**ASSIGNMENT29.3**

**UDF IN HIVE :**

Hive provides some built in functions which can be used to take a column as **input and these functions work on single row as Input for eg SUBSTR in hive**.

Suppose if we want to perform some function say converting a column to upper case etc ; there is no built in functions for such operation

So we can create a User Defined Function which perform this operation WHICH EXTENDS udf class

During query processing, an instance of the class is instantiated for each usage of the function in a query.

The evaluate() is called for each input row.

The result of evaluate() is returned to Hive.

It is legal to overload the evaluate method.

Hive will pick the method that matches in a similar way to Java method overloading.

Finally to use UDF, create jar and register the class as temporary function.

**import** org.apache.hadoop.hive.ql.exec.UDF;

**public** **class** ToUpper **extends** UDF{

**public** String evaluate (String input) {

**if** (input == **null**) {

**return** **null**;

}

**return** input.toUpperCase();

}

Similarly we can create a file and export it as Jar and register in hive and create temporary function which can be used

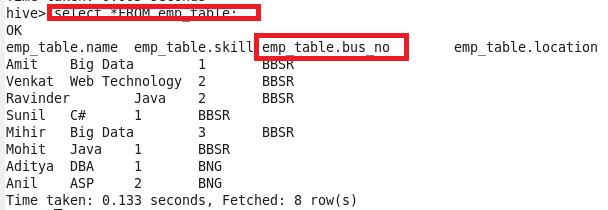
**Hive UDAF**

**User defined aggregate functions works on more than one row and gives single row as output.** e.g Hive built in MAX() or COUNT() functions. here the relation is many to one. Lets say you have a table with students name, id and total marks, so here if I have 10 rows in the table and if I have to find student who got maximum number then our query need to check each 10 row to find the maximum but ultimately we get only one output which is the maximum. Hope this justifies the many to one relationship.

For eg here I am working on a entire column and perform operation

Here we are finding the largest for example **I am having a employee dataset and I want to find a largest bus no**

Input :



**Code**

Since we are creating Udf we create a **hivemax class** that extends UDAF where the output variable is defined

then we are creating static class maxIntUDAFEvalator implements UDAFEvaluator(The actual class for doing the aggregation. Hive will automatically look for all internal classes of the UDAF that implements UDAFEvaluator.) which has the following methods

**init() method**

where the output variable will be initialized

**Iterate methdod()**

**Iterate through one row of original data. The number and type of arguments need to the same as we call this UDAF from Hive command line. This function should always return true.**

Here we are actually checking for the first time when output contains null so we are getting maxvalue

Now when it already contains value a comparison is made using **Math.max()** and it is set to output

**TerminatePartial()**

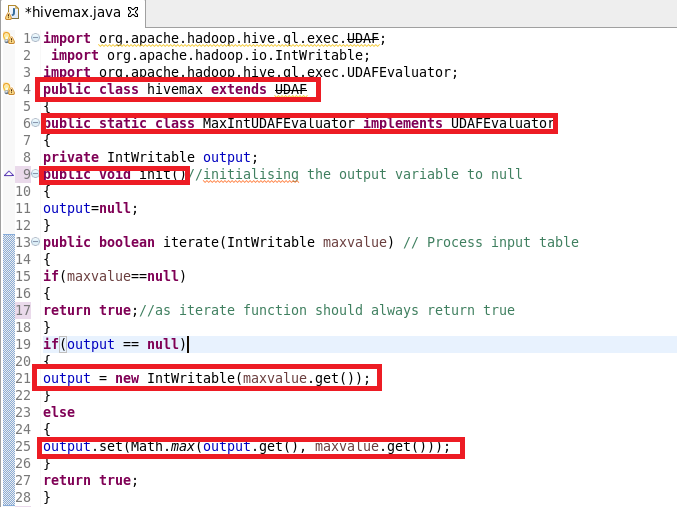
Terminate a partial aggregation and return the state. If the state is a primitive, just return primitive Java classes like Integer or String.

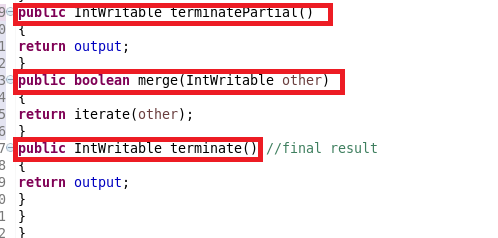
**terminatePartial()**

Merge with a partial aggregation. This function should always have a single argument which has the same type as the return value of terminatePartial().

