SPARK SQL

**SparkSession**

Before getting into **SparkSession**, let's understand the entry point.

* An entry point is where control is transferred from the operating system to the provided program.
* Before spark 2.0, the entry point to Spark Core was **sparkContext**.

Essentially,

* SparkContext allows your application to access the cluster through a resource manager.
* Prior to Spark 2.0, sparkContext was used as a channel to access all Spark functionalities
* The Spark driver program uses sparkContext to connect to the cluster through resource manager.
* SparkConf is required to create sparkContext objects, which stores configuration parameters like appName (to identify your Spark driver), the core number, and the memory size of the executor running on a worker node
* As we know, in previous versions, sparkContext is the entry point for Spark. As RDD was the main API, it was created and manipulated using context APIs
* For every other API, we needed to use different contexts.
  + For streaming, we needed **StreamingContext**, for SQL **sqlContext** and for hive **HiveContext**.
  + HiveContext is a super set of SQLContext that you would need if you want to access Hive tables, or to use richer functionalities and the trade- off is that HiveContext requires many dependencies to run.
* But as DataSet and DataFrame APIs are becoming new standalone APIs, we need an entry point build for them. So in Spark 2.0, we have a new entry point build for DataSet and DataFrame APIs called as SparkSession.

So in Spark 2.x, we have a new entry point for concepts such as **DataSet** and **Dataframe** API’s which is called as Spark Session.

* **SparkSession** is essentially combination of SQLContext, HiveContext and and in future StreamingContext (now streamingContext is not part of sparkSession).
* All the API’s available on those contexts are available on spark session also. Spark session internally has a spark context for actual computation.
* SparkSession also take care of handling DataSet and DataFrames which are the improved and recent concepts in Spark after spark 2.0

**Creating a SparkSession**

// The builder automatically reuse an existing SparkContext if one exists and creates a SparkContext if it does not exist.

A SparkSession can be created using a builder pattern.

Configuration options set in the builder are automatically propagated to Spark and Hadoop during I/O.

import org.apache.spark.sql.SparkSession

val sparkSession = ~~new~~ SparkSession.builder.getOrCreate()

|  |
| --- |
| **Concepts:**   * While instantiating sparkSession we don’t want to create a separate constructor using the key work ‘new’ because, sparkSession is a single-ton object. It will be instantiated automatically when instantiated. * And as it is single-ton object, it can be instantiated only once. We can create multiple instance for sparkSession. * In spark 1.x, we need to instantiate context() class for reach concept like Spark context, spark hive, spark core, spark sql… * In Spark 2.x, sparkSession have all the above classes included with in it. * So if we instantiate sparkSession, then no need to create instance separately for core, sql, hive etc.. * sparkSession supports Datasets and Dataframes along with all the above discussed points.   SparkSession.builder.getOrCreate()  In the above command, we invoke the abstract class SparkSession.  while invoking SparkSession we do also access the constructor **“builder”** and using this constructor we access its method **getOrCreate()**  **Builder**   * takes care of creating instance for all above discussed classes such as sparkContext, sparkSQL, sparkCore, sparkHive…. * Using this constructor we can access all the methods from its corresponding classes. * Few frequently accessed methods are * [appName](https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/sql/SparkSession.Builder.html#appName-java.lang.String-)(String name) * getOrCreate() * enableHiveSupport()   **appName(String name)**   * Sets a name for the application, which will be shown in the Spark web UI.   **getOrCreate()**   * Gets an existing [SparkSession](https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/sql/SparkSession.html" \o "class in org.apache.spark.sql) or, if there is no existing one, creates a new one based on the options set in this builder. * This method first checks whether there is a valid thread-**local** SparkSession, and if yes, return that one. * It then checks whether there is a valid **global** default SparkSession, and if yes, return that one. * If no valid global default SparkSession exists, the method creates a new SparkSession and assigns the newly created SparkSession as the global default. * In case an existing SparkSession is returned, the config options specified in this builder will be applied to the existing SparkSession. |

The single-ton object SparkSession internally has all the codes which we discussed earlier as show below as example.

Val conf = new org.apache.spark.sparkConf()

Val sparkcontext = new org.apache.spark.sparkContext(conf)

Val sparkSql = new org.apache.sql.SQLContext(sparkcontext)

….and so on………

By default when we use SparkSession(),

SparkContext() is the class , sparkContext is its object 🡪 in SparkSession()

SQLContext() is the class , sqlContext is the object 🡪 in SparkSession()

Here are some of the frequently used methods using the constructor “**builder**"

|  |  |
| --- | --- |
| SparkSession.Builder | appName(String name) |
| Sets a name for the application, which will be shown in the Spark web UI. |
| SparkSession.Builder | **config**(**SparkConf** conf) |
| Sets a list of config options based on the given SparkConf. |
| SparkSession.Builder | config(String key, boolean value) |
| Sets a config option. |
| SparkSession.Builder | config(String key, double value) |
| Sets a config option. |
| SparkSession.Builder | config(String key, long value) |
| Sets a config option. |
| SparkSession.Builder | config(String key, String value) |
| Sets a config option. |
| SparkSession.Builder | enableHiveSupport() |
| Enables Hive support, including connectivity to a persistent Hive metastore, support for Hive serdes, and Hive user-defined functions. |
| SparkSession | getOrCreate() |
| Gets an existing SparkSession or, if there is no existing one,  creates a new one based on the options set in this builder. |
| SparkSession.Builder | master(String master) |
| Sets the Spark master URL to connect to, such as "local" to run locally, "local[4]" to run locally with 4 cores, or "spark://master:7077" to run on a Spark standalone cluster. |
| SparkSession.Builder | **withExtensions**(scala.Function1<**SparkSessionExtensions**,scala.runtime.BoxedUnit> f) |
| Inject extensions into the SparkSession. |

**Implicits.\_**

//implicits object gives [implicit conversions](https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-SparkSession-implicits.html#methods) for converting scala objects (incl. RDDs) into a Dataset, DataFrame, Columns

When converting from RDD to Data frames, we need some implicit functions.

For that we need to import implicits.\_

import sparkSession.sqlContext.implicits.\_

//Check the version of the Spark

spark.version

//Get a spark context out of SparkSession

var sc1=spark.sparkContext

//Get a hiveContext context

val sparkSession = SparkSession.builder.enableHiveSupport.getOrCreate()

**Dataframe (collection of data organized as structured)**

* A Spark DataFrame is a distributed collection of data organized into named columns that provides operations to filter, group, or compute aggregates, and can be used with Spark SQL.
* Here are set of few characteristic features of DataFrame −
* Ability to process the data in the size of Kilobytes to Petabytes on a single node cluster to large cluster.
* Supports different data formats (Avro, csv, elastic search, and Cassandra) and storage systems (HDFS, HIVE tables, mysql, etc).
* State of art optimization and code generation through the Spark SQL Catalyst optimizer (tree transformation framework).
* Can be easily integrated with all Big Data tools and frameworks via Spark-Core.
* Provides API for Python, Java, Scala, and R Programming.

**Setting log level to Error** sparkSession.sparkContext.setLogLevel("ERROR")

## Working with Dataframes

* Spark SQL introduces a tabular functional data abstraction called DataFrame.
* It is designed to ease developing Spark applications for processing large amount of structured tabular data on Spark infrastructure.
* A DataFrame is a distributed collection of data, which is organized into named columns.
* Conceptually, it is equivalent to relational tables with good optimization techniques.
* DataFrame provides a domain-specific language API for structured data manipulation,

with structured and semi-structured data.

* DataFrame is a collection of [rows](https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-Row.html) with a [schema](https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-schema.html) that is the result of executing a structured query (once it will have been executed).
* DataFrame uses the immutable, in-memory, resilient, distributed and parallel capabilities of [RDD](https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-rdd.adoc), and applies a structure called schema to the data.
* Data frame at the background will create RDDs to process the data in Spark.

|  |  |
| --- | --- |
| **Normal Class** | **Single ton object** |
| sparkContext()  sparkSQL()  sparkCore()  sparkHive() | SparkSession()  This object has instance of all the other class (listed in left side). |

* DataFrames can be constructed by the following ways:

1. Create dataframe using case class with toDF and createdataframe functions (**Reflection**)
2. create dataframe from collections type such as **structType** and fields
3. creating dataframe from read method using **modules** such as csv,json,parquet

## Inferring the Schema using Reflection

## create RDD 🡪 apply schema on it 🡪 convert it to Dataframe

1. **using case class**
2. Create case class as per the structure of data
3. Iterate on every line of rdd, split on the delimiter and apply the structure calling the case class (SchemaedRDD)
4. Convert the schemaedRDD to DF.
5. Ready to write DSL
6. register the DF to temp view.
7. Write ISO sql on top of the temp view created.

