Spark SQL Fundamentals & Cookbook

More than 35 Exercises

Edition 1.0

HadoopExam Learning Resources

BY HADOOPEXAM.COM

Contents

Chpater-1: Apache Spark 6

- Introduction 6
- Interactive mode v/s application mode 6
- Cluster Manager 6
- Spark Framework on CDH 6.x 7
- Cloudera CDH 6 does not support 7
- Common issues while working with Spark 8
- Introduction to Spark Application 8
- Spark Application Execution Model Overview 9

Chpater-1: Spark SQL Introduction 10

- Introduction 10
- Sample program using both SQL Query and API 11

Chapter-2 Catalyst Optimizer 15

- Catalyst optimizer Introduction 15
- Objectives of Catalyst optimizer 15
- Catalyst Library 16
- Internal Representation 16
- Catalyst Tree 17
- Four phases of Catalyst optimization 19
- 1. Analysis phase 20
- **2.** Logical optimization 21
- **3.** Physical planning 22
- **4.** Code Generation 23

Chapter 3: Project Tungsten 25

- Introduction to Project Tungsten 25
- Code generation 29
- CPU Bound operations 29

Chapter-4: Setting up Spark Environment 30

- Compare free vs Paid Version 31
- Part-2 Installing Ubuntu Linux on VMWare 38
- Part-3: Setting Spark 2.0 env on Ubuntu 59

Chapter-5 SparkSQL Schema ... 62
Schema Inference ... 62
Explicitly assigning schema ... 62
Schema Inference using reflection ... 62
Explicitly creating schema using StructType and StructFields ... 64
Chapter 6: SparkSQL abstractions & Other Objects ... 69
About SparkSession ... 69
Submitting Spark applications ... 71
SparkConf object ... 71
Providing custom rules and optimization technique ... 71
SparkSQL Row (Catalyst Row) object ... 72
Resilient Distributed Dataset ... 74
DataFrame ... 74
Dataset ... 76
Dataset ... 77
DataFrame to Dataset conversion ... 78
Dataset and Type-safety ... 78
Dataset and Catalyst optimizer ... 78
Dataset and compile time type safety ... 78
Working with Dataset ... 79
Transient ... 80
Spark Case classes ... 81
Dataset vs RDD operations ... 81
Converting an RDD to Dataset ... 82
Local Datasets ... 82
Dataset and Project Tungsten ... 82
Dataset and Encoder ... 85
Chapter 7: DataFrameReader and DataFrameWriter ... 87
Assigning Schema, while reading the Data ... 89
Handling corrupted records in csv/json file ... 90
Reading a text file as whole ... 90
Setting time Zone for the data ... 90
Reading Data from JDBC data source ... 91

Filtering Data at source only 92
Reading SparkSQL table as DataFrame 92
DataFrameWriter 92
Partitioning and bucketing 92
Bucketing 93
Data Compressions 94
Columns in Dataset 94
Chapter 8: SparkSQL and Hive Support 98
Spark SQL and Hive Query Support 98
Hive Metastore 100
Hive Support in SparkSQL 100
Hive Query support using SparkSQL 100
Chapter 9: SparkSQL and JSON 102
Read JSON data in Spark 103
Example of loading multiple JSON files 104
Explicitly assigning schema to loaded JSON Data 105
Loading JSON data and use SQL query 106
Infer the schema from Data 107
SparkSQL using JSON data full example 108
Chapter 10: SparkSQL and Encoders 110
Implicit Objects 110
Encoders (Serialization and De-serialization) 110
Creating Encoders 111
Hands on Exercise for SparkSQL Encoders 112
Chapter 11: Caching and Checkpointing 114
Dataset and Caching 114
SparkSQL and Caching 114
Checkpointing in SparkSQL 115
Types of Checkpoints 115
Caching (disk only) v/s checkpointing 116
Performance Improvements 117
Other important points about checkpointing 117
Chapter-12: Dataset and Joins 120

Joins Introduction 120
Broadcast Join 124
SparkSQL and Hint 125
Chapter-13: RelationalGroupedDataset 127
RelationalGroupedDataset 127
Multi Dimension aggregations 127
Dataset Aggregation API 127
Hands on Exercises for Multi-Dimensional Operator 133
Chapter-14: SparkSQL Functions 139
Spark SQL Functions 139
Standard or User Defined Functions 139
UDF: User Defined Functions 140
Exercise for User Defined Function and User Defined Aggregate Functions 141
Aggregate functions 143
Collection functions 143
About explode function 144
Date and Time Functions 144
Window Aggregate Functions 144
Non-aggregate functions 147
Sorting functions 149
String functions 150
More Window Functions Example 150
Examples of rank and dense_rank functions (Window function) 151
NTILE (Window) function 152
Cumulative Distribution 153
Chapter-15: Dataset Actions and Transformations 155
Dataset Partitioning 155
About coalesce operator of Dataset 156
Dataset typed transformations 157
Actions on the Dataset 160
Chpater-16: Spark Certifications 162
Databricks Certifications 162
How to prepare for Databricks Spark Certifications 162

Cloudera Hadoop and Spark Developer Certifications 163
Hortonworks Spark Certification preparation material 163
MapR Spark Spark Certifications 164

About book

Apache Spark is one of the fastest growing technology in BigData computing world. It support multiple programming languages like Java, Scala, Python and R. Hence, many existing and new framework started to integrate Spark platform as well in their platform e.g. Hadoop, Cassandra, EMR etc. While creating Spark certification material HadoopExam technical team found that there is no proper material and book is available for the Spark SQL (version 2.x) which covers the concepts as well as use of various features and found difficulty in creating the material. Therefore, they decided to create full length book for Spark SQL and outcome of that is this book. In this book technical team try to cover both fundamental concepts of Spark SQL engine and many exercises approx. 35+ so that most of the programming features can be covered. There are approximately 35 exercises and total 15 chapters which covers the programming aspects of SparkSQL. All the exercises given in this book are written using Scala. However, concepts remain same even if you are using different programming language.

Feedback

This is second full length book from http://hadoopexam.com and we love the feedback so that we can improve the quality of the book. Please send your feedback on hadoopexam@gmail.com or admin@hadoopexam.com

Restrictions

Entire content of this book is owned by http://HadoopExam.com and before using it or publishing anywhere else either digitally on web or printing and distribution require prior written permission from HadoopExam.com. You can use the code or exercises in for your software development or in your software product (commercial as well as open source) and there is no need to take prior permission.

Source code and Data: You can download the source code and data from below location

http://hadoopexam.com/books/SparkSQL/Spark_SQL_Edition_1_Data_and_Source.zip

About SparkSQL

Spark SQL is a module created on top of Spark Core and introduced a data abstraction called DataFrames or Dataset which provides support for structured and semi-structured data. Spark SQL

provides a domain-specific language (DSL) to manipulate DataFrames/Datasets in Scala, Java, or Python (No Dataset in Python). It also provides SQL language support, with command-line interfaces and ODBC/JDBC server. Although DataFrames lack the compile-time type-checking afforded by RDDs, as of Spark 2.0, the strongly typed DataSet is fully supported by Spark SQL as well. More detail about Spark SQL you will find in this book.

Chpater-1: Apache Spark

Introduction

Apache Spark is the distributed framework for the batch and interactive data processing and give high performance computing (HPC) cluster. Currently, Spark exposes the API in the following language

- Java
- Python
- Scala

Interactive mode v/s application mode

Interactive mode is one of the best features provided by the Spark for testing Spark application and before productionizing for regular run. This can even help data scientist who want to run the ad-hoc query. During the data exploration phase, you can check the format, observations (e.g. Rows in table) and features (e.g. columns in a table) of the data.

Cluster Manager

Spark Framework its best performance when it is used in the cluster mode. And for Spark to run in the cluster it requires the cluster manager. There are various cluster manager are supported as below

- Standalone Cluster manager
- YARN (Yet another negotiator)
- Mesos cluster manager

If you are using Cloudera® CDH 6.0 onwards then it supports the only YARN cluster manager. While using the YARN as a cluster manager would give below two major cluster roles are used which are below

- YARN Resource Manager
- Node Manager roles

And with the Cloudera® CDH 6.x onwards standalone cluster manager is no more supported. However, keep in mind that when you use Spark on the Cloudera CDH framework then not all the features of Spark's are supported for example Stand Alone Cluster Manager is no more supported with CDH 6.x

Spark Framework on CDH 6.x

Following Apache Spark components are supported on the Cloudera CDH 6.x

- Spark SQL
  - This is one of the most popular framework in Spark 2.x onwards. Because it is highly recommended that you don't use the Spark RDD APPI until and unless absolute necessary. Using Spark SQL gives lot of advantages like if you already know the SQL (Structured Query Language) then you don't even need to learn Scala, Python and Java programming initially and you can start with it. I said initially because for writing complex Spark SQL application you have to have some knowledge of this programming.
- Spark Streaming
  - There are two part of the Spark Streaming one which was available since Spark 1.x and the other one is Spark structured streaming which was developed after the Spark 2.x which is fully depend on the new Spark SQL framework
    - Older Spark Streaming: Using the RDD framework
    - Structured Streaming: It uses the DataFrame/Dataset (Developed as part of Spark SQL)
- Machine Learning:
  - This is a solution for Machine Learning algorithms.

If you are using Cloudera® CDH 6 then they have removed the support of Spark 1.6 and introduced new framework Spark 2.x as part of their CDH distribution.

Cloudera CDH 6 does not support

There are many features which Cloudera Does not support from Spark and you need to refer the Cloudera Documentation always. Some of the major things I am listing below, which are not yet supported

- JDBC API for accessing Hive and Impala data
- Jupyter Notebooks are not supported
- Spark SQL Command Line Interface not supported
- GraphX is not supported

- SparkR is not supported
- Spark Structured Streaming: Following feature of the Spark Structured stream are not supported
    - Continuous processing
    - Stream static joins with HBase data
- Spark Cost based optimizer is not yet supported

Common issues while working with Spark

Python version

While using Spark 2.x you need to have Python version 2.7 and you need to update the same on all the hosts in the cluster. We have seen many of the Linux host comes with the default version of Python 2.6 , this is required and must do before running code in Spark 2.x

Cloudera CDH5 v/s CDH6

As discussed, Cloudera CDH 6 comes with the Spark 2.x and any Spark application you have written for the CDH 5 (Using Spark 1.x) then you have to migrate to the Spark 2.x compatible API because CDH 6 runs with the Spark 2.x

Introduction to Spark Application

If you already know the MapReduce concept than learning and understanding of the Spark application model is relatively easy for you. Very similar to MapReduce which is a self-contained application which runs the user provided code and do the computation. In the world of Hadoop it reads data from distributed filesystems like HDFS, S3 etc. Spark is also a distributed compute engine and run all the processes concurrently to get the fastest computation. If you compare MapReduce and Spark compute engine then Spark is much better because it has many advantages over the MapReduce like In memory data processing, catalysts optimizer, project tungsten etc.

In simple term MapReduce application (also known as a Job) has divided in three section as below

- **Map phase**: In this data would load from HDFS/S3 or any other supported storage and then Map function would be applied.
- **Shuffle phase**: This phase is optional and depend on what activity you are doing. If you do sorting, groupBy etc then this would be required and most of the time this is place where performance tuning needs to be done. Because in this phase network data transfer is done among the node in the Hadoop cluster.
- **Reduce phase**: In this case reduce function is applied and final calculated data persisted.

In case of Spark the highest level of unit is application. Using single Spark application you have either of the below

- Single batch job
- Interactive session (e.g. Spark-shell) with multiple jobs
- Long-lived continuous job which read new request process and persist the results.

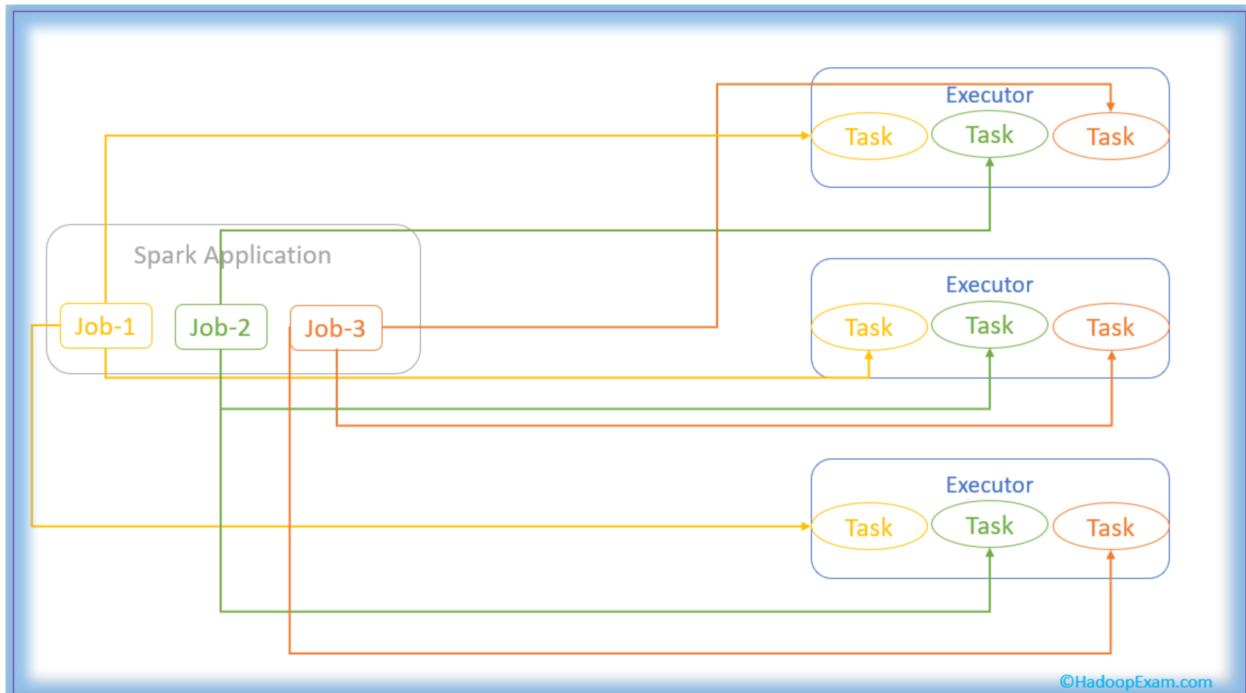
A single Spark application can have more than one Map and Reduce phases.

Spark Application Execution Model Overview

In the Spark execution model you need to understand the following runtime concepts such as

- Driver
- Executor
- Task
- Job
- Stage

It is very much important that you understand this concept before starting any Spark application to make it much more performant. Hence, when you submit your Spark application it requires initially a Driver (A Java process) and many distributed executors (These are also Java processes) distributed across the machines in your Spark cluster.



- **Spark Driver process:** main responsibility of Driver process is to manage your single application (please note it is not for your entire Spark cluster). It is just for your single application. It manages the flow of the job and schedule the sub tasks. If you are running your Spark job using the Spark-shell then spark-shell itself is a single Driver. You can launch more than one Spark-

shell instance and each would be come driver. When we submit Spark application on the YARN (Hadoop cluster) then Driver process may be started on one of the nodes in cluster or outside the cluster. When it is done outside the cluster then it is known as Gateway Node.

- **Executors:** Driver schedules the tasks and those tasks would be executed using the Executor process. Executor would also cache the data locally in-memory or disks (depend on the configuration and availability of the memory). Executor process has a number of slots for using the tasks and all would be run concurrently throughout the lifetime of the job if needed.

Spark has a concept of the transformation and action, so when your application invokes the action it would launch a new job. And each job is further divided into stages which is a collection of the tasks. Each task runs on the different set of data.

Chpater-1: Spark SQL Introduction

SparkSQL Introduction

Sample SparkSQL Exercise

Using SQL interface

Using DataFrame API

Introduction

SparkSQL was developed to provide higher level abstraction to Spark framework with SQL queries as well as using DataFrame/Dataset API. So that developer can easily work with the Spark. Writing code using core Spark, i.e. using RDD (Resilient Distributed Dataset) is quite cumbersome as well as there is no opportunity to automatically optimize the code written using RDD. Spark SQL heavily

depend on the Catalyst optimizer. So whatever code you write in Spark SQL will under the hood go through the various phases of Catalysts optimizer and optimize the code before final execution.

Hence, being a developer or programmer or data scientist you don't have to worry about how to optimize the code if it is written using Spark SQL. But if you have written code using core Spark means using only RDD than it is your responsibility to optimize the code and that is not so easy task.

Spark SQL module provides the opportunity to write code using programming interface as well as SQL query and even in a single program you can use both. There are many other things have been introduced with the SparkSQL which we need to understand as we move ahead like

- Dataset
- DataFrame
- Catalyst Optimizer
- Encoders
- SQL Queries

The most important thing which we need to first learn is about catalyst optimizer. And need to find the answer for following question.

- What is catalyst optimizer?
- Why it was created?
- How it works?
- How it helps in running the SparkSQL code?
- Why Spark operations directly executed on binary data?

Therefore, in next chapter, which is entirely devoted to catalyst optimizer. SparkSQL not only apply the optimization for computation but also it does optimization for storage level as well. Hence, there is one more component which helps in optimizing the SparkSQL computation and storage and that is known as Project Tungsten, and this component was developed to use modern hardware capability as much as possible for running the SparkSQL program.

Sample program using both SQL Query and API:

In below exercise we are creating Dataset object and then we will represent this dataset as a temporary view, so that SQL queries can be executed. In this exercise we will also see, how an RDD can be converted to Dataset. Don't worry if you don't understand this program fully, because we have yet to complete entire journey for SparkSQL. And by the time you finish this book, you will become expert of all this syntax.

Convert RDD to Spark Dataset and Use DSL as well as SQL syntax

```
//Define a Case class for HadoopExam course detail, with the 5 fields
case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)

//Create an RDD with 5 HECourses records.
val courseRDD = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HECourse(4, "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7)))

//Check the types of RDD
courseRDD

//Convert RDD into dataset, as RDD has schema information, so Dataset will automatically infer that schema. As HECourse case class is used to create Dataset, it will be using this case class to infer the schema.
val heCourseDS = courseRDD.toDS

heCourseDS: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

//Select the courses conducted in Mumbai, having price more than 5000
//Also, you can select the columns, you need (It is DSL or programming interface)
val filteredDS = heCourseDS.where('fee >5000).where('venue==="Mumbai").select('name,'fee, 'duration)

//You can see filteredDS is a DataFrame and not a Dataset (There is a slight difference between DataFrame and Dataset since Spark 2.0), we will discuss about this later on
filteredDS

//Lets make code more SQL friendly as it is SparkSQL
//Register Dataset as temporary view and will be added in Catalog
heCourseDS.createOrReplaceTempView("T_HECOURSE")

//Use SQL Query. To select the courses conducted in Mumbai, having price more than 5000
val filteredSQLDS = sql("SELECT * FROM T_HECOURSE  WHERE fee > 5000 AND venue = 'Mumbai' ")

//Show the result
filteredSQLDS.show()
```

See the execution and output of the above program in below image.

```
scala> case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)
defined class HECourse

scala> val courseRDD = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 500
0, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HECourse(4, "Scala", 4000, "Kolkata", 3),HECours
e(5, "HBase", 7000, "Banglore", 7)))
courseRDD: org.apache.spark.rdd.RDD[HECourse] = ParallelCollectionRDD[5] at parallelize at <console>:26

scala> val heCourseDS = courseRDD.toDS
heCourseDS: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

scala> val filteredDS = heCourseDS.where('fee >5000).where('venue==="Mumbai").select('name,'fee, 'duration)

filteredDS: org.apache.spark.sql.DataFrame = [name: string, fee: int ... 1 more field]

scala> heCourseDS.createOrReplaceTempView("T_HECOURSE")

scala> val filteredSQLDS = sql("SELECT * FROM T_HECOURSE  WHERE fee > 5000 AND venue = 'Mumbai' ")
filteredSQLDS: org.apache.spark.sql.DataFrame = [id: int, name: string ... 3 more fields]

scala> filteredSQLDS.show()
+---+------+----+------+--------+
| id|  name| fee| venue|duration|
+---+------+----+------+--------+
|  1|Hadoop|6000|Mumbai|       5|
+---+------+----+------+--------+
```

As you can see in the above program this is one of the biggest advantage that you can write both SQL interface (SQL queries) as well as programming interface (DataFrame API) in a single program and can create a complex pipeline by using the features of from both the interface. As we move ahead we will learn friendlier API and SQL queries functions.

Chapter-2 Catalyst Optimizer

- Catalyst optimizer Introduction
- Objectives of Catalyst Optimizer
- Optimization techniques
- Catalyst Library
- Scala Features used in Catalyst
  - Pattern Matching
- Catalyst phases
  - Analysis Phase
  - Logical Optimization
  - Physical planning
  - Byte Code Generation

Catalyst optimizer Introduction:

Catalyst optimizer is the heart of SparkSQL, whether you are using Python, Scala, Java, or R language to run SparkSQL code using either SQL queries, Dataset/DataFrame API or Dataset Lambda functions all are processed by Catalyst optimizer. Optimizer goes through four phases before submitted code is getting executed. Even while going through four phases, it makes sure your code runs fast and optimally on distributed cluster.

To do the optimization Catalyst uses various Scala features like Scala pattern matching, quasiquotes etc. which is based on functional programming construct of Scala.

Objectives of Catalyst optimizer:

1. **Optimization technique:** Adding new optimization techniques to the catalysts optimizer or to SparkSQL module should not be complicated process and must be easy.
2. **Extending Optimizer**: As a user or developer you should be able to add new rules which are specific to your data, as well as support for new data types can be added by you.
    a. **Data specific rules:** By which you should be able to push filtering or aggregations into newer external storage which are not already supported. Many common ones are already supported by SparkSQL itself e.g. JDBC sources Oracle, MySQL etc.

b. You define your own custom data types than you should be able to create Encoders (serialization and de-serialization : Learn from training) for these newer data types.

Optimization techniques: Catalyst optimizer supports two types of optimization techniques in various phased as below

1. Rule based optimization
2. Cost based optimization

Catalyst Library: Catalyst framework has its own library and many of the objects, features, API you can use to extend the framework.

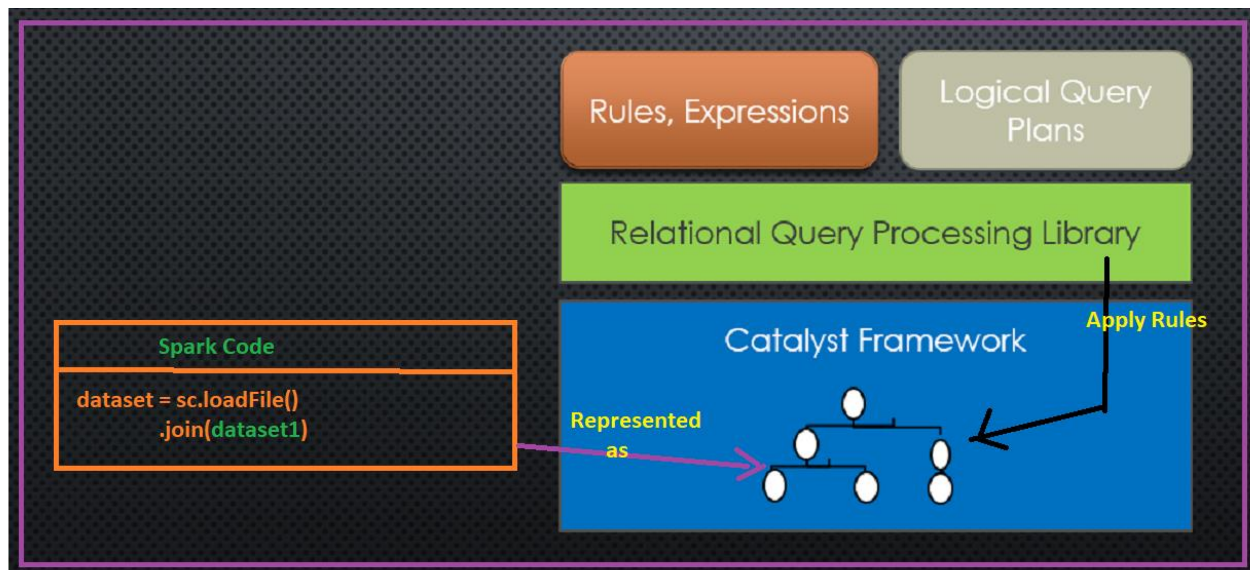


Figure 1: Spark SQL Catalyst Framework

Internal Representation: Spark SQL Code is internally represented as a Tree as you can see in above image. Catalyst optimizer, and all the rules are applied on these trees which will be transformed. Tree are immutable objects, so if you apply a rule on a tree (Which is representing your SparkSQL code) it will be transformed in to new tree. Trees are made of Scala TreeNode object.

- **Abstract Source Tree**: In catalyst optimizer these trees are called Abstract Source Tree.
- On top of Catalyst framework another library created which is specific to relational query processing, and used for
    - **Query processing**: In this expressions and logical query plans are created.
    - **Rules**: Set of rules which will be used when your Spark SQL code goes through various phases as below and even you can add your own custom rules. There are mainly four phases through which your Spark SQL code goes through before final execution, we will discuss them in more detail in next section
        1. Query analysis
        2. Logical optimization of the query
        3. Physical planning and generate one or more than one physical plan

QuickTechie

4. Code generations: generate Java Bytecode, finally your query will be executed on the JVM only.

Catalyst Tree: To understand more lets create a Catalyst Tree for small Spark SQL expressions as below.

```
val he="HadoopExam"
he :: ( lit("Learning" ) :: lit("Resources"))
```

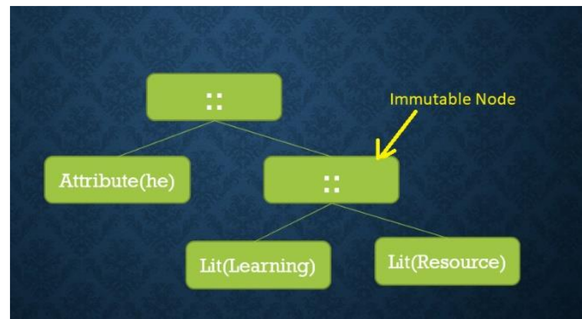


Figure 2: Catalyst Tree for above expression

- As you can see in above image, a node can have zero or more child nodes.
- Nodes are subclass of Scala TreeNode class.

Rules: As we have already discussed Tress are manipulated using rules. Each node in Tree is immutable as well as entire Tree is immutable. As you can see in below image Tree is transformed by applying Append (::) expressions.

Figure 3: Applying Rules to Catalyst Tree and New Tree created with Transformation

Pattern Matching: It is a feature of Scala programming, if you know Java programming Switch Case statement than it is easier to understand pattern matching, however it is much powerful than Java switch case. To understand pattern matching, you need to understand Scala apply and unapply method as well.

Apply and Unapply methods in Scala:

*If you want to learn more about [Scala programming consider online recorded training](http://hadoopexam.com) at http://hadoopexam.com

Let's have a class called HECourse as below

```
//Create a class, which represent Data with the Two fields name and fee of the course
class HECourse(val name:String, val fee:int)

//Define a companion object and implement or define an apply and unapply function
object HECourse{
	def apply(name : String, fee:int) : HECourse = new HECourse(name, fee)

	def unapply(heCourse : HECourse) : Option[(String, Int)] = {
		if(heCourse.fee == 0) None
		else Some(person.name, person.age)
	}
}

//Create an Object of Class HECourse and it does not require new keyword as we need in Java
//When you create an Object, Scala internally calls the apply method. Defined in Companion object.
//This apply method will create an Object of given class.
val hadoopCourse=HECourse("Hadoop Online" , 6000  )
val sparkCourse=HECourse("Spark Online" , 7000  )

//Unapply method will take input as an Object and extracts the values from it. Means, you provide a
HECourse object
//Which in turn extract the name and fee of that object.
//Return type of unapply will always be an Option type.
//If values are extracted than it will return Some object else None. However, it depends, how
implement it.
```

Scala Switch Case statement: Let's learn pattern matching

```
//Define a ,function such that
//Based on course name get the Fee value if course name does not matches
//return a default fee 5000
def getFee(Choice : String) : Int = choice match{
	case "Hadoop" => 6000
	case "Spark" => 7000
	case _ => 5000
}
```

Some other complex pattern matching:

```
// Based on other complex choices
def getHECourse(Choice : Any) : HECourse = choice match{
        case "Hadoop" => HECourse("Hadoop",6000) // If name of the course is Hadoop
        case "Spark" => HECourse("Spark" ,7000) //If name if the course us Spark
        case (x,y) => HECourse(x,y) //If input is tuple
        case (i : Int) => HECourse("Other", 5000) //If input type is Int
        case (i : String) => HECourse(i, 5000) //If single value with String
        case Map[_,_] => HECourse(m.key, m.value) //If choice type is Map
        case _ => HECourse("Other", 5000) // If no pattern matches
}
```

Based on pattern matching Catalyst Tree are transformed from one form to another form, by applying various rules.

In below example we can see, how a transformation is applied. In the given image we have a Rule for Add expression. That rule will be applied on Tree until all the "Add" expressions are transformed that would be done iteratively. Once done a final Tree would be returned.

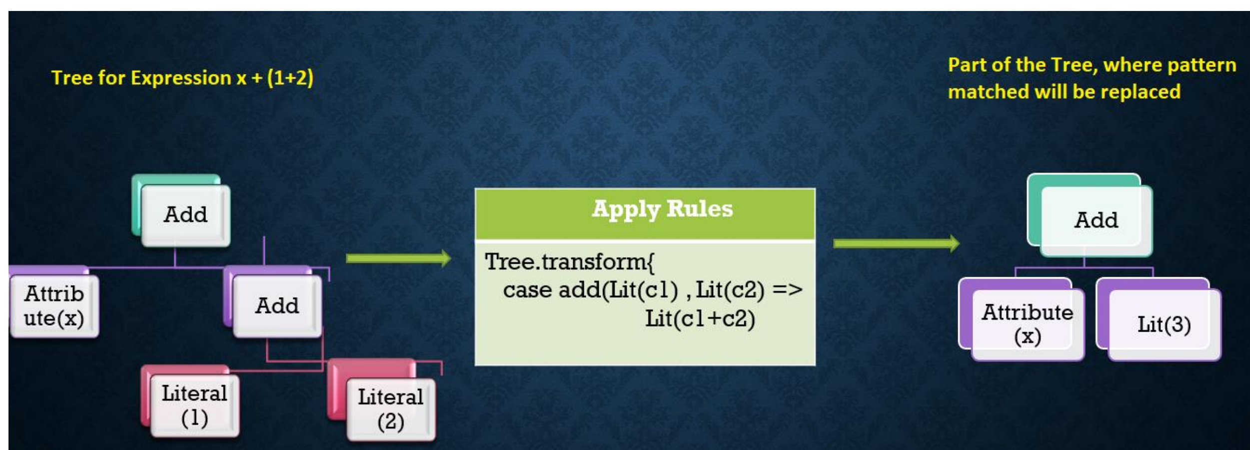


Figure 4: Applying transformation on a Catalyst Tree

Remember:

- Rules will be applied many times until all the pattern matching completed.
- Always new Tree will be created for each transformation. Because Trees in Catalyst are immutable.

Four phases of Catalyst optimization: Catalyst optimization has four phases as below.

1. **Analysis Phase**: Analyzing logical plan and resolve the references by applying rules.
2. **Logical phase**: Optimizing logical plan by applying rules.
3. **Physical planning**: From logical plans create one or more than one physical plans and out of which one will be selected based on lowest cost (cost will be calculated based on CPU, Network I/O and Memory)
4. **Code generation**: generate bytecode to be run on the JVM.

In each of the above phase different types of Tree nodes will be used as below

- Noes for expressions and known as Expression Nodes
- Noes for DataTypes
- Nodes for logical operators.
- Node for Physical operators.

Let's discuss each phase in detail

1. Analysis phase: In analysis phase, expression from either DataFrame API or SQL query will be represented as an Abstract Source Tree.

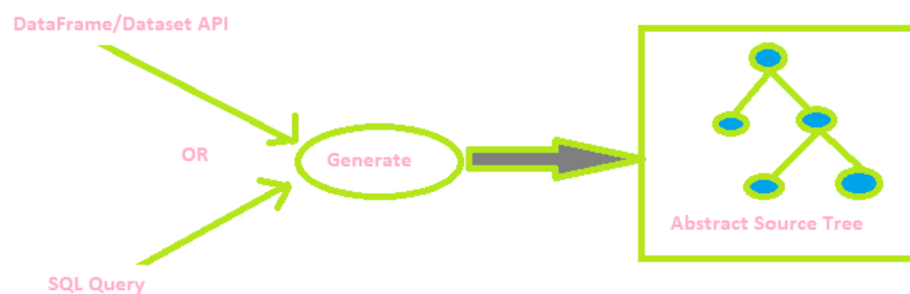


Figure 5: Representing SQL Query or Dataset API expressions as an Abstract Source Tree

Let's take an example of query as below

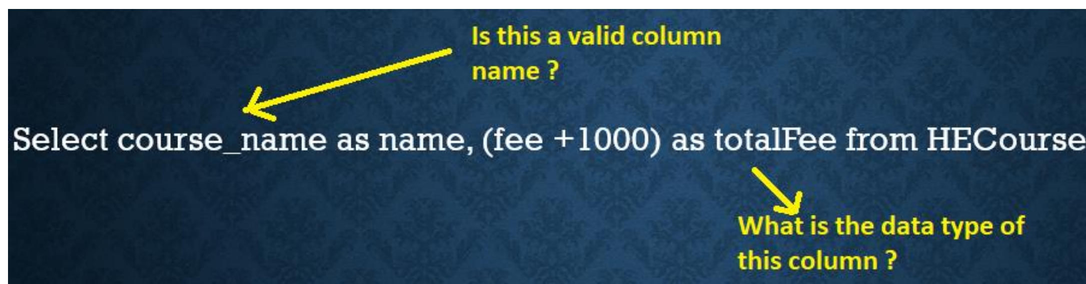


Figure 6: Query example for Analysis Phase of Catalyst

- We need to check whether this table (course_name) exists in catalog or not.
- If we don't know the types and valid name of selected columns than the plan generated will be called unresolved plan.
- **Resolve this plan**: To resolve table name, column validity and data types of the column are done in this phase. Once plan is resolved it is known as resolved plan. To resolve the plan following object will be referred.
    - **Catalog object (This is similar to Hive catalog, in Hive Framework)**
    - **Catalysts rules**

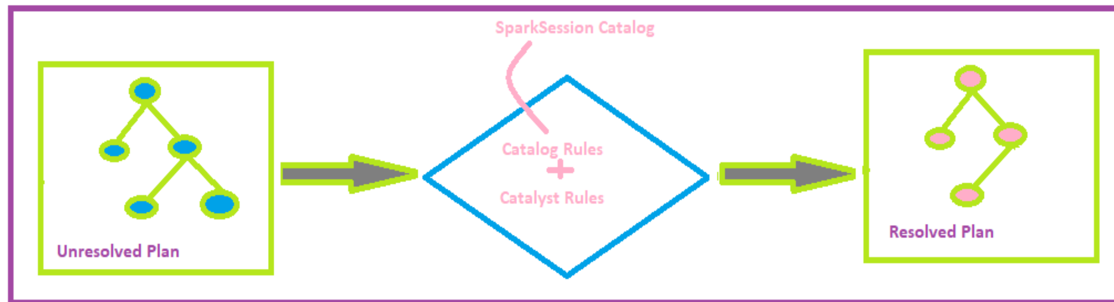


Figure 7: Applying rules convert unresolved plan to resolved plan in analysis phase of Catalyst

- During resolution, check for relations in catalog table.
- Map with the actual column name.
- Give unique_id to each attribute e.g. to course_name. Hence, it can further used for optimization.
- Finding the types for expressions

Figure 8: Finding the data types during expression evaluation

2. Logical optimization**:**

- This is a rule-based optimization on already resolved plan from analysis phase.
- It uses standard Scala features like below
  - **Constant folding:** It is a compilation technique in Scala. It helps in computing the expression in one shot for all the rows and not required to be executed for each row.
  - **Predicate pushdown:** Predicate means where clause of SQL Query or applying filters and where condition on the Dataset API. Spark will first check this predicate and directly send these predicates to the data source like CSV file, JDBC data source and Hive data source etc. Which helps data to be filtered at the source only and less data will be transferred over the network from the source.

Figure 9: Predicate pushdown, will send only filtered data to Spark

As of now it still does not support many predicates for the JDBC sources like limits, group by, data sorting etc. and these will be applied only after loading data in Spark.

- **Projection pruning:** Most of the time data from a table required to be selected from few columns and not all the columns. Like calculating aggregate values avg(), min(), max() etc. So, it will select only required columns and apply the aggregate functions. This is known as project pruning.

- **Physical operators:** As code is written using SQL queries and Dataset API with the supported operators. However, finally code will be executed over the RDD. Hence, actual operator of RDD needs to be derived before applying on RDD.

- If required new rules can be added.
- Similar to previous phase it will create another transformed optimized version.

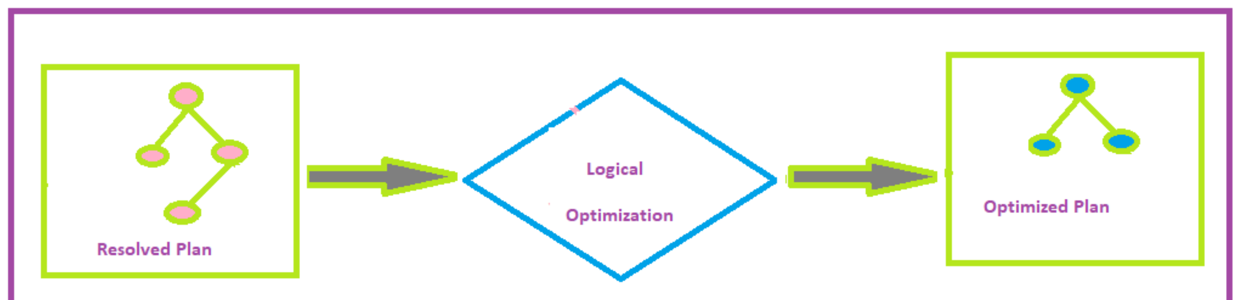


Figure 10: Applying logical plan on Resolved Plan

3. Physical planning**:** It will generate one or more than one physical plan. This is the phase in which cost based optimization will be applied to select the optimized physical plan.

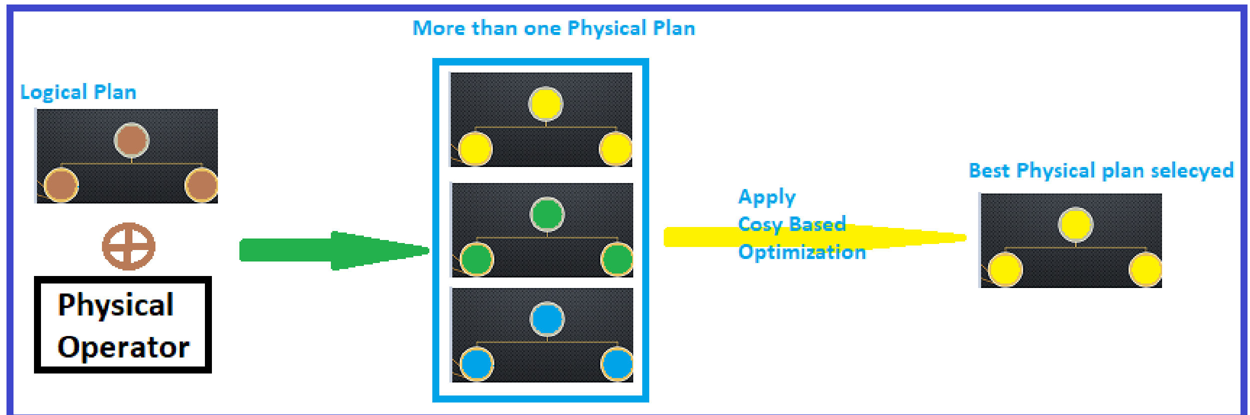


Figure 11: Cost based optimization to select best plan

Let's say you are joining Datasets and one of them is larger dataset (more than 100MB) and other one is smaller one (10 MB). Then Catalyst will select the plan which has broadcast join in it. Because dataset with smaller size will be distributed on each node and joins will happen locally.

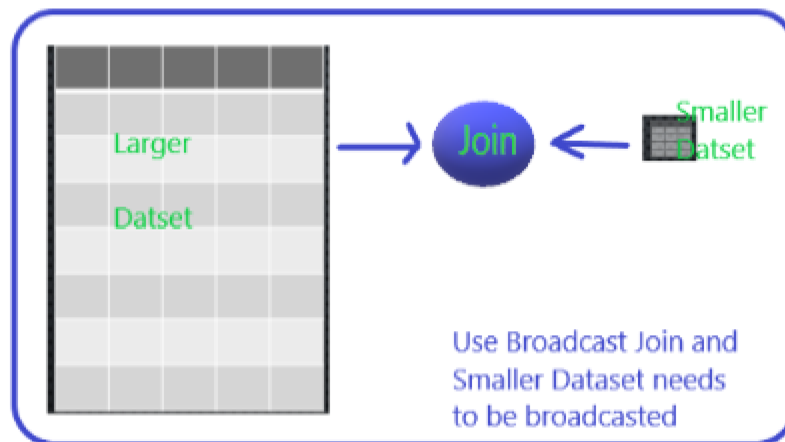


Figure 12: Broadcast Join between Larger and Smaller Dataset

4. Code Generation: In this phase Java byte code will be generated for various part of SQL queries and Dataset API. As mentioned previously in this phase Spark uses a Scala feature called Quasiquotes to generate Java Bytecode.

- If we run the code on each node without generating Java Bytecode than the processing will be slow. Suppose you are working on 100M of rows, than SparkSQL expression will be executed for each Dataset Row by iterating over it and that would be very expensive. Using Quasiquotes expression will directly applied on the data without copying objects and iterating.

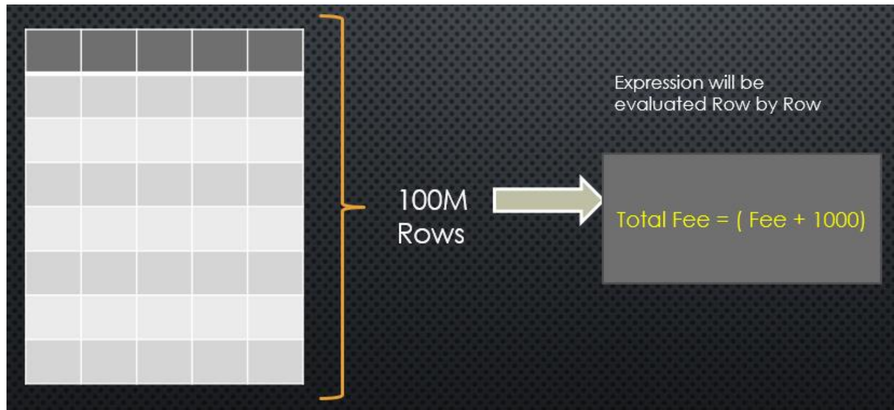


Figure 13: Running without code generation

Chapter 3: Project Tungsten

- Tungsten Introduction
  - Explicit Memory Management
  - Binary Data Processing
  - Cache aware computation
  - Code Generation for expressions

Introduction to Project Tungsten

Project Tungsten was developed to leverage the modern hardware capability and core focus was on memory and CPU usage by Spark. As you know day by day CPU are also improving and capacity of L1/L2/L3 cache of the CPU is increasing.

Let's assume it if you are working with the 250 node of Spark Cluster than how much overall CPU cache is available to you. Assuming each node has 8 core CPU than in total 250X8=2000 Cores are available. Each core can have 256KB CPU cache (L1/L2/L3) than total cache volume is available to you is 500MB which is ultra-fast, because it is attached to your CPU and help you to store your most frequently used data as well as during sorting and hashing it can be used. Hence, this is one of the example how this Project Tungsten is focusing and leveraging modern hardware for achieving high performant compute cluster.

Even Spark by-passes the in built features of garbage collection mechanism of java to improve the computation performance.

Following four were the main area of focus for Project Tungsten

1. Explicit Memory Management
2. Binary Data Processing
3. Cache aware computation
4. Code Generation for expressions

We will discuss each one in detail in next section.

Therefore, there are mainly three areas of improvements.

1. Network I/O and Disk I/O

2. In memory (RAM) storage
3. Leveraging CPU caches

Spark team had proved that improving on Disk and Network I/O overall gives 20% performance improvements but if you need more performance than you have to leverage the modern CPU caches as well and push the calculations as close to hardware as possible using L1/L2/L3 caches. As part of Project Tungsten entire focus was optimizing RAM and CPU cycles. Let's see each optimization technique one by one with little more detail.

Explicit Memory Management:

As you know Spark framework code is written using Scala and Scala code compiles to Java Bytecode and then finally run in JVM (Java Virtual Machine). JVM gives a lot of features to manage the object lifecycle from creation to destroy as well as platform independence etc. For general purpose applications and even all the enterprise application it is good to rely on this JVM features. Because lot of things which you will be doing like in C language allocation and de-allocation of memory is taken care by the JVM itself and even JVM default object life cycle management is also good enough for almost all enterprise and general-purpose applications. Until and unless you need ultra-high-speed computing like Spark computations on big data, low latency trading etc. For some extent you can optimize the Garbage collection algorithms of Java as well and that may be good enough for your application.

Let's see some basics of Java Garbage Collection mechanism in general

Figure 14: Java Object Garbage Collection Phases

As you can see in the above diagram, when you create an object it is first placed in the Eden space and whenever GC runs and if the object does not have references, it would be destroyed. But if still has references it would be moved to "Survivor-0" space. And same algorithm is applied, if object is still surviving it will be moved to the "Survivor-1" and at the end object will reach in the Old space. Hence, if object is retained for longer time and even on GC runs on space not frequently as it is done on Eden space. Which causes the object remain in the Old generation even if it does not have references. This is what happens in general with the GC algorithm.

This all mechanism is a big overhead while working with the Big Data and machine learning intensive computations. And Spark team decided they will not use this mechanism of GC and by-pass this al-together. And do the explicit memory management.

To do the explicit memory management in Java, there is an API available under the package called "sum.com.unsafe" package.

In one of the "Databricks blog" we found example as below if you have to store a String object in JVM with its default behavior it will take much more space than actually required for instance, if you have an String object as below

```
String str="Exam"
```


QuickTechie

Here "Exam" is a 4-character string and it should take only 4 bytes of UTF-8. But when internally JVM load this object in memory it will take around 48 Bytes, which is huge space consumption. Hence, this is one of the reason Spark team wanted to manage objects their own using various algorithms for explicit memory management from the API of "unsafe" package.

