# USING HIVE ADVANCED USER DEFINED FUNCTIONS WITH GENERIC AND COMPLEX DATA TYPES

Previously we wrote how to write user defined functions that can be called from Hive. You can write these in Java or Scala. (Python does not work for UDFs per se. Instead you can use those with the Hive TRANSFORM operation.)

Programs that extend **org.apache.hadoop.hive.ql.exec.UDF** are for primitive data types, i.e., int, string. Etc. If you want to process complex types you need to use **org.apache.hadoop.hive.ql.udf.generic.GenericUDF**. Complex types are array, map, struct, and uniontype.

Generic functions extend org.apache.hadoop.hive.ql.udf.generic.GenericUDF and implement the 4 interfaces shown below.

```
class MapUpper extends GenericUDF {override def initialize(args: Array):
ObjectInspector = {
}override def getDisplayString(arg0: Array ) : String = { return "silly me";
}override def evaluate(args: Array): Object = {}
```

This is the same as the simple UDF code, except there are two additional functions: **initialize** and **getDisplay**. Those set up an **ObjectInspector** and display a message if there is an error.

**initialize** looks at the value passed from Hive SQL to the function. There you check the argument count and type. Then it determines the type of argument that was passed to it. Then the evaluate

function uses that typeless-argument, which is contained in **org.apache.hadoop.hive.ql.udf.generic.GenericUDF.DeferredObject**.

As you can see from the Scala code above, that returns an object of type **Object**, meaning there is no type definition and no ability for the compiler to find errors (Thus will show up at runtime.).

The **initialize** function returns the type of argument expected by the **evaluate** function.

# Create Some Hive Map Data

We do not write a complete code example here. Instead we explain how you would set up to write a GenericUDF with a Map data type and give the general code outline above.

First, we create some data of Hive Map type. Run Hive and then execute:

```
create table students (student map<string,string>) ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',' COLLECTION ITEMS TERMINATED BY '-' MAP KEYS
TERMINATED BY ':'
LINES TERMINATED BY '\n';
```

This creates a table with one column: a map.

That will then let you parse a line of text line this:

```
name:Walker-class:algebra-grade:B-teacher:Newton
```

Then you can load that into Hive like this:

```
load data inpath '/home/walker/Documents/hive/students.txt' into table
students;
```

Which will then produce this output:

```
select * from students;
OK
{"name":"Walker","class":"algebra","grade":"B","teacher":"Newton"}
```

As you can see, each column is a (key->value) map.

Note that you can only load data into a Map column type using something like that. The Hive documentation makes clear that you cannot add values to a Map using SQL:

> "Hive does not support literals for complex types (array, map, struct, union), so it is not possible to use them in INSERT INTO...VALUES clauses. This means that the user cannot insert data into a complex datatype column using the INSERT INTO...VALUES clause."

# Run Program in Hive

The way you run a program like this in Hive is to make these Hive jar file available to Hive by setting the classpath to:

```
export CLASSPATH=/usr/local/hive/apache-hive-2.3.0-bin/lib/hive-
exec-2.3.0.jar:/usr/hadoop/hadoop-2.8.1/share/hadoop/mapreduce/hadoop-
mapreduce-client-core-2.8.1.jar:/home/walker/Documents/bmc/hadoop-
```

```
common-2.8.1.jar
```

All of those are contained in Hive and Hadoop lib folders, except for hadoop-common, which you download from Maven Central.

# Add Jar to Hive

After you have written and compiled your program you put it in a jar file. Then in Hive you make it available using, where MapUpper is the name of the example we use here:

```
add jar
/home/walker/Documents/bmc/udf/target/scala-2.12/mapupper_2.12-1.0.jar;create
temporary function MapUpper as 'MapUpper';
```

Then you can run this command to execute the MapUpper function against the **student** column in the **students** table.

```
select MapUpper(student) from students;
```

This will run some operation on the keys or values and return a new map. Or it could return a primitive type if that is what you need.