* Case classes are just regular classes that are: Immutable by default.
* Decomposable through pattern matching.
* Compared by structural equality instead of by reference.

case class Auction(auctionid: String, bid: Float, bidtime: Float, bidder: String, bidderrate:Integer, openbid: Float, price: Float, item: String, daystolive: Integer)

// load the data into an RDD

val auctionRDD = spark.sparkContext.textFile("file:///home/hduser/sparkdata/auctiondata")

// create an RDD of Auction objects

val ebay = auctionRDD.map(\_.split("~")).map(p => Auction(p(0), p(1).toFloat, p(2).toFloat, p(3), p(4).toInt, p(5).toFloat, p(6).toFloat, p(7), p(8).toInt))

// change ebay RDD of Auction objects to a DataFrame

val auction = spark.createDataFrame(ebay)

and now the RDD ebay is converted into a Dataframe.

1. **Using toDF() function**
2. Iterate on every line of rdd, split on the delimiter and make it tuples
3. Convert the schemaedRDD to DF using toDF() function.
4. In the toDF() function pass the structure as parameter.
5. Ready to write DSL
6. register the DF to temp view.
7. Write ISO sql on top of the temp view created.

val auctionRDD = spark.sparkContext.textFile("file:///home/hduser/sparkdata/auctiondata")

//in the below command, we split the above RDD delimited by “~” and then instead of using case class, we convert the splited RDD to data frame using toDF() function.

//While converting to dataframe using toDF() we pass the header of the data as parameter.

val ebay = auctionRDD.map(\_.split("~")).map(p => (p(0), p(1).toFloat, p(2).toFloat,p(3), p(4).toInt, p(5).toFloat, p(6).toFloat, p(7), p(8).toInt)).toDF("auctionid", "bid", "bidtime", "bidder", "bidderrate", "openbid", "price", "item", "daystolive")

## Create Dataframe programatically or using csv module

## Using spark.read dataFrameReader function, read the file directly based on its format.

//We can use (spark.read) function to read the data directly based on format.

//But reading the data with out any information will not have much information.

//as show below, we don’t have header /schema information

scala> val usdf = spark.read.csv("file:///home/hduser/sparkdata/usdata.csv")

usdf: org.apache.spark.sql.DataFrame = [\_c0: string, \_c1: string ... 11 more fields]

scala> usdf.printSchema()

root

|-- \_c0: string (nullable = true)

|-- \_c1: string (nullable = true)

|-- \_c2: string (nullable = true)

|-- \_c3: string (nullable = true)

|-- \_c4: string (nullable = true)

|-- \_c5: string (nullable = true)

|-- \_c6: string (nullable = true)

|-- \_c7: string (nullable = true)

|-- \_c8: string (nullable = true)

|-- \_c9: string (nullable = true)

|-- \_c10: string (nullable = true)

|-- \_c11: string (nullable = true)

|-- \_c12: string (nullable = true)

//instead of using just the module from read, if we use options() function we can make the data frame more informative.

val usschemadf = 🡪 this is using spark-csv module. We have spark modules for json,parquet etc

spark.read.option("header","true").option("inferschema","true").option("delimiter",",").csv("file:///home/hduser/sparkdata/usdata.csv")

or 🡪 // above command used the direct “CSV” module.

But the below one mentions format as CSV and loads the data using load function.

val usmodule = 🡪 this is using load function and format as “csv”

spark.read.option("header","true").option("inferschema","true").option("delimiter",",").foramt("csv").load("file:///home/hduser/sparkdata/usdata.csv")

scala> usschemadf.printSchema() or scaka> usmodule.printSchema()

root

|-- first\_name: string (nullable = true)

|-- last\_name: string (nullable = true)

|-- company\_name: string (nullable = true)

|-- address: string (nullable = true)

|-- city: string (nullable = true)

|-- county: string (nullable = true)

|-- state: string (nullable = true)

|-- zip: integer (nullable = true)

|-- age: integer (nullable = true)

|-- phone1: string (nullable = true)

|-- phone2: string (nullable = true)

|-- email: string (nullable = true)

|-- web: string (nullable = true)

## using STRUCT TYPE:

// StructType objects define the schema of Spark DataFrames. StructType objects contain a list of StructField objects that define the name, type, and nullable flag for each column in a DataFrame.

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

val custschema =

StructType(Array(StructField("first\_name", StringType,true)

,StructField("last\_name", StringType, true)

,StructField("company\_name", StringType,true)

,StructField("address", StringType, true)

,StructField("city", StringType,true)

,StructField("country", StringType, true)

,StructField("state", StringType,true)

,StructField("zip", StringType, true)

,StructField("age", IntegerType,true)

,StructField("phone1", StringType, true)

,StructField("phone2", StringType,true)

,StructField("email", StringType, true)

,StructField("website", StringType, true))

);

val uscsvdf1 = spark.read.option("delimiter",",").schema(custschema).csv("file:///home/hduser/sparkdata/usdata.csv")

scala> uscsvdf1.printSchema

root

|-- first\_name: string (nullable = true)

|-- last\_name: string (nullable = true)

|-- company\_name: string (nullable = true)

|-- address: string (nullable = true)

|-- city: string (nullable = true)

|-- country: string (nullable = true)

|-- state: string (nullable = true)

|-- zip: string (nullable = true)

|-- age: integer (nullable = true)

|-- phone1: string (nullable = true)

|-- phone2: string (nullable = true)

|-- email: string (nullable = true)

|-- website: string (nullable = true)

**Now we have seen the different ways of creating data frames.. Now lets see how we can do DSL and SQL operations on it.**

1. select & show

scala> **val auctShow = auction.select("bidder").show()**

+--------------+

| bidder|

+--------------+

| jake7870|

| davidbresler2|

|gladimacowgirl|

| daysrus|

...........

....

1. distinct & count

**val count = auction.select("auctionid").distinct.count()**

1. groupBy / agg / max / sum / alias / sort

import org.apache.spark.sql.functions.avg

val grpbid = auction.groupBy("item").agg(max("bid").alias("max\_bid"),sum("price").alias("sp")).sort($"sp".desc)

or

val grpbid = auction.groupBy("item").agg(max("bid").alias("max\_bid"),sum("price").alias("sp")).orderBy($"sp".desc)

1. Collect()

scala> **grpbid.collect**

res1: Array[org.apache.spark.sql.Row] = Array([cartier,5400.0,1806618.5611763], [palm,290.0,1367597.7882232666], [xbox,501.77,401664.219581604])

1. collectAsList()

scala> **grpbid.collectAsList**

res9: java.util.List[org.apache.spark.sql.Row] = [[cartier,5400.0,1806618.5611763], [palm,290.0,1367597.7882232666], [xbox,501.77,401664.219581604]]

1. where()

scala> val grpbidcartier = auction.where('item === "cartier").groupBy("item").agg(sum("price").alias("sp"))

1. filter()

scala> val grpbidcartier = auction.filter('item === "cartier").groupBy("item").agg(sum("price").alias("sp"))

grpbidcartier: org.apache.spark.sql.DataFrame = [item: string, sp: double]

scala> grpbidcartier.collect

res10: Array[org.apache.spark.sql.Row] = Array([cartier,1806618.5611763])

1. printSchema

auction.printSchema

scala> auction.printSchema

root

|-- auctionid: string (nullable = true)

|-- bid: float (nullable = false)

|-- bidtime: float (nullable = false)

|-- bidder: string (nullable = true)

|-- bidderrate: integer (nullable = true)

|-- openbid: float (nullable = false)

|-- price: float (nullable = false)

|-- item: string (nullable = true)

|-- daystolive: integer (nullable = true)

grpbid.printSchema

root

|-- item: string (nullable = true)

|-- max\_bid: float (nullable = true)

|-- sp: double (nullable = true)

1. Show()

scala> grpbid.show(1,false)

+-------+-------+---------------+

|item |max\_bid|sp |

+-------+-------+---------------+

|cartier|5400.0 |1806618.5611763|

+-------+-------+---------------+

only showing top 1 row

1. limit()

scala> auction.show(5,true)

or

scala> auction.limit(5).show()

+----------+-----+--------+--------------+----------+-------+-----+----+----------+

| auctionid| bid| bidtime| bidder|bidderrate|openbid|price|item|daystolive|

+----------+-----+--------+--------------+----------+-------+-----+----+----------+

|8213034705| 95.0|2.927373| jake7870| 0| 95.0|117.5|xbox| 3|

|8213034705|115.0|2.943484| davidbresler2| 1| 95.0|117.5|xbox| 3|

|8213034705|100.0|2.951285|gladimacowgirl| 58| 95.0|117.5|xbox| 3|

|8213034705|117.5|2.998947| daysrus| 10| 95.0|117.5|xbox| 3|

|8213060420| 2.0|0.065266| donnie4814| 5| 1.0|120.0|xbox| 3|

+----------+-----+--------+--------------+----------+-------+-----+----+----------+

only showing top 5 rows

1. head() / take()

scala> auction.head(3)

or

scala> auction.take(3)

res16: Array[org.apache.spark.sql.Row] = Array([8213034705,95.0,2.927373,jake7870,0,95.0,117.5,xbox,3], [8213034705,115.0,2.943484,davidbresler2,1,95.0,117.5,xbox,3], [8213034705,100.0,2.951285,gladimacowgirl,58,95.0,117.5,xbox,3])

