WE HAVE SEEN THAT HIVE HAS A WIDE VARIETY OF BUILT-IN FUNCTIONS

HIVE ALSO HAS THE ABILITY TO ALLOW USERS TO DEFINE CUSTOM FUNCTIONS

HIVE CUSTOM FUNCTIONS

THIS IS SUPER COOL

IT ALLOWS USERS A LOT OF CONTROL OVER DATA PROCESSING TASKS

HIVE CUSTOM FUNCTIONS

ANYBODY CAN WRITE AND REGISTER A FUNCTION TO HIVE

HIVE CUSTOM FUNCTIONS

THAT FUNCTION IS AVAILABLE FOR THE DURATION OF THE HIVE SESSION

HIVE CUSTOM FUNCTIONS

WHAT ARE SOME CUSTOM FUNCTIONS THAT YOU MIGHT WANT TO WRITE?

REPLACETEXT()

REPLACE ALL OCCURRENCES OF A STRING IN SOME TEXT

HIVE CUSTOM FUNCTIONS

REPLACETEXT()

WHAT ARE SOME CUSTOM FUNCTIONS THAT YOU MIGHT WANT TO WRITE?

CONTAINSSTRING()

CHECK WHETHER A STRING IS PRESENT IN A LIST OF STRINGS

HIVE CUSTOM FUNCTIONS

REPLACETEXT()

CONTAINSSTRING()

WHAT ARE SOME CUSTOM FUNCTIONS THAT YOU MIGHT WANT TO WRITE?

STD()

COMPUTE THE STANDARD DEVIATION OF A COLUMN OF VALUES

HIVE CUSTOM FUNCTIONS

REPLACETEXT()

CONTAINSSTRING()

STD()

YOU CAN IMPLEMENT THE LOGIC FOR THESE CUSTOM FUNCTIONS IN

JAVA

PYTHON

HIVE CUSTOM FUNCTIONS

YOU CAN IMPLEMENT THE LOGIC FOR THESE CUSTOM FUNCTIONS IN

IN JAVA THERE IS A SET OF CLASSES THAT CAN BE USED TO IMPLEMENT CUSTOM FUNCTIONS

JAVA

PYTHON

HIVE CUSTOM FUNCTIONS

THE JAVA CLASSES FOR THE CUSTOM FUNCTIONS DEPEND ON

THE TYPE OF FUNCTION

STANDARD/AGGREGATE/
TABLE GENERATING

THE TYPE OF INPUT/OUTPUT

PRIMITIVE/COLLECTION

HIVE CUSTOM FUNCTIONS

THE TYPE OF FUNCTION

STANDARD

AGGREGATE

TABLE GENERATING

THE TYPE OF INPUT/OUTPUT

PRIMITIVE

COLLECTION

HIVE CUSTOM FUNCTIONS

THE TYPE OF FUNCTION

STANDARD

AGGREGATE

TABLEGENERATINGFUNCTION

THE TYPE OF INPUT/OUTPUT

PRIMITIVE

COLLECTION

THIS IS THE SIMPLEST CUSTOM FUNCTION YOU CAN WRITE

HIVE CUSTOM FUNCTIONS

THE TYPE OF FUNCTION

STANDARD

AGGREGATE

TABLE GENERATING

THE TYPE OF INPUT/OUTPUT

PRIMITIVE

COLLECTION

IT'S CALLED A SIMPLE UDF

SIMPLE UDF

IT'S INPUT AND OUTPUT TYPES ARE BOTH PRIMITIVE

SIMPLE UDF

IT TAKES 1 ROW/ COLUMNS IN A ROW AND RETURNS A RESULT FOR 1 ROW

SIMPLE UDF

THIS FUNCTION SHOULD BE A SUBCLASS OF

UDF

SIMPLE UDF

UDF

UDF STANDS FOR USER DEFINED FUNCTION

SIMPLE UDF

UDF

EVEN HIVE BUILT-IN FUNCTIONS ARE SOMETIMES REFERRED TO AS UDFS BECAUSE THEY DERIVE FROM THIS CLASS

SIMPLE UDF

UDF

EVALUATE()

WITHIN THIS CLASS WE IMPLEMENT JUST 1 METHOD

SIMPLE UDF

UDF

EVALUATE()

THE SIGNATURE OF THIS METHOD WILL BE THE SIGNATURE OF THE CUSTOM HIVE FUNCTION

SIMPLE UDF

UDF

EVALUATE()

IE. THE ARGUMENTS, INPUT AND OUTPUT TYPES OF THE EVALUATE METHOD MATCH THOSE OF THE HIVE FUNCTION

SIMPLE UDF

UDF

EVALUATE()

BECAUSE THIS IS A SIMPLE UDF, IT WILL ONLY WORK WITH PRIMITIVE DATA TYPES

INT, BIGINT, CHAR,VARCHAR ETC

SIMPLE UDF

EACH OF THE PRIMITIVE DATA TYPES MAPS TO A CORRESPONDING HIVE WRITABLE CLASS

INT,
BIGINT,
CHAR,
VARCHAR

HIVE WRITABLES

INTWRITABLE,
LONGWRITABLE,
HIVECHARWRITABLE,
HIVEVARCHARWRITABLE

SIMPLE UDF

EACH OF THE PRIMITIVE DATA TYPES ALSO MAPS TO A CORRESPONDING JAVA PRIMITIVE CLASS

INT,
BIGINT,
CHAR,
VARCHAR

HIVE WRITABLES

INTWRITABLE,
LONGWRITABLE,
HIVECHARWRITABLE,
HIVEVARCHARWRITABLE

JAVA PRIMITIVES

INTEGER,
LONG ETC

SIMPLE UDF

UDF

EVALUATE()

THE INPUT/OUTPUT TYPES OF THE EVALUATE METHOD MUST BELONG TO ONE OF THESE PRIMITIVE CLASSES

HIVE WRITABLES

INTWRITABLE,
LONGWRITABLE,
HIVECHARWRITABLE,
HIVEVARCHARWRITABLE

JAVA PRIMITIVES

INTEGER,
LONG ETC

SIMPLE UDF

ALL THESE HIVE I/O CLASSES ARE PART OF A PACKAGE CALLED SERDE

INTWRITABLE,
LONGWRITABLE,
HIVECHARWRITABLE,
HIVEVARCHARWRITABLE

SIMPLE UDF

SERDE

SERIALIZATION, DESERIALIZATION

SIMPLE UDF

SERDE

THIS PACKAGE TELLS HIVE HOW TO DEAL WITH DIFFERENT DATA TYPES

SIMPLE UDF

SERDE

HOW THE DATA SHOULD BE READ FROM HDFS (DESERIALIZATION)

SIMPLE UDF

SERDE

HOW THE DATA SHOULD BE WRITTEN TO HDFS (SERIALIZATION)

SIMPLE UDF

SERDE

THE OBJECTS IN WHICH THE DATA WILL BE STORED IN MEMORY

SIMPLE UDF

LET'S WRITE A SIMPLE UDF TO REPLACE A STRING WITH ANOTHER STRING

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string
within a text with another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str1, String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a
string within a text with another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM
authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str2, String str3) {

        String rep = str.replace(str2, str3);
        result.set(rep);
        return result;

    }

}
```

WHENEVER YOU CREATE A FUNCTION USE A DESCRIPTION ANNOTATION TO DOCUMENT THE FUNCTION

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a
string within a text with another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM
authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str1, String str2) {

        String rep = str.replaceAll(str1, str2);
        result.set(rep);
        return result;

    }

}
```

THIS WILL BE THE RESULT OF THE DESCRIBE FUNCTION COMMAND FOR THIS FUNCTION

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string
within a text with another string",
        extended = "Example:\n" +
                   "   > SELECT
name,replaceText(name,'abc','def') FROM authors a;\n" +
                   "   abcabc   defdef"
)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str1, String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

THE DETAILS IN THE EXTENDED PARAMETER WILL ONLY SHOW UP IF YOU USE DESCRIBE FUNCTION EXTENDED

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a text with another
string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"

)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str1, String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

THE CLASS EXTENDS THE UDF CLASS

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a
text with another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str1, String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

WE INITIALIZE A VARIABLE TO HOLD THE RESULT

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a
text with another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str1, String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

THIS RESULT'S DATA TYPE WILL BE THE HIVE FUNCTIONS RETURN TYPE

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a
text with another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str1, String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

THE DATATYPE OF THIS RESULT HAS TO BE ONE OF THE PRIMITIVE DATA TYPES

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a
text with another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str1, String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

THERE ARE DIFFERENT CLASSES IN HIVE THAT MAP TO THE PRIMITIVE DATATYPES

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a
text with another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str1, String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

EX: TEXT MAPS TO THE STRING DATATYPE

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a
text with another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str1, String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

EX: LONGWRITABLE MAPS TO BIGINT

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a
text with another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str1, String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

IN SIMPLE UDFS, YOU CANNOT USE ANY COLLECTION DATA TYPES

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a
text with another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str1, String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

YOU CAN ALSO HAVE THE RESULT AS ONLY 1 DATA TYPE

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a
text with another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {
    private Text result = new Text();

    public Text evaluate(String str, String str1, String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

FOR INSTANCE, YOU CANNOT HAVE A FUNCTION THAT RETURNS INTEGER IN CERTAIN CIRCUMSTANCES AND TEXT IN OTHERS

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a text with
another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {

      private Text result = new Text();

      public Text evaluate(String str, String str1,
String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

WE NEED TO IMPLEMENT THE EVALUATE METHOD

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a text with
another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {

      private Text result = new Text();

      public Text evaluate(String str, String str1,
String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

THIS HOLDS THE LOGIC FOR THE FUNCTION

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a text with
another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {

      private Text result = new Text();

      public Text evaluate(String str, String str1,
String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

THE INPUTS, RETURN TYPE DEFINE THE FUNCTION SIGNATURE

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a text with
another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {

      private Text result = new Text();

      public Text evaluate(String str, String str1,
String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

THIS FUNCTION TAKES 3 STRINGS

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a text with
another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {

      private Text result = new Text();

      public Text evaluate(String str, String str1,
String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

IT LOOKS FOR STR1 IN STR AND REPLACES IT WITH STR2

SIMPLE UDF

```
@Description(
        name = "replaceText",
        value = "_FUNC_(text,str1,str2) - replaces all occurrences of a string within a text with
another string",
        extended = "Example:\n" +
                "  > SELECT name,replaceText(name,'abc','def') FROM authors a;\n" +
                "  abcabc  defdef"
)
public class simpleUDF extends UDF {

      private Text result = new Text();

       public Text evaluate(String str, String str1, String str2) {

        String rep = str.replace(str1, str2);
        result.set(rep);
        return result;

    }

}
```

WE IMPLEMENT THIS LOGIC AND RETURN THE RESULT

SIMPLE UDF

ONCE WE HAVE THE FUNCTION CLASS WRITTEN UP

1. CREATE A JAR

```
/Users/swethakolalapudi/udfs.jar;
```

SIMPLE UDF

2. ADD THIS JAR TO THE HIVE SESSION

```
hive> ADD JAR /Users/swethakolalapudi/udfs.jar;
```

SIMPLE UDF

3. CREATE A TEMPORARY FUNCTION USING THIS JAR

```
hive> create temporary function replaceText
      as 'com.handbook2.hive.simpleUDF';
```

THE NAME OF THE FUNCTION

SIMPLE UDF

3. CREATE A TEMPORARY FUNCTION USING THIS JAR

```
hive> create temporary function replaceText
      as 'com.handbook2.hive.simpleUDF';
```

THE FUNCTION WILL BE AVAILABLE UNDER THIS NAME FOR THE DURATION OF THE HIVE SESSION

SIMPLE UDF

3. CREATE A TEMPORARY FUNCTION USING THIS JAR

```
hive> create temporary function replaceText
      as 'com.handbook2.hive.simpleUDF';
```

THE FUNCTION CLASS NAME

SIMPLE UDF

4. USE THE FUNCTION FOR YOUR DATA PROCESSING TASKS!

```
hive> select replaceText("abcabc","abc","def");
```

```
defdef
```

HIVE CUSTOM FUNCTIONS

THE TYPE OF FUNCTION

STANDARD

AGGREGATE

TABLE GENERATING

THE TYPE OF INPUT/OUTPUT

PRIMITIVE

COLLECTION

SIMPLE UDF

HIVE CUSTOM FUNCTIONS

THE TYPE OF FUNCTION

STANDARD

AGGREGATE

TABLE GENERATING

THE TYPE OF INPUT/OUTPUT

PRIMITIVE

COLLECTION

WE CAN WRITE A STANDARD FUNCTION THAT USES ANY DATA TYPE

HIVE CUSTOM FUNCTIONS

THE TYPE OF FUNCTION

STANDARD

AGGREGATE

TABLE GENERATING

THE TYPE OF INPUT/OUTPUT

PRIMITIVE

COLLECTION

SUCH A FUNCTION IS CALLED A GENERIC UDF

GENERIC UDF

UNLIKE A SIMPLE UDF, A GENERIC UDF CAN HAVE ANY DATA TYPE

BOTH PRIMITIVE, COLLECTION

GENERIC UDF

SIMILAR TO THE SIMPLE UDF, IT TAKES IN 1 ROW AND RETURNS 1 ROW

GENERIC UDF

SAY YOU HAD A FUNCTION LIKE

CONTAINSSTRING()

CHECK WHETHER A STRING IS PRESENT IN A LIST OF STRINGS

GENERIC UDF

CONTAINSSTRING()

