
# Introduction to RHive

*Do The Math*

**Chicago, IL**
**Bangalore, India**
**www.mu-sigma.com**

**March 15, 2012**

# Agenda

- Introduction to RHive
- Why RHive?
- RHive vs Hive
- Getting Started
- Map Reduce in RHive
- User defined functions (UDFs) using RHive
- Examples
- Exercises

# Introduction to RHive : R and Hive

- RHive is an R extension facilitating distributed computing via Hive query.
- It is a R package that integrates R environment with Hive
- It allows easy usage of HiveQL (Hive Query Language) in R by facilitating usage of R objects and R functions in Hive
- Using RHive, it is possible to write HiveQL in R, launch this query from R, and interact with Hive
- R functions and R objects are exported to Hive and launched in Hive via RHive.
- RHive consists of the following components:
  - rhive – functions to interact with Hive from within R
  - rhive.hdfs – functions to interact with HDFS from within R
  - udf – functions to allow users to use R functions and R Objects in Hive.

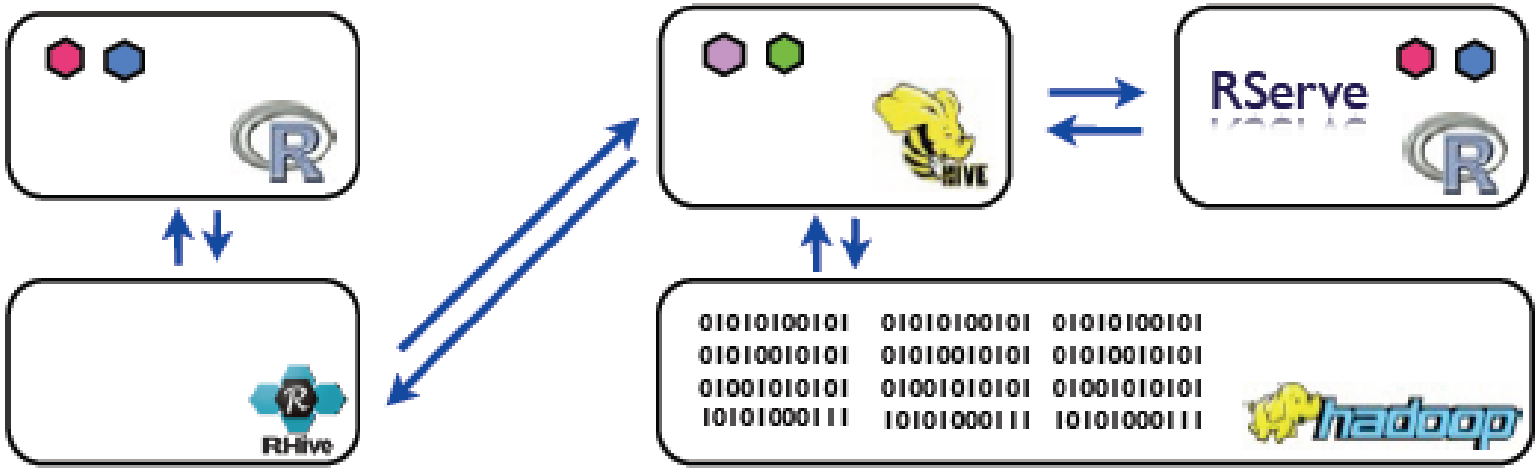
# Agenda

- Introduction to RHive
- Why Rhive?
- RHive vs Hive
- Getting Started
- Map Reduce in RHive
- User defined functions (UDFs) using RHive
- Examples
- Exercises

# Why RHive?

- Many analysts have been using and are familiar with R but R can't support the analysis of data of huge scale
- MapReduce in Hadoop is capable of handling big data of this scale but many analysts don't recognize this framework, less know how to use it
- However, they are more likely to be familiar with using SQL to gain an insight of dataset and preprocessing it
- Like SQL, Hive has an ad-hoc query engine which executes in Hadoop. RHive thereby provides a good solution to handle and analyze big data via integrating R and Hive
- R is the best solution for familiarity, Hive is the best solution for capability. RHive is inspired by this reason, the analysis of BIG DATA

# RHive – Architecture




R Function R Object RUDF RUDAF

Source: http://www.slideshare.net/miloveme/r-hive-introduction

# Agenda

- Introduction to RHive
- Why RHive?
- RHive vs Hive
- Getting Started
- Map Reduce in RHive
- User defined functions (UDFs) using RHive
- Examples
- Exercises