1. takeAsList()

scala> auction.takeAsList(3)

res21: java.util.List[org.apache.spark.sql.Row] = [[8213034705,95.0,2.927373,jake7870,0,95.0,117.5,xbox,3], [8213034705,115.0,2.943484,davidbresler2,1,95.0,117.5,xbox,3], [8213034705,100.0,2.951285,gladimacowgirl,58,95.0,117.5,xbox,3]]

1. Join() 🡪 copy/paste the below table format output into notepad to understand better.

* Here how it works is,
* First row from left table will be joined with all rows in second table.
* If first table has 2 rows and second has 4, then output will have 8 rows.
* 1st rows from table 1 mapped with all 4 rows in second table.
* Then 2nd row from table 1 again mapped with all 4 rows in second table.

scala> val auct1 = auction.limit(2)

auct1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [auctionid: string, bid: float ... 7 more fields]

scala> **auct1.show()**

+----------+-----+--------+-------------+----------+-------+-----+----+----------+

| auctionid| bid| bidtime| bidder|bidderrate|openbid|price|item|daystolive|

+----------+-----+--------+-------------+----------+-------+-----+----+----------+

|8213034705| 95.0|2.927373| jake7870| 0| 95.0|117.5|xbox| 3|

|8213034705|115.0|2.943484|davidbresler2| 1| 95.0|117.5|xbox| 3|

+----------+-----+--------+-------------+----------+-------+-----+----+----------+

scala> val auct2 = auction.limit(4)

auct2: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [auctionid: string, bid: float ... 7 more fields]

scala> auct2.show()

+----------+-----+--------+--------------+----------+-------+-----+----+----------+

| auctionid| bid| bidtime| bidder|bidderrate|openbid|price|item|daystolive|

+----------+-----+--------+--------------+----------+-------+-----+----+----------+

|8213034705| 95.0|2.927373| jake7870| 0| 95.0|117.5|xbox| 3|

|8213034705|115.0|2.943484| davidbresler2| 1| 95.0|117.5|xbox| 3|

|8213034705|100.0|2.951285|gladimacowgirl| 58| 95.0|117.5|xbox| 3|

|8213034705|117.5|2.998947| daysrus| 10| 95.0|117.5|xbox| 3|

+----------+-----+--------+--------------+----------+-------+-----+----+----------+

scala> **val auctjoin = auct1.join(auct2)**

auctjoin: org.apache.spark.sql.DataFrame = [auctionid: string, bid: float ... 16 more fields]

scala> auctjoin.show()

+----------+-----+--------+-------------+----------+-------+-----+----+----------+----------+-----+--------+--------------+----------+-------+-----+----+----------+

| auctionid| bid| bidtime| bidder|bidderrate|openbid|price|item|daystolive| auctionid| bid| bidtime| bidder|bidderrate|openbid|price|item|daystolive|

+----------+-----+--------+-------------+----------+-------+-----+----+----------+----------+-----+--------+--------------+----------+-------+-----+----+----------+

|8213034705| 95.0|2.927373| jake7870| 0| 95.0|117.5|xbox| 3|8213034705| 95.0|2.927373| jake7870| 0| 95.0|117.5|xbox| 3|

|8213034705| 95.0|2.927373| jake7870| 0| 95.0|117.5|xbox| 3|8213034705|115.0|2.943484| davidbresler2| 1| 95.0|117.5|xbox| 3|

|8213034705| 95.0|2.927373| jake7870| 0| 95.0|117.5|xbox| 3|8213034705|100.0|2.951285|gladimacowgirl| 58| 95.0|117.5|xbox| 3|

|8213034705| 95.0|2.927373| jake7870| 0| 95.0|117.5|xbox| 3|8213034705|117.5|2.998947| daysrus| 10| 95.0|117.5|xbox| 3|

|8213034705|115.0|2.943484|davidbresler2| 1| 95.0|117.5|xbox| 3|8213034705| 95.0|2.927373| jake7870| 0| 95.0|117.5|xbox| 3|

|8213034705|115.0|2.943484|davidbresler2| 1| 95.0|117.5|xbox| 3|8213034705|115.0|2.943484| davidbresler2| 1| 95.0|117.5|xbox| 3|

|8213034705|115.0|2.943484|davidbresler2| 1| 95.0|117.5|xbox| 3|8213034705|100.0|2.951285|gladimacowgirl| 58| 95.0|117.5|xbox| 3|

|8213034705|115.0|2.943484|davidbresler2| 1| 95.0|117.5|xbox| 3|8213034705|117.5|2.998947| daysrus| 10| 95.0|117.5|xbox| 3|

+----------+-----+--------+-------------+----------+-------+-----+----+----------+----------+-----+--------+--------------+----------+-------+-----+----+----------+

1. UnionAll: (just horizontal merging of data)

scala> val auct1 = auction.limit(2)

auct1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [auctionid: string, bid: float ... 7 more fields]

scala> val auct2 = auction.limit(4)

auct2: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [auctionid: string, bid: float ... 7 more fields]

scala> val auctunion = auct1.unionAll(auct2)

warning: there was one deprecation warning; re-run with -deprecation for details

auctunion: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [auctionid: string, bid: float ... 7 more fields]

scala> auctunion.show()

+----------+-----+--------+--------------+----------+-------+-----+----+----------+

| auctionid| bid| bidtime| bidder|bidderrate|openbid|price|item|daystolive|

+----------+-----+--------+--------------+----------+-------+-----+----+----------+

|8213034705| 95.0|2.927373| jake7870| 0| 95.0|117.5|xbox| 3|

|8213034705|115.0|2.943484| davidbresler2| 1| 95.0|117.5|xbox| 3|

|8213034705| 95.0|2.927373| jake7870| 0| 95.0|117.5|xbox| 3|

|8213034705|115.0|2.943484| davidbresler2| 1| 95.0|117.5|xbox| 3|

|8213034705|100.0|2.951285|gladimacowgirl| 58| 95.0|117.5|xbox| 3|

|8213034705|117.5|2.998947| daysrus| 10| 95.0|117.5|xbox| 3|

+----------+-----+--------+--------------+----------+-------+-----+----+----------+

1. System.out.println()

System.out.println(count)

1. //Stores the output in JSON format

auction.write.mode("overwrite").json("file:/home/hduser/sparkdata/auctiondata.json");

auctunion.write.mode("overwrite").json("file:/home/hduser/sparkdata/auctunion.json");

**Now lets see how to perform SQL operation on the dataframe.**

* To write a SQL query on a data frame, we need to create a tempview using that dataframe.
* createOrReplaceTempView() function is used to create temp view on top of a DF.

val usdataDF = spark.read.option("header","true").option("inferschema","true").option("delimiter",",").csv("file:///home/hduser/sparkdata/usdata.csv")

scala> usdataDF.createOrReplaceTempView("custinfo")

scala> spark.catalog.listDatabases.show(10,false);

+----------+---------------------+--------------------------------------------------------+

|name |description |locationUri |

+----------+---------------------+--------------------------------------------------------+

|custdb |null |hdfs://localhost:54310/user/hive/warehouse/custdb.db |

|default |Default Hive database|hdfs://localhost:54310/user/hive/warehouse |

|kjpractice|null |hdfs://localhost:54310/user/hive/warehouse/kjpractice.db|

|retail |null |hdfs://localhost:54310/user/hive/warehouse/retail.db |

|retail\_stg|null |hdfs://localhost:54310/user/hive/warehouse/retail\_stg.db|

+----------+---------------------+--------------------------------------------------------+

scala> spark.catalog.listTables.show(10,false);

+--------+--------+-----------+---------+-----------+

|name |database|description|tableType|isTemporary|

+--------+--------+-----------+---------+-----------+

|game |default |null |MANAGED |false |

|custinfo|null |null |TEMPORARY|true |

+--------+--------+-----------+---------+-----------+

* The meta data of SQL objects of dataframe are derby data base by default
* But in real time it will be stored in some remote data base.
* In our example its stored in MySQL. As show in the above highlighted .

spark.catalog.listTables.show(10,false);

sqlctx.sql("describe custinfo").show(10,false)

sql("select concat(first\_name,' ',last\_name),company\_name,address,city,phone1 from custinfo where upper(company\_name) like 'CH%'").show(10,false)

sql("select count(1),city from custinfo group by city having count(1) >1 order by city").show()

sql("select count(1) as cnt,city from custinfo group by city having count(1)> 5 order by

city").show()

val agecat = sqlctx.sql("select distinct age,case when age <=10 then 'childrens' when age >10

and age < 20 then 'teen' else 'others' end as agecat from custinfo order by agecat")

scala> agecat.show()