Binary Data Processing

While working with the data, Spark team prefer not to directly work with the JVM objects, rather they want to convert the object in pre-defined binary format and then work on it. Let's take an example of few records which we have loaded from csv file to SparkSQL Row objects as below.

ID	Course_Name	Training_Venue
101	Hadoop	Mumbai
102	Spark	Pune
103	Java	Bangalore
104	Scala	Hyderabad
105	Python	Chennai

Figure 15: Sample SparkSQL Row Data

When you represent this data in JVM then one of the row will be created as below. Let's take first record as an example which is stored as objects in JVM, which will take 5 objects in total to store a single record, and which can consume huge memory space as well.

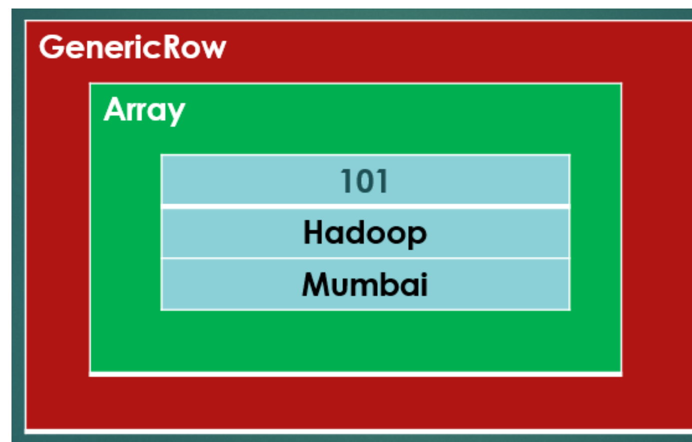


Figure 16: Representing a SparkSQL record using JVM objects

But Spark team will not use the storage as above and change it to some fixed define format, as flat data structure in memory as below.

Figure 17: SparkSQL each record should be stored in this format

Hence, if we take first record than it will be saved in following format

Figure 18: Internal representation of SparkSQL Row/Record

As you can see in the given data, there is no value which is null, hence first column will occupy only 0 length. Similarly, course_id=101 will take some fixed length which is as per the Int size binary format and next 48L is the length of course name represented as "Hadoop" with the fixed length as 48 and offset is 6. Similarly training venue is Mumbai, will be having fixed length 48 and offset as 6. This is the way all the records would be converted internally by Spark. And also, this is stored as binary format with the help of Encoders (we will see this in next chapter). Storing in binary format will not only save space in-memory but also helps you to do computation very fast without even iterating all the records. These rows size would always in the multiple of 8 bytes.

Hence, all the operations would be done on this binary data only. Which will help in avoiding serialization and de-serializations of the records and reducing lot of CPU cycles. Similarly, equality comparison and hashing, sorting all will be done on this binary data without converting or interpreting in any other formats.

Cache aware computations**:** In new and modern hardware we know there are various caches are available which are close to CPU and much faster to work on the data which stored in this memory. Data which is closer to CPU, computation on that data will be faster. In new modern hardware, we are having L1, L2 and L3 caches available. L1 case is closer to CPU and embedded on the same chip.

About CPU Cache (From Wikipedia)

A CPU cache is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory. A cache is a smaller, faster memory, closer to a processor core, which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2, etc.). All modern (fast) CPUs (with few specialized exceptions) have multiple levels of CPU caches. The first CPUs that used a cache had only one level of cache; unlike later level 1 caches, it was not split into L1d (for data) and L1i (for instructions). Almost all current CPUs with caches have a split L1 cache. They also have L2 caches and, for larger processors, L3 caches as well. The L2 cache is usually not split and acts as a common repository for the already split L1 cache. Every core of a multi-core processor has a dedicated L2 cache and is usually not shared between the cores. The L3 cache, and higher-level caches, are shared between the cores and are not split. An L4 cache is currently uncommon, and is generally on dynamic random-access memory (DRAM), rather than on static random-access memory (SRAM), on a separate die or chip. That was also the case historically with L1, while bigger chips have allowed integration of it and generally all cache levels, with the possible exception of the last level. Each extra level of cache tends to be bigger and be optimized differently.

Spark uses various algorithms which can leverage this memory hierarchy in algorithms like

- Sort based shuffle
- High Cardinality aggregations

- Sort-merge join operator

Code generation: Using Scala feature called Quasiquotes code will be generated for various part of Spark SQL queries and expressions in Dataset API. This again exploits the modern compilers and CPUs.

CPU Bound operations: As we know Spark is a distributed computation engine. Hence, most of the work is CPU bound and most of the expensive tasks are data shuffling which involved below two steps

1. Serialization and Deserialization
2. Hashing

Both the above steps are CPU bound and if not handled properly than it can impact the overall performance.

Let's see we have expression as below

Fee > 5000 && Fee < 10000

While computing this expression on all the rows, it will not iterate over each row. Rather Spark framework at runtime dynamically generates the bytecode for evaluating this expression, which can help in reducing boxing of primitive data as below.

Int I =5000;	Integer I = new Integer(5000)

For most of the in-built expressions Spark apply this code generation and would be applied on many places like In memory.

Chapter-4: Setting up Spark Environment

- VMWare workstation installation
- Installing Ubuntu Linux on VMWare
- Setting Spark env on Ubuntu

In this chapter we will go through all the required steps for setting up single node Spark Environment. You can use this environment to run all the exercises which are given in this book. Instructions provided in this chapter are applicable to Windows based operating system. However, similar approach you can take with the Mac OS and in case of UNIX based system, you don't have to install VMware, and you can directly use the Linux environment to install the Spark framework. We have divided this chapter mainly in three parts as below and each section will have various steps to be followed.

1. VMWare workstation installation
2. Installing Ubuntu Linux on VMWare
3. Setting Spark env on Ubuntu

Please follow the steps as below in sequence as provided.

Part-1: VMWare workstation installation

There are two version of VMWare workstations one is free version and another is Pro version. Free version support only single OS image at a time while pro version support more than one image of Operating System. However, we are setting environment with the single node. So free version will suffice for our requirement.

Step 1: Download VMWare Workstation

VMWare Read FAQ: Before starting, we would recommend you go through the below FAQs of VMWare if you are not aware about VMWare software.

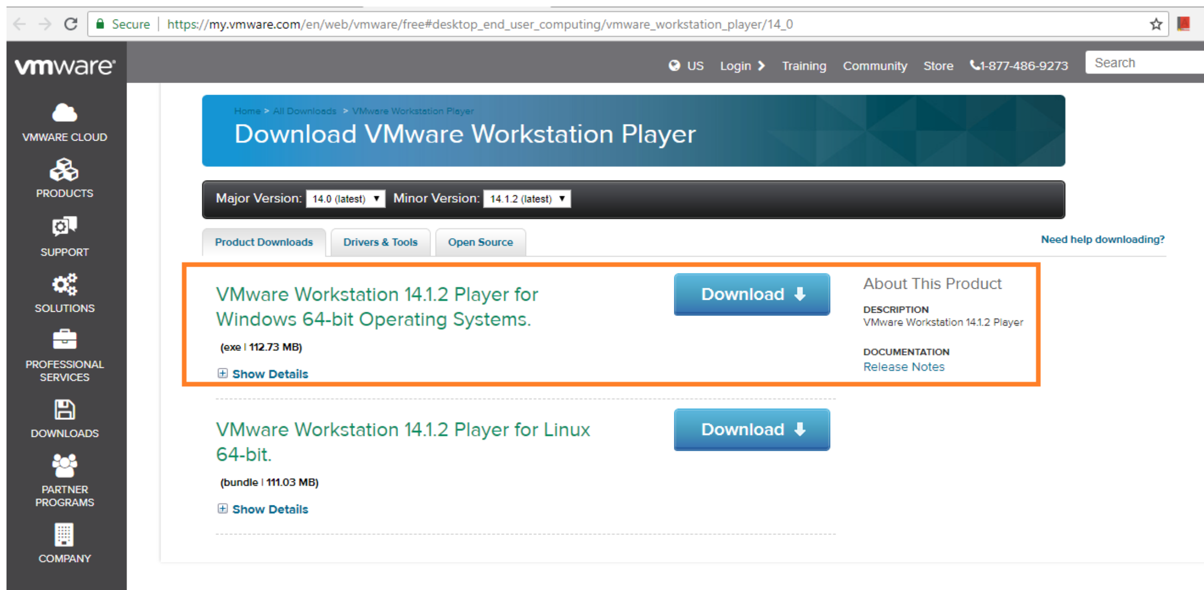
https://www.vmware.com/products/player/faqs.html

Download link: Please keep checking above FAQ for downloading latest version. Link may change.

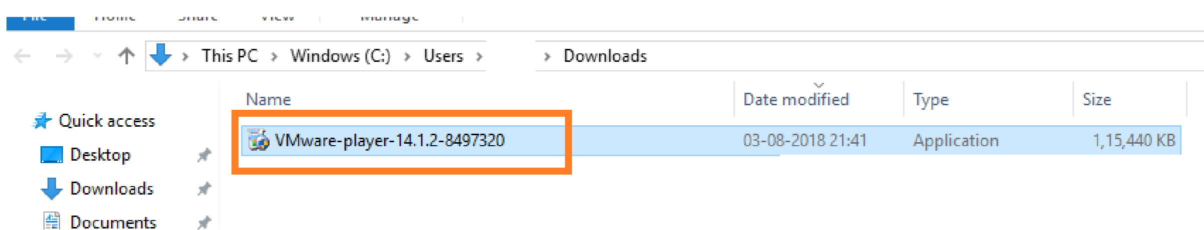
https://my.vmware.com/en/web/vmware/free#desktop_end_user_computing/vmware_workstation_player/14_0

Compare free vs Paid Version: Below links provide the detail comparison between free and pro version of VMware.

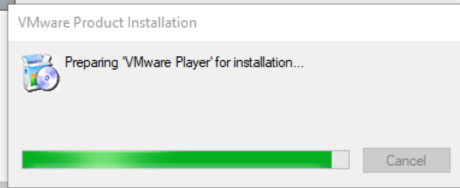
https://www.vmware.com/products/workstation-pro.html



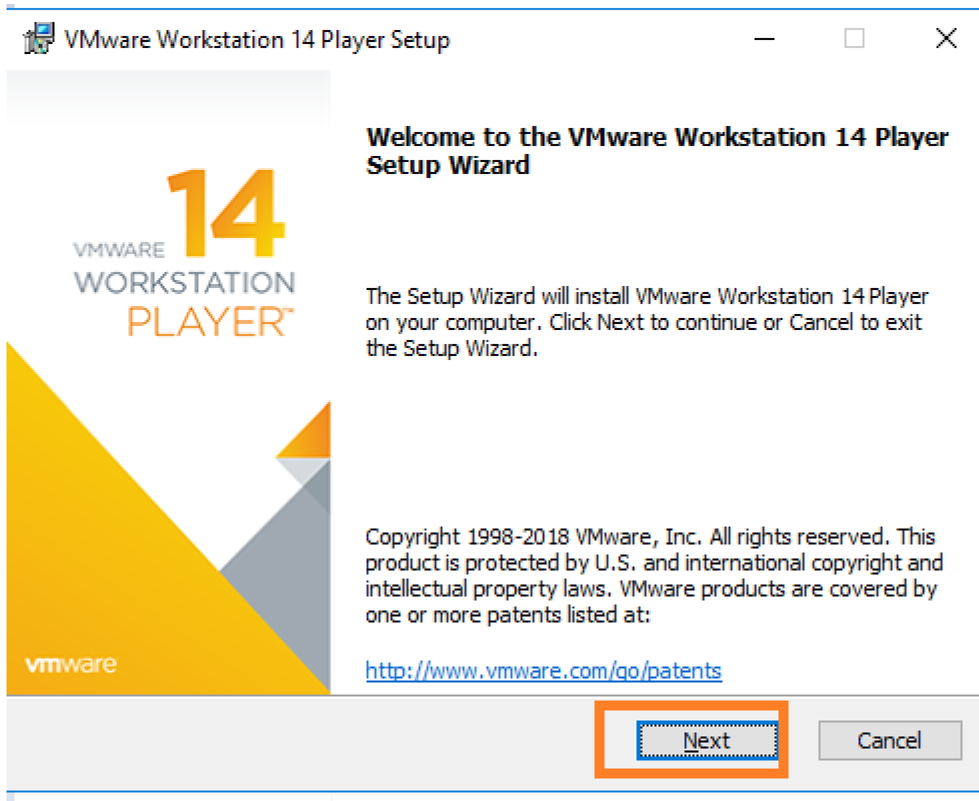
Step 2: Downloaded application will look like this, it is a Windows application.



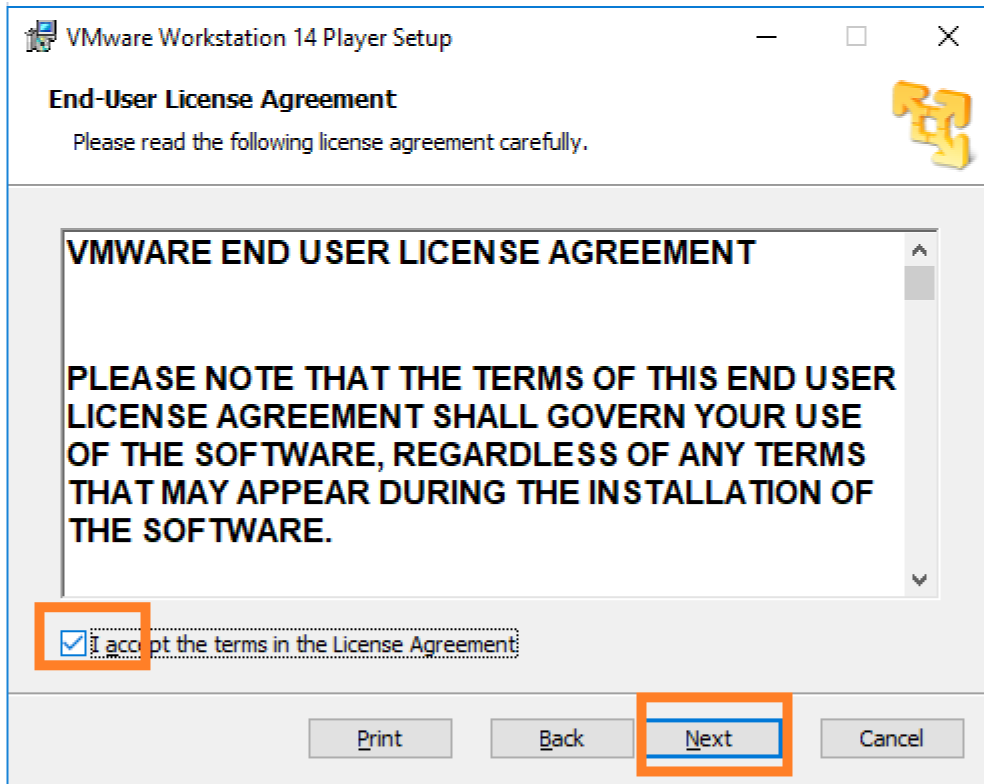
Step 3: Double click on it to install it. And you should see below Splash until it get installed.



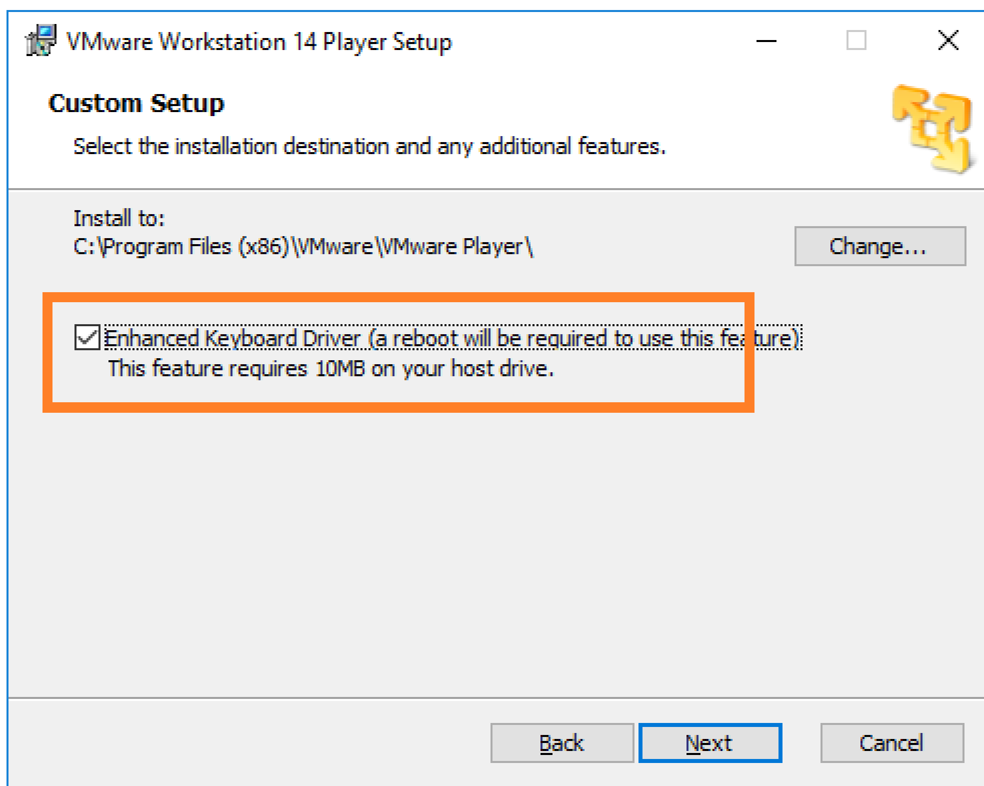
Step 4: Click next



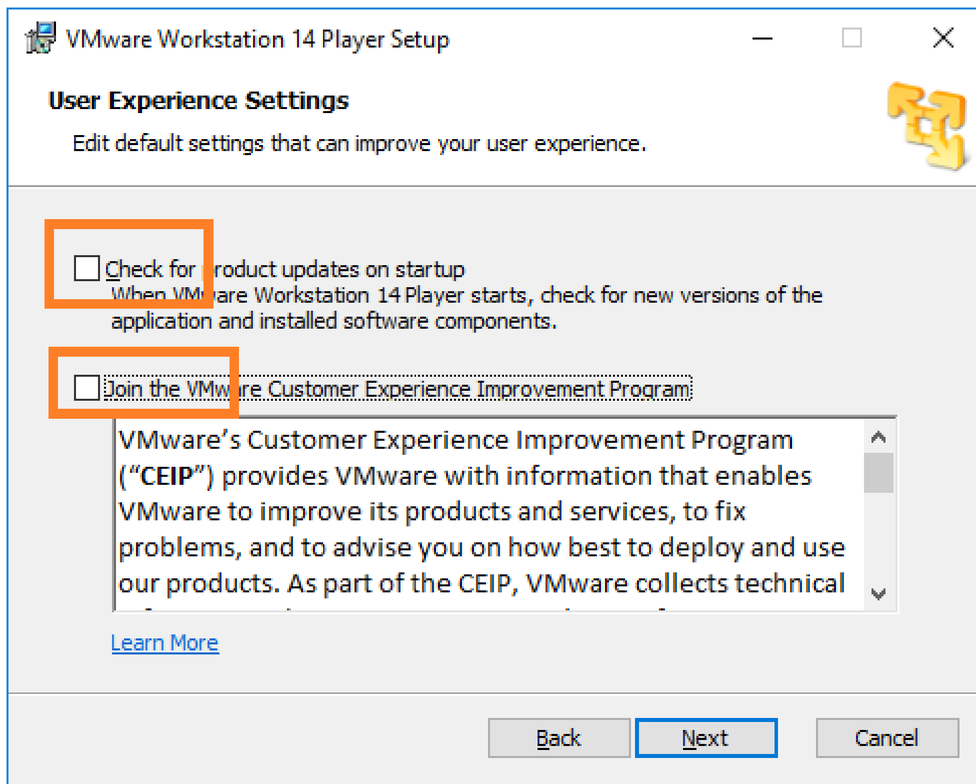
Step 5: Accept terms and click next



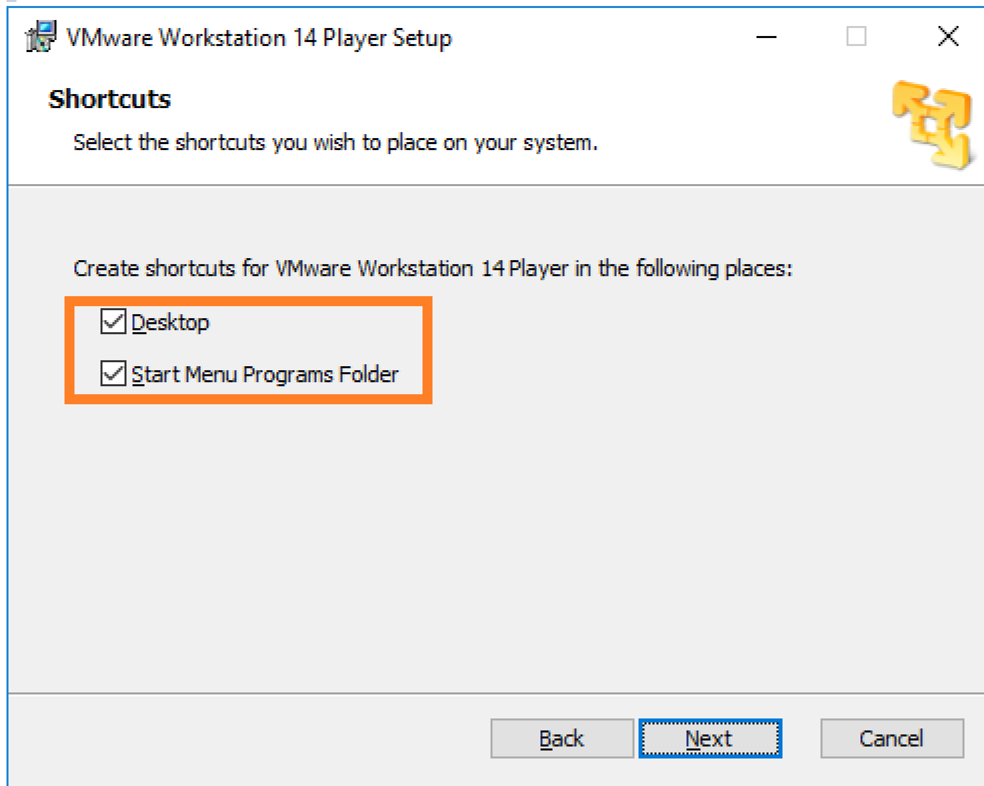
Step 6: Select Enhance Keyboard features.



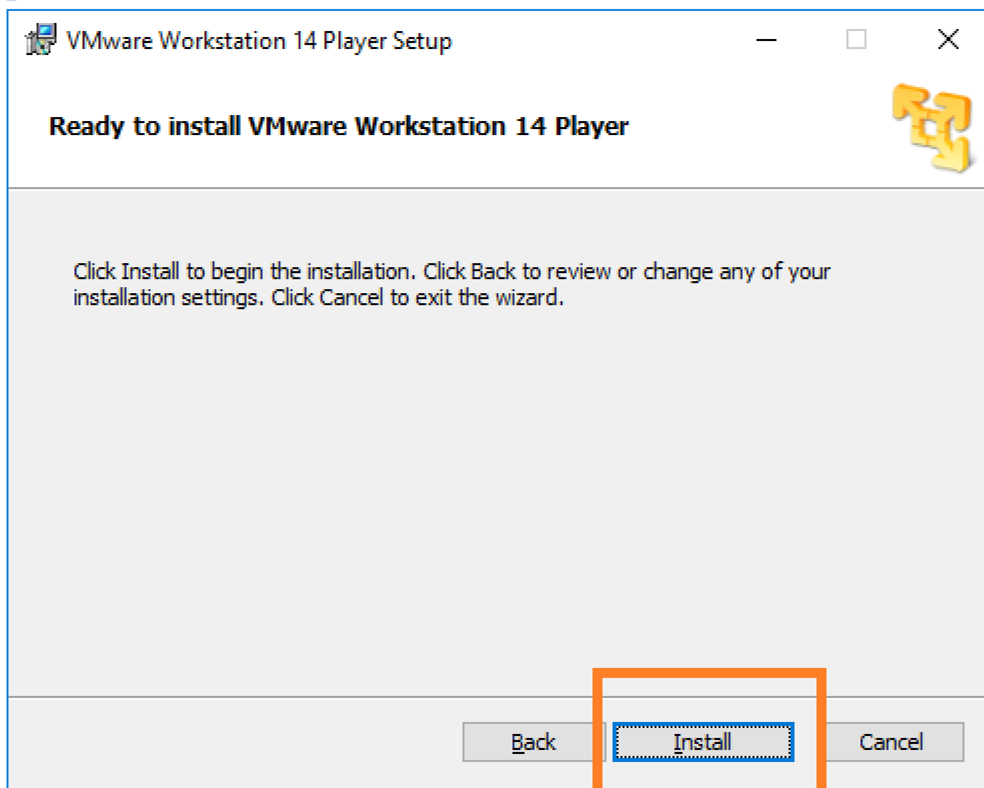
Step 7: Uncheck for default updates and click next.



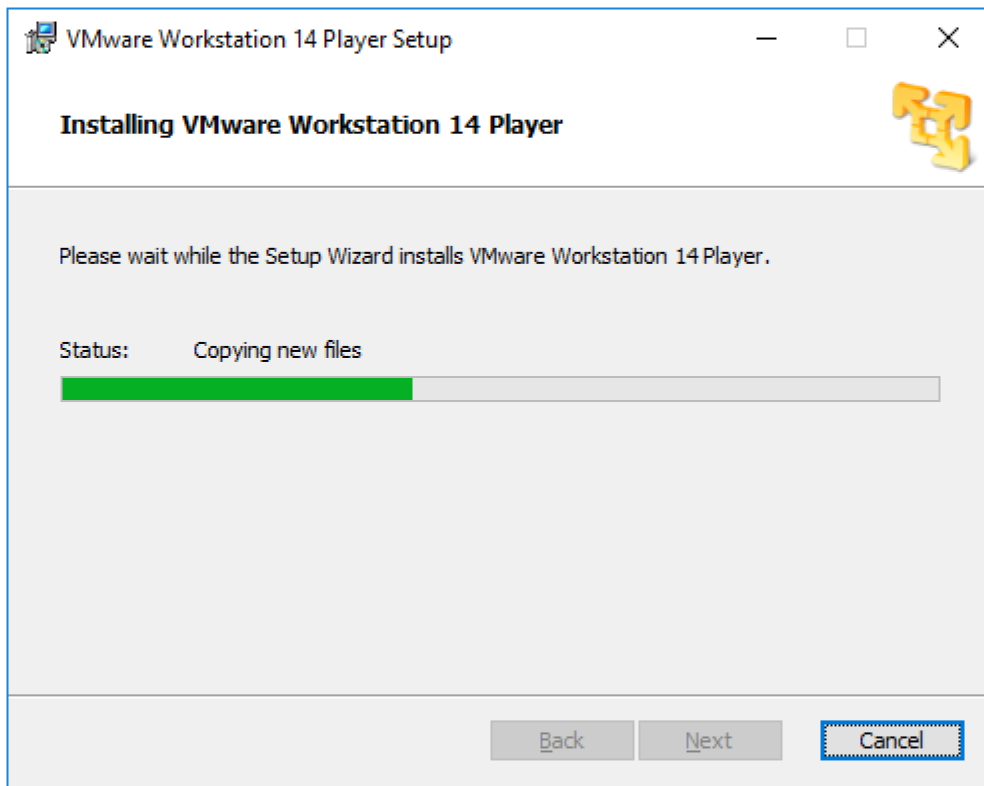
Step 8: Allow to create desktop icon and click next.



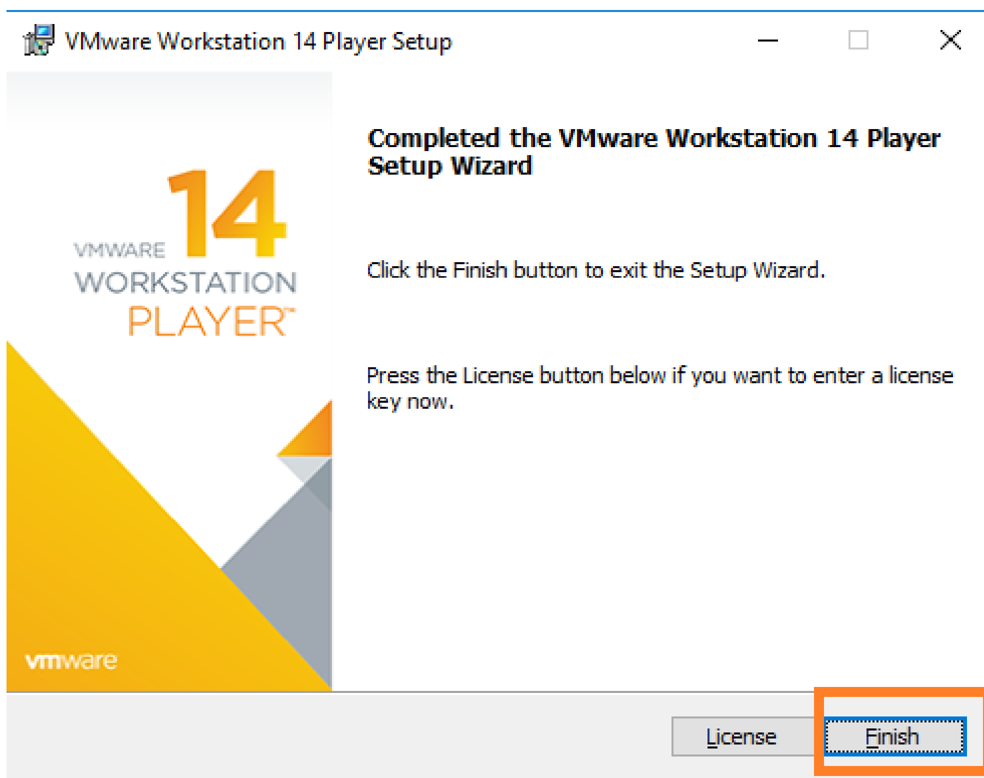
Step 9: Now click install.



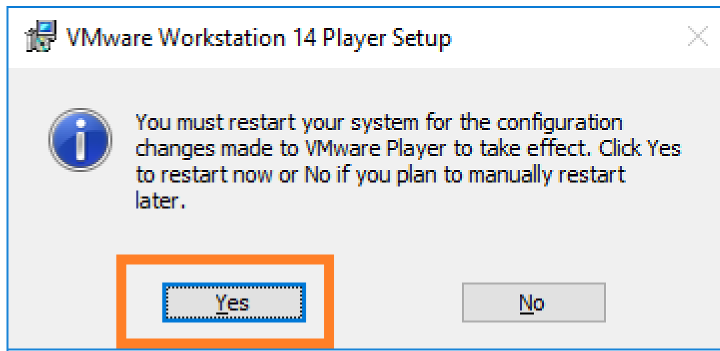
Step 10: You will see progress bar as below



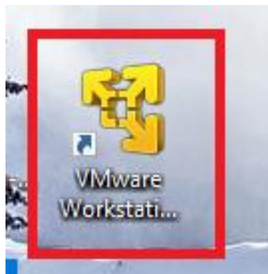
Step 11: Click finish once done



Step 12: It will ask you to restart the machine. Yes, do that

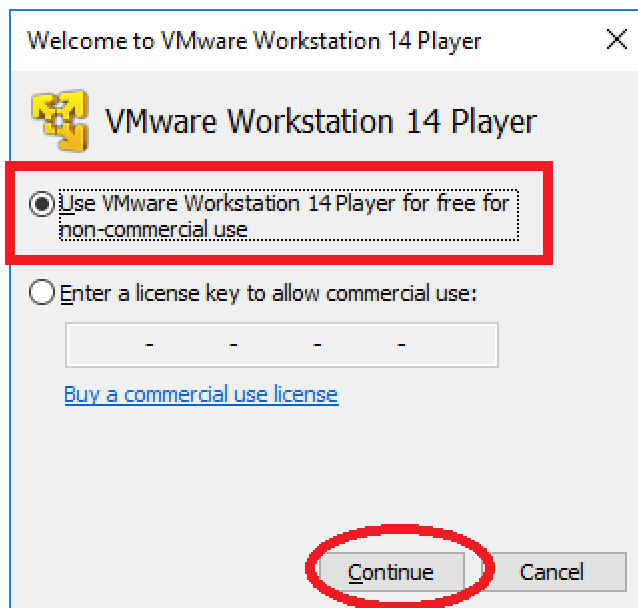


Step 13: Once you re-start your machine you will see following icon on your desktop.

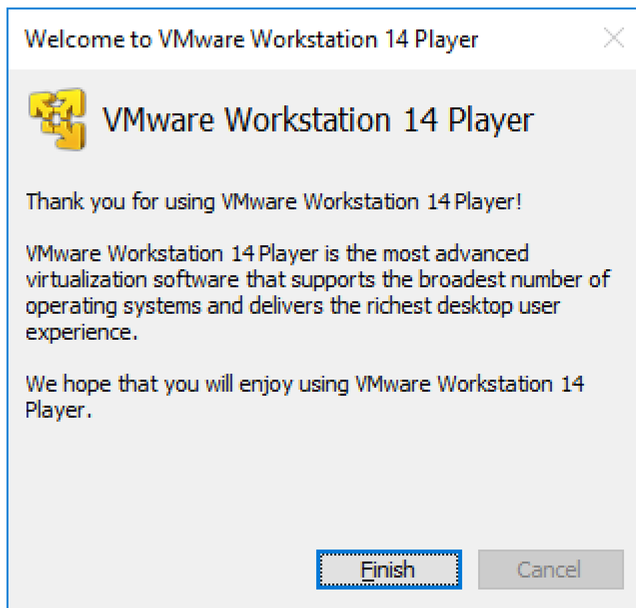


Step 14: Start VMWare workstation player.

Step 15: We will be using free version, if you want you can purchase it and use license keys to activate the same. Now click continue.



Click finish



Step 16: You must see the below screen, once it is started.

Step 17: In next module we will install Ubuntu Linux.

Part-2 Installing Ubuntu Linux on VMWare

Now we do have free version of VMWare workstation already setup on Windows OS, now download the Ubuntu Linux Image and create a VM Instance by following the steps below. Also we are providing the steps for enabling the virtualization of Windows machine, if it is disabled. And how to connect this VM image using Putty.

Step 1: Download Ubuntu from below link (It may change time to time)

http://releases.ubuntu.com/14.04/

Step 2: We are using below image (It may change time to time. So, select accordingly). It is 1 GB ISO image download.

ubuntu-14.04.5-desktop-amd64.iso

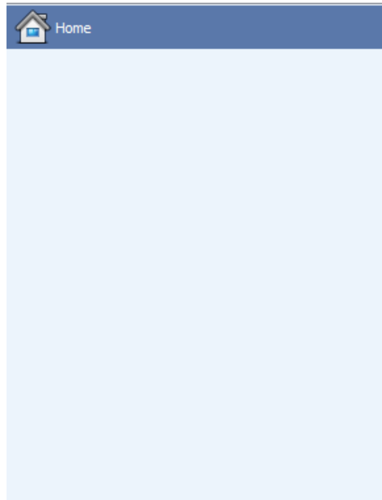
Name	Last modified	Size	Description
Parent Directory		-	
MD5SUMS	2017-02-16 22:30	307	
MD5SUMS-metalink	2016-08-04 20:46	568	
MD5SUMS-metalink.gpg	2016-08-04 20:46	933	
MD5SUMS.gpg	2017-02-16 22:30	933	
SHA1SUMS	2017-02-16 22:30	347	
SHA1SUMS.gpg	2017-02-16 22:30	933	
SHA256SUMS	2017-02-16 22:30	467	
SHA256SUMS.gpg	2017-02-16 22:30	933	
ubuntu-14.04.5-desktop-amd64.iso	2016-08-03 17:49	1.0G	Desktop image for 64-bit PC (AMD64) computers (standard download)
ubuntu-14.04.5-desktop-amd64.iso.torrent	2016-08-04 20:43	41K	Desktop image for 64-bit PC (AMD64) computers (BitTorrent download)
ubuntu-14.04.5-desktop-amd64.iso.zsync	2016-08-04 20:43	2.1M	Desktop image for 64-bit PC (AMD64) computers (zsync metafile)
ubuntu-14.04.5-desktop-amd64.list	2016-08-03 17:49	4.5K	Desktop image for 64-bit PC (AMD64) computers (file listing)
ubuntu-14.04.5-desktop-amd64.manifest	2016-08-03 17:41	60K	Desktop image for 64-bit PC (AMD64) computers (contents of live filesystem)
ubuntu-14.04.5-desktop-amd64.metalink	2016-08-04 20:46	44K	Ubuntu 14.04.5 LTS (Trusty Tahr)
ubuntu-14.04.5-desktop-i386.iso	2016-08-03 17:52	1.0G	Desktop image for 32-bit PC (i386) computers (standard download)
ubuntu-14.04.5-desktop-i386.iso.torrent	2016-08-04 20:43	42K	Desktop image for 32-bit PC (i386) computers (BitTorrent download)
ubuntu-14.04.5-desktop-i386.iso.zsync	2016-08-04 20:43	2.1M	Desktop image for 32-bit PC (i386) computers (zsync metafile)
ubuntu-14.04.5-desktop-i386.list	2016-08-03 17:52	3.8K	Desktop image for 32-bit PC (i386) computers (file listing)
ubuntu-14.04.5-desktop-i386.manifest	2016-08-03 15:57	59K	Desktop image for 32-bit PC (i386) computers (contents of live filesystem)

Step 3: Now import the Linux OS image of Ubuntu in VMWare Workstation Player.

Click on the “Create new virtual machine”

VMware Workstation 14 Player (Non-commercial use only)

Player ▾



Welcome to VMware Workstation 14 Player

Create a New Virtual Machine

Create a new virtual machine, which will then be added to the top of your library.

Open a Virtual Machine

Open an existing virtual machine, which will then be added to the top of your library.

Upgrade to VMware Workstation Pro

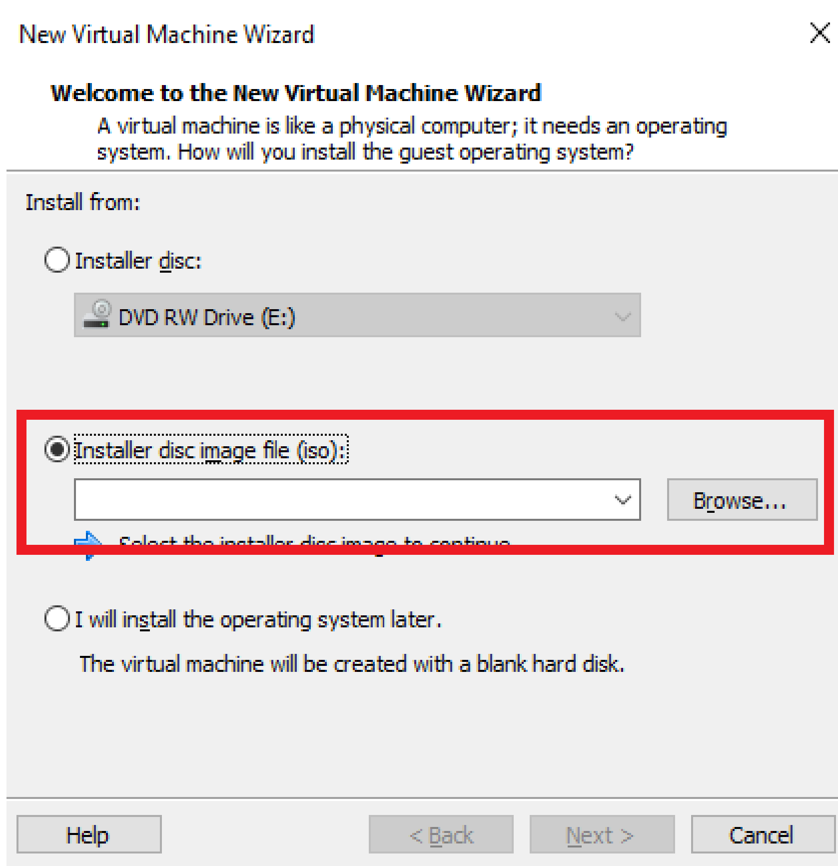
Get advanced features such as snapshots, virtual network management, and more.

Help

View online help.

Step 4: Select ISO image option as well as browse the Ubuntu ISO image.

Once done click next



New Virtual Machine Wizard

Welcome to the New Virtual Machine Wizard

A virtual machine is like a physical computer; it needs an operating system. How will you install the guest operating system?

Install from:

○ Installer disc:

DVD RW Drive (E:)

◉ Installer disc image file (iso):

C:\TempWork\ubuntu-14.04.5-desktop-amd64.iso | Browse...

Ubuntu 64-bit 14.04.5 detected.
This operating system will use Easy Install. (What's this?)

○ I will install the operating system later.

The virtual machine will be created with a blank hard disk.

Help | < Back | Next > | Cancel

Step 5: Please provide all the fields with the same value as below (hadoopexam) and click next.

New Virtual Machine Wizard

Easy Install Information

This is used to install Ubuntu 64-bit.

Personalize Linux

Full name: hadoopexam

User name: hadoopexam

Password: ••••••••••

Confirm: ••••••••••

Help | < Back | Next > | Cancel

Step 6: Give the name to Virtual machine as below (HadoopExam_ubuntu) and click next

New Virtual Machine Wizard

Name the Virtual Machine

What name would you like to use for this virtual machine?

Virtual machine name:

HadoopExam_ubuntu

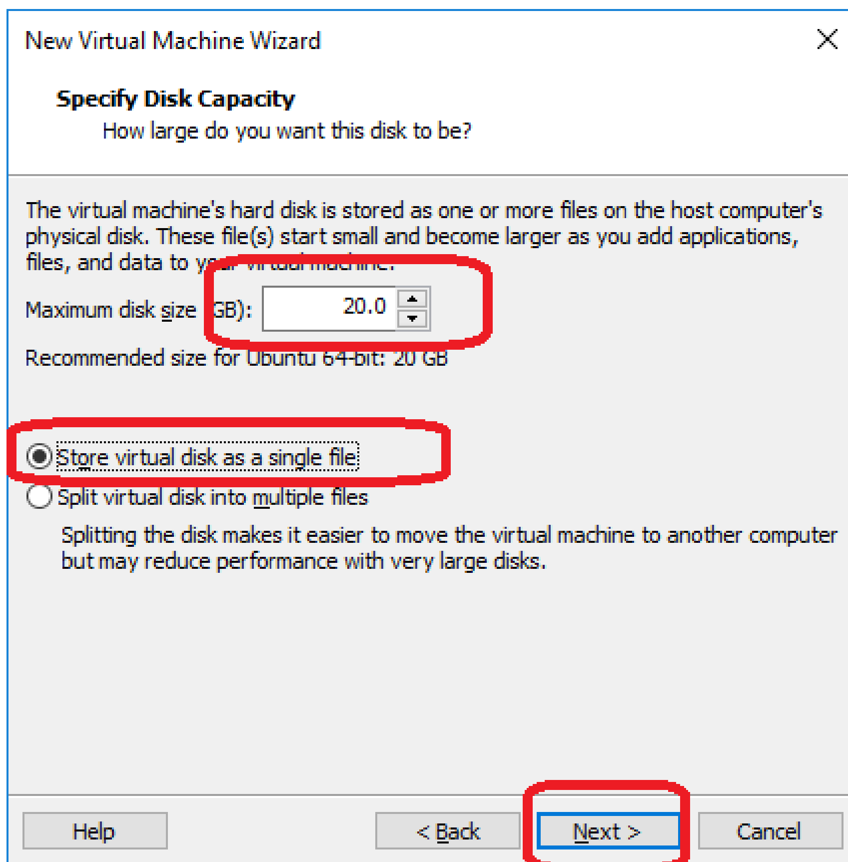
Location:

C:\Users\rajsh\Documents\Virtual Machines\HadoopExam_ubun

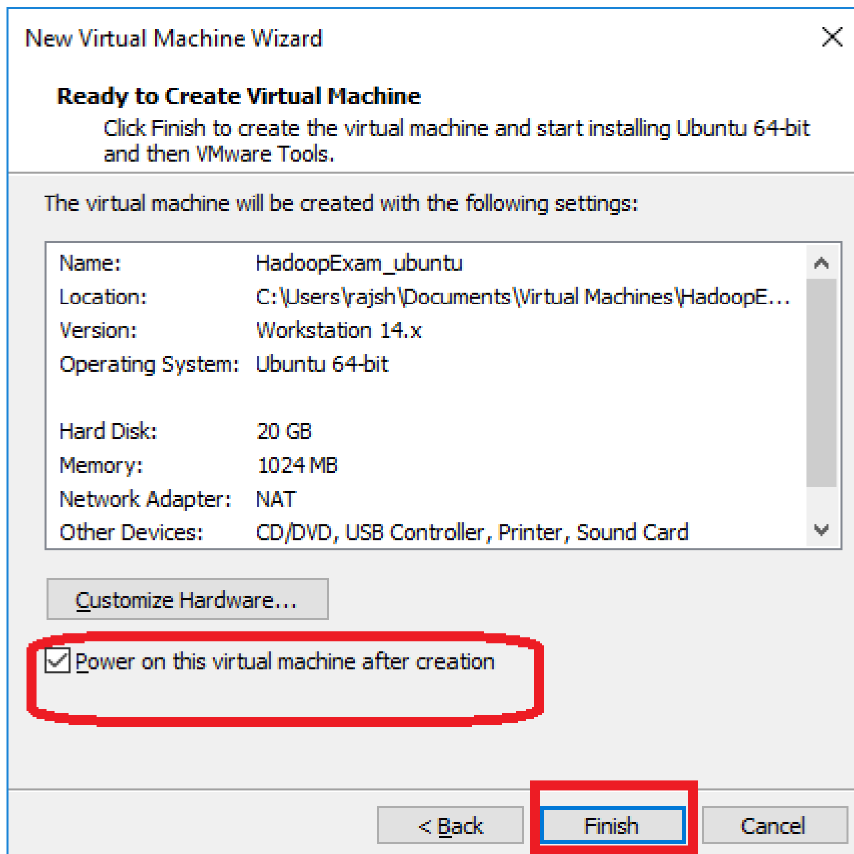
Browse...

