SPARK SQL

If you don't drive your business,
you will be driven out of business

SparkSQL Features

- Spark SQL is intended to work with structured & semi structured data.
- Spark SQL is added in Spark 1.0.
- Spark SQL provides Spark with information about the structure of both the data and the computation being performed.
- Supports both basic SQL syntax or HiveQL.
- It can also read data from an existing Hive Tables
- Spark SQL Components
  - ✓ Query Optimizer
  - ✓ Spark SQL Core
    - Execution of queries as RDDs
    - Reading in Parquet, JSON …
  - ✓ Hive Support
    - HQL, MetaStore

About Data frames

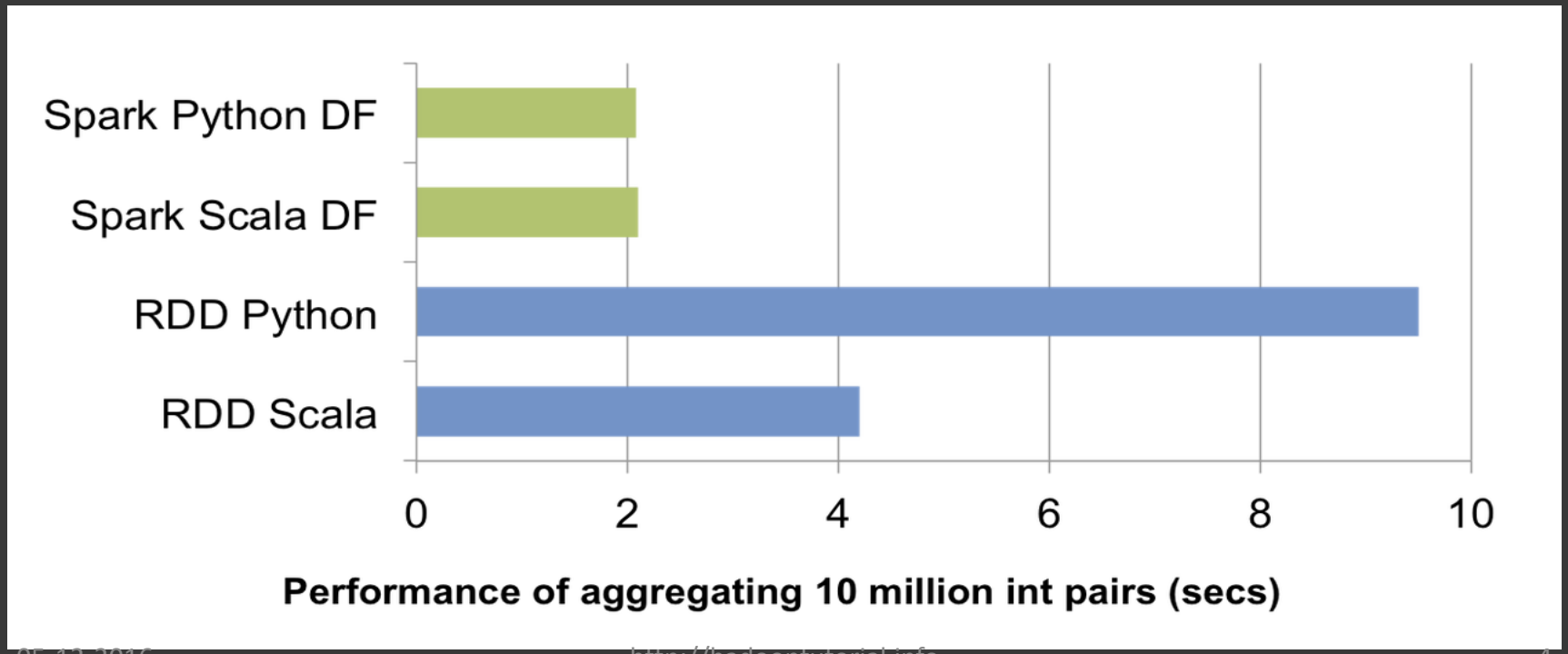
- A distributed collection of rows organized into named columns.
- Conceptually equivalent to a table.
- An abstraction for selecting, filtering, aggregating structured data
- Previously SchemaRDD (Spark < 1.3)
- Constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.
- Below shows the creation of SQLContext object,

