Spark SQL and Dataframes

Lesson 6

Hortonworks

Learning Objectives

- After you complete this lesson you should be able to:
  - Load multiples types of data
  - Perform SQL queries
  - Perform Dataframe operations
  - Understand some of the optimization engine

Hortonworks

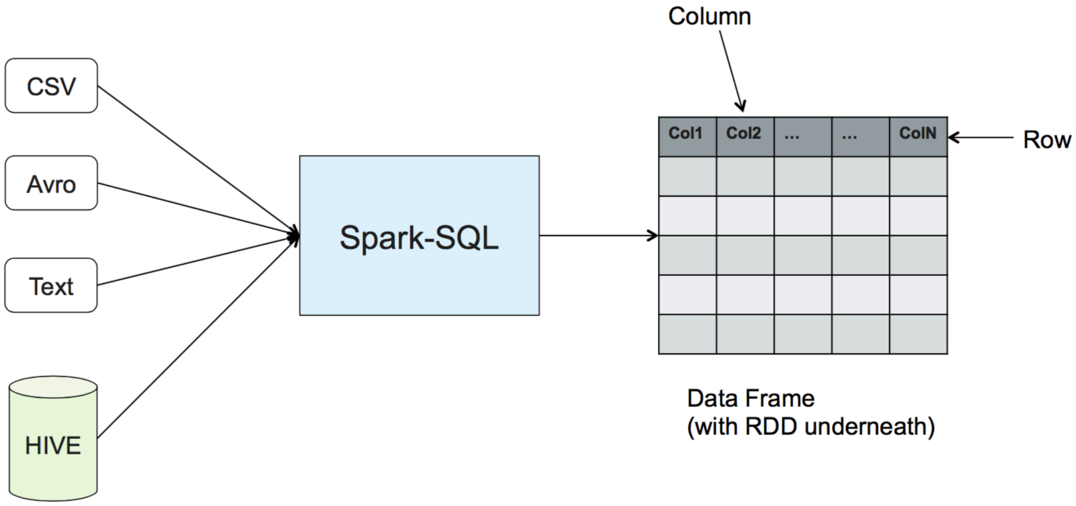
Spark SQL Overview

- A module built on top of Spark Core
- Provides a programming abstraction for distributed processing of large-scale structured data in Spark
- Data is described as a Dataframe with rows, columns and a schema
- Data manipulation and access is available with two mechanisms
  - SQL Queries
  - Dataframes API

The Dataframe Abstraction

- A Dataframe is inspired by the dataframe concept in R (dplr, Dataframe) or Python (pandas), but stored using RDDs underneath in a distributed manner
- A Dataframe is organized into named coimns
  - Underneath: an RDD of “Row” objects
- The Dataframe API is available in Scala, Java, Python, and R

The Dataframe Visually



CSV

Avro

Text

HIVE

Spark-SQL

Column

Col1	Col2	...	...	ColN

Row

Data Frame
(with RDD underneath)

Dataframes can be created from various source

- Dataframes from HIVE data
  - Reading/writing HIVE tables
- Dataframes from files:
  - Built-in: JSON, JDBC, Parquet, HDFS
  - External plug-in: CSV, HBASE, Avro, memsql, elasticsearch

SQLContext and HiveContext

- To use Spark-SQL you first create a SQLContext

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

- Alternatively you can create a HiveContext to connect with HIVE:

```
from pyspark.sql import HiveContext
hc = HiveContext(sc)
```

Hortonworks

Example: Using the Data Frames API

```
from pyspark.sql import HiveContext
hc = HiveContext(sc)

hc.sql(“use demo”)
df1 = hc.table(“crimes”)
            .select(“year”, “month”, “day”, “category”)
            .filter(“year > 2014”).head(5)
```

Hortonworks

Same example, using SQL syntax

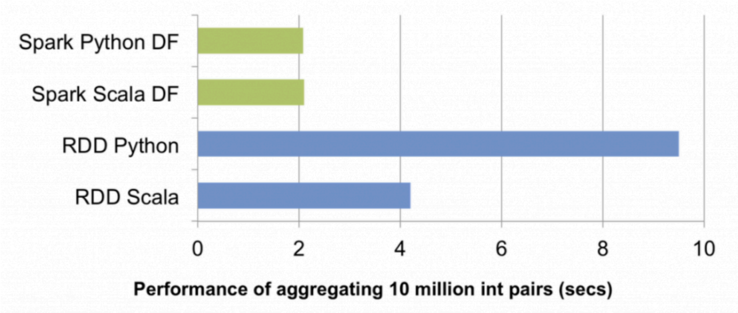
```
from pyspark.sql import HiveContext
hc = HiveContext(sc)

hc.sql(“use demo”)
df1 = hc.sql(“““
        SELECT year, month, day, category
        FROM crimes
        WHERE year > 2014”””).head(5)
```

Hortonworks

Dataframes vs Spark-Core?