# RHive Vs Hive

- RHive = R + Hive
- Hive is the open source implementation of data warehouse system for Hadoop that facilitates data summarization, ad-hoc queries, and the analysis of big datasets stored in Hadoop compatible file systems
- RHive is an R package that integrates Hive with R
- In RHive, small data is executed in R and the large data is executed in Hive

# Agenda

- Introduction to RHive
- Why RHive?
- RHive vs Hive
- Getting Started
- Map Reduce in RHive
- User defined functions (UDFs) using RHive
- Examples
- Exercises

# Getting Started

- `hive --service hiveserver` `# Run this as a background task to start RHive services)`
- `sudo R CMD Rserve` `# Start Rserve`

On the R console, run the following commands:

- The environment variables must be assigned to the respective home directories of Hadoop and Hive.

```
Sys.setenv(HIVE_HOME="/usr/local/hadoop/hive")
Sys.setenv(HADOOP_HOME="/usr/local/hadoop/hadoop")
```

- `library(RHive)` `# Load the RHive library`
- `rhive.init()` `# Initialize RHive`
- `rhive.connect()` `# Establish a connection`

# RHive Basic Functions

- **`rhive.query`** - execute Hive query in R.

  (ex) `rhive.query("SELECT * FROM Employee")`

- **`rhive.close`** - close Hive connection

  (ex) `rhive.close()`

- **`rhive.list.tables`** - get Hive table list

  (ex) `rhive.list.tables()`

- **`rhive.desc.table`** - get Hive table information

  (ex) `rhive.desc.table("Employee")`

# RHive Basic Functions (Contd.)

- **rhive.load.table** - retrieve table data from Hive to R

  (ex)
```
result <- rhive.aggregate("Employee","SUM","sal",groups="deptno")
   rhive.load.table(result)
```

- **rhive.write.table** - creates R's data frame into Hive and inserts all data

  (ex)
```
rhive.write.table(myDataFrame)  #A table by name myDataFrame is created
```

- **rhive.exist.table** - checks whether the table already exists in Hive

  (ex)
```
rhive.exist.table("Employee")
```

# RHIVE Basic Functions (Contd.)

The following functions are available only from RHive version 0.0-5:

- **rhive.basic.t.test** - runs Welch's t-test on two samples

  (ex) `rhive.basic.t.test("iris","sepallength","iris","petallength")`

- **rhive.block.sample** - creates a new table with data sampling by blocks

  (ex)
  ```
  seedNumber <- sample(1:2^16,1)
  rhive.block.sample("listvirtualmachines",seed=seedNumber)
  ```

- **rhive.basic.scale** - converts numerical data with 0 average and 1 deviation

  (ex) `scaled <- rhive.basic.scale("iris","sepallength")`

- **rhive.basic.by** - runs group by for a specified column

  (ex) `rhive.basic.by("iris","species","sum","sepallength")`

# RHIVE Basic Functions (Contd.)

The following functions are available only from RHive version 0.0-5:

- **`rhive.basic.merge`** - makes new data set from merging two tables, based on their common columns

  (ex) `rhive.basic.merge('iris','usarrests',by.x='sepallength',by.y='murder')`

- **`rhive.basic.mode`** - returns the mode and its frequency within a specified row of the Hive table

  (ex) `rhive.basic.mode('iris','sepallength')`

- **`rhive.basic.range`** - returns the maximum and minimum values within the specified numerical row of the Hive table

  (ex) `rhive.basic.range('iris','sepallength')`

# RHive HDFS Functions

- **`rhive.hdfs.ls()`** – Lists the contents of the HDFS. Does the same thing as "hadoop fs -ls“.
- **`rhive.hdfs.get`** – Brings the data in HDFS to local. This functions in the same way as "hadoop fs-get". (ex) `rhive.hdfs.get("/messages","/tmp/messages")`
- **`rhive.hdfs.put`** – Uploads the data in local to HDFS.
  (ex) `rhive.hdfs.put("/tmp/messages","/messages_new")`
- **`rhive.hdfs.rm`** – Deletes files in HDFS. Does the same thing as "hadoop fs -rm“.
  (ex) `rhive.hdfs.rm("/messages_new")`
- **`rhive.hdfs.rename`** – Changes the filename for files in HDFS, or moves directories. Does the same thing as "hadoop fs -mv“
  (ex) `rhive.hdfs.rename("/messages", "/messages_renamed")`
- **`rhive.hdfs.exists`** – Checks whether a file exists within HDFS.
  (ex) `rhive.hdfs.exists("/messages_renamed")`

# RHive HDFS Functions (Contd.)

- **`rhive.hdfs.mkdirs`** – Does the same thing as "hadoop fs -mkdir“. (ex) `rhive.hdfs.mkdirs("/newdir/newsubdir")`

- **`rhive.hdfs.close()`** – Closes the connection when you have completed using HDFS and no longer need to use it.