< Back | Next > | Cancel

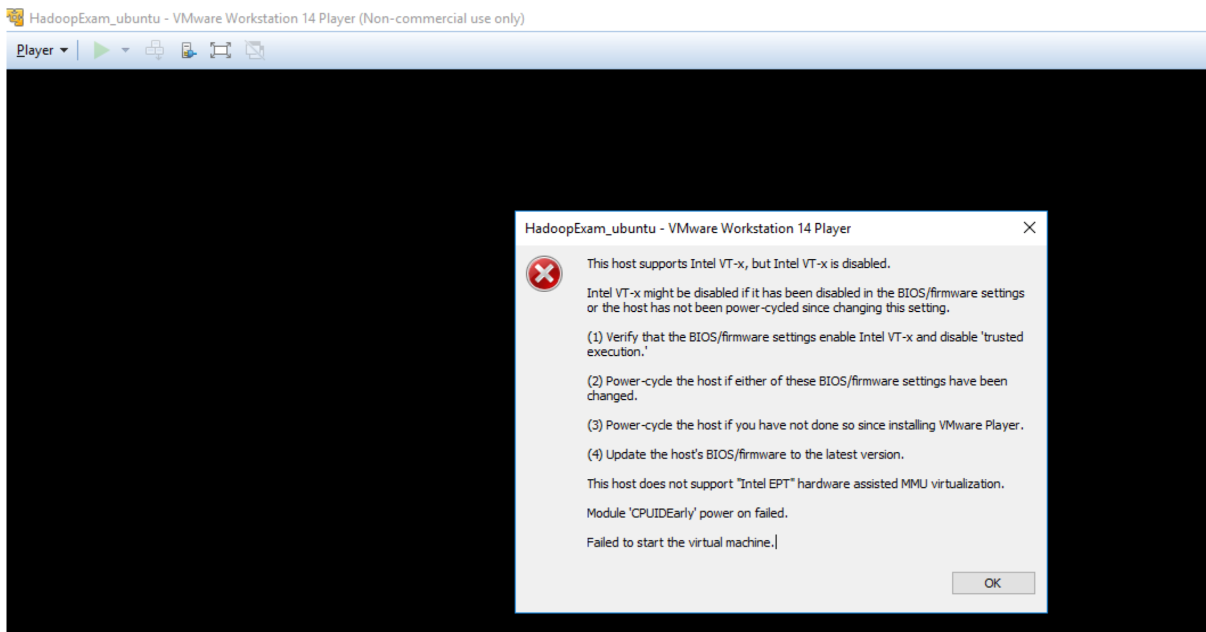
Step 7: Allocate the space 20GB is good enough. As well as create single file for Virtual Disk.



Step 8: Now power on the Virtual Machine.



Step 9 (Optional): If you see below screen, it means virtualization is disabled for your machine. Please try below steps for enabling it.

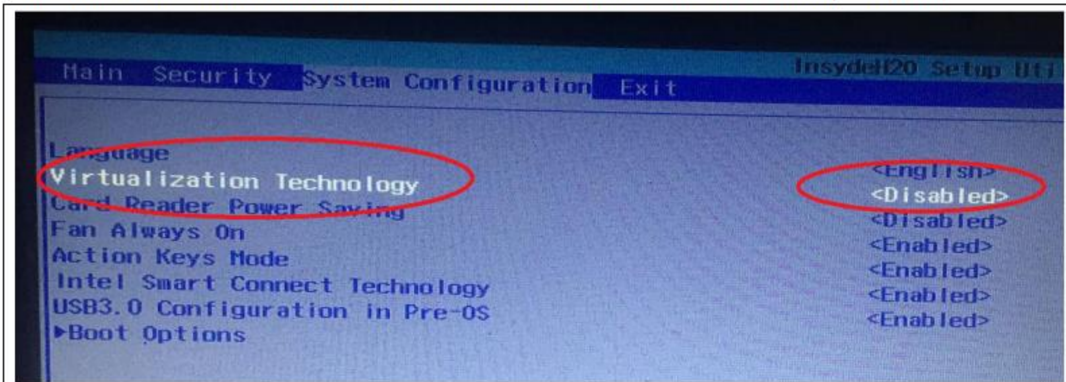


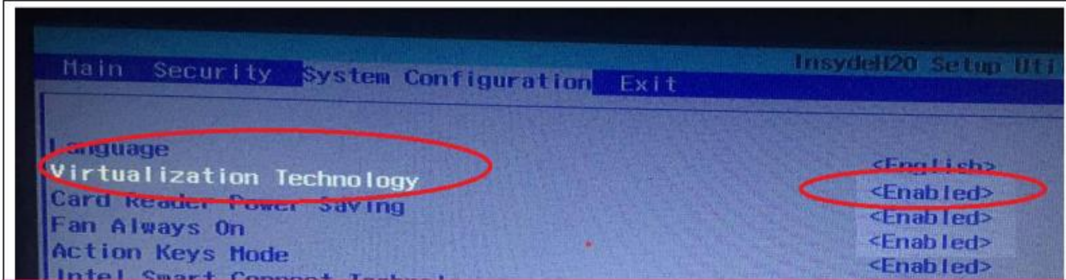
QuickTechie

If Virtualization is not enabled you will see following Pop-up, with error message below.

- This virtual machine is configured for 64-bit guest operating systems. However, 64-bit operation is not possible.
- This host supports Intel VT-x, but Intel VT-x is disabled.
- Intel VT-x might be disabled if it has been disabled in the BIOS/firmware settings or the host has not been power-cycled since changing this setting.
- (1) Verify that the BIOS/firmware settings enable Intel VT-x and disable 'trusted execution.'
- (2) Power-cycle the host if either of these BIOS/firmware settings have been

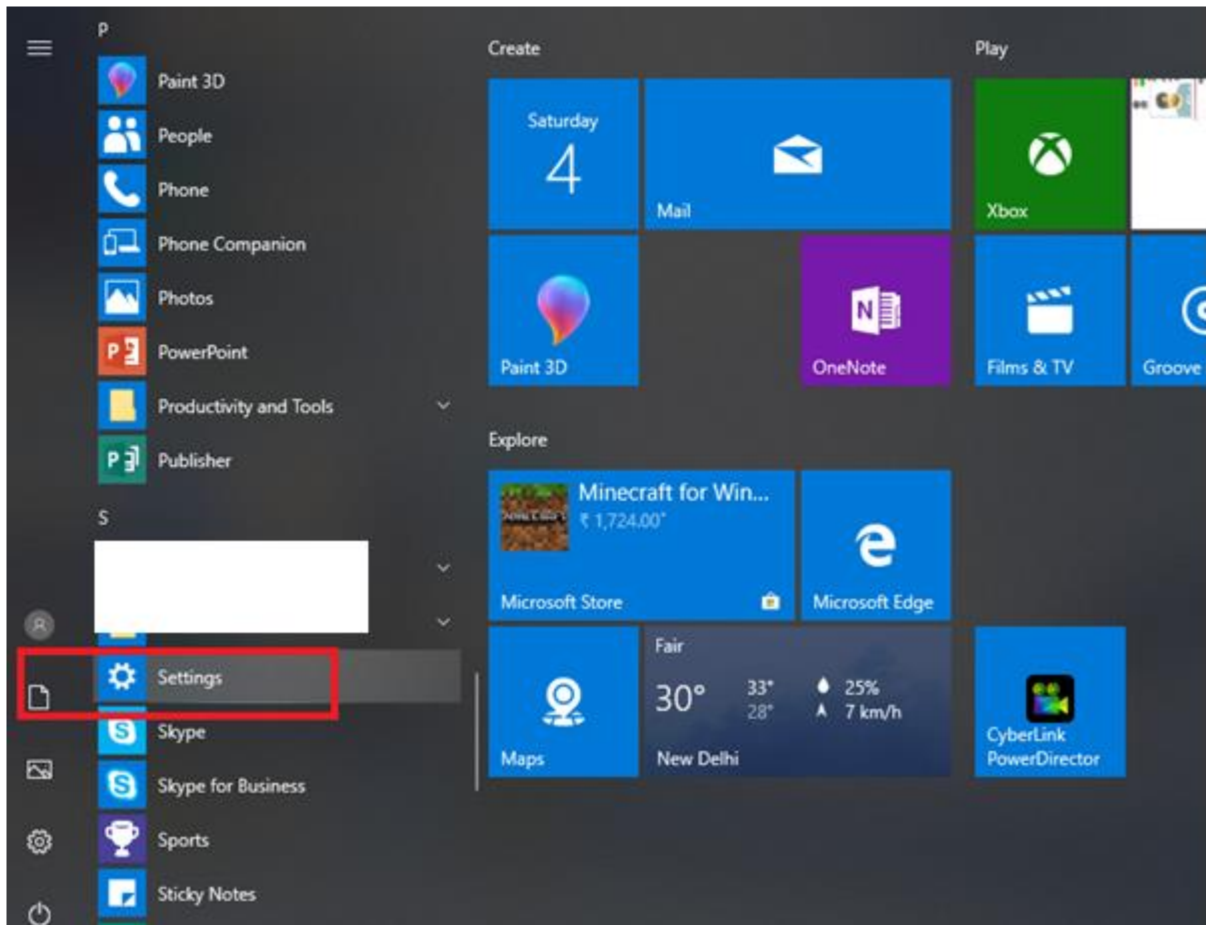
. To solve above problem must be simple. Go to your windows machine Bios setting and enable the virtualization as shown in below. (Your machine could have little different way to do this)





There are multiple ways to access BIOS setting. It varies from machine to machine.

Step 9A: There is one way where Windows 10 will allow you to access BIOS. Follow these steps



Step 9B: Select Update & Security option

Settings

Windows Settings

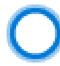
Find a setting

Accounts
Your accounts, email, sync, work, family

Time & Language
Speech, region, date

Gaming
Game bar, DVR, broadcasting, Game Mode

Ease of Access
Narrator, magnifier, high contrast

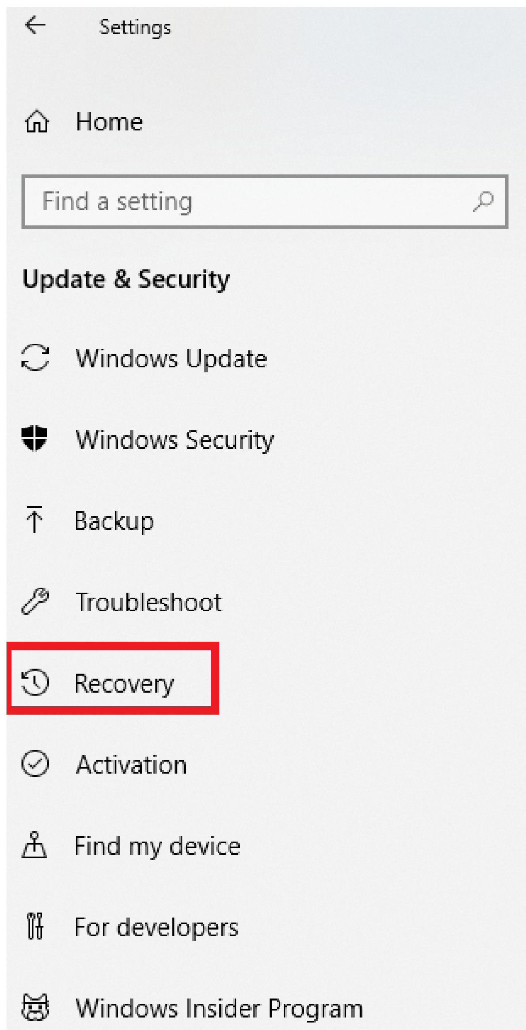


Cortana
Cortana language, permissions, notifications

Privacy
Location, camera

Update & Security
Windows Update, recovery, backup

Step 9C: Select "Recovery" option from left menu and click "Restart Now"



Recovery

Reset this PC

If your PC isn't running well, resetting it might help. This
choose to keep your personal files or remove them, and
reinstalls Windows.

Get started

Advanced startup

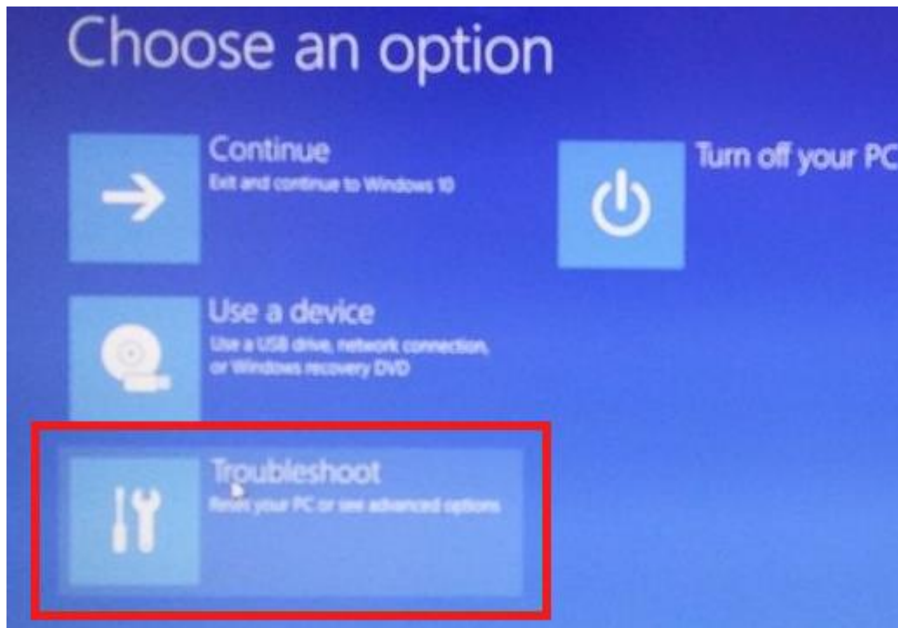
Start up from a device or disc (such as a USB drive or DV
change your PC's firmware settings, change Windows sta
settings, or restore Windows from a system image. This v
restart your PC.

Restart now

More recovery options

Learn how to start fresh with a clean installation of Wind

Step 9D : Then you will be given below screen where you need to select Troubleshoot option.

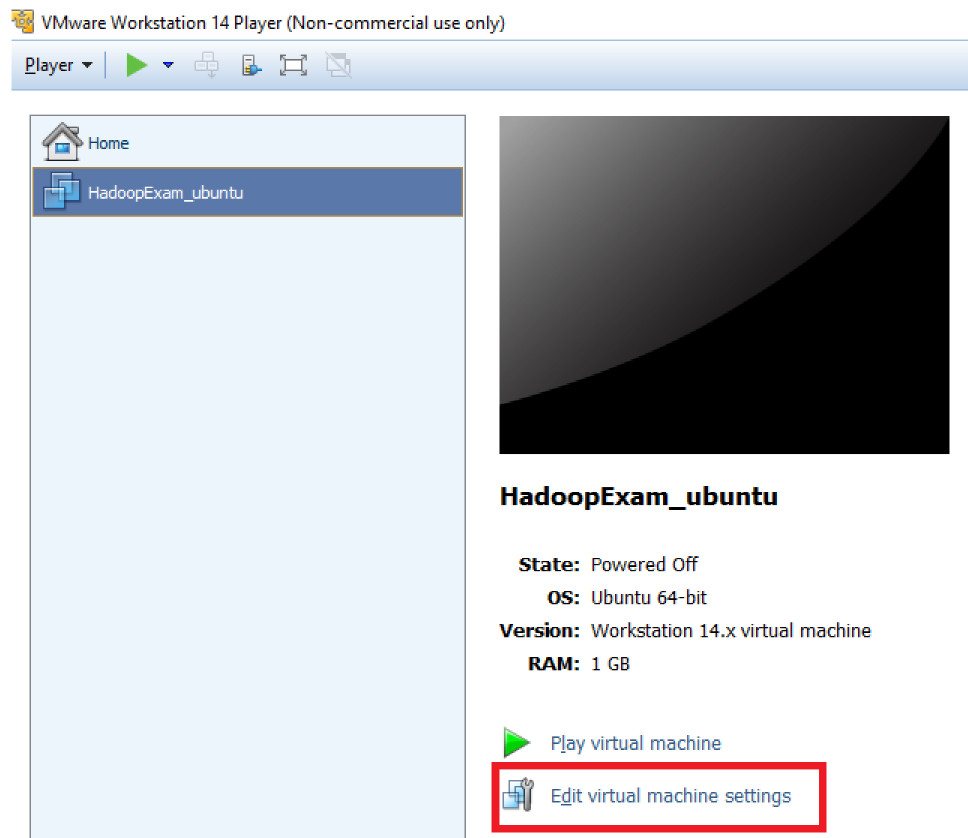


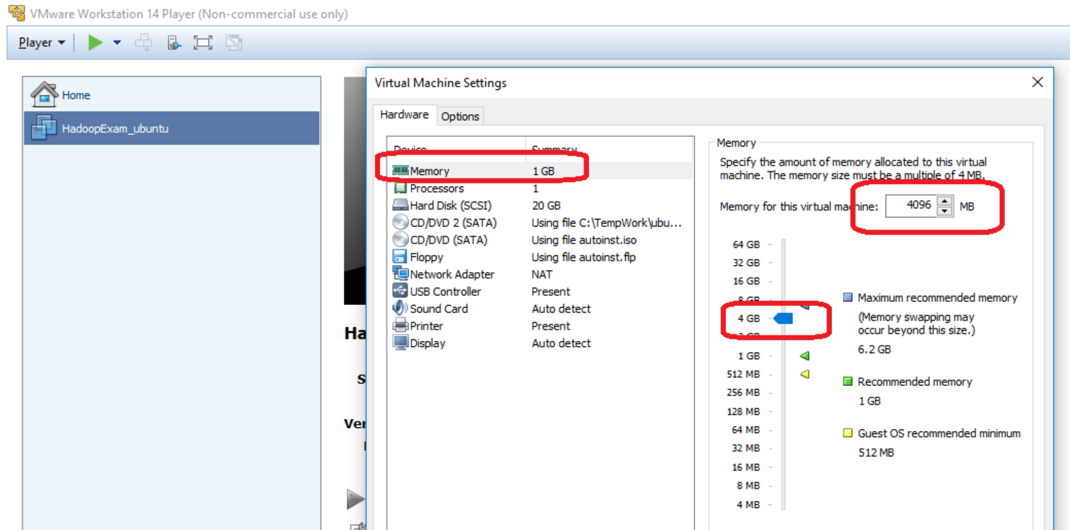
Step 9E : InTroubleshoot select “Advanced Options”

Select UEFI settings (This will help you to insert BIOS mode without quickly pressing F10 keys)

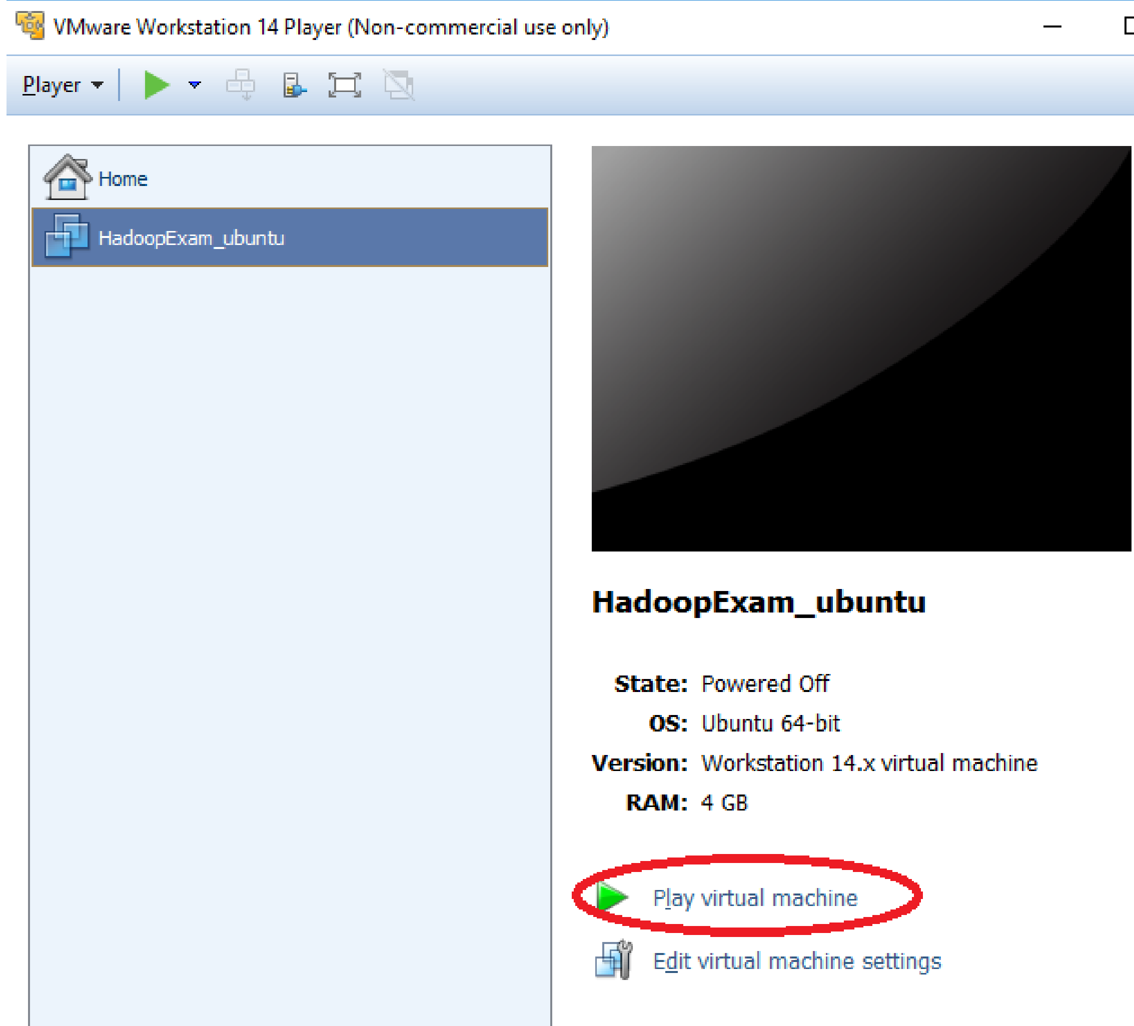
While restarting the machine you can select the options what you wanted to do.

Step 10: Increase the RAM Size to 4GB by clicking “Edit Virtual Machine Setting” as below. You can do it any time before power on the virtual machine.



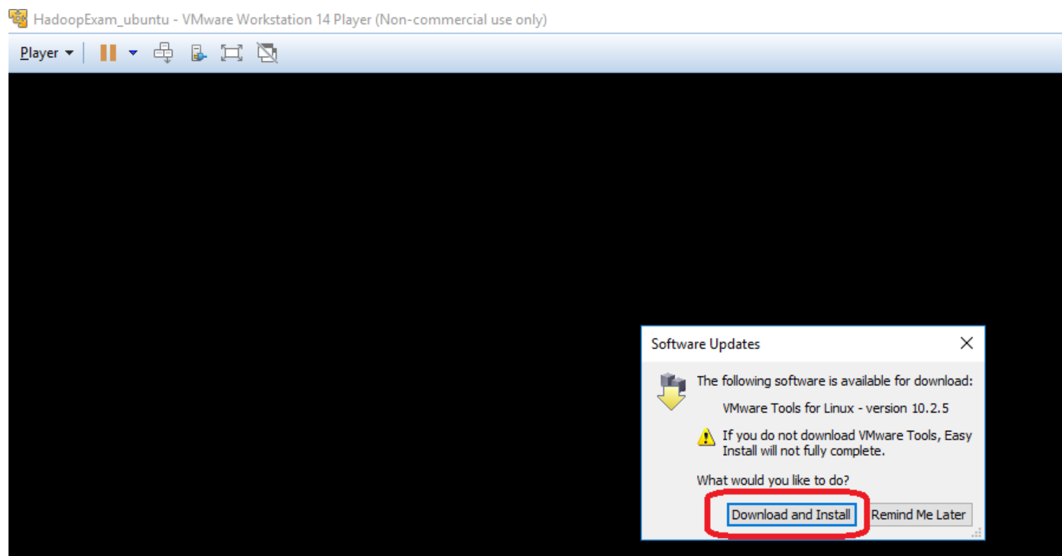


Step 11: Now we will start the Ubuntu Operating system.

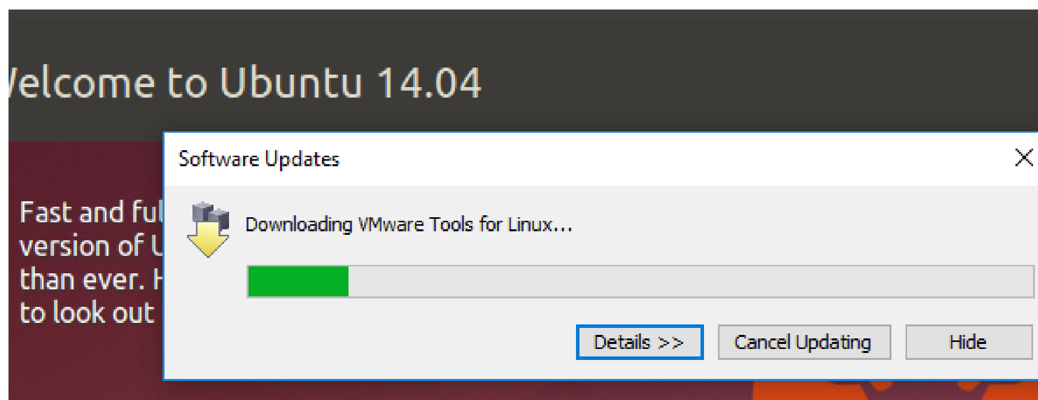


Step 12: Wait for couple of Minutes to start Virtual Machine. Meanwhile you see below screen.

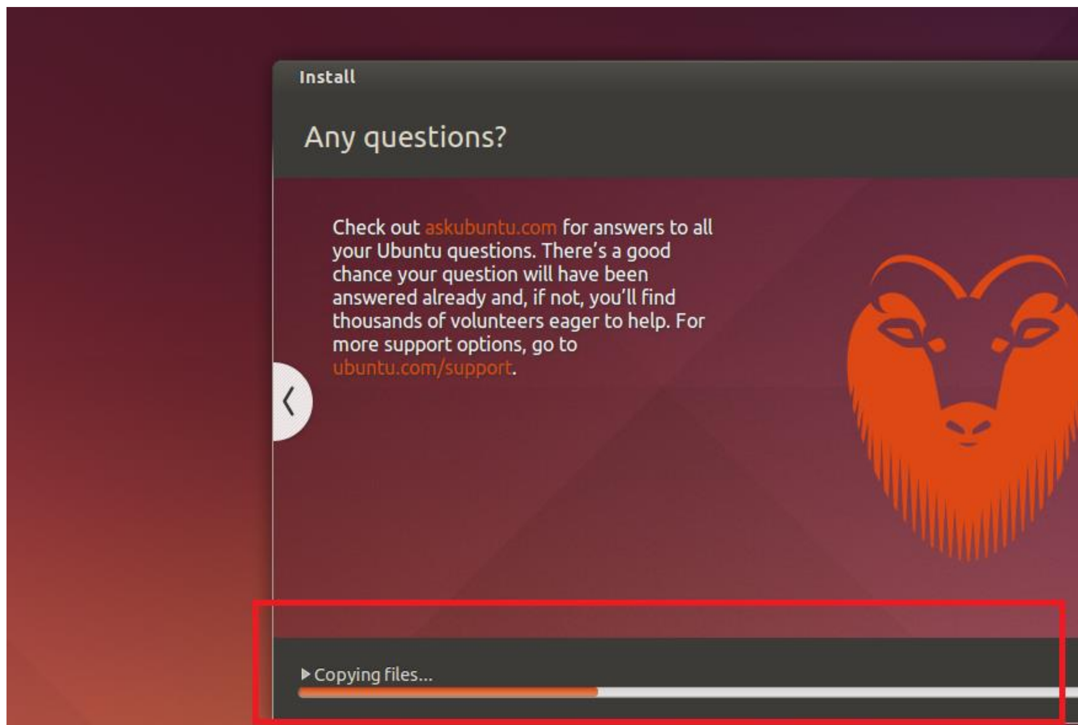
If you see below warning then do the Download and Install. It will be good.



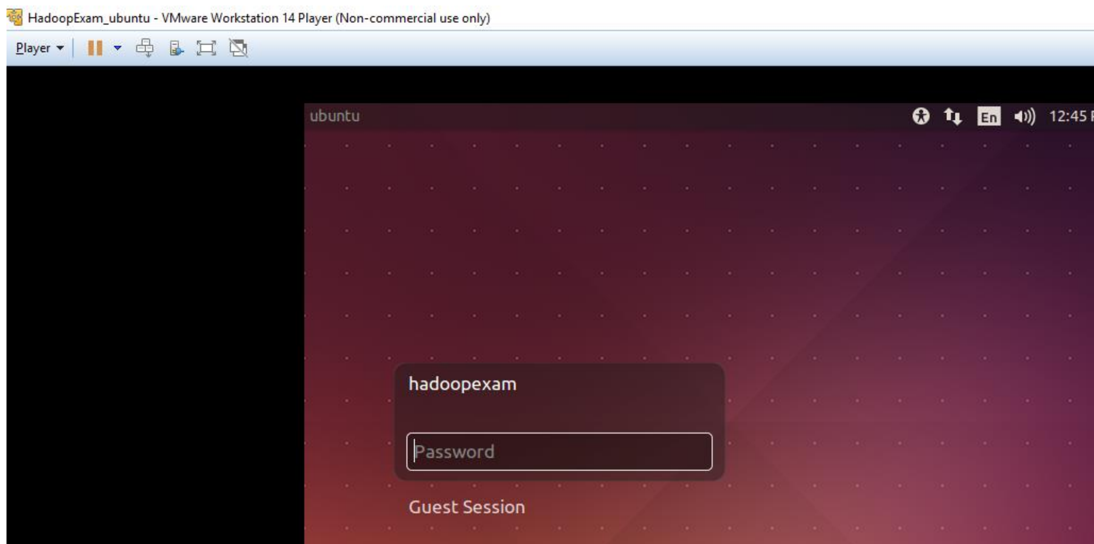
And you will see below progress bar



Ubuntu is getting started, wait for few minutes (First time it may take longer).



Once it is started, you will see below screen.



Step 13: Now login to Ubuntu system.

Username: hadoopexam

Password: hadoopexam

Step 14: Install ssh server, so that we can connect this machine from outside as well e.g. putty.

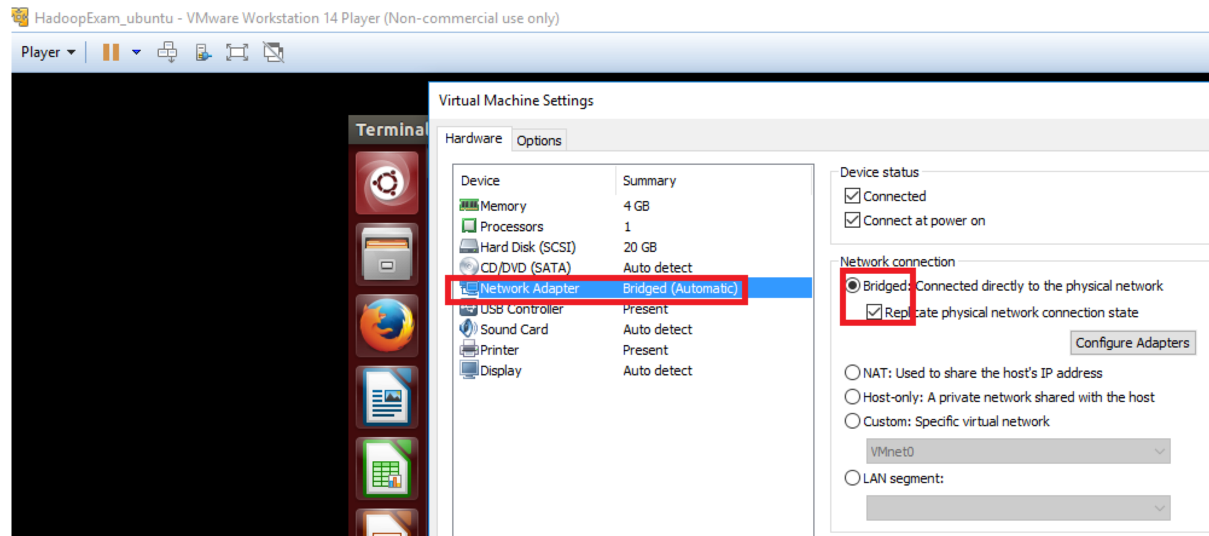
```
sudo apt-get install openssh-server
```

Step 15: Install VIM

```
sudo apt-get install vim

vim ~/.vimrc
set nocompatible
```

Step 16: Change the Network settings in the VM as below.



Connecting through Putty

Step 17: Download Putty software

```
https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html
```

Step 18: Select and download putty zip file

Secure | https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

pscp.exe (an SCP client, i.e. command-line secure file copy)

32-bit:	pscp.exe	(or by FTP)	(signature)
64-bit:	pscp.exe	(or by FTP)	(signature)

psftp.exe (an SFTP client, i.e. general file transfer sessions much like FTP)

32-bit:	psftp.exe	(or by FTP)	(signature)
64-bit:	psftp.exe	(or by FTP)	(signature)

puttytel.exe (a Telnet-only client)

32-bit:	puttytel.exe	(or by FTP)	(signature)
64-bit:	puttytel.exe	(or by FTP)	(signature)

plink.exe (a command-line interface to the PuTTY back ends)

32-bit:	plink.exe	(or by FTP)	(signature)
64-bit:	plink.exe	(or by FTP)	(signature)

pageant.exe (an SSH authentication agent for PuTTY, PSCP, PSFTP, and Plink)

32-bit:	pageant.exe	(or by FTP)	(signature)
64-bit:	pageant.exe	(or by FTP)	(signature)

puttygen.exe (a RSA and DSA key generation utility)

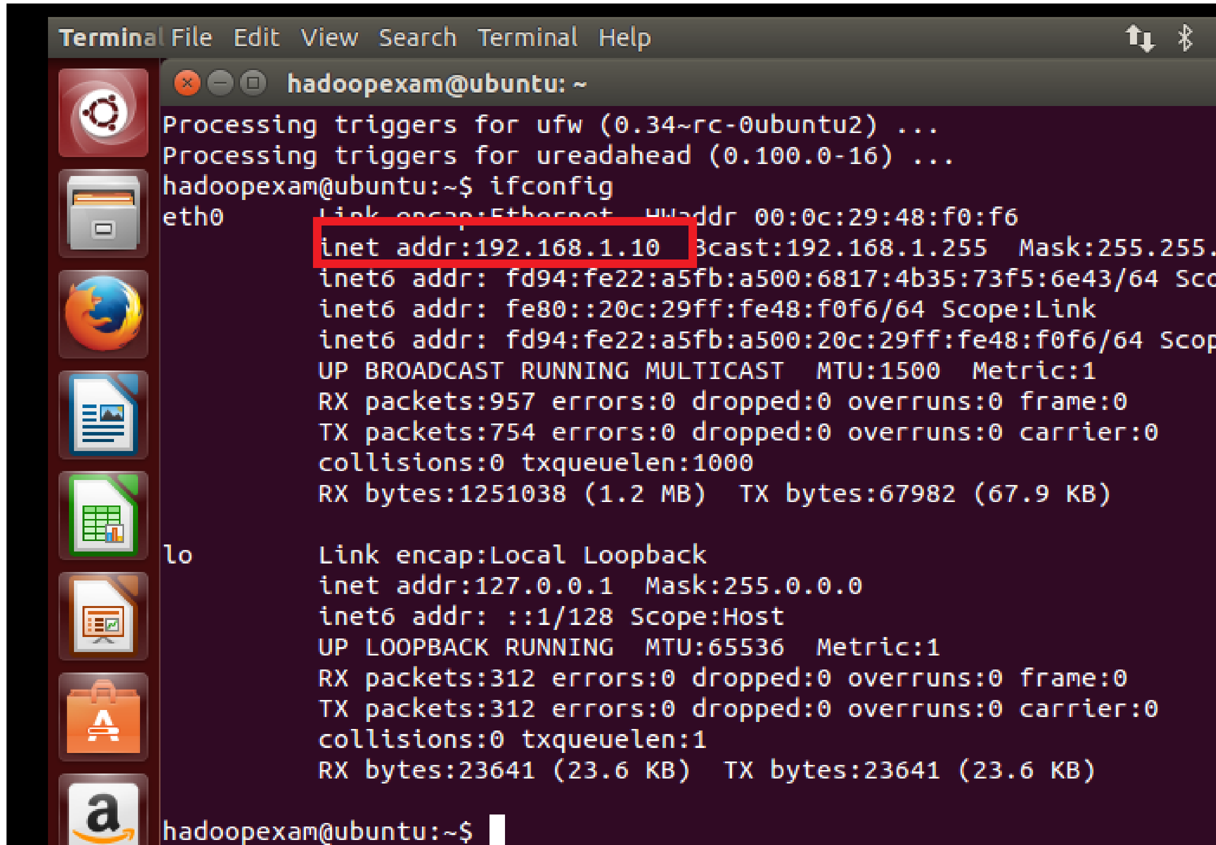
32-bit:	puttygen.exe	(or by FTP)	(signature)
64-bit:	puttygen.exe	(or by FTP)	(signature)

putty.zip (a .ZIP archive of all the above)

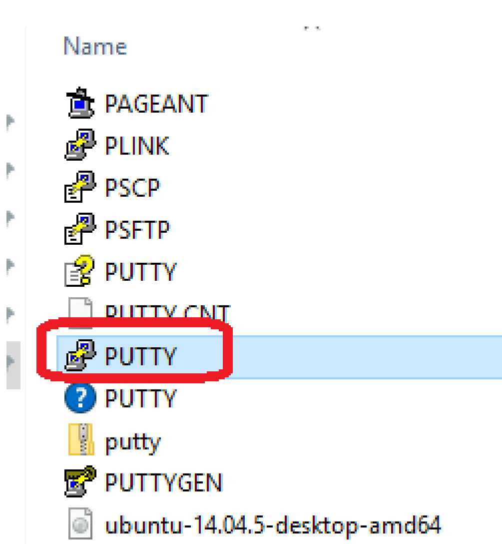
32-bit:	putty.zip	(or by FTP)	(signature)
64-bit:	putty.zip	(or by FTP)	(signature)

Once download unzip it.

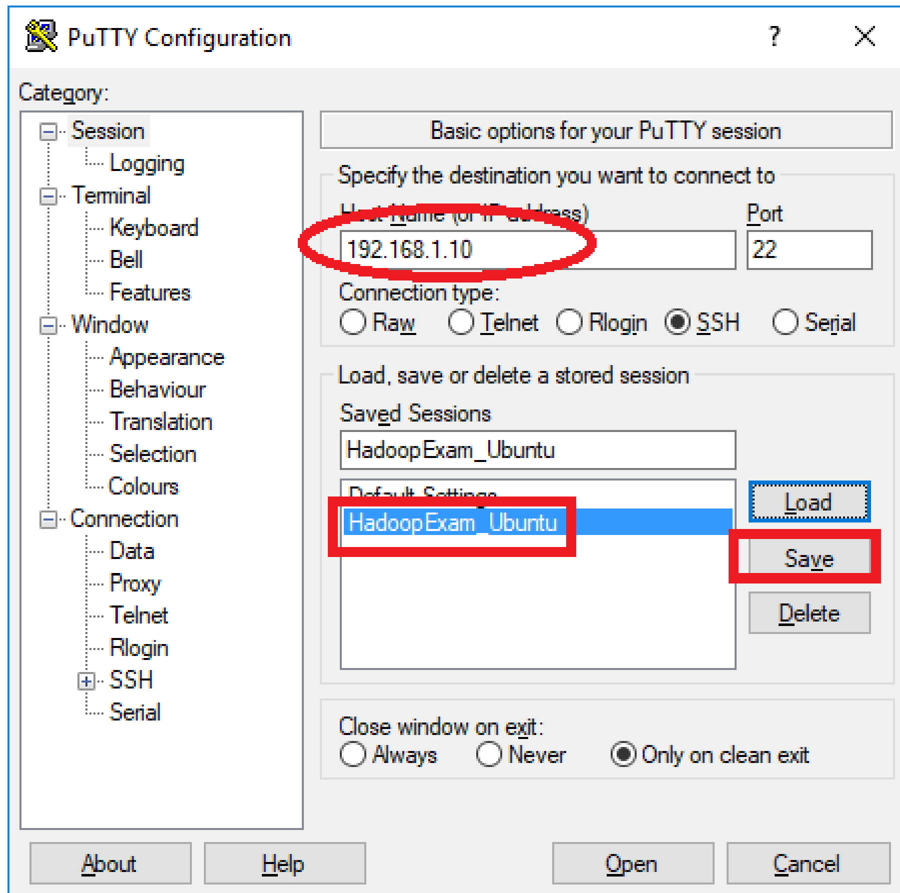
Step 19: From Ubuntu VMWare Machine, get the IP address.



Step 20: Now start putty.exe



Step 21: Save the session with IP address (This will be different in your case).



Step 22: Once connected you will see below popup.

Step 23: login using username and password.

hadoopexam@ubuntu: ~

```
login as: hadoopexam
hadoopexam@192.168.1.10's password:
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 4.4.0-31-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

420 packages can be updated.
328 updates are security updates.

New release '16.04.5 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

hadoopexam@ubuntu:~$
```

Now we will use Putty only to connect with this Ubuntu VM Image to setup any other software and to work with it.

Part-3: Setting Spark 2.0 env on Ubuntu**:** Download the latest version of Spark and install all other required software.

Step-1: Install the required basic software on Linux.

```
sudo apt-get install python-software-properties

sudo apt-add-repository ppa:webupd8team/java

sudo apt-get update

-Install Java8

sudo apt-get install oracle-java8-installer

#Check java version
java -version
```

Step-2 Vi editor settings

```
sudo apt-get install vim

vim ~/.vimrc
set nocompatible
```

Step-3: Install Scala

As we can see: #Spark runs on Java 8+, Python 2.7+/3.4+ and R 3.1+. For the Scala API, Spark 2.3.0 uses Scala 2.11. You will need to use a compatible Scala version (2.11.x).

Scala version → 2.11

```
mkdir scala
cd scala/
wget https://downloads.lightbend.com/scala/2.11.12/scala-2.11.12.tgz

sudo tar xvf scala-2.11.12.tgz

#Update bashrc file
vi  ~/.bashrc

export SCALA_HOME=/home/hadoopexam/scala/scala-2.11.12

export PATH=$PATH:$SCALA_HOME/bin

source ~/.bashrc

scala -version
```

Step 4: Install Python for pyspark (2.7.6 is already there, so we don't have to install it)

```
python
```

Step 5: Install Apache Spark

Install Spark single Node

```
mkdir spark2
cd spark2

(Below link may change, keep checking here for exact URL: https://spark.apache.org/downloads.html

wget http://www-eu.apache.org/dist/spark/spark-2.3.1/spark-2.3.1-bin-hadoop2.7.tgz

tar xvf spark-2.3.1-bin-hadoop2.7.tgz

mv spark-2.3.1-bin-hadoop2.7 spark23

vi ~/.bashrc

export SPARK_HOME=/home/hadoopexam/spark2/spark23
export PATH=$PATH:$SPARK_HOME/bin
```

Step 6: Start Spark

```
source ~/.bashrc

/home/hadoopexam/spark2/spark23/sbin/start-master.sh

Check the Spark UI
http://localhost:8080

http://192.168.1.10:8080/
```

Step 7: Start worker node

```
/home/hadoopexam/spark2/spark23/sbin/start-slave.sh spark://ubuntu:7077

ps -aef | grep -i "Spark"

jps
```

Step 8: Run Spark Example

```
cd /home/hadoopexam/spark2/spark23

./bin/run-example SparkPi 10
```

Step 9: Start the spark-shell and pyspark shell.

```
spark-shell
pyspark
```

Chapter-5 SparkSQL Schema

- Schema Inference
- Explicitly Assign schema
    - Reflection
    - Using StructType and StructField

Schema Inference

You can load data from the raw file or any other data sources. While loading the data you can either infer the schema from the data itself. While loading the JSON or csv file you can mention an option as infer schema with values as True and therefore it can infer the schema based on the data. SparkSQL engine, then sample some data to infer the schema from loaded sample data. Let's see an example below, while reading the data using DataFrameReader (spark.read), we are providing options that data is having header as well as infer a schema. However, sometime this approach may not be useful or results in a way it is expected. Because whatever Schema is inferred by SparkSQL engine, you may not wanted that. Hence, you have to explicitly assign a schema to your data, so that it can be structured accordingly.

```
//format agnostic load method
//Loading csv file
spark.read.format("csv").option("header",true).option("inferSchema",true).load("/home/hadoopexa
m/spark2/sparksql/HadooExam_Training.csv").show()

//Loading json file
//You can specify like json, csv, parquet, orc, text, jdbc etc.
spark.read.format("json").option("header",true).option("inferSchema",true).load("/home/hadoopex
am/spark2/sparksql/he_data_1.json").show()
```

Explicitly assigning schema

There are two approaches by which you can explicitly assign a schema to data.

1. By using Java reflection or Scala case classes
2. Explicitly creating the Schema using StructType and StructField

Schema Inference using reflection

If you know the schema for your data in advance and the total number of fields in the data is less than or equal to 22 then you can use Java case class to create the Schema and same can be assigned to data. See example below

```
//Define a Case class for HadoopExam course detail
case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)

//Create an RDD with 5 HECourses
val courseRDD = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HECourse(4, "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7)))

//Check the types of RDD
courseRDD

//Convert RDD into dataset, as RDD has schema information, so Dataset will automatically infer that schema.
val heCourseDS = courseRDD.toDS

heCourseDS: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

//Print the schema
heCourseDS.schema

//Print each individual datatype
heCourseDS.schema.foreach(println)

//Various representation of DataTypes
heCourseDS.schema.simpleString

//It will give you, how the schema is stored in catalog
heCourseDS.schema.catalogString

// SQL version of the schema
heCourseDS.schema.sql

//Json version of the schema
heCourseDS.schema.json

//Check the schema in formatted JSON
heCourseDS.schema.prettyJson
```

```
scala> case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)
defined class HECourse

scala> val courseRDD = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HE
Course(4, "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7)))
courseRDD: org.apache.spark.rdd.RDD[HECourse] = ParallelCollectionRDD[0] at parallelize at <console>:26

scala> val heCourseDS = courseRDD.toDS
2018-08-28 22:12:32 WARN  ObjectStore:568 - Failed to get database global_temp, returning NoSuchObjectException
heCourseDS: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

scala> heCourseDS.schema
res0: org.apache.spark.sql.types.StructType = StructType(StructField(id,IntegerType,false), StructField(name,StringType,true), StructField(fee,IntegerType,false), Struc
tField(venue,StringType,true), StructField(duration,IntegerType,false))

scala> heCourseDS.schema.foreach(println)
StructField(id,IntegerType,false)
StructField(name,StringType,true)
StructField(fee,IntegerType,false)
StructField(venue,StringType,true)
StructField(duration,IntegerType,false)
scala> heCourseDS.schema.simpleString
res2: String = struct<id:int,name:string,fee:int,venue:string,duration:int>

scala> heCourseDS.schema.catalogString
res3: String = struct<id:int,name:string,fee:int,venue:string,duration:int>

scala> heCourseDS.schema.sql
res4: String = STRUCT<`id`: INT, `name`: STRING, `fee`: INT, `venue`: STRING, `duration`: INT>

scala> heCourseDS.schema.json
res5: String = {"type":"struct","fields":[{"name":"id","type":"integer","nullable":false,"metadata":{}},{"name":"name","type":"string","nullable":true,"metadata":{}},{"
name":"fee","type":"integer","nullable":false,"metadata":{}},{"name":"venue","type":"string","nullable":true,"metadata":{}},{"name":"duration","type":"integer","nullabl
e":false,"metadata":{}}]}

scala> heCourseDS.schema.prettyJson
res6: String =
{
  "type" : "struct",
  "fields" : [ {
    "name" : "id",
    "type" : "integer",
    "nullable" : false,
    "metadata" : { }
  }, {
    "name" : "name",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "fee",
    "type" : "integer",
    "nullable" : false,
    "metadata" : { }
  }, {
    "name" : "venue",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "duration",
    "type" : "integer",
    "nullable" : false,
    "metadata" : { }
  } ]
```

In the above example HECourse is a case class having five fields. And data also it has five fields. We are creating the RDD using this case class and then finally converting into Dataset, so the schema would be preserved which was created using case class.