+---+---------+

|age| agecat|

+---+---------+

| 9|childrens|

| 10|childrens|

| 7|childrens|

| 8|childrens|

| 22| others|

| 20| others|

| 34| others|

| 55| others|

| 43| others|

| 25| others|

| 33| others|

| 21| others|

| 11| teen|

| 16| teen|

| 19| teen|

| 12| teen|

| 15| teen|

| 14| teen|

+---+---------+

**//Creating and registering functions:**

def addfunc (a:Int,b:Int):Int=

{

return a+b

}

import org.apache.spark.sql.functions.udf

spark.udf.register("addudf",addfunc \_) spark.catalog.listFunctions.filter('name like "%addu%").show(false)

spark.sql("SELECT addudf(20,30) FROM custinfo ").show()

//Stores the output in Parquet and orc format, Parquet files are self-describing so the schema is preserved

agecat.write.mode("overwrite").parquet("file:/home/hduser/sparkdata/agecategory.parque t");

agecat.write.mode("append").orc("file:/home/hduser/sparkdata/agecategory1.orc");

// Read in the parquet file created above

// Parquet files are self-describing so the schema is preserved

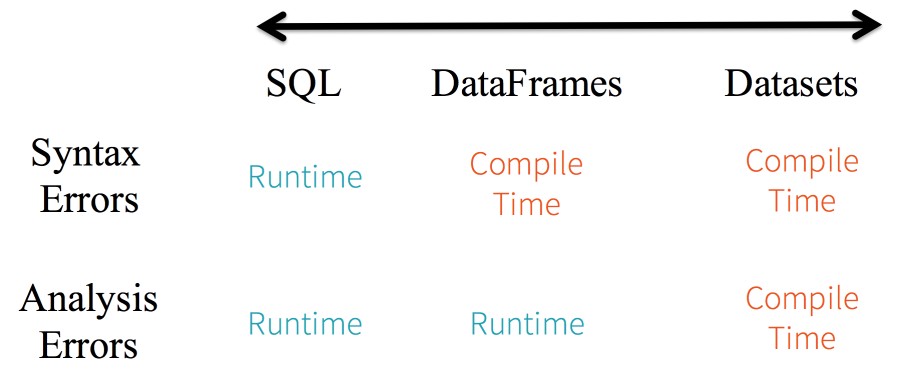
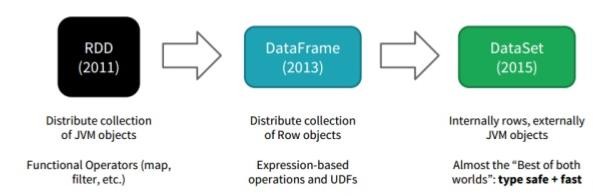
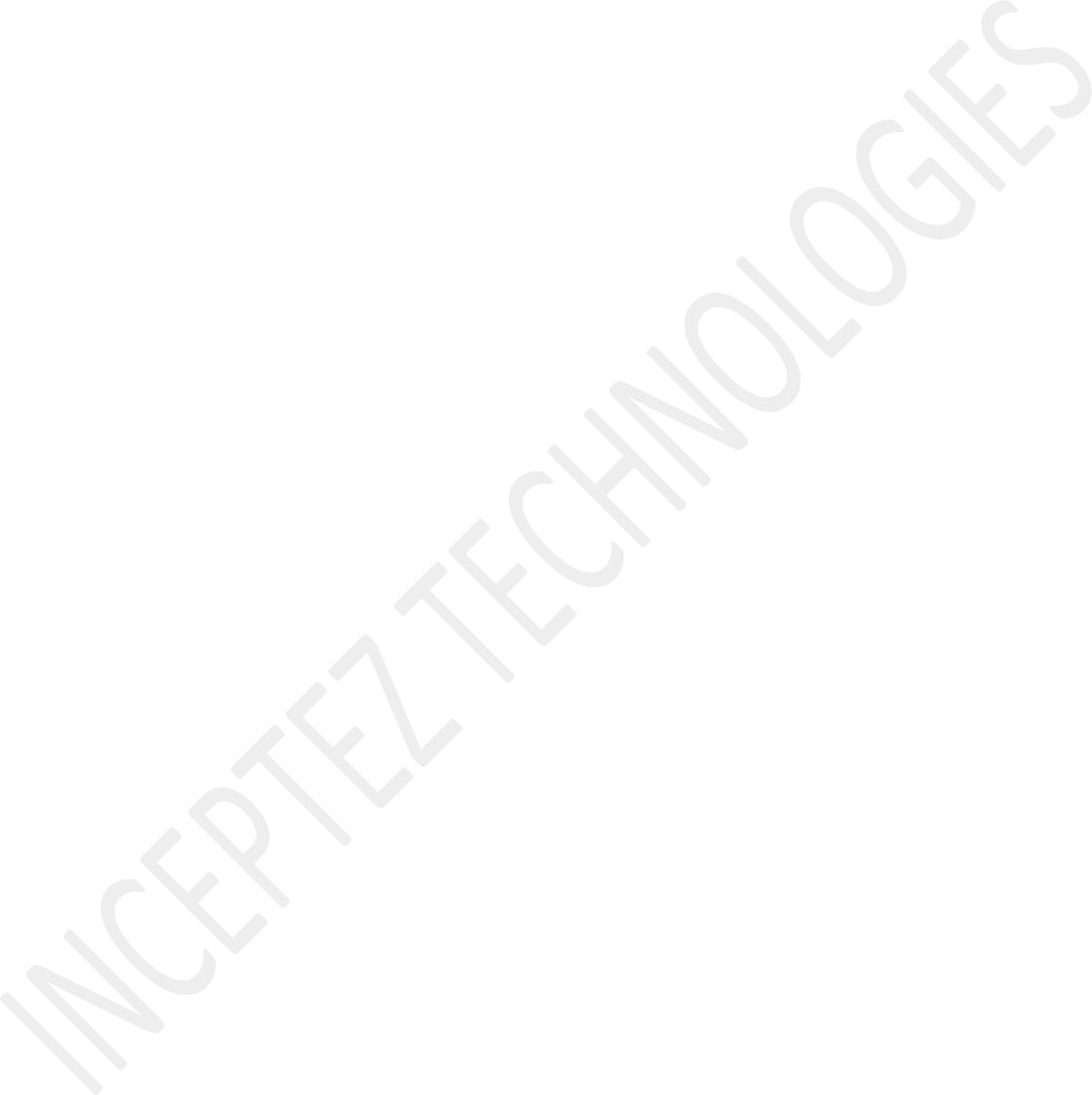
// The result of loading a Parquet file is also a DataFrame

val parquetagecatdf = spark.read.parquet("file:/home/hduser/sparkdata/agecategory.parquet")

// Parquet files can also be used to create a temporary view and then used in SQL statements

parquetagecatdf.createOrReplaceTempView("parquetagecat")

spark.sql("SELECT max(age),min(age),count(distinct agecat) FROM parquetagecat ").show()



Dataset Vs Dataframe

A Dataset is a strongly typed collection of domain-specific objects (like object of type auction

declared below) that can be transformed in parallel using functional or relational operations. Each Dataset also has an untyped view called a DataFrame, which is a Dataset of Row.