CHECK WHETHER A STRING IS PRESENT IN A LIST OF STRINGS

YOU CANNOT USE A SIMPLE UDF FOR THIS AS WE NEED TO WORK WITH LISTS

GENERIC UDF

THIS FUNCTION SHOULD BE A SUBCLASS OF

GENERICUDF

GENERICUDF

THIS CLASS IS MUCH MORE DIFFICULT TO WRITE

THERE IS QUITE A BIT OF BOILERPLATE CODE

GENERICUDF

THE BOILERPLATE IS IMPLEMENTED TO MAKE THE DATA PROCESSING MORE EFFICIENT

THE IDEA OF LAZY EVALUATION IS USED

GENERICUDF

LAZY EVALUATION

THE ROWS DATA IS NOT DESERIALIZED IN MEMORY, UNTIL IT USED BY THE CUSTOM FUNCTION

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

OUR CLASS SHOULD OVERRIDE 3 METHODS

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

THE GENERIC UDF USES AN OBJECTINSPECTOR CLASS TO HANDLE THE DATA

OBJECTINSPECTOR

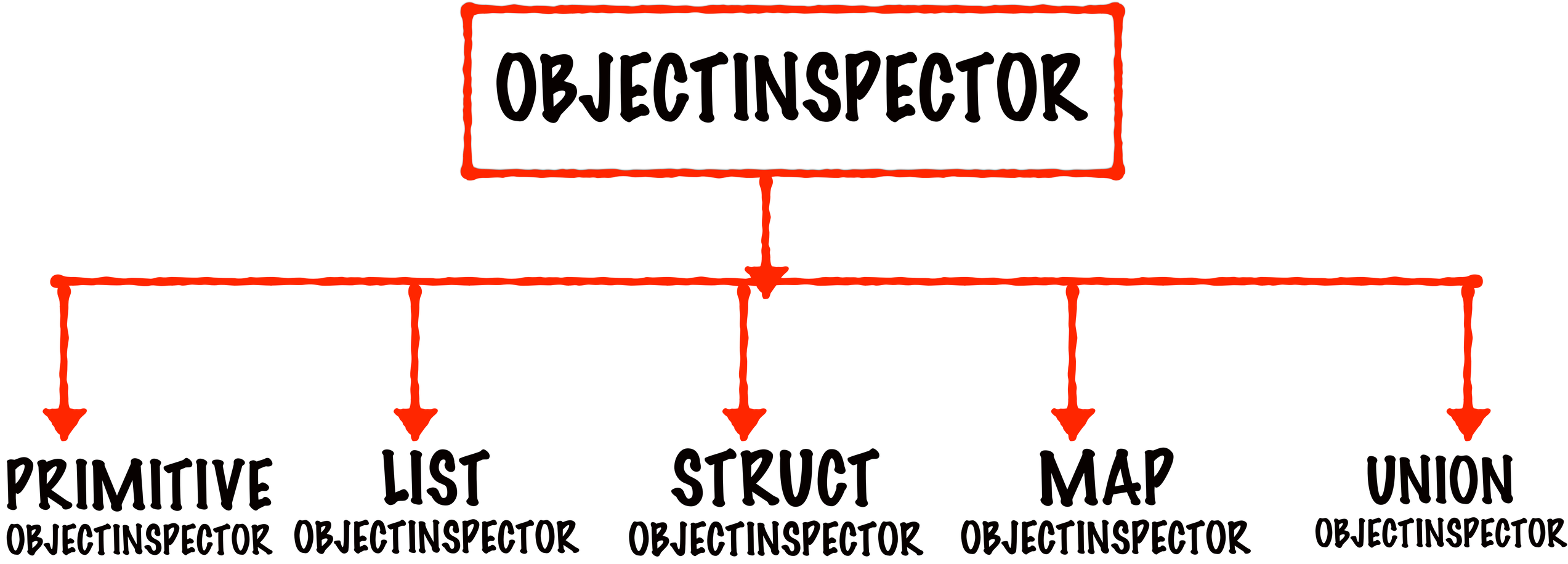
OBJECTINSPECTOR IS AN INTERFACE THAT HELPS US DEAL WITH ANY DATATYPE

OBJECTINSPECTOR

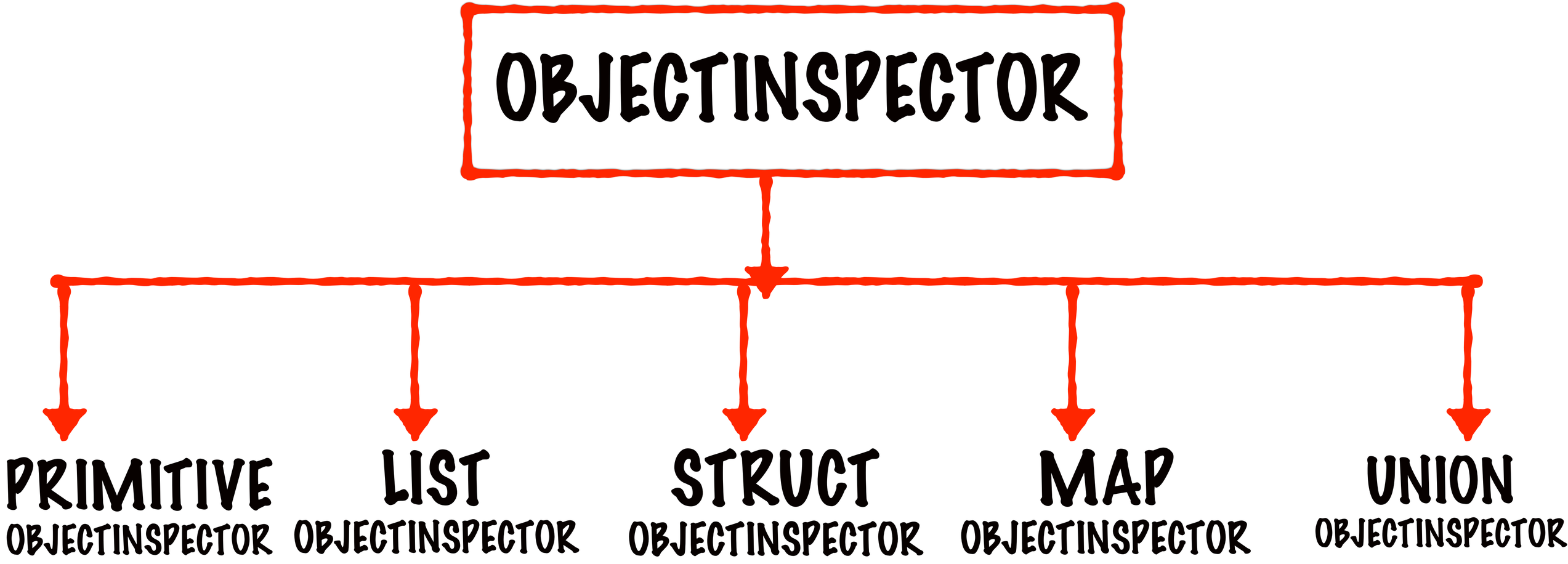
IT HAS METHODS TO

FIND THE DATATYPE OF AN INPUT/OUTPUT OBJECT

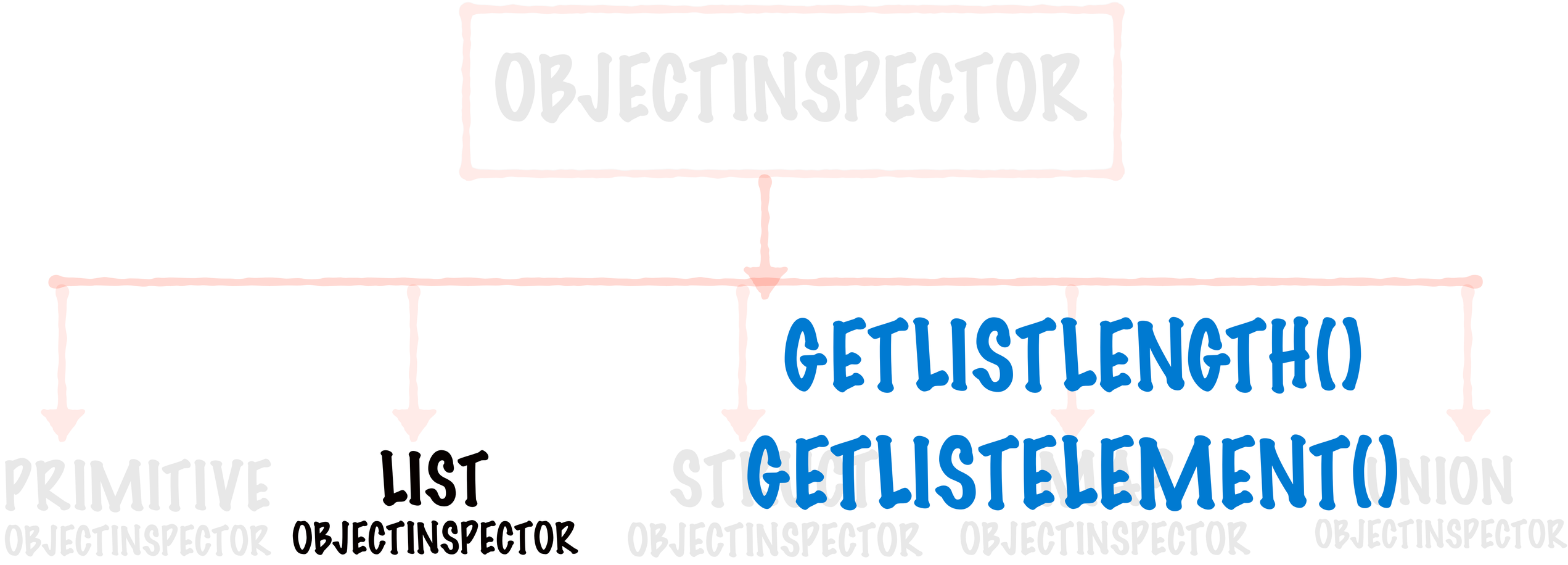
ACCESS THE DATA IN AN INPUT/OUTPUT OBJECT



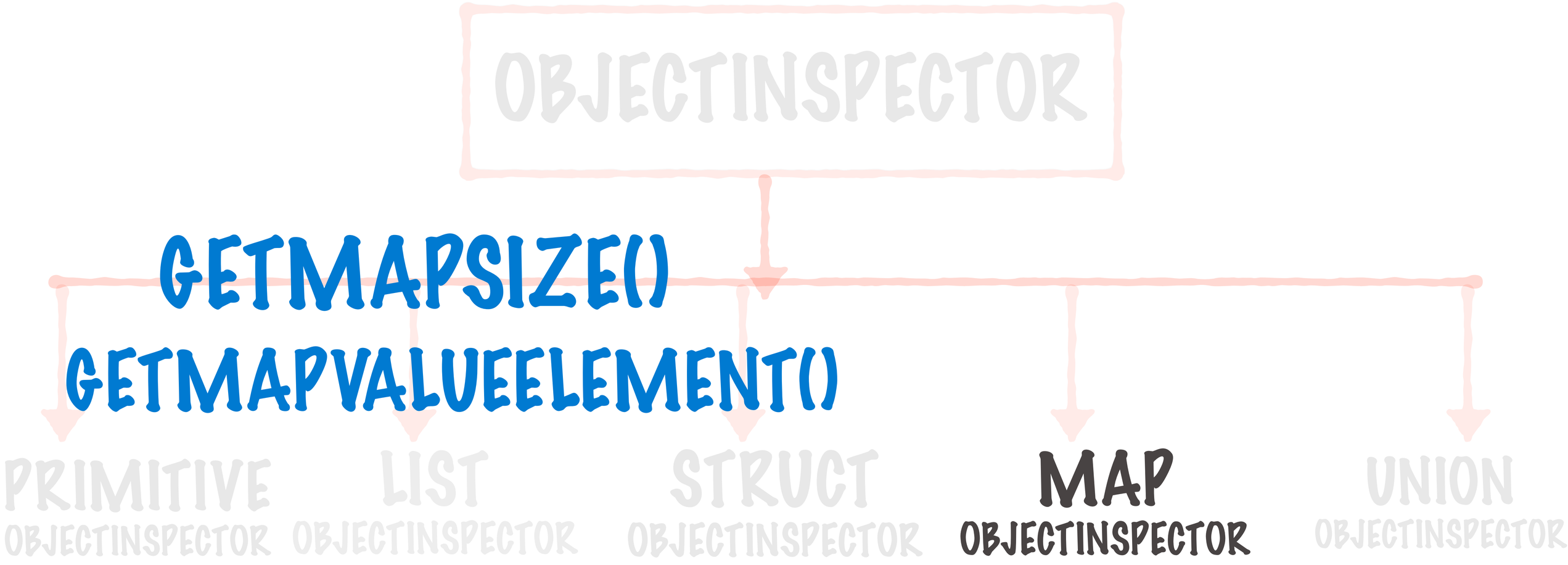
THERE ARE 5 MAJOR SUB INTERFACES OF THE OBJECTINSPECTOR



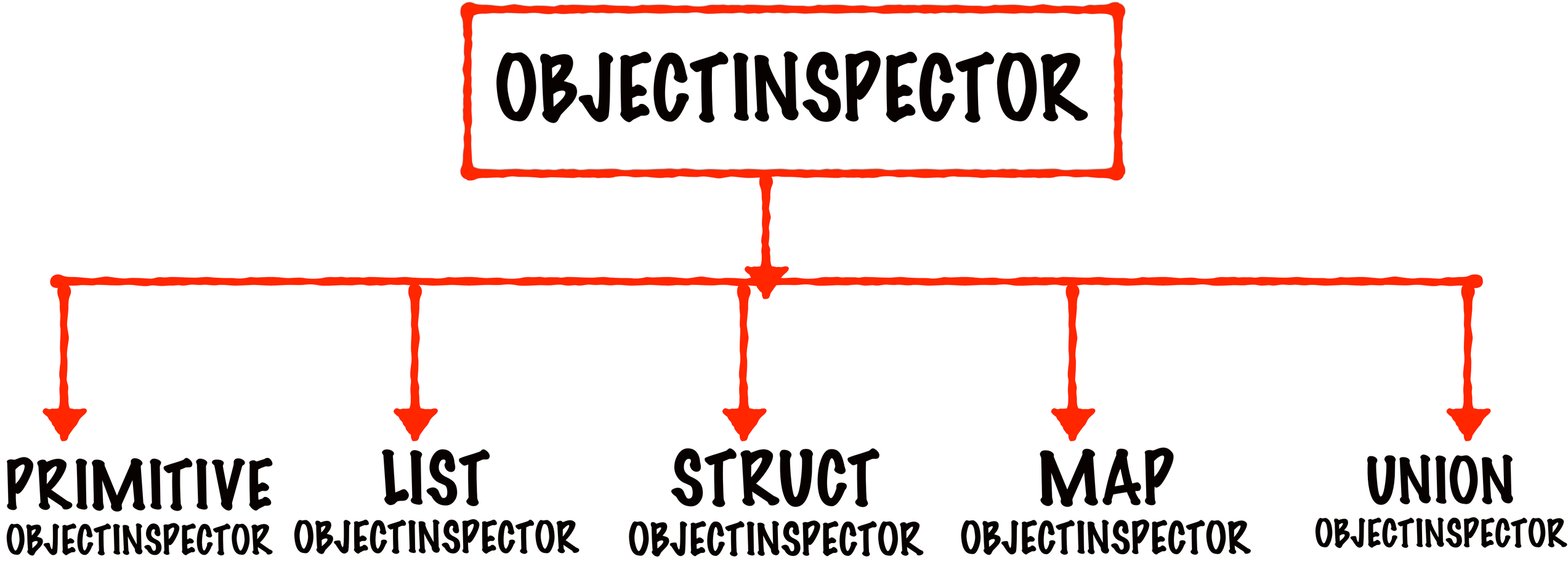
EACH OBJECT INSPECTOR HAS SPECIFIC METHODS TO HANDLE THE DATA OBJECT



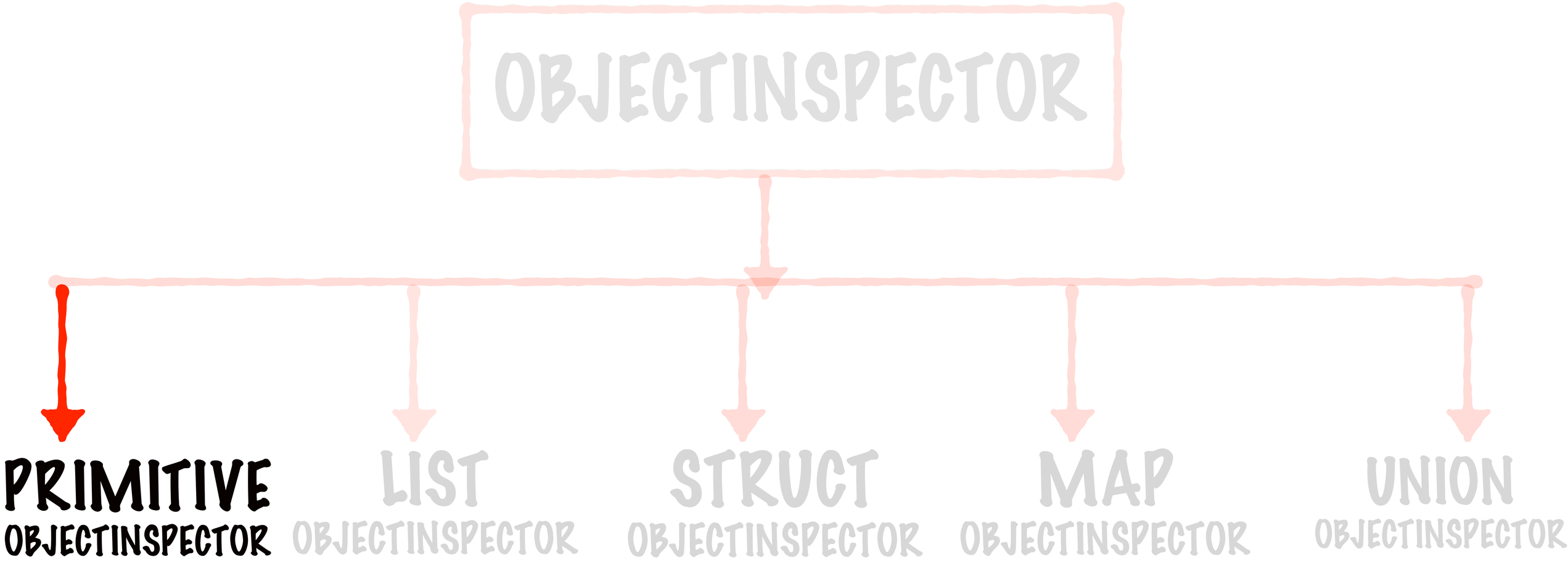
FOR INSTANCE A LISTOBJECTINSPECTOR HAS METHODS TO GET THE LENGTH OF THE LIST AND A SPECIFIC ELEMENT OF THE LIST



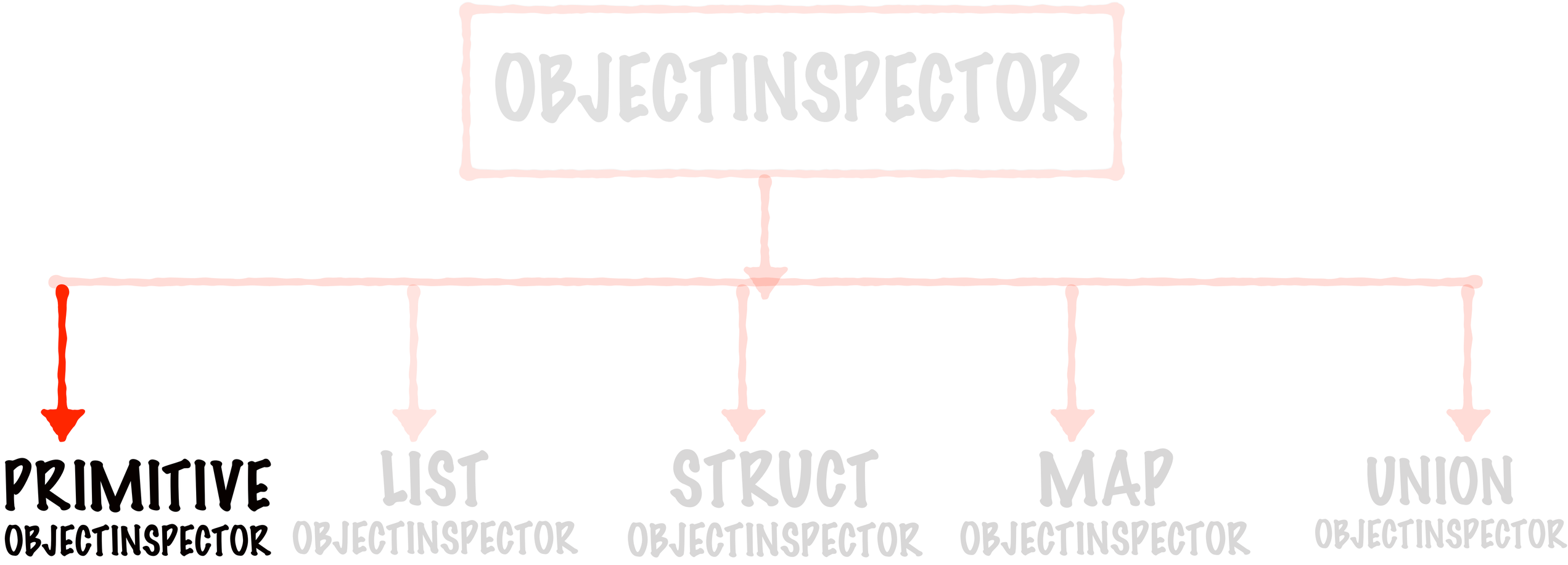
FOR INSTANCE A MAPOBJECTINSPECTOR HAS METHODS TO GET THE MAP SIZE AND VALUES



THE GENERIC UDF CAN USE ANY OF THESE OBJECTINSPECTORS



THE SIMPLE UDF IS RESTRICTED TO USING THE PRIMITIVEOBJECTINSPECTOR



THIS IS WHY SIMPLE UDFS CAN ONLY DEAL WITH PRIMITIVE DATA TYPES

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

THE GENERIC UDF USES AN OBJECTINSPECTOR CLASS TO HANDLE THE DATA

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

LET'S UNDERSTAND THE PURPOSE OF THESE 3 METHODS

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

THE INITIALIZE() METHOD RUNS IN THE BEGINNING WHEN A FUNCTION IS CALLED

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

IT RUNS ONLY ONCE FOR A QUERY AND NOT FOR EVERY ROW IN THE QUERY

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

THIS METHOD HAS 3 PURPOSES

1. VERIFY THE INPUT TYPES AGAINST THE EXPECTED TYPES

2. SET UP OBJECTINSPECTORS FOR THE INPUTS

3. SET UP OBJECTINSPECTORS FOR THE OUTPUT

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

THE OBJECTINSPECTORS SETUP HERE ARE REUSED FOR EACH ROW

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

THE EVALUATE METHOD IS WHERE THE LOGIC FOR THE FUNCTION IS IMPLEMENTED

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

THIS METHOD IS CALLED ONCE FOR EACH ROW

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

THE ROW WILL BE PASSED TO THE METHOD AS A DEFERREDOBJECT ARRAY

GENERICUDF

REPRESENTS THE VALUES IN 1 ROW

INITIALIZE()

EVALUATE(DEFERREDOBJECT[] ARGUMENTS)

GETDISPLAYSTRING()

THESE OBJECTS ARE NOT DESERIALIZED

GENERICUDF

INITIALIZE()

EVALUATE(DEFERREDOBJECT[] ARGUMENTS)

GETDISPLAYSTRING()

THEY ARE ONLY DESERIALIZED IF THE EVALUATE METHOD USES THEM

GENERICUDF

INITIALIZE()

EVALUATE(DEFERREDOBJECT[] ARGUMENTS)

GETDISPLAYSTRING()

THE OBJECTINSPECTORS ARE USED TO EXTRACT THE DATA FROM THESE DEFERRED OBJECTS

GENERICUDF

INITIALIZE()

EVALUATE(DEFERREDOBJECT[] ARGUMENTS)

GETDISPLAYSTRING()

THE LOGIC IS IMPLEMENTED AFTER GETTING THE DATA USING THE OBJECTINSPECTORS

GENERICUDF

INITIALIZE()

EVALUATE(DEFERREDOBJECT[] ARGUMENTS)

GETDISPLAYSTRING()

THIS COMBINATION OF DEFFEREDOBJECT+OBJECTINSPECTOR INTERFACES HELPS IN LAZY EVALUATION

GENERICUDF

INITIALIZE()

EVALUATE(DEFERREDOBJECT[] ARGUMENTS)

GETDISPLAYSTRING()

UNTIL THE DATA IS EXTRACTED USING OBJECT INSPECTOR IT REMAINS IN RAW BYTE FORM

GENERICUDF

INITIALIZE()

EVALUATE(DEFERREDOBJECT[] ARGUMENTS)

GETDISPLAYSTRING()

WHEN THE OBJECTINSPECTOR IS USED, WE GET THE RIGHT OBJECTS NEEDED TO PROCESS THE DATA FURTHER

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

THIS METHOD IS FOR DOCUMENTATION

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

IT WILL RETURN A STRING THAT WILL BE USED WHEN YOU TRY TO RUN THE EXPLAIN COMMAND

GENERICUDF

INITIALIZE()

EVALUATE()

GETDISPLAYSTRING()

THE EXPLAIN COMMAND

WHEN YOU USE THIS WITH A QUERY, IT SHOWS YOU AN EXECUTION PLAN FOR THAT QUERY, I.E. NUM MAPPERS, REDUCERS ETC

LET'S SEE THE CODE FOR THE CONTAINSSTRING() FUNCTION

GENERIC UDF CONTAINS STRING()

```java
public class genericUDF extends GenericUDF {

    ListObjectInspector listOI;
    StringObjectInspector elementOI;

    @Override
    public String getDisplayString(String[] arg0) {
        return “checks if the string is present in the list";    }

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {
        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("Function only takes 2 arguments: List<T>, T");
        }
        // 1. Check we received the right object types.
        ObjectInspector a = arguments[0];
        ObjectInspector b = arguments[1];
        if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
        }
        this.listOI = (ListObjectInspector) a;
        this.elementOI = (StringObjectInspector) b;

        // 2. Check that the list contains strings
        if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list of strings");
        }

        // the return type of our function is a boolean, so we provide the correct object inspector
        return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        // get the list and string from the deferred objects using the object inspectors
        List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
        String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

        // check for nulls
        if (list == null || arg == null) {
            return null;
        }

        // see if our list contains the value we need
```