Explicitly creating schema using StructType and StructFields

You would be using this approach if you don't know the schema in advanced and creating the schema string dynamically based on some criteria or number of fields are more than 22 (Because case classes can support up to 22 fields only)

In this case you will be using StructType and StructField classes to create the schema. StructType will have sequence of StructFields. StructType can be nested as well. For example we can create schema for HECourse as below

```
//Import sql types
import org.apache.spark.sql.types._

//Create Schema for the JSON data
val heschema = StructType(
  StructField("id", LongType, nullable = false) ::
  StructField("name",StringType,nullable = false) ::
  StructField("fee", DoubleType, nullable = false) ::
```

```
StructField("venue", StringType, nullable = false) ::
StructField("Duration", LongType, nullable = false) :: Nil)

//Use defined schema while loading the data
val jsonData=
spark.read.format("json").schema(heschema).load("/home/hadoopexam/spark2/sparksql/he_data_1.json").select("name", "fee" , "venue").where($"fee" > 5000)

//Check the output
jsonData.show()
```

```
scala> import org.apache.spark.sql.types._
import org.apache.spark.sql.types._

scala> val heschema = StructType(
     |   StructField("id", LongType, nullable = false) ::
     |   StructField("name",StringType,nullable = false) ::
     |   StructField("fee", DoubleType, nullable = false) ::
     |   StructField("venue", StringType, nullable = false) ::
     |   StructField("Duration", LongType, nullable = false) :: Nil)
heschema: org.apache.spark.sql.types.StructType = StructType(StructField(id,LongType,false), StructField(name,
StringType,false), StructField(fee,DoubleType,false), StructField(venue,StringType,false), StructField(Duratio
n,LongType,false))

scala> val jsonData= spark.read.format("json").schema(heschema).load("/home/hadoopexam/spark2/sparksql/he_data
_1.json").select("name", "fee" , "venue").where($"fee" > 5000)
jsonData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [name: string, fee: double ... 1 more field
]

scala> jsonData.show()
+------+------+--------+
|  name|   fee|   venue|
+------+------+--------+
|Hadoop|6000.0|  Mumbai|
| HBase|7000.0|Banglore|
+------+------+--------+
```

There are many ways by which you can create schema explicitly, see the example below. Once schema is created you can print it in various format like simple String, Tree or JSON format

```
//Import StructType class
import org.apache.spark.sql.types.StructType

//It is not type safe, actual data type derivation happens during runtime
//If values and types does not matched during schema assignment it will throw error
//Adding the fields to StructTypes one by one
val sampleSchema = new StructType().add("course_id", "int").add("course_name",
"string").add("course_fee", "int").add("venue", "string")

//Sample schema using DSL
//It is not type safe, actual data type derivation happens during runtime
//If values and types does not matched during schema assignment it will throw error
val sampleSchema = new
StructType().add($"course_id".int).add($"course_name".string).add($"course_fee".int).add($"venue".string)

//Another way this is type-safe available compile time as well as run time
```

```
import org.apache.spark.sql.types.{IntegerType, StringType}

val sampleSchema = new StructType().add("course_id", IntegerType).add("course_name",
StringType).add("course_fee", IntegerType).add("venue", StringType)

//Print schema
sampleSchema.printTreeString
//You can print json schema as well

println(sampleSchema.prettyJson)

//Create Row object and assign schema
//Import the required classes and package
import org.apache.spark.sql._
import org.apache.spark.sql.types._

//Define a course_detail type which can hold upto three venues
val course_detail = StructType( StructField("name", StringType, true) :: StructField("Fee", IntegerType,
false) :: StructField("City", StringType, false) :: StructField("Zip", IntegerType, false) ::Nil)

//Check the structure of the defined schema
course_detail.prettyJson
course_detail.printTreeString

// Create  Rows instances
val row = Row("Hadoop" ,5000,"Mumbai" ,400001 )
val row1 = Row("Spark" ,5000,"Pune" ,111045 )
val row2 = Row("Cassandra" ,5000,"Banglore" ,530068 )

//Accessing values from Row using ordinal position
row(0)
row(2)
row(3)

//You can access fields of Row based on type as well
 row.getString(0)
 row.getString(2)
 row.getInt(3)

//Now create the DataFrame using the schema we have created above
val HEDF= spark.createDataFrame(spark.sparkContext.parallelize(Seq(row, row1,
row2)),course_detail)

//Check whether valid schema is assigned or not
HEDF.printSchema

//Check the data
HEDF.show()
HEDF.schema
```

```
scala> import org.apache.spark.sql.types.StructType
import org.apache.spark.sql.types.StructType

scala> val sampleSchema = new StructType().add("course_id", "int").add("course_name", "string").add("course_fe
e", "int").add("venue", "string")
sampleSchema: org.apache.spark.sql.types.StructType = StructType(StructField(course_id,IntegerType,true), Stru
ctField(course_name,StringType,true), StructField(course_fee,IntegerType,true), StructField(venue,StringType,t
rue))

scala> val sampleSchema = new StructType().add($"course_id".int).add($"course_name".string).add($"course_fee".
int).add($"venue".string)
sampleSchema: org.apache.spark.sql.types.StructType = StructType(StructField(course_id,IntegerType,true), Stru
ctField(course_name,StringType,true), StructField(course_fee,IntegerType,true), StructField(venue,StringType,t
rue))

scala> import org.apache.spark.sql.types.{IntegerType, StringType}
import org.apache.spark.sql.types.{IntegerType, StringType}

scala> val sampleSchema = new StructType().add("course_id", IntegerType).add("course_name", StringType).add("c
ourse_fee", IntegerType).add("venue", StringType)
sampleSchema: org.apache.spark.sql.types.StructType = StructType(StructField(course_id,IntegerType,true), Stru
ctField(course_name,StringType,true), StructField(course_fee,IntegerType,true), StructField(venue,StringType,t
rue))
scala> sampleSchema.printTreeString
root
 |-- course_id: integer (nullable = true)
 |-- course_name: string (nullable = true)
 |-- course_fee: integer (nullable = true)
 |-- venue: string (nullable = true)


scala> println(sampleSchema.prettyJson)
{
  "type" : "struct",
  "fields" : [ {
    "name" : "course_id",
    "type" : "integer",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "course_name",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "course_fee",
    "type" : "integer",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "venue",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  } ]
scala> import org.apache.spark.sql._
import org.apache.spark.sql._

scala> import org.apache.spark.sql.types._
import org.apache.spark.sql.types._

scala>

scala> val course_detail = StructType( StructField("name", StringType, true) :: StructField("Fee", IntegerType
, false) :: StructField("City", StringType, false) :: StructField("Zip", IntegerType, false) ::Nil)
course_detail: org.apache.spark.sql.types.StructType = StructType(StructField(name,StringType,true), StructFie
ld(Fee,IntegerType,false), StructField(City,StringType,false), StructField(Zip,IntegerType,false))

scala> val row = Row("Hadoop" ,5000,"Mumbai" ,400001 )
row: org.apache.spark.sql.Row = [Hadoop,5000,Mumbai,400001]

scala> val row1 = Row("Spark" ,5000,"Pune" ,111045 )
row1: org.apache.spark.sql.Row = [Spark,5000,Pune,111045]

scala> val row2 = Row("Cassandra" ,5000,"Banglore" ,530068 )
row2: org.apache.spark.sql.Row = [Cassandra,5000,Banglore,530068]

scala> val HEDF= spark.createDataFrame(spark.sparkContext.parallelize(Seq(row, row1, row2)),course_detail)
HEDF: org.apache.spark.sql.DataFrame = [name: string, Fee: int ... 2 more fields]

scala> HEDF.printSchema
root
 |-- name: string (nullable = true)
 |-- Fee: integer (nullable = false)
 |-- City: string (nullable = false)
 |-- Zip: integer (nullable = false)


scala> HEDF.schema
res12: org.apache.spark.sql.types.StructType = StructType(StructField(name,StringType,true), StructField(Fee,I
ntegerType,false), StructField(City,StringType,false), StructField(Zip,IntegerType,false))
```

When you create or assign schema, it will give structure to your data with the following three things.

- Name of the columns
- Types of the columns
- Nullability (Whether the value of the column can be null or not)

Dataset will always have schema, where

- Schema can be inferred at runtime using case classes.
- At compile time you can assign schema explicitly.

Schema is a StructField, which has collections of StructFields and each StructField represent a column name in a Dataset.

```
StructType =➔ [Collection of StructFields]
```

You will be using below package

```
org.apache.spark.sql.types
```

In SparkSQL, Catalyst SQL parser is responsible for deriving actual datatypes. All the Datatypes information is stored in String format in external catalog. And can be represented as

- Catalog string (the way it is stored in catalog)
- JSON: Compact JSON format of datatype information
- Pretty Json: Indented JSON format of Datatypes information.
- Simple String: Readable string representation for the type.
- Sql string: SQL representation of Datatypes

Values in StriuctType: Values in StructType

- Represented using Row object.
- A StructType will hold a StructFiled, however StructField can have another StructType in it, which can represent nested or complex Row object.

Chapter 6: SparkSQL abstractions & Other Objects

- Introduction
- SparkSession
- SparkConf
- SparkSQL Row object
- Resilient Distributed Dataset
- DataFrame
- Dataset
- Dataset and Catalyst

Introduction

In this chapter we will be focusing on important abstractions like DataSet, DataFrame, SparkSession and SparkConf objects. You will come across all these objects while working with the SparkSQL and learning them in detail is important.

About SparkSession: SparkSession is an entry point for the Spark programming, and you can use Dataset and DataFrame API through SparkSession object. In Spark-shell SparkSession object is already available as “spark” variable. If you are creating an Application then you have to use below method to create an instance of SparkSession, each Spark application required at least one instance of SparkSession. You can use below approach to create a session.

API Doc Link

```
SparkSession.builder
    .master("local")
    .appName("HadoopExamAPP")
    .config("spark.sql.autoBroadcastJoinThreshold", "100000") //example to set config values
    .getOrCreate()
```

getOrCreate method will get the existing session if it is available and if not available than create the new one and return. However, internally it checks first that if there is any SparkSession available on thread-local instance, if yes than return it. If no instance available in thread-local than check if there is any global SparkSession is available if it is available than return it.

Even SparkSession is already exists and returned and you change the configuration parameters on that SparkSession than it will overwrite all the existing configuration values.

If you wanted to use Hive features like connectivity with the Hive metastore, support of the Hive Ser-De and Hive UDF (User defined functions) than you can enable the Hive support using below method

```
SparkSession.builder.enableHiveSupport()
```

If you are working in spark-shell or notepad wanted to get the existing SparkSession object use the below method.

```
SparkSession.builder().getOrCreate()
```

SparkSession.builder.master will help you to provide the cluster manager URLs, whether it's a local cluster manager or a local with multiple cores or you can provide YARN cluster manager URL etc.

```
//Its local, cluster manager, and Spark will run locally
SparkSession.builder.master("local")

//Run locally with n number of cores
SparkSession.builder.master("local[4]")

//Run with Spark Standalone cluster manager
SparkSession.builder.master("spark://master:port")
```

When you use local mode than it will run locally and Spark Job will not be distributed with other nodes in the cluster, it is good for testing. Value "n" represent how many cores you want to use from the local node while running your job. This approach is good if you are running Spark code using Eclipse or intelliJ and want to debug your code.

Using the YARN as master:

- **Client mode**: In this case Spark driver program will run on the client machine from where you initiate the Spark Job. It may be possible that this machine may not be part of your entire Spark cluster (can be a gateway machine your desktop machine etc. which is connected with the YARN cluster). And all the Spark Jobs executor will be executed on the YARN cluster Node Manager. (Learn YARN in detail from this link on http://hadoopexam.com)

- **Cluster mode**: In production you will be using this approach, it will run the Spark Driver in one of the node in the YARN cluster and provide High Availability.

You can use below approach to do various YARN related configurations

```
//resource manager host
SparkSession.Builder.config("spark.hadoop.yarn.resourcemanager.hostname","HadoopExamHOST");

//Address of resource manager of YARN
SparkSession.Builder.config ("spark.hadoop.yarn.resourcemanager.address","HadoopExamHOST:8032");

//Name node details of the HDFS
SparkSession.Builder.config("spark.yarn.access.namenodes", "hdfs://HadoopExamHOST1:8020,hdfs://HadoopExamHOST2:8020");
```

Remember:

- You should start using SparkSession instead of SQLContext and HiveContext previously you were using them for SparkSQL and Hive respectively.
- Single Spark application can have more than one SparkSession object.

Submitting Spark applications

Once you created SparkSession object in your application and using the below spark-submit command you can submit the Spark job

```
//Example of submitting Spark Application
spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \  # can be client for client mode
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000
```

SparkConf object (**API Doc Link**):

We have seen that there are various options which can help you to configure Spark applications either using spark-submit or running applications using spark-shell. All uses this SparkConf object. All the configurations are done using key-value pairs. We can create SparkConf instance using new SparkConf(), which are then converted as Java system properties. If there are any parameters specified on system and you want to override them then you can create new SparkConf() object and set the properties you want to override.

If you are writing test cases than you can create instance using new SparkConf(false), which will ignore all the properties which are defined at system level and only the properties which you have defined will be consider, during unit testing, to set the multiple properties at once you can use method chaining for example

```
new SparkConf(Boolean loadDefault).setMaster("local").setAppName("HadoopExamAPP")
```

Default Value of the parameter loadDefault is true, it means use all the default properties if not overridden. If you set loadDefault parameter as "false", then you have to provide all required parameters.

Remember: As soon as you submit your application to cluster, it will be cloned and passed to all the nodes on the cluster where your submitted application will be running and once it is submitted its configuration cannot be changed further, hence Spark does not support changing the Spark configuration at runtime.

Providing custom rules and optimization technique

As we have learned in previous chapter that we can provide custom analyze rules, optimization rules, customized parsers or any new planning strategy. Hence, catalyst optimizer can be extended and customized, in that case you have to use below method from SparkSession

```
public SparkSession.Builder
withExtensions(scala.Function1<SparkSessionExtensions,scala.runtime.BoxedUnit> f)
```

This will inject custom extension. This feature was available since Spark 2.2.0 onwards only.

There is one more approach through which you can provide rule customization for Apache SparkSession by using configurations, you will be using “spark.sql.extensions” config key and “SparkSessionExtensions” as a value part, which has implementation for custom rules. That is what is one of the main intention of the SparkSQL developer that end user can provide their own optimization techniques to catalyst optimizer.

SparkSQL Row (Catalyst Row) object (**API Doc Link**):

It is a generic object in SparkSQL which represent one record in a DataFrame and you can access the fields from Row object using either column name or based on their index position. You can create Row object by providing values like

```
//Create Row object using values
Row(value1, value2, value3… valuen)

//creating Row object using Seq of values
Row(Seq(value1, value2, value3… valuen))
```

It seems they are very similar to array, and you can access the fields using

1. Index Position
2. Column Name
3. Scala pattern matching

A Row object can have schema as well, but that is not mandatory. Row encoders are responsible for assigning schema to a row. You can access the schema for a Row object using Row.schema() method.

Let’s see the example with the Row instance.

```
//Import Row object
import org.apache.spark.sql.Row

// Create  Rows instances
val row = Row("Hadoop" ,5000,"Mumbai" ,400001 )
val row1 = Row("Spark" ,5000,"Pune" ,111045 )
val row2 = Row("Cassandra" ,5000,"Banglore" ,530068 )

//Accessing values from Row using ordinal position
row(0)
row(1)
row(2)
row(3)
```

```
//You can access fields of Row based on type as well
//But if you are using typed methods than you have to check whether these values are not null.
 row.getString(0)
 row.getInt(1)
 row.getString(2)
 row.getInt(3)

//Import types
import org.apache.spark.sql._
import org.apache.spark.sql.types._

//Define a course_detail type which can hold upto three venues
val course_detail = StructType( StructField("name", StringType, true) :: StructField("Fee", IntegerType,
false) :: StructField("City", StringType, false) :: StructField("Zip", IntegerType, false) ::Nil)

//Now create the DataFrame using the schema we have created above
val HEDF= spark.createDataFrame(spark.sparkContext.parallelize(Seq(row, row1,
row2)),course_detail)

//Check whether valid schema is assigned or not
HEDF.printSchema

//Check the data
HEDF.show()
HEDF.schema
```

```
scala> import org.apache.spark.sql.Row
import org.apache.spark.sql.Row

scala> val row = Row("Hadoop" ,5000,"Mumbai" ,400001 )
row: org.apache.spark.sql.Row = [Hadoop,5000,Mumbai,400001]

scala> val row1 = Row("Spark" ,5000,"Pune" ,111045 )
row1: org.apache.spark.sql.Row = [Spark,5000,Pune,111045]

scala> val row2 = Row("Cassandra" ,5000,"Banglore" ,530068 )
row2: org.apache.spark.sql.Row = [Cassandra,5000,Banglore,530068]

scala> row(0)
res0: Any = Hadoop

scala> row(1)
res1: Any = 5000

scala>  row.getString(0)
res2: String = Hadoop

scala>  row.getInt(1)
res3: Int = 5000

scala> import org.apache.spark.sql._
import org.apache.spark.sql._

scala> import org.apache.spark.sql.types._
import org.apache.spark.sql.types._

scala> val course_detail = StructType( StructField("name", StringType, true) :: StructField("Fee", IntegerType, false) :: StructField("City", StringType, false) :: Stru
ctField("Zip", IntegerType, false) ::Nil)
course_detail: org.apache.spark.sql.types.StructType = StructType(StructField(name,StringType,true), StructField(Fee,IntegerType,false), StructField(City,StringType,fal
se), StructField(Zip,IntegerType,false))

scala> val HEDF= spark.createDataFrame(spark.sparkContext.parallelize(Seq(row, row1, row2)),course_detail)
2018-08-29 16:02:21 WARN  ObjectStore:568 - Failed to get database global_temp, returning NoSuchObjectException
HEDF: org.apache.spark.sql.DataFrame = [name: string, Fee: int ... 2 more fields]

scala> HEDF.show()
+---------+----+--------+------+
|     name| Fee|    City|   Zip|
+---------+----+--------+------+
|   Hadoop|5000|  Mumbai|400001|
|    Spark|5000|    Pune|111045|
|Cassandra|5000|Banglore|530068|
+---------+----+--------+------+
```

Resilient Distributed Dataset

RDD is the lowest representation of data in Spark. Every processing in Spark done using RDD, whether you use SparkSQL abstractions like DataFrame/Dataset. An RDD spread across multiple machines in a Spark cluster, it provides APIs so you can work on it. You can create an RDD from different types of data source, e.g. text files, a database via JDBC, etc.

Apache Definitions of RDD: RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

To learn RDD API and For Hands On session we recommend this training from Spark Core training on http://HadoopExam.com

DataFrame:

Similar to RDD, it is also distributed and immutable collections of data. You can imagine DataFrame as an RDBMS table with column name and rows. But DataFrame rows are divided and saved across various machines in Spark cluster as shown in below image.

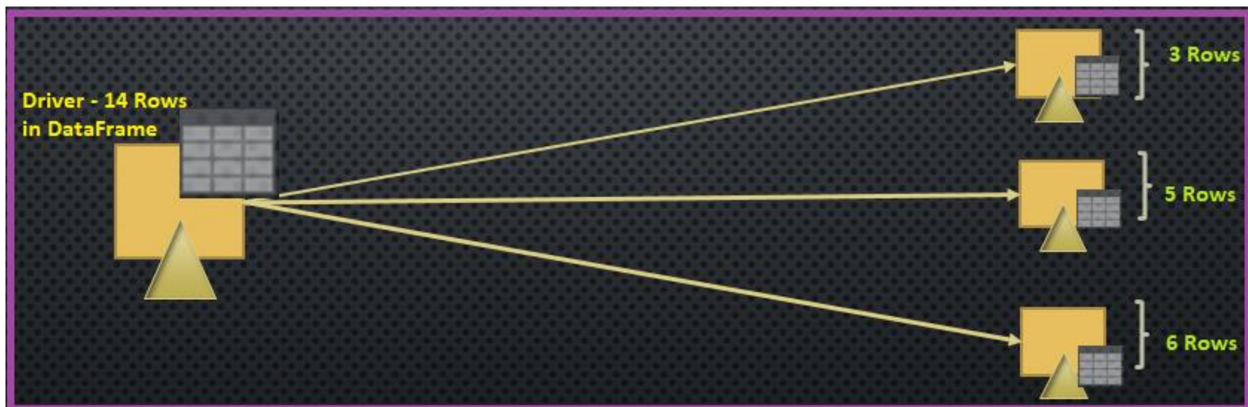


Figure 19: Partitioned DataFrame object across cluster nodes

- DataFrame helps in writing SparkSQL code using simpler API, and it is very similar to Python and R DataFrame.
- DataFrame is higher level abstraction of RDD.
- DataFrame represents Dataset with the generic Row object. So you can have below similarities between Dataset and DataFrame.

```
DataFrame == Dataset<Row>
```

Here Row is a generic object, and does not have type information attached to it.

Whenever you work with Dataset or DataFrame you are working with the Row objects. In case of DataFrame it can be generic Row object and in case of Dataset it will be typed Dataset objects.

Even, you can apply schema information to DataFrame object as well. To work with DataFrame you have following two approaches.

- SQL queries
- Query DSL (It can check the syntax at compile time)

Programmatically assigning schema: You will be using this approach when

- Schema needs to be created dynamically based on some conditions or requirement.
- If total number of fields are more than 22

For creating schema programmatically, we have to use following Spark classes, specific to Schema

- StructType
- StructFields

Where StructType is a sequence of StructFields. It can be done as below

```
var heDF = spark.read.format("csv").schema(customSchemaString).load("csv file path").toDF("columnNames String")
```

In above case, whatever column names and StructFields you have provided in custom schema must match. If it does not matches than there will be an error.

Exercise - Work with DataFrames

```
//Create an RDD with 5 HECourses
val courseRDD = sc.parallelize(Seq((1, "Hadoop", 6000, "Mumbai", 5), (2, "Spark", 5000, "Pune", 4), (3, "Python", 4000, "Hyderabad", 3) , (4, "Scala", 4000, "Kolkata", 3), (5, "HBase", 7000, "Banglore", 7)))

//Create a DataFrame from RDD
courseRDD.toDF

//Lets check the data
courseRDD.toDF.show

//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail
case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)

//Create a DataFrame with 5 HECourses
spark.createDataFrame(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HECourse(4, "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7))).show()

//Creating DataFrame from csv file
val heDF = spark.read.format("com.databricks.spark.csv").option("header", "true").load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv")
heDF.printSchema
```

```
scala> val courseRDD = sc.parallelize(Seq((1, "Hadoop", 6000, "Mumbai", 5), (2, "Spark", 5000, "Pune", 4), (3, "Python", 4000, "Hyderabad", 3) , (4, "Scala", 4000, "Kol
kata", 3), (5, "HBase", 7000, "Banglore", 7)))
courseRDD: org.apache.spark.rdd.RDD[(Int, String, Int, String, Int)] = ParallelCollectionRDD[7] at parallelize at <console>:31

scala> courseRDD.toDF
res5: org.apache.spark.sql.DataFrame = [_1: int, _2: string ... 3 more fields]

scala> case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)
defined class HECourse

scala> spark.createDataFrame(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HECourse(4,
 "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7))).show()
+---+------+----+---------+--------+
| id|  name| fee|    venue|duration|
+---+------+----+---------+--------+
|  1|Hadoop|6000|   Mumbai|       5|
|  2| Spark|5000|     Pune|       4|
|  3|Python|4000|Hyderabad|       3|
|  4| Scala|4000|  Kolkata|       3|
|  5| HBase|7000| Banglore|       7|
+---+------+----+---------+--------+


scala> val heDF = spark.read.format("com.databricks.spark.csv").option("header", "true").load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv")
heDF: org.apache.spark.sql.DataFrame = [ID: string, Name: string ... 4 more fields]

scala> heDF.printSchema
root
 |-- ID: string (nullable = true)
 |-- Name: string (nullable = true)
 |-- Fee: string (nullable = true)
 |-- Venue: string (nullable = true)
 |-- Date: string (nullable = true)
 |-- Duration: string (nullable = true)
```

In the above example we can see that there are multiple ways by which we can create DataFrame like from RDD, from Sequence of Case classes and loading files etc.

DataFrame is an un-typed or generic collection of rows. You can convert Dataset to DataFrame using a method toDF() of Dataset, also this is one of the good way if you want to re-name the columns in Dataset, we will see example in next section.

Dataset (API Doc Link) : It is available since Spark 1.6, Dataset is available only in Scala and Java version of SparkSQL and not available in PySpark and SparkR. Dataset is a strongly typed, hence each member of the Row in a Dataset will have assigned column name as well as data type. If there is no datatype attached to the columns in Dataset<Row> object than that would be called a DataFrame. So we can say that DataFrame is equivalent to Dataset<Row>. There are two types of operations which you can apply in the Dataset which are Transformation and Actions very similar to RDD.

- **Dataset Transformations**: Transformation will work on Dataset and create another DataSet. Dataset is an immutable object, hence transformation will always results in a new Dataset object. Transformations are lazy operations and will be triggered only when an action will be called on Dataset. Example of transformation operations are map(), filter(), select, aggregate(using groupBy) operations.

- **Dataset Actions**: Action will always trigger computations and return final results back to Driver program. Example of actions are count (), show (), collect (), even writing Dataset results on the filesystem is also an action.

Question: What do you think about printSchema() method of Dataset is an action or transformation?

Answer: printSchema does not initiate any computation and no new Job will be launched. All the schema information stored with the Dataset will be shown, hence it is a transformation. (It's a common interview question)

Dataset internally stores the logical plan and represents the computation which is required to produce this Dataset, as soon as action is triggered on that Dataset this logical plan will be submitted for optimization by catalyst optimizer and finally one or more than one physical plan will be generated and

cost based optimizer will be choosing the best physical plan (you can provide hints as well in some cases, so that catalyst choose the plan based on the hint provided by you). If you use explain (Boolean) method of Dataset, it will give you the entire detail about the plan which will be used.

Dataset will always have **Encoders** (We will be having entire chapter dedicated to Encoders), If you know the Java serialization and de-serialization then Encoders are the same thing but much more efficient than Java default serialization and de-serialization mechanism. For all the commonly used datatypes like int, float, Boolean etc. encoders are already provided by the SparkSQL. If you are having some custom datatypes than you have to define your own custom Encoders. Let's say we have an object called *HECourse(int ID, name String, int fee)* with three fields, which are common primitive datatypes and Spark already have defined Encoders for Integer and String, so you do not have to create custom encoders.

With the help of Encoders, Spark at runtime generate binary data for this HECourse instance and even SparkSQL is so smart enough that it will work only on this binary data, without converting back them to original object that gives a lot of performance boost for the Catalyst optimizer. And binary data take much less memory space than actual object. We will be using DataFrameReader (discussed in next chapter) to create Dataset object.

Dataset (Type-safety)

Both Dataset and DataFrame are higher level abstraction to work with Apache SparkSQL. Using this API you can work with structured (e.g. csv) as well as with semi-structured data (JSON). Dataset is an abstraction which has features of both RDD and DataFrame. Datasets are created or represent object in JVM. If any object present in JVM it means it had resolved its reference name as well as its type is known to the system.

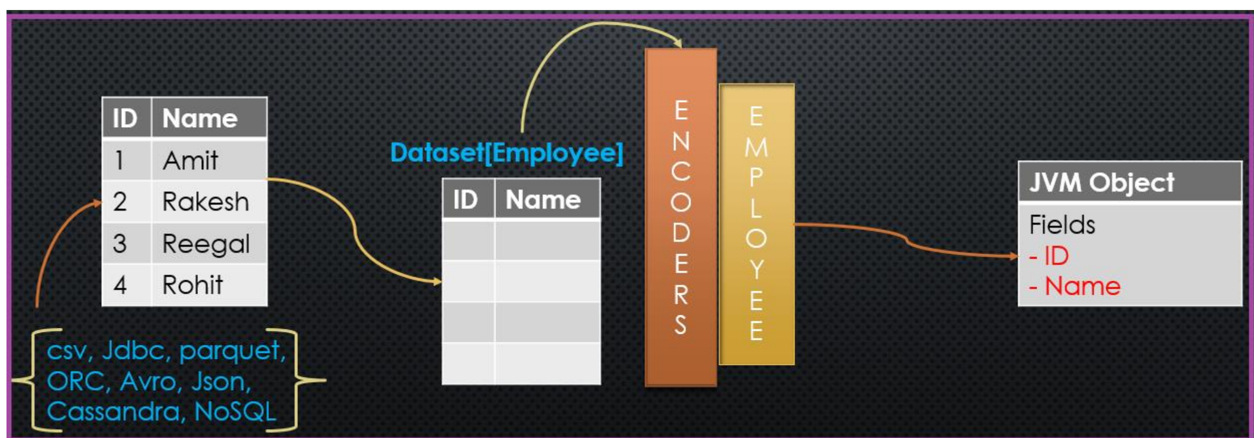


Figure 20: Structured Data to Dataset and Its JVM fields mapping

Similar to RDD, Dataset distributed as well immutable. Hence, if you want to create new Dataset from existing Dataset you have to use transformation API, which can help you to get the desired Dataset from existing Dataset.

- It is there since Spark 1.6
- More focus was performance, and use of SparkSQL catalyst engine.
- Dataset is not available in Python and R language Spark API. But same functionality can be achieved using DataFrame because Python is dynamic type of language. And Scala and Java are static type language.
- Since Spark 2.0, you don't have to learn separate API for DataFrame and Dataset. There will be a single API.

DataFrame to Dataset conversion:

Covert a DataFrame to Dataset using Scala case classes. Case classes helps in deriving schema using reflection.

```
Df.as[T] → DS
```

Case classes can only be used when

- Types of the all the fields are known in advance.
- Number of fields in a case class are less than or equal to 22 (It is a limit of Scala language)
- In case of DataFrame, if column name are not known in advanced then it will be created using generic column named like (_c0, _c1,…._cn)

Dataset and Type-safety

Hence, whenever you work with the Dataset, it means types of each element in Dataset is known. Knowing the data type gives huge benefit while code needs to be optimized by the catalysts optimizer. And also if you are using any Integrate Development Environment like eclipse or IntelliJ they can predict the syntax and compile time error. If code is written using Dataset, same cannot be available using DataFrame and RDD.

Dataset and Catalyst optimizer

Both DataFrame and Dataset take advantage of Catalyst optimizer, but it is possible DataFrame does not have type information of the elements and Catalyst may not be able to do the full optimization. However, Dataset will always have type information attached, hence Catalyst optimizer can take advantage of this to optimize the code written using Dataset. One of the example by which Dataset take advantage of catalyst optimizer is Dataset will expose its expressions and data fields to a query planner where all the fields and types would be resolved at first.

Even after Spark 2.0 DataFrame is also a Dataset of generic objects. We can conclude that relation as below.

```
DataFrame == Dataset[Row]
```

Where Row are generic type of an object.

Dataset and compile time type safety

While working with the Dataset types of all the data in Dataset is known at compile time only and because of that Catalyst optimizer eagerly check whenever you write the code whether types you

are using is compatible or not (very similar the way, you write Java code in eclipse, and you get to know whether you have used valid type or not against the data). If they are not compatible than compile time error will be thrown and you need to fix it at compile time only. Once, you fix compile time, it will save you having type related issue in the production (run-time).

If you are working with the RDD, which has type information attached for each element stored in it. But behind the scene there is no optimizer which can help in checking that the code you are writing is correct w.r.t. type. Hence, if you have written code directly using RDD than you can get production issue regarding data types. Therefore we can say catalyst optimizer will help in getting type-safety at compile time as well as optimizing the code written using Spark SQL.

Dataset also support Lambda function similar to RDD. Other than that you can use SQL query and DSL (Domain specific language).

Working with Dataset: You can assume Dataset as a logical plan in a SparkSession, a logical plan describe how all the computations can be applied. We can create Dataset using

- Files (csv, sequence, avro, parquets, JSON etc)
- From Hive tables.
- RDBMS tables.
- NOSQL databases like Cassandra, HBase etc.

As Datasets have schema information column types, column names. Hence, Dataset can have following variations of the operations

1. Using Scala functions or lambdas

```
Dataset.filter(fee => fee>=6000).count()
```

2. Column bases SQL expressions

```
Dataset.filter('fee >=6000).count()
Dataset.filter($fee >=6000).count()
```

3. SQL Query

```
Dataset.createorReplaceTempView("HECOurse")
sql(select * from HECourse)
```

Only Dataset API in SparkSQL which can provide syntax and analysis checks at compile time. Using DataFrame, RDD or SQL query you cannot do or achieve compile time safety. Because Dataset has type information, then Row data can be

- Accessed using fields
- And no need to cast values of the accessed columns.

Dataset needs below things

- SparkSession (This is a transient variable)
- Query execution plan (This is a transient variable)
- Encoders (SerDe and this is not a transient variable of Dataset)

Encoders of Datasets are not transient because it is used while serializations and de-serialization. But SparkSession and Execution plan is not required during serialization.

Transient: If a variable is transient then it would not be serialized.

To select particular column from the Dataset, you can use column name or col function of the Dataset. SparkSQL revolves around Dataset, which internally uses Catalyst optimizer. Let's see few example to work with Dataset and then slowly we will move further for more API functions.

Exercise : Convert RDD to Spark Dataset and Use DSL as well as SQL syntax

```
//Define a Case class for HadoopExam course detail
case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)

//Create an RDD with 5 HECourses
val courseRDD = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HECourse(4, "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7)))

//Check the types of RDD
courseRDD

//Convert RDD into dataset, as RDD has schema information, so Dataset will automatically infer that schema.
val heCourseDS = courseRDD.toDS

heCourseDS: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

//Select the courses conducted in Mumbai, having price more than 5000
//Also, you can select the columns, you need (It is DSL)
val filteredDS = heCourseDS.where('fee >5000).where('venue==="Mumbai").select('name,'fee, 'duration)

//You can see filteredDS is a DataFrame and not a Dataset (There is a slight difference between DataFrame and Dataset since Spark 2.0), we will discuss about this later on
filteredDS

//Lets make code more SQL friendly as it is SparkSQL
//Register Dataset as temporary view and will be added in Catalog
heCourseDS.createOrReplaceTempView("T_HECOURSE")

//Use SQL Query
val filteredSQLDS = sql("SELECT * FROM T_HECOURSE  WHERE fee > 5000 AND venue = 'Mumbai' ")

//Show the result
filteredSQLDS.show()
```

```
scala> case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)
defined class HECourse

scala> val courseRDD = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HE
Course(4, "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7)))
courseRDD: org.apache.spark.rdd.RDD[HECourse] = ParallelCollectionRDD[17] at parallelize at <console>:33

scala> val heCourseDS = courseRDD.toDS
heCourseDS: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

scala> val filteredDS = heCourseDS.where('fee >5000).where('venue==="Mumbai").select('name,'fee, 'duration)
filteredDS: org.apache.spark.sql.DataFrame = [name: string, fee: int ... 1 more field]

scala> filteredDS
res8: org.apache.spark.sql.DataFrame = [name: string, fee: int ... 1 more field]

scala> filteredDS.show
+------+----+--------+
|  name| fee|duration|
+------+----+--------+
|Hadoop|6000|       5|
+------+----+--------+


scala> heCourseDS.createOrReplaceTempView("T_HECOURSE")

scala> val filteredSQLDS = sql("SELECT * FROM T_HECOURSE  WHERE fee > 5000 AND venue = 'Mumbai' ")
filteredSQLDS: org.apache.spark.sql.DataFrame = [id: int, name: string ... 3 more fields]

scala> filteredSQLDS.show()
+---+------+----+------+--------+
| id|  name| fee| venue|duration|
+---+------+----+------+--------+
|  1|Hadoop|6000|Mumbai|       5|
+---+------+----+------+--------+
```

Spark Case classes: Spark case classes are same as Java POJO, you just define the name of the class and all the member variable with their types and Scala will create internally Java Beans for example in below case name of the class is HECourse and there are in-total 5 fields and for each fields getter and setter methods will be defined by the Scala. Other than that Scala will define toString and hashCode() method as well for the case classes. Case classes are very convenient ways to define schema for the data in Dataset as well as RDD.

```
case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)
```

Case class has one limitation, if number of fields in your class is more than 22 than case classes cannot be used, because Scala has a limit for case class with upto 22 fields. Then what to do if there are more than 22 fields in case of case class, you have to create schema your own using StructType and StructField and then bind that schema to your Dataset values.

Dataset vs RDD operations

If you have already been using RDD for your Spark programming, you will see that working with the SparkSQL for similar operation is much easier that directly working with the RDD.

Other than that we can conclude that

- Using RDD you cannot run SQL query while with Dataset you can do
- Using RDD code optimization is your headache and using Dataset it is taken care by Catalyst optimizer
- Writing transformation code is much more convenient if used with Dataset than RDD.
- Dataset uses more efficient Encoders than RDD. Hence, further improvement for the performance.
- You can visualize the data in tabular format, which is not always possible with the RDD. Hence, it make writing code even friendlier.
- Most of the time you will be writing lesser code while using Dataset than RDD for doing the same operations.
- Dataset is highly performant that RDD.

Converting an RDD to Dataset: You can convert an RDD to Dataset using toDS method as below

```
val heCourseDS = courseRDD.toDS
```

Print the explain plan in all three cases

Some understanding of explain plans

In this case syntax and datatypes are checked during Analysis phase only, and in case of DataFrame it is not possible.

```
//Type-1 Using Scala function (Lambda)
heCourseDS.filter(record => record.fee > 5000).explain(true)

//Type-2 Column based SQL expression
heCourseDS.filter('fee > 5000).explain(true)

//Type-3 As SQL Query
heCourseDS.filter("fee > 5000").explain(true)
```

```
scala> heCourseDS.filter(record => record.fee > 5000).explain(true)
== Parsed Logical Plan ==
'TypedFilter <function1>, class $line34.$read$$iw$$iw$HECourse, [StructField(id,IntegerType,false), StructField(name,StringType,true), StructField(fee,IntegerType,false
), StructField(venue,StringType,true), StructField(duration,IntegerType,false)], unresolveddeserializer(newInstance(class $line34.$read$$iw$$iw$HECourse))
+- AnalysisBarrier
      +- SerializeFromObject [assertnotnull(assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true])).id AS id#98, staticinvoke(class org.apache.spark.unsafe.types
.UTF8String, StringType, fromString, assertnotnull(assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true])).name, true, false) AS name#99, assertnotnull(assertnot
null(input[0, $line34.$read$$iw$$iw$HECourse, true])).fee AS fee#100, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull
(assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true])).venue, true, false) AS venue#101, assertnotnull(assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, t
rue])).duration AS duration#102]
         +- ExternalRDD [obj#97]

== Analyzed Logical Plan ==
id: int, name: string, fee: int, venue: string, duration: int
TypedFilter <function1>, class $line34.$read$$iw$$iw$HECourse, [StructField(id,IntegerType,false), StructField(name,StringType,true), StructField(fee,IntegerType,false)
, StructField(venue,StringType,true), StructField(duration,IntegerType,false)], newInstance(class $line34.$read$$iw$$iw$HECourse)
+- SerializeFromObject [assertnotnull(assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true])).id AS id#98, staticinvoke(class org.apache.spark.unsafe.types.UTF8S
tring, StringType, fromString, assertnotnull(assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true])).name, true, false) AS name#99, assertnotnull(assertnotnull(i
nput[0, $line34.$read$$iw$$iw$HECourse, true])).fee AS fee#100, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(asser
tnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true])).venue, true, false) AS venue#101, assertnotnull(assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true]))
.duration AS duration#102]
   +- ExternalRDD [obj#97]

== Optimized Logical Plan ==
SerializeFromObject [assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true]).id AS id#98, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType,
 fromString, assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true]).name, true, false) AS name#99, assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true]).
fee AS fee#100, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true]).venue
, true, false) AS venue#101, assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true]).duration AS duration#102]
+- Filter <function1>.apply
   +- ExternalRDD [obj#97]

== Physical Plan ==
*(1) SerializeFromObject [assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true]).id AS id#98, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, String
Type, fromString, assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true]).name, true, false) AS name#99, assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, tr
ue]).fee AS fee#100, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true]).
venue, true, false) AS venue#101, assertnotnull(input[0, $line34.$read$$iw$$iw$HECourse, true]).duration AS duration#102]
+- *(1) Filter <function1>.apply
   +- Scan ExternalRDDScan[obj#97]
```

Local Datasets: A Dataset is considered local, if any calculation applied on Dataset does not use the executors on the cluster. But queries on the Dataset can be optimized to run locally, on the same node from where it is submitted.

Dataset and Project Tungsten

As we mentioned previously Project Tungsten is created to get the advantage of the modern hardware as much as possible. Hence, this Tungsten will help in encoding the Dataset in-memory only.

Dataset operations which does not fall in either Transformations or actions: SparkSQL dataset has some operations which does not fall in either transformations or actions like Cache, persist etc. Let's see the example below for such methods.

Various Dataset operators or functions (This are not transformation or Actions)

```
//Using as operator to convert a DataFrame (Generic Data type Dataset) to a Dataset (Strongly typed Dataset)

//Lets create a DataFrame
val heDF = spark.read.format("csv").option("header",true).option( "Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv")

//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail
case class HECourse(ID: Int, Name: String, Fee : Int, Venue: String, Date:String, Duration:Int)

//Converting to dataset
val heCourseDS = heDF.as[HECourse]

//Check the types of DS
heCourseDS

//Cache the Dataset( MEMORY_AND_DISK)
heCourseDS.cache()

//You can check, about this cached data (Ip of your Spark host)
http://192.168.239.134:4040/storage/

//It should not, yet cached
//Now call action, so calculation will happen and all the transformations will be called
//Which can result in caching the Dataset
//After that check the above web page by refreshing it
heCourseDS.count()

//check whether dataset is cached or not in Spark-shell itself
heCourseDS.queryExecution.withCachedData

//Un-persist the data
heCourseDS.unpersist()

//Now check the storage page
http://192.168.239.134:4040/storage/


//Check the execution plan of next select statement so that, you will get to know about InMemory dataset
heCourseDS.select($"ID", $"Name").explain(extended = true)
```

```
//Checkpoint Dataset, it should throw exception, as you have not set the checkpoint dir
heCourseDS.checkpoint

//Lets set the checkpoint dir (Directory will be created)
spark.sparkContext.setCheckpointDir("/home/hadoopexam/spark2/sparksql/checkpoint")
spark.sparkContext.getCheckpointDir.get
//Debug to check
println(heCourseDS.queryExecution.toRdd.toDebugString)

//Converting Dataset to DataFrame
heCourseDS.toDF().show()

//Rename the columns
heCourseDS.toDF("CourseId","CourseName","CourseFee","CourseVenue","CourseDate","CourseDurat
ion").show()

//Un-persisting the RDD
heCourseDS.unpersist
```

Not secure | 192.168.239.134:4040/storage/

Spark 2.3.1 Jobs Stages Storage Environment Executors SQL Spark shell application

Storage

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Dis
55	*(1) FileScan csv [ID#299,Name#300,Fee#301,Venue#302,Date#303,Duration#304] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<ID:int,Name:string,Fee:int,Venue:string,Date:string,Duration:int>	Memory Deserialized 1x Replicated	1	100%	4.1 KB	0.0 B

Figure 21: Cached Data in Spark Storage UI

```
scala> val heDF = spark.read.format("csv").option("header",true).option( "Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv")
heDF: org.apache.spark.sql.DataFrame = [ID: int, Name: string ... 4 more fields]

scala> case class HECourse(ID: Int, Name: String, Fee : Int, Venue: String, Date:String, Duration:Int)
defined class HECourse

scala> val heCourseDS = heDF.as[HECourse]
heCourseDS: org.apache.spark.sql.Dataset[HECourse] = [ID: int, Name: string ... 4 more fields]

scala> heCourseDS.cache()
res17: heCourseDS.type = [ID: int, Name: string ... 4 more fields]

scala> heCourseDS.count()
res18: Long = 100

scala> heCourseDS.queryExecution.withCachedData
res19: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
InMemoryRelation [ID#299, Name#300, Fee#301, Venue#302, Date#303, Duration#304], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
   +- *(1) FileScan csv [ID#299,Name#300,Fee#301,Venue#302,Date#303,Duration#304] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/hadoopexam/spark2/
sparksql/HadooExam_Training.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<ID:int,Name:string,Fee:int,Venue:string,Date:string,Duration:int>

scala> heCourseDS.unpersist()
res20: heCourseDS.type = [ID: int, Name: string ... 4 more fields]
scala> heCourseDS.checkpoint
org.apache.spark.SparkException: Checkpoint directory has not been set in the SparkContext
  at org.apache.spark.rdd.RDD.checkpoint(RDD.scala:1548)
  at org.apache.spark.sql.Dataset.checkpoint(Dataset.scala:594)
  at org.apache.spark.sql.Dataset.checkpoint(Dataset.scala:539)
  ... 53 elided

scala> spark.sparkContext.setCheckpointDir("/home/hadoopexam/spark2/sparksql/checkpoint")

scala> spark.sparkContext.getCheckpointDir.get
res24: String = file:/home/hadoopexam/spark2/sparksql/checkpoint/5de0c2f7-a8f0-45cf-82e6-ee3faec3fb4a

scala> heCourseDS.toDF().show()
+---+------------------+-----+----------+-----------+--------+
| ID|              Name|  Fee|     Venue|       Date|Duration|
+---+------------------+-----+----------+-----------+--------+
|  1|         HE Hadoop| 9000|    Mumbai|01-Aug-2018|       2|
|  2|          HE Spark| 7000|   Kolkata|04-Aug-2018|       3|
scala> heCourseDS.toDF("CourseId","CourseName","CourseFee","CourseVenue","CourseDate","CourseDuration").show()
+--------+------------------+---------+-----------+-----------+--------------+
|CourseId|        CourseName|CourseFee|CourseVenue| CourseDate|CourseDuration|
+--------+------------------+---------+-----------+-----------+--------------+
|       1|         HE Hadoop|     9000|     Mumbai|01-Aug-2018|             2|
```

As you can see in above example using **as** operator of a DataFrame we can convert it into a Dataset

```
val heCourseDS = heDF.as[HECourse]
```

Caching the Dataset will help future operations to complete much faster, because it does not have to calculate the Dataset again, and for any future calculation it will use the cached Dataset. (For caching and checkpointing, we will have separate dedicated chapter). Caching operation itself is not a transformation or action but Dataset will be cached only after you call action on Dataset. Once Dataset is cached, you can open Spark UI and can check under the storage tab whether Dataset was persisted or not. There is an API option also available to check whether Dataset is persisted or not as below.

```
heCourseDS.queryExecution.withCachedData
```

If you don’t need Dataset further, you can drop the cached Dataset using un-persist method on the Dataset.

