Spark Data Model 2

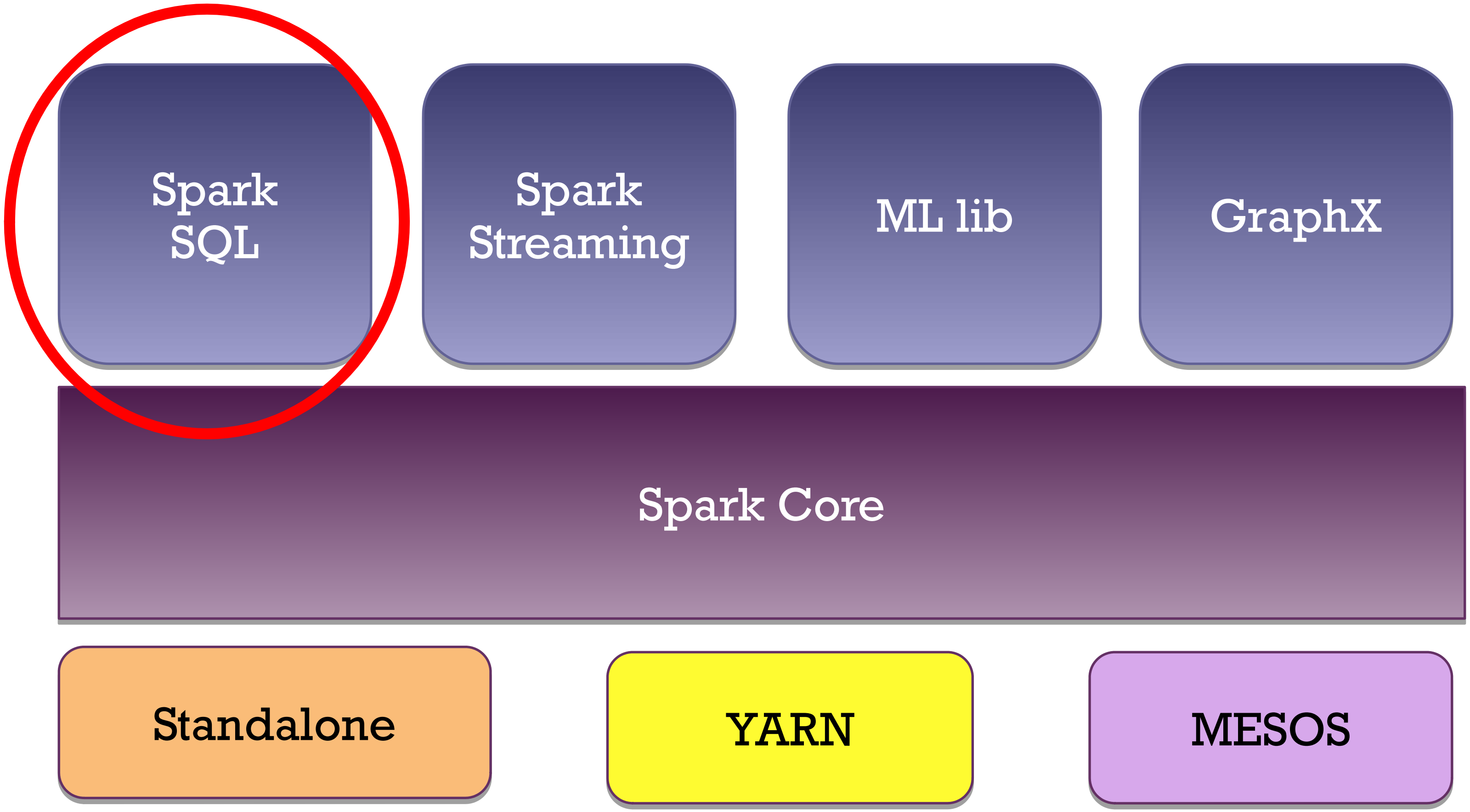
DataFrames
Working with DataFrames
Spark SQL
Dataset
Spark and Hive
Data Formats

Lesson Objectives

- Understand DataFrames and Datasets
- Understand what Spark SQL is and the needs it fulfills
- Learn Spark SQL architecture and API
- Use Spark SQL for querying

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Spark Illustrated

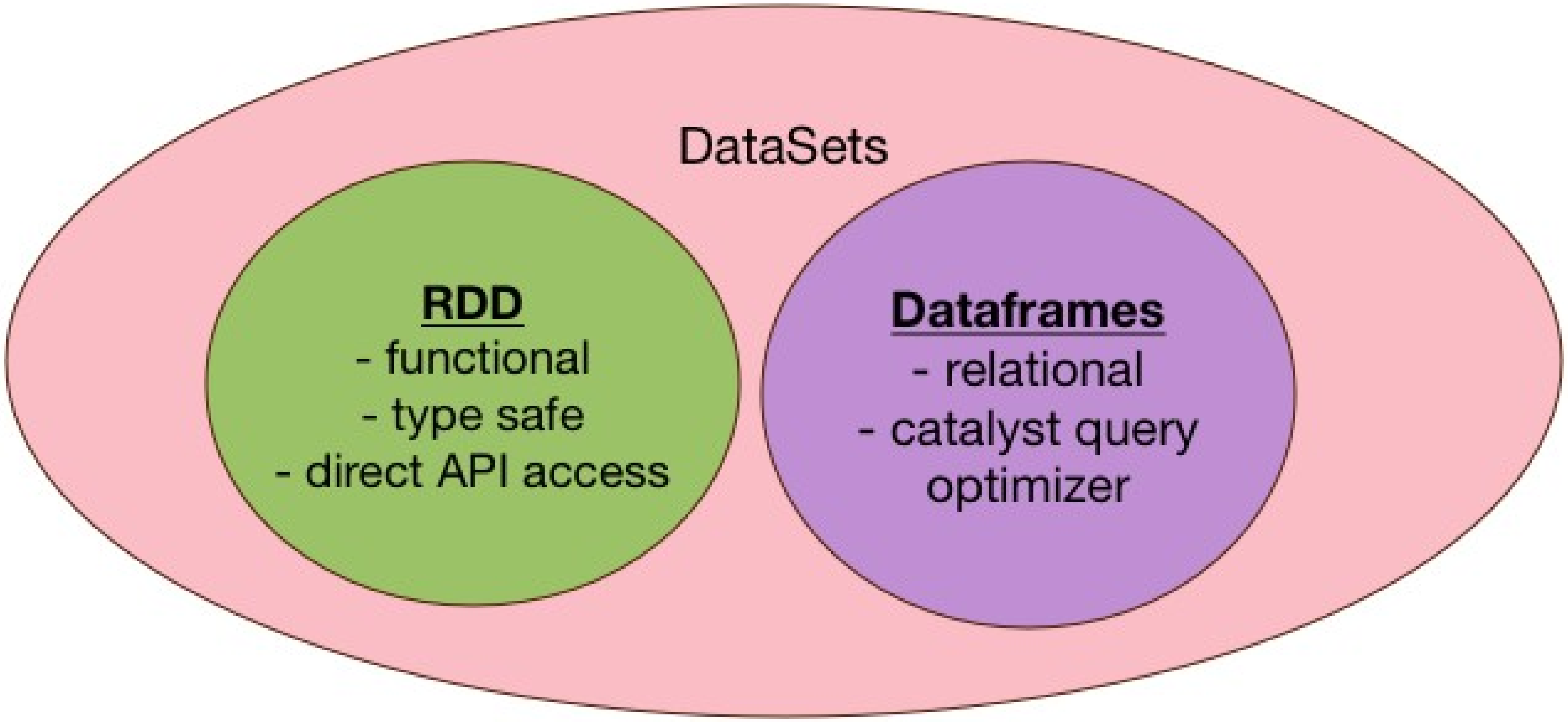


Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

DataFrames

➔ **DataFrames**
Working with DataFrames
Spark SQL
Dataset
Spark and Hive
Data Formats