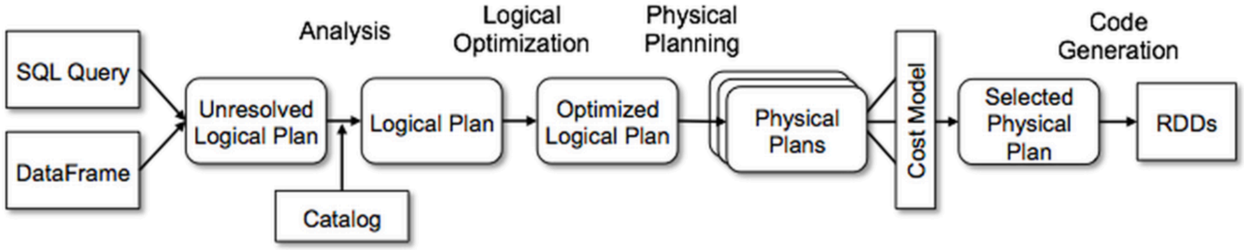
- Spark-SQL uses an optimization engine (Catalyst)
- Catalyst understands the structure of data and sematics of operations, and performs optimizations



Hortonworks

Catalyst: Spark-SQL optimizer

- Query or dataframe operations modeled as a tree
- Logical plan created and optimized
- Various physical plans created; best chosen
- Code generation and execution



Creating a Dataframe: from a table in HIVE

- Load the entire table

```
df=hc.table(“patients”
```

- Load using a SQL Query

```
df1 = hc.sql(“SELECT * from patients WHERE age>50”)
df2 = hc.sql(““““
    SELECT col1 as timestamp, SUBSTR(date,1,4) as year, event
    FROM events
    WHERE year > 2014”””)
```

Creating a Dataframe: from a file

- From a JSON file

```
df = hc.jsonFile(“somefile.json”)
df = hc.read.json(“somefile.json) **
```

- From Parquet file

```
df = hc.parquetFile(“somefile.parquet)
```

- From a CSV file:

```
df = hc.read.format(“com.databricks.spark.csv”)
        .options(header=‘true’).load(“somefile.csv”) **
```

Create a Dataframe: from an RDD

- Create an RDD of Row() objects and use toDF()

```
from pyspark.sql import Row
rdd = sc.parallelize([Row(name=‘Alice’, age=12, height=80),
                      Row(name=‘Bob’, age=15, height=120)])
df = rdd.toDF()
```

- Or let Spark-SQL infer the schema using createDataFrame()

```
rdd = sc.parallelize([(‘Alice’, 12, 80), (‘Bob’, 15, 120)])
df = hc.createDataFrame(rdd, [‘name’, ‘age’, ‘height’])
```

Create a Dataframe: from text (Python)

```
from pyspark.sql import SQLContext, Row
sqlContext = SQLContext(sc)

lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))
```

Cont… (Python)

```
# Infer the schema, and register the DataFrame as a table.
schemaPeople = sqlContext.inferSchema(people)
schemaPeople.registerTempTable("people")

# SQL can be run over DataFrames that have been registered as
a table.
teenagers = sqlContext.sql("SELECT name FROM people
WHERE age >= 13 AND age <= 19")
```

Create a Dataframe: from text (Scala)

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// this is used to implicitly convert an RDD to a DataFrame.
import sqlContext.implicits._

case class Person(name: String, age: Int)