Dataset and Encoder

If you know what is the serialization and de-serialization in any other programming language that it is the same concept with the different name. Dataset will always have encoders, which helps in serializing Java object (internally Spark runs on Scala and Scala runs on Java, so everything boils down to Java object only). Why SparkSQL created Encoders when Java serialization is already exists and even other open source serialization mechanism already exists like using Kryo engine. Reason is speed, as you knows Java serialization is inherently slow and similarly Kryo is also not as per Spark SQL expectations. Hence, SparkSQL team had developed new engine for serialization which is known as encoder. We would have entire chapter dedicated to encoders.

Hence, encoders helps in converting object from Dataset tabular format to JVM objects as well as it maintains the mapping between fields name of java object with the table column name. For many of the commonly used and primitive data types encoders are provided by the Spark SQL like Integer, Long, Double, String etc. Even if you are using Scala case classes or Java pojo and all the attributes of that are used have Encoders than entire class will also have Encoders, you don't have to define it explicitly. However, if you have your own custom data type and encoders may not be available for them than you have to develop the encoders for that. Tabular representation of Dataset is stored using Tungsten.

Encoders use the runtime code generation and create custom bytecode for serialization and de-serialization. That's is the reason they are much more efficient and faster. Dataset also get the benefits of Encoders in reducing the overall size of the serialized object and as you know if object size is reduced than sending data over the network is much more faster and take lesser network I/O as well as disk I/O. Once the data serialized using Encoders they are kept in Tungsten binary format. SparkSQL can run all the operations on that binary data, without even de-serializing it. If you have worked in Java than you have seen whenever you want to use the sterilized object for computation you have to first de-serialize it and then apply required operations (huge waste of time and disk I/O). Spark Tungsten will help here not to de-serialize the object for operation and apply all the operations on the serialized object only.

Chapter 7: DataFrameReader and DataFrameWriter

- DataFrameReader
- DataFrameWriter
- Hive Partitions and Bucketing
- Data Compression

DataFrameReader (**API Doc Link**): It is a class used to load the data in Spark from external systems like HDFS, local file system, JDBC store or supported NoSQL systems. To get the instance of DataFrameReader we have to use SparkSession object.

```
SparkSession.read()
```

DataFrameReader provides various methods to read the data from respective external systems like reading csv, jdbc, json, orc, parquet, exiting spark sql tables and text files. All the API methods of DatFrameReader return Dataset<Row> object, except textFile method which return Dataset<String> methods.

```
Dataset<Row>   csv(Dataset<String> csvDataset)
Dataset<Row>   csv(scala.collection.Seq<String> paths)
Dataset<Row>   csv(String... paths)
Dataset<Row>   csv(String path)
Dataset<Row>   jdbc(String url, String table, java.util.Properties properties)
Dataset<Row>   jdbc(String url, String table, String[] predicates, java.util.Properties
connectionProperties)
Dataset<Row>   jdbc(String url, String table, String columnName, long lowerBound, long upperBound,
int numPartitions, java.util.Properties connectionProperties)
Dataset<Row>   json(Dataset<String> jsonDataset)
Dataset<Row>   json(scala.collection.Seq<String> paths)
Dataset<Row>   json(String... paths)
Dataset<Row>   json(String path)
Dataset<Row>   load()
Dataset<Row>   load(scala.collection.Seq<String> paths)
Dataset<Row>   load(String... paths)
Dataset<Row>   load(String path)
Dataset<Row>   orc(scala.collection.Seq<String> paths)
Dataset<Row>   orc(String... paths)
Dataset<Row>   orc(String path)
Dataset<Row>   parquet(scala.collection.Seq<String> paths)
Dataset<Row>   parquet(String... paths)
Dataset<Row>   parquet(String path)
```

```
Dataset<Row>   table(String tableName)
Dataset<Row>   text(scala.collection.Seq<String> paths)
Dataset<Row>   text(String... paths)
Dataset<Row>   text(String path)
Dataset<String>  textFile(scala.collection.Seq<String> paths)
Dataset<String>  textFile(String... paths)
Dataset<String>  textFile(String path)
```

If you see above API methods to load the external data, then you will find that there are two types of functions, one which are format agnostic e.g. load(String path) method and other one is format specific like csv(String path)

If you are using format agnostic load method than you have to specify format explicitly using format method of the DataFrameReader. We will see few of the examples of both the variations in next steps.

All the methods return Dataset<Row> or Dataset<String> method, so we can use any of the following ways to do operations.

1. Scala Lambda expressions
2. Scala Dataset API
3. SQL Query language by converting Dataset in either temp or global views.

Example of DataFrameReader

```
//format agnostic load method
//Loading csv file
spark.read.format("csv").option("header",true).option("inferSchema",true).load("/home/hadoopexa
m/spark2/sparksql/HadooExam_Training.csv").show()

//Loading json file
//You can specify like json, csv, parquet, orc, text, jdbc etc.
spark.read.format("json").option("header",true).option("inferSchema",true).load("/home/hadoopexa
m/spark2/sparksql/he_data_1.json").show()

//format specific methods (Will be loaded as DataFrame)
spark.read.json("/home/hadoopexam/spark2/sparksql/he_data_1.json").show()
spark.read.json("/home/hadoopexam/spark2/sparksql/he_data_1.json").printSchema

//Read csv data
spark.read.csv("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").show()
spark.read.csv("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").printSchema

//Reading as textFile, this is the function which returns Dataset<String> object rather than
Dataset<Row>
spark.read.textFile("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").show()

//Split the text (Why do you want to use this method ? , gives an opportunity to pre-process the data
spark.read.textFile("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").map(line =>
line.split(",")).show()
```

```
scala> spark.read.format("csv").option("header",true).option("inferSchema",true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").show()
2018-08-29 17:21:01 WARN  ObjectStore:568 - Failed to get database global_temp, returning NoSuchObjectException
+---+------------------+-----+----------+-----------+--------+
| ID|              Name|  Fee|     Venue|       Date|Duration|
+---+------------------+-----+----------+-----------+--------+
|  1|         HE Hadoop| 9000|    Mumbai|01-Aug-2018|       2|
|  2|          HE Spark| 7000|   Kolkata|04-Aug-2018|       3|
scala> spark.read.format("json").option("header",true).option("inferSchema",true).load("/home/hadoopexam/spark2/sparksql/he_data_1.json").show()
+--------+----+---+------+---------+
|duration| fee| id|  name|    venue|
+--------+----+---+------+---------+
|       5|6000|  1|Hadoop|   Mumbai|
scala> spark.read.csv("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").show()
+---+------------------+-----+----------+-----------+--------+
|_c0|               _c1|  _c2|       _c3|        _c4|     _c5|
+---+------------------+-----+----------+-----------+--------+
| ID|              Name|  Fee|     Venue|       Date|Duration|
|  1|         HE Hadoop| 9000|    Mumbai|01-Aug-2018|       2|
scala> spark.read.textFile("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").show()
+--------------------+
|               value|
+--------------------+
|ID,Name,Fee,Venue...|
|1,HE Hadoop,9000,...|
|2,HE Spark,7000,K...|
scala> spark.read.textFile("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").map(line => line.split(",")).show()
+--------------------+
|               value|
+--------------------+
|[ID, Name, Fee, V...|
|[1, HE Hadoop, 90...|
|[2, HE Spark, 700...|
```

As we have seen most of the functions return Dataset<Row> object, but textFile() method return Dataset<String> , we can take the advantage of this. Suppose you are loading some text data, which is not formatted the way or have some character which you want to remove before processing e.g. each record in a text file are starting with ~, so what you can do is first load the entire file using textFile() which will return Dataset<String> and then iterating on each record, you can remove this extra field. So textFile() method give us opportunity to pre-process text data.

Assigning Schema, while reading the Data**:** In below example we are creating instance of the column using $ sign. There are multiple ways to create column instance, we will be discussing all the different ways to create column instance. We are creating explicit schema using StructType and then assigning this schema while loading csv data.

Reading DataFrame and applying schema
//Now define the schema for this data import org.apache.spark.sql.types.StructType val schema = new StructType().add($"id".long).add($"name".string).add($"fee".long.copy(nullable = false)).add($"venue".string).add($"date".string) *//Apply the schema explicitly* spark.read.schema(schema).csv("/home/hadoopexam/spark2/sparksql/HadooExam_Training_noheader.csv").show()

```
scala> import org.apache.spark.sql.types.StructType
import org.apache.spark.sql.types.StructType

scala> val schema = new StructType().add($"id".long).add($"name".string).add($"fee".long.copy(nullable = false)).add($"venue".string).add($"date".string)
schema: org.apache.spark.sql.types.StructType = StructType(StructField(id,LongType,true), StructField(name,StringType,true), StructField(fee,LongType,false), StructFie
d(venue,StringType,true), StructField(date,StringType,true))

scala> spark.read.schema(schema).csv("/home/hadoopexam/spark2/sparksql/HadooExam_Training_noheader.csv").show()
+---+------------------+-----+----------+-----------+
| id|              name|  fee|     venue|       date|
+---+------------------+-----+----------+-----------+
|  1|         HE Hadoop| 9000|    Mumbai|01-Aug-2018|
|  2|          HE Spark| 7000|   Kolkata|04-Aug-2018|
```

It is always recommended that, if you know the schema in advance than use that. Because it will improve the performance by avoiding extra scan. In case of JSON schema inference is automatically enabled from data. But if you know the schema than provide it.

While loading the csv file it assumes default data separator as ",". If data has some other separator then you have to use option called "sep" and provide value accordingly.

Handling corrupted records in csv/json file:

While reading csv file using DataFrameReader there are option available to handle the corrupted records and these options are below, which can be defined using "mode"

- **PERMISSIVE**: This is a default mode, it means whenever corrupted record is found in data puts the malformed string into a field configured by columnNameOfCorruptRecord, and sets other fields to null. If you want to hold corrupt records than you have to define option as below

```
spark.read .option("mode", "PERMISSIVE") .option("columnNameOfCorruptRecord",
"he_corrupted_records")
```

In this case it will read corrupted record and keep as part of DataFrame, you have to define "columnNameOfCorruptRecord "as part of custom schema. If a schema does not have the field, it drops corrupt records during parsing. However, in this mode please note that if number of fields are more or less than defined schema. It will not consider that record as corrupt record. If number if fields are less than remaining column will be set as null and if number of tokens are more than it will drop those extra fields.
As we have provided the option "columnNameOfCorruptRecord" , what it does it will keep the corrupted records with under the new column name "he_corrupted_records".

- **DROPMALFORMED:** In this mode corrupted record will be dropped.
- **FAILFAST:** As soon as corrupted record is found, it will throw an exception and also show the corrupted record as part of exception.

There are various options available for reading files and all the common issues are taken care. You can refer the API DOC for all available options.

Reading a text file as whole:

To read a text file you will be using below method

```
SparkSession.read.text("filepath")
SparkSession.read.option("wholetext" , True).text("filepath")
```

Method text of the DataFrameReader will read each individual line as a single Row object in a Dataset. And by providing an option with the "wholetext" you can read entire file contents as a single string.

Setting time Zone for the data:

While parsing the data you found that the data uses the time and date information and you want to assign specific timezone to that data, you can set the options on DataFrameReader.

```
Spark.read.option("timezone", "America/New_York").read("path to csv file")
```

Reading Data from JDBC data source: To create a DataFrame/DataSet from the JDBC table, you can use jdbc method of the DataFrameReader. It require connection properties and URL to JDBC table. In below example we are creating connection with the SQL server to read "course" table data in a DataFrame.

```
//Set the required values, for making JDBC connections
val sqlServerHostName = "db.hadoopexam.com"
val dbPort = 1433
val databaseName = "HE"

// Creating JDBC URL, avoid passing username and password in this string
val jdbc_url = s"jdbc:sqlserver://${sqlServerHostName}:${dbPort};database=${databaseName}"

// Java Properties object to hold other connection properties
import java.util.Properties
val conProperties = new Properties()

//Add user name and password in connection properties
conProperties.put("user", "hadoopexam")
conProperties.put("password", "hadoopexam")

//Use DataFrameReader to  make connection with the JDBC datasource, for specific table (course).
val course_data = spark.read.jdbc(jdbc_url, "courses", conProperties)

//Print the schema
course_data.printSchema

//Apply query on the data read from JDBC table
display(course_data.select("fee", "salary").groupBy("name").avg("fee"))
```

There is an overloaded "jdbc" method, which is available from DataFrameReader which allows you to create partitions from the JDBC data in Spark cluster, as below.

```
public Dataset<Row> jdbc(String url,
                   String table,
                   String columnName,
                   long lowerBound,
                   long upperBound,
                   int numPartitions,
                   java.util.Properties connectionProperties)
```

There are 4 new parameters than previous example

1. **columnName** : This should be a integer type column, based on this partitioning will be done.
2. **lowerBound** : Minimum value of the column to decide the partitioning.

3. **upperBoud**: Maximum value of the column for deciding partitioning.
4. **numPartitions** : Number of partitions you want to create. These all above three values numPartitions, upperBound and lowerBound will become where clause of your query while selecting the data from JDBC source.

Avoid creating lot of partitions, Spark cluster may be able to handle those partitions but JDBC datasource may not able to and that external database can crash.

Filtering Data at source only**:** You must not read data which you don't need, hence unnecessary network I/O can be reduced and for that you have to use an overloaded jdbc method of DataFrameReader as below.
Latest API DOC :

```
public Dataset<Row> jdbc(String url,
                    String table,
                    String[] predicates,
                    java.util.Properties connectionProperties)
```

As a third parameter you can provide an array of String which are the expression for where clause. Each expression will define a partition for the DataFrame.

Reading SparkSQL table as DataFrame: Use below method to read Spark SQL table as a DataFrame

```
public Dataset<Row> table(String tableName)
```

There are many overloaded methods to read the data from various data sources like parquet, orc etc. which are having the similar method signature, for your specific need refer the latest API Doc.

DataFrameWriter:

To read the data from external data sources we have used DataFrameReader, similar to this writing data to external data sources we can use DataFrameWriter. However, remember that DataFrameReader was created using SparkSession object, but DataFrameWriter will be created using Dataset.write() method. It's a member of Dataset and not SparkSession object.

Partitioning and bucketing**:** While writing data to disk you can decide how to organize the storage so that querying that data would be optimized. Suppose you are getting data on daily basis with high volume, and you query data based on each day to avoid un-necessary disk access you will be creating (Directory structure) partitions for each day as below.

```
-year=2018/month=01/day=01  -- This partition for 1st Jan 2018 data
-year=2018/month=01/day=02  -- This partition for 2nd Jan 2018 data
```

This is same as you do in Hive, hence partitioned written using Spark can be read by Hive as well with the same partition info. So whenever you query this data back you must use partitioned column as part of predicates in your query, so that you will get the best performance. However, if you have distinct values in tens of thousands then avoid using that column as a partitioned column, because it will create tens of thousands of small files which is not good. So to choose the partition column very carefully, in above scenario we can see that it will be creating 365 partitions for a year, which is ok if you are storing good amount of data for each day. If data stored on HDFS and you will create huge number of partitions than that would be load for NameNode, because for each file NameNode have an entry and will take more memory to store all the partitions.

If data volume is not high on daily basis than you can partition data based on month rather than day basis. You can use this partitioning scheme for any data JSON, Parquet.

Bucketing: As we have seen above partition will create folder for each partition and store the data in that folder. If you define bucketing as well than the based on bucketed column it will create a file. Let's assume we are storing courses/books/trainings from HadoopExam.com watched/visited by each user of daily basis. We will be defining subscriber_id as a bucketing column. Bucketing will create equal number of files in a partition. Suppose we have 100000 subscriber than inside the folder it may create 4 buckets/files and each file will be storing 25000 subscriber detail as below

```
-year=2018/month=01/day=01/bucket_1
-year=2018/month=01/day=01/bucket_2
-year=2018/month=01/day=01/bucket_3
-year=2018/month=01/day=01/bucket_4

-year=2018/month=01/day=02/bucket_1
-year=2018/month=01/day=02/bucket_2
-year=2018/month=01/day=02/bucket_3
-year=2018/month=01/day=02/bucket_4
```

There are lot of advantages when you do the bucketing while sorting on bucketed column, it will be performant. Suppose you are joining two tables with having bucket on the same columns than also it would be performant, because they are joined bucket by bucket. Number of buckets are defined by user and always remain constant.

So remember for partitioning you should choose the column which does not have very high distinct values e.g. in 10's of thousands and more. But in case of bucketing you should choose a column which has very high cardinality and data can be evenly distributed among the buckets. Again important point is you need to choose the columns for partitioning and bucketing based on the query you will using on this data.

API Methods (API Doc): Currently following API methods available for DataFrameWriter

```
DataFrameWriter<T>        bucketBy(int numBuckets, String colName, scala.collection.Seq<String> colNames)
DataFrameWriter<T>        bucketBy(int numBuckets, String colName, String... colNames)
void                      csv(String path)
DataFrameWriter<T>        format(String source)
```


QuickTechie

void	insertInto(String tableName)
void	jdbc(String url, String table, java.util.Properties connectionProperties)
void	json(String path)
DataFrameWriter<T>	mode(SaveMode saveMode)
DataFrameWriter<T>	mode(String saveMode)
DataFrameWriter<T>	option(String key, boolean value)
DataFrameWriter<T>	option(String key, double value)
DataFrameWriter<T>	option(String key, long value)
DataFrameWriter<T>	option(String key, String value)
DataFrameWriter<T>	options(scala.collection.Map<String,String> options)
DataFrameWriter<T>	options(java.util.Map<String,String> options)
void	orc(String path)
void	parquet(String path)
DataFrameWriter<T>	partitionBy(scala.collection.Seq<String> colNames)
DataFrameWriter<T>	partitionBy(String... colNames)
void	save()
void	save(String path)
void	saveAsTable(String tableName)
DataFrameWriter<T>	sortBy(String colName, scala.collection.Seq<String> colNames)
DataFrameWriter<T>	sortBy(String colName, String... colNames)
void	text(String path)

Other important points for DataFrameReader and DataFrameWriter:

- By default DataFrameReader assumes that input files are parquet files, if it has different format than you have to specify the format explicitly while reading data.
- If you want to change default read format than you have to change the property called "spark.sql.sources.default"

Data Compressions: Spark can read write compressed format, and default compression formats are

1. Lzo
2. Snappy
3. Gzip
4. None

You can specify the compressions as below

```
dataFrame.write.option("compression" , "None").save("HE_DATA_FILE")
```

Columns in Dataset: While working with the SparkSQL, you will find that there are various ways by which a column in a Dataset is reoffered. Let's see all the representation below

- **"*":** as ANSI SQL standard this represents all the columns in a Dataset
- **Free Column References:** These are the columns which are not yet associated with any of the Datasets.
    - **Columns using "'" (Apostrophe)**

      ```
      org.apache.spark.sql.column='columnName //this is a free column
      ```

    - Using "$" prefix

      ```
      val namedCol:Column=$"name"
      ```

- Using col function

```
val namedColumn = col("columnName")
val city = Column("city")
```

These are the all ways by which you can create column objects and until you bound them to a Dataset they are known as Free columns. Once they are attached to Dataset, it will be known as bound columns.

Exercise : Reading and Writing with compression

```
//Saving DataFrame as compressed file
spark.read.schema(schema).csv("/home/hadoopexam/spark2/sparksql/HadooExam_Training_noheader.csv").write.option("compression" ,
"gzip").save("/home/hadoopexam/spark2/sparksql/HadooExam_1")

//Now read back this compressed file
spark.read.load("/home/hadoopexam/spark2/sparksql/HadooExam_1")
```

Exercise : Reading table data using DataFrameReader

```
val jsonData= spark.read.format("json").load("/home/hadoopexam/spark2/sparksql/he_data_1.json")

//Check the types of data
jsonData

//Register as a temp table
jsonData.createOrReplaceTempView("T_HECOURSE")

//Check whether table exists or not
spark.catalog.tableExists("T_HECOURSE")
//Loading table data as DataFrame
val he_table = spark.read.table("T_HECOURSE")
```

```
scala> spark.read.schema(schema).csv("/home/hadoopexam/spark2/sparksql/HadooExam_Training_noheader.csv").write.option("compression" , "gzip").save("/home/hadoopexam/spa
rk2/sparksql/HadooExam_12")

scala> spark.read.load("/home/hadoopexam/spark2/sparksql/HadooExam_1")
res10: org.apache.spark.sql.DataFrame = [id: bigint, name: string ... 3 more fields]

scala> val jsonData= spark.read.format("json").load("/home/hadoopexam/spark2/sparksql/he_data_1.json")
jsonData: org.apache.spark.sql.DataFrame = [duration: bigint, fee: bigint ... 3 more fields]

scala> jsonData.createOrReplaceTempView("T_HECOURSE")

scala> spark.catalog.tableExists("T_HECOURSE")
res12: Boolean = true

scala> val he_table = spark.read.table("T_HECOURSE")
he_table: org.apache.spark.sql.DataFrame = [duration: bigint, fee: bigint ... 3 more fields]
```

Exercise: Save the Datasets using DataFrameWriter

```
//Default storage is parquet format
spark.read.textFile("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").write.save("/home/hadoopexam/spark2/sparksql/HadooExam_Training_1")

//Write as Json format
spark.read.textFile("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").write.format("json").save("/home/hadoopexam/spark2/sparksql/HadooExam_Training_2")

//Another way to save as json (format specific method)
spark.read.textFile("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").write.json("/home/hadoopexam/spark2/sparksql/HadooExam_Training_3")

//Save as table
spark.read.textFile("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").write.saveAsTable("T_COURSE")
sql("select * from T_COURSE").show()

//All currently available tables in the catalog
spark.catalog.listTables.show

//Saving data using partition by
spark.read.format("csv").option("header",true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").write.partitionBy("fee").save("/home/hadoopexam/spark2/sparksql/HadooExam_Training_4")

//Inserting data into existing table in catalog
//take count before inserting the data
sql("select * from T_COURSE").count()
spark.read.text("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").write.insertInto("T_COURSE")
sql("select * from T_COURSE").show()
sql("select * from T_COURSE").count()
```

```
scala> spark.read.textFile("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").write.save("/home/hadoopexam/spark2/sparksql/HadooExam_Training_1")

scala> spark.read.textFile("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").write.format("json").save("/home/hadoopexam/spark2/sparksql/HadooExam_Training_2")


scala> spark.read.textFile("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").write.json("/home/hadoopexam/spark2/sparksql/HadooExam_Training_3")
```

```
scala> sql("select * from T_COURSE").show()
+--------------------+
|               value|
+--------------------+
|ID,Name,Fee,Venue...|
|1,HE Hadoop,9000,...|
```

```
scala> spark.catalog.listTables.show
+----------+--------+-----------+---------+-----------+
|      name|database|description|tableType|isTemporary|
+----------+--------+-----------+---------+-----------+
|  t_course| default|       null|  MANAGED|      false|
|t_hecourse|    null|       null|TEMPORARY|       true|
+----------+--------+-----------+---------+-----------+


scala> spark.read.format("csv").option("header",true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").write.partitionBy("fee").save("/home/hadoopexam/sp
ark2/sparksql/HadooExam_Training_4")

scala> sql("select * from T_COURSE").count()
res21: Long = 202

scala> spark.read.text("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv").write.insertInto("T_COURSE")

scala> sql("select * from T_COURSE").show()
+--------------------+
|               value|
+--------------------+
|ID,Name,Fee,Venue...|
|1,HE Hadoop,9000,...|
```

```
scala> sql("select * from T_COURSE").count()
res24: Long = 303
```

Chapter 8: SparkSQL and Hive Support

- SparkSQL and Hive
  - Managed Tables
  - External Tables
- Choice between external and internal table
- Hive metastore
- Hive Support in SparkSQL

Spark SQL and Hive Query Support

Hive is older than Spark and previous project which was created in Spark to support Hive was known as Shark. However, Shark was only supporting Hive framework. So it was decided to not continue with the Shark and create a new project to support general purpose SQLs. And new project is Spark SQL, which is based on Catalyst optimizer.

Apache Spark and Hadoop framework complements each other on various front for example Hadoop provides distributed storage using HDFS and a matured cluster manager using YARN. And Spark is purely a computational framework which uses this distributed framework during computation and no new infrastructure needs to be created.

And most of the structured data was stored using Hive, and Hive has concept of external and managed tables. And all the information regarding these tables are stored in Hive catalog which is usually stored in another RDBMS solution like MySQL, Oracle etc.

- *Managed Tables or internal tables*:
  All the tables whose lifecycle is managed by Hive framework is known as internal or managed tables. Hive has one directory called or referred as warehouse directory. It is not necessary to have directory name to warehouse, but usually people keep the names as warehouse only for example /home/cloudera/warehouse, you will find on Cloudera platform. Hence, whatever tables are managed by Hive it will create a directory with the table name in this warehouse directory. For example, we have created a table called "TBL_HADOOPEXAM" then hive will create a directory as "/home/cloudera/warehouse/tbl_hadoopexam" and all the data will be stored in this directory. It may be stored in further nested directory, if table was created using partitions and buckets. As I mentioned internal tables are managed tables and managed by Hive. If you run the drop table DDL than table will be dropped as well as data and directory from HDFS "/home/cloudera/warehouse/tbl_hadoopexam" will be deleted.
  Internal tables have the following features.

- Entire lifecycle of the Hive internal tables are managed by Hive only
- As soon as you drop table it will drop the data as well as metadata associated with the tables from both the metastore and namenode of HDFS.
- Data will always be stored under warehouse directory.
- You can have any directory name as a warehouse directory, by setting the configuration on Hadoop using "**hive.metastore.warehouse.dir**"
- Even you can create managed table other than warehouse directory, while creating table you can provide the location where you want to create the table. But if you don't want to specify the location than Hive managed table data will always be created in warehouse directory.

- *External Tables*:
  In case of external tables data lifecycle is not managed by Hive and data should not be stored in warehouse directory. You can have your data stored in HDFS in any other directory. Even existing data you can convert into external tables without moving them into warehouse directory. When you define an external table than only metadata for the table will be created in the metastore. Following are the features of Hive external table

  - Any existing data in HDFS can be created as a Hive table without moving them into warehouse directory.
  - If you drop Hive external table than data will not be deleted and remain as it is on HDFS.
  - If you drop table than metadata of the table will be removed from metastore.

Making choice between external and internal Hive tables: You can decide whether you should go for external table or internal table for Hive.

Use External Table when:

- When data already present on HDFS and you want to use tool other than Hive, like Pig means accessing the same data from more than one tool.
- If there is a need for keeping the data even after dropping Hive table.
- You don't want to give data access control to Hive, but want to HDFS ACLs (Refer Hadoop Admin Training from http://hadoopexam.com for ACLs) or some other data access mechanism then you should use external table.
- If you want your same data should have more than one schema, than external table is the way. You can access same data with different table name and schema.

Use internal table:

- Entire lifecycle of data are managed by Hive, hence as soon as you drop the table data would be deleted from HDFS. And that is ok, for your requirement than use internal tables.
- If existing data is not well structured and the writing efficient query is not possible than you will be transforming the existing data in the format you need. And then use efficient data formats like Parquet, Avro etc. and do the partitioning and bucketing on the data. And create Hive internal table.

- If you are deleting any partitions of Hive managed table manually than table state may be in-correct and Hive would still be having stale information about that deleted partitions. So it is always recommended that you use the Hive query to alter the table and its data.

Hive Metastore:

While creating the tables you have to provide that what is the location of the data, what is name of table, what is the name of columns, types of columns, data partitioning, bucketing, ser-de (serializes and de-serializes) detail. These all information needs to be stored somewhere and that will be stored in Hive metastore. You can use RDBMS like Oracle or MySQL to store this metadata. If you don't configure than Hive will be using in-process instance of Derby (which is not good for production use cases).

Hive Support in SparkSQL

Developer of SparkSQL wanted to still continue to support HiveQL. Hence it was maintained and SparkSQL supports features of Hive like reading Hive table's metadata from metastore, Hive Query Language and Hive User defined functions etc. To have Hive support with SparkSQL, you don't have to do any changes in your Hive data and Hive metastore.

Some of the features which are not supported from Hive are below (You can find full list from this link):

- Hive tables bucketed using Hash Partitioning
- Hadoop Archival
- Hive optimization using Index. (Because SparkSQL stores data in-memory while doing computation. Hence, there is no as such need of doing indexing on the data stored in Hive)
- **Merging Small files**: If Spark SQL query generates lot of small files than it will not merge all the small files. But Hive has this optimization features. Because having lot of small files are not good for HDFS.
- Not all the UDF (User defined functions) from Hive are supported.

Hive Query support using SparkSQL:

SparkSQL supports various Hive features as well as HiveQL, and please note that it is not necessary to have Hive setup already available to run Hive Query or to use Hive features. You can use all the supported Hive features and HiveQL without even setting up Hive. Hence, this will give you the advantage if you have already worked with Hive and familiar with Hive features and wants to use them in SparkSQL.

SaprkSQL supports the queries written using HiveQL. HiveQL is very similar to SQL, but does not follow ANSI standard of SQL. Hence, you can say that HiveQL is a SQL-like language. The reason HiveQL is supported because HiveQL is quite old and more matured than SparkSQL, even it has support for more complex queries than SparkSQL. We will see one example, how to use HiveQL in SparkSQL in next section. It internally uses the HiveContext and launches the Spark Jobs to run HiveQL.

In the below example the syntax which we are seeing "FROM HEVIEW" is a Hive way of writing SQL and that is supported without even setting the Hive.

Exercise : Load Data from file, this also shows that SparkSQL supports Hive Query Language Syntax

```
//Load data from a file, and as file has header. So by providing options you can say, derive column from file header only.
sql("CREATE OR REPLACE TEMPORARY VIEW HEVIEW USING csv OPTIONS ('path'='/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv', 'header'='true')")

//Select all the data from View, using HiveQL syntax
sql("FROM HEVIEW").show

//Print schema detail for HEVIEW
sql("desc EXTENDED HEVIEW").show()
```

```
scala> sql("CREATE OR REPLACE TEMPORARY VIEW HEVIEW USING csv OPTIONS ('path'='/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv', 'header'='true')")
2018-08-29 22:29:42 WARN  ObjectStore:568 - Failed to get database global_temp, returning NoSuchObjectException
res0: org.apache.spark.sql.DataFrame = []

scala> sql("FROM HEVIEW").show
+---+------------------+-----+----------+-----------+--------+
| ID|              Name|  Fee|     Venue|       Date|Duration|
+---+------------------+-----+----------+-----------+--------+
|  1|         HE Hadoop| 9000|    Mumbai|01-Aug-2018|       2|
scala> sql("desc EXTENDED HEVIEW").show()
+--------+---------+-------+
|col_name|data_type|comment|
+--------+---------+-------+
|      ID|   string|   null|
|    Name|   string|   null|
|     Fee|   string|   null|
|   Venue|   string|   null|
|    Date|   string|   null|
|Duration|   string|   null|
+--------+---------+-------+
```

Chapter 9: SparkSQL and JSON

- Introduction
- Querying JSON data
- Approaches to read JSON data
- JSON and SQL Query

JSON data introduction

SparkSQL supports not only querying JSON data, but also supports to store the query results in JSON format. Since last few years using JSON format data has increased because of its lightweight and well fit for semi-structured data. Use of JSON format is gradually increasing like in mobile applications, web applications, and messages over the queues etc. In Spark support for JSON data was introduced since Spark 1.1

Working with JSON data in other System:

If you have ever used JSON data in your application than you would have not used them directly. Most of the time you will receive the JSON data or messages and then transform them in a format which is well fit for processing. Suppose you are working with Java programming language than you will read JSON data and then transform them into Java Beans/POJO and do the processing on this Beans whatever you wanted to do and then finally you will convert these beans back to the JSON format before saving into disk or database, this is one of the simple scenario there are many complex pipelines are created while working with the JSON data. So overall you will spend lot of time, processing and memory to transform JSON data in supported form and then converting back to the JSON format.

While doing all these processing you have to take care for serialization-deserialization of JSON data, schema manipulation etc.

Querying JSON data:

You can use both SparkSQL and Hive to query the JSON data. However, querying complex JSON data using HiveQL is not very simple. You have to write complex queries and need to use UDFs.

But Spark SQL provides following features to query the JSON data

- You can use simple query and it will not be as complex as in HiveQL (We will see example in next section)
- Automatic Schema inference (From JSON data SparkSQL can infer the schema means name and data types of the columns).
- Even querying nested fields are very simple

Read JSON data in Spark

There are multiple ways to load or work with the JSON data in Spark SQL. We will see these approaches one by one.

Approach-1: Use the DataFrameReader to format agnostic load method. To get the instance of DataFrameReader you will be using SparkSession. SparkSession is an entry point object for Spark 2.0 onwards and previously used contexts objects like SparkContext and HiveContext can be accessed using SparkSession itself as well. In Spark-shell this SparkSession object will be available as "spark" object.

To get the DataFrameReader object from SparkSession you will be using read method of the SparkSession object. See its API doc as below

```
spark.read.format("json").load("/path/to/file.json")
```

DataFrameReader is helpful in reading non-stream data and will be provided as DataFrame object. Let's see a complete example to load JSON data and using Dataset API query the JSON data.

Format agnostic method to load JSON data and then Query it

```
//Once loaded you can apply DSL on the dataset
val jsonData=
spark.read.format("json").load("/home/hadoopexam/spark2/sparksql/he_data_1.json").select("name
", "fee" , "venue").where($"fee" > 5000)
```

```
scala> val jsonData= spark.read.format("json").load("/home/hadoopexam/spark2/sparksql/he_data_1.json").select("name", "fee" , "venue").where($"fee" > 5000).show
+------+----+--------+
|  name| fee|   venue|
+------+----+--------+
|Hadoop|6000|  Mumbai|
| HBase|7000|Banglore|
+------+----+--------+
```

As load method is format agnostic, so we are providing format information explicitly using DataFrameReader format method to specify input data format. DataFrameReader in Spark was available in since Spark 1.4 version. Once you get the instance of DataFrame or Dataset object you can use API to query them as we are doing in above example.

Approache-2: Using format specific methods, since Spark 2.2 onwards you can use format specific method to read JSON data as below.

```
spark.read.json("/home/hadoopexam/spark2/sparksql/he_data_1.json").show()
```

```
scala> spark.read.json("/home/hadoopexam/spark2/sparksql/he_data_1.json").show()
+--------+----+---+------+---------+
|duration| fee| id|  name|    venue|
+--------+----+---+------+---------+
|       5|6000|  1|Hadoop|   Mumbai|
|       4|5000|  2| Spark|     Pune|
|       3|4000|  3|Python|Hyderabad|
|       3|4000|  4| Scala|  Kolkata|
|       7|7000|  5| HBase| Banglore|
+--------+----+---+------+---------+
```

In above example also we are reading, JSON data using DataFrameReader json method and return the DataFrame object. Which can be queried similarly using Dataset API as we did in previous example. In both the approach, you can load files from multiple paths as well.

```
spark.read.json("file1.json" , "file2.json").show()
```

Example of loading multiple JSON files

DataFrame API also provides a method called inputFiles() which return the array of all the files used to read the source data, as you can see in below example we are loading two JSON files and then getting the input data source file using index positions.

```
//Loading data from multiple files into Dataset
val jsonDataTwoFiles=
spark.read.format("json").load("/home/hadoopexam/spark2/sparksql/he_data_1.json","/home/hado
opexam/spark2/sparksql/he_data_2.json")

//Check the types of data
jsonDataTwoFiles

//Getting input file details, if data loaded using files
jsonDataTwoFiles.inputFiles(0)
jsonDataTwoFiles.inputFiles(1)
```

```
scala> val jsonDataTwoFiles= spark.read.format("json").load("/home/hadoopexam/spark2/sparksql/he_data_1.json","/home/hadoopexam/spark2/sparksql/he_data_2.json"
jsonDataTwoFiles: org.apache.spark.sql.DataFrame = [duration: bigint, fee: bigint ... 3 more fields]

scala> jsonDataTwoFiles.show
+--------+----+---+------+---------+
|duration| fee| id|  name|    venue|
+--------+----+---+------+---------+
|       5|6000|  1|Hadoop|   Mumbai|
|       4|5000|  2| Spark|     Pune|
|       3|4000|  3|Python|Hyderabad|
|       3|4000|  4| Scala|  Kolkata|
|       7|7000|  5| HBase| Banglore|
|       5|6000|  6|  Java|   Mumbai|
|       4|5000|  7|     C|     Pune|
|       3|4000|  8|   CPP|Hyderabad|
|       3|4000|  9|  Perl|  Kolkata|
|       7|7000| 10|   JEE| Banglore|
+--------+----+---+------+---------+


scala> jsonDataTwoFiles.inputFiles(0)
res5: String = file:///home/hadoopexam/spark2/sparksql/he_data_1.json

scala> jsonDataTwoFiles.inputFiles(1)
res6: String = file:///home/hadoopexam/spark2/sparksql/he_data_2.json
```

API Doc (Link) for inputFiles (Defined in Dataset API, since Spark 2.0) method says

public String[] inputFiles()

Returns a best-effort snapshot of the files that compose this Dataset. This method simply asks each constituent BaseRelation for its respective files and takes the union of all results. Depending on the source relations, this may not find all input files. Duplicates are removed.

Approach-3: Using JSON file directly in Query: Even we can directly use the file path in the SQL query and create temporary view from JSON file as you can see in the below example.

Load Data from file, this also shows that SparkSQL supports Hive Query Language Syntax
//Load data from a JSON file.

QuickTechie

```
sql("CREATE OR REPLACE TEMPORARY VIEW HEVIEW USING org.apache.spark.sql.json OPTIONS ('path'='/home/hadoopexam/spark2/sparksql/he_data_1.json')")

//Select all the data from View, using HiveQL syntax
sql("FROM HEVIEW").show

//Print schema detail for HEVIEW
sql("desc EXTENDED HEVIEW").show()
```

```
scala> sql("CREATE OR REPLACE TEMPORARY VIEW HEVIEW USING org.apache.spark.sql.json OPTIONS ('path'='/home/hadoopexam/spark2/sparksql/he_data_1.json')")
res7: org.apache.spark.sql.DataFrame = []

scala> sql("CREATE OR REPLACE TEMPORARY VIEW HEVIEW USING org.apache.spark.sql.json OPTIONS ('path'='/home/hadoopexam/spark2/sparksql/he_data_1.json')").show
++
||
++
++


scala> sql("FROM HEVIEW").show
+--------+----+---+------+---------+
|duration| fee| id|  name|    venue|
+--------+----+---+------+---------+
|       5|6000|  1|Hadoop|   Mumbai|
|       4|5000|  2| Spark|     Pune|
|       3|4000|  3|Python|Hyderabad|
|       3|4000|  4| Scala|  Kolkata|
|       7|7000|  5| HBase| Banglore|
+--------+----+---+------+---------+


scala> sql("desc EXTENDED HEVIEW").show()
+--------+---------+-------+
|col_name|data_type|comment|
+--------+---------+-------+
|duration|   bigint|   null|
|     fee|   bigint|   null|
|      id|   bigint|   null|
|    name|   string|   null|
|   venue|   string|   null|
+--------+---------+-------+
```

Above all approaches are common to read data from JSON file in Spark.

Explicitly assigning schema to loaded JSON Data:

As we have already learned in previous chapter that how to create or define schema in SparkSQL and assign the schema to DataFrame. In the below example we are defining schema using StructType (it is available since 1.3.0) and StructField.

DataFrameReader object has an option to assign schema explicitly and then load the data, as you can see in below. Advantage of assigning schema explicitly will improve the performance by reducing load time. Because, with this it does not have to infer the schema

Explicitly Assign Schema to JSON data

```
//Import sql types
import org.apache.spark.sql.types._

//Create Schema for the JSON data
val heschema = StructType(
  StructField("id", LongType, nullable = false) ::
  StructField("name",StringType,nullable = false) ::
  StructField("fee", DoubleType, nullable = false) ::
  StructField("venue", StringType, nullable = false) ::
  StructField("Duration", LongType, nullable = false) :: Nil)
```

```
//Use defined schema while loading the data
val jsonData=
spark.read.format("json").schema(heschema).load("/home/hadoopexam/spark2/sparksql/he_data_1.
json").select("name", "fee" , "venue").where($"fee" > 5000)

//Check the output
jsonData.show()
```

```
scala> import org.apache.spark.sql.types._
import org.apache.spark.sql.types._

scala> val heschema = StructType(
     |   StructField("id", LongType, nullable = false) ::
     |   StructField("name",StringType,nullable = false) ::
     |   StructField("fee", DoubleType, nullable = false) ::
     |   StructField("venue", StringType, nullable = false) ::
     |   StructField("Duration", LongType, nullable = false) :: Nil)
heschema: org.apache.spark.sql.types.StructType = StructType(StructField(id,LongType,false), StructField(name,StringType,false), StructField(fee,DoubleType,false), Stru
ctField(venue,StringType,false), StructField(Duration,LongType,false))

scala> val jsonData= spark.read.format("json").schema(heschema).load("/home/hadoopexam/spark2/sparksql/he_data_1.json").select("name", "fee" , "venue").where($"fee" > 5
000)
jsonData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [name: string, fee: double ... 1 more field]

scala> jsonData.show
+------+------+--------+
|  name|   fee|   venue|
+------+------+--------+
|Hadoop|6000.0|  Mumbai|
| HBase|7000.0|Banglore|
+------+------+--------+
```

It is not necessary to specify the schema entirely for all the fields in JSON file, you can provide the schema for subset of the fields in the file.

Loading JSON data and use SQL query

Approach is same is we did in previous example to load the JSON data, but here we will register DataFrame as a view and then run the SQL query on that.

```
Loading data using JSON and then use If condition in the SQL query
val jsonData= spark.read.format("json").load("/home/hadoopexam/spark2/sparksql/he_data_1.json")

//Check the types of data
jsonData

//Register as a temp table
jsonData.createOrReplaceTempView("T_HECOURSE")

//Query with if condition
sql("SELECT name, IF(fee > 6000, fee, 5000) FROM (SELECT name, fee FROM T_HECOURSE)
temp").show
```

```
scala> val jsonData= spark.read.format("json").load("/home/hadoopexam/spark2/sparksql/he_data_1.json")
jsonData: org.apache.spark.sql.DataFrame = [duration: bigint, fee: bigint ... 3 more fields]

scala> jsonData.createOrReplaceTempView("T_HECOURSE")

scala> sql("SELECT name, IF(fee > 6000, fee, 5000) FROM (SELECT name, fee FROM T_HECOURSE) temp").show
+------+------------------------------------------------------------+
|  name|(IF((fee > CAST(6000 AS BIGINT)), fee, CAST(5000 AS BIGINT)))|
+------+------------------------------------------------------------+
|Hadoop|                                                        5000|
| Spark|                                                        5000|
|Python|                                                        5000|
| Scala|                                                        5000|
| HBase|                                                        7000|
+------+------------------------------------------------------------+
```

However, if you see in the query we are using IF condition as well, which is not available in ANSI SQL, but instead of that you can use CASE… WHEN approach in ANSI SQL. Programmers feels more comfortable with this IF features in SparkSQL queries.

Infer the schema from Data

You can provide the option to infer the schema from data itself rather than specifying schema explicitly. With this approach it may slow down loading process because it will have one extra step to process the data. It requires data to be scanned for inferring the schema.

```
//Loading json file and specifying option to infer the schema from data itself
spark.read.format("json").option("inferSchema",true).load("/home/hadoopexam/spark2/sparksql/he_data_1.json").show()
```

```
scala> spark.read.format("json").option("inferSchema",true).load("/home/hadoopexam/spark2/sparksql/he_data_1.json").show()
+--------+----+---+------+---------+
|duration| fee| id|  name|    venue|
+--------+----+---+------+---------+
|       5|6000|  1|Hadoop|   Mumbai|
|       4|5000|  2| Spark|     Pune|
|       3|4000|  3|Python|Hyderabad|
|       3|4000|  4| Scala|  Kolkata|
|       7|7000|  5| HBase| Banglore|
+--------+----+---+------+---------+
```

Printing the schema: you can use below DataFrame methods to check the schema of loaded DataFrame/Dataset

```
spark.read.json("/home/hadoopexam/spark2/sparksql/he_data_1.json").printSchema
```

```
scala> spark.read.json("/home/hadoopexam/spark2/sparksql/he_data_1.json").printSchema
root
 |-- duration: long (nullable = true)
 |-- fee: long (nullable = true)
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- venue: string (nullable = true)
```

While reading the json file or Dataset [String], by default infer the schema when loaded using DataFrameReader.json() method. By default json method assume your each line is a single JSON object. If you have a multiline JSON file than you have to use multiline option while loading this JSON file. Multiline option is available only after Spark 2.2 version only.

```
spark.read.option("multiLine", true).json("/home/hadoopexam/spark2/sparksql/multiline_he_data_1.json")
```

SparkSQL using JSON data full example

Below example is not specific to JSON data, but we load the data from JSON file and create DataFrame and DataSet objects and trying various options.