Spark Data Model Evolution



Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

DataFrames

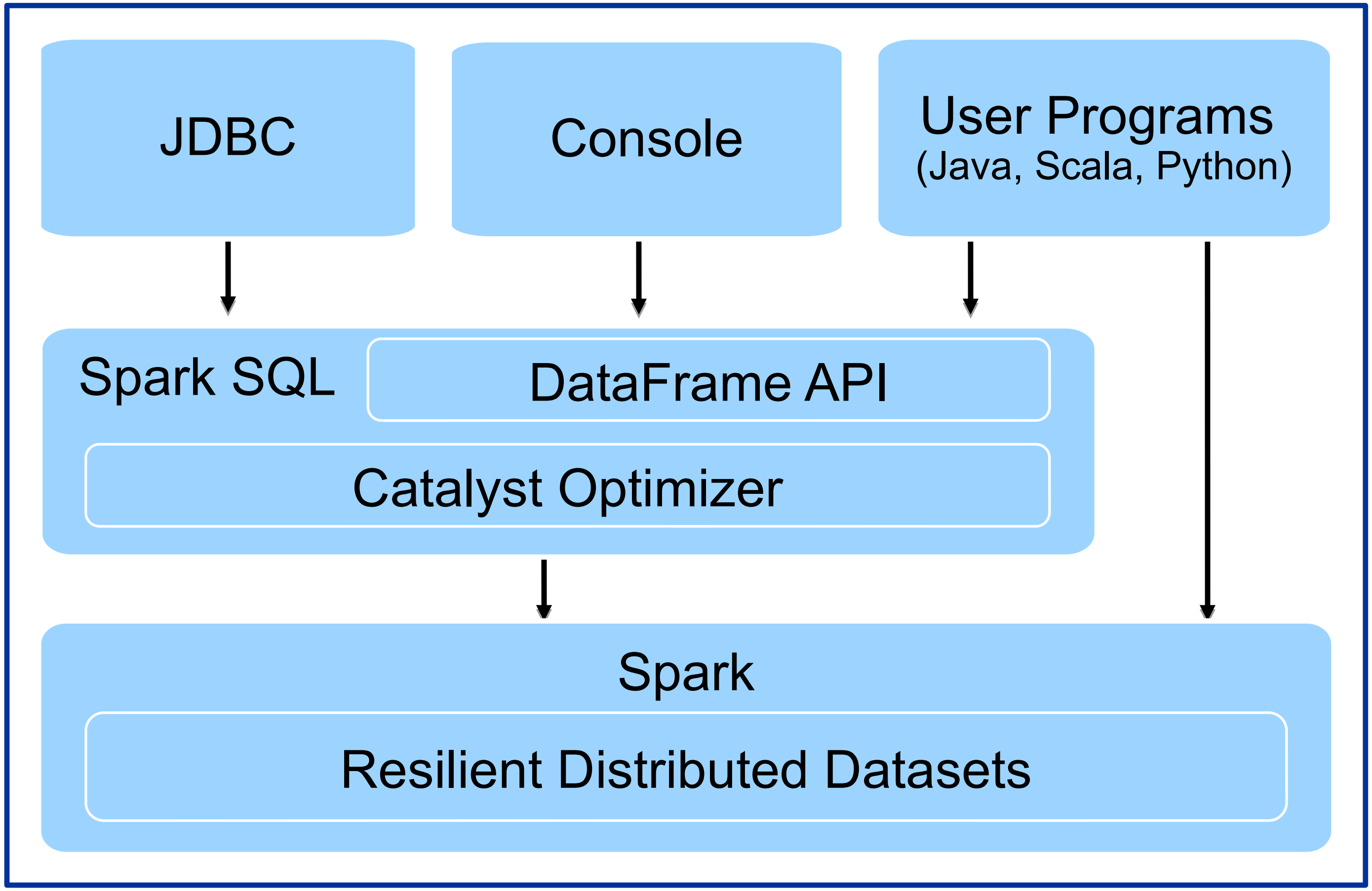
- **DataFrame**: Distributed collection of data organized into **named columns**
- Equivalent to relational table or data frame in R/Python
- Can be built from many data sources
  - RDDs, structured data files, Hive tables, external databases
- High-level API: Java/Scala/Python/R
- Can query using DSL and SQL!

	A	B	C
1	Name	Gender	Age
2	John	M	35
3	Jane	F	40
4	Mike	M	18
5	Sue	F	19

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

DataFrames Architecture

- Built on RDD
- Uses Catalyst to optimize queries
- API: Java/Python/Scala/R



Source: "Spark SQL: Relational Data Processing in Spark" by Michael Armbrust, et al.

DataFrames vs. RDDs

- RDDs have data
- DataFrames also have schema
  DataFrame = RDD + schema
- Unified way to load/save data in multiple formats
- Provides high-level operations
  - Count/sum/average
  - Select columns & filter them

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Supported Formats

Built-In

- { JSON }
- JDBC
- Parquet
- HIVE
- MySQL
- PostgreSQL
- HDFS
- amazon web services S3
- H2

External

- AVRO
- CSV
- dBase
- APACHE HBASE
- elasticsearch.
- cassandra
- amazon web services Amazon Redshift
- memsql
- and more…

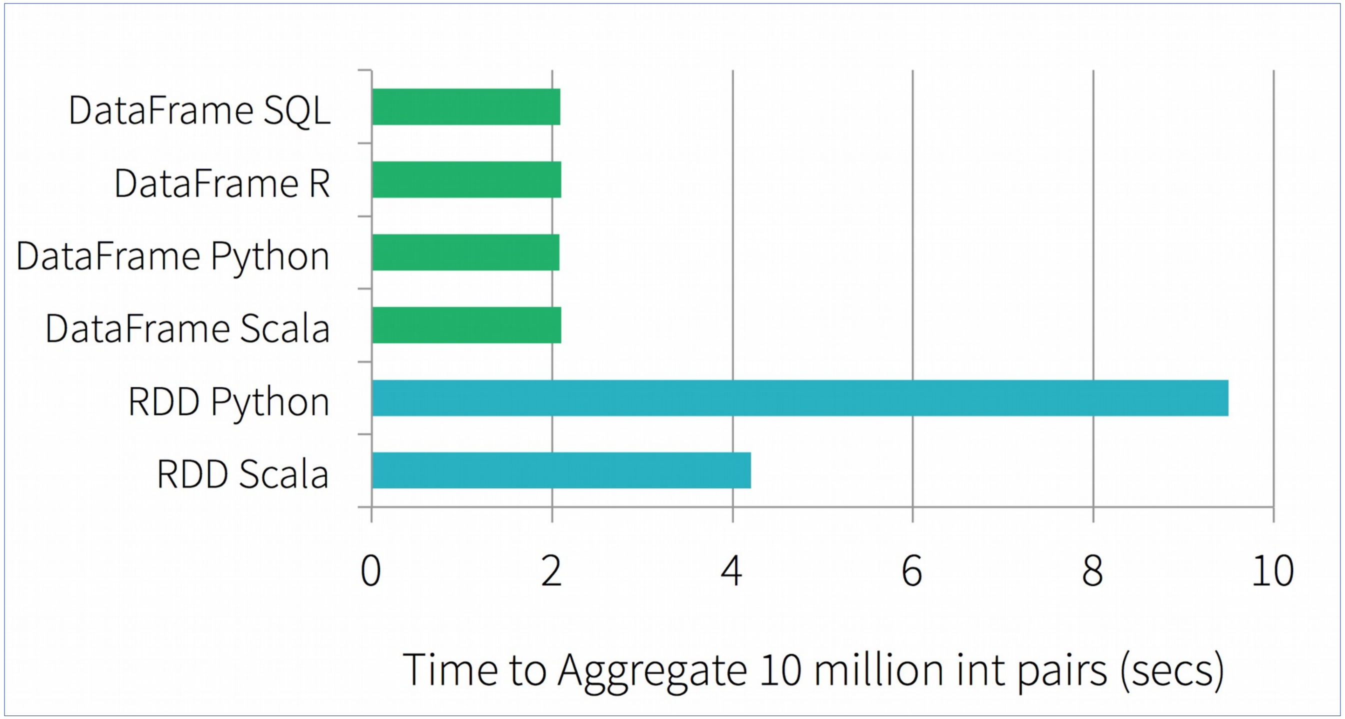
Case for DataFrames

- High level, simple API
- Supports SQL!
- Supports multiple data formats natively (JSON, Parquet, etc.)
- High performance
  - Catalyst Optimizer
    - Generates optimized code
    - Takes advantage of all the benefits and tweaks
  - Efficient memory usage
    - Uses Tungston engine to efficiently store/access objects in memory

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Performance

- Generally performs very well–often better than vanilla Spark
- Below, we illustrate the performance of running group-by aggregation on 10 million integer pairs on one machine
- Spark SQL (the DF lines) outperforms their vanilla counterparts



Source: "Spark Data Frames" by Michael Armbrust

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Working with DataFrames

Creating a DataFrame from JSON (Scala)


```
{"name": "John", "age": 35, "gender" : "M", "weight": 200.5 }
{"name": "Jane", "age": 40, "gender" : "F" , "weight": 150.2}
{"name": "Mike", "age": 18, "gender" : "M" , "weight": 120}
{"name": "Sue",  "age": 19, "gender" : "F" , "weight": 100}
```

```
val peopleDF = spark.read.json("people.json")
peopleDF: org.apache.spark.sql.DataFrame = [age: long, name:
string, gender:string, weight:double]

peopleDF.printSchema
root
 |-- age: long (nullable = true)
 |-- gender: string (nullable = true)
 |-- name: string (nullable = true)
 |-- weight: double (nullable = true)
peopleDF.show
+---+------+----+------+
|age|gender|name|weight|
+---+------+----+------+
| 35|     M|John| 200.5|
| 40|     F|Jane| 150.2|
| 18|     M|Mike| 120.0|
| 19|     F| Sue| 100.0|
+---+------+----+------+
```