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

- SQLContext object is by default available in spark shell.
- SQL query returns data frame object

DF vs RDD

- Manipulation of DFs is similar way to the "native" distributed collections in RDDs.
- Unlike RDDs , Data frames keep track of the schema .
- Execution in DF will be more optimised when compared to RDD
- DFs represents a logical plan but because of their "lazy" nature no execution occurs until the user calls a specific "output operation“.



About Data frames

- Below code shows the creating dataframes using SQLContext.

```
val df = sqlContext.read.load
            ("examples/src/main/resources/users.parquet")
```

- To see the data in the data frame, just use show() function

```
      df.show()
```

- In similar way, we can create DFs in java and python as shown below,

```
SQLContext sqlContext = new
      org.apache.spark.sql.SQLContext(sc);

DataFrame df = sqlContext.read().
      json("examples/src/main/resources
                  /people.json");
// Displays the content of the DataFrame to stdout
df.show();
```

About Data frames

- Below code shows the creating data frames from native spark API.

```
 case class  Employee(e_id: Int, name: String, address:
String)
 val emp = sc.textFile("emp.txt").map(_.split("\t"))
 val emp_bean = emp.map(p => Employee (p(0).toInt,
 p(1).trim, p(2).trim))
 val df1 = emp_bean.toDF()
 df1.show()
 df1.printSchema
```

Operations on DF

- First we will a json file and perform operations on that,

```
val df = sqlContext.read.json
        ("file:///home/cloudera/data.json")
df: org.apache.spark.sql.DataFrame = [city: string, id:
string, name: string]
```

- Displaying data inside DF

```
df.show()
+----+---+-----+
|city| id| name|
+----+---+-----+
| kkd|  1| Siva|
| hyd|  2|Kumar|
| bng|  3| Ravi|
| amt|  4|  Ram|
+----+---+-----+
```

Operations on DF

- print the schema in tree format.

```
df.printSchema()
root
 |-- city: string (nullable = true)
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
```

- print the data in specific column.

```
df2.select("name","age").show
+-------+---+
|   name|age|
+-------+---+
|Snehali| 25|
|   Jags| 30|
|    Rev| 19|
| Larisa| 32|
|    Raj| 24|
|    Raj| 24|
+-------+---+
```

Operations on DF

- Filters data using some condition.

```
df.filter(df("id") > 1).show()
+----+---+-----+
|city| id| name|
+----+---+-----+
| hyd|  2| teja|
| bng|  3|veera|
| amt|  4| jaya|
+----+---+-----+
```

- To count number of rows in dataframe.

```
df.count()
res9: Long = 4
```

Operations on DF

- Group data by column name.

```
df.groupBy("name").count().show()
+-----+-----+
| name|count|
+-----+-----+
| jaya|    1|
| siva|    1|
| mani|    1|
|veera|    1|
+-----+-----+
```

- Converting DF to RDD

```
val rows= df.rdd
rows:org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[42]
rows.foreach(println)
[amt,4,jaya]
[kkd,1,ravi]
[hyd,2,teja]
[bng,3,veera]
```

Operations on DF

- Sort by column name

```
df.sort("name").show()
+----+---+-----+
|city| id| name|
+----+---+-----+
| amt|  4| jaya|
| kkd|  1| ravi|
| hyd|  2| teja|
| bng|  3|veera|
+----+---+-----+
```

- OrderBY - This is an alias for sort function.
- df.dtypes(1) – This will give you info about column

```
scala> df.dtypes
res13: Array[(String, String)] =
Array((city,StringType), (id,StringType),
(name,StringType))

scala> df.dtypes(0)
res15: (String, String) = (city,StringType)
```