Some useful methods of Datasets, which is loaded using JSON

```
//Loading data from multiple files into Dataset
val jsonDataTwoFiles=
spark.read.format("json").load("/home/hadoopexam/spark2/sparksql/he_data_1.json","/home/hadoopexam/spark2/sparksql/he_data_2.json")

//Check the types of data
jsonDataTwoFiles

//Getting input file details, if data loaded using files
jsonDataTwoFiles.inputFiles(0)
jsonDataTwoFiles.inputFiles(1)

//To check whether Dataset is local or not, it means when you run the collect and take methods, it check this dataset is available locally. Hence, no need to run the executor on worker node if return true.
jsonDataTwoFiles.isLocal

//Returns all the columns of Datasets
jsonDataTwoFiles.columns

//Columns with their datatypes
jsonDataTwoFiles.dtypes

//Schema for the Dataset
jsonDataTwoFiles.schema

//Global view v/s local view
jsonDataTwoFiles.createTempView("V_HELOCAL1")
jsonDataTwoFiles.createGlobalTempView ("V_HEGLOBAL1")

//Select data from both the views
sql("select * from V_HELOCAL1").show()

//This is available across cross sessions in an application. Once application terminated its gone
sql("select * from global_temp.V_HEGLOBAL1").show()

//Converting Dataset to DataFrame (Strongly typed to Generic types) and even you can re-name the columns
jsonDataTwoFiles.toDF("NUMBEROFDAYS","FIXEDFEE","COURSE_ID","COURSE_NAME" ,
"TRAINING_VENUE").printSchema
```

```
//Converting back from DataFrame to Dataset
case class HECourse(id: String, name: String, fee : String, venue: String, duration: String)

jsonDataTwoFiles.toDF("duration","fee","id","name" , "venue").as[HECourse]

//Find the Datatype information in case of DataFrame and Dataset
jsonDataTwoFiles.map(data => data.getClass.getName).show(false)

jsonDataTwoFiles.toDF("duration","fee","id","name" , "venue").as[HECourse].map(data =>
data.getClass.getName).show(false)
```

```
scala> val jsonDataTwoFiles= spark.read.format("json").load("/home/hadoopexam/spark2/sparksql/he_data_1.json","/home/hadoopexam/spark2/sparksql/he_data_2.json")
jsonDataTwoFiles: org.apache.spark.sql.DataFrame = [duration: bigint, fee: bigint ... 3 more fields]

scala> jsonDataTwoFiles.inputFiles(0)
res17: String = file:///home/hadoopexam/spark2/sparksql/he_data_1.json

scala> jsonDataTwoFiles.inputFiles(1)
res18: String = file:///home/hadoopexam/spark2/sparksql/he_data_2.json

scala> jsonDataTwoFiles.isLocal
res19: Boolean = false

scala> jsonDataTwoFiles.columns
res20: Array[String] = Array(duration, fee, id, name, venue)

scala> jsonDataTwoFiles.dtypes
res21: Array[(String, String)] = Array((duration,LongType), (fee,LongType), (id,LongType), (name,StringType), (venue,StringType))

scala> jsonDataTwoFiles.schema
res22: org.apache.spark.sql.types.StructType = StructType(StructField(duration,LongType,true), StructField(fee,LongType,true), StructField(id,LongType,true), StructFie
d(name,StringType,true), StructField(venue,StringType,true))

scala> jsonDataTwoFiles.createTempView("V_HELOCAL1")

scala> jsonDataTwoFiles.createGlobalTempView ("V_HEGLOBAL1")
scala> sql("select * from V_HELOCAL1").show()
+--------+----+---+------+---------+
|duration| fee| id|  name|    venue|
+--------+----+---+------+---------+
|       5|6000|  1|Hadoop|   Mumbai|
|       4|5000|  2| Spark|     Pune|
scala> sql("select * from global_temp.V_HEGLOBAL1").show()
+--------+----+---+------+---------+
|duration| fee| id|  name|    venue|
+--------+----+---+------+---------+
|       5|6000|  1|Hadoop|   Mumbai|
|       4|5000|  2| Spark|     Pune|
scala> jsonDataTwoFiles.toDF("NUMBEROFDAYS","FIXEDFEE","COURSE_ID","COURSE_NAME" , "TRAINING_VENUE").printSchema
root
 |-- NUMBEROFDAYS: long (nullable = true)
 |-- FIXEDFEE: long (nullable = true)
 |-- COURSE_ID: long (nullable = true)
 |-- COURSE_NAME: string (nullable = true)
 |-- TRAINING_VENUE: string (nullable = true)


scala> case class HECourse(id: String, name: String, fee : String, venue: String, duration: String)
defined class HECourse

scala> jsonDataTwoFiles.toDF("duration","fee","id","name" , "venue").as[HECourse]
res28: org.apache.spark.sql.Dataset[HECourse] = [duration: bigint, fee: bigint ... 3 more fields]

scala> jsonDataTwoFiles.map(data => data.getClass.getName).show(false)
+-------------------------------------------------------------+
|value                                                        |
+-------------------------------------------------------------+
|org.apache.spark.sql.catalyst.expressions.GenericRowWithSchema|
|org.apache.spark.sql.catalyst.expressions.GenericRowWithSchema|
```

Chapter 10: SparkSQL and Encoders

- Implicit Objects
- Encoders
- Creating Encoders
- Encoders Exercise

Implicit Objects: Implicit object helps in converting Scala objects into

- Dataset
- DataFrame
- Columns

Implicit also defines the Encoders for Scala primitive types like Int, Float, Double, String etc. To access all the implicit, you have to import below package

```
import  spark.implicit._
```

Other than that, this implicit package helps in

- Converting RDD to Dataset
- Converting Scala symbol '$' to SparkSQL column.

Encoders (Serialization and De-serialization):

Encoders are similar concepts of Hive Ser-De or serialization and De-serialization of Java objects in JVM. However, they are created using the performance keeping in mind and they are highly performant compare to other Hive Ser-de and Java default serialization mechanism. In SparkSQL Encoders are used for serializing and de-serializing of Rows in Datasets. Hence, every dataset has Encoders. Let's have sample code

```
case class Course(id :Long, courseName:String)
val courseEncoder = Encoders.product[Course]
```

Column in Datasets are mapped to JVM object fields using Encoders.

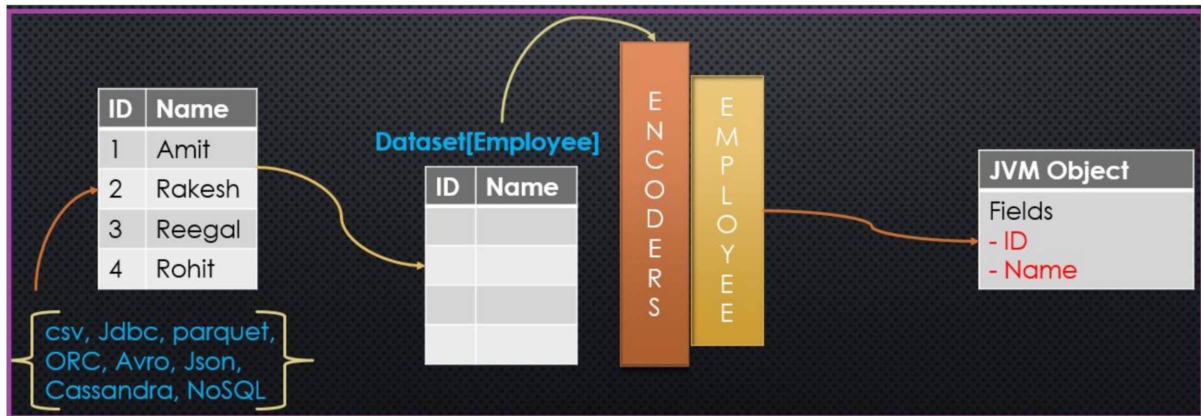


Figure 22: Encoders and JVM Object fields mapping

Hence, encoders are used to convert JVM object of Type T to and from the internal (Internal Binary format) SpaerkSQL representation. As we stated previously Dataset will always have Encoders

Dataset[T] --------(Will have) ----→ Encoders[T]

Creating Encoders: You can use following approaches to create SparkSQL encoders

1. Implicitly using SparkSession

```
import spark.implicit._
import org.apache.spark.sql.Encoders
val heDS=Seq(5000,6000,7000).toDS() //Here toDS function is using
spark.implict.newIntEncoder
```

```
scala> import spark.implicit._
<console>:1: error: identifier expected but 'implicit' found.
import spark.implicit._
             ^

scala> import org.apache.spark.sql.Encoders
import org.apache.spark.sql.Encoders

scala> val heDS=Seq(5000,6000,7000).toDS() //Here toDS function is using spark.implict.newIntEncoder
heDS: org.apache.spark.sql.Dataset[Int] = [value: int]
```

2. Explicitly:

```
Dataframe.as[HECourse] //Here, HECourse is a case class
```

```
import spark.implicit._
import org.apache.spark.sql.Encoders
Encoders.Long -> class[BigInt]
```

Hence, similarly encoders can be created for Java object types like Boolean, Integer, Long, Double, String, Timestamp etc.

For custom datatypes Encoders may not available and you have to create Encoders your own. Or you can use other sterilization as below.

- Java Serialization
- Kryo serialization engine

However, as we discussed Encoders are much faster than above. Because

1. They have schema information for each record in a Dataset.
2. Each column in a Row object is mapped with the fields in JVM object.

This extra information helps in serializing and de-serializing the primitive types very fast.

Hands on Exercise for SparkSQL Encoders:

Using the Encoders

```
//Case class representing Row object in a Dataset
case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)

//import the Encoders
import org.apache.spark.sql.Encoders

//Creating Encoder for case class
val hecourseEncoders=Encoders.product[HECourse]

//Check whether Encoder hold the schema information or not, and this schema is used to encode the
object as a Row
hecourseEncoders.schema

import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
hecourseEncoders.asInstanceOf[ExpressionEncoder[HECourse]]

//Getting serialized and de-serializer from the Encoder
hecourseEncoders.asInstanceOf[ExpressionEncoder[HECourse]].serializer
hecourseEncoders.asInstanceOf[ExpressionEncoder[HECourse]].deserializer

//Lets serialize a sample record
hecourseEncoders.asInstanceOf[ExpressionEncoder[HECourse]].toRow(HECourse(1,"Hadoop",5000,"
NewYork",5))

//Lets de-serialize the sterilized data

val herow=
hecourseEncoders.asInstanceOf[ExpressionEncoder[HECourse]].toRow(HECourse(1,"Hadoop",5000,"
NewYork",5))

import org.apache.spark.sql.catalyst.dsl.expressions._
```

```
hecourseEncoders.asInstanceOf[ExpressionEncoder[HECourse]].resolveAndBind(Seq(DslSymbol('id).int, DslSymbol('name).string, DslSymbol('fee).int,
DslSymbol('venue).string,DslSymbol('duration).int)).fromRow(herow)

//Serialize using some other serialization
Encoders.kryo[HECourse]
Encoders.javaSerialization[HECourse]
```

```
scala> import org.apache.spark.sql.Encoders
import org.apache.spark.sql.Encoders

scala> Encoders.Long -> class[BigInt]
<console>:1: error: ';' expected but 'class' found.
Encoders.Long -> class[BigInt]
                 ^

scala> case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)
defined class HECourse

scala> import org.apache.spark.sql.Encoders
import org.apache.spark.sql.Encoders

scala> val hecourseEncoders=Encoders.product[HECourse]
hecourseEncoders: org.apache.spark.sql.Encoder[HECourse] = class[id[0]: int, name[0]: string, fee[0]: int, venue[0]: string, duration[0]: int]

scala> hecourseEncoders.schema
res32: org.apache.spark.sql.types.StructType = StructType(StructField(id,IntegerType,false), StructField(name,StringType,true), StructField(fee,IntegerType,false), Stru
ctField(venue,StringType,true), StructField(duration,IntegerType,false))

scala> import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder

scala> hecourseEncoders.asInstanceOf[ExpressionEncoder[HECourse]]
res33: org.apache.spark.sql.catalyst.encoders.ExpressionEncoder[HECourse] = class[id[0]: int, name[0]: string, fee[0]: int, venue[0]: string, duration[0]: int]

scala> hecourseEncoders.asInstanceOf[ExpressionEncoder[HECourse]].serializer
res34: Seq[org.apache.spark.sql.catalyst.expressions.Expression] = List(assertnotnull(assertnotnull(input[0, HECourse, true])).id AS id#535, staticinvoke(class org.apa
he.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(assertnotnull(input[0, HECourse, true])).name, true, false) AS name#536, assertnotnull(assertno
null(input[0, HECourse, true])).fee AS fee#537, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(assertnotnull(input[
, HECourse, true])).venue, true, false) AS venue#538, assertnotnull(assertnotnull(input[0, HECourse, true])).duration AS duration#539)

scala> hecourseEncoders.asInstanceOf[ExpressionEncoder[HECourse]].deserializer
res35: org.apache.spark.sql.catalyst.expressions.Expression = newInstance(class HECourse)

scala> hecourseEncoders.asInstanceOf[ExpressionEncoder[HECourse]].toRow(HECourse(1,"Hadoop",5000,"NewYork",5))
res36: org.apache.spark.sql.catalyst.InternalRow = [0,1,3000000006,1388,3800000007,5,706f6f646148,6b726f5977654e]

scala> val herow= hecourseEncoders.asInstanceOf[ExpressionEncoder[HECourse]].toRow(HECourse(1,"Hadoop",5000,"NewYork",5))
herow: org.apache.spark.sql.catalyst.InternalRow = [0,1,3000000006,1388,3800000007,5,706f6f646148,6b726f5977654e]

scala> import org.apache.spark.sql.catalyst.dsl.expressions._
import org.apache.spark.sql.catalyst.dsl.expressions._

scala>

scala> hecourseEncoders.asInstanceOf[ExpressionEncoder[HECourse]].resolveAndBind(Seq(DslSymbol('id).int, DslSymbol('name).string, DslSymbol('fee).int, DslSymbol('venue
.string,DslSymbol('duration).int)).fromRow(herow)
res37: HECourse = HECourse(1,Hadoop,5000,NewYork,5)

scala> Encoders.kryo[HECourse]
res38: org.apache.spark.sql.Encoder[HECourse] = class[value[0]: binary]

scala> Encoders.javaSerialization[HECourse]
res39: org.apache.spark.sql.Encoder[HECourse] = class[value[0]: binary]
```

Chapter 11: Caching and Checkpointing

- Dataset and Caching
- SparkSQL and checkpointing
  - Eager Checkpointing
  - Non-Eager Checkpointing
- Local Checkpointing
- Checkpoint and performance improvements

Dataset and Caching

As you know, if we want to use the transformation output in later step of calculations, then you cache an RDD, which saves time in future steps. Similarly Dataset can be cached. But again Dataset are more efficient than RDD, Dataset will take lesser space compare to RDD to store the same amount of data why? Because Dataset already know the types of each elements/attributes and take advantage of this. So that while caching them optimally layout the Dataset and save the memory space. Even, Dataset has Encoders which helps in further reducing the space consumed by Dataset by providing detailed information of the JVM objects.

SparkSQL and Caching: We can cache the RDD in Core Spark, similarly in SparkSQL DataFrame/Dataset can be cached. Caching will give advantages only when Dataset and DataFrame are used more than once in an application. If there is no re-use of DataFrame and Dataset then it is wastage of memory. So it is always better to un-persist the Dataset, if it is not used further (Timely un-persisting is an optimization technique in SparkSQL).

```
dataset.unpersist() //un-persisting a dataset
```

Sometime you see when you try to cache a Dataset, your application may crash. Reason, what type of caching you have configured and size of Dataset. Suppose size of the Dataset is quite bigger and not enough memory is available than application will crash. And also caching parameters configured one is “MEMORY_ONLY”. Change this configuration to “MEMORY_AND_DISK”. By doing this you are able to persist bigger Dataset as well, even memory space is limited. Because with this configuration, whatever data which does not fit in memory will be saved.

Checkpointing in SparkSQL:

This is different than caching, still it helps in freezing the contents or saving the contents so that if saved contents needs to be used in future it will be highly performant. Benefits of the checkpointing are

- Logical plan will be truncated.
- It is highly beneficial for iterative programming like machine learning algorithms, where algorithms need to be executed again and again on the same data. It will also good for truncating the logical plan, because in machine learning algorithms logical plan grows almost exponentially.
- Data will be materialized and finally saved on the disk. It is always advisable that you use the file system where data loss will be avoided like HDFS.

Types of Checkpoints: There are two types of checkpoints

1. *Eager checkpointing*: In this case as soon as checkpoint is reached it will truncate the lineage and start new lineage after checkpointing.

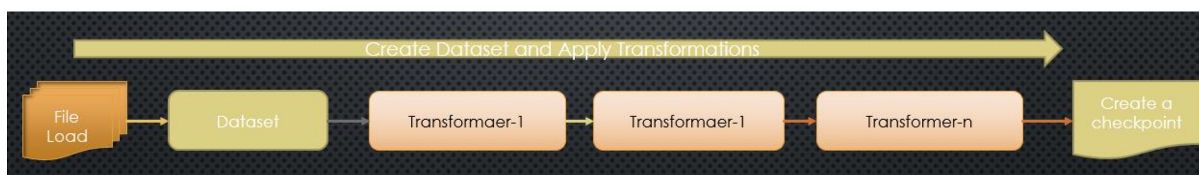


Figure 23: Creating checkpoint after transformations

  - If DataFrame/Dataset size is huge than it will take sometime to save the DataFrame over the disk. All the DataFrame which are partitioned across the nodes and saved. Because of data size, performance can be impacted when first time it is saved. Until entire data is saved to checkpoint directory no further steps will be executed.

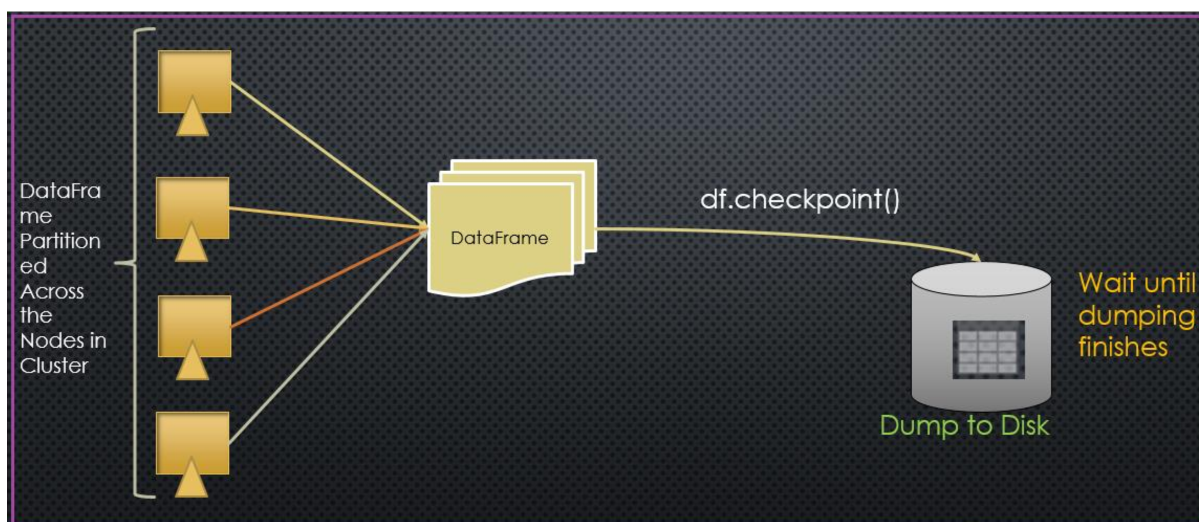


Figure 24: DataFrame checkpoint

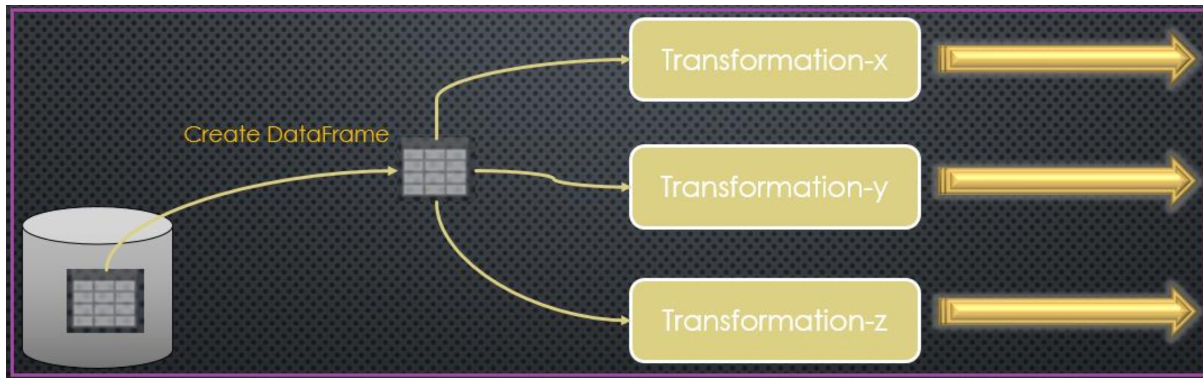


Figure 25: Create DataFrame from Check pointed Data

2. *Non-eager/lazy checkpointing*: In this case lineage will not be cut, even after creating the data checkpoint, it will still use the previous lineage.

Local checkpointing: In this case data will be saved locally on each executor locally.

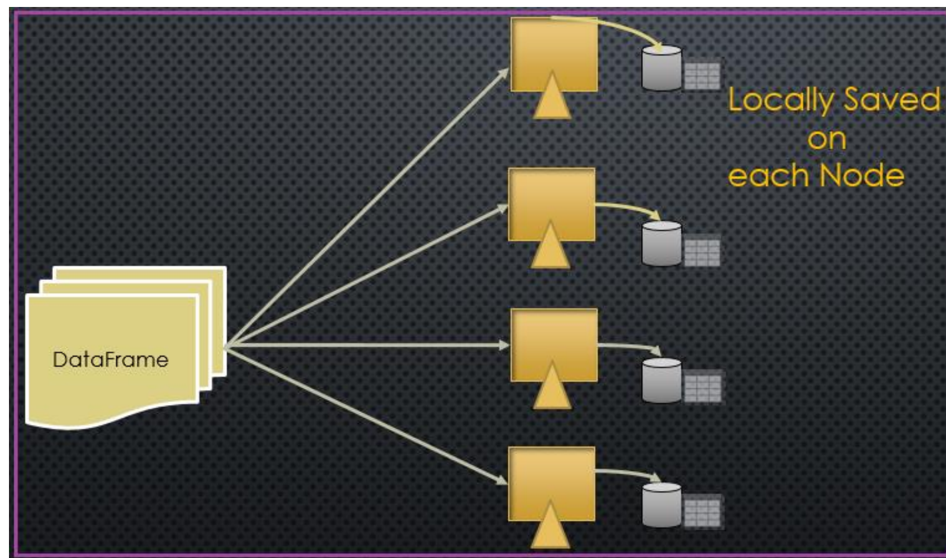


Figure 26: DataFrame Saved locally on each node

Local checkpoints are stored in the executors using caching subsystem and they are not reliable.

Caching (disk only) v/s checkpointing: What is the difference between caching (disk only) and checkpointing.

- DataFrame.persist(disk only)
- DataFrame.checkpoint(Eager only)

DataFrame.persist will serialize the data and keep the data either in cache(memory) or disk. In this case it will remember the lineage. If DataFrame is lost even from disk or memory than it can be created using lineage.

However, eager checkpoint will not store the lineage but rather cut the lineage and data will be persisted on the disk. New DataFrame will be created from the Data store in checkpoint directory. And

any new transformation after checkpoint will start a new lineage. In case any node crashes after checkpoint creation it will start lineage from the point where last checkpoint was created by loading data from checkpoint dir.

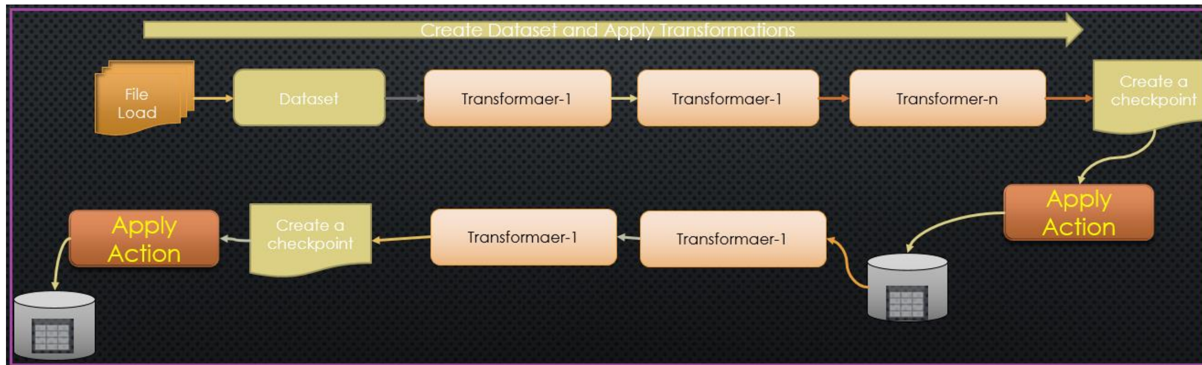


Figure 27: Lineage will cut as soon as action called

Performance Improvements:

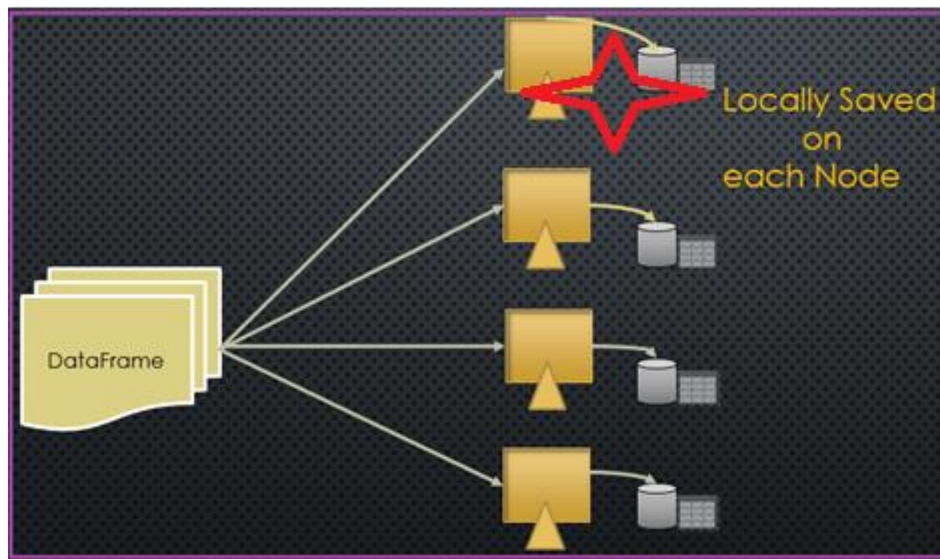


Figure 28: Node crash with the DataFrame Partition

If we don't use caching or checkpointing than Spark will have to re-compute the entire lineage in case of loss of any data on any node, as shown in above image and this will result in huge performance issue.

- **Checkpointing is more reliable**: If you are working with the larger dataset and computation is quite complex then checkpointing will be better. Because after doing complex computation data will be stored on the disk and also cut the lineage. However, checkpointing will be slower for the larger dataset.

Other important points about checkpointing:

- It is good for iterative algorithm like Machine Learning, where lineage can grow exponentially.

- Checkpointing will cut the lineage of underline RDD (Because it is a feature of RDD)
- Eager: It would be done immediately.
- Lazy: Done only when action is executed.
- Checkpoint Directory: Checkpoint will store data in a directory. Hence, it is mandatory that you have already set the checkpoint dir as below

```
SparkContext.setCheckPointDir()
```

- If checkpoint dir is not set and you call the checkpoint method on DataFrame, it will give error.

Exercise: Various Dataset operators or functions (This are not transformation or Actions)

```
//Using as operator to convert a DataFrame (Generic Data type Dataset) to a Dataset (Strongly typed Dataset)

//Lets create a DataFrame
val heDF = spark.read.format("csv").option("header",true).option( "Inferschema",
true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv")

//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail
case class HECourse(ID: Int, Name: String, Fee : Int, Venue: String, Date:String, Duration:Int)

//Converting to dataset
val heCourseDS = heDF.as[HECourse]

//Check the types of DS
heCourseDS

//Cache the Dataset( MEMORY_AND_DISK)
heCourseDS.cache()

//You can check, about this cached data
http://192.168.239.133:4040/storage/

//It should not, yet cached
//Now call action, so calculation will happen and all the transformations will be called
//Which can result in caching the Dataset
//After that check the above web page by refreshing it
heCourseDS.show()

//check whether dataset is cached or not in Spark-shell itself
heCourseDS.queryExecution.withCachedData

//Un-persist the data
heCourseDS.unpersist()

//Now check the storage page
http://192.168.239.133:4040/storage/
```


QuickTechie

```
//Check the execution plan of next select statement so taht, you will get to know about InMemory dataset
heCourseDS.select($"ID", $"Name").explain(extended = true)

//Checkpoint Dataset, it should throw exception, as you have not set the checkpoint dir
heCourseDS.checkpoint

//Lets set the checkpoint dir (Directory will be created)
spark.sparkContext.setCheckpointDir("/home/hadoopexam/spark2/sparksql/checkpoint")
spark.sparkContext.getCheckpointDir.get
//Debug to check
println(heCourseDS.queryExecution.toRdd.toDebugString)

//Converting Dataset to DataFrame
heCourseDS.toDF().show()

//Rename the columns
heCourseDS.toDF("CourseId","CourseName","CourseFee","CourseVenue","CourseDate","CourseDuration").show()

//Un-persisting the RDD
heCourseDS.unpersist
```

Chapter-12: Dataset and Joins

- Joins Introduction
- Broadcast Join
- Outer Joins
- Inner Joins
- Joins with Hints

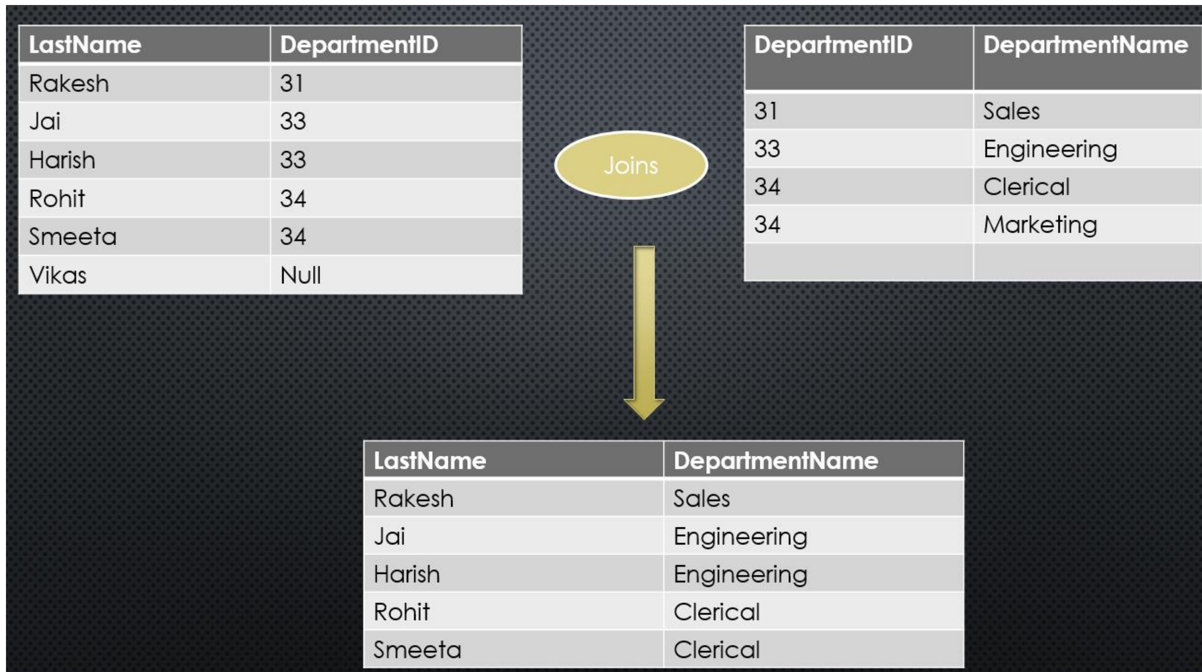
Joins Introduction:

A Join is a way to retrieve information from two or more datasets. There are various types of joins. A normal JOIN, which is also called an INNER JOIN, a LEFT OUTER JOIN, a RIGHT OUTER JOIN, a FULL OUTER JOIN and CROSS JOIN.

SQL Example of inner join

Suppose a you wanted to know what employee worked in what department. While someone could just compare the ID numbers between the two tables, a way to have the information in one place is by doing a JOIN, also known as an INNER JOIN. Because they have one type of data in common, the department ID, the tables can be joined together.

```
SELECT LastName, DepartmentName FROM employee join department on
department.DepartmentID = employee.DepartmentID;
```



LastName	DepartmentID
Rakesh	31
Jai	33
Harish	33
Rohit	34
Smeeta	34
Vikas	Null

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
34	Marketing

LastName	DepartmentName
Rakesh	Sales
Jai	Engineering
Harish	Engineering
Rohit	Clerical
Smeeta	Clerical

Figure 29: SQL Inner Join example

Outer Joins: Inner joins are fine if both tables have a matching record. However, if one table does not have a record for what the join is being built on, the query will fail. But if a database programmer needs to grab information in an event that there is not a matching record for a row on one of the tables, they need to use an outer join. Types of outer joins are

- Left
- Right
- Full outer join
- Cross Joins

We will be doing joins example using SparkSQL. However, concept for joining dataset in SparkSQL and tables in SQL Databases are same. So if you have ever done this things in RDBMS then it would be quite easy for you.

Explanation for

- *Left Join*: A left outer join (also known as a left join) will contain all records from the left dataset, even if the right dataset does not have a matching record for each row.
- *Right Join*: A right outer join works almost like a left outer join, except with how the datasets are handled reversed. This time, all of the relevant information will be returned from the right dataset, even if the left table does not have a matching result. If the left dataset does not have a matching result, null will be in the place of the missing data.
- *Full outer join*: The FULL OUTER JOIN return all records when there is a match in either left dataset or right dataset records.

- *Cross Join*: The CROSS JOIN produces a result set which is the number of rows in the first dataset multiplied by the number of rows in the second dataset if no WHERE clause is used along with CROSS JOIN. This kind of result is called as Cartesian Product.

Spark Joins Hands on Exercises:

Exercise-1 : Spark SQL Dataset Joins

//Lets create two datasets
val heDF1 = spark.read.format("csv").option("header",true).option("Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv")

//Create another Dataset
val heDF2= sc.parallelize(Seq((1, "Hadoop", 6000, "Mumbai", 5), (2, "Spark", 5000, "Pune", 4), (3, "Python", 4000, "Hyderabad", 3))).toDF("ID","Name","Fee","City","Days")

//Inner Join
heDF1.join(heDF2, "ID").show()

//Left Join
heDF1.join(heDF2, Seq("ID") , "left").show()

```
scala> val heDF1 = spark.read.format("csv").option("header",true).option( "Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv")
2018-08-31 23:42:41 WARN  ObjectStore:568 - Failed to get database global_temp, returning NoSuchObjectException
heDF1: org.apache.spark.sql.DataFrame = [ID: int, Name: string ... 4 more fields]

scala> val heDF2= sc.parallelize(Seq( (1, "Hadoop", 6000, "Mumbai", 5), (2, "Spark", 5000, "Pune", 4), (3, "Python", 4000, "Hyderabad", 3))).toDF("ID","Name","Fee","Cit
y","Days")
heDF2: org.apache.spark.sql.DataFrame = [ID: int, Name: string ... 3 more fields]

scala> heDF1.join(heDF2, "ID").show()
+---+-----------+----+---------+-----------+--------+------+----+---------+----+
| ID|       Name| Fee|    Venue|       Date|Duration|  Name| Fee|     City|Days|
+---+-----------+----+---------+-----------+--------+------+----+---------+----+
|  1|  HE Hadoop|9000|   Mumbai|01-Aug-2018|       2|Hadoop|6000|   Mumbai|   5|
|  2|   HE Spark|7000|  Kolkata|04-Aug-2018|       3| Spark|5000|     Pune|   4|
|  3|HE SparkSQL|6000|Hyderabad|07-Aug-2018|       4|Python|4000|Hyderabad|   3|
+---+-----------+----+---------+-----------+--------+------+----+---------+----+


scala> heDF1.join(heDF2, Seq("ID") , "left").show()
+---+--------------------+-----+----------+-----------+--------+------+----+------+----+
| ID|                Name|  Fee|     Venue|       Date|Duration|  Name| Fee|  City|Days|
+---+--------------------+-----+----------+-----------+--------+------+----+------+----+
| 31|             HE LDAP| 7000| Singapore|31-Aug-2018|       4|  null|null|  null|null|
| 85|           HE DotNet| 7000|    Mumbai|13-Aug-2018|       2|  null|null|  null|null|
```

//Right Join
heDF1.join(heDF2, Seq("ID") , "right").show()

//Full outer Join
heDF1.join(heDF2, Seq("ID") , "fullouter").show()

```
scala> heDF1.join(heDF2, Seq("ID") , "right").show()
+---+-----------+----+---------+-----------+--------+------+----+---------+----+
| ID|       Name| Fee|    Venue|       Date|Duration|  Name| Fee|     City|Days|
+---+-----------+----+---------+-----------+--------+------+----+---------+----+
|  1|  HE Hadoop|9000|   Mumbai|01-Aug-2018|       2|Hadoop|6000|   Mumbai|   5|
|  2|   HE Spark|7000|  Kolkata|04-Aug-2018|       3| Spark|5000|     Pune|   4|
|  3|HE SparkSQL|6000|Hyderabad|07-Aug-2018|       4|Python|4000|Hyderabad|   3|
+---+-----------+----+---------+-----------+--------+------+----+---------+----+


scala> heDF1.join(heDF2, Seq("ID") , "fullouter").show()
+---+--------------------+-----+----------+-----------+--------+------+----+------+----+
| ID|                Name|  Fee|     Venue|       Date|Duration|  Name| Fee|  City|Days|
+---+--------------------+-----+----------+-----------+--------+------+----+------+----+
| 31|             HE LDAP| 7000| Singapore|31-Aug-2018|       4|  null|null|  null|null|
| 85|           HE DotNet| 7000|    Mumbai|13-Aug-2018|       2|  null|null|  null|null|
```

//Broadcast Join using function
heDF1.join(broadcast(heDF2), "ID").show()

```
scala> heDF1.join(broadcast(heDF2), "ID").show()
+---+-----------+----+---------+-----------+--------+------+----+---------+----+
| ID|       Name| Fee|    Venue|       Date|Duration|  Name| Fee|     City|Days|
+---+-----------+----+---------+-----------+--------+------+----+---------+----+
|  1|  HE Hadoop|9000|   Mumbai|01-Aug-2018|       2|Hadoop|6000|   Mumbai|   5|
|  2|   HE Spark|7000|  Kolkata|04-Aug-2018|       3| Spark|5000|     Pune|   4|
|  3|HE SparkSQL|6000|Hyderabad|07-Aug-2018|       4|Python|4000|Hyderabad|   3|
+---+-----------+----+---------+-----------+--------+------+----+---------+----+
```

//Define a Case class for HadoopExam course detail
//Using JoinsWith operator
case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)

val heDS1 = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HECourse(4, "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7) ,HECourse(4, "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7) ,HECourse(11, "Scala", 4000, "Kolkata", 3),HECourse(12, "HBase", 7000, "Banglore", 7))).toDS()

val heDS2 = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3))).toDS()

//Now apply the joinsWith operation, it will help you to provide the required conditions

//apply inner join
val resultDS = heDS1.joinWith(heDS2 , heDS1("ID") === heDS2("ID"))
resultDS.show
resultDS.printSchema

```
scala> case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)
defined class HECourse

scala>

scala> val heDS1 = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HECour
se(4, "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7) ,HECourse(4, "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7) ,HECour
se(11, "Scala", 4000, "Kolkata", 3),HECourse(12, "HBase", 7000, "Banglore", 7))).toDS()
heDS1: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

scala>

scala> val heDS2 = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3))).toDS(
)
heDS2: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

scala>

scala> val resultDS = heDS1.joinWith(heDS2 , heDS1("ID") === heDS2("ID"))
resultDS: org.apache.spark.sql.Dataset[(HECourse, HECourse)] = [_1: struct<id: int, name: string ... 3 more fields>, _2: struct<id: int, name: string ... 3 more fields>
]

scala> resultDS.show
+--------------------+--------------------+
|                  _1|                  _2|
+--------------------+--------------------+
|[1, Hadoop, 6000,...|[1, Hadoop, 6000,...|
|[3, Python, 4000,...|[3, Python, 4000,...|
|[2, Spark, 5000, ...|[2, Spark, 5000, ...|
+--------------------+--------------------+
```

//apply Left join
val resultDS = heDS1.joinWith(heDS2 , heDS1("ID") === heDS2("ID") , "left")
resultDS.show
resultDS.printSchema

//apply right join
val resultDS = heDS1.joinWith(heDS2 , heDS1("ID") === heDS2("ID") , "right")
resultDS.show
resultDS.printSchema

```
scala> val resultDS = heDS1.joinWith(heDS2 , heDS1("ID") === heDS2("ID") , "left")
resultDS: org.apache.spark.sql.Dataset[(HECourse, HECourse)] = [_1: struct<id: int, name: string ... 3 more fields>, _2: struct<id: int, name: string ... 3 more fields>
]

scala> resultDS.show
+--------------------+--------------------+
|                  _1|                  _2|
+--------------------+--------------------+
|[12, HBase, 7000,...|                null|
|[1, Hadoop, 6000,...|[1, Hadoop, 6000,...|
|[3, Python, 4000,...|[3, Python, 4000,...|
|[5, HBase, 7000, ...|                null|
|[5, HBase, 7000, ...|                null|
|[4, Scala, 4000, ...|                null|
|[4, Scala, 4000, ...|                null|
|[11, Scala, 4000,...|                null|
|[2, Spark, 5000, ...|[2, Spark, 5000, ...|
+--------------------+--------------------+


scala> val resultDS = heDS1.joinWith(heDS2 , heDS1("ID") === heDS2("ID") , "right")
resultDS: org.apache.spark.sql.Dataset[(HECourse, HECourse)] = [_1: struct<id: int, name: string ... 3 more fields>, _2: struct<id: int, name: string ... 3 more fields>
]

scala> resultDS.show
+--------------------+--------------------+
|                  _1|                  _2|
+--------------------+--------------------+
|[1, Hadoop, 6000,...|[1, Hadoop, 6000,...|
|[3, Python, 4000,...|[3, Python, 4000,...|
|[2, Spark, 5000, ...|[2, Spark, 5000, ...|
+--------------------+--------------------+
```

Broadcast Join

In this join one of the dataset will be broadcasted to the nodes, which are holding partition of another bigger dataset. This types of join is known as replicated join because smaller dataset will be replicated on the other node in Spark Cluster.

Smaller dataset will be broadcasted. How do you find that the dataset is smaller one and needs and can be broadcasted? There is a configuration parameter which is set with the default value of 10MB. If dataset is less than 10 MB in size than it will be broadcasted. Parameter is as below.

spark.sql.autoBroadCastJoinThreshold

In broadcast join always smaller dataset should be broadcasted, so that network IO will be lesser. If you send large data over the n/w then it would be a bigger overhead. Sometime it is also known as Map-side join mostly in Hadoop world. Because joins is accomplished using just Mapper part of the Map-Reduce framework.

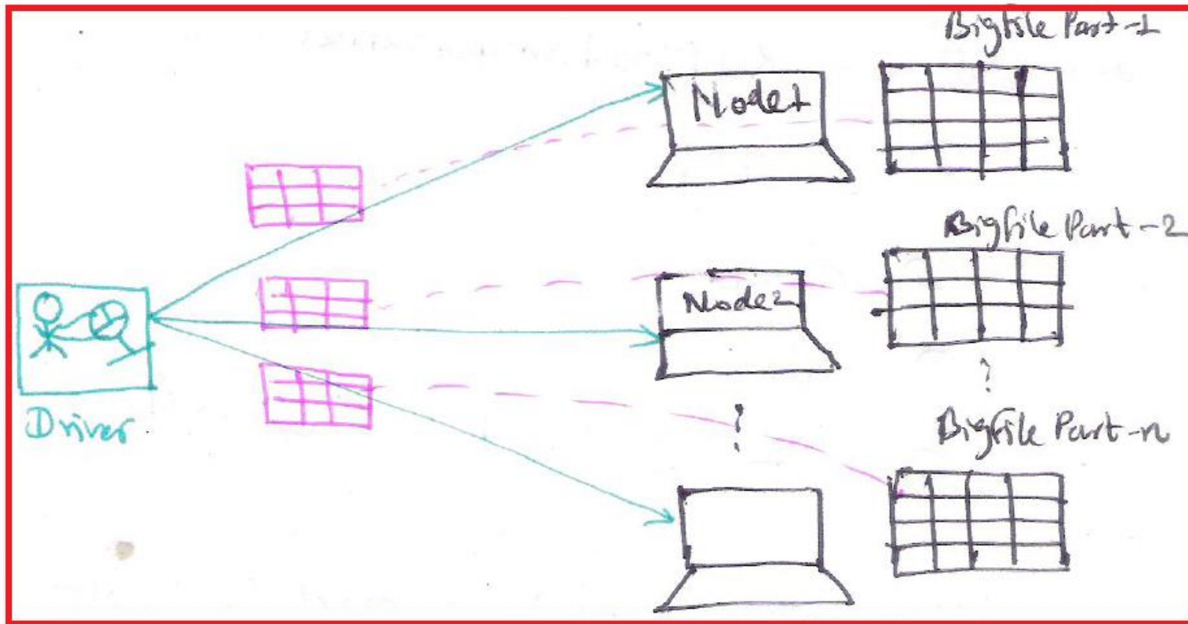


Figure 30: Broadcast Join

Driver is responsible to sending the smaller dataset over the network to each node on the cluster where bigger chunk of the data is residing and join will be locally applied on that node.

SparkSQL and Hint

While running the SQL query using Spark SQL you can provide the hint and hint will help the optimizer correct plan to execute your query. Hint are used while optimizing the logical plan. You can apply hint to query as well as dataset API

Hints are currently available only for the join operation.