Creating a DataFrame from JSON (Python)


```
{"name": "John", "age": 35, "gender" : "M", "weight": 200.5 }
{"name": "Jane", "age": 40, "gender" : "F" , "weight": 150.2}
{"name": "Mike", "age": 18, "gender" : "M" , "weight": 120}
{"name": "Sue",  "age": 19, "gender" : "F" , "weight": 100}
```

```
peopleDF = spark.read.json("people.json")
// in v1.6 use: val peopleDF = sqlContext.read.json("people.json")
peopleDF: org.apache.spark.sql.DataFrame = [age: long, name: string,
gender:string]

peopleDF.printSchema()
root
 |-- age: long (nullable = true)
 |-- gender: string (nullable = true)
 |-- name: string (nullable = true)
 |-- weight: double (nullable = true)

peopleDF.show()
+---+------+----+------+
|age|gender|name|weight|
+---+------+----+------+
| 35|     M|John| 200.5|
| 40|     F|Jane| 150.2|
| 18|     M|Mike| 120.0|
| 19|     F| Sue| 100.0|
+---+------+----+------+
```

Querying of a DataFrame Using DSL (Scala)

```
// This import is needed to use the $-notation
import spark.implicits._

val df = spark.read.json("people.json”)
df.select("name").show()
+----+
|name|
+----+
|John|
|Jane|
|Mike|
| Sue|
+----+
```

{"name": "John", "age": 35, "gender" : "M", "weight": 200.5 }
{"name": "Jane", "age": 40, "gender" : "F" , "weight": 150.2}
{"name": "Mike", "age": 18, "gender" : "M" , "weight": 120}
{"name": "Sue", "age": 19, "gender" : "F" , "weight": 100}

```
df.filter(df("name")==="John").show() // note equal is ===
df.filter("name == 'John'").show
df.filter($"name" === "John").show
+---+------+----+
|age|gender|name|
+---+------+----+
| 35|     M|John|
+---+------+----+

df.filter(df("age") >35).show()
df.filter("age>20").show
df.filter($"age" > 20).show

+---+------+----+
|age|gender|name|
+---+------+----+
| 35|     M|John|
| 40|     F|Jane|
+---+------+----+
```

Querying of a DataFrame Using DSL (Python)

```
df = spark.read.json("people.json")
df.select("name").show()
+----+
|name|
+----+
|John|
|Jane|
|Mike|
| Sue|
+----+
```

```
{"name": "John", "age": 35, "gender" : "M", "weight": 200.5 }
{"name": "Jane", "age": 40, "gender" : "F" , "weight": 150.2}
{"name": "Mike", "age": 18, "gender" : "M" , "weight": 120}
{"name": "Sue",  "age": 19, "gender" : "F" , "weight": 100}
```

```
df.filter(df("name")=="John").show()
df.filter("name == 'John'").show()
+---+------+----+
|age|gender|name|
+---+------+----+
| 35|     M|John|
+---+------+----+
```

```
df.filter(df["age"] >35).show()
df.filter("age>20").show

+---+------+----+
|age|gender|name|
+---+------+----+
| 35|     M|John|
| 40|     F|Jane|
+---+------+----+
```

The DataFrame DSL

- The DataFrame DSL supports many common operations
  - Selecting columns, joining, filtering
  - Aggregation (count, sum, average, etc.)

- A query in the DSL is composed of:
  - An operation or operations
  - Expressions passed as arguments to the operators

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Examining a DSL Query

- `df.filter(df("name")==="John").show()`
  - The `filter()` call is straightforward–it's a regular method
  - An operation that filters rows based on the passed in expression
- `df("name")==="John"`
  - `df("name")` specifies the column named "**name**"
    - It's the same as `df.apply("name")`
    - Scala magic converts `df1("name")` to the call to apply
    - This returns a Column object
  - "`where the value in the name column is equal to John`"
  - `===` (three equal signs) is the equality operator in this DSL, not ==
- `df.filter("name == 'John'").show`
  - Shorter version

Supported Data Types

Type	Description	In Scala / Java	In Python
Numeric Types			
ByteType	1-byte signed integer numbers. Range = -128 to 127	Byte	Int or Long
ShortType	2-byte signed integer numbers. Range = -32768 to 32767	Short	Int or Long
IntegerType	4-byte (32 bit) signed integer numbers. Range = -2,147,483,648 (-2^{31}) to 2,147,483,647 (2^{31})	Integer	Int or Long
LongType	8-byte (64 bit) signed integer numbers Range = -2^{63} to 2^{63}	Long	Long
FloatType	4-byte (32 bit) single-precision floating point numbers	Float	Float
DoubleType	8-byte (64 bit) double-precision floating point numbers	Double	Float
DecimalType	arbitrary-precision signed decimal numbers	java.math.BigDecimal	decimal.Decimal

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Supported Data Types

Type	Description	In Scala / Java	In Python
StringType	String / text values	String	string
BinaryType	Binary / blob data	Array[Byte]	bytearray
BooleanType	True / False	Boolean	bool
Dates			
DateType	Date with year, month, day.	java.sql.Date	datetime.date
TimestampType	Timestamp with year, month, day, hour, minute, and second.	java.sql.Timestamp	datetime.datetime

Supported Data Types

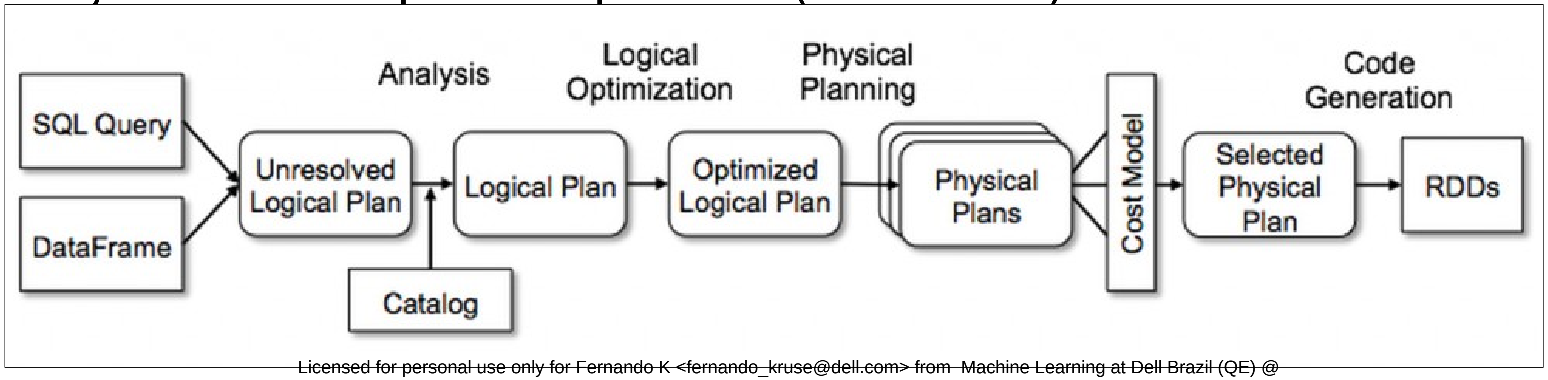
Type	Description	In Scala / Java	In Python
Complex Types			
ArrayType	Sequence of elements	scala.collection.Seq	list, tuple, or array
MapType	Key → Value pairs	scala.collection.Map	dict

StructType	Random structure with one or more fields
	
	Address {
	street_number,
	street_name,
	city,
	state,
	zip
	}
		org.apache.spark.sql.Row	list or tuple

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Query Optimizer : Catalyst

- Dataframes are also lazily evaluated
  - Catalyst can optimize bunch of instructions together
  - It can combine / short-circuit / re-order operations
- Re-ordering operations
  - For example filter operations can be moved up if possible
  - Cuts down data
- Using schema information optimizer can perform additional optimization
  - It can inspect the logical meaning of operation rather arbitrary functions
- Prefer dataframe operations (filter, map, join …etc)
  - Optimizer knows how to execute these efficiently
- Optimizer may not be able to optimize arbitrary user functions
- Catalyst is a 'multi phase' optimizer (see below)



Some Optimizations : Predicate Pushdown

- **Pushdown** is moving filters close to data
- Cuts down amount of data that has to be processed
- Improves performance

```
{"name": "John", "age": 35, "gender" : "M", "weight": 200.5 }
{"name": "Jane", "age": 40, "gender" : "F" , "weight": 150.2}
{"name": "Mike", "age": 18, "gender" : "M" , "weight": 120}
{"name": "Sue",  "age": 19, "gender" : "F" , "weight": 100}
```