GENERIC UDF CONTAINS STRING()

```
public class genericUDF extends GenericUDF {

    ListObjectInspector listOI;
    StringObjectInspector elementOI;

    @Override
    public String getDisplayString(String[] arg0) {
        return "checks if the string is present in the list";    }

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {
        if (arguments.length != 2) {
            throw new UDFArgumentException("Function only takes two arguments: List<T>, T");
        }
        // 1. Check we received the right object types.
        ObjectInspector a = arguments[0];
        ObjectInspector b = arguments[1];
        if (!(a instanceof ListObjectInspector || b instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
        }
        this.listOI = (ListObjectInspector) a;
        this.elementOI = (StringObjectInspector) b;

        // 2. Check that the list contains strings
        if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list of strings");
        }

        // the return type of our function is a boolean, so we provide the correct object inspector
        return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        // get the list and string from the deferred objects using the object inspectors
        List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
        String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

        // check for nulls
        if (list == null || arg == null) {
            return null;
        }
```

OUR CODE IS IMPLEMENTED IN A CLASS THAT EXTENDS GENERICUDF

GENERIC UDF CONTAINS STRING()

```
public class genericUDF extends GenericUDF {

    ListObjectInspector listOI;
    StringObjectInspector elementOI;

    @Override
    public String getDisplayString(String[] arg0) {
        return “checks if the string is present in the list";    }

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {
        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("Function only takes 2 arguments: List<T>, T");
        }
        // 1. Check we received the right object types.
        ObjectInspector a = arguments[0];
        ObjectInspector b = arguments[1];
        if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
        }
        this.listOI = (ListObjectInspector) a;
        this.elementOI = (StringObjectInspector) b;

        // 2. Check that the list contains strings
        if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list of strings");
        }

        // the return type of our function is a boolean, so we provide the correct object inspector
        return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        // get the list and string from the deferred objects using the object inspectors
        List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
        String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

        // check for nulls
        if (list == null || arg == null) {
            return null;
```

WE SETUP MEMBER VARIABLES WHICH WILL REPRESENT THE INPUT DATA TYPES

GENERIC UDF CONTAINS STRING()

```
public class genericUDF extends GenericUDF {
```

```
ListObjectInspector listOI;
StringObjectInspector elementOI;
```

```
    @Override
    public String getDisplayString(String[] arg0) {
        return “checks if the string is present in the list";    }

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {
        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("Function only takes 2 arguments: List<T>, T");
        }
        // 1. Check we received the right object types.
        ObjectInspector a = arguments[0];
        ObjectInspector b = arguments[1];
        if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
        }
        this.listOI = (ListObjectInspector) a;
        this.elementOI = (StringObjectInspector) b;

        // 2. Check that the list contains strings
        if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list of strings");
        }

        // the return type of our function is a boolean, so we provide the correct object inspector
        return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        // get the list and string from the deferred objects using the object inspectors
        List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
        String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

        // check for nulls
        if (list == null || arg == null) {
            return null;
```

THE INPUTS WILL BE A LIST AND STRING

GENERIC UDF CONTAINS STRING()

```
public class genericUDF extends GenericUDF {
```

```
ListObjectInspector listOI;
StringObjectInspector elementOI;
```

```
    @Override
    public String getDisplayString(String[] arg0) {
        return “checks if the string is present in the list";    }

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {
        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("Function only takes 2 arguments: List<T>, T");
        }
        // 1. Check we received the right object types.
        ObjectInspector a = arguments[0];
        ObjectInspector b = arguments[1];
        if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
        }
        this.listOI = (ListObjectInspector) a;
        this.elementOI = (StringObjectInspector) b;

        // 2. Check that the list contains strings
        if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector))
            throw new UDFArgumentException("first argument must be a list of strings");
        }

        // the return type of our function is a boolean, so we provide the correct object inspector
        return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        // get the list and string from the deferred objects using the object inspectors
        List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
        String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

        // check for nulls
        if (list == null || arg == null) {
            return null;
```

THESE ARE REPRESENTED USING THE CORRESPONDING OBJECTINSPECTORS

GENERIC UDF CONTAINS STRING()

```
public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {
    if (arguments.length != 2) {
        throw new UDFArgumentLengthException("Function only takes 2 arguments: List<T>, T");
    }
    // 1. Check we received the right object types.
    ObjectInspector a = arguments[0];
    ObjectInspector b = arguments[1];
    if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
    }
    this.listOI = (ListObjectInspector) a;
    this.elementOI = (StringObjectInspector) b;

    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
```

WE START BY OVERRIDING THE INITIALIZE METHOD

GENERIC UDF CONTAINS STRING()

```
public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {
    if (arguments.length != 2) {
        throw new UDFArgumentLengthException("Function only takes 2 arguments: List<T>, T");
    }
    // 1. Check we received the right object types.
    ObjectInspector a = arguments[0];
    ObjectInspector b = arguments[1];
    if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
    }
    this.listOI = (ListObjectInspector) a;
    this.elementOI = (StringObjectInspector) b;

    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
```

THIS METHOD RUNS EXACTLY ONCE FOR A QUERY

GENERIC UDF CONTAINS STRING()

```
lyString(String[] ar  {
the string is present in the  ", ")
```

```
public ObjectInspector
initialize(ObjectInspector[] arguments)
throws UDFArgumentException {
    if (arguments.length != 2) {
        throw new UDFArgumentLengthException("Function only takes 2 arguments: List<T>, T");
    }
    // 1. Check we received the right object types.
    ObjectInspector a = arguments[0];
    ObjectInspector b = arguments[1];
    if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
    }
    this.listOI = (ListObjectInspector) a;
    this.elementOI = (StringObjectInspector) b;

    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
```

IT CHECKS THE INPUT DATATYPES

GENERIC UDF CONTAINS STRING()

```
public ObjectInspector
initialize(ObjectInspector[] arguments)
throws UDFArgumentException {
    if (arguments.length != 2) {
        throw new UDFArgumentLengthException("Function only takes 2 arguments: List<T>, T");
    }
    // 1. Check we received the right object types.
    ObjectInspector a = arguments[0];
    ObjectInspector b = arguments[1];
    if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
    }
    this.listOI = (ListObjectInspector) a;
    this.elementOI = (StringObjectInspector) b;

    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
```

THIS METHOD WILL RETURN AN OBJECTINSPECTOR WITH THE OUTPUT DATA TYPE

GENERIC UDF CONTAINS STRING()

```
public class genericUDF extends GenericUDF {

    ListObjectInspector listOI;
    StringObjectInspector elementOI;

    @Override
    public String getDisplayString(String[] arg0) {
        return "checks if the string is present in the list";    }

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {
```

```
if (arguments.length != 2) {
throw new
UDFArgumentLengthException("Function only
takes 2 arguments: List<T>, T");
}
```

```
        // 1. Check we received the right object types.
        ObjectInspector a = arguments[0];
        ObjectInspector b = arguments[1];
        if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector))
            throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
        }
        this.listOI = (ListObjectInspector) a;
        this.elementOI = (StringObjectInspector) b;

        // 2. Check that the list contains strings
        if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector))
            throw new UDFArgumentException("first argument must be a list of strings");
        }

        // the return type of our function is a boolean, so we provide the correct object inspector
        return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        // get the list and string from the deferred objects using the object inspectors
```

WE FIRST CHECK IF THE RIGHT NUMBER OF ARGUMENTS ARE GIVEN TO THE FUNCTION

GENERIC UDF CONTAINS STRING()

```
public class genericUDF extends GenericUDF {

    ListObjectInspector listOI;
    StringObjectInspector elementOI;

    @Override
    public String getDisplayString(String[] arg0) {
        return "checks if the string is present in the list";    }

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {
if (arguments.length != 2) {
throw new
UDFArgumentLengthException("Function only
takes 2 arguments: List<T>, T");
}

        // 1. Check we received the right object types.
        ObjectInspector a = arguments[0];
        ObjectInspector b = arguments[1];
        if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
        }
        this.listOI = (ListObjectInspector) a;
        this.elementOI = (StringObjectInspector) b;

        // 2. Check that the list contains strings
        if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list of strings");
        }

        // the return type of our function is a boolean, so we provide the correct object inspector
        return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        // get the list and string from the deferred objects using the object inspectors
```

IF NOT, WE THROW AN EXCEPTION

GENERIC UDF CONTAINS STRING()

```
    return "checks if the string is present in the list";    }

@Override
public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {
    if (arguments.length != 2) {
        throw new UDFArgumentLengthException("Function only takes 2 arguments: List<T>, T");
    }
    // 1. Check we received the right object types.
    ObjectInspector a = arguments[0];
    ObjectInspector b = arguments[1];
if (!(a instanceof ListObjectInspector) || !(b instanceof
StringObjectInspector)) {
throw new UDFArgumentException("first argument must be a list /
array, second argument must be a string");
            }
    this.listOI = (ListObjectInspector) a;
    this.elementOI = (StringObjectInspector) b;

    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
        if (arg.equals(s)) return new Boolean(true);
```

THEN WE CHECK IF THE INPUT DATA TYPES MATCH EXPECTED TYPES

GENERIC UDF CONTAINS STRING()

```
    return "checks if the string is present in the list";    }

@Override
public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentExcepti
    if (arguments.length != 2) {
        throw new UDFArgumentLengthException("Function only takes 2 arguments: List<T>, T");
    }
    // 1. Check we received the right object types.
    ObjectInspector a = arguments[0];
    ObjectInspector b = arguments[1];
    if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
    }
```

```
this.listOI = (ListObjectInspector) a;
this.elementOI = (StringObjectInspector) b;
```

```
    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
        if (arg.equals(s)) return new Boolean(true);
    }
    return new Boolean(false);
}
```

WE SETUP THE INPUT OBJECT INSPECTORS

GENERIC UDF CONTAINS STRING()

```
    return "checks if the string is present in the list";    }

@Override
public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentExcepti
    if (arguments.length != 2) {
        throw new UDFArgumentLengthException("function only takes 2 arguments: List<T>, T");
    }
    // 1. Check we received the right object types.
    ObjectInspector a = arguments[0];
    ObjectInspector b = arguments[1];
    if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
    }
```

```
this.listOI = (ListObjectInspector) a;
this.elementOI = (StringObjectInspector) b;
```

```
    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveExce

    // get the list and string from the deferred objects using the
    List<String> list = (List<String>) this.listOI.getList(argumen
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we nee
    for(String s: list) {
        if (arg.equals(s)) return new Boolean(tru
    }
    return new Boolean(false);
}
```

BY SETTING THESE UP IN INITIALIZE, THE SAME OBJECT INSPECTORS WILL BE REUSED FOR EACH ROW

GENERIC UDF CONTAINS STRING()

```
    ObjectInspector b = arguments[1];
    if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
    }
    this.listOI = (ListObjectInspector) a;
    this.elementOI = (StringObjectInspector) b;

    // 2. Check that the list contains strings
```

```
    if(!(listOI.getListElementObjectInspector()
instanceof StringObjectInspector)) {
throw new UDFArgumentException("first argument must
be a list of strings");
        }
```

```
    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = (String) this.elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
        if (arg.equals(s)) return new Boolean(true);
    }
    return new Boolean(false);
}
```

WE NEED TO MAKE 1 LAST CHECK, I.E. ARE THE LIST ELEMENTS STRINGS?

GENERIC UDF CONTAINS STRING()

```
    ObjectInspector b = arguments[1];
    if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
    }
    this.listOI = (ListObjectInspector) a;
    this.elementOI = (StringObjectInspector) b;

    // 2. Check that the list contains strings
```

```
    if(!(listOI.getListElementObjectInspector()
instanceof StringObjectInspector)) {
throw new UDFArgumentException("first argument must
be a list of strings");
            }
```

```
    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list of strings from the deferred objects, the order is important
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
        if (arg.equals(s)) return new Boolean(true);
    }
    return new Boolean(false);
}
```

WE ARE NOW SURE THAT THE RIGHT DATA TYPES ARE BEING PASSED TO THE FUNCTION

GENERIC UDF CONTAINS STRING()

```
    if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
    }
    this.listOI = (ListObjectInspector) a;
    this.elementOI = (StringObjectInspector) b;

    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
return
PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
    }

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // se[illegible] that contains the value we need
    for(S[illegible] s : [illegible]) {
        if (arg.equals(s)) return new Boolean(true);
    }
    return [illegible]Boolean(fa[illegible]);
}
```

FINALLY, THE INITIALIZE METHOD NEEDS TO RETURN AN OBJECTINSPECTOR FOR THE OUTPUT DATA TYPE

GENERIC UDF CONTAINS STRING()

```
    if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
    }
    this.listOI = (ListObjectInspector) a;
    this.elementOI = (StringObjectInspector) b;

    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
return
PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
    }

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
        if (arg.equals(s)) return new Boolean(true);
    }
    return new Boolean(false);
}
```

CONTAINSSTRING IS A BOOLEAN FUNCTION

RETURNS TRUE IF THE LIST CONTAINS THE STRING

GENERIC UDF CONTAINS STRING()

```
    if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
    }
    this.listOI = (ListObjectInspector) a;
    this.elementOI = (StringObjectInspector) b;

    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
return
PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
    }

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
        if (arg.equals(s)) return new BooleanWritable(true);
    }
    return new BooleanWritable(false);
}
```

WE USE A FACTORY TO SETUP THE OUTPUT OBJECTINSPECTOR

GENERIC UDF CONTAINS STRING()

```
    if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
    }
    this.listOI = (ListObjectInspector) a;
    this.elementOI = (StringObjectInspector) b;

    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
return
PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
    }

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we ne
    for(String s: list) {
        if (arg.equals(s)) return new Boolean(true);
    }
    return new Boolean(false);
}
```

THIS IS ADVISED SO THAT THE SAME OBJECTINSPECTOR CAN BE REUSED FOR EACH ROW

GENERIC UDF CONTAINS STRING()

```
    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
        if (arg.equals(s)) return new Boolean(true);
    }
    return new Boolean(false);
}
```

WE MOVE ON TO THE EVALUATE METHOD

GENERIC UDF CONTAINS STRING()

```
    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
        if (arg.equals(s)) return new Boolean(true);
    }
    return new Boolean(false);
}
```

THE METHOD IS CALLED ONCE FOR EACH ROW

GENERIC UDF CONTAINS STRING()

```
    // 2. Check that the list contains strings
    if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
        if (arg.equals(s)) return new Boolean(true);
    }
    return new Boolean(false);
}
```

THE ROW DATA IS PASSED IS AN DEFERREDOBJECT ARRAY

GENERIC UDF CONTAINS STRING()

```
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
        if (arg.equals(s)) return new Boolean(true);
    }
    return new Boolean(false);
}
```

WE USE OUR OBJECTINSPECTORS TO EXTRACT THE DATA FROM THE ROW

GENERIC UDF CONTAINS STRING()

```
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
        if (arg.equals(s)) return new Boolean(true);
    }
    return new Boolean(false);
}
```

LISTOBJECTINSPECTOR HAS A GETLIST() METHOD TO EXTRACT THE LIST FROM THE OBJECT

GENERIC UDF CONTAINS STRING()

```
        throw new UDFArgumentException("first argument must be a list of strings");
    }

    // the return type of our function is a boolean, so we provide the correct object inspector
    return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
}

@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
    if (list == null || arg == null) {
        return null;
    }

    // see if our list contains the value we need
    for(String s: list) {
        if (arg.equals(s)) return new Boolean(true);
    }
    return new Boolean(false);
}
```

STRINGOBJECTINPECTOR OVERRIDES THE GETPRIMITIVEJAVAOBJECT() METHOD FROM PRIMITIVEOBJECTINSPECTOR

GENERIC UDF CONTAINS STRING()

```
@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
if (list == null || arg == null) {
            return null;
        }

// see if our list contains the value we need
for(String s: list) {
    if (arg.equals(s)) return new Boolean(true);
    }
return new Boolean(false);
}
```

FINALLY! WE HAVE THE FUNCTION LOGIC

GENERIC UDF

CONTAINS STRING()

```
@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
if (list == null || arg == null) {
            return null;
        }

// see if our list contains the value we need
for(String s: list) {
    if (arg.equals(s)) return new Boolean(true);
    }
return new Boolean(false);
  }
```

ITERATE THROUGH THE LIST AND CHECK AGAINST THE STRING

GENERIC UDF CONTAINS STRING()

```
@Override
public Object evaluate(DeferredObject[] arguments) throws HiveException {

    // get the list and string from the deferred objects using the object inspectors
    List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
    String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

    // check for nulls
if (list == null || arg == null) {
            return null;
        }

// see if our list contains the value we need
for(String s: list) {
    if (arg.equals(s)) return new Boolean(true);
    }
return new Boolean(false);
  }
```

IF THE STRING MATCHES RETURN TRUE, ELSE FALSE

GENERIC UDF CONTAINS STRING()

```
public class genericUDF extends GenericUDF {

    ListObjectInspector listOI;
    StringObjectInspector elementOI;

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {
        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("Function only takes 2 arguments: List<T>, T");
        }
        // 1. Check we received the right object types.
        ObjectInspector a = arguments[0];
        ObjectInspector b = arguments[1];
        if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
        }
        this.listOI = (ListObjectInspector) a;
        this.elementOI = (StringObjectInspector) b;

        // 2. Check that the list contains strings
        if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list of strings");
        }

        // the return type of our function is a boolean, so we provide the correct object inspector
        return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        // get the list and string from the deferred objects using the object inspectors
        List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
        String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

        // check for nulls
        if (list == null || arg == null) {
```

```
    @Override
    public String getDisplayString(String[] arg0) {
        return "checks if the string is present in the list";
}
```

WE ALSO NEED TO OVERRIDE THE GETDISPLAYSTRING METHOD

GENERIC UDF CONTAINS STRING()

```
public class genericUDF extends GenericUDF {

    ListObjectInspector listOI;
    StringObjectInspector elementOI;

    @Override
    public String getDisplayString(String[] arg0) {
        return “checks if the string is present in the list";
}

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {
        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("Function only takes 2 arguments: List<T>, T");
        }
        // 1. Check we received the right object types.
        ObjectInspector a = arguments[0];
        ObjectInspector b = arguments[1];
        if (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list / array, second argument must be a string");
        }
        this.listOI = (ListObjectInspector) a;
        this.elementOI = (StringObjectInspector) b;

        // 2. Check that the list contains strings
        if(!(listOI.getListElementObjectInspector() instanceof StringObjectInspector)) {
            throw new UDFArgumentException("first argument must be a list / array");
        }

        // the return type of our function is a boolean, so we provide the correct object
        return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        // get the list and string from the deferred objects using the object inspectors
        List<String> list = (List<String>) this.listOI.getList(arguments[0].get());
        String arg = elementOI.getPrimitiveJavaObject(arguments[1].get());

        // check for nulls
        if (list == null || arg == null) {
```

THIS METHOD IS JUST FOR DOCUMENTATION PURPOSES

GENERIC UDF

GENERIC UDFS ARE ALSO USEFUL WHEN YOU NEED A FUNCTION THAT CAN DEAL WITH MULTIPLE DATA TYPES

GENERIC UDF

CONSIDER A CONDITIONAL FUNCTION USED FOR REMOVING NULL VALUES

NVL(X,Y)

IF X IS NOT NULL RETURN X,
ELSE RETURN Y

GENERIC UDF

CONSIDER A CONDITIONAL FUNCTION USED FOR REMOVING NULL VALUES

NVL(X,Y)

