# UDTF

User defined tabular function works on one row as input and returns multiple rows as output.

So here the relation is one to many. e.g Hive built in EXPLODE() function.

Now let’s 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.

## Example-1

Create a UDTF that split comma separated value and show them row wise

|  |
| --- |
| create table books ( col1 string) |

|  |
| --- |
| insert into table test values ('1,2,3,4,5'); |

Java code

|  |
| --- |
| package org.cyb.UDTF;  import java.util.ArrayList;  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 StringExplodeUDTF extends GenericUDTF {  private PrimitiveObjectInspector stringOI = null;  private Object[] fwdObj = null;  @Override  public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException{  if (args.length != 1) {  throw new UDFArgumentException("StringExplodeUDTF() takes exactly one argument");  }  ArrayList<String> fieldNames = new ArrayList<String>();  ArrayList<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>();    stringOI = (PrimitiveObjectInspector) args[0];  fieldNames.add("col");  fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);    return ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames,  fieldOIs);      }  @Override  public void close() throws HiveException {  // TODO Auto-generated method stub    }  @Override  public void process(Object[] record) throws HiveException {    String value = stringOI.getPrimitiveJavaObject(record[0]).toString();    String [] valuearray = value.split(",");  fwdObj = new Object[valuearray.length];    for (String val: valuearray) {  fwdObj[0] = val;  this.forward(fwdObj);  }    }  } |

Create jar, Add jar file and Create function

|  |
| --- |
| hive> **add jar /home/shalaj/UDTF.jar** ;  Added [/home/shalaj/UDTF.jar] to class path  Added resources: [/home/shalaj/UDTF.jar]  hive> **CREATE TEMPORARY FUNCTION stringExplode as 'org.cyb.UDTF.StringExplodeUDTF'**;  OK  Time taken: 1.671 seconds |

To Create permanent function upload jar on hdfs and create function like below

|  |
| --- |
| **CREATE FUNCTION stringExplode AS 'org.cyb.UDTF.StringExplodeUDTF' USING JAR 'hdfs://mac127:8020/user/shalaj/udfs/UDTF.jar';** |

Hdfs location is mandatory here

Now run the query

|  |
| --- |
| hive> select stringExplode(col1) from test;  1  2  3  4  5 |

Or use lateral view

|  |
| --- |
| hive> select mycol1 from test lateral view stringExplode(col1) mytable1 as mycol1;  1  2  3  4  5 |

That is how we can get many rows from one column using UDTF

## Example-2

UDTF can be used to split a column into multiple column as well which we will look in below example.**.**

**books.data**

|  |
| --- |
| 1001 6438391,book1-title,Author1  1002 5288292,book2-title,Author2  1003 6878195,book3-title,Author3 |

Table Script

|  |
| --- |
| create table books (  book\_id string,  book\_detail string)  row format delimited  fields terminated by "\t"; |

Load data into table

|  |
| --- |
| load data local inpath '/home/shalaj/book.data' into table books; |

**A Practical Example**

Let’s suppose that we would like to create a cleaner table of books.the new table will have:

* Separate columns for ISBN, Title and Author.

**Java code**

|  |
| --- |
| package org.cyb.UDTF;  import java.util.ArrayList;  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.PrimitiveObjectInspector.PrimitiveCategory;  import org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInspectorFactory;  public class ExpandBookDetail extends GenericUDTF{  private Object[] fwdObj = null;  private PrimitiveObjectInspector bookDtlOI = null;    public StructObjectInspector initialize(ObjectInspector[] arg)  {  ArrayList<String> fieldNames = new ArrayList<String>();  ArrayList<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>();    bookDtlOI = (PrimitiveObjectInspector) arg[0];    fieldNames.add("ISBN");  fieldOIs.add(PrimitiveObjectInspectorFactory.getPrimitiveJavaObjectInspector(PrimitiveCategory.INT));    fieldNames.add("TITLE");  fieldOIs.add(PrimitiveObjectInspectorFactory.getPrimitiveJavaObjectInspector(PrimitiveCategory.STRING));    fieldNames.add("AUTHOR");  fieldOIs.add(PrimitiveObjectInspectorFactory.getPrimitiveJavaObjectInspector(PrimitiveCategory.STRING));    fwdObj = new Object[3];  return ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames, fieldOIs);  }    public void process(Object[] record) throws HiveException  {  String bookDtl = bookDtlOI.getPrimitiveJavaObject(record[0]).toString();    String str[] = bookDtl.split(",");  fwdObj[0] = Integer.parseInt(str[0]);  fwdObj[1] = str[1];  fwdObj[2] = str[2];    this.forward(fwdObj);    }    public void close()  {    }  } |