- Example:
  df.filter ("age > 30")
- Naïve approach:
  - Read all data and filter out records
- Smarter approach
  - Don't even read data that doesn't match the filter
  - We are filtering on 'age', so let's apply the following conditions
    (age != null) && (age > 30)

Predicate Pushdown Filter (Example)


```
// data
{"name": "John", "age": 35, "gender": "M" }
{"name": "Jane", "age": 40, "gender": "F" }
{"name": "Mike", "age": 18, "gender": "M" }
{"name": "Sue",  "age": 19, "gender": "F" }
```

Read all records

```
{"name": "John", "age": 35, "gender": "M" }
{"name": "Jane", "age": 40, "gender": "F" }
{"name": "Mike", "age": 18, "gender": "M" }
{"name": "Sue",  "age": 19, "gender": "F" }
```

Filter: age > 30

```
{"name": "John", "age": 35, "gender": "M" }
{"name": "Jane", "age": 40, "gender": "F" }
```

```
// data
{"name": "John", "age": 35, "gender": "M" }
{"name": "Jane", "age": 40, "gender": "F" }
{"name": "Mike", "age": 18, "gender": "M" }
{"name": "Sue",  "age": 19, "gender": "F" }
```

Read with pushdown filter : age != null && age > 30

```
{"name": "John", "age": 35, "gender": "M" }
{"name": "Jane", "age": 40, "gender": "F" }
```

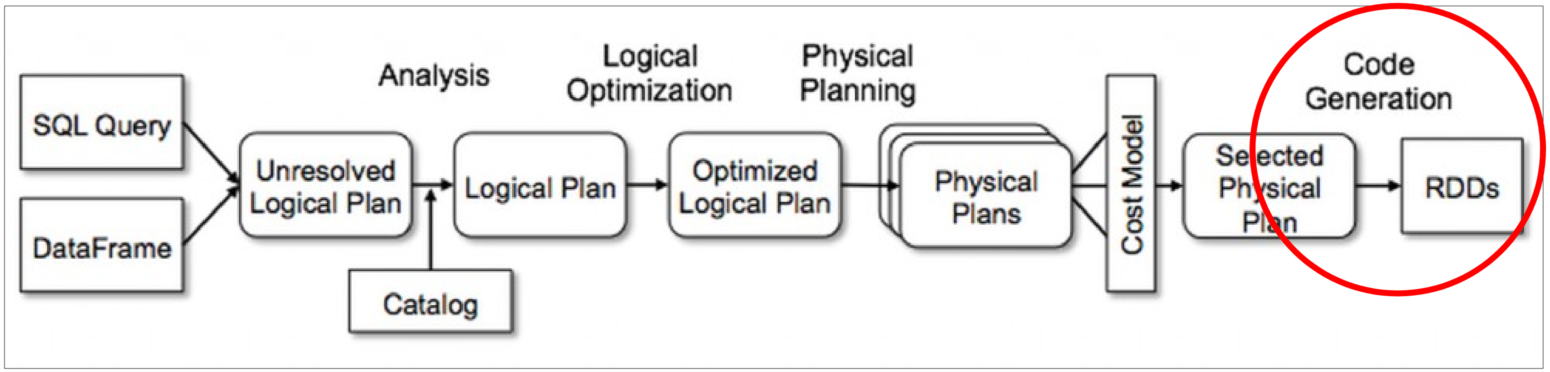
Query Plans Illustrated Using EXPLAIN


```
> val df2 = df1.filter("age > 30")
df2: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [age: bigint,
gender: string, name: string]

> df2.explain(true)
== Parsed Logical Plan ==
'Filter ('age > 30)
+- Relation[age#231L,gender#232,name#233] json

== Analyzed Logical Plan ==
age: bigint, gender: string, name: string
Filter (age#231L > cast(30 as bigint))
+- Relation[age#231L,gender#232,name#233] json

== Optimized Logical Plan ==
Filter (isnotnull(age#231L) && (age#231L > 30))
+- Relation[age#231L,gender#232,name#233] json

== Physical Plan ==
*Project [age#231L, gender#232, name#233]
+- *Filter (isnotnull(age#231L) && (age#231L > 30))
   +- *FileScan json [age#231L,gender#232,name#233] Batched: false, Format:
JSON, Location: InMemoryFileIndex[file:data/people.json], PartitionFilters:
[], PushedFilters: [IsNotNull(age), GreaterThan(age,30)], ReadSchema:
struct<age:bigint,gender:string,name:string>
```

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Code Generation

- As a final step, Catalyst may generate code for execution plans
- This is done using Janino compiler
- Codegen can really boost performance (sometimes 10x) for some queries!



Lab: DataFrames

Lab

- **Overview**:
  In this lab, we'll create a data frame from a JSON file.
  - We'll examine it and do some basic querying on it.

- **Builds on previous labs**:
  None

- **Approximate time**:
  20-30 minutes

- **Instructions**:
  4-DataFrame/4.1-DataFrame.md

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Spark SQL

DataFrames

Working with DataFrames

➔ **Spark SQL**

Dataset

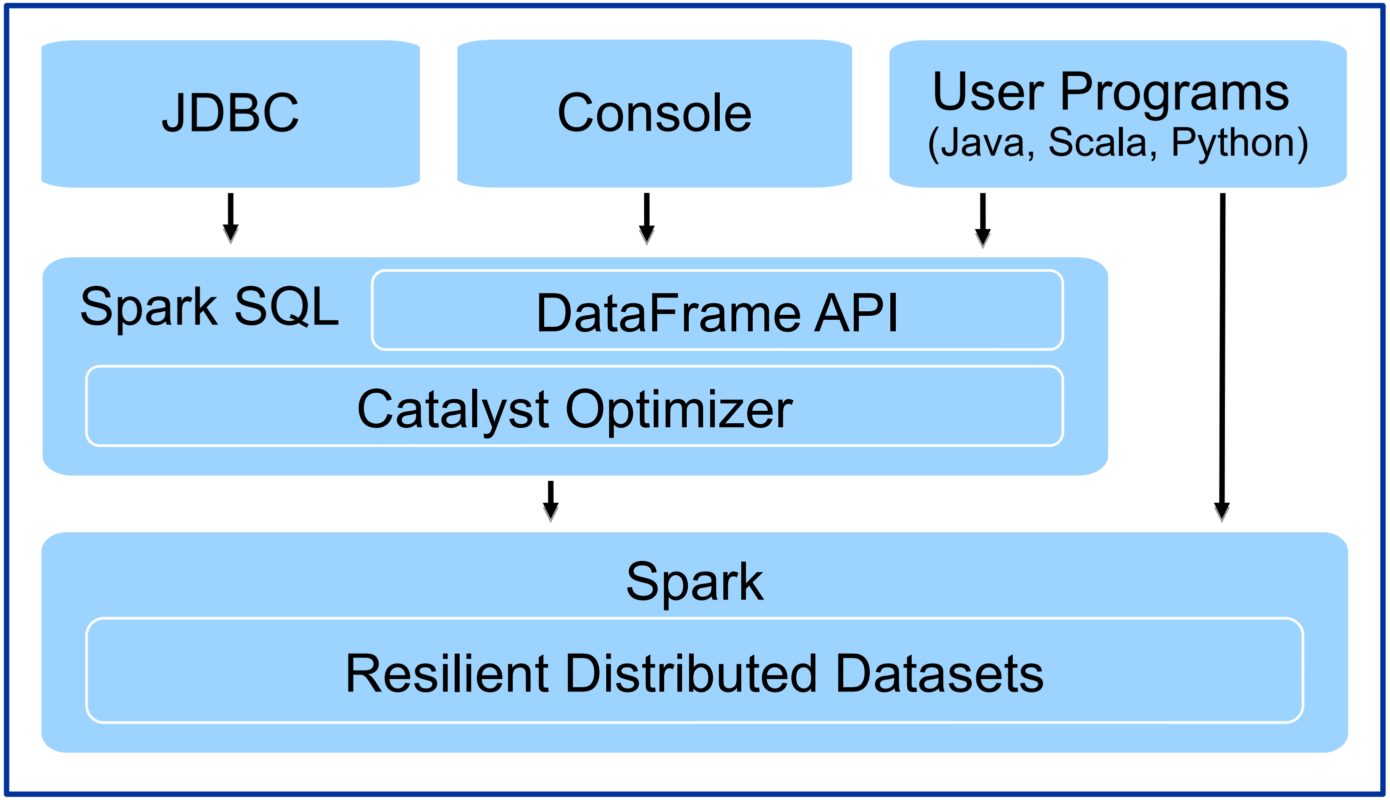
Spark and Hive

Data Formats

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Spark SQL

- **Easy**: Query using SQL (SQL2003 compliance in Spark v2)
- **Performant:** Catalyst Optimizer translates SQL into efficient queries
- Built on DataFrames
- Provides external SQL interfaces via JDBC/ODBC and a console



Source: "Spark SQL: Relational Data Processing in Spark" by Michael Armbrust, et al.

