hive, using a Thrift server that interacts with the Java client.

Now, let’s look at an example of accessing Hive Server using Thrift in Java.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveJdbcClient

{

private static String driverName = "org.apache.hive.jdbc.HiveDriver";

/\*\*

\* @param args

\* @throws SQLException

\*/

public static void main(String args[]) throws SQLException{

try{

Class.forName(driverName);

}

catch(ClassNotFoundException e){

//TODO Auto-generated catch block

e.printStackTrace();

System.exit(1);

}

//replace "hive" here with the name of the user the queries should run

Connection con                                 = DriverManager.getConnection("jdbc:hive2://localhost:10000/default","acadgild","");

Statement stmt = con.createStatement();

String tableName = "testHiveDriverTable1";

stmt.execute("drop table if exists " +tableName);

stmt.execute("create table " +tableName+ "(key int, value string)");

//show tables

String sql = "show tables " +tableName+ "";

System.out.println("Running: " +sql);

ResultSet res = stmt.executeQuery(sql);

if(res.next()){

System.out.println(res.getString(1));

}

//describe table

sql = "describe " +tableName;

System.out.println("Running: " +sql);

res = stmt.executeQuery(sql);

while(res.next()){

System.out.println(res.getString(1) + "\t" +res.getString(2));

}

}

}

**Code Explanation:**

* In line 6, we are taking a class named **HiveJdbcClient.**
* In line 8, we are declaring a private static string variable named **driverName**, which will store the string **“org.apache.hive.jdbc.HiveDriver”** .
* In line 14, we are declaring a try catch block.
* In line 15, the **Class.forName**(**driverName**) method returns the Class object associated with the class or interface with the given string name, using the given class loader.
* Line 17 throws an error ClassNotFoundException, if the **driverName** class not found and exits the program.
* In line 23, we are trying to establish a connection with hive server where **localhost:10000** is the Hive server port number and **acadgild** is the password of the url **localhost:10000**.
* In line 24, we are using createstatement() method to create a statement instance for sending SQL statements to the database. Here, Statement is an interface that represents an SQL statement.
* In line 25, we are declaring a String variable named **tableName,**which will store the string **“testHiveDriverTable1**“.
* In line 26, in order to execute an SQL query, we should use execute method of the interface statement. Here “drop table if exists” is a statement which compares the table name and drops the table if it already exists in the Hive server default database.
* In line 27, we are creating a table named **testHiveDriverTable1**and its columns as key and value and there data types are int and string, respectively.
* In line 29, we are declaring a string named **sql,**where we are storing the value as the command **show tables** with the table name **testHiveDriverTable1**.
* In line 30, we are printing the string **sql**variable value, command **show tables** with the table name **testHiveDriverTable1**.
* In line 31, we are declaring a ResultSet interface object **res**and storing show tables command value of String variable **sql** into ResultSet interface object **res**.
* In line 32, we are declaring an if condition; if interface **res** has next value, it will print the table present in the hive server.
* In line 33, we are storing the string “**describe**” with the table name **testHiveDriverTable1**into the string variable **sql**.
* In line 36, we are printing the string **sql** variable value.
* In line 38, we are executing SQL command **describe** on the table name **testHiveDriverTable1**and to store that table contents into the ResultSet interface object **res**.
* In line 49, we are declaring a while condition until Object **res** has next value performs the operation within the while loop.
* Line 50 prints the tables present in the Hive server table using show tables command and print the description of the existing table