# Apply Functions in RHive – *napply()* and *sapply()*

- **napply** - R apply function for Numeric type
  - `rhive.napply(table-name,FUN,col1,...)`
- **sapply** - R apply function for String type
  - `rhive.sapply(table-name,FUN,col1,...)`
- Use the *rhive.load.table* function to view the results in R.

(Ex) R function which sums all passed columns

```
sumCols<-function(arg1,...)
{
sum(arg1,...)
}
result<-rhive.napply("tab",sumCols,col1,col2,col3,col4)
rhive.load.table(result)
```

## *Aggregate Function*

- The *rhive.aggregate* function is used to aggregate data stored in HDFS using HIVE functions
  - `rhive.aggregate(tablename,hiveFUN,...,groups)`
- Use the *rhive.load.table* function to view the results in R.

(Ex) Aggregate using SUM(Hive aggregation function)

```
result<-rhive.aggregate("emp","SUM","sal",groups="deptno")

rhive.load.table(result)
```

# Agenda

- Introduction to RHive
- Why RHive?
- RHive vs Hive
- Getting Started
- Map Reduce in RHive
- User defined functions (UDFs) using RHive
- Examples
- Exercises

# RHIVE MapReduce Function

- **rhive.mapapply** - takes the tables, columns, function as arguments but runs only mapper

```
rhive.mapapply(tablename, mapperFUN, mapinput=NULL, mapoutput=NULL, by=NULL,
args=NLL, buffersize=-1L, verbose=FALSE, hiveclient
=rhive.defaults('hiveclient'))
```

- **rhive.reduceapply** - same as above but performs only reducer

```
rhive.reduceapply(tablename, reducerFUN, reduceinput=NULL,
reduceoutput=NULL, args=NULL, buffersize=-1L, verbose=FALSE, hiveclient
=rhive.defaults('hiveclient'))
```

- **rhive.mrapply** - performs both Map and Reduce steps

```
rhive.mrapply(tablename, mapperFUN, reducerFUN, mapinput=NULL,
mapoutput=NULL, by=NULL, reduceinput=NULL,reduceoutput=NULL,
mapper_args=NULL, reducer_args=NULL, buffersize=-1L, verbose=FALSE,
hiveclient =rhive.defaults('hiveclient'))
```

# RHive MapReduce Example

```
#Word Count Example
#Input Table – mytable, with one column, words.
#Map Function – Read input table row by row and send every word as key with 1
                 as value
map <-function(key,value){
if(is.null(value))
{
  put(NA,1)
}
lapply(value,function(v){lapply(strsplit(x=v, split="")[[1]],
        function(word)put(word,1))})
}

#Reduce Function – Sum the values of all similar keys
reduce <-function(key,values){
put(key,sum(as.numeric(values)))
}

#Call the map-reduce function in RHive
result<-
  rhive.mrapply("mytable",map,reduce,c("NULL","words"),c("word","one"),by="word
  ",c("word","one"),c("word","count"))
head(result)
```

# Agenda

- Introduction to RHive
- Why RHive?
- RHive vs Hive
- Getting Started
- Map Reduce in RHive
- User Defined Functions (UDFs) using RHive
- Examples
- Exercises

# User defined functions (UDFs) in RHive

- UDF (User Defined Function)
- UDAF (User Defined Aggregate Function)
- UDTF (User Defined Table create Function)

# rhive.assign

- The *rhive.assign* function assigns the functions and variables made in R so that they may be referenced from Hive.

  (Ex)

```
newsum<-function(value)
{
  value+1
}
rhive.assign("newsum",newsum)
```

- You can also assign objects that are not Functions.

```
coef1<-3.141593
rhive.assign("coef1",coef1)
```

# rhive.export

- The *rhive.export* function prepares objects made in R by actually deploying them.

  (Ex)

```
sum3values<-function(a,b,c)
{
a+b+c
}
rhive.assign("sum3values",sum3values)
rhive.export(sum3values)
```

# rhive.exportAll

- The *rhive.exportAll* function serves to entirely deploy all symbols starting with the same string for the first argument.