Operations on DF

- df.explain - will give you details of DF and from which file its loaded

```
df.explain
== Physical Plan ==
Scan
JSONRelation[file:/home/cloudera/Desktop/data.json][city#9,id#
10,name#11]
```

- Limit records by number of rows

```
df.limit(2).show
+----+---+----+
|city| id|name|
+----+---+----+
| kkd|  1|ravi|
| hyd|  2|teja|
+----+---+----+
```

Operations on DF

- Where - Filters rows using the given condition. This is an alias for filter.

```
df.filter($"id" > 2)
df.where($"id" > 2)
```

- head – returns the first record of DF

```
df.head
res36: org.apache.spark.sql.Row = [kkd,1,ravi]
```

- First – returns the first record of DF. This alias for head
- df.na.drop() - Dropping rows containing any null values.
- take(n) - Returns the first n rows in the DataFrame.

```
df.take(2).foreach(w=>println(w.getString(0)+"-----
"+w.getString(1)+"------"+w.getString(2)))
kkd-----1------Siva
hyd-----2------Kumar
```

Operations on DF

- Collect - Returns an array that contains all rows in this DataFrame.

```
df.collect.foreach(w=>println(w.getString(2)))
ravi
teja
veera
jaya
```

- collectAsList - Returns a Java list that contains all rows in this DataFrame. So you can apply all list functions.
- toJavaRDD – Convert DF to java RDD

```
df.toJavaRDD
org.apache.spark.api.java.JavaRDD[org.apache.spark.sql.Row] = MapPartitionsRDD[42] at rdd at <console>:21
```

Operations on DF

- agg - Aggregates on the entire DataFrame without groups.

```
df.groupBy().agg(avg("age"),
max("sal")).show()
+--------+--------+
|avg(age)|max(sal)|
+--------+--------+
|    24.2|    9857|
+--------+--------+
```

- drop(colName) – drops the column specified in braces and returns new DF.

```
df.drop("id").show
+----+----+-----+----+
| age|city| name| sal|
+----+----+-----+----+
|  24| bng|veera|9857|
|  25| amt| jaya|1203|
|  23| kkd| ravi|4500|
+----+----+-----+----+
```

Operations on DF

- dropDuplicates – remove duplicate records and returns new DF

```
df.dropDuplicates.show
+----+----+----+-----+----+
| age|city|  id| name| sal|
+----+----+----+-----+----+
|  24| bng|   3|veera|9857|
|  25| amt|   4| jaya|1203|
|  23| kkd|   1| ravi|4500|
|  24| hyd|   2| teja|1278|
+----+----+----+-----+----+
```

- distinct - This is an alias for dropDuplicates.
- cache – it will put DF in to in-memory

```
df.cache
res72: df.type = [age: string, city: string, id:
string, name: string, sal: string]
```

Operations on DF

- persist – This alias for cache function
- unpersist – Removes DF from in-memory.
- join – joins to DFs and returns new DF

```
val df = sqlContext.read
       .json("/home/cloudera/Desktop/data.json")
val df1 = sqlContext.read
       .json("/home/cloudera/Desktop/data1.json")
df.join(df1,"id").show
+---+----+---+----+----+----+
|age|city| id|name| sal| add|
+---+----+---+----+----+----+
| 23| kkd|  1|ravi|4500|GHMC|
| 23| kkd|  1|ravi|4500|GHMC|
+---+----+---+----+----+----+
```

Operations on DF

- map – works similar as in RDD

```
df.map(w=> w(0)).foreach(println)
25
23
24
```

- registerTempTable – Registers this DataFrame as a temporary table using the given name.

```
val teenagers = sqlContext.sql("SELECT name, age FROM
emp WHERE age > 23 AND age <= 25")
    org.apache.spark.sql.DataFrame = [name: string, age:
    string]
teenagers.show()
    +-----+---+
    | name|age|
    +-----+---+
    | teja| 24|
    | jaya| 25|
    +-----+---+
```

HiveContext

- An instance of the Spark SQL execution engine that integrates with data stored in Hive.
- Configuration for Hive is read from hive-site.xml on the classpath

Modifier and Type	Method and Description
void	**analyze**(String tableName)Analyzes the given table in the current database to generate statistics, which will be used in query optimizations.
void	**refreshTable**(String tableName)Invalidate and refresh all the cached the metadata of the given table.
void	**setConf**(String key, String value)Set the given Spark SQL configuration property.
DataFrame	**sql**(String sqlText)Executes a SQL query using Spark, returning the result as a **DataFrame**.