val people = sc.textFile("examples/src/main/resources/
people.txt").map(_.split(",")).map(p => Person(p(0),
p(1).trim.toInt)).toDF()
```

Hortonworks

Example Dataframes

For the next few slides, let’s create two data frames:

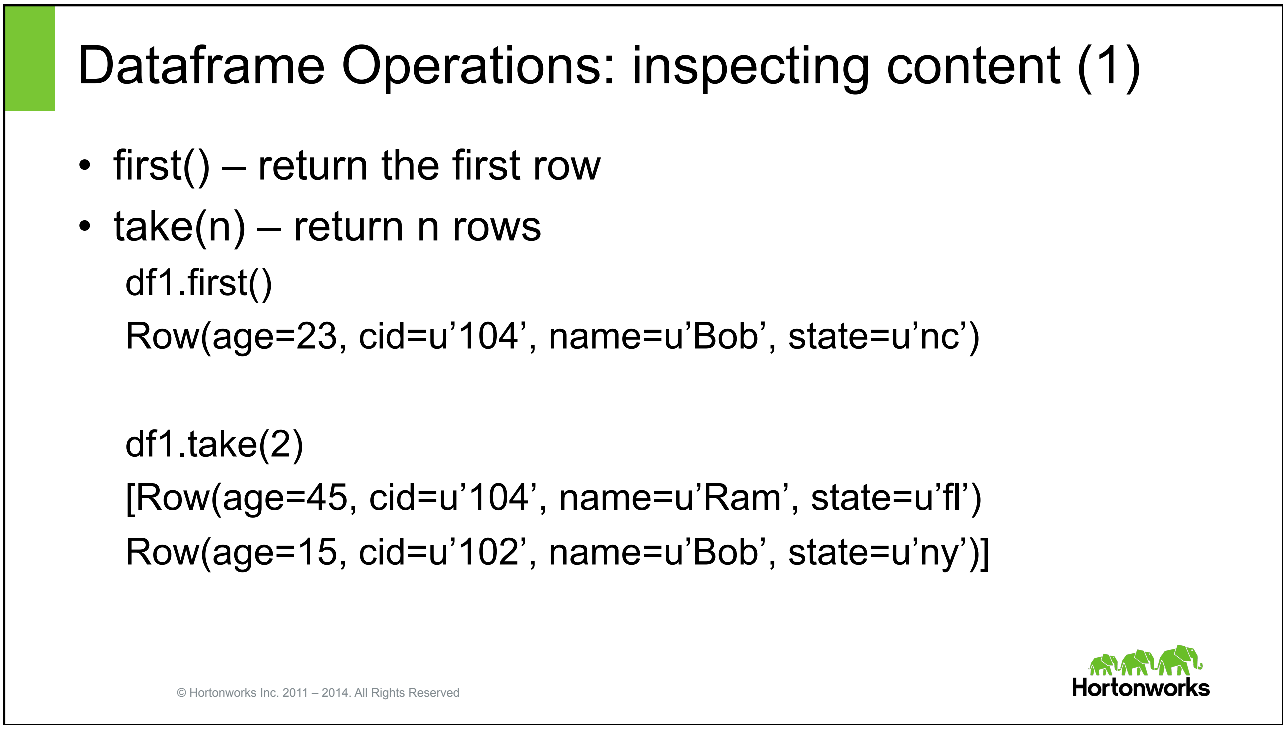
```
df1 = sc.parallelize(
      [Row(cid=‘101’, name='Alice', age=25, state=‘ca’), \
       Row(cid=‘102’, name=’Bob', age=15, state=‘ny’), \
       Row(cid=‘103’, name=‘Bob’, age=23, state=‘nc’), \
        Row(cid=‘104’, name=’Ram', age=45, state=‘fl’)]).toDF()