X,Y CAN BE OF ANY DATA TYPE

AS LONG AS THEY BOTH HAVE THE SAME DATA TYPE

GENERIC UDF

NVL(X,Y)

IF YOU USED A SIMPLE UDF

YOU WOULD NEED TO WRITE A DIFFERENT FUNCTION FOR EACH DIFFERENT DATA TYPE

GENERIC UDF

NVL(X,Y)

WITH A GENERIC UDF

YOU CAN RESOLVE THE DATA TYPE ON IN THE INITIALIZE METHOD AND THEN USE IT

GENERIC UDF

LETS LOOK AT THE CODE FOR NVL

GENERIC UDF

NVL

```java
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

      @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("NVL only takes 2 arguments");
        }

        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0])) && !(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments in NVL must have the same type, but they are different:"+arguments[0].getTypeName()+" , "+arguments[1].getTypeName());
        }


      return returnOIResolver.get();
    }


    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

        if (retVal==null){
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
        }
        return retVal;
    }

    @Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {
  private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
   private ObjectInspector[] argumentOIs;

     @Override
   public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

       argumentOIs=arguments;

       if (arguments.length != 2) {
           throw new UDFArgumentLengthException("NVL only takes 2 arguments");
       }

       returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

       if (!(returnOIResolver.update(arguments[0])) && !(returnOIResolver.update(arguments[1]))) {
           throw new UDFArgumentException("The arguments in NVL must have the same type, but they are different:"+arguments[0].getTypeName()+" , "+arguments[1].getTypeName());
       }


     return returnOIResolver.get();
   }


   @Override
   public Object evaluate(DeferredObject[] arguments) throws HiveException {

       Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

       if (retVal==null){
           retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
       }
       return retVal;
   }

   @Override
   public String getDisplayString(String[] children) {
       StringBuilder sb = new StringBuilder();
       sb.append("if ");
       sb.append(children[0]);
       sb.append(" is null returns");
       sb.append(children[1]);
       return sb.toString();
   }

}
```

WE WRITE A SUBCLASS OF GENERICUDF

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {

private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
private ObjectInspector[] argumentOIs;

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("NVL only takes 2 arguments");
        }

        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0])) && !(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments in NVL must have the same type, but they are different:"+arguments[0].getTypeName()+" , "+arguments[1].getTypeName());
        }


    return returnOIResolver.get();
    }


    @Override
    public Object evaluate(DeferredObject[] [illegible]) [illegible] Hive [illegible] {

        Object retVal=returnOIResolver.conve[illegible]IN[illegible]ssa[illegible]argumen[illegible]].[illegible]().[illegible]umen[illegible][0]

        if (retVal==null){
            re[illegible]IRes[illegible]
        }
        return ret[illegible]
    }

    @Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

WE SETUP AN ARRAY OF OBJECTINSPECTORS FOR THE INPUTS

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {
private GenericUDFUtils.ReturnObjectInspectorResolver
returnOIResolver;
private ObjectInspector[] argumentOIs;

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("NVL only takes 2 arguments");
        }

        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0])) && !(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments in NVL must have the same type, but they are different:"+arguments[0].getTypeName()+" , "+arguments[1].getTypeName());
        }

    return returnOIResolver.get();
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

        if (retVal==null) {
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
        }
        return retVal;
    }

    @Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

WE DON'T KNOW WHAT TYPE OF OBJECTINSPECTORS WE'LL NEED

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {
private GenericUDFUtils.ReturnObjectInspectorResolver
returnOIResolver;
private ObjectInspector[] argumentOIs;

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("NVL only takes 2 arguments");
        }

        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0], [illegible]returnOIResolver.update(arguments[1], [illegible]
            throw new UDF[illegible]Exception("[illegible] NVL must have the same [illegible] type [illegible] iff [illegible]=:"+arguments[0].getTypeName()+[illegible]+arguments[1].getTypeName[illegible]
        }

      return returnOIResolver.get();
    }

    @Override
    public Object[illegible] throws [illegible]

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

        if (retVal==null){
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get[illegible]s[1][illegible]
        }
        return retVal;
    }

    @Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

WE USE A RESOLVER CLASS THAT CAN LOOK AT AN OBJECTINSPECTOR AND TELL US ITS PROPERTIES

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {
private GenericUDFUtils.ReturnObjectInspectorResolver
returnOIResolver;
private ObjectInspector[] argumentOIs;

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("NVL only takes 2 arguments");
        }

        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0])) && !(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments in NVL must have the same type, but they are different:"+arguments[0].getTypeName()+" , "+arguments[1].getTypeName());
        }

    return returnOIResolver.get();
    }

    @Override
    public [illegible]
        [illegible]
        if [illegible]
        }
        return retVal;
    }

    @Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

IT CAN ALSO BE USED TO CAST AN OBJECT INSPECTOR TO A SPECIFIED TYPE

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

    @Override
     public ObjectInspector initialize(ObjectInspector[] arguments)
throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("NVL only takes 2 arguments");
        }

        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0])) && !(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments in NVL must have the same type, but they are different:"+arguments[0].getTypeName()+" , "+arguments[1].getTypeName());
        }


      return returnOIResolver.
    }


    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentO
                                                                                    0]);

        if (retVal==null){
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[
        }
        return retVal;
    }

    @Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

WE START WIT THE INITIALIZE METHOD

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

      @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("NVL only takes 2 arguments");
        }

        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0])) && !(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments in NVL must have the same type, but they are different:"+arguments[0].getTypeName()+" , "+arguments[1].getTypeName());
        }


      return returnOIResolver.get();
    }


    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

        if (retVal==null){
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
        }
        return retVal;
    }

    @Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

THE INPUT OBJECTINSPECTORS

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

      @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;
```

```
if (arguments.length != 2) {
 throw new UDFArgumentLengthException("NVL only takes 2 arguments");
          }
```

```
        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0])) && !(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments in NVL must have the same type, but they are different:"+arguments[0].getTypeName()+" , "+arguments[1].getTypeName());
        }

      return returnOIResolver.get
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(argu
        if (retVal==null){
            retVal=returnOIResolver.convertIfNecessary(arguments
        }
        return retVal;
    }

    @Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

IF THERE ARE MORE THAN 2, THEN THROW AN ERROR

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

      @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("NVL only takes 2 arguments");
        }
```

```
returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);
```

```
        if (!(returnOIResolver.update(arguments[0])) && !(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments in NVL must have the same type, but they are different:"+arguments[0].getTypeName()+" , "+arguments[1].getTypeName());
        }


      return returnOIResolver.get();
    }


    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

        if (retVal==null){
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
        }
        return retVal;
    }

    @Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

INSTANTIATE THE RESOLVER

GENERIC UDF

NVL

```
if (!(returnOIResolver.update(arguments[0])) && !
(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments in NVL must
have the same type, but they are different:"+arguments[0].getTypeName()
+" , "+arguments[1].getTypeName());
        }
```

WE USE THE RESOLVER'S UPDATE METHOD TO IDENTIFY THE OBJECTINSPECTOR CLASSES

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {

  private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
   private ObjectInspector[] argumentOIs;

     @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

       argumentOIs=arguments;

       if (arguments.length != 2) {
           throw new UDFArgumentLengthException("NVL only takes 2 arguments");
       }

       returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);
```

```
if (!(returnOIResolver.update(arguments[0])) && !
(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments in NVL must
have the same type, but they are different:"+arguments[0].getTypeName()
+" , "+arguments[1].getTypeName());
        }
```

```
    return returnOIResolver.g
  }

  @Override
  public Object evaluate(DeferredObject[] arguments) throws HiveException {

      Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

      if (retVal==null){
          retVal=returnOIRe                    [1].                    argume          ])
      }
      return
  }

  @Override
  public St                                    ) {
      String[] sb = new StringBuilder(
      sb.append("if ");
      sb.append(children[0]);
      sb.append(" is null returns");
      sb.append(children[1]);
      return sb.toString();
  }

}
```

WE APPLY A CHECK TO SEE IF BOTH ARGUMENTS ARE OF THE SAME TYPE

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

      @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("NVL only takes 2 arguments");
        }

        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0])) && !(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments in NVL must have the same type, but they are different:"+arguments[0].getTypeName()+" , "+arguments[1].getTypeName());
        }

        return returnOIResolver.get();
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

        if (retVal
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
        }
        return retVal;
    }

    @Override
    public String[] getDisplayStrings(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if (");
        sb.append(children[0]);
        sb.append(" is null, ");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

THE INITIALIZE METHOD NEEDS TO RETURN THE OUTPUT OBJECTINSPECTOR

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

      @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("NVL only takes 2 arguments");
        }

        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0])) && !(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments in NVL must have the same type, but they are different:"+arguments[0].getTypeName()+" , "+arguments[1].getTypeName());
        }

        return returnOIResolver.get();
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

        if (retVal==null){
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
        }
        return retVal;
    }
    @Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("nvl(");
        sb.append(children[0]);
        sb.append(" is not null, ");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

THIS WILL BE THE SAME AS THE INPUT OI TYPE WHICH IS NOW STORED IN THE RESOLVER

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

      @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("NVL only takes 2 arguments");
        }
        returnOIResolver = (GenericUDFUtilsReturnObjectInspectorResolver)(new GenericUDFUtilsReturnObjectInspectorResolver(true));
        if (!returnOIResolver.update(arguments[0]) || !returnOIResolver.update(arguments[1]))
            throw new UDFArgumentException("The arguments in NVL must have the same type, but they are different: "+arguments[0].getTypeName()+" , "+arguments[1].getTypeName());
        }


      return returnOIResolver.get();
    }


    @Override
    public Object evaluate(DeferredObject[] arguments)
throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

        if (retVal==null){
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
        }
        return retVal;
    }

    @Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

NEXT WE HAVE THE EVALUATE METHOD

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

      @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length!=2){
            throw new UDFArgumentLengthException("NVL only takes two arguments");
        }

        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0])) && !(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments must share a common type, or be of different type groups");
        }


      return returnOIResolver.get();
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {
```

WE EXTRACT THE FIRST ARGUMENT USING THE RELEVANT OBJECTINSPECTOR

```
Object retVal=
returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

        if (retVal==null){
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
        }
        return retVal;
    }

    @Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

      @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;
        if (arguments.length!=2) {
            throw new UDFArgumentLengthException("NVL takes two arguments");
        returnOIResolver=(GenericUDFUtils.ReturnObjectInspectorResolver)new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0])) && !(returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentTypeException(0,"The arguments to NVL should have the same or compatible types");
        }

    return returnOIResolver.get();
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);
```

IF THE FIRST ARGUMENT IS NULL, WE EXTRACT AND RETURN THE SECOND ARGUMENT

```
        if (retVal==null){

retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
        }
        return retVal;
    }

    @Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }

}
```

GENERIC UDF

NVL

```
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

      @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("NVL only takes 2 arguments");
        }

        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);
        if (!(returnOIResolver.update(arguments[0]) && returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentTypeException("The arguments for NVL must have the same type, but they are different: "+arguments[0].getTypeName()+" and "+arguments[1].getTypeName());
        }
        return returnOIResolver.get();
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

        if (retVal==null){
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
        }
        return retVal;
    }
```

LAST WE HAVE THE DOCUMENTATION BIT

```
@Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }
}
```

GENERIC UDF

NVL

THIS METHOD IS WRITTEN IN A DIFFERENT WAY HERE

```
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

      @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("The function NVL accepts 2 arguments");
        }

        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0]) && returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentException("The arguments in NVL must have the same type but they are different: "+arguments[0].getTypeName() , "+arguments[1].getTypeName());
        }

      return returnOIResolver.get();
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

        if (retVal==null){
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
        }
        return retVal;
    }
```

```
  @Override
      public String getDisplayString(String[] children) {
          StringBuilder sb = new StringBuilder();
          sb.append("if ");
          sb.append(children[0]);
          sb.append(" is null returns");
          sb.append(children[1]);
          return sb.toString();
      }
}
```

GENERIC UDF

NVL

CHILDREN IS AN ARRAY OF THE EXPRESSIONS THAT ARE PASSED TO THE FUNCTION

```
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

      @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length != 2) {
            throw new UDFArgumentLengthException("The function NVL takes 2 arguments");
        }
        returnOIResolver = (GenericUDFUtils.ReturnObjectInspectorResolver) new GenericUDFUtils.ReturnObjectInspectorResolver(true);
        boolean[] b = new boolean[2];
        if (returnOIResolver.update(arguments[0], b[0]) && returnOIResolver.update(arguments[1], b[1]))
            throw new UDFArgumentException("The arguments to NVL must have the same type, but they are different: "+arguments[0].getTypeName() + ", "+arguments[1].getTypeName());
        }

      return returnOIResolver.get();
    }

      @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

        if (retVal==null){
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
        }
        return retVal;
    }

    @Override
        public String getDisplayString(String[] children) {
            StringBuilder sb = new StringBuilder();
            sb.append("if ");
            sb.append(children[0]);
            sb.append(" is null returns");
            sb.append(children[1]);
            return sb.toString();
        }
}
```

GENERIC UDF

NVL

THE EXPRESSIONS COULD BE COLUMN NAMES OR FUNCTIONS(COLUMN NAMES)

```
class genericUDF2 extends GenericUDF {

   private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

      @Override
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {

        argumentOIs=arguments;

        if (arguments.length!=2){
            throw new UDFArgumentLengthException("NVL only takes 2 arguments");
        }

        returnOIResolver=new GenericUDFUtils.ReturnObjectInspectorResolver(true);

        if (!(returnOIResolver.update(arguments[0],null)&&returnOIResolver.update(arguments[1],null))){
            throw new UDFArgumentException("The arguments in NVL must have the same type, but they are different: "+arguments[0].getTypeName()+" and "+arguments[1].getTypeName());
        }

        return returnOIResolver.get();
    }

    @Override
    public Object evaluate(DeferredObject[] arguments) throws HiveException {

        Object retVal=returnOIResolver.convertIfNecessary(arguments[0].get(),argumentOIs[0]);

        if (retVal==null){
            retVal=returnOIResolver.convertIfNecessary(arguments[1].get(),argumentOIs[1]);
        }
        return retVal;
    }

  @Override
     public String getDisplayString(String[] children) {
         StringBuilder sb = new StringBuilder();
         sb.append("if ");
         sb.append(children[0]);
         sb.append(" is null returns");
         sb.append(children[1]);
         return sb.toString();
     }
}
```

GENERIC UDF

NVL

EG: IF WE USE NVL(NAME,SURNAME), THIS METHOD WILL RETURN

"IF NAME IS NULL RETURNS SURNAME"

```
@Override
    public String getDisplayString(String[] children) {
        StringBuilder sb = new StringBuilder();
        sb.append("if ");
        sb.append(children[0]);
        sb.append(" is null returns");
        sb.append(children[1]);
        return sb.toString();
    }
}
```

HIVE CUSTOM FUNCTIONS

THE TYPE OF FUNCTION

STANDARD

AGGREGATE

TABLE GENERATING

THE TYPE OF INPUT/OUTPUT

PRIMITIVE

COLLECTION

GENERIC UDF

HIVE CUSTOM FUNCTIONS

THE TYPE OF FUNCTION

STANDARD

AGGREGATE

TABLE GENERATING

THE TYPE OF INPUT/OUTPUT

PRIMITIVE

COLLECTION

AGGREGATE FUNCTIONS OPERATE ON MULTIPLE ROWS AND FLATTEN THEM

HIVE CUSTOM FUNCTIONS

THE TYPE OF FUNCTION

STANDARD

AGGREGATE

TABLE GENERATING

THE TYPE OF INPUT/OUTPUT

PRIMITIVE

COLLECTION

AGGREGATE FUNCTIONS ARE NORMALLY USED WITH A GROUP BY CLAUSE

HIVE CUSTOM FUNCTIONS

THE TYPE OF FUNCTION

STANDARD

AGGREGATE

TABLE GENERATING

THE TYPE OF INPUT/OUTPUT

PRIMITIVE

COLLECTION

A CUSTOM AGGREGATE FUNCTION THAT USES PRIMITIVE DATA TYPES IS CALLED A SIMPLE UDAF

SIMPLE UDAF

LET'S SAY YOU HAD A TABLE WITH STOCK PRICES

Company	Date	Opening	Closing
GM	2015-01-13	24.54	23.67
FORD	2015-01-13	52.54	47.67
GM	2015-01-14	26.54	28.67
FORD	2015-01-14	44.54	47.67

CALCULATE THE MAX OF THE OPENING PRICES FOR EACH COMPANY

SIMPLE UDAF

Company	Date	Opening	Closing
GM	2015-01-13	24.54	23.67
FORD	2015-01-13	52.54	47.67
GM	2015-01-14	26.54	28.67
FORD	2015-01-14	44.54	47.67

HIVE ALREADY HAS A BUILT-IN MAX FUNCTION

SIMPLE UDAF

Company	Date	Opening	Closing
GM	2015-01-13	24.54	23.67
FORD	2015-01-13	52.54	47.67
GM	2015-01-14	26.54	28.67
FORD	2015-01-14	44.54	47.67

```
select max(opening) from
stockPrices group by company;
```

SIMPLE UDAF

Ticker	Date	Opening	Closing
GM	2015-01-13	24.54	23.67
FORD	2015-01-13	52.54	47.67
GM	2015-01-14	26.54	28.67
FORD	2015-01-14	44.54	47.67