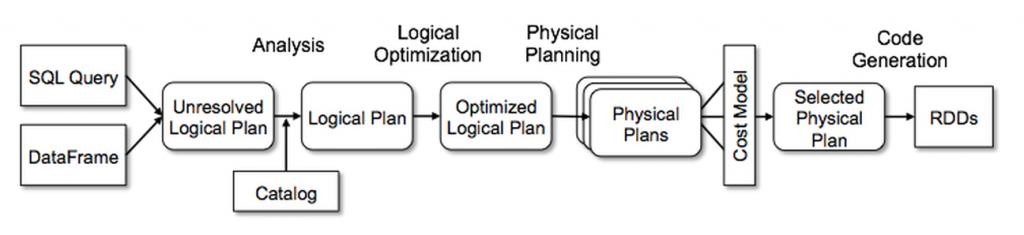
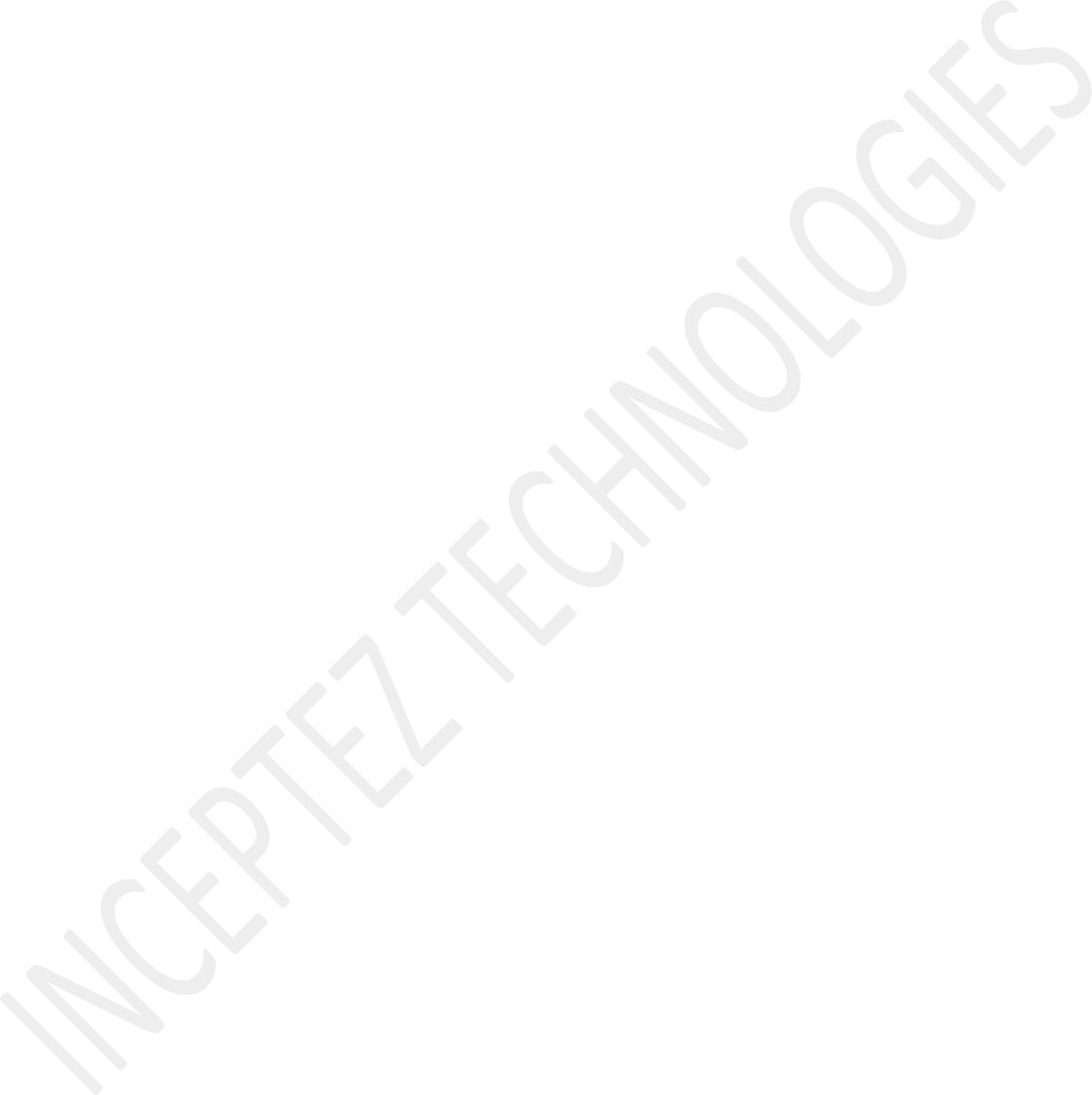
Dataframe is merged with Dataset API. So we can use any method available for dataframe in datasets.

In summation, the choice of when to use RDD or DataFrame and/or Dataset seems obvious. While the former offers you low-level functionality and control, the latter allows custom view and structure, offers high-level and domain specific operations, saves space, and executes at superior speeds.

To simplify Spark for developers, how to optimize and make it performant it was decided to elevate the low-level RDD APIs to a high-level abstraction as DataFrame and Dataset and to build this unified data abstraction across libraries a top [Catalyst optimizer](https://databricks.com/glossary/catalyst-optimizer) and Tungsten.

The goal of Project Tungsten is to improve Spark execution by optimizing Spark jobs for CPU and memory efficiency (as opposed to network and disk I/O which are considered fast enough).

* Tungsten includes specialized in-memory data structures tuned for the type of operations required by Spark, improved code generation, and a specialized wire protocol.



* As Tungsten does not depend on Java objects, both on-heap and off-heap allocations are supported. Since Tungsten no longer depends on working with Java objects, you can use either on-heap (in the JVM) or off-heap storage.
* Tungsten will not do deserialization when processing data, for example of this is with sorting, a common and expensive operation using tungsten this can be done without having to deserialize the data again.
* By avoiding the memory and GC overhead of regular Java objects, Tungsten is able to process larger data sets than the same hand-written aggregations.

## Catalyst Optimizer:

Catalyst supports both rule-based and cost-based optimization.

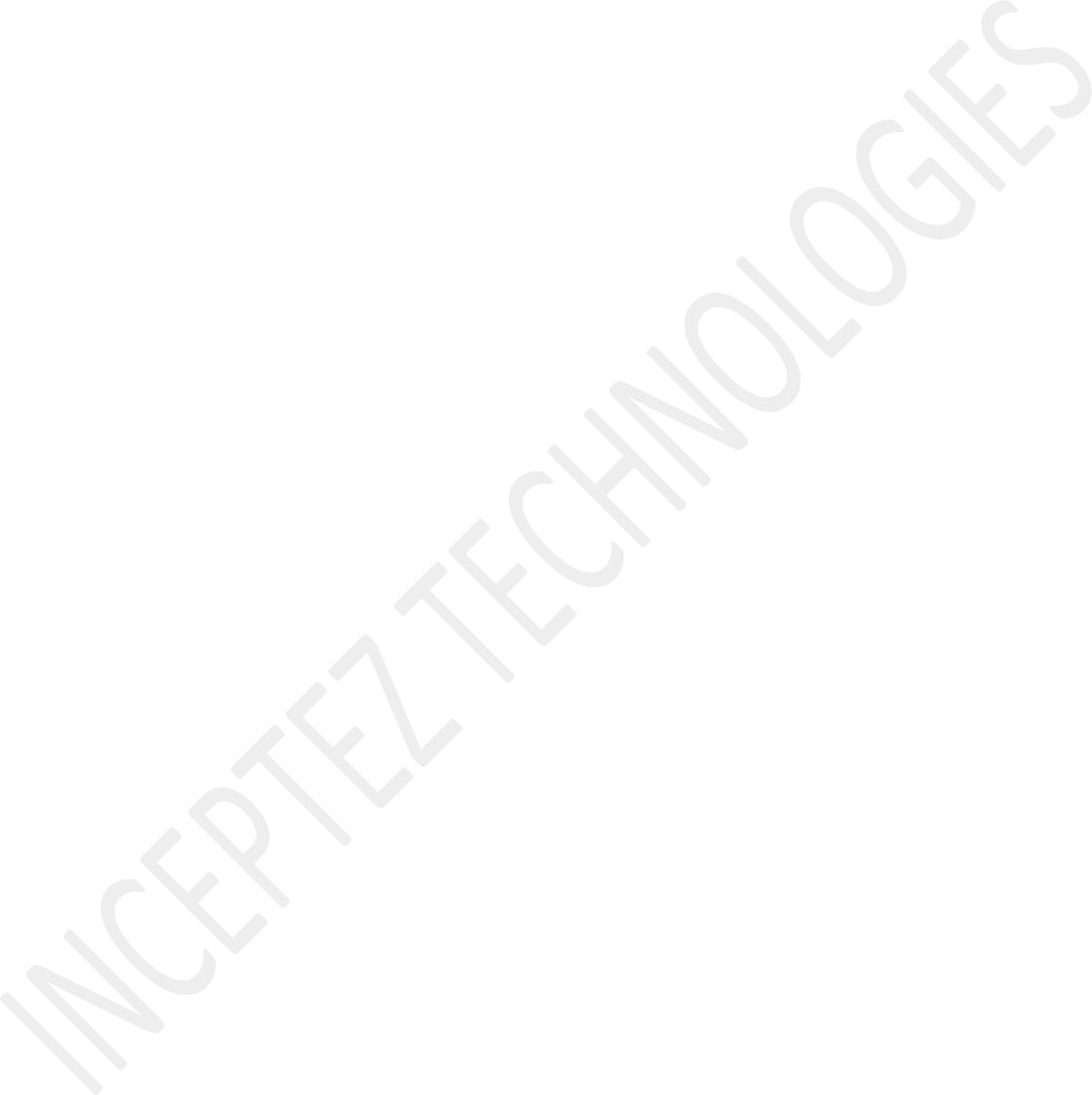
At its core, Catalyst contains a general library for representing trees and applying rules to manipulate them. On top of this framework, we have built libraries specific to relational query processing (e.g., expressions, logical query plans), and several sets of rules that handle different phases of query execution: analysis, logical optimization, physical planning, and code generation to compile parts of queries to Java bytecode. For the latter, we use another Scala feature, [quasiquotes](http://docs.scala-lang.org/overviews/quasiquotes/intro.html), that makes it easy to generate code at runtime from composable expressions. Finally, Catalyst offers several public extension points, including external data sources and user-defined types.