df2 = sc.parallelize(
      [Row(cid=‘101’, date=‘2015-03-12’, product=‘toaster’, price=200), \
       Row(cid=‘104’, date=‘2015-04-12’, product=‘iron’, price=120), \
       Row(cid=‘102’, date=‘2014-12-31’, product=‘fridge’, price=850), \
       Row(cid=‘102’, date=‘2015-02-03’, product=‘cup’, price=5)]).toDF()
```

	age	cid	name	state
0	25	101	Alice	ca
1	15	102	Bob	ny
2	23	103	Bob	nc
3	45	104	Ram	fl

	cid	date	price	product
0	101	2015-03-12	200	toaster
1	104	2015-04-12	120	iron
2	102	2014-12-31	850	fridge
3	102	2015-02-03	5	cup

Hortonworks

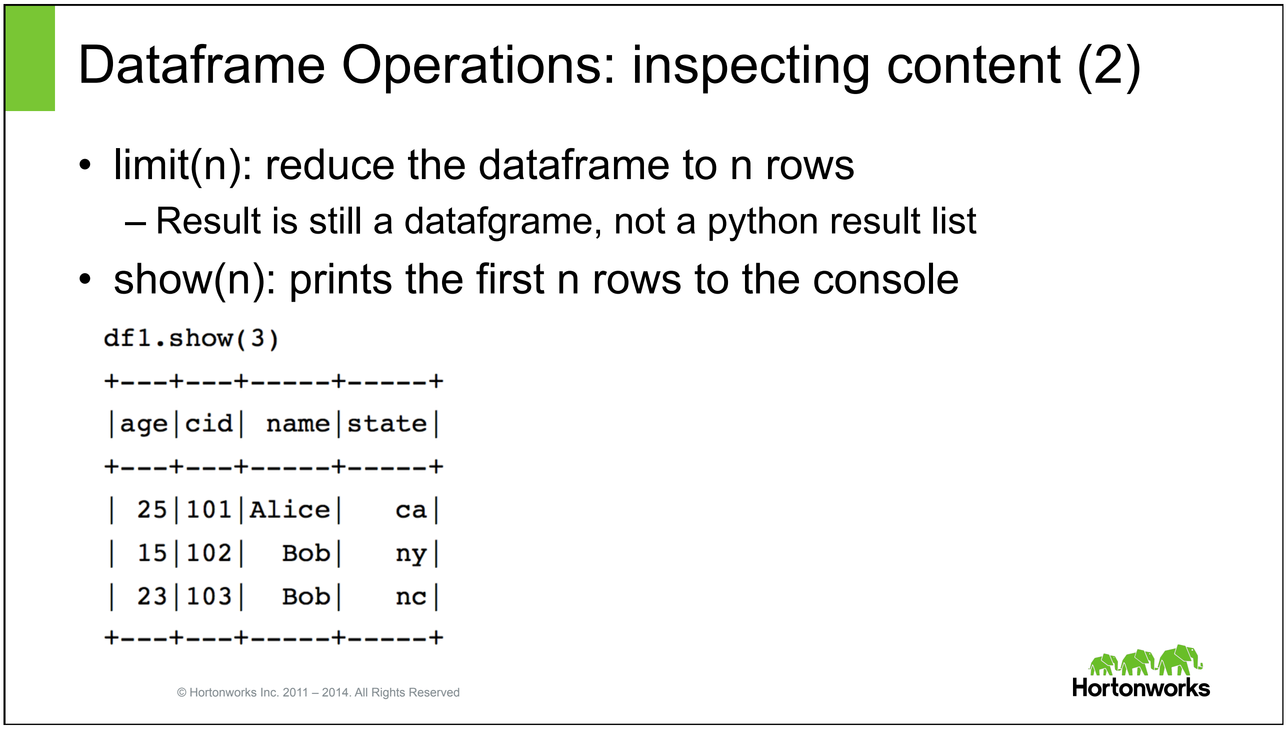


Dataframe Operations: inspecting content (1)

- first() – return the first row
- take(n) – return n rows

```
df1.first()
Row(age=23, cid=u’104’, name=u’Bob’, state=u’nc’)

df1.take(2)
[Row(age=45, cid=u’104’, name=u’Ram’, state=u’fl’)
Row(age=15, cid=u’102’, name=u’Bob’, state=u’ny’)]
```



Dataframe Operations: inspecting content (2)

- limit(n): reduce the dataframe to n rows
  - – Result is still a datafgrame, not a python result list
- show(n): prints the first n rows to the console

```
df1.show(3)
+---+---+-----+-----+
|age|cid| name|state|
+---+---+-----+-----+
| 25|101|Alice|   ca|
| 15|102|  Bob|   ny|
| 23|103|  Bob|   nc|
+---+---+-----+-----+
```

Dataframe Operations: Inspecting Schema

```
df1.columns  #Display column names
[u’age’, u’cid’, u’name’, u’state’]

df1.dtypes  #Display column names and types
[('age', 'bigint'), ('cid', 'string'), ('name', 'string'), ('state', 'string')]

df1.schema  #Display detailed schema
StructType(List(StructField(age,LongType,true),StructField(cid,StringType,true),StructField(name,StringType,true),StructField(state,StringType,true)))
```

Dataframe operations: Counting Rows

```
df1.count()
4
```

Note**

count() returns number of non-duplicate rows

df1.rdd.count() returns number of actual rows

Dataframe Operations: Removing duplicates

df1.distinct().show()

Removing duplicate rows by key*

df1.drop_duplicates([“name”]).show()

Hortonworks

Saving Dataframes as a file

- Not Many Options in 1.3

  df.saveAsParquetFile(“/output.parquet”)

  df.rdd.saveAsTextFile(“/path/to/output.txt”)

- Most options in 1.4 Available

  df.write.format(“parquet”).save(“output.parquet”) *

  df.write.format(“com.databricks.spark.avro”).save(“output.avro”)

Hortonworks

Conclusion and Key Points

Hortonworks