ORC Files in SQLContext

```
//Loading using SqlContext
import org.apache.spark.sql._
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
val people =
sqlContext.read.format("orc").load("hdfs://quickstart.cloudera:8020/user/cloudera/orc")

//sqlContext.read.orc("hdfs://quickstart.cloudera:8020/user/cloudera/orc")
scala> val orcDf2 = hc.read.format("orc").load("/user/cloudera/orc/000000_0")
orcDf2: org.apache.spark.sql.DataFrame = [_col0: int, _col1: string ... 1 more field]

scala> val parquetDf2 =
hc.read.format("parquet").load("/user/cloudera/parquet/000000_0")
parquetDf2: org.apache.spark.sql.DataFrame = [e_id: int, name: string ... 1 more field]

scala> val jsonDf2 = hc.read.format("json").load("file:///home/cloudera/data.json")
jsonDf2: org.apache.spark.sql.DataFrame = [city: string, id: bigint ... 1 more field]
```

Parquet Files in Spark

```
val emp = sqlContext.read.parquet("
hdfs://quickstart.cloudera:8020/user/cloudera/parquet")

//sqlContext.read.load("/user/cloudera/parquet")

emp.registerTempTable(emp)
sqlContext.sql("select name,e_id from emp where e_id>123")
```

JSON Files Parsing

```
import scala.util.parsing.json.JSON
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val df =
sqlContext.jsonFile("file:///home/Cloudera/data.json")
```

UDFs in Spark

- Creating UDF

```
def toUpper(name:String) : String = {
name.toUpperCase
}
```

- Below code shows loading data and registering the udf.

```
val df = sqlContext.read.json("/home/cloudera/user_data.json")
df.registerTempTable("user_data")

sqlContext.udf.register("toUpper", toUpper(_:String))
val user_data = sqlContext.sql("select id,toUppar(name) from
user_data")
```

- Shows below o/p

```
 user_data.show()
 +---+------+
 | id|   _c1|
 +---+------+
 |  1|  SIVA|
 |  2|  TEJA|
 |  5| KUMAR|
 +---+------+
```

SQL Data Types

- DataType - The base type of all Spark SQL data types
  - ArrayType,
  - BinaryType,
  - BooleanType,
  - DateType,
  - MapType,
  - NullType,
  - NumericType,
    - ✓ ByteType,
    - ✓ DecimalType, DoubleType, FloatType,
    - ✓ IntegerType, LongType, ShortType
  - StringType,
  - StructType,
  - TimestampType

SQL Data Types

public class StructType extends DataType
implements scala.collection.Seq<StructField>, scala.Product, scala.Serializable

A StructType object can be constructed by StructType(fields: Seq[StructField])
For a StructType object, one or multiple StructFields can be extracted by names. If multiple StructFields are extracted, a StructType object will be returned.

```
import org.apache.spark.sql._
val struct = StructType(
    StructField("a", IntegerType, true) ::
    StructField("b", LongType, false) ::
    StructField("c", BooleanType, false) :: Nil)
// Extract a single StructField.
val singleField = struct("b")
// singleField: StructField = StructField(b,LongType,false)
```

public class StructField extends Object implements scala.Product, scala.Serializable

- A field inside a StructType.
  - param: name - The name of this field.
  - param: dataType - The data type of this field.
  - param: nullable - Indicates if values of this field can be null values.