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Why Spark SQL?

- Spark API (Scala, Java or Python) is still fairly complex
  - Easier to code than Hadoop/MR, but still a large API
  - Difficult to model some areas via the standard Spark API
- It is hard to do specific optimizations
  - The objects and transformation functions are opaque to Spark
  - So Spark does mainly generalized optimizations
- Many data pipelines require both relational and procedural querying capabilities
  - The lack of SQL capabilities limits Spark
- SQL and relational querying are well known
  - And there is a lot of data in a format to a relational interface

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Querying DataFrames Using SQL (Scala)

```
// Step 1: read a DataFrame
val df = spark.read.json("people.json”)

// Step 2: Register DF as temporary table
df.createOrReplaceTempView("people")
// in 1.6 use: df.registerTempTable("people")

// Step 3: Query away
spark.sql("select * from people").show()
+---+------+----+
|age|gender|name|
+---+------+----+
| 35|     M|John|
| 40|     F|Jane|
| 18|     M|Mike|
| 19|     F| Sue|
+---+------+----+

spark.sql("select * from people where age > 30").show()
+---+------+----+
|age|gender|name|
+---+------+----+
| 35|     M|John|
| 40|     F|Jane|
+---+------+----+
```

{"name": "John", "age": 35, "gender" : "M", "weight": 200.5 }
{"name": "Jane", "age": 40, "gender" : "F" , "weight": 150.2}
{"name": "Mike", "age": 18, "gender" : "M" , "weight": 120}
{"name": "Sue", "age": 19, "gender" : "F" , "weight": 100}

Querying DataFrames Using SQL (Python)


```
// Step 1: read a DataFrame
df = spark.read.json("people.json")

// Step 2: Register DF as temporary table
df.createOrReplaceTempView("people")
// in 1.6 use: df.registerTempTable("people")

// Step 3: Query away
spark.sql("select * from people").show()
+---+------+----+
|age|gender|name|
+---+------+----+
| 35|     M|John|
| 40|     F|Jane|
| 18|     M|Mike|
| 19|     F| Sue|
+---+------+----+

spark.sql("select * from people where age > 30").show()
+---+------+----+
|age|gender|name|
+---+------+----+
| 35|     M|John|
| 40|     F|Jane|
+---+------+----+
```

{"name": "John", "age": 35, "gender" : "M", "weight": 200.5 }
{"name": "Jane", "age": 40, "gender" : "F" , "weight": 150.2}
{"name": "Mike", "age": 18, "gender" : "M" , "weight": 120}
{"name": "Sue", "age": 19, "gender" : "F" , "weight": 100}

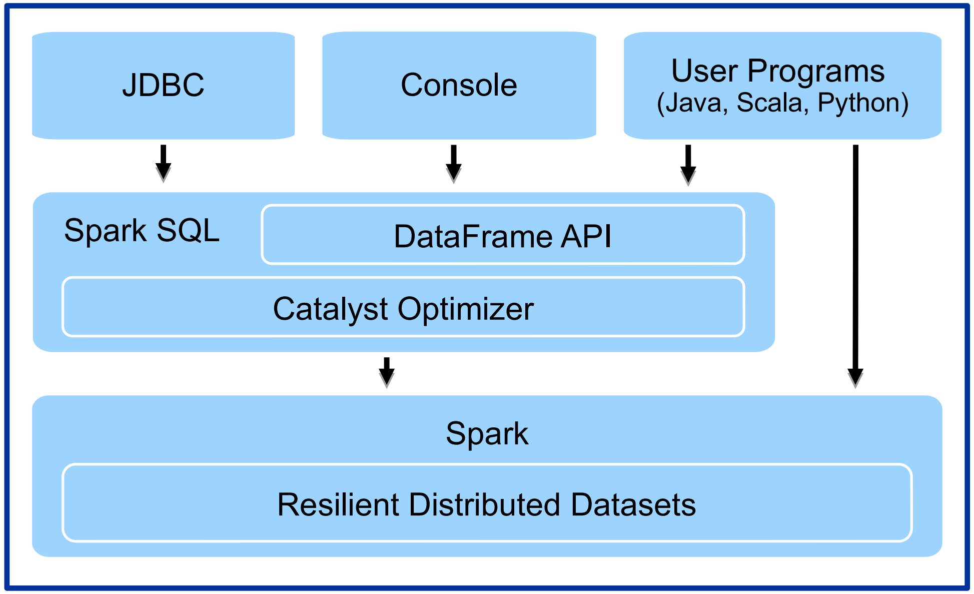
Table Scope

- Temporary View
  - Use **createOrReplaceTempView**
  - Table is only valid during the scope of current session
- Global Temporary View
  - Can outlast the session that created it
  - Until the end of Spark application
  - Use **createGlobalTempView**
- Persistent Table
  - Can be saved using Hive metastore
  - Use df.write.**saveAsTable** command

```
df.createGlobalTempView("people")

// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()

spark.newSession().sql("SELECT * FROM global_temp.people").show()

df.saveAsTable("hiveTable")
```

Should I use DataFrames Instead of RDD ?

- YES! (whenever possible)
- Dataframes are faster
- Dataframes are easier
- They are applicable to most use cases



Source: "Spark SQL: Relational Data Processing in Spark" by Michael Armbrust, et al.

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Lab: Spark SQL

Lab

- **Overview**:
  In this lab, we'll work with SQL queries instead of the DSL.
  - While you're building your DataFrames, examine some of them using explain(true) to see the query plan for them.
- **Builds on previous labs**:
  None
- **Approximate time**:
  20-30 minutes
- **Instructions**:
  - **Standalone (4.2) : 04-DataFrame/4.2-sql.md**
  - **Hadoop Env: 04-DataFrame/4.2H-spark-sql-hadoop.md**

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Dataset

DataFrames

Working with DataFrames

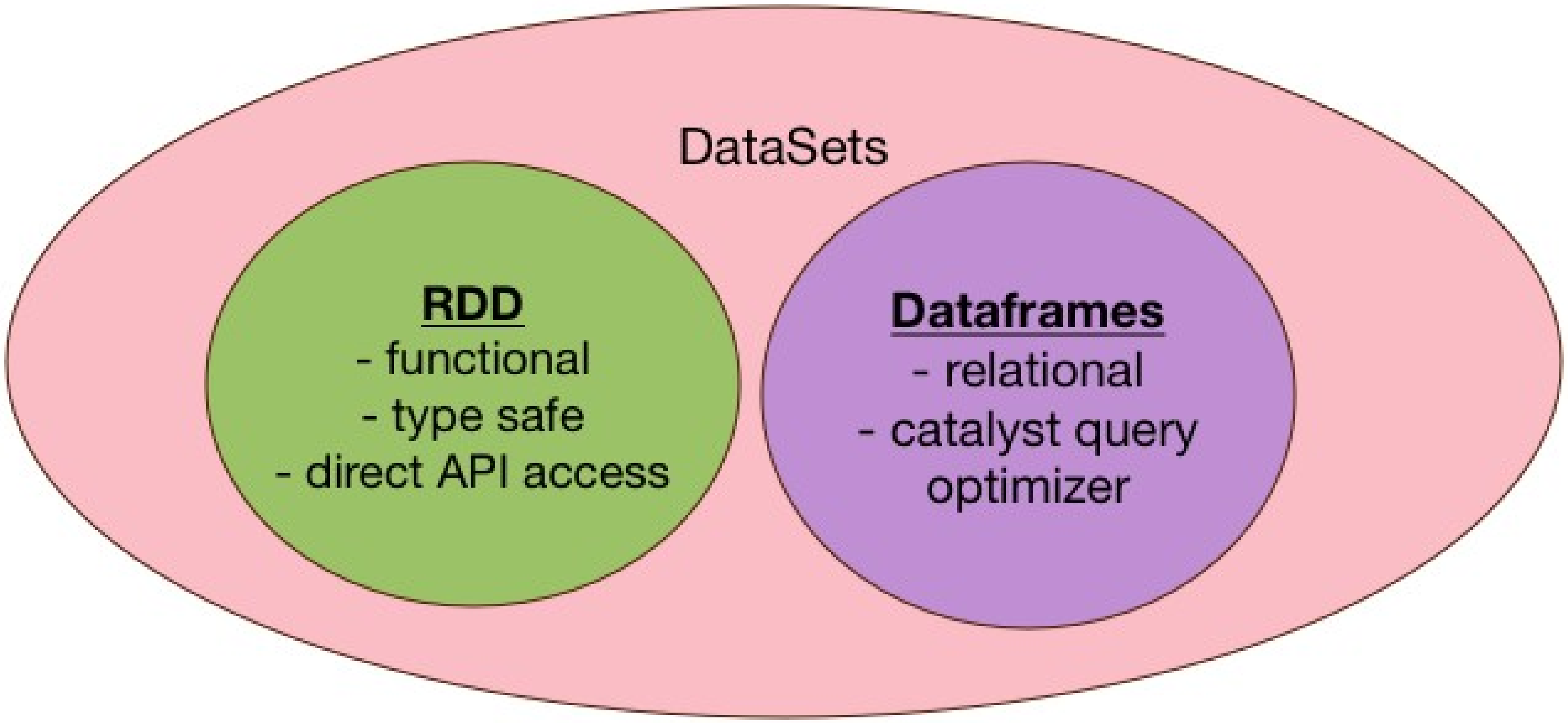
Spark SQL

➔ Dataset

Spark and Hive

Data Formats

Dataset



Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Dataset

- Dataset unifies RDD and DataFrame

- Type:
  - Strongly typed: Dataset[T] (Dataset[Person])
  - Untyped: Dataset [Row]