  (Ex)

```
sumAllColumns<-function(prev,values)
{
  if (is.null(prev)){ prev <- rep(0.0,length(values))}
  prev+values
}
sumAllColumns.partial<-function(values) { values }
sumAllColumns.merge<- function(prev,values)
{
  if (is.null(prev)){ prev <- rep(0.0,length(values))}
  prev+values
}
sumAllColumns.terminate<- function(values) { values }
```

# rhive.exportAll (Contd.)

```
rhive.assign("sumAllColumns",sumAllColumns)
rhive.assign("sumAllColumns.partial",sumAllColumns.partial)
rhive.assign("sumAllColumns.merge",sumAllColumns.merge)
rhive.assign("sumAllColumns.terminate",sumAllColumns.terminate)
rhive.exportAll("sumAllColumns")
```

**The last line is actually same as the following:**

```
rhive.exportAll("sumAllColumns")
rhive.export("sumAllColumns")
rhive.export("sumAllColumns.partial")
rhive.export("sumAllColumns.merge")
rhive.export("sumAllColumns.terminate")
```

# RUDF - R User Defined Functions

- TYPE: return type SELECT R (‘R function name’ ,col1, col2, ..., TYPE)
- R function which sums all passed columns

```
sumCols<-function(arg1,...)
{
 sum(arg1,...)
 }

rhive.assign(‘sumCols’,sumCols)

rhive.exportAll(‘sumCols’,hadoop-clusters)

result<-rhive.query(“SELECT R(‘sumCols’,col1,col2,col3,col4,0.0)FROM tab”)

plot(result)
```

# RHive - UDF usage

```
library(RHive)

rhive.write.table(USArrests)

sumCrimes <- function(column1, column2, column3) { column1 + column2 + column3 }

rhive.assign("sumCrimes",sumCrimes)

rhive.export("sumCrimes")

rhive.query("SELECT rowname, urbanpop, R('sumCrimes',murder,assault, rape, 0.0)
  FROM usarrests")

rhive.close()
```

# RHive UDF SQL Vs HQL

- **RHive UDF SQL**

```
rhive.query("SELECT rowname, urbanpop, R('sumCrimes',murder,assault, rape,
  0.0) FROM usarrests")
```

- **Hive SQL**

```
rhive.query("SELECT rowname, urbanpop, murder + assault + rape AS crimes FROM
  usarrests")
```

# UDAF – User Defined Aggregate Function

- RA() function is used to call UDAFs, so SQL's GROUP BY syntax must be used along with it
- Makes use of the RA() function, which returns only one value, and is always of the character type
- Hive processes the returned results and finally sends them to RHive

(Ex)

```
Result<-rhive.query("SELECT species, RA('sumAllColumns', sepallength,
  sepalwidth, petallength, petalwidth) FROM iris GROUP BY species")
Print(Result)
```

```
species
1 setosa
2 versicolor
3  virginica
                                                X_c1
1 250.29999999999998,171.40000000000003,73.10000000000001,12.299999999999995
2  296.8,138.50000000000003,212.99999999999997,66.3
3        329.3999999999999,148.7,277.59999999999997,101.29999999999998
```

# UDAF (Contd.)

- The 2nd column, X_c1, is a value made by UDAF and it consists of character type
- You can also see the values are distinguished by “,”s between them
- To make this back into a numeric vector, R Functions like *strsplit()* must be used. However, even if there are no problems with using that when there is a small number of Records, a problem occurs otherwise
- The example above has only 3 Records but when applying the same procedure for big tables, you might encounter millions of Records
- Hence the values returned by UDAF must be each split and made into column values

# UDTF – User Defined Table Create Function

- In order that values returned by UDAF be split and made into column values, we need sub-queries and UDTF.

  (Ex)

```
result <- rhive.query( "SELECT unfold(dummytable.dummycolumn, 0.0, 0.0, 0.0,
0.0, ',') AS (sepallength, sepalwidth, petallength, petalwidth) FROM (
SELECT RA('sumAllColumns', sepallength, sepalwidth, petallength, petalwidth)
AS dummycolumn FROM iris GROUP BY species ) dummytable")

print(result)
```

| | sepallength | sepalwidth | petallength | petalwidth |
|---|---|---|---|---|
| 1 | 250.3 | 171.4 | 73.1 | 12.3 |
| 2 | 296.8 | 138.5 | 213.0 | 66.3 |
| 3 | 329.4 | 148.7 | 277.6 | 101.3 |