BUT, LET'S SEE HOW WE CAN IMPLEMENT A CUSTOM FUNCTION TO ACHIEVE THE SAME OUTCOME

SIMPLE UDAF

THIS IS AN EXAMPLE OF A

USER DEFINED AGGREGATE FUNCTION

SIMPLE UDAF

THESE FUNCTIONS ARE IMPLEMENTED USING SUBCLASS OF THE CLASS

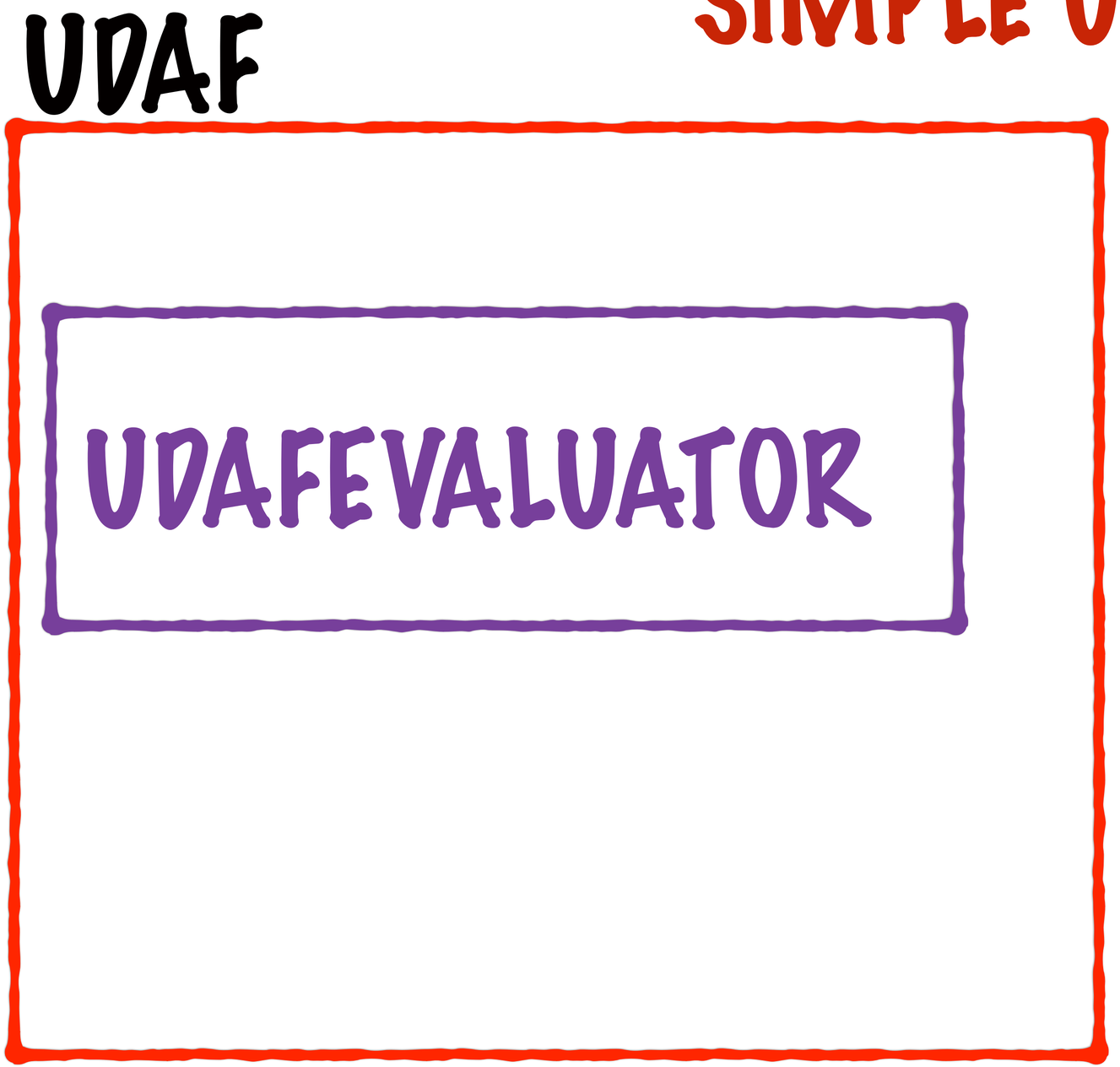
UDAF

SIMPLE UDAF

FUNCTIONS LIKE SUM, AVG, COUNT, MAX ETC ALL DERIVE FROM A GENERIC VERSION OF THE UDAF CLASS

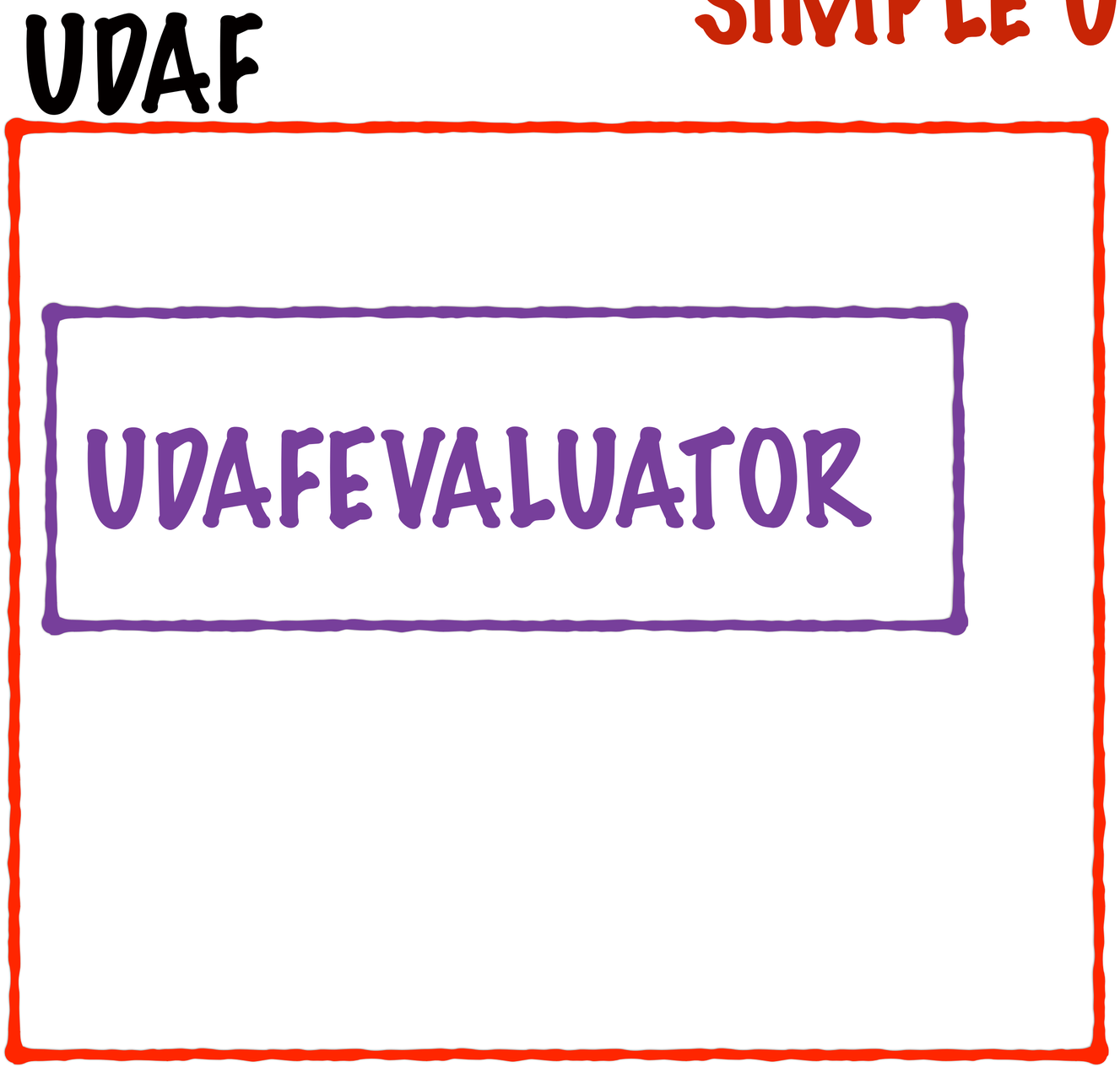
UDAF

SIMPLE UDAF



WITHIN THE UDAF CLASS WE NEED TO SETUP A UDAFEVALUATOR CLASS

SIMPLE UDAF



THE UDAFEVALUATOR IS WHERE THE LOGIC FOR THE FUNCTION WILL RESIDE

SIMPLE UDAF

UDAF

UDAFEVALUATOR

INIT()
ITERATE()
TERMINATEPARTIAL()
MERGE()
TERMINATE()

THIS CLASS HAS 5 METHODS WE NEED TO IMPLEMENT

SIMPLE UDAF

UDAF

UDAFEVALUATOR

INIT()
ITERATE()
TERMINATEPARTIAL()
MERGE()
TERMINATE()

LET'S UNDERSTAND HOW THESE METHODS ARE USED

SIMPLE UDAF

UDAF

UDAFEVALUATOR

INIT()
ITERATE()
TERMINATEPARTIAL()
MERGE()
TERMINATE()

THESE METHODS ARE CALLED AT DIFFERENT STAGES OF THE MAPREDUCE JOB

UDAF

SIMPLE UDAF

WHEN A QUERY IS RUN IN HIVE, HIVE SETS UP A MAP REDUCE JOB

LET'S QUICKLY RECAP HOW THE MAP REDUCE WORKS FOR AN AGGREGATION FUNCTION

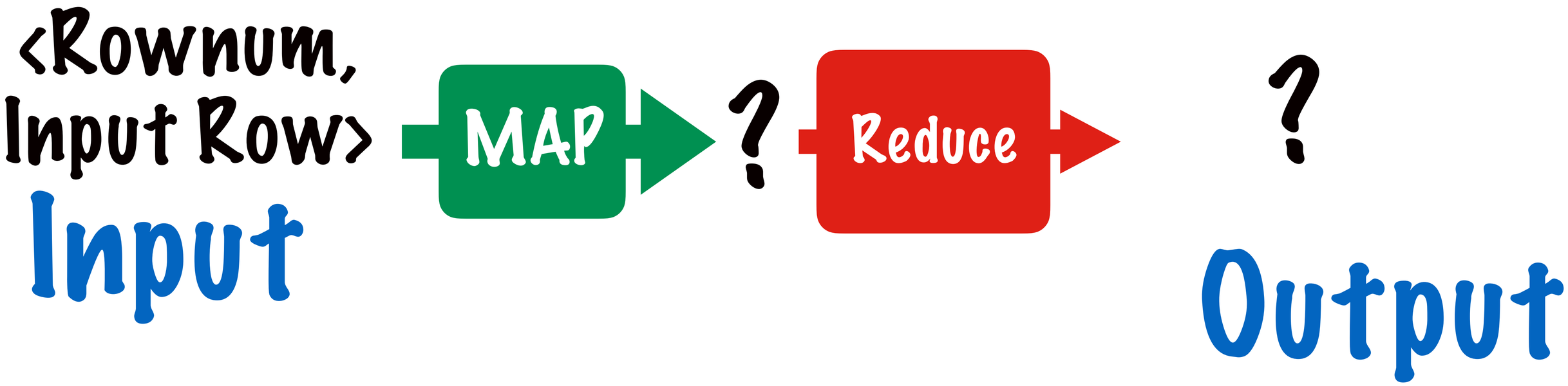
UDAF

SIMPLE UDAF

WHEN A QUERY IS RUN IN HIVE, HIVE SETS UP A MAP REDUCE JOB

LET'S QUICKLY RECAP HOW THE MAP REDUCE WORKS FOR AN AGGREGATION FUNCTION

SELECT company, Max(OPEN)
GROUP BY company



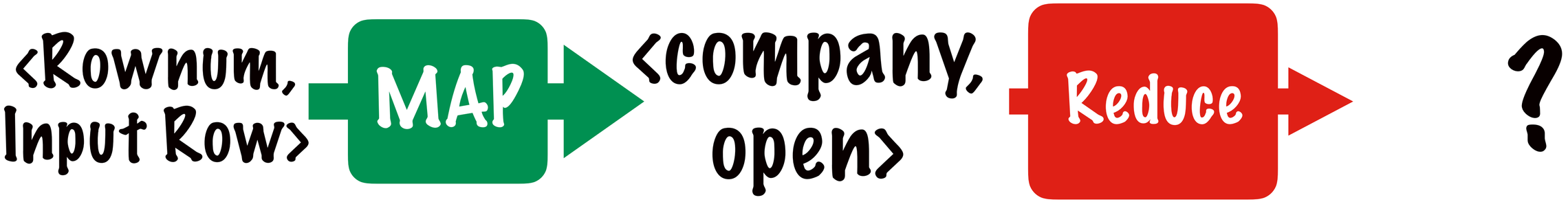
The input row looks like this

Ticker	Date	Opening	Closing
GM	2015-01-13	24.54	23.67

Ticker	Date	Opening	Closing
GM	2015-01-13	24.54	23.67

The aggregation operation naturally fits inside the reducer

SELECT company, Max(OPEN)
GROUP BY company



The Reducer will simply use the specified aggregation function to combine values that share a key

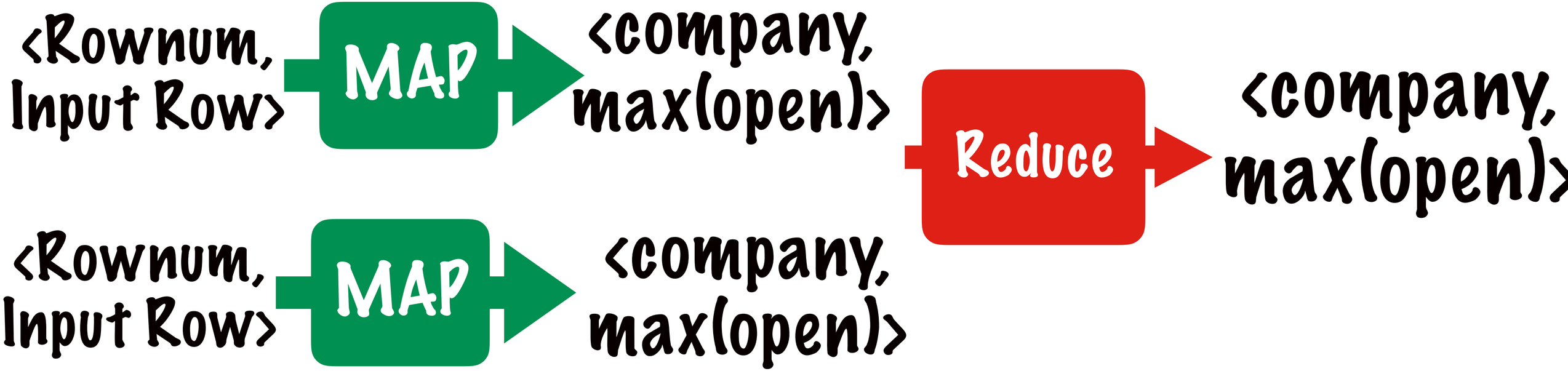
SELECT company, Max(OPEN)
GROUP BY company

The Reducer will simply use the specified aggregation function to combine values that share a key

SELECT company, Max(OPEN)
GROUP BY company

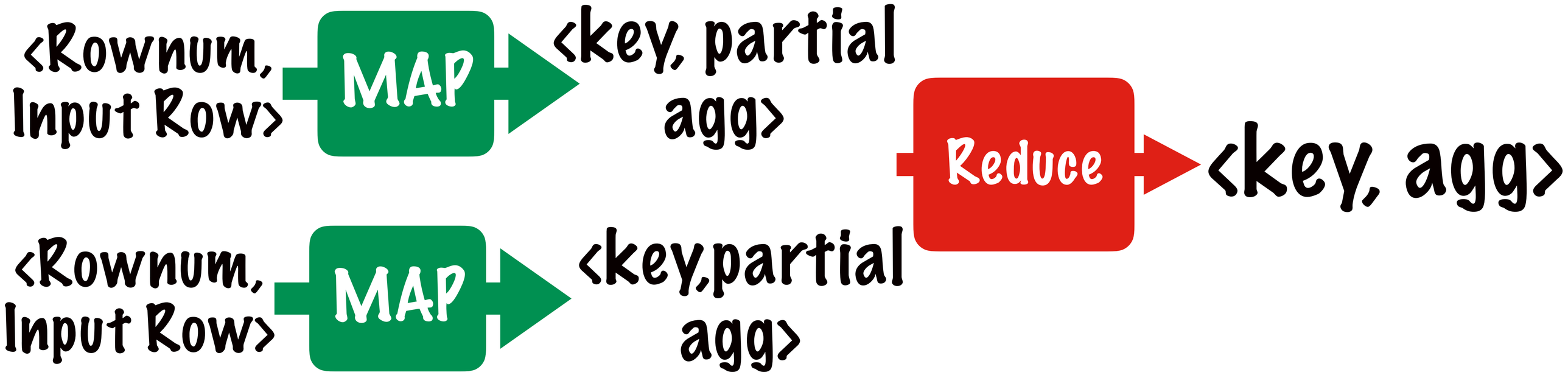
On each mapper, we could choose to do a partial aggregation

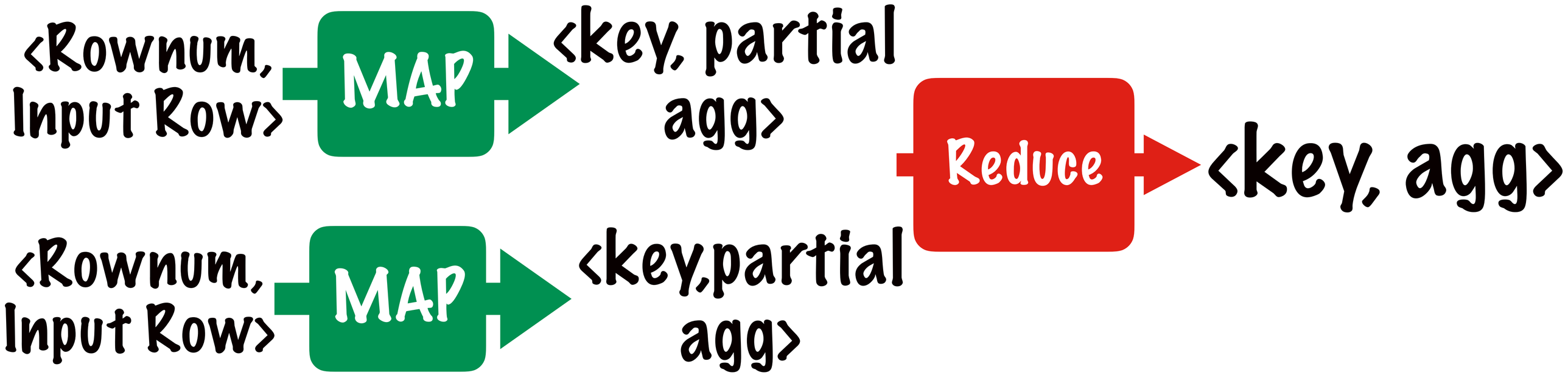
SELECT company, Max(OPEN)
GROUP BY company



We can find the max open within each mapper and the reduce will do the final aggregation

Summarizing





Let's see how the methods fit into this flow

UDAFEVALUATOR

- INIT()
- ITERATE()
- TERMINATEPARTIAL()
- MERGE()
- TERMINATE()

THE INIT METHOD IS CALLED ONCE BY EACH MAPPER/REDUCER

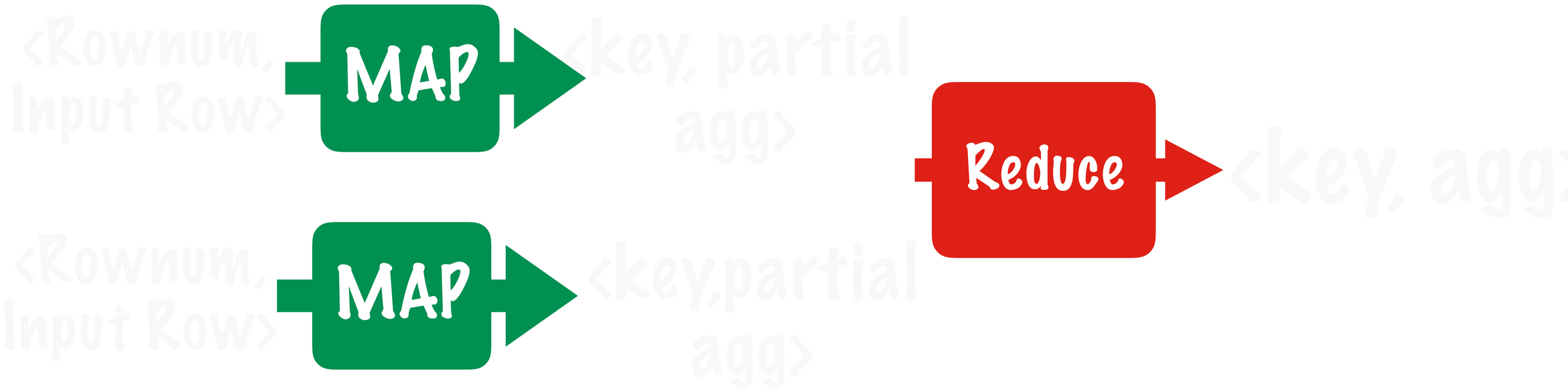
UDAFEVALUATOR

ITERATE() INIT()

TERMINATEPARTIAL()

MERGE()

TERMINATE()



THIS METHOD WILL INITIALIZE A RESULT VARIABLE

UDAFEVALUATOR

ITERATE() INIT()

TERMINATEPARTIAL()

MERGE()

TERMINATE()