- DataFrame is an alias for untyped generic object
  - **DataFrame = Dataset [Row]**

- Dataset API available in Java/Scala
  - Not in Python
  - Since Python is dynamic it can not support strongly typed Dataset operations
  - most features of generic Dataset APIs are available in Python however

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Creating a Dataset

- Most common way to create a Dataset is to read a data source (CSV/JSON/Parquet)
- We can also programmatically create a DS
- Convert an existing RDD into a DS

```
// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._

// create programmatically
> val ds = spark.createDataset(List("one","two","three"))
org.apache.spark.sql.Dataset[String] = [value: string]

// read a file
> val df = spark.read.json("people.json")
df: org.apache.spark.sql.DataFrame = [age: bigint, gender: string ... 1 more field]

// converting from RDD
> val rdd: RDD[Person] = ??? // assume this RDD and 'Person' class exists
> val dataset: Dataset[Person] = spark.createDataset[Person](rdd)
```

Dataset Usage


```
// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._

// create a DataFrame & query
val df = spark.read.json("people.json")
df: org.apache.spark.sql.DataFrame = [age: bigint, gender: string ... 1 more
field]
df.filter("age > 20").show

// read CSV directly, infer columns from header
val df2 = spark.read.option("header", "true").csv("data/people2.csv")
df2: org.apache.spark.sql.DataFrame = [name: string, gender: string ... 1 more
field]
---------------------

// load a plain String RDD and query using functional programming
val t = spark.read.textFile("data/twinkle/sample.txt")
t: org.apache.spark.sql.Dataset[String] = [value: string]

t.filter(_.contains("twinkle")).collect
Array[String] = Array(twinkle twinkle little star, twinkle twinkle little star)

-----------------------
```

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Dataset and Schema

- Dataset can **"infer"** schema from common formats like JSON/Parquet/ORC
- This is convenient but may have performance overhead
- Parquet/ORC formats store schema alongside with data
  - Inferring is very quick!
- For JSON, Spark has to parse the data to figure out the schema
  - Can be expensive on large scale
  - There is an option to "sample" the data

```
// go through all records
> val df = spark.read.json("people.json")

// go through 30%  records
> val df = spark.read
                          .option("samplingRatio", 0.3)
                          .json("people.json")

> df.printSchema
Root
 |-- age: long (nullable = true)
 |-- gender: string (nullable = true)
 |-- name: string (nullable = true)
```

Reading CSV Files as Dataset

- Starting with Spark v2.0 CSV reader is included
- If header is present, it is used to infer column names
- It will try to infer the schema by going over the data
- In our case it is inferring all columns as 'String'
- We can specify the schema when loading

```
> val p = spark.read
              .option("header", "true")
              .csv("data/people2.csv")

> p.columns
Array[String] = Array(name, gender, age)

> p.printSchema
root
 |-- name: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- age: string (nullable = true)
```

```
name,gender,age
John,M,35
Jane,F,40
Mike,M,18
Sue,F,19
```

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Specifying Schema 1 (Scala)


```
// This is used to implicitly convert an RDD to a DataFrame.import
spark.implicits._
import org.apache.spark.sql._
import org.apache.spark.sql.types._

// StructField(name: String, dataType: DataType, nullable: Boolean = true,
metadata: Metadata = Metadata.empty)
// both 'DataTypes.IntegerType' and 'IntegerType' will work


val nameField = StructField("name", StringType)
val genderField = StructField("gender", StringType)
val ageField = StructField("age", IntegerType)

val peopleSchema = StructType(Array(nameField, genderField, ageField))

val peopleDF = spark.read.
               option("header", "true").
               schema(peopleSchema).
               csv("data/people2.csv")
peopleDF.printSchema
Root
 |-- name: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- age: integer (nullable = true)
```

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Specifying Schema 1 (Python)


```
// StructField(name: String, dataType: DataType, nullable: Boolean = true,
metadata: Metadata = Metadata.empty)
// both 'DataTypes.IntegerType' and 'IntegerType' will work


nameField = StructField("name", StringType(), True)
genderField = StructField("gender", StringType(), True)
ageField = StructField("age", IntegerType(), True)

peopleSchema = StructType([nameField, genderField, ageField])

peopleDF = spark.read.
               option("header", "true").
               schema(peopleSchema).
               csv("data/people2.csv")

peopleDF.printSchema()
Root
 |-- name: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- age: integer (nullable = true)
```

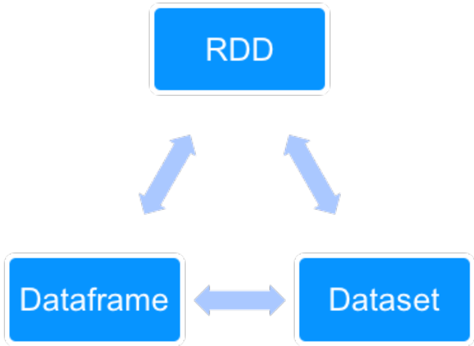
Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Specifying Schema 2 (Scala)

```
// This is used to implicitly convert an RDD to a DataFrame.
import spark.implicits._
import org.apache.spark.sql._
import org.apache.spark.sql.types._

// define schema
final case class Person (
    name: String,
    gender: String,
    age:Integer)

// read as simple text
val p = spark.sparkContext.textFile("data/people.csv")
p: org.apache.spark.rdd.RDD[String]

// turn it into RDD[Person]
val peopleRDD = p.map (line => {
                       val tokens = line.split(",")
                       val name = tokens(0)
                       val gender = tokens(1)
                       val age = tokens(2).toInt
                       new Person (name, gender, age) // return Person
                     })
peopleRDD: org.apache.spark.rdd.RDD[Person] = MapPartitionsRDD[4]

// convert to Dataset
val peopleDS = peopleRDD.toDS

peopleDS: org.apache.spark.sql.Dataset[Person] = [name: string, gender: string ... 1 more field]
```

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Inter-operating RDD/DataFrame/Dataset (Scala)




```
// This is used to implicitly convert an RDD to a DataFrame.
import spark.implicits._
import org.apache.spark.sql._
import org.apache.spark.sql.types._

> peopleRDD
peopleRDD: org.apache.spark.rdd.RDD[Person] = MapPartitionsRDD[4]

// ==== convert RDD to Dataset
> val peopleDS = peopleRDD.toDS
org.apache.spark.sql.Dataset[Person] = [name: string, gender: string ... 1 more field]
// another approach
> val peopleDS2 = spark.createDataset[Person](peopleRDD)

// === Access RDD in Dataset
> peopleDS.rdd
org.apache.spark.rdd.RDD[Person] = MapPartitionsRDD[47]

// === convert Dataset to DataFrame
> val df2 = peopleDS.toDF
df2: org.apache.spark.sql.DataFrame = [name: string, gender: string ... 1 more field]

// === convert DataFrame to Dataset
> val ds2 = df2.as[Person]
ds2: org.apache.spark.sql.Dataset[Person] = [name: string, gender: string ... 1 more field]

// DataFrame  & RDD
> df2.rdd
org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[51]
```

Lab: Dataset

Lab

- **Overview**:
  Work with Dataset
- **Builds on previous labs**:
  None
- **Approximate time**:
  20-30 minutes
- **Instructions:**
  - **Standalone (4.3): 04-DataFrame/4.3-dataset.md**
  - **Hadoop: 04-DataFrame/4.3H-dataset-hadoop.md**
- **Solution:**
  - **Spark-solutions/04-DataFrame/4.3-dataset-solution.md**