Defining schema programmatically

Loading file

```
val people = sc.textFile("/home/cloudera/people.txt")
import org.apache.spark.sql.Row;
import
org.apache.spark.sql.types.{StructType,StructField,StringType,
IntegerType};
```

Now we can define schema in two ways as shown below

```
val schema =
StructType(Array(StructField("id",IntegerType,true),StructFiel
d("name",StringType,true)))
                                (or)
val schemaString = "id,name"
val schema =
  StructType(
    schemaString.split(",").map(fieldName =>
StructField(fieldName, StringType, true)))
```

Defining schema programmatically

- Now mapping schema with data.

```
val rowRDD = people.map(_.split(",")).map(p =>
Row(p(0).toInt, p(1).trim))

val peopleDataFrame = sqlContext.createDataFrame(rowRDD,
schema)
Scala> val empRDD = emp.map(_.split("\t")).map(p =>
Employee(p(0).toInt, p(1).trim, p(2).trim))
empRDD: org.apache.spark.rdd.RDD[Employee] =
MapPartitionsRDD[19] at map at <console>:27

scala> val df3 = empRDD.toDF
df3: org.apache.spark.sql.DataFrame = [id: int, name:
string, city: string]
```

- Querying table

```
peopleDataFrame.registerTempTable("people")
val results = sqlContext.sql("SELECT name FROM people")
results.map(t => "Name: " + t(0)).collect().foreach(println)
```

Custom case class

- Case classes in Scala 2.10 can support only up to 22 fields.
- If we have more than 22 columns, we need to use custom classes as shown below

spark_case_classes_more_than_22_columns

Parquet Datasets

```
//writing df into parquet
people.write.parquet("people.parquet")

// Read in the parquet file created above.
// The result of loading a Parquet file is also a DataFrame.

val parquetFile = sqlContext.read.parquet("people.parquet")

//Parquet files can also be registered as tables and then used in SQL statements.

parquetFile.registerTempTable("parquetFile")

val teenagers = sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13
AND age <= 19")
```

Partitioned Parquet tables

- Creating and loading a partitioned data

```
import sqlContext.implicits._

val df = Seq(
(2012, 8, "Batman", 9.8),
(2012, 8, "Hero", 8.7),
(2012, 7, "Robot", 5.5),
(2011, 7, "Git", 2.0)).toDF("year", "month", "title", "rating")

df.write.partitionBy("year",
"month").parquet("hdfs://localhost:8020/user/cloudera/parq/")
```

- Getting data from particular partition

```
 val sqlContext = new SQLContext(sc)
val df =
sqlContext.read.parquet("hdfs://localhost:8020/user/cloudera/pa
rq/")
df.printSchema()
df.filter("year = 2011").show
```

Parquet tables Schema Merging

- Like ProtocolBuffer, Avro, and Thrift, Parquet also supports schema evolution.
- Users can start with a simple schema, and gradually add more columns to the schema as needed.
- The Parquet data source is now able to automatically detect this case and merge schemas of all these files.
- Schema merging is a relatively expensive and it is turned off by default. Set the global SQL option spark.sql.parquet.mergeSchema to true to enable it.

```
import sqlContext.implicits._
val df1 = sc.makeRDD(1 to 5).map(i => (i, i * 2)).toDF("single", "double")
df1.write.parquet("data/test_table/key=1")
// Create another DataFrame in a new partition directory,
val df2 = sc.makeRDD(6 to 10).map(i => (i, i * 3)).toDF("single", "triple")
df2.write.parquet("data/test_table/key=2")
// Read the partitioned table
val df3 = sqlContext.read.option("mergeSchema",
"true").parquet("data/test_table")
df3.printSchema()

// The final schema consists of all 3 columns in the Parquet files together
// with the partitioning column appeared in the partition directory paths.
// root
// |-- single: int (nullable = true)
// |-- double: int (nullable = true)
// |-- triple: int (nullable = true)
// |-- key : int (nullable = true)
```

Partitioned ORC tables

- Creating and loading a partitioned data

```
import sqlContext.implicits._
val df = Seq(
(2012, 8, "Batman", 9.8),
(2012, 8, "Hero", 8.7),
(2012, 7, "Robot", 5.5),
(2011, 7, "Git", 2.0)).toDF("year", "month", "title",
"rating")

df.write.partitionBy("year",
"month").orc("hdfs://localhost:8020/user/cloudera/orc123/"
)
```