Exercise-2 : Joins with hints

```
//Lets create a DataFrame
val heDF1 = spark.read.format("csv").option("header",true).option( "Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv")

//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail
case class HECourse(ID: Int, Name: String, Fee : Int, Venue: String, Date:String, Duration:Int)

//Converting to dataset
val heCourseDS = heDF1.as[HECourse]

//Lets create a DataFrame
val heDF2 = spark.read.format("csv").option("header",true).option( "Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Learners_stats.csv")

//You can even use case class to create DataFrame
//Define a Case class for HadoopExam learners stats
```

```
case class HEStats(ID: Int, LearnersCount: String, Website:String)

//Converting to dataset
val heDF2New = heDF2.toDF("ID", "LearnersCount", "Website")
val heStatsDS = heDF2New.as[HEStats]

//Do the joins and apply hint as broadcast
heCourseDS.join(heStatsDS.hint("broadcast"), "ID")

//Check whether hint is resolved or not
heCourseDS.join(heStatsDS.hint("broadcast"), "ID").queryExecution.logical

//Parameter to check current smaller file size threshold
spark.conf.get("spark.sql.autoBroadcastJoinThreshold").toInt
```

```
scala> val heDF1 = spark.read.format("csv").option("header",true).option( "Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv")
2018-08-31 23:51:11 WARN  ObjectStore:568 - Failed to get database global_temp, returning NoSuchObjectException
heDF1: org.apache.spark.sql.DataFrame = [ID: int, Name: string ... 4 more fields]

scala> case class HECourse(ID: Int, Name: String, Fee : Int, Venue: String, Date:String, Duration:Int)
defined class HECourse

scala> val heCourseDS = heDF1.as[HECourse]
heCourseDS: org.apache.spark.sql.Dataset[HECourse] = [ID: int, Name: string ... 4 more fields]

scala> val heDF2 = spark.read.format("csv").option("header",true).option( "Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Learners_stats.csv")
heDF2: org.apache.spark.sql.DataFrame = [ID: int, Learners Count: int ... 1 more field]

scala> case class HEStats(ID: Int, LearnersCount: String, Website:String)
defined class HEStats

scala> val heDF2New = heDF2.toDF("ID", "LearnersCount", "Website")
heDF2New: org.apache.spark.sql.DataFrame = [ID: int, LearnersCount: int ... 1 more field]

scala> val heStatsDS = heDF2New.as[HEStats]
heStatsDS: org.apache.spark.sql.Dataset[HEStats] = [ID: int, LearnersCount: int ... 1 more field]

scala> heCourseDS.join(heStatsDS.hint("broadcast"), "ID")
res0: org.apache.spark.sql.DataFrame = [ID: int, Name: string ... 6 more fields]

scala> heCourseDS.join(heStatsDS.hint("broadcast"), "ID").queryExecution.logical
res1: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
'Join UsingJoin(Inner,List(ID))
:- Relation[ID#10,Name#11,Fee#12,Venue#13,Date#14,Duration#15] csv
+- ResolvedHint (broadcast)
   +- Project [ID#39 AS ID#45, Learners Count#40 AS LearnersCount#46, SubscribedFrom#41 AS Website#47]
      +- Relation[ID#39,Learners Count#40,SubscribedFrom#41] csv

scala> spark.conf.get("spark.sql.autoBroadcastJoinThreshold").toInt
res2: Int = 10485760
```

Chapter-13: RelationalGroupedDataset

- Introduction to RelationalGroupedDataset
- Multi-Dimension Aggregations
- Pivot Operator
- ROLLUP, CUBE Operator

RelationalGroupedDataset: Whenever you apply group by function on Dataset it will return RelationalGroupedDataset, it is little different and you cannot directly apply action API on it. You have to apply some aggregate function on this Dataset to get the results, you cannot directly print the contents of RelationalGroupedDataset for example

```
Dataset.groupByKey(_.city).agg(typed.sum[case class](_.fee).toDF("City" , "Total Fee")
```

These are the below operators of the Dataset which return RelationalGroupedDatset. We will discuss all these operators in detail in next section.

- Group By
- Rollup
- Cube
- Pivot

Multi Dimension aggregations: There are two operators which can help you get the total, sub-total and grand total and they are known as multi-dimension operators, which are below

- Rollup
- Cube

Remember:

- You cannot print or collect RelationalGroupedDataset.
- Calling count() on grouped dataset is a transformation and not considered as an action. Hence, you have to call the collect method on the result.

Dataset Aggregation API: Aggregation API will help in working with the group of data and applying aggregations on it. In database management an aggregate function is a function where the values of multiple rows are grouped together to form a single value of more significant meaning or measurement such as a set, a bag or a list.

Group by clause is used to group the results of a SELECT query or Select API call on Dataset based on one or more columns. It is also used with SQL functions to group the result from one or more tables.

Let's check various aggregate functions, aggregate functions require the group by clause, if you want to apply aggregations on subset of rows else aggregations will be done on entire dataset.

Exercise : Various aggregations example

```
//Lets create a DataFrame
val heDF = spark.read.format("csv").option("header",true).option( "Inferschema",
true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv")

//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail
case class HECourse(ID: Int, Name: String, Fee : Int, Venue: String, Date:String, Duration:Int)

//Converting to dataset
val heCourseDS = heDF.as[HECourse]

//Register as a temp table
heCourseDS.createOrReplaceTempView("T_HECOURSE")

//Cache the table
sql("CACHE TABLE T_HECOURSE")

//Check the storage page UI and must be there as an In Memory table
http://192.168.239.133:4040/storage/

//You can also check whether table is cached or not
spark.catalog.isCached("T_HECOURSE")

sql("CACHE TABLE T_HECOURSE")

//Clear the cache
sql("CLEAR CACHE")

//Import Required Storage level
import org.apache.spark.storage.StorageLevel

//Defining storage level, by default it is MEMORY_ONLY
spark.catalog.cacheTable("T_HECOURSE", StorageLevel.MEMORY_ONLY)

//It is required, you call this to cache the table
sql("SELECT * FROM T_HECOURSE").show()

sql("CLEAR CACHE")

//Defining storage level, by default it is MEMORY_AND_DISK
spark.catalog.cacheTable("T_HECOURSE", StorageLevel.MEMORY_AND_DISK)
```

```
//It is required, you call this to cache the table
sql("SELECT * FROM T_HECOURSE").show()

//Applying aggregate using SQL query, quite easy, we should be able to do this using Dataset API
spark.sql("SELECT SUM(FEE), Venue FROM T_HECOURSE GROUP BY venue").show()

//Lets calculate entire fee collected
heCourseDS.agg(sum('fee) as "TotalFee").show()

//Now calculate the fee for each venue
heCourseDS.groupBy('venue).agg(sum('fee) as "TotalFee").show()

//Calculate avergae fee
heCourseDS.groupBy('venue).agg(avg('fee) as "TotalFee").show()

//Calculate max fee
heCourseDS.groupBy('venue).agg(max('fee) as "TotalFee").show()

//Course count for each city using groupByKey
heCourseDS.groupByKey(x => x.Venue).count().show()

//Now calculate more than one aggrgation altogether
heCourseDS.groupBy('venue).agg(sum('fee) as "TotalFee" , max('fee) , min('fee) , count('fee), avg('fee)
).show()
```

```
scala> val heDF = spark.read.format("csv").option("header",true).option( "Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv")
heDF: org.apache.spark.sql.DataFrame = [ID: int, Name: string ... 4 more fields]

scala> case class HECourse(ID: Int, Name: String, Fee : Int, Venue: String, Date:String, Duration:Int)
defined class HECourse

scala> val heCourseDS = heDF.as[HECourse]
heCourseDS: org.apache.spark.sql.Dataset[HECourse] = [ID: int, Name: string ... 4 more fields]

scala> heCourseDS.createOrReplaceTempView("T_HECOURSE")

scala> spark.sql("SELECT SUM(FEE), Venue FROM T_HECOURSE GROUP BY venue").show()
+--------+----------+
|sum(FEE)|     Venue|
+--------+----------+
|   46000| Singapore|
|   37000| Frankfurt|
```

```
scala> heCourseDS.agg(sum('fee) as "TotalFee").show()
+--------+
|TotalFee|
+--------+
|  855000|
+--------+


scala> heCourseDS.groupBy('venue).agg(sum('fee) as "TotalFee").show()
+----------+--------+
|     venue|TotalFee|
+----------+--------+
| Singapore|   46000|
| Frankfurt|   37000|
```

```
scala> heCourseDS.groupBy('venue).agg(avg('fee) as "TotalFee").show()
+----------+--------+
|     venue|TotalFee|
+----------+--------+
| Singapore|  9200.0|
| Frankfurt|  9250.0|
```

```
scala> heCourseDS.groupBy('venue).agg(max('fee) as "TotalFee").show()
+----------+--------+
|     venue|TotalFee|
+----------+--------+
| Singapore|   12000|
| Frankfurt|   12000|
```

```
scala> heCourseDS.groupByKey(x => x.Venue).count().show()
+----------+--------+
|     value|count(1)|
+----------+--------+
| Singapore|       5|
| Frankfurt|       4|
```

```
scala> heCourseDS.groupBy('venue).agg(sum('fee) as "TotalFee" , max('fee) , min('fee) , count('fee), avg('fee) ).show()
+----------+--------+--------+--------+----------+--------+
|     venue|TotalFee|max(fee)|min(fee)|count(fee)|avg(fee)|
+----------+--------+--------+--------+----------+--------+
| Singapore|   46000|   12000|    7000|         5|  9200.0|
| Frankfurt|   37000|   12000|    7000|         4|  9250.0|
|   Beijing|   50000|   12000|    7000|         5| 10000.0|
```

Another example of group by operations

Exercise : Find the price based on City as well and more on GroupBy operations

//Load the data

val heDF1 = spark.read.format("csv").option("header",true).option("Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training_double.csv")

//Select the total price for each Course and Venue

val feeForVenueAndCourse=heDF1.groupBy("Name","Venue").agg(sum("Fee") as "TotalFee").select($"Venue" , $"Name" , $"TotalFee")

```
scala> val feeForVenueAndCourse=heDF1.groupBy("Name","Venue").agg(sum("Fee") as "TotalFee").select($"Venue" , $"Name" , $"TotalFee").show
+----------+-------------------+--------+
|     Venue|               Name|TotalFee|
+----------+-------------------+--------+
|    Mumbai|          HE Hadoop|   18000|
|    Mumbai|          HE DotNet|   14000|
|  Shenzhen|          HE NetAPP|   12000|
```

//Select the total price for each Venue

val feeForVenue=heDF1.groupBy("Venue").agg(sum("Fee") as "TotalFee").select($"Venue" , lit("Total Price from this Venue") as "Name" , $"TotalFee")

```
scala> val feeForVenue=heDF1.groupBy("Venue").agg(sum("Fee") as "TotalFee" ).select($"Venue" , lit("Total Price from this Venue") as "Name" , $"TotalFee").show
+----------+--------------------+--------+
|     Venue|                Name|TotalFee|
+----------+--------------------+--------+
| Singapore|Total Price from ...|   92000|
| Frankfurt|Total Price from ...|   74000|
|   Beijing|Total Price from ...|  100000|
```

//Select the total price for each Course

val feeForCourse=heDF1.groupBy("Name").agg(sum("Fee") as "TotalFee").select(lit("Total Price from this Course") as "Venue" , $"Name" , $"TotalFee")

```
scala> val feeForCourse=heDF1.groupBy("Name").agg(sum("Fee") as "TotalFee" ).select(lit("Total Price from this Course")  as "Venue" , $"Name" , $"TotalFee").show
+--------------------+--------------------+--------+
|               Venue|                Name|TotalFee|
+--------------------+--------------------+--------+
|Total Price from ...|           HE Hadoop|   18000|
|Total Price from ...|      HE SAS BigData|   24000|
|Total Price from ...|HE SAS Administrator|    8000|
```

//Now show all the prices together using Union

val feeForVenueAndCourse=heDF1.groupBy("Name","Venue").agg(sum("Fee") as "TotalFee").select($"Venue" , $"Name" , $"TotalFee")

val feeForVenue=heDF1.groupBy("Venue").agg(sum("Fee") as "TotalFee").select($"Venue" , lit("Total Price from this Venue") as "Name" , $"TotalFee")

val feeForCourse=heDF1.groupBy("Name").agg(sum("Fee") as "TotalFee").select(lit("Total Price from this Course") as "Venue" , $"Name" , $"TotalFee")

feeForVenueAndCourse.union(feeForVenue).union(feeForCourse).sort($"Venue".asc_nulls_last).show ()

```
scala> val feeForVenueAndCourse=heDF1.groupBy("Name","Venue").agg(sum("Fee") as "TotalFee").select($"Venue" , $"Name" , $"TotalFee")
feeForVenueAndCourse: org.apache.spark.sql.DataFrame = [Venue: string, Name: string ... 1 more field]

scala> val feeForVenue=heDF1.groupBy("Venue").agg(sum("Fee") as "TotalFee" ).select($"Venue" , lit("Total Price from this Venue") as "Name" , $"TotalFee")
feeForVenue: org.apache.spark.sql.DataFrame = [Venue: string, Name: string ... 1 more field]

scala> val feeForCourse=heDF1.groupBy("Name").agg(sum("Fee") as "TotalFee" ).select(lit("Total Price from this Course")  as "Venue" , $"Name" , $"TotalFee")
feeForCourse: org.apache.spark.sql.DataFrame = [Venue: string, Name: string ... 1 more field]

scala> feeForVenueAndCourse.union(feeForVenue).union(feeForCourse).sort($"Venue".asc_nulls_last).show()
+-------+--------------------+--------+
|  Venue|                Name|TotalFee|
+-------+--------------------+--------+
|Beijing|HE Google Cloud A...|   14000|
|Beijing|Total Price from ...|  100000|
|Beijing|  HE EMC Datascience|   24000|
```

//Save the result and you can check

feeForVenueAndCourse.union(feeForVenue).union(feeForCourse).sort($"Venue".asc_nulls_last).write
.format("csv").save("/home/hadoopexam/spark2/sparksql/HadooExam_Training_GroupBy_csv")

//Let's try to do the same operation using SQL query

val heDF1 = spark.read.format("csv").option("header",true).option("Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training_double.csv")

//Register temp view

heDF1.createOrReplaceTempView("T_HEDATA")

//Import Spark SQL Function

import org.apache.spark.sql._

//Desired SQL Query
//Grouping Set is equivalent of Union of each Group by operations
//Which will provide total as well as grand total

```
val heGroupByDataSet = sql("""
  SELECT Venue, COALESCE(Name, "Total price per venue") as Name, SUM(Fee) as TotalFee
  FROM T_HEDATA
  GROUP BY Venue, Name
  GROUPING SETS ((Venue, Name), (Venue))
  ORDER BY Venue ASC NULLS LAST, Name ASC NULLS LAST
  """)

heGroupByDataSet.show()
```

```
scala> val heDF1 = spark.read.format("csv").option("header",true).option( "Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training_double.csv")
heDF1: org.apache.spark.sql.DataFrame = [ID: int, Name: string ... 4 more fields]

scala> heDF1.createOrReplaceTempView("T_HEDATA")

scala> import org.apache.spark.sql._
import org.apache.spark.sql._

scala> val heGroupByDataSet = sql("""
     |   SELECT Venue, COALESCE(Name, "Total price per venue") as Name, SUM(Fee) as TotalFee
     |   FROM T_HEDATA
     |   GROUP BY Venue, Name
     |   GROUPING SETS ((Venue, Name), (Venue))
     |   ORDER BY Venue ASC NULLS LAST, Name ASC NULLS LAST
     |   """)
heGroupByDataSet: org.apache.spark.sql.DataFrame = [Venue: string, Name: string ... 1 more field]

scala>
     | heGroupByDataSet.show()
+-------+--------------------+--------+
|  Venue|                Name|TotalFee|
+-------+--------------------+--------+
|Beijing|HE Datascience Al...|   24000|
|Beijing|  HE EMC Datascience|   24000|
|Beijing|HE Google Cloud A...|   14000|
```

//Grouping Set is equivalent of Union of each Group by operations
//Which will provide total as well as grand total

```
val heGroupByDataSet = sql("""
  SELECT Venue, COALESCE(Name, "Total price per venue") as Name, SUM(Fee) as TotalFee
  FROM T_HEDATA
  GROUP BY Venue, Name
  GROUPING SETS ((Venue, Name), (Venue),())
  ORDER BY Venue ASC NULLS LAST, Name ASC NULLS LAST
  """)

//Check the data values
heGroupByDataSet.show()
```

```
scala> val heGroupByDataSet = sql("""
     |   SELECT Venue, COALESCE(Name, "Total price per venue") as Name, SUM(Fee) as TotalFee
     |   FROM T_HEDATA
     |   GROUP BY Venue, Name
     |   GROUPING SETS ((Venue, Name), (Venue),())
     |   ORDER BY Venue ASC NULLS LAST, Name ASC NULLS LAST
     |   """)
heGroupByDataSet: org.apache.spark.sql.DataFrame = [Venue: string, Name: string ... 1 more field]

scala>
     | //Check the data values

scala> heGroupByDataSet.show()
+-------+--------------------+--------+
|  Venue|                Name|TotalFee|
+-------+--------------------+--------+
|Beijing|HE Datascience Al...|   24000|
|Beijing|  HE EMC Datascience|   24000|
|Beijing|HE Google Cloud A...|   14000|
```

```
//Save the datavalues
heGroupByDataSet.repartition(1).write.format("csv").save("/home/hadoopexam/spark2/sparksql/HadooExam_Training_groupingset")

//Grouping Set is equivalent of Union of each Group by operations
//Which will provide total as well as grand total
//Result similar to cube
//We created separate dataset for this (Check for HE Spark and total and subtotal)
val heDF1 = spark.read.format("csv").option("header",true).option( "Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training_double_1.csv")

//Register temp view
heDF1.createOrReplaceTempView("T_HEDATA")

val heGroupByDataSet = sql("""
  SELECT Venue, COALESCE(Name, "Total price per venue") as Name, SUM(Fee) as TotalFee
  FROM T_HEDATA
  GROUP BY Venue, Name
  GROUPING SETS ((Venue, Name), (Venue),(Name) , () )
  ORDER BY Venue ASC NULLS LAST, Name ASC NULLS LAST
  """)

//Check the data values
heGroupByDataSet.show()
```

```
scala> heDF1.createOrReplaceTempView("T_HEDATA")

scala>

scala> val heGroupByDataSet = sql("""
     |   SELECT Venue, COALESCE(Name, "Total price per venue") as Name, SUM(Fee) as TotalFee
     |   FROM T_HEDATA
     |   GROUP BY Venue, Name
     |   GROUPING SETS ((Venue, Name), (Venue),(Name) , () )
     |   ORDER BY Venue ASC NULLS LAST, Name ASC NULLS LAST
     |   """)
heGroupByDataSet: org.apache.spark.sql.DataFrame = [Venue: string, Name: string ... 1 more field]

scala> heGroupByDataSet.show()
+-------+-------------------+--------+
|  Venue|               Name|TotalFee|
+-------+-------------------+--------+
|Beijing|HE Datascience Al...|   24000|
|Beijing|  HE EMC Datascience|   24000|
```

//Save the datavalues
//You should get total price of all the courses
//Total price for each individual course
//Total Price for combination of Venue and Course

```
heGroupByDataSet.repartition(1).write.format("csv").save("/home/hadoopexam/spark2/sparksql/HadooExam_Training_groupingset_3")
```

Hands on Exercises for Multi-Dimensional Operator:

Rollup operator:

- Rollup operator gives 1 step granular total/sum than group by.
- It helps in getting total and sub-total of the data.

Exercise : Understand GroupBy, Rollup and Cube operations

//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail

```
case class HEEmployee(ID: Int, Name: String, gender : String, Salary: Int, Department:String)

val HEEmployeeDS = sc.parallelize(Seq( HEEmployee(1, "Deva", "Male", 5000, "Sales"), HEEmployee(2, "Jugnu", "Female", 6000, "HR"), HEEmployee(3, "Kavita", "Female", 7500, "IT"), HEEmployee(4, "Vikram", "Male", 6500, "Marketing"), HEEmployee(5, "Shabana", "Female", 5500, "Finance"), HEEmployee(6, "Shantilal", "Male", 8000, "Sales"), HEEmployee(7, "Vinod", "Male", 7200, "HR"), HEEmployee(8, "Vimla", "Female", 6600, "IT"), HEEmployee(9, "Jasmin", "Female", 5400, "Marketing"), HEEmployee(10, "Lovely", "Female", 6300, "Finance"), HEEmployee(11, "Mohan", "Male", 5700, "Sales"), HEEmployee(12, "Purvish", "Male", 7000, "HR"), HEEmployee(13, "Jinat", "Female", 7100, "IT"), HEEmployee(14, "Eva", "Female", 6800,"Marketing"), HEEmployee(15, "Jitendra", "Male", 5000, "Finance"))).toDS()

HEEmployeeDS.show()
HEEmployeeDS.printSchema()
```

//Register as a temp table

```
HEEmployeeDS.createOrReplaceTempView("T_HEEMPLOYEE")
```

//Select data from table
sql("select * from T_HEEMPLOYEE").show()

//Apply Group By
sql("SELECT department, sum(salary) as Salary_Sum   FROM T_HEEMPLOYEE GROUP BY department").show()

```
scala> sql("SELECT department, sum(salary) as Salary_Sum   FROM T_HEEMPLOYEE  GROUP BY department").show()
+----------+----------+
|department|Salary_Sum|
+----------+----------+
|     Sales|     18700|
|        HR|     20200|
|   Finance|     16800|
| Marketing|     18700|
|        IT|     21200|
+----------+----------+
```

Here you can see the sum of the salaries of all employees grouped by their department. However, we cannot see the grand total, which is the sum of the salaries of all the employees belonging to all the departments in the company.

An alternative way to look at it is to say that the GROUP BY clause did not retrieve the total sum of the salaries of all the employees in the company. You can use ROLLUP operator for the requirement.

ROLLUP operator is helpful in calculating sub-totals and grand totals for a set of columns passed to the "GROUP BY ROLLUP" clause.

Let's see how the ROLLUP clause helps us calculate the total salaries of the employees grouped by their departments and the grand total of the salaries of all the employees in the company. To do this we will work through a simple example query.

In this code, we used the ROLLUP operator to calculate the grand total of the salaries of the employees from all the departments. However, for the grand total ROLLUP will return a NULL for department. To avoid this, we have used the "Coalesce" clause. This will replace NULL with the text "All Departments" and display the department name of each department in the Department column.

sql("SELECT coalesce (department, 'All Departments') AS Department, sum(salary) as Salary_Sum FROM T_HEEMPLOYEE GROUP BY ROLLUP (department)")

```
scala> sql("SELECT coalesce (department, 'All Departments') AS Department, sum(salary) as Salary_Sum FROM T_HEEMPLOYEE GROUP BY ROLLUP (department)").show
+---------------+----------+
|     Department|Salary_Sum|
+---------------+----------+
|          Sales|     18700|
|All Departments|     95600|
|        Finance|     16800|
|             IT|     21200|
|             HR|     20200|
|      Marketing|     18700|
+---------------+----------+
```

Finding Subtotals Using ROLLUP Operator
The ROLLUP operator can also be used to calculate sub-totals for each column, based on the groupings within that column.

Let’s look at an example where we want the sum of employee salaries at a department and gender level along with a sub-total along with a grand total for all salaries of all male and female employees belonging to all departments`.

This query returns the table below. As you can see it returns the sum of the salaries of the employees of each department divided into three categories: Male, Female and All Genders. The sub-totals are the lines with “All” in them. The last line is the grand total and so has an “All” in both columns.

sql("SELECT coalesce (department, 'All Departments') AS Department, coalesce (gender,'All Genders') AS Gender, sum(salary) as Salary_Sum FROM T_HEEMPLOYEE GROUP BY ROLLUP (department, gender)").orderBy("Department").show()

```
scala> sql("SELECT coalesce (department, 'All Departments') AS Department, coalesce (gender,'All Genders') AS Gender, sum(salary) as Salary_Sum FROM T_HEEMPLOYEE GROUP
BY ROLLUP (department, gender)").orderBy("Department").show()
+---------------+-----------+----------+
|     Department|     Gender|Salary_Sum|
+---------------+-----------+----------+
|All Departments|All Genders|     95600|
|        Finance|       Male|      5000|
|        Finance|     Female|     11800|
```

sql("SELECT coalesce (department, 'All Departments') AS Department, coalesce (gender,'All Genders') AS Gender, sum(salary) as Salary_Sum FROM T_HEEMPLOYEE GROUP BY CUBE (department, gender)").orderBy("Department").show()

```
scala> sql("SELECT coalesce (department, 'All Departments') AS Department, coalesce (gender,'All Genders') AS Gender, sum(salary) as Salary_Sum FROM T_HEEMPLOYEE GROUP
BY CUBE (department, gender)").orderBy("Department").show()
+---------------+-----------+----------+
|     Department|     Gender|Salary_Sum|
+---------------+-----------+----------+
|All Departments|     Female|     51200|
|All Departments|All Genders|     95600|
```

A PIVOT query is essentially a SELECT specifying which columns you want and how to PIVOT and GROUP them. To write a pivot query, follow these steps.

Exercise : Apply Pivot operations on Dataset

```
val courseStudents = Seq(
("Hadoop", 13000,2011),
("Spark", 11000,2013),
("Cassandra", 12000,2015),
("Java", 19000,2010),
("Python", 13000,2009),
("SQL", 24000,2009),
("Scala", 34000,2013),
("Hadoop", 12000,2012),
("Spark", 12000,2014),
("Cassandra", 11000,2016),
("Hadoop", 13000,2011),
("Spark", 11000,2013),
("Cassandra", 12000,2015),
("Java", 19000,2010),
("Python", 13000,2009),
("SQL", 24000,2009),
```

("Scala", 34000,2013),
("Hadoop", 12000,2012),
("Spark", 12000,2014),
("Cassandra", 11000,2016)
).toDF("name", "students" , "year")

val pivotedData = courseStudents.groupBy("name").pivot("year").sum("students")
pivotedData.show()

```
scala> val pivotedData = courseStudents.groupBy("name").pivot("year").sum("students")
pivotedData: org.apache.spark.sql.DataFrame = [name: string, 2009: bigint ... 7 more fields]

scala> pivotedData.show()
+---------+-----+-----+-----+-----+-----+-----+-----+-----+
|     name| 2009| 2010| 2011| 2012| 2013| 2014| 2015| 2016|
+---------+-----+-----+-----+-----+-----+-----+-----+-----+
|Cassandra| null| null| null| null| null| null|24000|22000|
|    Scala| null| null| null| null|68000| null| null| null|
|      SQL|48000| null| null| null| null| null| null| null|
|   Python|26000| null| null| null| null| null| null| null|
|    Spark| null| null| null| null|22000|24000| null| null|
|     Java| null|38000| null| null| null| null| null| null|
|   Hadoop| null| null|26000|24000| null| null| null| null|
+---------+-----+-----+-----+-----+-----+-----+-----+-----+
```

//Pivot based on specific year
val pivotedData = courseStudents.groupBy("name").pivot("year" , Seq("2009" , "2010" , "2012" , "2013")).sum("students")
pivotedData.show()

```
scala> val pivotedData = courseStudents.groupBy("name").pivot("year" , Seq("2009" , "2010" , "2012" , "2013")).sum("students")
pivotedData: org.apache.spark.sql.DataFrame = [name: string, 2009: bigint ... 3 more fields]

scala> pivotedData.show()
+---------+-----+-----+-----+-----+
|     name| 2009| 2010| 2012| 2013|
+---------+-----+-----+-----+-----+
|Cassandra| null| null| null| null|
|    Scala| null| null| null|68000|
|      SQL|48000| null| null| null|
|   Python|26000| null| null| null|
|    Spark| null| null| null|22000|
|     Java| null|38000| null| null|
|   Hadoop| null| null|24000| null|
+---------+-----+-----+-----+-----+
```

//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail
case class HEEmployee(ID: Int, Name: String, gender : String, Salary: Int, Department:String)

val HEEmployeeDS = sc.parallelize(Seq(HEEmployee(1, "Deva", "Male", 5000, "Sales"), HEEmployee(2, "Jugnu", "Female", 6000, "HR"), HEEmployee(3, "Kavita", "Female", 7500, "IT"), HEEmployee(4, "Vikram", "Male", 6500, "Marketing"), HEEmployee(5, "Shabana", "Female", 5500, "Finance"), HEEmployee(6, "Shantilal", "Male", 8000, "Sales"), HEEmployee(7, "Vinod", "Male", 7200, "HR"), HEEmployee(8, "Vimla", "Female", 6600, "IT"), HEEmployee(9, "Jasmin", "Female", 5400, "Marketing"), HEEmployee(10, "Lovely", "Female", 6300, "Finance"), HEEmployee(11, "Mohan", "Male", 5700, "Sales"), HEEmployee(12, "Purvish", "Male", 7000, "HR"), HEEmployee(13, "Jinat", "Female", 7100, "IT"), HEEmployee(14, "Eva", "Female", 6800,"Marketing"), HEEmployee(15, "Jitendra", "Male", 5000, "Finance")
, HEEmployee(15, "Rajkumar", "Male", 4500, "Finance")
, HEEmployee(15, "Satish", "Male", 4500, "Finance")
, HEEmployee(15, "Himmat", "Male", 3500, "Finance"))).toDS()

```
//Pivot data based on department
val pivotedData = HEEmployeeDS.groupBy("gender").pivot("Department").sum("Salary")
pivotedData.show()
```

```
scala> val pivotedData = HEEmployeeDS.groupBy("gender").pivot("Department").sum("Salary")
pivotedData: org.apache.spark.sql.DataFrame = [gender: string, Finance: bigint ... 4 more fields]

scala> pivotedData.show()
+------+-------+-----+-----+---------+-----+
|gender|Finance|   HR|   IT|Marketing|Sales|
+------+-------+-----+-----+---------+-----+
|Female|  11800| 6000|21200|    12200| null|
|  Male|  17500|14200| null|     6500|18700|
+------+-------+-----+-----+---------+-----+
```

The GROUP BY clause is used to group the results of aggregate functions according to a specified column. However, the GROUP BY clause doesn't perform aggregate operations on multiple levels of a hierarchy. For example, you can calculate the total of all hadoopexam employee salaries for each department (one level of hierarchy) but you cannot calculate the total salary of all employees regardless of the department they work in (two levels of hierarchy).

ROLLUP operators let you extend the functionality of GROUP BY clauses by calculating subtotals and grand totals for a set of columns. The CUBE operator is similar in functionality to the ROLLUP operator; however, the CUBE operator can calculate subtotals and grand totals for all permutations of the columns specified in it.

Exercise: Rollup, Cube using Dataset API

```
//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail
case class HEEmployee(ID: Int, Name: String, gender : String, Salary: Int, Department:String)

//Create Sample Dataset
val HEEmployeeDS = sc.parallelize(Seq( HEEmployee(1, "Deva", "Male", 5000, "Sales"), HEEmployee(2, "Jugnu", "Female", 6000, "HR"), HEEmployee(3, "Kavita", "Female", 7500, "IT"), HEEmployee(4, "Vikram", "Male", 6500, "Marketing"), HEEmployee(5, "Shabana", "Female", 5500, "Finance"), HEEmployee(6, "Shantilal", "Male", 8000, "Sales"), HEEmployee(7, "Vinod", "Male", 7200, "HR"), HEEmployee(8, "Vimla", "Female", 6600, "IT"), HEEmployee(9, "Jasmin", "Female", 5400, "Marketing"), HEEmployee(10, "Lovely", "Female", 6300, "Finance"), HEEmployee(11, "Mohan", "Male", 5700, "Sales"), HEEmployee(12, "Purvish", "Male", 7000, "HR"), HEEmployee(13, "Jinat", "Female", 7100, "IT"), HEEmployee(14, "Eva", "Female", 6800,"Marketing"), HEEmployee(15, "Jitendra", "Male", 5000, "Finance"))).toDS()

//Apply Rollup, so that we can do the total and subtotal
HEEmployeeDS.rollup($"department").agg(sum("Salary")).sort($"department".asc_nulls_last).show()

//Apply Rollup, also change the value if it is null, in the case of total
HEEmployeeDS.rollup($"department").agg(sum("Salary") as "Salary").select( coalesce($"department", lit("X_SalaryAllDept")) as "department", $"Salary").sort($"department".asc_nulls_last).show()
```

```
scala> HEEmployeeDS.rollup($"department").agg(sum("Salary")).sort($"department".asc_nulls_last).show()
+----------+-----------+
|department|sum(Salary)|
+----------+-----------+
|   Finance|      16800|
|        HR|      20200|
|        IT|      21200|
| Marketing|      18700|
|     Sales|      18700|
|      null|      95600|
+----------+-----------+


scala> HEEmployeeDS.rollup($"department").agg(sum("Salary") as "Salary").select( coalesce($"department", lit("X_SalaryAllDept")) as "department", $"Salary").sort($"depa
rtment".asc_nulls_last).show()
+---------------+------+
|     department|Salary|
+---------------+------+
|        Finance| 16800|
|             HR| 20200|
|             IT| 21200|
|      Marketing| 18700|
|          Sales| 18700|
|X_SalaryAllDept| 95600|
```

//Appy Rollup using Multiple columns and get Total Salary for each dept as well

HEEmployeeDS.rollup($"department", $"Gender").agg(sum("Salary") as "Salary").select(
$"department",coalesce($"Gender", lit("X_TotalSalaryForDept")) as
"Gender",$"Salary").sort($"department".asc_nulls_last, $"Gender".asc_nulls_last).show()

```
scala> HEEmployeeDS.rollup($"department", $"Gender").agg(sum("Salary") as "Salary").select( $"department",coalesce($"Gender", lit("X_TotalSalaryForDept")) as "Gender",$
"Salary").sort($"department".asc_nulls_last, $"Gender".asc_nulls_last).show()
+----------+--------------------+------+
|department|              Gender|Salary|
+----------+--------------------+------+
|   Finance|              Female| 11800|
|   Finance|                Male|  5000|
```

//Apply cube operation and get one more level of subotal e.g. Gender

HEEmployeeDS.cube($"department", $"Gender").agg(sum("Salary") as "Salary").select(
coalesce($"department", lit("X_GenderSalaryTotal")) as "department"
,coalesce($"Gender", lit("X_TotalSalaryForDept")) as "Gender"
,$"Salary").sort($"department".asc_nulls_last, $"Gender".asc_nulls_last).show()

```
scala> HEEmployeeDS.cube($"department", $"Gender").agg(sum("Salary") as "Salary").select( coalesce($"department", lit("X_GenderSalaryTotal")) as "department"
     | ,coalesce($"Gender", lit("X_TotalSalaryForDept")) as "Gender"
     | ,$"Salary").sort($"department".asc_nulls_last, $"Gender".asc_nulls_last).show()
+-------------------+--------------------+------+
|         department|              Gender|Salary|
+-------------------+--------------------+------+
|            Finance|              Female| 11800|
|            Finance|                Male|  5000|
```

Chapter-14: SparkSQL Functions

- **Introduction to SparkSQL Functions**
  - Standard or User Defined functions
  - Aggregate functions
  - Collection Functions
  - Date and Time Functions
  - Window aggregate functions
  - Math Functions
  - Non-aggregate functions
  - Sorting functions
  - String functions

Spark SQL Functions: SparkSQl has various functions which can be divided in few sections as below. We will be going through each one with the example as well as some in detail.

1. Standard or User Defined functions
2. Aggregate functions
3. Collection Functions
4. Date and Time Functions
5. Window aggregate functions
6. Math Functions
7. Non-aggregate functions
8. Sorting functions
9. String functions

To find the DataFrame functions, falls under which category. Please refer this API link.

Standard or User Defined Functions: These types of functions will be applied on each row of Dataset/DataFrame and also generate a single value.

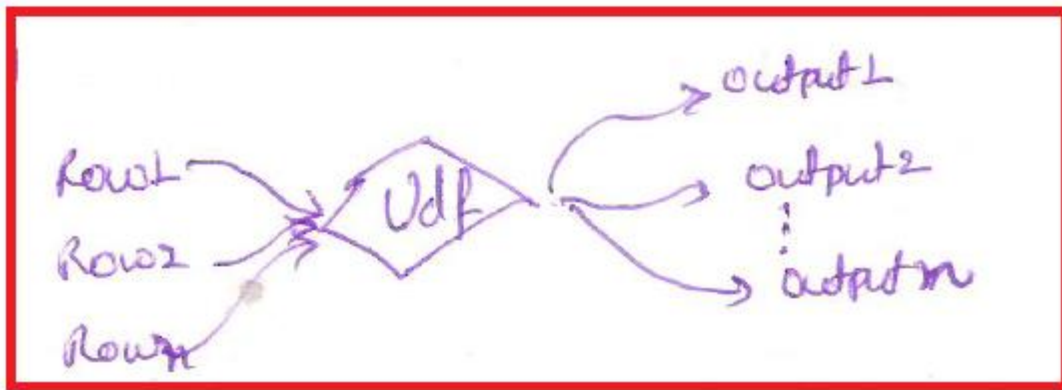


Figure 31: Standard Udf function input and output

UDF: User Defined Functions**:** You can define some functions for your custom requirement, in case functionality or function is not available in SparkSQL library.

Remember: It is possible that Catalyst may not be able to optimize UDF of your custom requirement. Hence, create UDF only when there is an absolute need.

You can use UDF for both

- Dataset API
- SparkSQL queries

Following are the ways by which you can define UDF functions

1. Inline UDF creation: Below is an example of defining inline UDF functions

```
val heCalculateTotalSalary = UDF( _ => _ +20%)
```

   In above case function defined is an anonymous function using Scala Lambda features.

2. Explicitly creating Scala function: Below is the pseudo code for creating Scala function and then we can use this function as UDF

```
val calculateTotalSalary = udf( _ => {
                                        -----------------
                                        -----------------
                                        -----------------
                          }
```

You can use these functions with the Dataset API, without doing any other step. But if you want to use them in a SQL query than you have to register this function using below syntax.

```
spark.udf.register("provideRefrenceName" , "functionNameWhichWasDefined")
spark.udf.register("totalSal" , "calculateTotalSalary")
```

Exercise for User Defined Function and User Defined Aggregate Functions:

Exercise : User Defined Functions

```
//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail
case class HEEmployee(ID: Int, Name: String, gender : String, Salary: Int, Department:String)

val HEEmployeeDS = sc.parallelize(Seq( HEEmployee(1, "Deva", "Male", 5000, "Sales"), HEEmployee(2, "Jugnu", "Female", 6000, "HR"), HEEmployee(3, "Kavita", "Female", 7500, "IT"), HEEmployee(4, "Vikram", "Male", 6500, "Marketing"), HEEmployee(5, "Shabana", "Female", 5500, "Finance"), HEEmployee(6, "Shantilal", "Male", 8000, "Sales"), HEEmployee(7, "Vinod", "Male", 7200, "HR"), HEEmployee(8, "Vimla", "Female", 6600, "IT"), HEEmployee(9, "Jasmin", "Female", 5400, "Marketing"), HEEmployee(10, "Lovely", "Female", 6300, "Finance"), HEEmployee(11, "Mohan", "Male", 5700, "Sales"), HEEmployee(12, "Purvish", "Male", 7000, "HR"), HEEmployee(13, "Jinat", "Female", 7100, "IT"), HEEmployee(14, "Eva", "Female", 6800,"Marketing"), HEEmployee(15, "Jitendra", "Male", 5000, "Finance")
, HEEmployee(15, "Rajkumar", "Male", 4500, "Finance")
, HEEmployee(15, "Satish", "Male", 4500, "Finance")
, HEEmployee(15, "Himmat", "Male", 3500, "Finance"))).toDS()

//Define UDF function, which add 20% bonus to salary
//However, remember for SparkSQL, it is difficult to optimize UDF functions.
//Hence, you should use as less as possible.
val calculateTotalSalary = (sal : Int)  => {sal * 20 /100} + sal

//Now define this function as an udf
//using UDF, you get custom transformation methods.
val calTotalSalaryWithBonus = udf(calculateTotalSalary)

//Use this UDF function to get total salary
HEEmployeeDS.withColumn("TotalSalary", calTotalSalaryWithBonus($"Salary")).show
```

```
scala> val HEEmployeeDS = sc.parallelize(Seq( HEEmployee(1, "Deva", "Male", 5000, "Sales"), HEEmployee(2, "Jugnu", "Female", 6000, "HR"), HEEmployee(3, "Kavita", "Femal
e", 7500, "IT"), HEEmployee(4, "Vikram", "Male", 6500, "Marketing"), HEEmployee(5, "Shabana", "Female", 5500, "Finance"), HEEmployee(6, "Shantilal", "Male", 8000, "Sale
s"), HEEmployee(7, "Vinod", "Male", 7200, "HR"), HEEmployee(8, "Vimla", "Female", 6600, "IT"), HEEmployee(9, "Jasmin", "Female", 5400, "Marketing"), HEEmployee(10, "Lov
ely", "Female", 6300, "Finance"), HEEmployee(11, "Mohan", "Male", 5700, "Sales"), HEEmployee(12, "Purvish", "Male", 7000, "HR"), HEEmployee(13, "Jinat", "Female", 7100,
 "IT"), HEEmployee(14, "Eva", "Female", 6800,"Marketing"), HEEmployee(15, "Jitendra", "Male", 5000, "Finance")
     | , HEEmployee(15, "Rajkumar", "Male", 4500, "Finance")
     | , HEEmployee(15, "Satish", "Male", 4500, "Finance")
     | , HEEmployee(15, "Himmat", "Male", 3500, "Finance"))).toDS()
HEEmployeeDS: org.apache.spark.sql.Dataset[HEEmployee] = [ID: int, Name: string ... 3 more fields]

scala> val calculateTotalSalary = (sal : Int)  => {sal * 20 /100} + sal
calculateTotalSalary: Int => Int = <function1>

scala> val calTotalSalaryWithBonus = udf(calculateTotalSalary)
calTotalSalaryWithBonus: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction(<function1>,IntegerType,Some(List(IntegerType)))

scala> HEEmployeeDS.withColumn("TotalSalary", calTotalSalaryWithBonus($"Salary")).show
+---+--------+------+------+----------+-----------+
| ID|    Name|gender|Salary|Department|TotalSalary|
+---+--------+------+------+----------+-----------+
|  1|    Deva|  Male|  5000|     Sales|       6000|
|  2|   Jugnu|Female|  6000|        HR|       7200|
```

```
//Using Lambda function directly for same thing, and create udf with succinct code
import org.apache.spark.sql.types._
val calTotalSalaryWithBonus = udf((sal : Int)  => {sal * 20 /100} + sal, IntegerType)
//Use the UDF in dataset
HEEmployeeDS.withColumn("TotalSalary",calTotalSalaryWithBonus('Salary) ).show
```

```
scala> import org.apache.spark.sql.types._
import org.apache.spark.sql.types._

scala> val calTotalSalaryWithBonus = udf((sal : Int)  => {sal * 20 /100} + sal, IntegerType)
calTotalSalaryWithBonus: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction(<function1>,IntegerType,None)

scala> HEEmployeeDS.withColumn("TotalSalary",calTotalSalaryWithBonus('Salary) ).show
+---+---------+------+------+----------+-----------+
| ID|     Name|gender|Salary|Department|TotalSalary|
+---+---------+------+------+----------+-----------+
|  1|     Deva|  Male|  5000|     Sales|       6000|
|  2|    Jugnu|Female|  6000|        HR|       7200|
```

```
//Register the UDF function
spark.udf.register("TotalSalaryAndBonus", calTotalSalaryWithBonus)

//Check your function is registered or not
spark.catalog.listFunctions.filter('name like "%Bonus%").show(false)

//Now use this function in dataset
HEEmployeeDS.withColumn("TotalSalary",calTotalSalaryWithBonus('Salary) ).show

//Lets make code more SQL friendly as it is SparkSQL
//Register Dataset as temporary view and will be added in Catalog
HEEmployeeDS.createOrReplaceTempView("T_HECOURSE")

//Use the registered UDF
sql("select ID, Name, gender, Salary, Department, TotalSalaryAndBonus(Salary) as TotalSalary from
T_HECOURSE" ).show()
```

```
scala> spark.udf.register("TotalSalaryAndBonus", calTotalSalaryWithBonus)
res31: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction(<function1>,IntegerType,None)

scala> spark.catalog.listFunctions.filter('name like "%Bonus%").show(false)
+-------------------+--------+-----------+---------+-----------+
|name               |database|description|className|isTemporary|
+-------------------+--------+-----------+---------+-----------+
|TotalSalaryAndBonus|null    |null       |null     |true       |
+-------------------+--------+-----------+---------+-----------+


scala> HEEmployeeDS.withColumn("TotalSalary",calTotalSalaryWithBonus('Salary) ).show
+---+---------+------+------+----------+-----------+
| ID|     Name|gender|Salary|Department|TotalSalary|
+---+---------+------+------+----------+-----------+
|  1|     Deva|  Male|  5000|     Sales|       6000|
|  2|    Jugnu|Female|  6000|        HR|       7200|
```

Similarly you can define aggregate function, but it is quite complex to implement. Follow the example below to step by step creating UDAF.

Exercise : User defined Aggregate functions

```
import org.apache.spark.sql.expressions._
import org.apache.spark.sql.types._
import org.apache.spark.sql._

object HEUDAF {

  // Define the logic for creating your SparkSQL UDAF function. By extending class
UserDefinedAggregateFunction
  private class SumProductAggregateFunction extends UserDefinedAggregateFunction {


  // Define input data schema as well as result data schema
  //Input data will be as ID: integer , Name: string , gender: string , Salary: integer, Department:
string
```

```
  def inputSchema: StructType =   {
     new StructType().add("ID", IntegerType).add("Name", StringType).add("gender",
StringType).add("Salary", IntegerType).add("Department", StringType)
}

  // Define Buffer Schema  Output = (Integer total)
    def bufferSchema: StructType = new StructType().add("total", IntegerType)
   def dataType: DataType = IntegerType

// true means UDAF's output given an input is deterministic
    def deterministic: Boolean = true

// Initialize the result to 0 MutableAggregationBuffer represents a Row object
    def initialize(buffer: MutableAggregationBuffer): Unit = { buffer.update(0, 0) }

//Update the buffer
    def update(buffer: MutableAggregationBuffer, row: Row): Unit = {
// Intermediate result to be updated
val interMediateSum   = buffer.getInt(0)
// First input parameter
val salary = row.getInt(0)
// Update the intermediate result
buffer.update(0, interMediateSum + (salary))      }

    // Merge intermediate result sums by adding them
    def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
      buffer1.update(0, buffer1.getInt(0) + buffer2.getInt(0))
    }
    // The final result will be contained in 'buffer'
    def evaluate(buffer: Row): Any = {
      buffer.getInt(0)
    }
  }
 }
```

Aggregate functions: These types of functions will work on group of rows and return a single value for example min, max, avg, count functions.