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Lab: Caching in SQL

Lab

- **Overview**:
  Try caching on Datasets and DataFrames

- **Approximate time**:
  20-30 minutes

- **Instructions:**
  - **04-DataFrame/4.4-caching-2-sql.md**

- **Solution:**

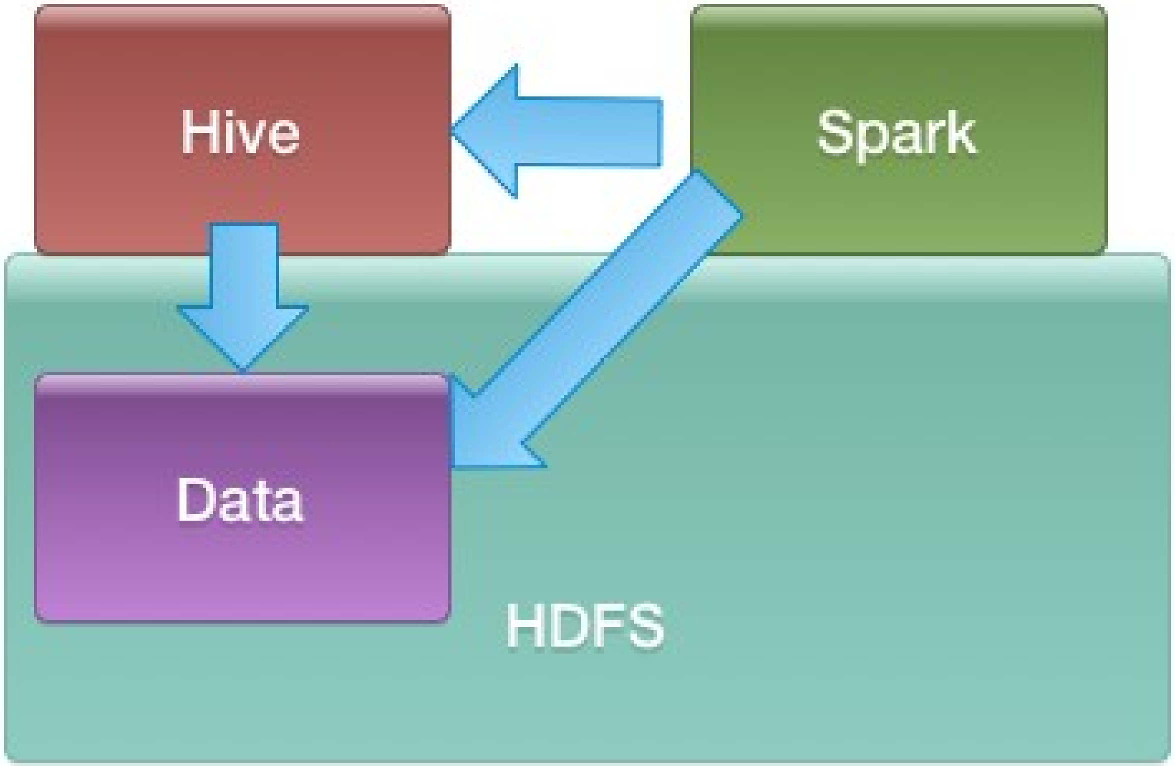
Spark and Hive

DataFrames
Working with DataFrames
Spark SQL
Dataset
➔ Spark and Hive
Data Formats

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Spark and Hive

- Spark can load data directly from HDFS
(See previous labs)
- Spark can also **natively** read/write data from Hive tables



Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Accessing Hive Tables from Spark (V1.6)

```
// hive shell
hive> show tables;
clickstream
```

```
// Spark shell

> sqlContext.tableNames
clickstream

> val t = sqlContext.table("clickstream")
> t.printSchema
'clickstream'schema will be printed

> t.show
'clickstream'data will be shown

> sqlContext.sql("select * from clickstream limit 10").show

> sqlContext.sql("select action, count(*) as total from
clickstream group by action").show
```

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @

2019-03-12

Spark Catalog (V2 and Later)

- Catalog is the interface to work with a *metastore*
  - Hive/temporary tables
- Package: **org.apache.spark.sql.catalog**

Package org.apache.spark.sql.catalog

Class Summary

Class	Description
Catalog	Catalog interface for Spark.
Column	A column in Spark, as returned by `listColumns` method in **`Catalog`**.
Database	A database in Spark, as returned by the `listDatabases` method defined in **`Catalog`**.
Function	A user-defined function in Spark, as returned by `listFunctions` method in **`Catalog`**.
Table	A table in Spark, as returned by the `listTables` method in **`Catalog`**.

Spark Catalog Usage


```
> spark.catalog
org.apache.spark.sql.catalog.Catalog

> spark.catalog.[TAB]
cacheTable            dropGlobalTempView   getTable        listTables
 uncacheTableclearCache             dropTempView         isCached
refreshByPathcreateExternalTable   functionExists       listColumns
refreshTablecurrentDatabase       getDatabase          listDatabases
setCurrentDatabasedatabaseExists       getFunction          listFunctions   tableExists

> spark.catalog.listDatabases.show(false)
+-------+---------------------+-------------------------+
|name   |description          |locationUri              |
+-------+---------------------+-------------------------+
|default|Default Hive database|file:/user/hive/warehouse|
+-------+---------------------+-------------------------+

> spark.catalog.listTables.show(false)
+--------+---------+-----------+-----------+-----------+
|name    |database |description|tableType  |isTemporary|
+--------+---------+-----------+-----------+-----------+
| people | null    |           | TEMPORARY |  true     |
| logs   | default |           | MANAGED   |  false    |
+--------+---------+-----------+-----------+-----------+
```

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Accessing Hive Tables from Spark (V2 later)

```
// hive shell
hive> show tables;
clickstream
```

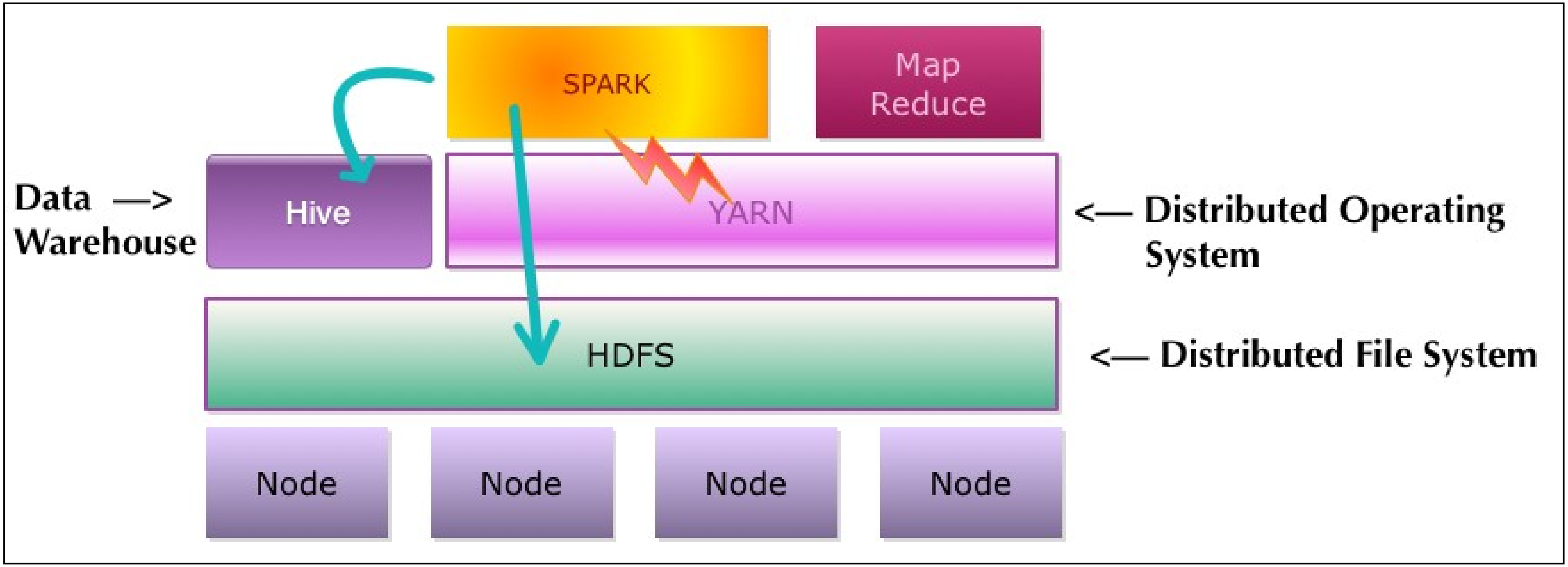
```
// Spark shell

> spark.catalog.listTables.show
clickstream

> val t = spark.catalog.getTable("clickstream")
org.apache.spark.sql.catalog.Table

> spark.sql("select * from clickstream limit 10").show
+---+------+----+
|age|gender|name|
+---+------+----+
| 35|     M|John|
| 40|     F|Jane|
+---+------+----+

> spark.sql("select action, count(*) as total from clickstream
group by action").show
```

Spark/Hive/Hadoop

- Hive is the data warehouse for Hadoop
- Using Spark, we can directly query Hive tables using SQL!
- No need to re-define tables
- Spark gives faster query times than Hive
  - Much faster than Hive on MapReduce engine
  - Slightly faster than Hive on Tez
- Save data in Hive tables