Pick one—DataFrames and/or Dataset or RDDs APIs—that meets your needs and use-case

// A JSON dataset is pointed to a path. The path can be either a single text file or a directory storing text files

//weakly typed Dataframes

hadoop fs -put ~/sparkdata/auctiondata.json

val auctionjson = "/user/hduser/auctiondata.json"; val auctionjsonDF = spark.read.json(auctionjson); auctionjsonDF.distinct.show

//Strongly typed DataSets case class auctionclass

(auctionid:String,bid:Double,bidder:Int,bidderrate:Long,bidtime:Doubl e,daystolive:Long,item:String,openbid:Double,price:Double)

val auctionjson = "file:/home/hduser/sparkdata/auctiondata.json"; val auctionjsonDS = spark.read.json(auctionjson).as[auctionclass]; case class auctionclass

(auctionid:String,bid:Double,bidder:String,bidderrate:Long,bidtime:Do

uble,daystolive:Long,item:String,openbid:Double,price:Double) val auctionjsonDS = spark.read.json(auctionjson).as[auctionclass];

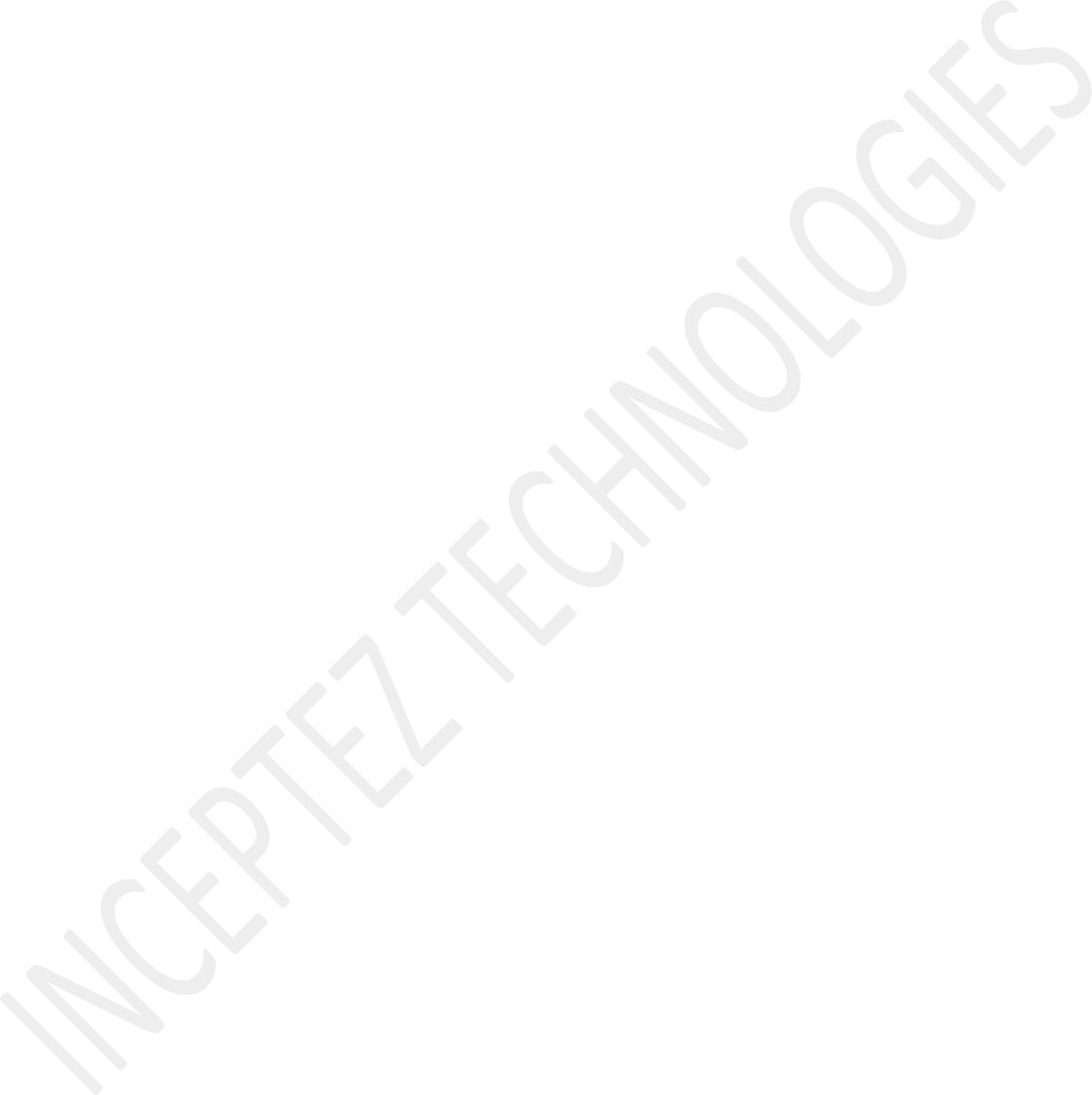
// The inferred schema can be visualized using the printSchema() method auctionjsonDF.printSchema();

auctionjsonDS.printSchema();

// Creates a temporary view using the DataSet auctionjsonDS.createOrReplaceTempView("auctionjsontable");

val auctionquery = spark.sql("SELECT \* FROM auctionjsontable") auctionquery.show();

## Hive operations:



import spark.implicits.\_

import spark.sql

//Initialize hive context wrapping spark context

~~val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)~~

//Get a hiveContext context

val sparkSession =

SparkSession.builder.enableHiveSupport.getOrCreate()

//Create a hive table

sql("create database if not exists sparkdb")

sql("use sparkdb")

sql("DROP TABLE IF EXISTS employee ")

sql("CREATE TABLE IF NOT EXISTS employee(id INT, name STRING, age INT) ROW FORMAT

DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'")

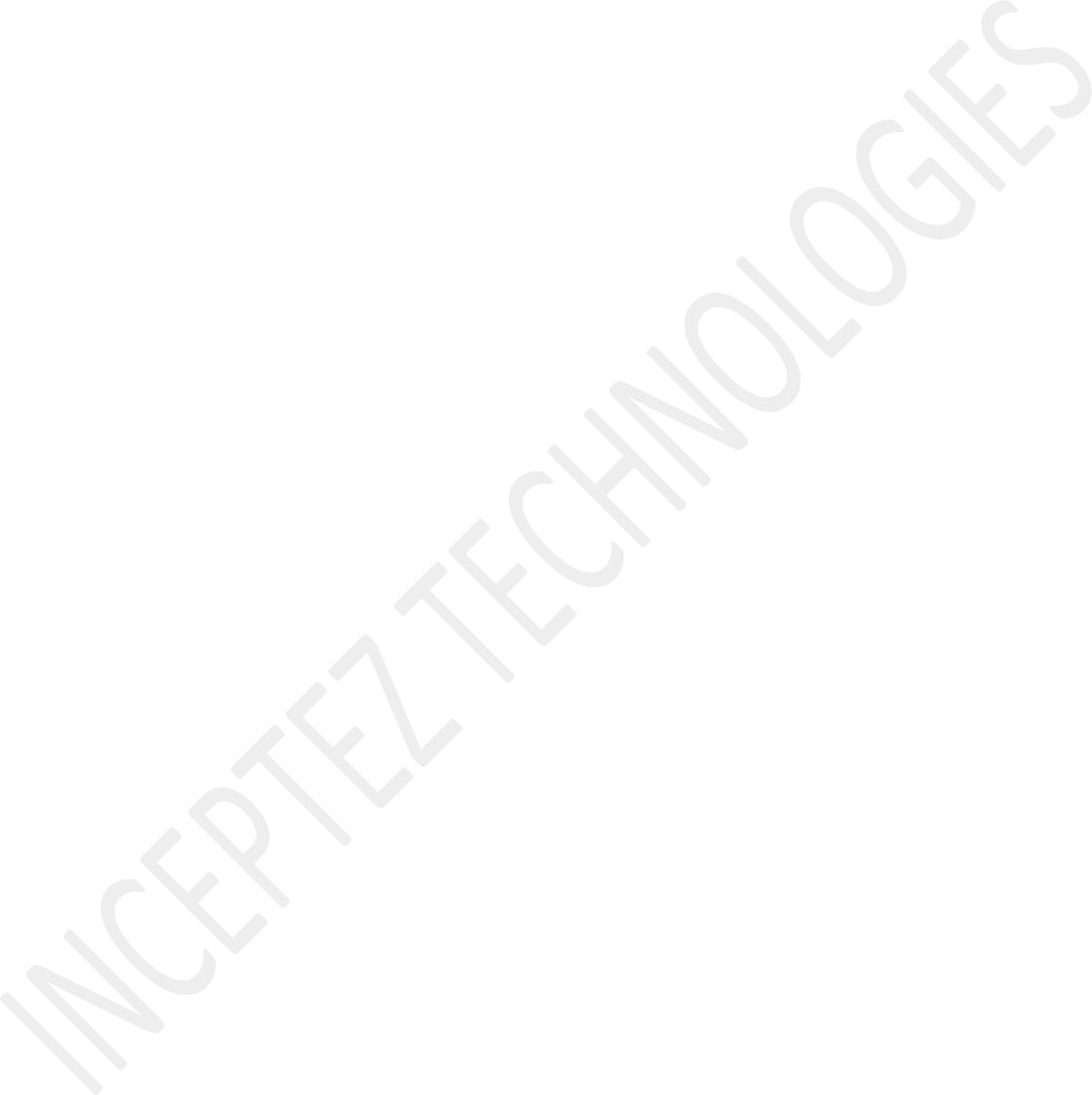
spark.catalog.listDatabases.show(10,false);

spark.catalog.listTables.show(10,false);