AS THE MAPPER/ REDUCER GO THROUGH EACH ROW, THEY UPDATE THE RESULT VARIABLE

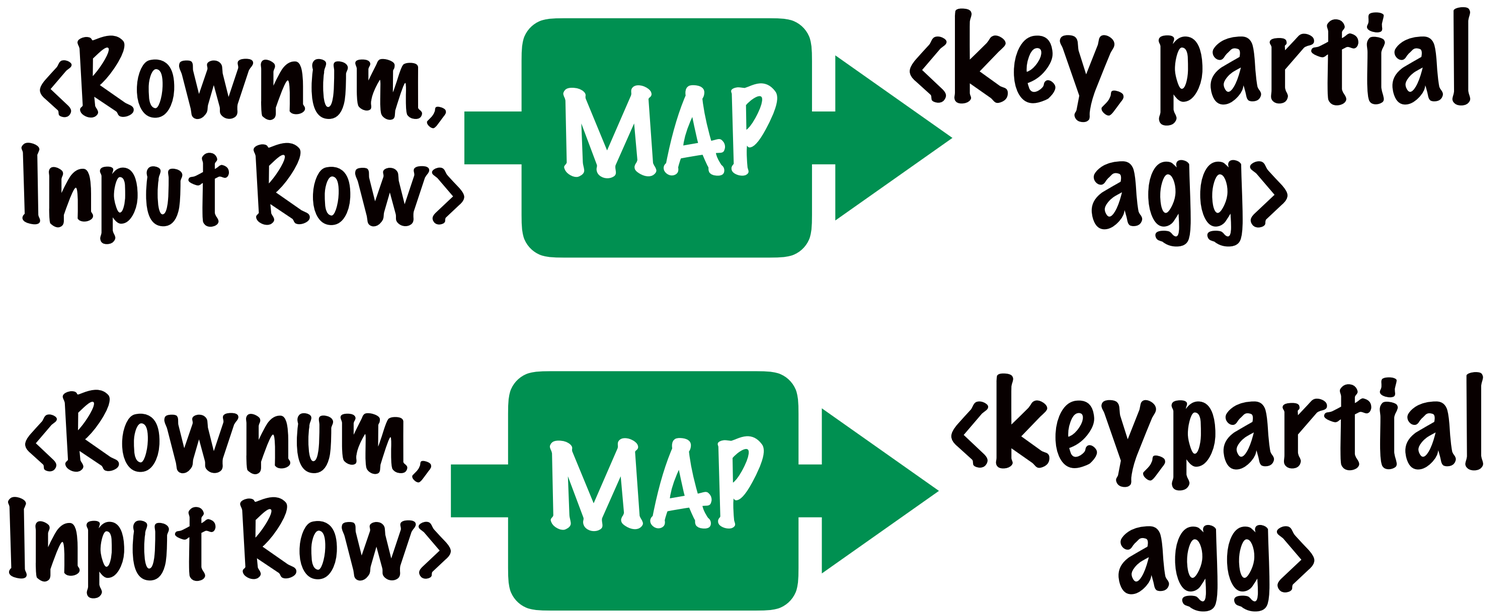
UDAFEVALUATOR

ITERATE() INIT()

TERMINATEPARTIAL()

MERGE()

TERMINATE()



THESE 2 METHODS ARE CALLED BY THE MAPPERS

UDAFEVALUATOR

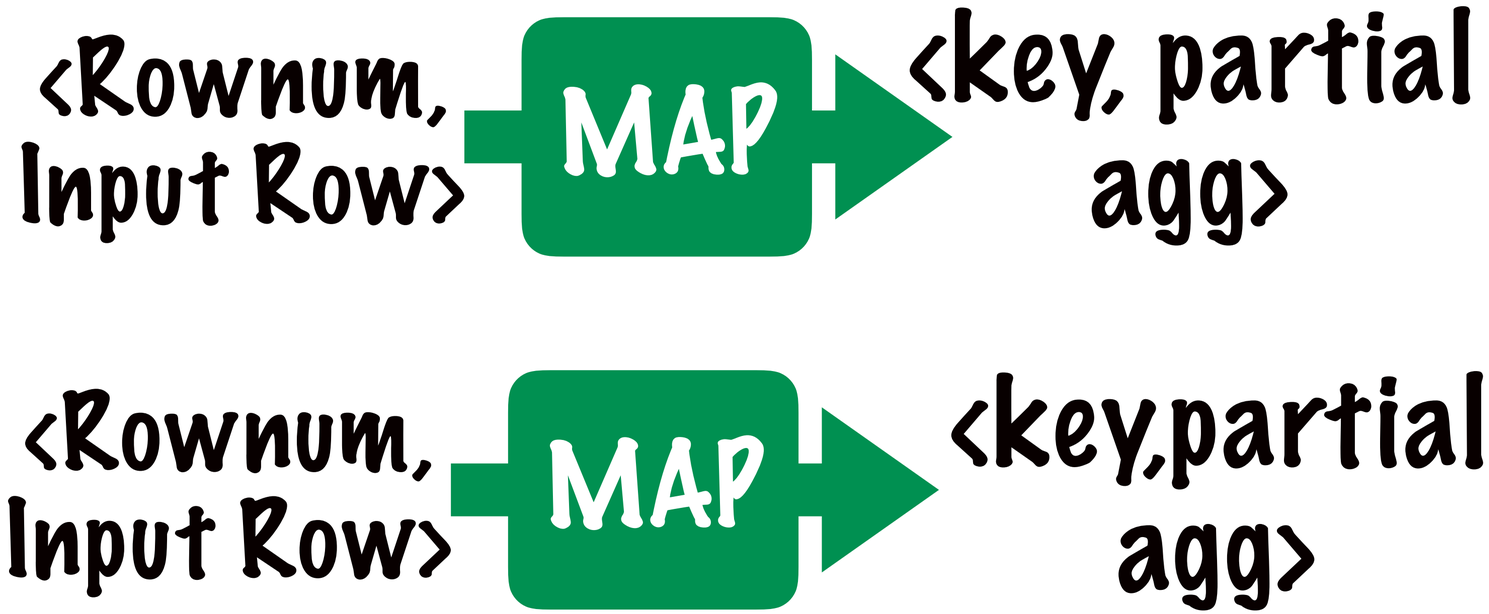
INIT()

ITERATE()

TERMINATEPARTIAL()

MERGE()

TERMINATE()



IN THE ITERATE METHOD THE PARTIAL AGGREGATION LOGIC IS EXECUTED

UDAFEVALUATOR

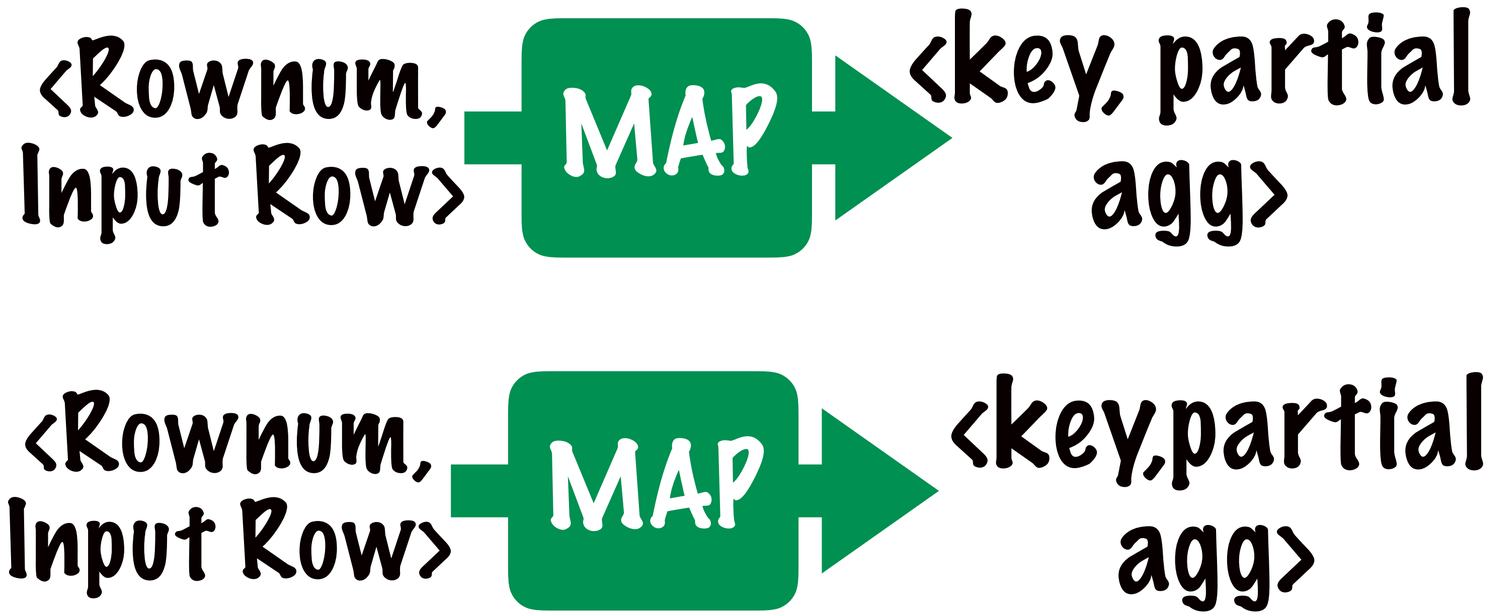
INIT()

ITERATE()

TERMINATEPARTIAL()

MERGE()

TERMINATE()



THE ITERATE METHOD ACTS ON 1 ROW AT A TIME

UDAFEVALUATOR

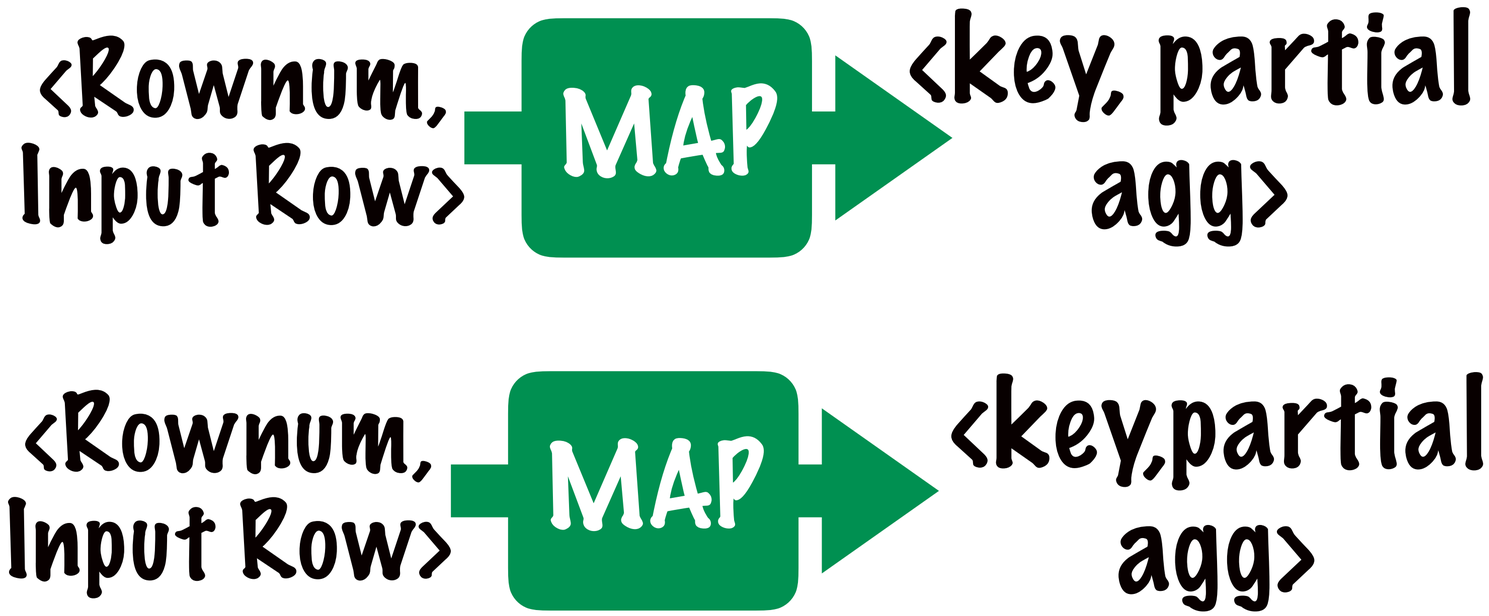
INIT()

ITERATE()

TERMINATEPARTIAL()

MERGE()

TERMINATE()



THIS LOGIC UPDATES A VARIABLE WITH THE PARTIAL RESULT TILL NOW

UDAFEVALUATOR

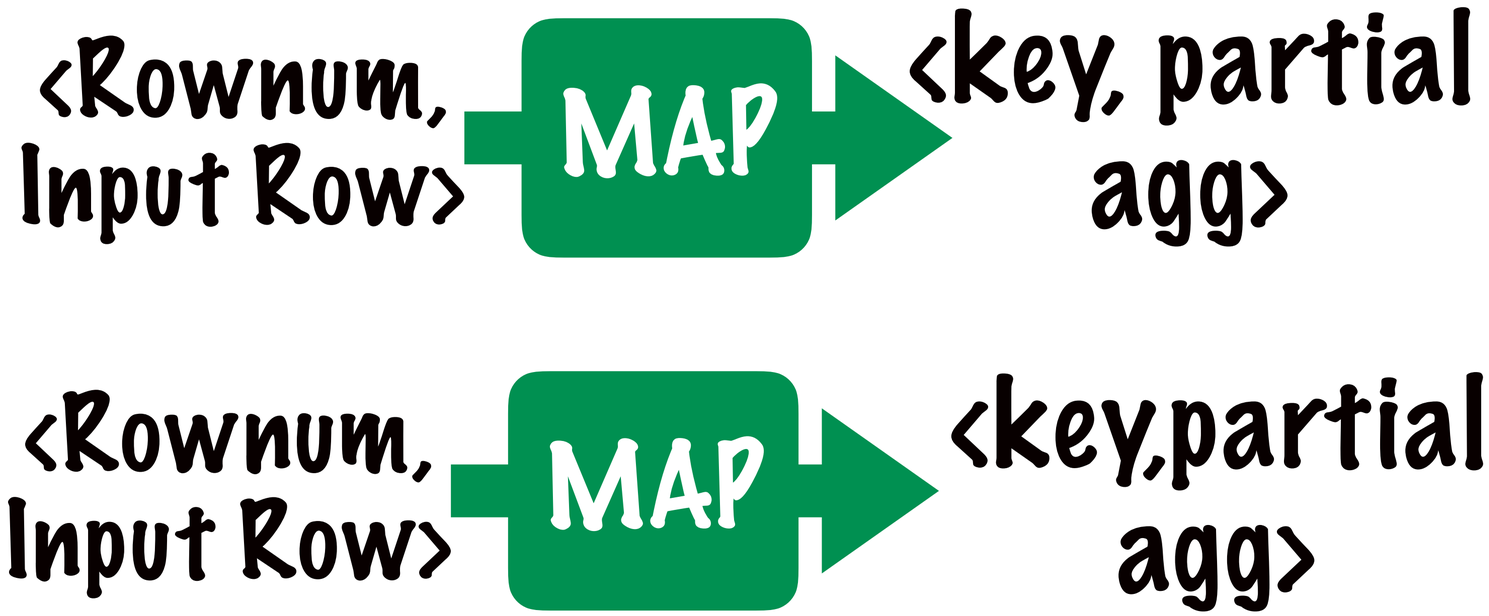
INIT()

ITERATE()

TERMINATEPARTIAL()

MERGE()

TERMINATE()



WHEN TERMINATE PARTIAL IS CALLED THE RESULT VARIABLE IS RETURNED

UDAFEVALUATOR

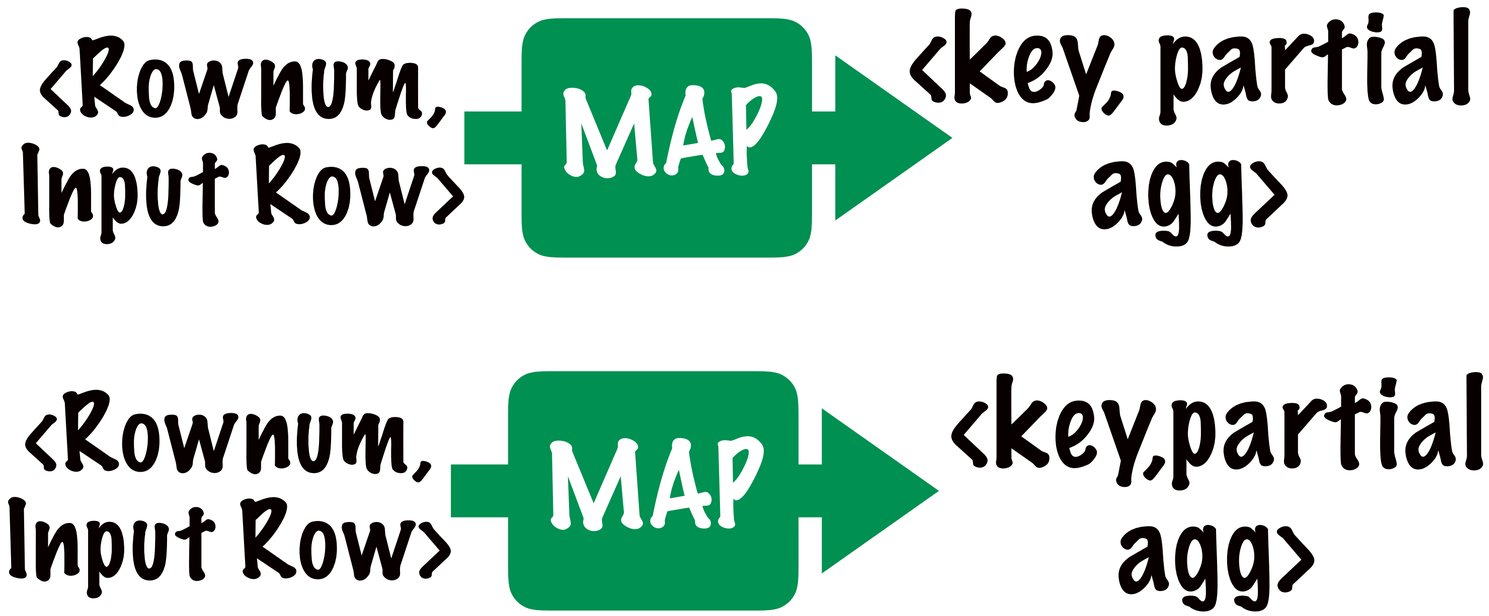
INIT()

ITERATE()

TERMINATEPARTIAL()

MERGE()

TERMINATE()



THE OUTPUTS FROM TERMINATEPARTIAL OF EACH MAPPER GO TO THE REDUCER

UDAFEVALUATOR

INIT()

ITERATE()

TERMINATEPARTIAL()

MERGE()

TERMINATE()

<key, partial agg>

<key,partial agg>

Reduce

<key, agg>

MERGE AND TERMINATE ARE CALLED BY THE REDUCER

UDAFEVALUATOR

ITERATE() INIT()

TERMINATEPARTIAL()

MERGE()

TERMINATE()

<key, partial agg>

<key,partial agg>

Reduce

<key, agg>

LIKE ITERATE, MERGE UPDATES A RESULT VARIABLE BASED ON THE CURRENT ROW

UDAFEVALUATOR

ITERATE() INIT()

TERMINATEPARTIAL()

MERGE()

TERMINATE()

<key, partial agg>

<key,partial agg>

<key, agg>

TERMINATE WILL RETURN THE RESULT VARIABLE

UDAFEVALUATOR

ITERATE() INIT()

TERMINATEPARTIAL()

MERGE()

TERMINATE()

```
SELECT company, Max(OPEN)
GROUP BY company
```

LET'S LOOK AT THE CODE FOR A CUSTOM MAX FUNCTION

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements  UDAFEvaluator {

        private  DoubleWritable partialResult;
        public void init() {
            partialResult = null;
        }