Create jar, Add jar file and Create function

|  |
| --- |
| hive> **Add jar /home/shalaj/BookUDTF.jar**  Added [/home/shalaj/BookUDTF.jar] to class path  Added resources: [/home/shalaj/BookUDTF.jar]  hive> **CREATE TEMPORARY FUNCTION ExpandBookDetails as 'org.cyb.UDTF.ExpandBookDetail';**  OK  Time taken: 1.65 seconds  hive> |

|  |
| --- |
| hive> select \* from books;  OK  **books.book\_id** **books.book\_detail**  1001 6438391,book1-title,Author1  1002 5288292,book2-title,Author2  1003 6878195,book3-title,Author3 |

|  |
| --- |
| select ExpandBookDetails(book\_detail) as (ISBN,Title,Author) from books;  **isbn title author**  6438391 book1-title Author1  5288292 book2-title Author2  6878195 book3-title Author3 |

**Using lateral view**

|  |
| --- |
| Select a.isbn,a.title,a.author from books **lateral view** ExpandBookDetails(book\_detail) a as isbn,title,author  **a.isbn a.title a.author**  6438391 book1-title Author1  5288292 book2-title Author2  6878195 book3-title Author3 |

## Example-3

The table that will be used for demonstration is called people. It has one column - name, which contains names of individuals and couples.

It is stored in a file called **people.txt**

|  |
| --- |
| Shalaj Shukla  Rishi and Neetu Kapur  Siddhartha and Shreya Miglani  Tom Hank  Sachin and Anjali Tendulkar  Deepak |

Create people table in hive

|  |
| --- |
| create table people (name string)  ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'; |

Load data into table

|  |
| --- |
| hive> load data local inpath '/home/shalaj/people.txt' into table people; |

**A Practical Example**

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

UDTF Java code

|  |
| --- |
| **package** org.cyb.UDTF;  **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** NameParserGenericUDTF **extends** GenericUDTF {  **private** PrimitiveObjectInspector stringOI = **null**;  @Override  **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");  }    // input  stringOI = (PrimitiveObjectInspector) args[0];  // output  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 ==1) {  result.add(**new** Object[] {tokens[0],""});  }**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  }  } |

Create jar, Add jar file and Create function

|  |
| --- |
| hive> **Add jar /home/shalaj/NameParserUDTF.jar**  Added [/home/shalaj/NameParserUDTF.jar] to class path  Added resources: [/home/shalaj/NameParserUDTF.jar]  hive> **CREATE TEMPORARY FUNCTION process\_names as 'org.cyb.UDTF.NameParserGenericUDTF';**  OK  Time taken: 1.65 seconds  hive> |

Now run below query to check existing records

|  |
| --- |
| Hive>select \* from people; |

**Output:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | **people.name** | | --- | --- | | 1 | Shalaj Shukla | | 2 | Rishi and Neetu Kapur | | 3 | Siddhartha and Shreya Miglani | | 4 | Tom Hank | | 5 | Sachin and Anjali Tendulkar | | 6 | Deepak | |

**Query with UDTF function**

|  |
| --- |
| SELECT **process\_names(name)** as (name, surname) from people; |

**Output:**

|  |  |  |
| --- | --- | --- |
|  | **adtable.name** | **adtable.surname** |
| 1 | Shalaj | Shukla |
| 2 | Rishi | Kapur |
| 3 | Neetu | Kapur |
| 4 | Siddhartha | Miglani |
| 5 | Shreya | Miglani |
| 6 | Tom | Hank |
| 7 | Sachin | Tendulkar |
| 8 | Anjali | Tendulkar |
| 9 | Deepak |  |

**Now run query with lateral view**

|  |
| --- |
| SELECT  adTable.name,  adTable.surname  FROM people  **lateral view process\_names(name)** adTable as name, surname; |

**Output:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | **adtable.name** | **adtable.surname** | | --- | --- | --- | | 1 | Shalaj | Shukla | | 2 | Rishi | Kapur | | 3 | Neetu | Kapur | | 4 | Siddhartha | Miglani | | 5 | Shreya | Miglani | | 6 | Tom | Hank | | 7 | Sachin | Tendulkar | | 8 | Anjali | Tendulkar | | 9 | Deepak |  | |

Both queries have same results