//Load data

sql("LOAD DATA LOCAL INPATH '/home/hduser/sparkdata/employee.txt' INTO TABLE employee")

//View data

val results = sql("FROM employee SELECT id, name, age") results.show()

## Additional Hive Usecases (For self practices):

//Drop table

sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'")

sql("LOAD DATA LOCAL INPATH '/home/hduser/sparkdata/sampledata' OVERWRITE INTO TABLE src")

sql("SELECT \* FROM src").show() sql("drop table src")

//Write the table content into the given hdfs location in orc format

results.write.format("orc").save("hdfs:/user/hduser/emp\_orc")

//Read the table from the given hdfs location in orc format and create a dataset out of it. case class empclass (id :Int,name :String,age:Int);

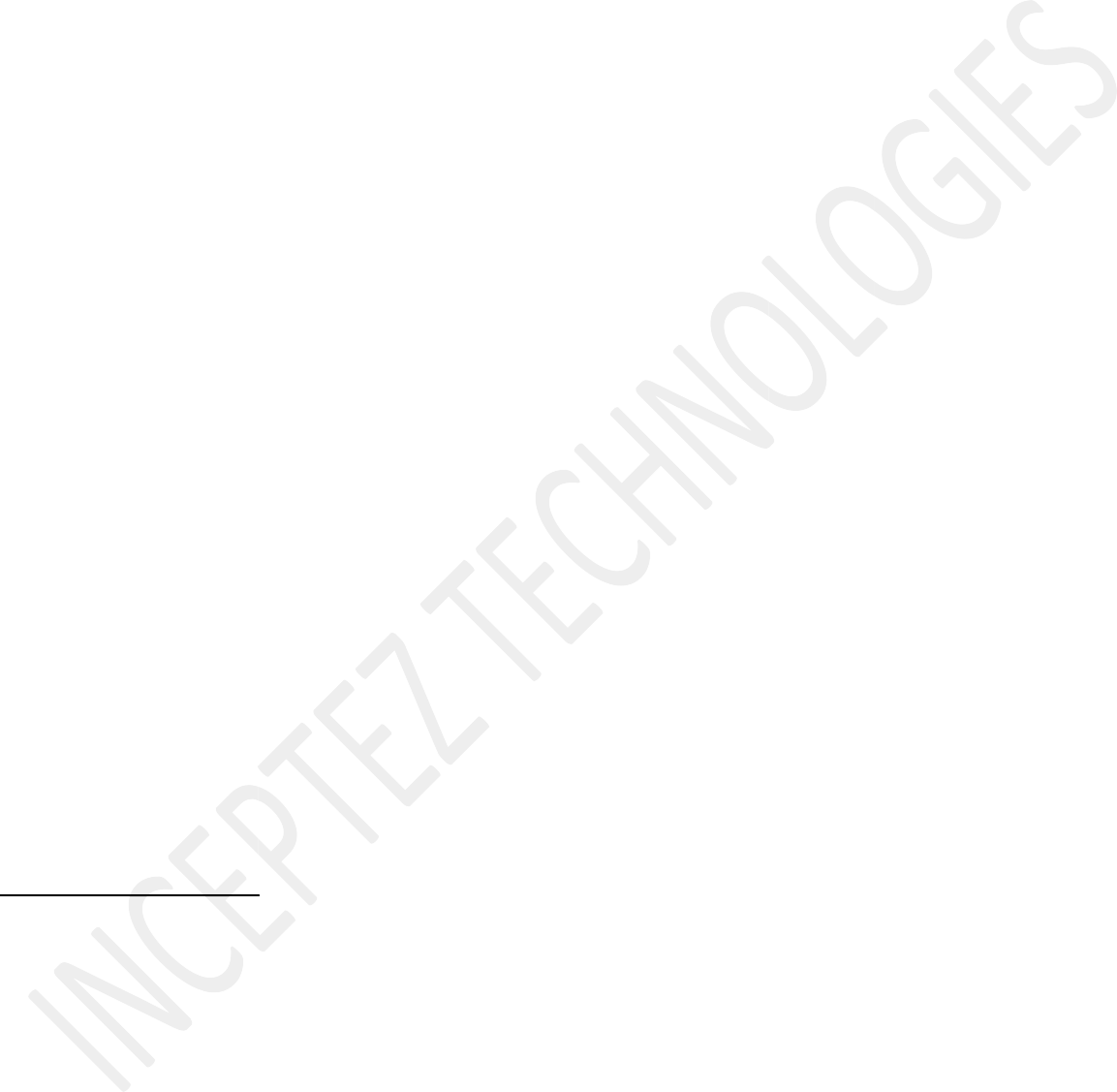
val empdata\_orc = spark.read.format("orc").load("hdfs:/user/hduser/emp\_orc").as[empclass];

empdata\_orc.createOrReplaceTempView("orcdata") sql("SELECT \* from orcdata").collect.foreach(println)

## Hive Partitioning:

sql("""create table txnrecords(txnno INT, txndate STRING, custno INT, amount DOUBLE, category STRING, product STRING, city STRING, state STRING, spendby STRING) row format delimited fields terminated by ','

lines terminated by '\n' stored as textfile""")

sql("LOAD DATA LOCAL INPATH '/home/hduser/hive/data/txns' OVERWRITE INTO TABLE txnrecords")

sql("select \* from txnrecords order by 1 limit 10").show

sql ("""create external table exttxnrecsByCat(txnno INT, txndate STRING, custno INT, amount DOUBLE, product STRING, city STRING, state STRING, spendby STRING)

partitioned by (category STRING) row format delimited fields terminated by ','

stored as textfile

location '/user/hive/warehouse/exttxnrecsbycat'""")

sql("""Insert into table exttxnrecsbycat partition (category='Games') select txnno,txndate,custno,amount,product,city,state,spendby from txnrecords

where category='Games'""")