        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(value.get(),partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }

        public boolean merge(DoubleWritable other) {
           return iterate(other);
        }


        public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

SIMPLE UDAF

MAX

```
public class simpleUDAF extends UDAF {
    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(value.get(), partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }

        public boolean merge(DoubleWritable other) {
           return iterate(other);
        }

        public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

A SUBCLASS OF UDAF

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements
UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }
            partialResult.set(Math.max(value.get(), partialResult.get()));
            return  true;
        }
        public DoubleWritable terminatePartial(){
            return partialResult;
        }
        public boolean merge(DoubleWritable other) {
           return iterate(other);
        }

        public DoubleWritable terminate(){
            return partialResult;
        }
    }
```

WITHIN UDAF WE NEED TO IMPLEMENT A UDAFEVALUATOR

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements  UDAFEvaluator {

        private  DoubleWritable partialResult;

        public void init() {
            partialResult = null;
        }



        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(value.get(), partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }

        public boolean merge(DoubleWritable other) {
           return iterate(other);
        }


        public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

WE SETUP A VARIABLE WHERE THE RESULT WILL BE STORED

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements  UDAFEvaluator {

        private  DoubleWriable partialResult;
        public void init() {
            partialResult = null;
        }



        public boolean iterate(DoubleWritable[] ...) {
            if(value == null){
                return ...
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(partialResult.get(), ...));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }

        public boolean merge(DoubleWritable other) {
           return iterate(other);
        }


        public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

AS EACH MAPPER/REDUCER GOES THROUGH THE ROWS, THEY WILL UPDATE THIS VARIABLE

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements  UDAFEvaluator {

        private  DoubleWriable partialResult;
        public void init() {
            partialResult = null;
        }



        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(value.get(), partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }

        public boolean merge(DoubleWritable other) {
           return iterate(other);
        }


        public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

THIS RESULT HAS TO BE OF A PRIMITIVE DATATYPE (SINCE WE ARE USING A SIMPLE UDAF)

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {
    public static class maxevaluator implements  UDAFEvaluator {

        private  DoubleWritable partialResult;
        public void init() {
                partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable()
            }

            partialResult.set(Math.max(value.get(),partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }

        public boolean merge(DoubleWritable other) {
           return iterate(other);
        }


        public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

WHEN A MAPPER/REDUCER FIRST STARTS, THE RESULT VARIABLE IS SET TO NULL

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(value.get(), partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }

        public boolean merge(DoubleWritable other){
           return iterate(other);
        }

        public DoubleWritable terminate(){
            return partialResult;
        }
    }
```

THE ITERATE METHOD WILL GET ONE ROW AT A TIME IN THE MAPPER

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(value.get(),partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }

        public boolean merge(DoubleWritable other){
           return iterate(other);
        }

        public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

THE METHOD NEEDS TO RETURN TRUE FOR THE MAPPER TO MOVE ON TO THE NEXT ROW

SIMPLE UDAF

MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;


        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
                }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(value.get(), partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }

        public boolean merge(DoubleWritable other) {
           return iterate(other);
        }

        public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

IF IT GET'S A NULL VALUE, THEN WE JUST RETURN TRUE AND MOVE ON TO THE NEXT ROW

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;


        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
```

```
if(partialResult == null){
 partialResult = new DoubleWritable();
}
```

```
            partialResult.set(Math.max(value.get(),partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }

        public boolean merge(DoubleWritable other){
            return iterate(other);
        }


        public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

IF IT IS THE FIRST ROW, THEN WE INITIALIZE THE PARTIAL RESULT

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;


        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }
```

```
partialResult.set(Math.max(value.get(),partialResult.get()));
return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }

        public boolean merge(DoubleWritable ot
            return iterate(ot
        }

        public DoubleWritable terminate(){
            return partialResult;
        }
    }
```

WE COMPARE THE CURRENT VALUE TO THE PARTIAL RESULT AND CHOOSE THE MAX

SIMPLE UDAF

MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;


        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }
```

```
partialResult.set(Math.max(value.get(),partialResult.get()));
return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }

        public boolean merge(DoubleWritable other){
            return iterate(other);
        }

        public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

WE RETURN TRUE TO MOVE ON TO THE NEXT ROW

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;

        public boolean iterate(DoubleWritable value) {
            if(value == null)
                return  true;
            }
            if(partialResult == null) {
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(value.get(),partialResult.get()));
            return  true;
        }

        public boolean merge(DoubleWritable other) {
           return iterate(other);
        }

        public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

ONCE THE MAPPER IS DONE WITH ALL THE ROWS, IT WILL CALL THE TERMINATE PARTIAL METHOD

```
public DoubleWritable terminatePartial(){
        return partialResult;
    }
```

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialRes

        public boolean iterate(DoubleWr ab
            if(value == null){
                return  true;
            }
            if(partialResul ==nu
                partialResu he Doub le(
            }

            partialResult.set(Math.max(value.get(),partialResult.get())),
            return  true;
        }

        public DoubleWritable terminatePartial(){
                return partialResult;
            }

        public boolean merge(DoubleWritable other) {
           return iterate(other);
        }

        public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

THE RESULT VARIABLE THAT'S UPDATED IN ITERATE IS RETURNED

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {

    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;

        public boolean iterate(DoubleWritable value) {
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(value.get(),partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
                return partialResult;
        }

        public boolean merge(DoubleWritable other) {
           return iterate(other);
        }

        public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

THE RETURN TYPE SHOULD MATCH THE RESULT VARIABLE TYPE

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {
    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;


        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(value.get(),partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }
```

MERGE AND TERMINATE RUN ON THE REDUCER, BUT USE THE SAME LOGIC AS ITERATE, TERMINATEPARTIAL

```
        public boolean merge(DoubleWritable other) {
                return iterate(other);
            }


        public DoubleWritable terminate(){
                return partialResult;
            }

    }
```

SIMPLE UDAF

MAX

```
public class simpleUDAF extends UDAF {
    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(value.get(),partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }

        public boolean merge(DoubleWritable other) {
                return iterate(other);
            }


        public DoubleWritable terminate(){
                return partialResult;
            }

    }
```

THE RETURN TYPE OF TERMINATEPARTIAL SHOULD MATCH THE INPUT TYPE OF MERGE

SIMPLE UDAF

MAX

```
public class simpleUDAF extends UDAF {
    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;


        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(value.get(),partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }
```

MERGE WILL GET THE RESULT FROM 1 MAPPER AT A TIME AND USE THE SAME LOGIC AS ITERATE TO CALCULATE MAX

```
public boolean merge(DoubleWritable other) {
        return iterate(other);
    }


public DoubleWritable terminate(){
        return partialResult;
    }

  }
```

SIMPLE UDAF MAX

```
public class simpleUDAF extends UDAF {
    public static class maxevaluator implements  UDAFEvaluator {

        public void init() {
            partialResult = null;
        }

        private  DoubleWritable partialResult;

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new DoubleWritable();
            }

            partialResult.set(Math.max(value.get(),partialResult.get()));
            return  true;
        }

        public DoubleWritable terminatePartial(){
            return partialResult;
        }
```

TERMINATE WILL RETURN THE FINAL RESULT FROM THE REDUCER

```
    public boolean merge(DoubleWritable other) {
            return iterate(other);
        }


    public DoubleWritable terminate(){
            return partialResult;
        }

    }
```

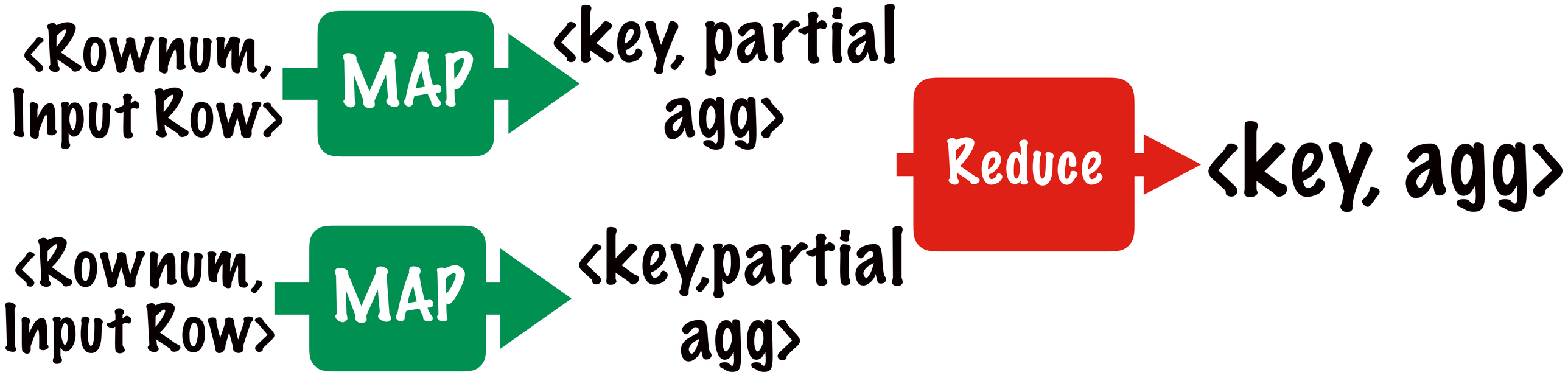
SIMPLE UDAF

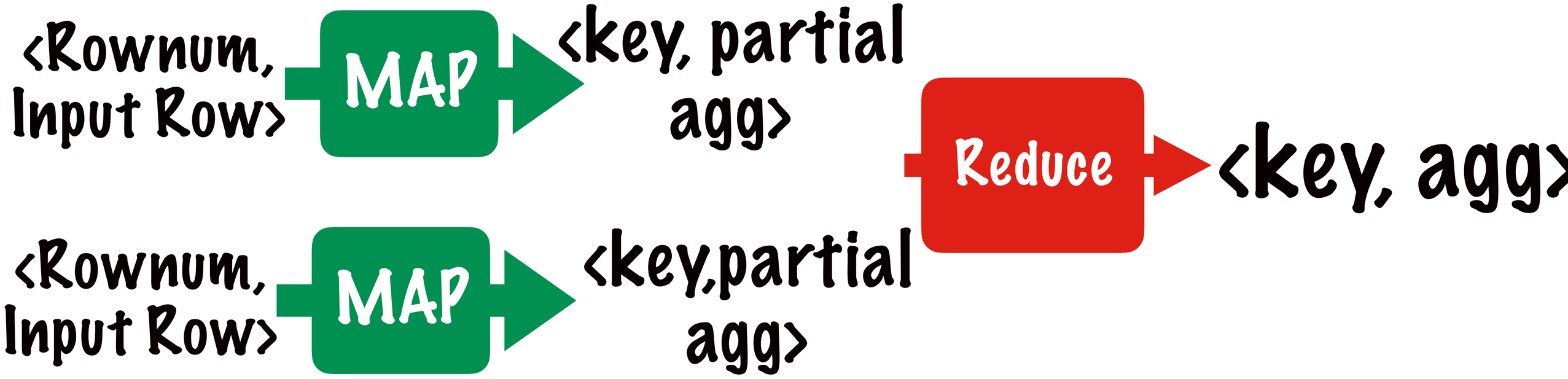
LET'S GO BACK TO OUR PRICES TABLE

Company	Date	Opening	Closing
GM	2015-01-13	24.54	23.67
FORD	2015-01-13	52.54	47.67
GM	2015-01-14	26.54	28.67
FORD	2015-01-14	44.54	47.67

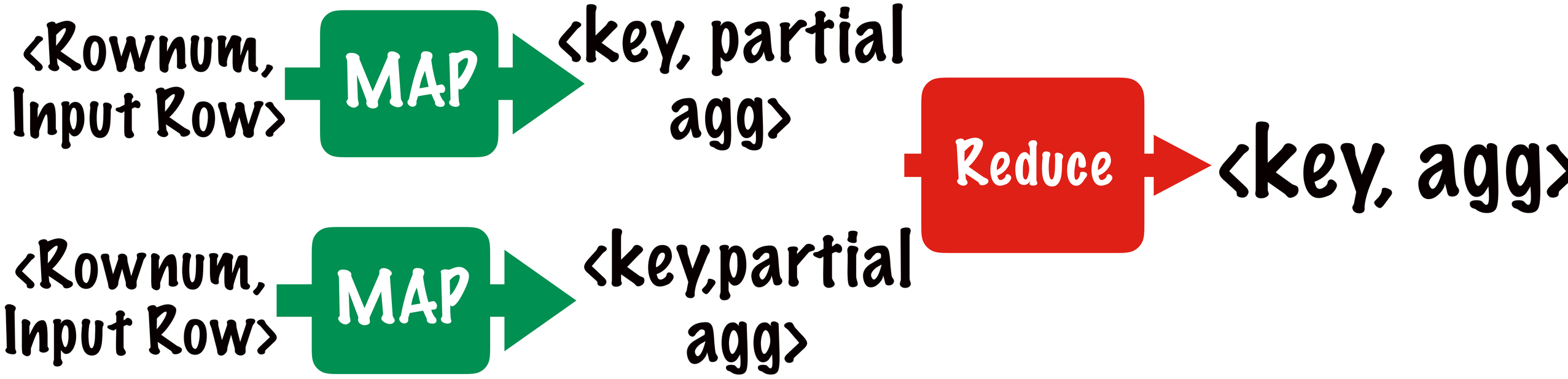
WHAT IF WE WANTED TO CALCULATE THE STANDARD DEVIATION?

THERE WOULD BE A MAP REDUCE JOB

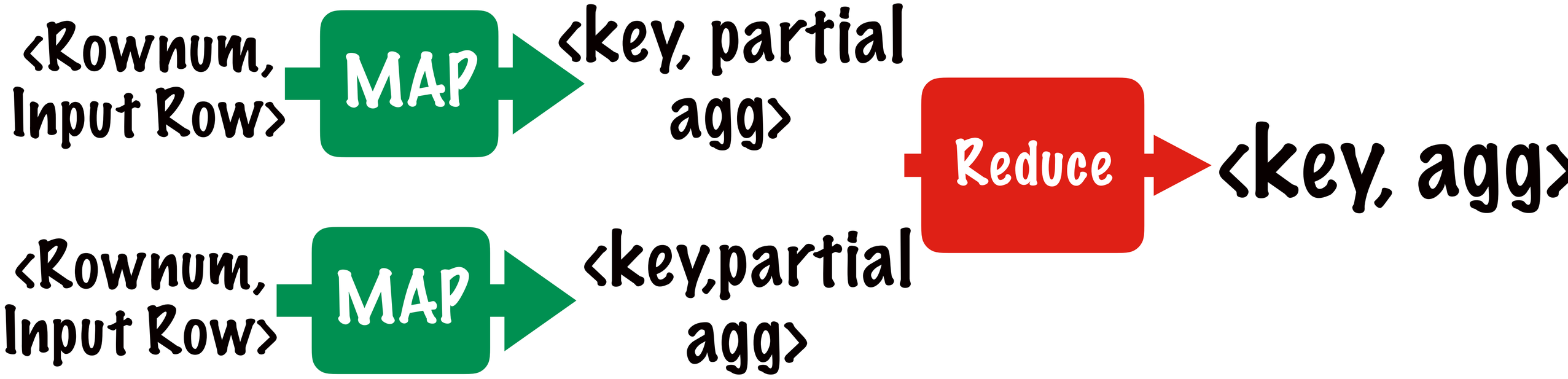




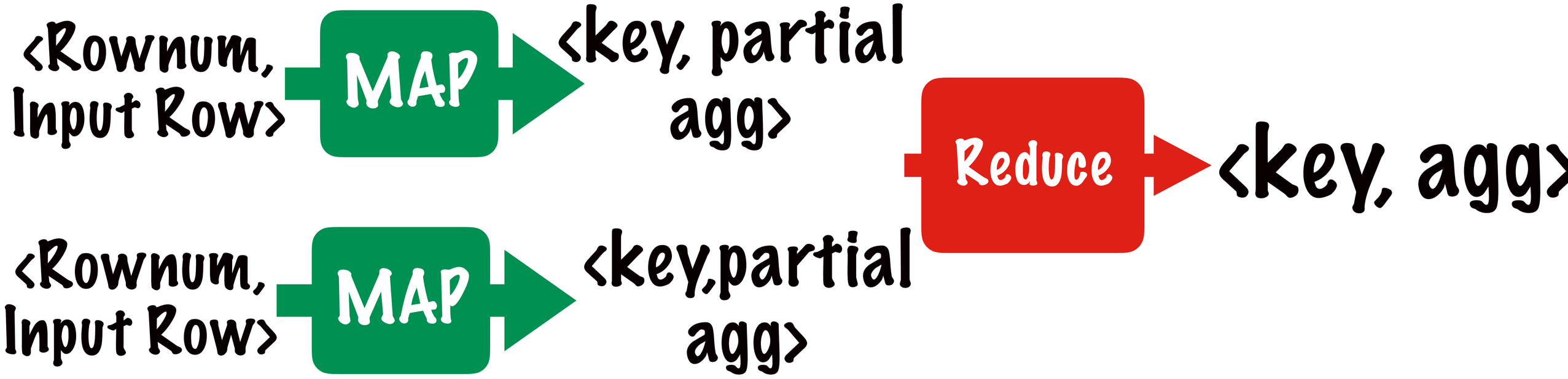
IN THE CASE OF MAX, BOTH THE MAPPER AND REDUCER COULD AGGREGATE USING MAX



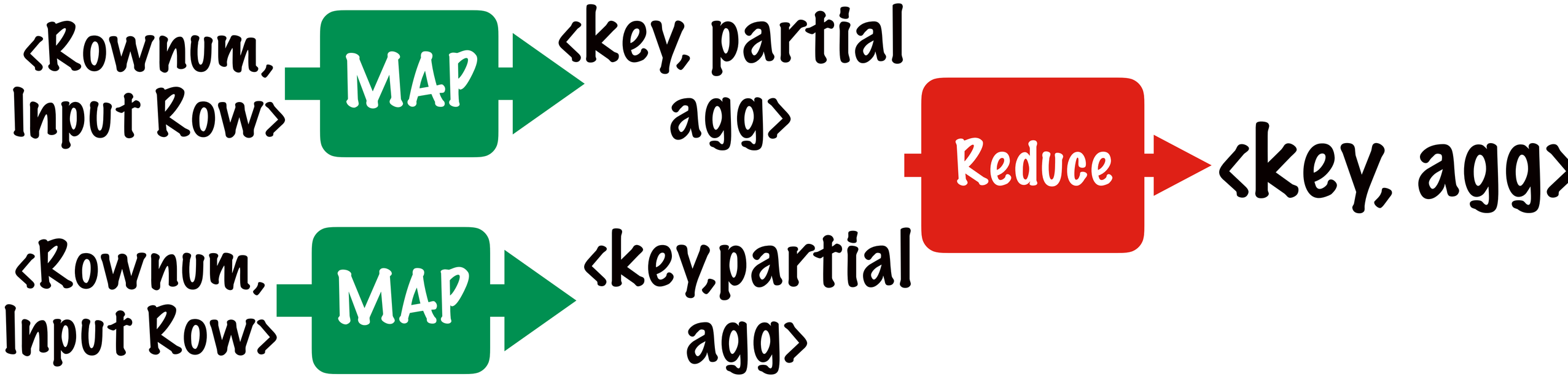
THIS WON'T WORK WITH STANDARD DEVIATION



STD (STD OF PARTIAL LIST) <> STD OF FULL LIST



IN THE PARTIAL AGGREGATION, WE WILL COLLECT A LIST OF VALUES



ONCE WE HAVE THE COMPLETE LIST IN THE REDUCER WE WILL USE IT TO COMPUTE THE STANDARD DEVIATION

LET'S LOOK AT THE CODE FOR STANDARD DEVIATION

SIMPLE UDAF

STANDARD DEVIATION

```java
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

       public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
            double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
            double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }

    }
}
```

SIMPLE UDAF

STANDARD DEVIATION

```
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

        public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
            double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
            double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }

    }
}
```

WE START WITH A UDAFEVALUATOR CLASS INSIDE A UDAF CLASS

SIMPLE UDAF

STANDARD DEVIATION

```
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

            public static class partial_result{
                 int number_of_terms;
                 double sum_of_terms;
                 List<Double> all_terms = new ArrayList<>();
             }

             private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
            double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
            double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }

