**Assignment 29.3**

**Problem Statement:**

Explain Brief of the following in brief

● Hive UDF

● Hive UDAF

● Hive UDTF

● Thrift server

**Answer:**

At times, the query the user wants to write can’t be expressed easily (or at all) using the built-in functions provided by Hive. Under such scenarios, Hive allows a user to write his/her own code known as UDFs, to process and invoke results through Hive queries.

There are three types of UDFs (User Defined Functions) in Hive, that can be used by users and they are as follows:

* Regular UDF
* UDAF – User Defined Aggregation Function
* UDTF – User Defined Tabular Function
* **UDF(user\_defined function):**

UDF is a user-defined function that takes a single input value and produces a single output value. When used in a query, we can call it once for each row in the result set.

Example:

input.toString().toUpperCase();

input.toString().toLowerCase();

The above methods will convert a string of lowercase to uppercase and vice versa.

**Steps to execute udf:**

hive> ADD JAR /home/workspace/pudaf.jar;

hive> CREATE TEMPORARY FUNCTION Func AS 'udfclass\_name';

hive> select id, Func(amount) from orders group by id;

* **UDAF(user\_defined aggregate function):**

UDAF is a user-defined aggregate function (UDAF) that accepts a group of values and returns a single value. Users can implement UDAFs to summarize and condense sets of rows in the same style as the built-in COUNT, MAX(), SUM(), and AVG() functions.

How to write UDAF?

Create Java class which extends org.apache.hadoop.hive.ql.exec.hive.UDAF;

Create Inner Class which implements UDAFEvaluator

Implment five methods ()

init() – The init() method initalizes the evaluator and resets its internal state. We are using new Column() in code below to indicate that no values have been aggregated yet.

iterate() – this method is called everytime there is anew value to be aggregated. The evaulator should update its internal state with the result of performing the agrregation (we are doing sum – see below). We return true to indicate that input was valid.

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.

merge() – this method is called when Hive decides to combine one partial aggregation with another.

terminate() – this method is called when the final result of the aggregation is needed.

Compile and Package JAR

CREATE TEMPORARY FUNCTION in hive CLI

Run Aggregation Query – Verify Output

For example,

Meanudaf.java

package org.hardik.letsdobigdata;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

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

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

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

import org.apache.hadoop.hive.ql.metadata.HiveException;

import org.hardik.letsdobigdata.MeanUDAF.MeanUDAFEvaluator.Column;

@Description(name = "Mean", value = "\_FUNC(double) - computes mean", extended = "select col1, MeanFunc(value) from table group by col1;")

public class Meanudaf extends UDAF {

static final Log LOG = LogFactory.getLog(MeanUDAF.class.getName());

public static class MeanUDAFEvaluator implements UDAFEvaluator {

public static class Column {

double sum = 0;

int count = 0;

}

private Column col = null;

public MeanUDAFEvaluator() {

super();

init();

}

public void init() {

LOG.debug("Initialize evaluator");

col = new Column();

}

public boolean iterate(double value) throws HiveException {

LOG.debug("Iterating over each value for aggregation");

if (col == null)

throw new HiveException("Item is not initialized");

col.sum = col.sum + value;

col.count = col.count + 1;

return true;

}

public Column terminatePartial() {

LOG.debug("Return partially aggregated results");

return col;

}

public boolean merge(Column other) {

LOG.debug("merging by combining partial aggregation");

if(other == null) {

return true;

}

col.sum += other.sum;

col.count += other.count;

return true;

}

public double terminate(){

LOG.debug("At the end of last record of the group - returning final result");

return col.sum/col.count;

}

}

}

hive> ADD JAR /home/cloudera/workspace/pudaf.jar;

hive> CREATE TEMPORARY FUNCTION MeanFunc AS 'Meanudaf';

Verify Output

Execute below group by query, our function is called MeanFunc

hive> select customer\_id, MeanFunc(amount) from orders group by customer\_id;

* **Hive udtf(user\_defined table generating function):**

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

We can create a custom Hive UDTF by extending the GenericUDTF abstract class and then implementing the initialize, process, and possibly close methods.

package com.Myhiveudtf;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

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

import org.apache.hadoop.hive.ql.metadata.HiveException;

import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;

import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;

import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;

import org.apache.hadoop.hive.serde2.objectinspector.PrimitiveObjectInspector;

import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;

import org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInspectorFactory;

public class Myudtf extends GenericUDTF {

private PrimitiveObjectInspector stringOI = null;

public StructObjectInspector initialize(ObjectInspector[] args) throws 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");

}

stringOI = (PrimitiveObjectInspector) args[0];

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

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

fieldNames.add("id");

fieldNames.add("phone\_number");

fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);

fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);

return ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames, fieldOIs);

}

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

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

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

return result;

}

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

if (tokens.length == 2){

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

}

else if (tokens.length == 3){

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

result.add(new Object[] { tokens[0], tokens[2]});

}

return result;

}

Initialize()

The Hive calls the initialize method to notify the UDTF the argument types to expect. The UDTF must then return an object inspector corresponding to the row objects that the UDTF will generate.

@Override

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

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

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

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

while (it.hasNext()){

Object[] r = it.next();

forward(r);

}

}

Process()

Once initialize() method has been called, Hive will give rows to the UDTF using the process() method. While in process() function, the UDTF can produce and forward rows to other operators by calling forward() method.

public void close() throws HiveException {

}

}

Close()

Finally, Hive will call the close() method when all the rows have passed to the UDTF. This function allows for any cleanup that is necessary before returning from the User Defined Table Generating Function. It is important to note that we cannot write any records from this function.

Steps for Executing Hive UDTF:

Step 1: After writing the above code in Eclipse, add the below mentioned jar files in the program and then export it in the Hadoop environment as a jar file.

Step 2: Create a table named ‘phone’ with a single column named ‘id’.

Step 3: Load the input data set phn\_num contents into the table phone.

Step 4: Check if the data contents are loaded or not, using select statement.

Step 5: Add the jar file with the complete path of the jar made as shown above.

Step 6: Create a temporary function as shown below.

Step 7: Use the select statement to populate the above table of strings with its primary id.

From the above screenshot, we can see that we have populated a single column, which contains multiple values to its primary id.

* **Thrift server:**

Apache Hive is a data warehouse software that facilitates reading, writing and managing large data sets residing in distributed storage using SQL.

Let’s consider a scenario, where the user is looking forward to performing an operation on Hive server, and the Hadoop cluster 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.

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.