- It can be can seen that the UDAF return values are all split into columns by the “unfold” UDTF

- Unfold is the UDTF Function supported by RHive, so there is no need to separately apply R code

# Agenda

- Introduction to RHive
- Why RHive?
- RHive vs Hive
- Getting Started
- Map Reduce in R
- User defined functions (UDFs) using RHive
- Examples
- Exercises

# Examples

On the R console, perform the following:

I) **Create a table *Employee* and select employees who earn more than $5000 a month.**

```
 > rhive.query("CREATE TABLE Employee(Emp_Id INT, Emp_Name STRING,
Emp_Email STRING, Emp_Salary DOUBLE) ROW FORMAT DELIMITED FIELDS
TERMINATED BY ',' STORED AS TEXTFILE")

 > rhive.query('LOAD DATA LOCAL INPATH "/home/hadoop/employee.csv"
OVERWRITE INTO TABLE Employee')

 > rhive.query('SELECT * FROM Employee WHERE Emp_Salary > 5000')
```

## II) Example - To Predict Flight Delay

```
library(RHive)

rhive.connect()

# Retrieve training set from large dataset stored in HDFS

train <- rhive.query("SELECT dayofweek,arrdelay,distance FROM airlines TABLESAMPLE(BUCKET 1 OUT OF 10000
   ON rand())

train$arrdelay <- as.numeric(train$arrdelay)

train$distance <- as.numeric(train$distance)

train <- train[!(is.na(train$arrdelay) | is.na(train$distance)),]

model <- lm(arrdelay ~ distance + dayofweek,data=train)

# Export R object data

rhive.assign("model", model)

# Analyze big data using model calculated by R

predict_table <- rhive.napply("airlines",function(arg1,arg2,arg3) {

if(is.null(arg1) | is.null(arg2) | is.null(arg3)) return(0.0)

res <- predict.lm(model, data.frame(dayofweek=arg1,arrdelay=arg2,distance=arg3))

return(as.numeric(res)) }, 'dayofweek', 'arrdelay', 'distance')
```



Source: http://www.slideshare.net/miloveme/r-hive-introduction

# Agenda

- Introduction to RHive
- Why RHive?
- RHive vs Hive
- Getting Started
- Map Reduce in R
- User defined functions (UDFs) using RHive
- Examples
- Exercises

# Exercise 1

- Create a table *Customer* with fields *Cust_Id, Cust_Name* and *Cust_Type,* and load data from *customer.csv i*nto it.

- Write a simple RHIVE code to replace all NULL values in *Cust_Type* with the default customer type, *Silver.*

| Cust _Id | Cust_Name | Cust_Type |
|---|---|---|
| 1 | Tom | NULL |
| 2 | Tim | Gold |
| 3 | Larry | NULL |
| 4 | John | Platinum |

The output would look like the following :

| Cust_Id | Cust_Name | Cust_Type |
|---|---|---|
| 1 | Tom | Silver |
| 2 | Tim | Gold |
| 3 | Larry | Silver |
| 4 | John | Platinum |

# Exercise 2

- Write a simple RHIVE code which takes as input a table name & a categorical column name (of that table) and creates (k-1) dummy variable columns for k distinct categorical column values.
  - For example, for the table *Customer* with *Cust_Type* as a categorical variable (with distinct *Cust_Type* values as *Silver, Gold & Platinum* customers) :

| Cust _Id | Cust_Name | Cust_Type |
|---|---|---|
| 1 | Tom | Silver |
| 2 | Tim | Gold |
| 3 | Larry | Silver |
| 4 | John | Platinum |

The output would look like the following :

| Cust_Id | Cust_Name | Silver_dummy | Platinum_dummy |
|---|---|---|---|
| 1 | Tom | 1 | 0 |
| 2 | Tim | 0 | 0 |
| 3 | Larry | 1 | 0 |
| 4 | John | 0 | 1 |

# Appendix

# RHive – Pre-requisites

- Java 1.6
- R 2.13.0
- Rserve 0.6-0
- rJava 0.9-0
- Hadoop 0.20.x (x >= 1)
- Hive 0.8.x (x >= 0)

# References

- https://github.com/nexr/RHive
- https://github.com/nexr/RHive/wiki
- http://cran.r-project.org/web/packages/RHive
- https://github.com/nexr/RHive/wiki/UserGuides
- http://www.slideshare.net/miloveme/r-hive-introduction

# Thank You

**Chicago, IL**
**Bangalore, India**
**March 15, 2012**
**www.mu-sigma.com**