- Getting data from particaulr partition

```
 val sqlContext = new SQLContext(sc)
val df =
sqlContext.read.orc("hdfs://localhost:8020/user/cloudera/o
rc123/")
df.printSchema()
df.filter("year = 2011").foreach(println)
```

Saving to Persistent Tables

- When working with a HiveContext, DataFrames can also be saved as persistent tables using the saveAsTable command
- Unlike the registerTempTable command, df.write.saveAsTable() will materialize the contents of the dataframe and create a pointer to the data in the HiveMetastore.
- Persistent tables will still exist even after your Spark program has restarted, as long as you maintain your connection to the same metastore.
- A DataFrame for a persistent table can be created by calling the table method on a SQLContext with the name of the table.
- By default saveAsTable will create a “managed table”, meaning that the location of the data will be controlled by the metastore. Managed tables will also have their data deleted automatically when a table is dropped.

Hive Tables

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)

sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt'
INTO TABLE src")

// Queries are expressed in HiveQL
sqlContext.sql(" SELECT key, value FROM src").show
```

Compatibility with Hive

- **./bin/spark-sql** - Run Hive queries from the command line

- **Supported Hive Features**
  - ✓ Hive query statements, including: SELECT, GROUP BY, ORDER BY, SORT BY.
  - ✓ All Hive operators
  - ✓ UDFs & UDAFs
  - ✓ Joins (INNER, OUTER, SEMI)
  - ✓ Window Functions
  - ✓ Partitioning
  - ✓ Views
- **Unsupported Hive Functionality**
  - ✓ Tables with buckets
  - ✓ UNION type
  - ✓ Unique join
  - ✓ Column statistics collecting
  - ✓ Hive optimizations are not yet included in Spark

JDBC to other Databases

```
val jdbcDF = sqlContext.read.format("jdbc").options(
  Map("url" -> "jdbc:postgresql:dbserver",
  "dbtable" -> "schema.tablename")).load()
```

Property Name	Meaning
url	The JDBC URL to connect to.
dbtable	The JDBC table that should be read. Note that anything that is valid in a FROM clause of a SQL query can be used.
driver	The class name of the JDBC driver needed to connect to this URL. This class will be loaded on the master and workers before running an JDBC commands to allow the driver to register itself with the JDBC subsystem.
partitionColumn, lowerBound, upperBound, numPartitions	These options must all be specified if any of them is specified. They describe how to partition the table when reading in parallel from multiple workers. partitionColumn must be a numeric column from the table in question
fetchSize	The JDBC fetch size, which determines how many rows to fetch per round trip.

JDBC to MySQL Databases

```
val dataframe_mysql =
sqlContext.read.format("jdbc").option("url",
"jdbc:mysql://localhost:3306/scala").
      option("driver",
"com.mysql.jdbc.Driver").option("dbtable",
"scala_test").option("user", "root").
      option("password", "cloudera").load()

dataframe_mysql.saveAsTable("scala.scala_test_1")

dataframe_mysql.sqlContext.sql("select * from
scala_test").collect.foreach(println)

df.write.format("jdbc").option("url",
"jdbc:mysql://localhost:3306/retail_db").option("driver",
"com.mysql.jdbc.Driver").option("dbtable",
"categories2").option("user", "root").option("password",
"cloudera")
```

Spark Application with SqlContext

```
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.sql.hive.HiveContext

val conf = new
SparkConf().setMaster("local[*]").setAppName("SparkSQLEX")
val sc = new SparkContext(conf)
val hiveCtx = new HiveContext(sc)

val input = hiveCtx.jsonFile(inputFile)
//val input = hiveCtx.read.json(inputFile)
input.registerTempTable("tweets")
hiveCtx.cacheTable("tweets")
val topTweets = hiveCtx.sql("SELECT text, retweetCount FROM
tweets ORDER BY retweetCount LIMIT 10")
```

THANK YOU