sql("select count(1) from exttxnrecsbycat").show

## Adding UDFS in hive

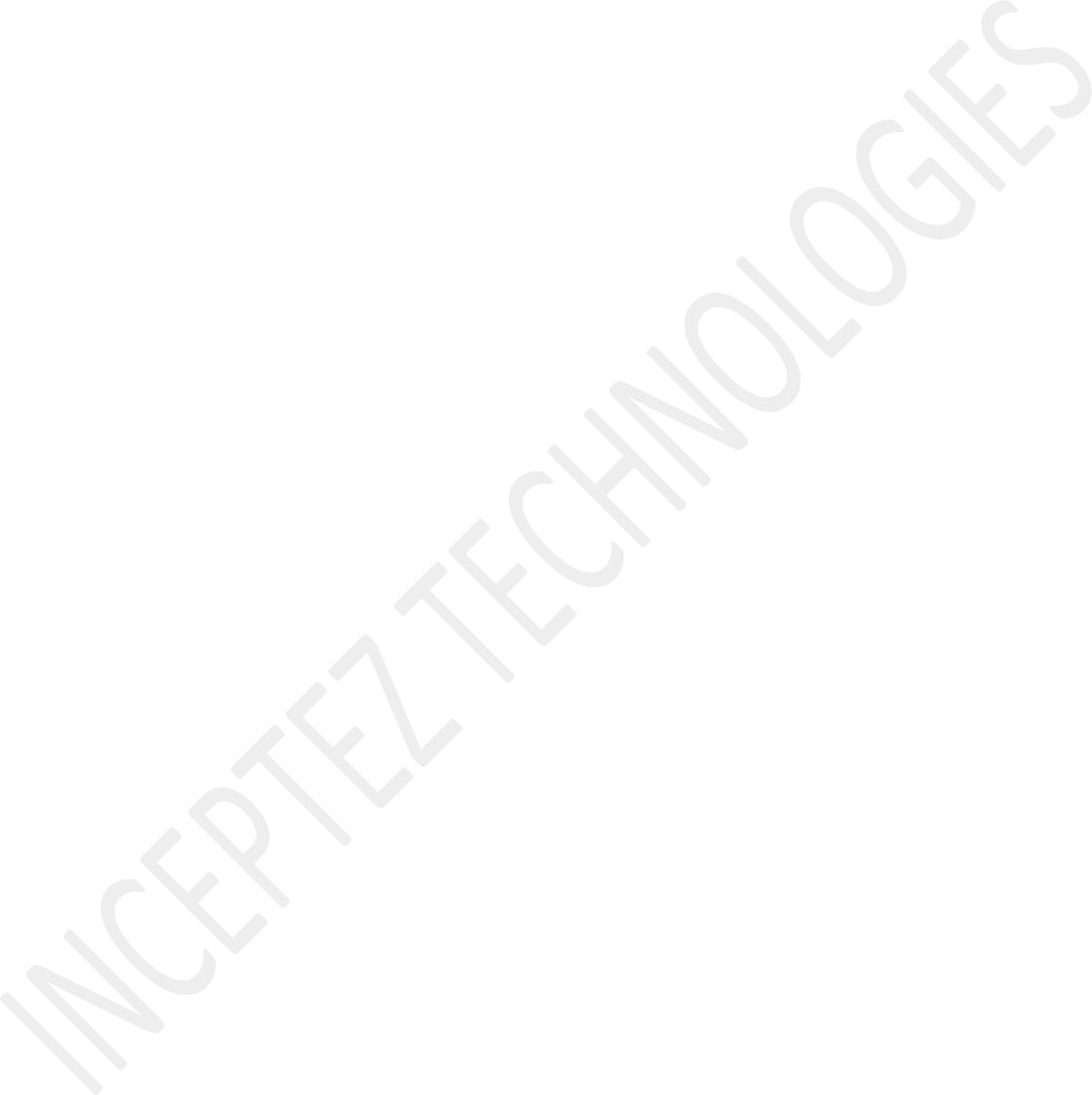
spark-shell --jars /home/hduser/hive/replaceword.jar sql("use sparkdb")

sql("create function repwords as 'inceptez.training.replacewords.replaceword'")

sql("select repwords(product,'Archery','Arc') from txnrecords where product='Archery' limit 10")

# Supported Hive Features

Spark SQL supports the vast majority of Hive features, such as:

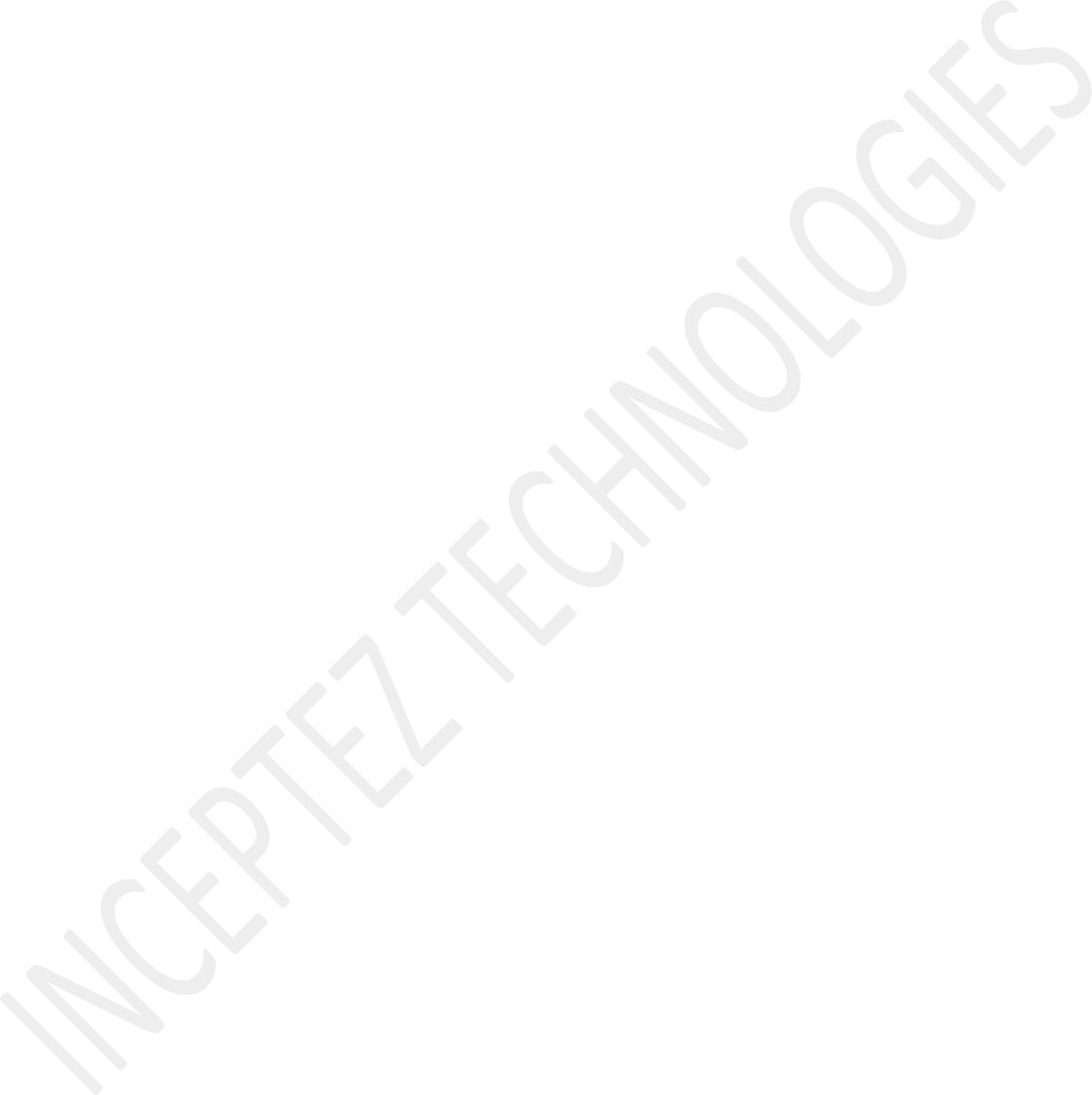
* Hive query statements, including:
  + SELECT
  + GROUP BY
  + ORDER BY
  + CLUSTER BY
  + SORT BY
* All Hive operators, including:
  + Relational operators (=, ⇔, ==, <>, <, >, >=, <=, etc)
  + Arithmetic operators (+, -, \*, /, %, etc)
  + Logical operators (AND, &&, OR, ||, etc)
  + Complex type constructors
  + Mathematical functions (sign, ln, cos, etc)
  + String functions (instr, length, printf, etc)
* User defined functions (UDF)
* User defined aggregation functions (UDAF)
* User defined serialization formats (SerDes)
* Window functions
* Joins
* Unions
* Sub-queries
  + SELECT col FROM ( SELECT a + b AS col from t1) t2
* Sampling
* Explain
* Partitioned tables including dynamic partition insertion
* View
* All Hive DDL Functions, including:
  + CREATE TABLE
  + CREATE TABLE AS SELECT
  + ALTER TABLE
* Most Hive Data types, including:
  + TINYINT
  + SMALLINT
  + INT
  + BIGINT
  + BOOLEAN
  + FLOAT
  + DOUBLE
  + STRING
  + BINARY
  + TIMESTAMP
  + DATE
  + ARRAY<>
  + MAP<>
  + STRUCT<>

# Unsupported Hive Functionality

Below is a list of Hive features that we don’t support yet. Most of these features are rarely used in Hive deployments.

**Major Hive Features**

* Tables with buckets: bucket is the hash partitioning within a Hive table partition. Spark SQL doesn’t support buckets yet.



**Hive Input/Output Formats**

* File format for CLI: For results showing back to the CLI, Spark SQL only supports TextOutputFormat.
* Hadoop archive

**Hive Optimizations**

A handful of Hive optimizations are not yet included in Spark. Some of these (such as indexes) are less

important due to Spark SQL’s in-memory computational model. Others are slotted for future releases of Spark SQL.

* Block level bitmap indexes and virtual columns (used to build indexes)
* Automatically determine the number of reducers for joins and groupbys: Currently in Spark SQL, you need to control the degree of parallelism post-shuffle using “SET spark.sql.shuffle.partitions=[num\_tasks];”.
* Meta-data only query: For queries that can be answered by using only meta data, Spark SQL still launches tasks to compute the result.
* STREAMTABLE hint in join: Spark SQL does not follow the STREAMTABLE hint.
* Merge multiple small files for query results: if the result output contains multiple small files, Hive can optionally merge the small files into fewer large files to avoid overflowing the HDFS metadata. Spark SQL does not support that.