**Terminate()**

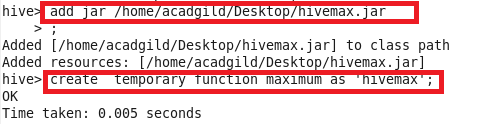
Terminates the aggregation and return the final result.





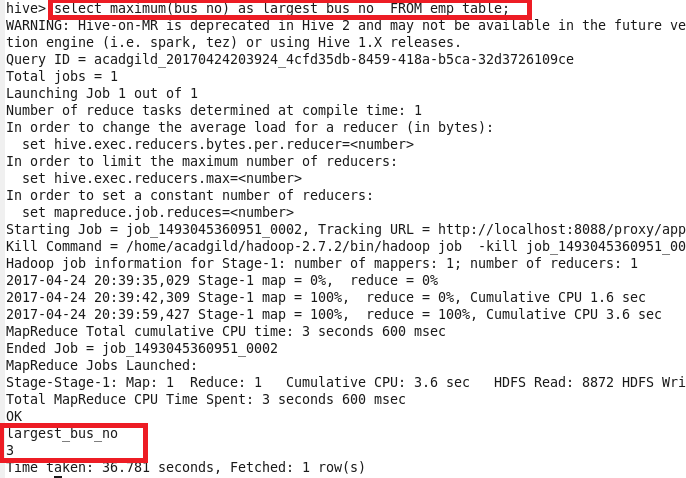
**Registering JAR:**

**ADDED THE JAR FILE AND CREATED A TEMPORARY FUNCTION AS SHOWN**

****

**OUTPUT**

**The Output is verified from the above input**

****

**UDTF IN HIVE**

UDTF:  User defined tabular function works on one row as input and returns multiple rows as output. So here the relation in one to many. e.g Hive built in EXPLODE() function. Now lets take an array column USER\_IDS as ARRAY<10,12,5,45> then SELECT EXPLODE(USER\_IDS) as ID FROM T\_USER. will give 10,12,5,45 as four different rows in output. UDTF can be used to split a column into multiple column as well which we will look in below example. Here alias "AS" clause is mandatory .

**Example:**

**We have a table where there is a column which contains first name and last name in a single column and I am using custom UDTF to split it into 2 columns first name and last name as shown below**

public class NameParserGenericUDTF extends GenericUDTF {

private PrimitiveObjectInspector stringOI = null;

@Override

public StructObjectInspector initialize(ObjectInspector[] args) UDFArgumentException {

if (args.length != 1) {

throw new UDFArgumentException("NameParserGenericUDTF() takes exactly one argument");

}

if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE

&& ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {

throw new UDFArgumentException("NameParserGenericUDTF() takes a string as a parameter");

}

// input inspectors

stringOI = (PrimitiveObjectInspector) args[0];

// output inspectors -- an object with two fields!

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

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

fieldNames.add("name");

fieldNames.add("surname");

fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);

fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);

return ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames, fieldOIs);

}

public ArrayList<Object[]> processInputRecord(String name){

ArrayList<Object[]> result = new ArrayList<Object[]>();

// ignoring null or empty input

if (name == null || name.isEmpty()) {

return result;

}

String[] tokens = name.split("\\s+");

if (tokens.length == 2){

result.add(new Object[] { tokens[0], tokens[1] });

}else if (tokens.length == 4 && tokens[1].equals("and")){

result.add(new Object[] { tokens[0], tokens[3] });

result.add(new Object[] { tokens[2], tokens[3] });

}

return result;

}

@Override

public void process(Object[] record) throws HiveException {

final String name = stringOI.getPrimitiveJavaObject(record[0]).toString();

ArrayList<Object[]> results = processInputRecord(name);

Iterator<Object[]> it = results.iterator();

while (it.hasNext()){

Object[] r = it.next();

forward(r);

}

}

@Override

public void close() throws HiveException {

// do nothing

}

}

**THRIFT SERVER**

Let’s consider a scenario, where the user is looking forward to performing an operation on Hive server, and the [Hadoop cluster](https://acadgild.com/big-data/big-data-development-training-certification) or Hive software setup is not installed in his/her system. The solution for the above scenario is that the user can write codes in other languages and access Hive server using Apache Thrift interface.

In this post, we will learn about the concept of Thrift and the working of Hive Thrift server using code sample of Java for accessing the Hive server.

what is Apache Thrift?

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.

When should you use Thrift?

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

What is a HiveServer?

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.