Collection functions: Function under this category works on the collections like array, maps etc. To understand these types of functions lets take an example of explode function of DataFrame

```
//Sample data in a file "HE_TRAINING.json"
{"Hadoop":6000,"City":["Mumbai","Hyderabad"]}

//We wanted to convert this data in file as below.
Hadoop,City
6000,Mumbai
```

```
6000,Hyderabad

//We can do it using below Collection function of DataFrame
import org.apache.spark.sql.functions.explode

//Load JSON data as a DataFrame
val heTraining= spark.read.json("/home/hadoopexam/spark2/sparksql/HE_TRAINING.json")

//We need to flatten the data in a City for each Training course
val heTrainingCityFee= heTraining.withColumn("City", explode($"City"))
heTrainingCityFee.show()
```

```
scala> val heTraining= spark.read.json("/home/hadoopexam/spark2/sparksql/HE_TRAINING.json")
heTraining: org.apache.spark.sql.DataFrame = [City: array<string>, Hadoop: bigint]

scala> val heTrainingCityFee= heTraining.withColumn("City", explode($"City"))
heTrainingCityFee: org.apache.spark.sql.DataFrame = [City: string, Hadoop: bigint]

scala> heTrainingCityFee.show()
+----------+------+
|      City|Hadoop|
+----------+------+
|    Mumbai|  6000|
| Hyderabad|  6000|
|   NewYork|  7000|
|Washington|  7000|
|    Sidney|  8000|
|    London|  8000|
|   Kolkata|  9000|
|    Jaipur|  9000|
+----------+------+
```

About explode function: It is very similar, as we have used with the RDD. It will create a new Row for each value or element in a given array or Map. In above dataset City is an array with the two values in it. [Mumbai. Hyderabad]. Which will be generating new row for each city.

Date and Time Functions: As name suggests these are the functions for working on the time and dates manipulation. Please see the below example for Date and Time function.

Window Aggregate Functions: These functions work on a group of rows. However, we have to explicitly define how to create groups out of entire set of rows. And in a group, it will be calculating the single value for each row in a group.

Exercise : Various Date Time and window Functions

```
//This all functions are part of package
import org.apache.spark.sql.functions
//Lets do some arithmatic function on date and timestamp using sql
//Substract date by 1 day
spark.sql("select date_sub(current_timestamp(), 1)").show()
//Get current date
spark.sql("select current_date() ").show()
//add days
 spark.sql("select date_add(current_date() , 2)").show()
 //substract days
 spark.sql("select          date_sub(current_date() , 2)").show()
```

```
scala> import org.apache.spark.sql.functions
import org.apache.spark.sql.functions

scala> spark.sql("select date_sub(current_timestamp(), 1)").show()
2018-09-01 04:30:46 WARN  ObjectStore:568 - Failed to get database global_temp, returning NoSuchObjectException
+---------------------------------------------+
|date_sub(CAST(current_timestamp() AS DATE), 1)|
+---------------------------------------------+
|                                   2018-08-31|
+---------------------------------------------+


scala> spark.sql("select current_date() ").show()
+--------------+
|current_date()|
+--------------+
|    2018-09-01|
+--------------+


scala>  spark.sql("select date_add(current_date() , 2)").show()
+--------------------------+
|date_add(current_date(), 2)|
+--------------------------+
|                2018-09-03|
+--------------------------+


scala>  spark.sql("select date_sub(current_date() , 2)").show()
+--------------------------+
|date_sub(current_date(), 2)|
+--------------------------+
|                2018-08-30|
+--------------------------+
```

```
//add months
 spark.sql("select add_months(current_date() , 2)").show()
//Difference between two dates (Current date - current date+2 months)
spark.sql("select datediff(current_date() , add_months(current_date() , 2))").show()
//Getting the last day of the months
spark.sql("select            last_day(current_date()) ").show()
//Getting the hour part
spark.sql("select hour(current_timestamp())").show()
//Extract month part
spark.sql("select month(current_timestamp())").show()
//Getting number of months between two dates
 spark.sql("select          months_between(add_months(current_date() , 2), current_date())").show()
//Get the quarter of the date
spark.sql("select quarter(current_timestamp())").show()

//Getting similar output from datasets API
spark.range(1).select(current_timestamp).show()
spark.range(1).select(hour(current_timestamp())).show()
spark.range(1).select(last_day(current_timestamp())).show()

//Date Formatting
spark.range(1).select(date_format(current_timestamp, "dd-MMM-yyyy")).show()
spark.range(1).select(date_format(current_timestamp, "dd-MMM-yyyy hh:mm:ss")).show()
spark.range(1).select(date_format(current_timestamp, "dd-MMM-yyyy hh:mm:ss")).show()

//Getting unix timestamp (Also known as unix epoc timestamp)
spark.range(1).select(unix_timestamp).show()

//This way any column of Dataset you can convert into Date Datatype
```

```
spark.range(1).select(to_date(lit("2018-07-27"))).show()

//Creating window with slide duration.
spark.sql("select window(current_timestamp() ,'5 minutes' , '1 minutes')").take(20)

val heCourses = sc.parallelize(Seq(
(1, "2018-01-01", 20000),
(1, "2018-01-02", 23000),
(1, "2018-01-03", 90000),
(1, "2018-01-04", 55000),
(1, "2018-01-05", 20000),
(1, "2018-01-06", 23000),
(1, "2018-01-07", 90000),
(1, "2018-01-08", 55000),
(2, "2018-01-01", 80000),
(2, "2018-01-02", 90000),
(2, "2018-01-03", 100000),
(2, "2018-01-04", 80000),
(2, "2018-01-05", 90000),
(2, "2018-01-06", 100000),
(2, "2018-01-07", 80000),
(2, "2018-01-08", 90000)
)).toDF("course_id", "start_date" , "fee").withColumn("start_date", col("start_date").cast("date"))

 heCourses.show()

// calculating the total fee every day across courses
val totalFeeEveryDay = heCourses.groupBy(window($"start_date", "1 days")).
  agg(sum("fee") as "total_fee").
  select("window.start", "window.end", "total_fee")

totalFeeEveryDay.orderBy('start).show()

//Total fee collected in every two day
val totalFeeEvery2ndDay = heCourses.groupBy(window($"start_date", "2 days")).
  agg(sum("fee") as "total_fee").
  select("window.start", "window.end", "total_fee")

totalFeeEvery2ndDay.orderBy('start).show()
```

```
scala> val totalFeeEveryDay = heCourses.groupBy(window($"start_date", "1 days")).
     |   agg(sum("fee") as "total_fee").
     |   select("window.start", "window.end", "total_fee")
totalFeeEveryDay: org.apache.spark.sql.DataFrame = [start: timestamp, end: timestamp ... 1 more field]

scala> totalFeeEveryDay.orderBy('start).show()
+-------------------+-------------------+---------+
|              start|                end|total_fee|
+-------------------+-------------------+---------+
|2017-12-31 16:00:00|2018-01-01 16:00:00|   100000|
|2018-01-01 16:00:00|2018-01-02 16:00:00|   113000|
|2018-01-02 16:00:00|2018-01-03 16:00:00|   190000|
|2018-01-03 16:00:00|2018-01-04 16:00:00|   135000|
|2018-01-04 16:00:00|2018-01-05 16:00:00|   110000|
|2018-01-05 16:00:00|2018-01-06 16:00:00|   123000|
|2018-01-06 16:00:00|2018-01-07 16:00:00|   170000|
|2018-01-07 16:00:00|2018-01-08 16:00:00|   145000|
+-------------------+-------------------+---------+


scala> val totalFeeEvery2ndDay = heCourses.groupBy(window($"start_date", "2 days")).
     |   agg(sum("fee") as "total_fee").
     |   select("window.start", "window.end", "total_fee")
totalFeeEvery2ndDay: org.apache.spark.sql.DataFrame = [start: timestamp, end: timestamp ... 1 more field]

scala>
     | totalFeeEvery2ndDay.orderBy('start).show()
+-------------------+-------------------+---------+
|              start|                end|total_fee|
+-------------------+-------------------+---------+
|2017-12-31 16:00:00|2018-01-02 16:00:00|   213000|
|2018-01-02 16:00:00|2018-01-04 16:00:00|   325000|
|2018-01-04 16:00:00|2018-01-06 16:00:00|   233000|
|2018-01-06 16:00:00|2018-01-08 16:00:00|   315000|
+-------------------+-------------------+---------+
```

Non-aggregate functions: By looking at below exercise you can see few of the selected non-aggregate functions. For example

- **array** --> Creates a new array column. The input columns must all have the same data type.
- **expr** --> Parses the expression string into the column that it represents.
- **struct** --> Creates a new struct column that composes multiple input columns.
- **monotonically_increasing_id** --> A column expression that generates monotonically increasing 64-bit integers. The generated ID is guaranteed to be monotonically increasing and unique, but not consecutive. The current implementation puts the partition ID in the upper 31 bits, and the record number within each partition in the lower 33 bits. The assumption is that the data frame has less than 1 billion partitions, and each partition has less than 8 billion records.

Exercise - : Some more utility functions

- **expr**
- **array**
- **struct**
- **monotonically_increasing_id**

//expr function, you will be passing any expression in String to this function and based on this data can be filtered.

//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail

```
case class HEEmployee(ID: Int, Name: String, gender : String, Salary: Int, Department:String)

val HEEmployeeDS = sc.parallelize(Seq( HEEmployee(1, "Deva", "Male", 5000, "Sales"), HEEmployee(2, "Jugnu", "Female", 6000, "HR"), HEEmployee(3, "Kavita", "Female", 7500, "IT"), HEEmployee(4, "Vikram", "Male", 6500, "Marketing"), HEEmployee(5, "Shabana", "Female", 5500, "Finance"), HEEmployee(6, "Shantilal", "Male", 8000, "Sales"), HEEmployee(7, "Vinod", "Male", 7200, "HR"),
```

```
HEEmployee(8, "Vimla", "Female", 6600, "IT"), HEEmployee(9, "Jasmin", "Female", 5400,
"Marketing"), HEEmployee(10, "Lovely", "Female", 6300, "Finance"), HEEmployee(11, "Mohan",
"Male", 5700, "Sales"), HEEmployee(12, "Purvish", "Male", 7000, "HR"), HEEmployee(13, "Jinat",
"Female", 7100, "IT"), HEEmployee(14, "Eva", "Female", 6800,"Marketing"), HEEmployee(15,
"Jitendra", "Male", 5000, "Finance")
, HEEmployee(15, "Rajkumar", "Male", 4500, "Finance")
, HEEmployee(15, "Satish", "Male", 4500, "Finance")
, HEEmployee(15, "Himmat", "Male", 3500, "Finance")), 2).toDS()

//Check the data in Dataset
HEEmployeeDS.show()

//Now create an expression. These expressions are Column types.
val maleExpr = expr("gender='Male'")
val femaleExpr= expr("gender='Female'")
val salExpr= expr("Salary >=6600")

//Now apply these filters to the data
HEEmployeeDS.filter(maleExpr).show
HEEmployeeDS.filter(femaleExpr).show
HEEmployeeDS.filter(salExpr).show

//Now lets create array by combining multiple columns in dataset and drop the same column from
output
HEEmployeeDS.filter(salExpr).withColumn("Array" , array('Name,'gender,
'Department)).drop("Name","gende", "Department").show

//Now lets create array by combining multiple columns in dataset and drop the same column from
output
HEEmployeeDS.filter(salExpr).withColumn("Struct" , struct('Name,'gender,
'Department)).drop("Name","gende", "Department").show

//Even both support mixed datatypes as well
HEEmployeeDS.filter(salExpr).withColumn("Array" , array('gender,
'Department,'Salary)).drop("gender", "Department", "Salary").show
HEEmployeeDS.filter(salExpr).withColumn("Struct" , struct('gender,
'Department,'Salary)).drop("gender", "Department", "Salary").show



//monotonically_increasing_id()
//Lets create data with 4 partitions
val HEEmployeeDS = sc.parallelize(Seq( HEEmployee(1, "Deva", "Male", 5000, "Sales"), HEEmployee(2,
"Jugnu", "Female", 6000, "HR"), HEEmployee(3, "Kavita", "Female", 7500, "IT"), HEEmployee(4,
"Vikram", "Male", 6500, "Marketing"), HEEmployee(5, "Shabana", "Female", 5500, "Finance"),
HEEmployee(6, "Shantilal", "Male", 8000, "Sales"), HEEmployee(7, "Vinod", "Male", 7200, "HR"),
HEEmployee(8, "Vimla", "Female", 6600, "IT"), HEEmployee(9, "Jasmin", "Female", 5400,
"Marketing"), HEEmployee(10, "Lovely", "Female", 6300, "Finance"), HEEmployee(11, "Mohan",
```

```
"Male", 5700, "Sales"), HEEmployee(12, "Purvish", "Male", 7000, "HR"), HEEmployee(13, "Jinat",
"Female", 7100, "IT"), HEEmployee(14, "Eva", "Female", 6800,"Marketing"), HEEmployee(15,
"Jitendra", "Male", 5000, "Finance")
, HEEmployee(15, "Rajkumar", "Male", 4500, "Finance")
, HEEmployee(15, "Satish", "Male", 4500, "Finance")
, HEEmployee(15, "Himmat", "Male", 3500, "Finance")), 4).toDS()

// Now generate monotonically_increasing_id() for each row.
//It is a 64 bit integers
//Generated ID must be unique and increasing only.
//It can not be consecutive
//The current implementation puts the partition ID in the upper 31 bits, and the record number within
each partition in the lower 33 bits.
//The assumption is that the data frame has less than 1 billion partitions, and each partition has less
than 8 billion records.
HEEmployeeDS.withColumn("unique_id", monotonically_increasing_id).show()
```

Sorting functions: This function will be used to sort the data. We will be using these functions in some other full-length exercises in other section of the book. Below is the list of functions, which falls under this category are (API Doc)

asc(columnName: String):	Returns a sort expression based on ascending order of the column. **Example**: df.sort(asc("dept"), desc("age"))
asc_nulls_last(columnName: String)	Returns a sort expression based on ascending order of the column, and null values appear after non-null values. Example: df.sort(asc_nulls_last("dept"), desc("age"))
desc(columnName: String)	Returns a sort expression based on the descending order of the column. **Example**: df.sort(asc("dept"), desc("age"))
desc_nulls_first(columnName: String)	Returns a sort expression based on the descending order of the column, and null values appear before non-null values. **Example**: df.sort(asc("dept"), desc_nulls_first("age"))
def desc_nulls_last(columnName: String)	Returns a sort expression based on the descending order of the column, and null values appear after non-null values. **Example**: df.sort(asc("dept"), desc_nulls_last("age"))
asc_nulls_first(columnName: String)	Returns a sort expression based on ascending order of the column, and null values return before non-null values. **Example**: df.sort(asc_nulls_last("dept"), desc("age"))

String functions: As other programming language these functions are used to manipulate the string. We will not go into detail of this function, please refer the API doc for understanding and each individual function. Example of the few functions under this category are below.

1. **concat** : Concatenates multiple input columns together into a single column. If all inputs are binary, concat returns an output as binary. Otherwise, it returns as string.
2. **initcap**: Returns a new string column by converting the first letter of each word to uppercase.
3. **length**: Computes the character length of a given string or number of bytes of a binary string.

These all are trivial and self-explanatory functions, if you have experience with any other programming language than similar functions you would have found with them.

More Window Functions Example: Lead and Lag are part of a class of functions called window functions. When you write a query, as the SparkSQL processes each row, lead will look ahead at the next row in the result set in the context of the current row being processed. Lag will look behind in the result set to the row that was processed. However, it is not necessary that you look only just next (in case of lead) or previous (in case of lag) rows. Rather by defining length you can check values in next n rows (in case of lead) or previous n rows values (in case of lag). Let's see an example for lead and lag function to understand the functionality.

Exercise: Lead and Lag Function

```
//To use the various functions, we may have to import sql functions
import org.apache.spark.sql.functions._

//You can check the available number of functions
spark.catalog.listFunctions.count

//Window partition ranked function
//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail
case class HEEmployee(ID: Int, Name: String, gender : String, Salary: Int, Department:String)

val HEEmployeeDS = sc.parallelize(Seq( HEEmployee(1, "Deva", "Male", 5000, "Sales"), HEEmployee(2,
"Jugnu", "Female", 6000, "HR"), HEEmployee(3, "Kavita", "Female", 7500, "IT"), HEEmployee(4,
"Vikram", "Male", 6500, "Marketing"), HEEmployee(5, "Shabana", "Female", 5500, "Finance"),
HEEmployee(6, "Shantilal", "Male", 8000, "Sales"), HEEmployee(7, "Vinod", "Male", 7200, "HR"),
HEEmployee(8, "Vimla", "Female", 6600, "IT"), HEEmployee(9, "Jasmin", "Female", 5400,
"Marketing"), HEEmployee(10, "Lovely", "Female", 6300, "Finance"), HEEmployee(11, "Mohan",
"Male", 5700, "Sales"), HEEmployee(12, "Purvish", "Male", 7000, "HR"), HEEmployee(13, "Jinat",
"Female", 7100, "IT"), HEEmployee(14, "Eva", "Female", 6800,"Marketing"), HEEmployee(15,
"Jitendra", "Male", 5000, "Finance")
, HEEmployee(15, "Rajkumar", "Male", 4500, "Finance")
, HEEmployee(15, "Satish", "Male", 4500, "Finance")
, HEEmployee(15, "Himmat", "Male", 3500, "Finance"))).toDS()

//Create a Window based on the Gender to rank their salary
//For the same salary it will assign same rank
import org.apache.spark.sql.expressions.Window
```

```
val genderPartitionedSpec = Window.partitionBy('gender).orderBy($"Salary".desc_nulls_last)

//Lag function will help you find the previous value in the same column
HEEmployeeDS.withColumn("previousValue", lag('Salary, 1) over genderPartitionedSpec).show()

//How to find previous second last value in a column
HEEmployeeDS.withColumn("previousValue", lag('Salary, 2) over genderPartitionedSpec).show()

//Similarly third last and increase the range as per need
HEEmployeeDS.withColumn("previousValue", lag('Salary, 3) over genderPartitionedSpec).show()

//Get the difference between previous value and current value
HEEmployeeDS.withColumn("previousValue", lag('Salary, 1) over genderPartitionedSpec).select('ID,
'Name, 'gender, 'Department,  'Salary, 'previousValue, ('Salary-'previousValue) as "salaryDiff").show()

//Now opposite of that using lead function
HEEmployeeDS.withColumn("lead", lead('Salary, 1) over genderPartitionedSpec).show()
HEEmployeeDS.withColumn("leadBy2", lead('Salary, 2) over genderPartitionedSpec).show()
```

Examples of rank and dense_rank functions (Window function):

dense_rank() : This function returns the rank of each row within a result set partition, with no gaps in the ranking values. The rank of a specific row is one plus the number of distinct rank values that come before that specific row.

rank(): Returns the rank of each row within the partition of a result set. The rank of a row is one plus the number of ranks that come before the row in question. Please note that there is a little difference between rank and dense_rank function, dense_rank will give continuous ranking values if more than one record has same rank, but in case of rank it will produce a gap. Refer the example below to understand in detail. For in-depth definition of similar function refer this MS doc

Exercise-28: Apply Various Rank Functions on Dataset

```
//To use the various functions, we may have to import sql functions
import org.apache.spark.sql.functions._

//You can check the available number of functions
spark.catalog.listFunctions.count

//Window partition ranked function
//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail
case class HEEmployee(ID: Int, Name: String, gender : String, Salary: Int, Department:String)

val HEEmployeeDS = sc.parallelize(Seq( HEEmployee(1, "Deva", "Male", 5000, "Sales"),
HEEmployee(2, "Jugnu", "Female", 6000, "HR"), HEEmployee(3, "Kavita", "Female", 7500, "IT"),
HEEmployee(4, "Vikram", "Male", 6500, "Marketing"), HEEmployee(5, "Shabana", "Female", 5500,
"Finance"), HEEmployee(6, "Shantilal", "Male", 8000, "Sales"), HEEmployee(7, "Vinod", "Male",
```

```
7200, "HR"), HEEmployee(8, "Vimla", "Female", 6600, "IT"), HEEmployee(9, "Jasmin", "Female",
5400, "Marketing"), HEEmployee(10, "Lovely", "Female", 6300, "Finance"), HEEmployee(11,
"Mohan", "Male", 5700, "Sales"), HEEmployee(12, "Purvish", "Male", 7000, "HR"),
HEEmployee(13, "Jinat", "Female", 7100, "IT"), HEEmployee(14, "Eva", "Female",
6800,"Marketing"), HEEmployee(15, "Jitendra", "Male", 5000, "Finance")
, HEEmployee(15, "Rajkumar", "Male", 4500, "Finance")
, HEEmployee(15, "Satish", "Male", 4500, "Finance")
, HEEmployee(15, "Himmat", "Male", 3500, "Finance"))).toDS()

//Create a Window based on the Gender to rank their salary
//For the same salary it will assign same rank
import org.apache.spark.sql.expressions.Window
val genderPartitionedSpec = Window.partitionBy('gender).orderBy($"Salary".desc_nulls_last)

HEEmployeeDS.withColumn("rank", rank over genderPartitionedSpec).show

//Create a Window based on the Departrment to rank their salary
val departmentPartitionedSpec =
Window.partitionBy('Department).orderBy($"Salary".desc_nulls_last)

HEEmployeeDS.withColumn("rank", rank over departmentPartitionedSpec).show

//Create a Window based on the Departrment as well as gender to rank their salary
val departmentGenderPartitionedSpec = Window.partitionBy('Department,
'gender).orderBy($"Salary".desc_nulls_last)

HEEmployeeDS.withColumn("rank", rank over departmentGenderPartitionedSpec).show

//Lets get percent rank
//For the same salary it will assign same rank
import org.apache.spark.sql.expressions.Window
val genderPartitionedSpec = Window.partitionBy('gender).orderBy($"Salary".desc_nulls_last)

HEEmployeeDS.withColumn("percentRank", percent_rank over genderPartitionedSpec).show

//Use the dens_rank
//It will give you the continuous rank
import org.apache.spark.sql.expressions.Window
val genderPartitionedSpec = Window.partitionBy('gender).orderBy($"Salary".desc_nulls_last)

HEEmployeeDS.withColumn("denseRank", dense_rank over genderPartitionedSpec).show
```

NTILE (Window) function: Distributes the rows in an ordered partition into a specified number of groups. The groups are numbered, starting at one. For each row, NTILE returns the number of the group to which the row belongs.

row_number() function : Numbers the output of a result set. More specifically, returns the sequential number of a row within a partition of a result set, starting at 1 for the first row in each partition.

Exercise : Some other useful window based functions

```
//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail
case class HEEmployee(ID: Int, Name: String, gender : String, Salary: Int, Department:String)

val HEEmployeeDS = sc.parallelize(Seq( HEEmployee(1, "Deva", "Male", 5000, "Sales"), HEEmployee(2, "Jugnu", "Female", 6000, "HR"), HEEmployee(3, "Kavita", "Female", 7500, "IT"), HEEmployee(4, "Vikram", "Male", 6500, "Marketing"), HEEmployee(5, "Shabana", "Female", 5500, "Finance"), HEEmployee(6, "Shantilal", "Male", 8000, "Sales"), HEEmployee(7, "Vinod", "Male", 7200, "HR"), HEEmployee(8, "Vimla", "Female", 6600, "IT"), HEEmployee(9, "Jasmin", "Female", 5400, "Marketing"), HEEmployee(10, "Lovely", "Female", 6300, "Finance"), HEEmployee(11, "Mohan", "Male", 5700, "Sales"), HEEmployee(12, "Purvish", "Male", 7000, "HR"), HEEmployee(13, "Jinat", "Female", 7100, "IT"), HEEmployee(14, "Eva", "Female", 6800,"Marketing"), HEEmployee(15, "Jitendra", "Male", 5000, "Finance")
, HEEmployee(15, "Rajkumar", "Male", 4500, "Finance")
, HEEmployee(15, "Satish", "Male", 4500, "Finance")
, HEEmployee(15, "Himmat", "Male", 3500, "Finance"))).toDS()

//Create a Window based on the Gender to rank their salary
//Assign Sequence by ordering on salary
import org.apache.spark.sql.expressions.Window
val genderPartitionedSpec = Window.partitionBy('gender).orderBy($"Salary".desc_nulls_last)


HEEmployeeDS.withColumn("rowNumber", row_number() over genderPartitionedSpec).show()

//Select ntile (Various percentire)
//If we divide salary in 3 quartile than in which quartile it fall
HEEmployeeDS.select('*, ntile(3) over genderPartitionedSpec as "ntile").show

//Divide with 25% and see in which 25%, employee salary false
HEEmployeeDS.select('*, ntile(4) over genderPartitionedSpec as "ntile").show
```

Cumulative Distribution: This function calculates the cumulative distribution of a value within a group of values. In other words, CUME_DIST calculates the relative position of a specified value in a group of values. Assuming ascending ordering, the CUME_DIST of a value in row r is defined as the number of rows with values less than or equal to that value in row r, divided by the number of rows evaluated in the partition or query result set. CUME_DIST is similar to the PERCENT_RANK function.

Exercise : Cumulative Distribution

```
//To use the various functions, we may have to import sql functions
import org.apache.spark.sql.functions._

//You can even use case class to create DataFrame
```

```
//Define a Case class for HadoopExam course detail
case class HEEmployee(ID: Int, Name: String, gender : String, Salary: Int, Department:String)

val HEEmployeeDS = sc.parallelize(Seq( HEEmployee(1, "Deva", "Male", 5000, "Sales"), HEEmployee(2,
"Jugnu", "Female", 6000, "HR"), HEEmployee(3, "Kavita", "Female", 7500, "IT"), HEEmployee(4,
"Vikram", "Male", 6500, "Marketing"), HEEmployee(5, "Shabana", "Female", 5500, "Finance"),
HEEmployee(6, "Shantilal", "Male", 8000, "Sales"), HEEmployee(7, "Vinod", "Male", 7200, "HR"),
HEEmployee(8, "Vimla", "Female", 6600, "IT"), HEEmployee(9, "Jasmin", "Female", 5400,
"Marketing"), HEEmployee(10, "Lovely", "Female", 6300, "Finance"), HEEmployee(11, "Mohan",
"Male", 5700, "Sales"), HEEmployee(12, "Purvish", "Male", 7000, "HR"), HEEmployee(13, "Jinat",
"Female", 7100, "IT"), HEEmployee(14, "Eva", "Female", 6800,"Marketing"), HEEmployee(15,
"Jitendra", "Male", 5000, "Finance")
, HEEmployee(15, "Rajkumar", "Male", 4500, "Finance")
, HEEmployee(15, "Satish", "Male", 4500, "Finance")
, HEEmployee(15, "Himmat", "Male", 3500, "Finance"))).toDS()

//Create Cumulative Distribution Window Spec
import org.apache.spark.sql.expressions.Window
val cumDisWindowSpec = Window.partitionBy('gender).orderBy('Salary)
HEEmployeeDS.withColumn("cumeDist", cume_dist over cumDisWindowSpec).show

//Create Cumulative Distribution Window Spec
val cumDisWindowSpec = Window.partitionBy('Department,'gender).orderBy('Salary)
HEEmployeeDS.withColumn("cumeDist", cume_dist over cumDisWindowSpec).show
```

Chapter-15: Dataset Actions and Transformations

- Dataset Partitioning
- About coalesce operator of Dataset
- Dataset typed transformation

Dataset Partitioning:

As we already know that Spark is a Distributed computation engine, where on different data same computation happens on each node in parallel. Part of entire collection of data reside over each node is known as a partition.

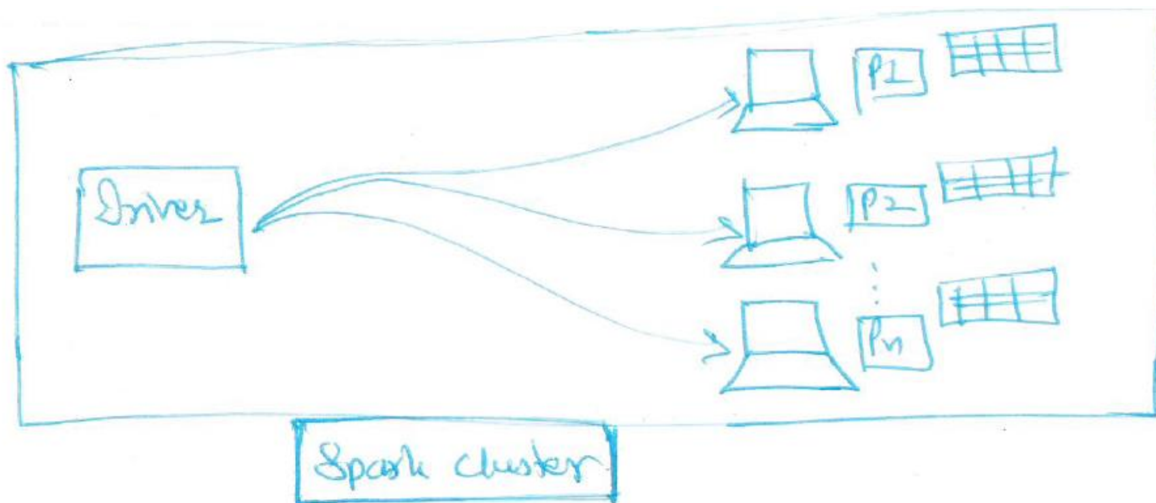


Figure 32: Spark Cluster with Dataset Partitioning

Data would be partitioned based on the following, to decide what partition strategy to be used:

1. Number of cores in executors
2. Size of the data

Based on above two values, Spark optimizes the parallelism while processing the data. Also there is a one parameter which decides number of partitions for a Dataset, which is below.

```
spark.sql.shuffle.partitions
```

This parameter is having default value as 200. If you want to change the value in your SparkSession, you can use **spark.conf.set** operator to update this value, similarly other configuration parameters you can change. Here spark is an instance of SparkSession.

If you want to check what all are the partitions are currently available than you have to use below function of the Dataset.

```
heDS.rdd.partitions.size()
```

heDS : It is a Dataset.

As you can see partitioning is done on the RDD and not directly on the Dataset object. Hence, we are first retrieving underline RDD of the Dataset and checking what is the total number of partitions exists for this RDD.

Repartitioning: If you want to re-partition the data than you have to use below operator.

```
heDS.repartition(x) //Here x, is a number value for partitions to be created
```

About coalesce operator of Dataset: It is considered as a typed transformation of a Dataset.

- This also helps you to re-partition the Dataset in the given number of partitions.
- Let's see the scenario, what happens

  If current partitions are more than requested partitions

Current → 5 and Requested → 3 *// It will generate new dataset with 3 partitions*
Current → 5 and Requested → 6*// It will remain as 5 partitions only*

Exercise for Partitions and coalesce functions

Exercise : Partitions and coalesce functions

```
//Create a dataset with 3 partitions
case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)

val heDS2 = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000,
"Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3)) , 3).toDS()

//Check number of partitions
heDS2.rdd.partitions.size

//Repartition the Dataset in 1
val heDSNew=heDS2.repartition(1)

//Check number of partitions
heDSNew.rdd.partitions.size
```

```
//Create a dataset with 3 partitions
val heDS2 = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000,
"Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3)) , 3).toDS()

//Check number of partitions
heDS2.rdd.partitions.size

//Repartition the Dataset in 1
val heDSNew=heDS2.coalesce (1)

//Check number of partitions
heDSNew.rdd.partitions.size

//Repartition the Dataset in 5
val heDSNew=heDS2.coalesce (5)

//Check number of partitions
heDSNew.rdd.partitions.size
```

```
scala> case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)
defined class HECourse

scala> val heDS2 = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3)) , 3).t
oDS()
heDS2: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

scala> heDS2.rdd.partitions.size
res19: Int = 3

scala> val heDSNew=heDS2.repartition(1)
heDSNew: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

scala> heDSNew.rdd.partitions.size
res20: Int = 1

scala> val heDS2 = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3)) , 3).t
oDS()
heDS2: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

scala> heDS2.rdd.partitions.size
res21: Int = 3

scala> val heDSNew=heDS2.coalesce (1)
heDSNew: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

scala> heDSNew.rdd.partitions.size
res22: Int = 1

scala> val heDSNew=heDS2.coalesce (5)
heDSNew: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

scala> heDSNew.rdd.partitions.size
res23: Int = 3
```

Dataset typed transformations: Lets see some of the typed transformations of the Dataset

- **as operator of DataFrame**: This you will be using to convert a DataFrame (Generic Rows Dataset) to a Dataset (typed rows of the Dataset)
- **dropDuplicates**: Returns a new DataFrame with duplicate rows removed, considering only the subset of columns, if you have provided else it will consider all the columns. If you dont provide the list of columns than it is similar to distinct function
- **except:** except returns distinct records from the left dataset that aren't output by the right dataset
- filter: Based in given condition only those records will return, which satisfy the given condition in filter functions.
- sort: Data will be sorted based on the given column.

- distinct: From given dataset, it will return only distinct records.

Exercise : Dataset Type Transformations

```
//Define a Case class for HadoopExam course detail
case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)

//Create an RDD with 5 HECourses
val courseRDD = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HECourse(4, "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7)))

//Check the types of RDD
courseRDD

//Convert RDD into dataset, as RDD has schema information, so Dataset will automatically infer that schema.
val heCourseDS = courseRDD.toDS

heCourseDS: org.apache.spark.sql.Dataset[HECourse] = [id: int, name: string ... 3 more fields]

//If you want to change the partitions use coalesce or repartition method.
heCourseDS.coalesce(1)
heCourseDS.repartition(1)

//Lets create a DataFrame
val heDF = spark.read.format("csv").option("header",true).option( "Inferschema", true).load("/home/hadoopexam/spark2/sparksql/HadooExam_Training.csv")

//You can even use case class to create DataFrame
//Define a Case class for HadoopExam course detail
case class HECourse(ID: Int, Name: String, Fee : Int, Venue: String, Date:String, Duration:Int)

//Converting to dataset
val heCourseDS = heDF.as[HECourse]


//Define a Case class for HadoopExam course detail
case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)

//Getting distinct rows from Dataset
val heDS = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HECourse(4, "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7) ,HECourse(4, "Scala", 4000, "Kolkata", 3),HECourse(5, "HBase", 7000, "Banglore", 7) ,HECourse(11, "Scala", 4000, "Kolkata", 3),HECourse(12, "HBase", 7000, "Banglore", 7))).toDS()

//Getting distinct values from Dataset
```

```
heDS.distinct().show()

//Remove duplicates using selected columns
heDS.dropDuplicates("name","fee","venue","duration").show()

//Removing all the common rows from a Dataset
val heDS2 = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000,
"Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3))).toDS()

//except will remove all the rows from heDS which are present in heDS2 and also gives unique rows
//from heDS
heDS.except(heDS2).show()

//Using filter function
heDS.filter(data => data.fee>6000).show()

//Adding new column to existing dataset with calculations
//Import Required types
import org.apache.spark.sql._

//Define a function
def withTax(dataset: Dataset[HECourse]) = dataset.withColumn("fee_tax", 'fee + 500)

//Apply the function, so that new Dataset will be created with the new calculated columns
heDS.transform(withTax).show

//Applying flatMap functionality
//Create a case class for HadoopExam website posted comments
case class HEComments(id: Long, comment: String)

//Create Sample Data Dataset
val heWebComments = Seq(HEComments(1001, "I wanted to subscribe custom package"),
HEComments(1002, "Hi this is Lokesh and want to subscribe entire Spark package")).toDS

//Apply the flatMap so you can find all the words in comments for further analysis
heWebComments.flatMap(s => s.comment.split("\\s+")).show

//Union two datasets
val heDS = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000,
"Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HECourse(4, "Scala", 4000, "Kolkata",
3),HECourse(5, "HBase", 7000, "Banglore", 7) ,HECourse(4, "Scala", 4000, "Kolkata", 3),HECourse(5,
"HBase", 7000, "Banglore", 7) ,HECourse(11, "Scala", 4000, "Kolkata", 3),HECourse(12, "HBase", 7000,
"Banglore", 7))).toDS()

//Create another Dataset
val heDS2 = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark", 5000,
"Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3))).toDS()
```

```
//Apply union and then sort (In SQL it is equivalent to UNION ALL), if you want to remove duplicate
values, you have to use distinct as well (that's the reason unionAll have been deprecated
heDS.union(heDS2).sort("id").show()

//De-duplication
heDS.union(heDS2).distinct().sort("id").show()

//Converting your Dataset contents in JSON format
heDS.toJSON.show()
```

Actions on the Dataset: In this section we will do exercise considering actions of the DataFrame/Dataset.

- toDS : Convert RDD to Dataset
- collect():
- collectAsList():
- describe:
- summary:

Exercise : Some actions on the Dataset

```
//Define a Case class for HadoopExam course detail
case class HECourse(id: Int, name: String, fee : Int, venue: String, duration:Int)

//Create an RDD with 5 HECourses
val courseRDD = sc.parallelize(Seq(HECourse(1, "Hadoop", 6000, "Mumbai", 5),HECourse(2, "Spark",
5000, "Pune", 4),HECourse(3, "Python", 4000, "Hyderabad", 3) ,HECourse(4, "Scala", 4000, "Kolkata",
3),HECourse(5, "HBase", 7000, "Banglore", 7)))

//Check the types of RDD
courseRDD

//Convert RDD into dataset, as RDD has schema information, so Dataset will automatically infer that
schema.
val heCourseDS = courseRDD.toDS

//Collect method, which can cause OutOfMemory issue as well
heCourseDS.collect()

//Collect As List, same can lead to OutOfMemory
heCourseDS.collectAsList

//Describe a Dataset columns (Exploratory Data Analysis, Are you new to DataScience?)
//If you want to calculate analytics your own, you have to use agg functions
heCourseDS.describe("fee","duration").show()

//You can also use the summary method for the same calculations also does quartiles calculations.
heCourseDS.summary().show()
```

```
//Specifying what to calculate
heCourseDS.summary("count", "min", "max").show()
```

Chpater-16: Spark Certifications

- Databricks Certifications
- Cloudera Hadoop & Spark Developer Certifications
- Hortonworks HDPCD-Spark Certifications
- MapR Spark Developer Certifications

Databricks Certifications

Databricks is a company which was founded by Spark developer and this is the company which is actively involved with the Apache Spark development with the latest releases. Hence, if you want to go for Latest version of Spark Certifications than you should consider Spark certifications from Databricks. There are currently two certifications which are provided by Databricks one is in Scala and other is in Python programming language. Syllabus remain same, but it is more of for the learners who are comfortable in specific language. It seems Databricks will also launch one more certification for Spark using language. As per Databricks certification site following topics will be covered in certification exam and exam will be focusing on Spark 2.x version.

- Spark Basics
- Spark Streaming
- Spark Architecture
- Spark ML
- Spark Performance and Debugging
- Spark SQL
- GraphFrames

How to prepare for Databricks Spark Certifications?

To prepare for any technology certification of your interest you should consider 2-3 months' timeline. If you get the well organized and focused material. If you don't get the organized and certification focus material it would be difficult to prepare the exam and timeline can increate upto 9-12 moths and by the time many things would change with respect to certifications. Hence, to prepare you for the Databricks Spark Developer certifications in Scala and Python you should consider below preparation material.

This material is available on http://www.HadoopExam.com

- *Trainings* : If you are not familiar and having average experience of the Spark frameworks than we recommend below trainings which will help you prepare for these certifications
  - Apache Spark Professional (Include 2.x latest Version setup) Training with Hands On Lab
  - Spark 2.X SQL (Using Scala) Professional Training with Hands On Sessions
  - Scala Professional Training with HandsOn Session
  - Python Professional Training with Hands-on Sessions

- *Practice Questions and Answers*: To save time and focused approach for Spark certifications you should go through the below certification material for Databricks Spark certifications.
  - Databricks Certified Developer Apache Spark 2.x for Scala (Cert No : PR000003)
  - Databricks Certified Developer Apache Spark 2.x for Python (Cert No : PR000005) : PySpark

Cloudera Hadoop and Spark Developer Certifications: Cloudera is a pioneer for Hadoop framework and they have lot of frameworks for BigData paradigm. Cloudera provide one of the mostly used Hadoop Framework and known as CDH (Cloudera Hadoop Distribution). CDH is bundle of various big data software and one of them is Spark. Cloudera also focuses on Spark for data processing rather than traditional MapReduce frameworks. Hence, they are also delivering Spark software as part of their CDH distribution. Cloudera has various certifications for Hadoop and BigData professionals. For the Spark developer one of the most popular certifications since last 2 yrs. is been this CCA175 (Cloudera Hadoop and Spark Developer certification)

In this certification 30%-40% focus is on Spark and remaining part is Hadoop Data Processing.

How to prepare for CCA175?

On http://www.HadoopExam.com this is the certification preparation material which is most subscribed among many top 10 certifications. HadoopExam provide a combined package for preparing CCA175 which include below three products.

- Spark Professional Training. with Hanson Session
- Hadoop Professional Training with Hands-on Session
- CCA175 Spark and Hadoop Developer Certifications (Includes 111 Solved Scenarios and Complimentary videos for selected solutions)

Hortonworks Spark Certification preparation material **by http://www.HadoopExam.com :**

Total 65 Solved scenarios which includes in depth complex scenarios solved for Spark Scala Application, RDD, Broadcast Variables, Accumulators, RDD transformations, RDD Actions, DataFrames, SparkSQL, SparkSQL using Hive and many more objective mentioned by Hortonworks and covering the entire syllabus. It's a performance based exam to do hands-on task on **HDP platform (Complementary Selected videos will be provided to help with this.)** Practice and Sample Problem with its solutions will be provided in HadoopExam Simulator only (Check Videos to understand more). In real exam you will be asked many problems which can be solved mixing the objectives. We

have added complex scenario as well as step by step solutions for each practice task. These problem scenarios not only helps for HDPCD exam, but also it will help in your real life BigData problems, which can be solved using Spark. Hence solve these scenarios and become BigData experts, with Hands-on**.** **Once you complete all 65 problem scenarios your own, you will be ready to clear real HDPCD Spark Developer exam. This is one of the most demanding certification among the Hadoop Developer on** HadoopExam.com

This certification is helpful for all the analyst, Hadoop Developer and Data scientists who are working in various industry like E-Commerce, Finance, Investment Banks, Health Care, Telecom and with many startups.

This entire package will prepare you for Hadoop as well as prepare you for HDPCD: Spark in few days. (It will take approx. 60Hrs to complete all these material)

- Hadoop Professioanl Training with HandsOn Session
- Spark Professional Training with HandsOn Session
- Hortonworks HDPCD:Spark Developer Certifications

MapR Spark Spark Certifications**:** The *MapR Certified Spark v2.1 Developer* credential proves that you have ability to use Spark to work with large datasets to perform analytics on streaming data. This credential measures your understanding of the Spark API to perform basic machine learning or SQL tasks on a given datasets.

This material is available on http://www.HadoopExam.com

- Trainings : If you are not familiar and having average experience of the Spark frameworks than we recommend below trainings which will help you prepare for these certifications
    - Apache Spark Professional (Include 2.x latest Version setup) Training with Hands On Lab
    - Spark 2.X SQL (Using Scala) Professional Training with Hands On Sessions
    - Scala Professional Training with HandsOn Session
    - Scala Professional Training with HandsOn Session

Practice Questions and Answers: To save time and focused approach for Spark certifications you should go through the below certification material for Databricks Spark certifications

- About MapR MCSD: MapR® Certified Spark Developer: Total 220+ Solved Questions: Recently updated based on learners feedback.

Other Products, which may interests you from http://HadoopExam.com

1. Databricks Spark 2.x Spark Developer Certification Scala
2. Databricks Spark 2.x (PySpark) Developer Certification Python

3. Apache Spark Professional Training with Hands On Lab Sessions
4. Oreilly Databricks Apache Spark Developer Certification Simulator (Retired)
5. Hortonworks Spark Developer Certification
6. Cloudera CCA175 Hadoop and Spark Developer Certification
7. MCSD : MapR Spark (Scala) Certified Developer
8. CCA 175 : Cloudera® Hadoop & Spark Developer : 95 Solved Scenarios
9. CCA159: Cloudera® Data Analyst Certification : 73 Solved Scenarios
10. CCA131 : Cloudera Hadoop Administrator Certification : 92 Solved Scenarios
11. CCP:DE 575 : Cloudera Hadoop Data Engineer : 79 Solved Scenarios
12. Training : CDH : Cloudera Hadoop Admin Beginner Course-1 : 30 Training Modules
13. Hadoop Professional Training
14. HBase Professional Training
15. Hadoop Package Deal
16. HDPCD : Hadoop (HDP) No Java Certification : 74 Solved Scenarios
17. HDPCD-Spark : HDP Certified Developer : 65 Solved Scenarios
18. HDPCA : HDP Certified Administrator : 57 Solved Scenarios
19. Hortonworks Certification Package Deal
20. Data Science Certification EMC® E20-007 (Data Science Associate)
21. EMC® Data Science Specialist (E20-065)
22. Cloudera Data Scinece DS-200 (235 Questions + 150 Page Study Notes) : Retired
23. MCSD : MapR Spark (Scala) Certified Developer
24. MapR Hadoop Developer Certification
25. MapR HBase NoSQL Certifcation
26. MapR Package Deal
27. AWS Solution Architect Associate : Training
28. AWS Solution Architect Associate Certification Preparation
29. AWS Solution Architect Professional Certification Preparation
30. AWS Sysops Certification Preparation
31. AWS Developer Certification Preparation
32. IBM C2090-102: IBM Big Data Architect : Total 240 Questions : Highest number of Questions : 95% Questions with explanations
33. Professional Certification Apache Cassandra(Datastax) : Total 207 Questions : Highest number of Questions : 95% Questions with explanations
34. SAS Base Certification Professional Training
35. SAS Base Programming Certification(A00-211)
36. SAS Certified Advanced Programmer for SAS 9 Credential
37. SAS Certified Statistical Business Analyst Using SAS 9: Regression and Modeling Credential
38. SAS Certified Platform Administrator 9 (A00-250) Certification Practice Questions

39. SAS Package Deal
40. HBase professional Training with HandsOn Sessions
41. MapR HBase certification preparations
42. Microsoft Azure 70-532 Developing Azure Solution Certification
43. Microsoft Azure 70-533 Implementing Microsoft Azure Infrastructure Solutions
44. Oracle 1Z0-337 Oracle Oracle Infrastructure as a Service Certified Implementation Specialist
45. Full length HandsOn Step By Step Training for Java 1z0-808)
46. Scala Professional Trainings with HandsOn Session
47. Python Professional Trainings with HandsOn Session
48. Java SE-8 Programmer-1 (1z0-808) Certification
49. Java SE-8 Programmer-2 (1z0-809)
50. JAVA EE Web Services Developer (1z0-897)
51. Oracle® 1Z0-060 : Upgrade to Oracle Database 12c Administrator
52. Questions for Oracle 1Z0-061 : Oracle Database 12c: SQL Fundamentals