    }
}
```

WE SETUP A PARTIAL RESULT CLASS TO HOLD OUR INTERMEDIATE RESULT

SIMPLE UDAF

STANDARD DEVIATION

```
public class simpleUDAF extends UDAF {
    public static class stdevaluator implements  UDAFEvaluator {
        public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
            double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
            double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }
    }
}
```

IN THIS PARTIAL RESULT WE WILL KEEP TRACK OF THE

COUNT OF VALUES TILL NOW

SUM OF VALUES TILL NOW

LIST OF VALUES TILL NOW

SIMPLE UDAF

STANDARD DEVIATION

```
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

        public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_t
            partialResult.sum_of_term

            return  true;
        }
        public DoubleWritable terminate
            if(partialResult == null){return  null;}
            double mean = partialResult.sum_of_terms
            double sum_of_squares=0;
            for (double term : partialResult.all_term
                sum_of_squares+=(term-mean)*(term-mean
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }

    }
}
```

THIS PARTIAL RESULT IS CALCULATED AT EACH MAPPER AND PASSED ON TO THE REDUCER

ONCE THE REDUCER GETS ALL THE PARTIAL RESULTS FROM EACH MAPPER, IT WILL CALCULATE A FINAL RESULT

SIMPLE UDAF

STANDARD DEVIATION

```
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

        public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
            double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
            double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }

    }
}
```

WHEN A MAPPER/ REDUCER STARTS WE RESET THE RESULT TO NULL

SIMPLE UDAF

STANDARD DEVIATION

```
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

        public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
            double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
            double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }
    }
}
```

IN THE ITERATE METHOD WE SET UP A NEW PARTIAL RESULT IF IT'S THE FIRST ROW

SIMPLE UDAF

STANDARD DEVIATION

```
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

        public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }
        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
            double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
            double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }

    }
}
```

WITH EACH ROW WE UPDATE THE COUNT, SUM, LIST OF VALUES

SIMPLE UDAF

STANDARD DEVIATION

```
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

        public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
                return partialResult;
            }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
            double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
            double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }
    }
}
```

THE TERMINATE PARTIAL WILL RETURN THE RESULT COLLECTED FROM ONE MAPPER

SIMPLE UDAF

STANDARD DEVIATION

```java
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

        public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;

            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
            double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
            double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }
    }
}
```

THE RESULTS FROM EACH MAPPER GO TO THE REDUCER IN THE MERGE METHOD

SIMPLE UDAF

STANDARD DEVIATION

```java
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

        public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;

            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
            double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
            double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }
    }
}
```

THE MERGE METHOD GETS THE RESULT FROM 1 MAPPER AT A TIME

SIMPLE UDAF

STANDARD DEVIATION

```java
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

        public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;

            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
            double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
            double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }

    }
}
```

WE COLLECT ALL THE PARTIAL RESULTS FROM EACH MAPPER INTO 1 PARTIAL RESULT OBJECT

SIMPLE UDAF

STANDARD DEVIATION

```java
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

        public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
```

THE TERMINATE METHOD IS CALLED ONCE THE REDUCER HAS COLLECTED ALL THE RESULTS FROM ALL MAPPERS

```java
        public DoubleWritable terminate(){
                if(partialResult == null){return  null;}
                double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
                double sum_of_squares=0;
                for (double term : partialResult.all_terms){
                    sum_of_squares+=(term-mean)*(term-mean);
                }

                return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

            }
    }
}
```

SIMPLE UDAF

STANDARD DEVIATION

```
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

       public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
```

```
double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
double sum_of_squares=0;
```

```
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }

    }
}
```

NOW WE HAVE THE FULL SUM, COUNT, LIST OF VALUES

SIMPLE UDAF

STANDARD DEVIATION

```
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

       public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
```

```
double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
double sum_of_squares=0;
```

```
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }

    }
}
```

WE CAN USE IT TO CALCULATE THE STANDARD DEVIATION

SIMPLE UDAF

STANDARD DEVIATION

```
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

       public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
```

```
double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
double sum_of_squares=0;
```

```
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }

    }
}
```

STD = SQRT(SUM OF SQUARES OF THE DEVIATIONS FROM THE MEAN/NUMBER OF TERMS)

SIMPLE UDAF

STANDARD DEVIATION

STD = SQRT(SUM OF SQUARES OF THE DEVIATIONS FROM THE **MEAN**/NUMBER OF TERMS)

```
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

       public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
              double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
              double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }

    }
}
```

SIMPLE UDAF

STANDARD DEVIATION

STD = SQRT(SUM OF SQUARES OF THE DEVIATIONS FROM THE MEAN/NUMBER OF TERMS)

```
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

       public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}


                double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
                double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                        sum_of_squares+=(term-mean)*(term-mean);
                    }


            return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

        }

    }
}
```

SIMPLE UDAF

STANDARD DEVIATION

```
public class simpleUDAF extends UDAF {

    public static class stdevaluator implements  UDAFEvaluator {

       public static class partial_result{
            int number_of_terms;
            double sum_of_terms;
            List<Double> all_terms = new ArrayList<>();
        }

        private  partial_result partialResult;

        public void init() {
            partialResult = null;
        }

        public boolean iterate(DoubleWritable value){
            if(value == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }
            partialResult.number_of_terms +=1;
            partialResult.sum_of_terms = partialResult.sum_of_terms + value.get();
            partialResult.all_terms.add(value.get());
            return  true;
        }

        public partial_result terminatePartial(){
            return partialResult;
        }

        public boolean merge(partial_result other){
            if(other == null){
                return  true;
            }
            if(partialResult == null){
                partialResult = new partial_result();
            }

            partialResult.all_terms.addAll(other.all_terms);
            partialResult.number_of_terms +=other.number_of_terms;
            partialResult.sum_of_terms +=  other.sum_of_terms;


            return  true;
        }
        public DoubleWritable terminate(){
            if(partialResult == null){return  null;}
            double mean = partialResult.sum_of_terms / partialResult.number_of_terms;
     double sum_of_squares=0;
            for (double term : partialResult.all_terms){
                sum_of_squares+=(term-mean)*(term-mean);
            }

    return new DoubleWritable(java.lang.Math.sqrt(sum_of_squares/partialResult.number_of_terms) );

            }
    }
}
```

STD = SQRT(SUM OF SQUARES OF THE DEVIATIONS FROM THE MEAN/NUMBER OF TERMS)

HIVE CUSTOM FUNCTIONS

THE TYPE OF FUNCTION

STANDARD

AGGREGATE

TABLE GENERATING

THE TYPE OF INPUT/OUTPUT

PRIMITIVE

COLLECTION

SIMPLE UDAF

HIVE CUSTOM FUNCTIONS

THE TYPE OF FUNCTION

STANDARD

AGGREGATE

TABLE GENERATING

THE TYPE OF INPUT/OUTPUT

PRIMITIVE

COLLECTION

GENERIC UDTF

GENERIC UDTF

YOU HAVE A TABLE WITH A LIST OF NAMES

PETER TOOLE

SAM AND CATHY GREY

FRED SMITH

THE NAMES INCLUDE BOTH INDIVIDUALS AND COUPLES

GENERIC UDTF

THE OBJECTIVE IS TO CONVERT THIS INTO FIRST NAMES , LAST NAMES OF INDIVIDUALS

PETER TOOLE

SAM AND CATHY GREY

FRED SMITH

PETER	TOOLE
SAM	GREY
CATHY	GREY
FRED	SMITH

GENERIC UDTF

NAMESPLIT() FUNCTION

PETER TOOLE

SAM AND CATHY GREY

FRED SMITH

PETER	TOOLE
SAM	GREY
CATHY	GREY
FRED	SMITH

GENERIC UDTF

THIS FUNCTION TAKES A ROW AND IF IT IS A COUPLE RETURNS 2 ROWS

EACH ROW HAS 2 COLUMNS
FIRST NAME , LAST NAME

THIS IS EXACTLY WHAT TABLE GENERATING FUNCTIONS DO

GENERIC UDTF

EXPLODE() IS AN EXAMPLE OF A BUILT IN TABLE GENERATING FUNCTION

IT CONVERTS ARRAYS/MAPS TO ROWS

GENERIC UDTF

EXPLODE() IS AN EXAMPLE OF A BUILT IN TABLE GENERATING FUNCTION

IT CONVERTS ARRAYS/MAPS TO ROWS

GENERIC UDTF

HOW DO WE WRITE A CUSTOM GENERIC UDTF?

GENERIC UDTF

THIS FUNCTION SHOULD BE A SUBCLASS OF

GENERICUDTF

GENERICUDTF

INITIALIZE()

PROCESS()

CLOSE()

OUR CLASS SHOULD OVERRIDE 3 METHODS

GENERICUDTF

INITIALIZE()

PROCESS()

CLOSE()

THIS METHOD HAS 3 PURPOSES

1. VERIFY THE INPUT TYPES AGAINST THE EXPECTED TYPES

2. SET UP OBJECTINSPECTORS FOR THE INPUTS

3. SET UP OBJECTINSPECTORS FOR THE OUTPUT

GENERICUDTF

INITIALIZE()

PROCESS()

CLOSE()

OBJECTINSPECTOR FOR THE OUTPUT

A TABLE GENERATING FUNCTION GIVES ROWS OF A TABLE AS OUTPUT

GENERICUDTF

INITIALIZE()

PROCESS()

CLOSE()

OBJECTINSPECTOR FOR THE OUTPUT

ROWS OF A TABLE ARE TREATED AS STRUCTS

GENERICUDTF

INITIALIZE()

PROCESS()

CLOSE()

OBJECTINSPECTOR FOR THE OUTPUT

THE STRUCT WILL HAVE 2 FIELDS TO REPRESENT

FIRST NAME, LAST NAME

OBJECTINSPECTOR FOR THE OUTPUT

GENERICUDTF

INITIALIZE()

PROCESS()

CLOSE()

THE OUTPUT OBJECTINSPECTOR IS

STRUCTOBJECTINSPECTOR WITH 2 FIELDS

EACH FIELD IS A PRIMITIVEOBJECTINPSPECTOR

GENERICUDTF

INITIALIZE()

PROCESS()

CLOSE()

THIS METHOD WILL PROCESS EACH ROW AND RETURN THE RELEVANT OUTPUT

GENERICUDTF

INITIALIZE()

PROCESS()

CLOSE()

THIS METHOD IS CALLED WHEN THERE ARE NO MORE INPUT ROWS

ANY CLEANUP CODE CAN BE PUT INTO THIS FUNCTION

GENERICUDTF

LET'S SEE THE CODE FOR THE NAMESPLIT() FUNCTION

GENERIC UDTF NAMESPLIT()

```java
public class genericUDTF extends GenericUDTF {

    private PrimitiveObjectInspector stringOI = null;

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentException("NameSplit() 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
    }
}
}
```

GENERIC UDTF

NAMESPLIT()

```java
public class genericUDTF extends GenericUDTF {
    private PrimitiveObjectInspector stringOI = null;

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentException("NameSplit() 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[0].endsWith(",") && tokens[1] != null && tokens[1].length() > 0) {
            result.add(new Object[]{ tokens[1], tokens[0].substring(0, tokens[0].length() - 1) });
        }else if (tokens[0].length() > 0 && tokens[1] != null && tokens[1].length() > 0) {
            result.add(new Object[]{ tokens[0], tokens[1] });
            result.add(new Object[]{ tokens[1], tokens[0] });
        }
        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
    }
}
}
```

WE NEED A SUBCLASS OF GENERICUDTF

GENERIC UDTF

NAMESPLIT()

```
public class genericUDTF extends GenericUDTF {

    private PrimitiveObjectInspector stringOI = null;

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentException("NameSplit() 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
    }
}
}
```

THIS WILL BE THE OBJECTINSPECTOR FOR THE INPUT

GENERIC UDTF

NAMESPLIT()

```
public class genericUDTF extends GenericUDTF {
    private PrimitiveObjectInspector stringOI = null;

  @Override
      public StructObjectInspector initialize(ObjectInspector[] args)
throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentExc
        }

        // input inspectors
        stringOI = (PrimitiveObjec

        // output inspectors -- an
        List<String> fieldNames =
        List<ObjectInspector> fiel
        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
    }
}
}
```

THE INITIALIZE METHOD NEEDS TO RETURN A STRUCTOBJECTINSPECTOR

GENERIC UDTF

NAMESPLIT()

```
public class genericUDTF extends GenericUDTF {
    private PrimitiveObjectInspector stringOI = null;

 @Override
     public StructObjectInspector initialize(ObjectInspector[] args)
throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentException(
        }

        // input inspectors
        stringOI = (PrimitiveObjectI

        // output inspectors -- an o
        List<String> fieldNames = new
        List<ObjectInspector> fieldO
        fieldNames.add("name");
        fieldNames.add("surname");
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
        return ObjectInspectorFactory.getStandardStructObjectInspector(
    }

    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("a
            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
    }
}
}
```

THIS IS BECAUSE THE OUTPUT FOR THE UDTF WILL BE A ROW REPRESENTED BY A STRUCT

GENERIC UDTF

NAMESPLIT()

```
public class genericUDTF extends GenericUDTF {

    private PrimitiveObjectInspector stringOI = null;

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one
argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() !=
PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentException("NameSplit() 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){
```

WE DO BASIC CHECKS

GENERIC UDTF NAMESPLIT()

```
public class genericUDTF extends GenericUDTF {

    private PrimitiveObjectInspector stringOI = null;

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one
argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() !=
PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentException("NameSplit() 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.getStandardStructObje
    }

    public ArrayList<Object[]> processInputRecord(String nam
```

FIRST TO SEE IF NO OF ARGUMENTS = 1

GENERIC UDTF

NAMESPLIT()

```
public class genericUDTF extends GenericUDTF {

    private PrimitiveObjectInspector stringOI = null;

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one
argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() !=
PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentException("NameSplit() 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){
```

THEN WE CHECK IF THE INPUT ARGUMENT IS A STRING

GENERIC UDTF

NAMESPLIT()

```
public class genericUDTF extends GenericUDTF {

    private PrimitiveObjectInspector stringOI = null;

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentException("NameSplit() takes a string as a parameter");
        }

    // input inspectors
    stringOI = (PrimitiveObjectInspector) args[0];

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

THIS CONDITION CHECKS IF THE OBJECTINSPECTOR IS A SUBCLASS OF PRIMITIVEOBJECTINSPECTOR

GENERIC UDTF

NAMESPLIT()

```
public class genericUDTF extends GenericUDTF {
    private PrimitiveObjectInspector stringOI = null;

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {
            if (args.length != 1) {
                throw new UDFArgumentException("NameSplit() takes exactly one argument");
            }


        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
&& ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
                throw new UDFArgumentException("NameSplit() 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);
}
```

THIS CONDITION CHECKS IF THE PRIMITIVEOBJECTINSPECTOR IS A SUBCLASS OF STRINGOBJECTINSPECTOR

GENERIC UDTF

NAMESPLIT()

```
public class genericUDTF extends GenericUDTF {

    private PrimitiveObjectInspector stringOI = null;

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentException("NameSplit() 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 {

        [illegible]

        [illegible]

        [illegible]

        [illegible]

        [illegible]
    }

    @Override
    public void close() throws HiveException {
        // do nothing
    }
}
}
```

WE SET UP THE INPUT OBJECTINSPECTOR

GENERIC UDTF NAMESPLIT()

```
public class genericUDTF extends GenericUDTF {

    private PrimitiveObjectInspector stringOI = null;

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentException("NameSplit() 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[]>();
        String[] tokens = name.split("\\s+");
        if (tokens.length == 2 && tokens[0].isEmpty())
            tokens = new String[]{tokens[1]};
        if (tokens.length == 2 && tokens[0].length() == 1 && tokens[1].length() == 1)
            result.add(new Object[] { tokens[0], tokens[1] });
        result.add(new Object[] { tokens[2], tokens[3] });
    }

    return result;
}
```

WE SET UP THE OUTPUT OBJECTINSPECTOR

GENERIC UDTF

NAMESPLIT()

```
public class genericUDTF extends GenericUDTF {

    private PrimitiveObjectInspector stringOI = null;

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentException("NameSplit() 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[
        }else if (tokens.length
            result.add(new Obje
            result.add(new Obje
        }

        return result;
    }
    @Override
    public void process(Object
        final String name = string
        ArrayList<Object[]> results = processInputRecord(name);

        Iterator<Object[
        while (it.hasNext
            Object[] r =
            forward(r);
        }
    }

    @Override
    public void close() t
        // do nothing
    }
}
}
```

STRUCTOBJECTINSPECTOR NEEDS THE FIELD NAMES AND DATA TYPES OF THE ROW

GENERIC UDTF

NAMESPLIT()

```
public class genericUDTF extends GenericUDTF {

    private PrimitiveObjectInspector stringOI = null;

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentException("NameSplit() 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
    }
}
}
```

FIELD NAMES ARRAY

GENERIC UDTF

NAMESPLIT()

```
public class genericUDTF extends GenericUDTF {

    private PrimitiveObjectInspector stringOI = null;

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {

        if (args.length != 1) {
            throw new UDFArgumentException("NameSplit() takes exactly one argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
                && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
            throw new UDFArgumentException("NameSplit() 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
    }
}
}
```

THE DATA TYPES OF THE FIELDS

GENERIC UDTF

NAMESPLIT()

```
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] });
    }
```

WE WRITE A HELPER FUNCTION TO PROCESS THE INPUT ROWS

GENERIC UDTF

NAMESPLIT()

```
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] });
    }
```

IT RETURNS AN ARRAY OF OBJECTS, EACH WITH A FIRST NAME , LAST NAME

GENERIC UDTF

NAMESPLIT()

```
}

if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
        && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
    throw new UDFArgumentException("NameSplit() 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] });
    }
```

IT SPLITS THE INPUT NAME INTO WORDS

GENERIC UDTF

NAMESPLIT()

```
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] });
    }
```

IF THERE ARE ONLY 2 WORDS, DIRECTLY ADD THEM AS A NEW OBJECT TO THE ARRAY

GENERIC UDTF

NAMESPLIT()

```
    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] });
        }
```

IF THERE ARE 4 WORDS, IT'S A COUPLE

GENERIC UDTF

NAMESPLIT()

```
}

if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
        && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.STRING) {
    throw new UDFArgumentException("NameSplit() 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] });
    }
```

ADD 2 OBJECTS, FOR EACH MEMBER OF THE COUPLE

GENERIC UDTF

NAMESPLIT()

```
    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
    }
}
}
```

THE PROCESS METHOD WILL CALL THE HELPER FUNCTION WE JUST WROTE

GENERIC UDTF

NAMESPLIT()

```
    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);
        }
```

IT TAKES THE INPUT RECORD AND EXTRACTS THE NAME FROM IT

```
    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 {
```

GENERIC UDTF

NAMESPLIT()

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

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

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

IT THEN CALLS THE HELPER FUNCTION

```
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 {
```

GENERIC UDTF

NAMESPLIT()

```
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);
```

ITERATE THROUGH THE RESULTS OF THE HELPER FUNCTION AND PASS THEM TO THE FORWARD METHOD

```
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 {
```

GENERIC UDTF

NAMESPLIT()

```
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);
```

THE FORWARD METHOD WILL TAKE THIS OBJECT AND OUTPUT A ROW

```
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 {
```

GENERIC UDTF

NAMESPLIT()

```
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);
```

THE ROW WILL HAVE THE SPECIFICATIONS OF THE STRUCTOBJECTINSPECTOR FROM INITIALIZE()

GENERIC UDTF

NAMESPLIT()

```
@Override
public void close() throws HiveException {
    // do nothing
}
```

WE LEAVE THE CLOSE() METHOD EMPTY AS WE DON'T NEED ANY CLEANUP AFTER WE ARE DONE