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Spark/Hadoop/Hive/YARN



Lab: Spark and Hive

Lab

- **This lab only for Hadoop environment**

- **Overview**:
  Use Spark SQL to query data from Hive tables

- **Builds on previous labs**:
  None

- **Approximate time**:
  20-30 minutes

- **Instructions**:
  - **Hadoop Env: 04-DataFrame/4.5H-spark-hive.md**

Data Formats

DataFrames

Working with DataFrames

Spark SQL

Dataset

Spark and Hive

➔ **Data Formats**

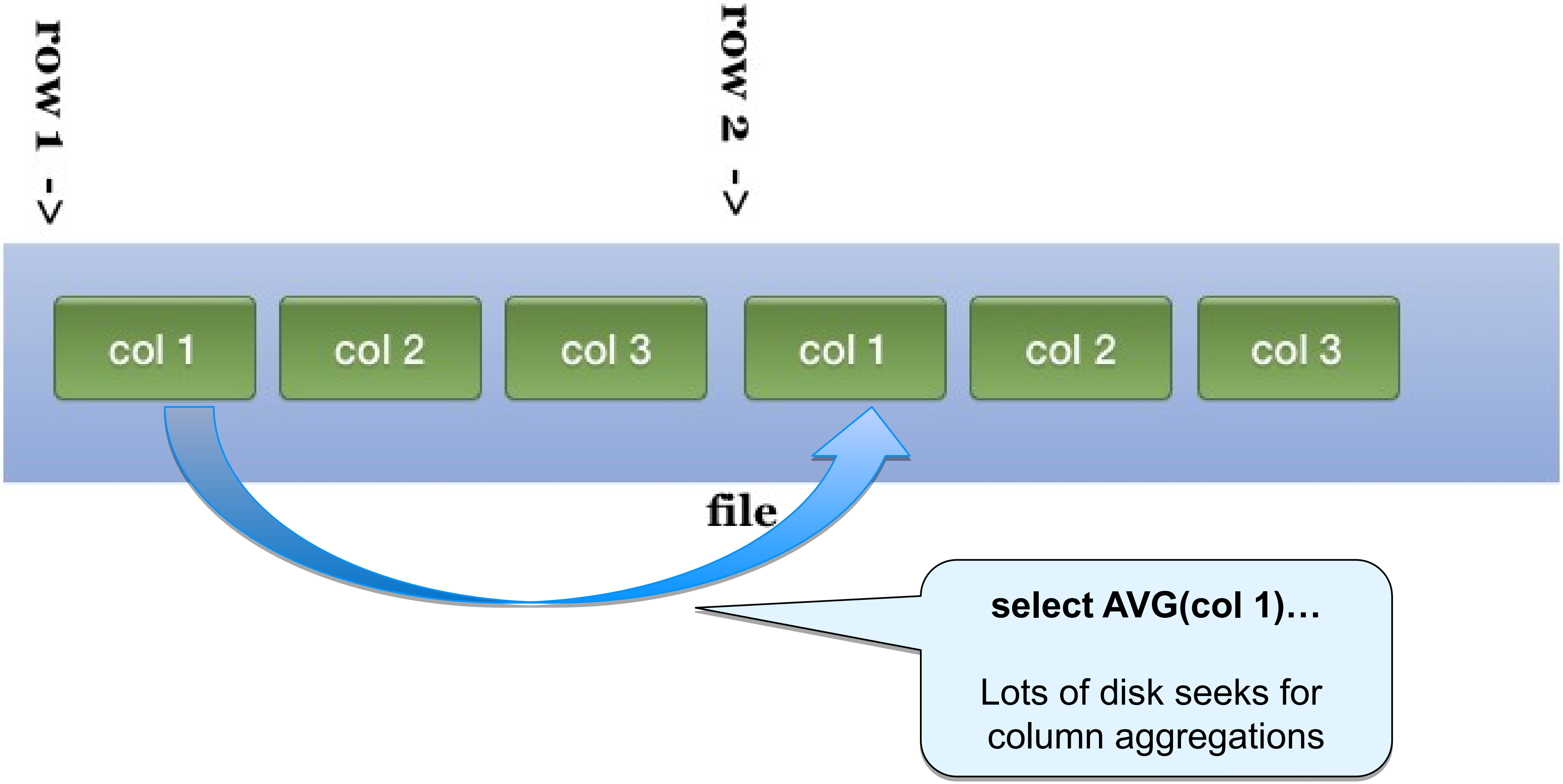
Row-Based/Column-Based Stores

- Row-Based
  - Records are stored together
  - Most RDBMS use this
  - Heavy indexing
  - Great for 'select * ' kind of query (fetching all columns)

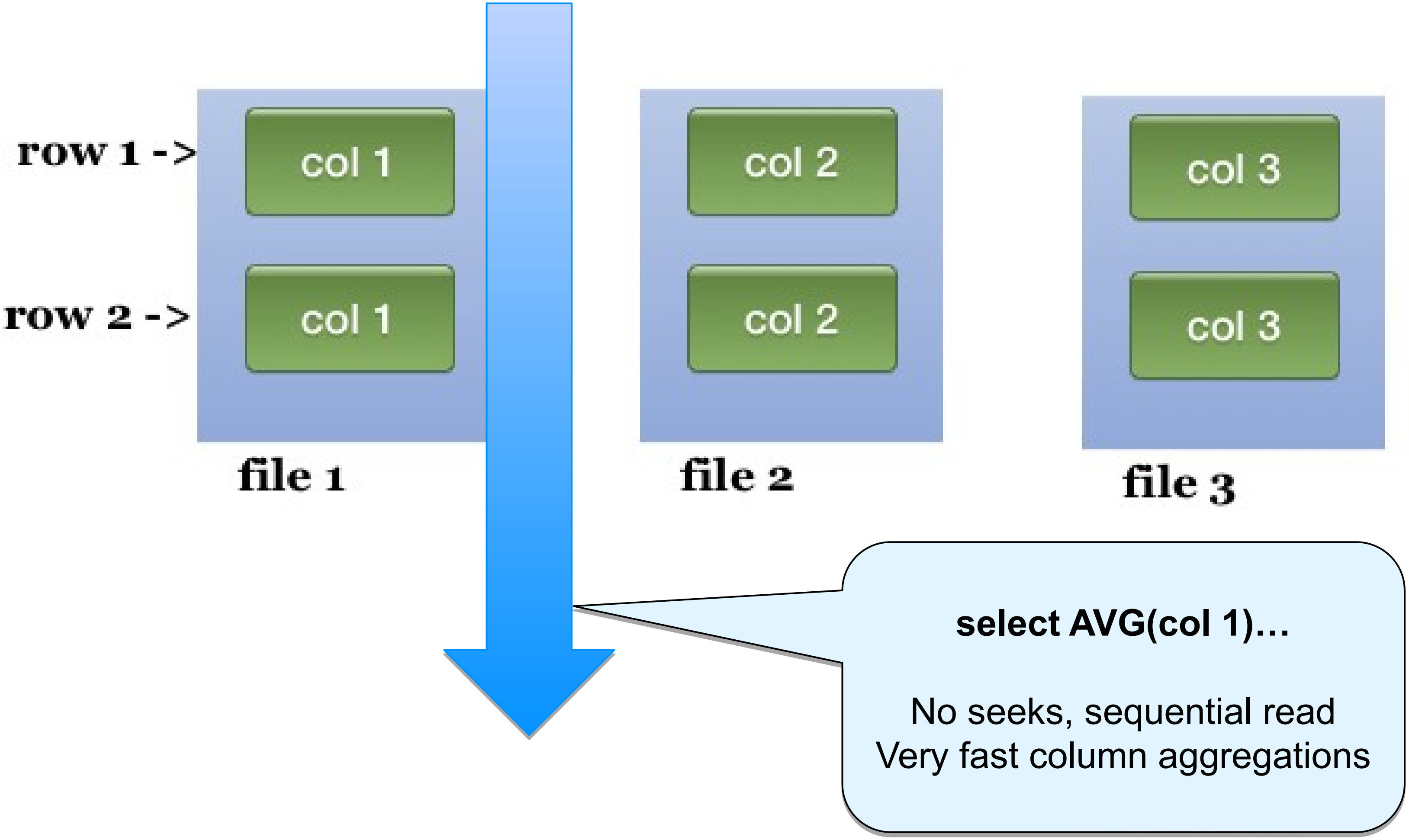
- Columnar format:
  - Stores columns physically together (instead of rows)
  - Optimized for fast column-based aggregations
    select MAX(temp) from sensors;
  - Doesn't work well for 'select * ' queries where all columns are fetched.

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Row-Based Store



Columnar Store



Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Data Formats

- Most common data formats:
  - Text
  - Binary/Sequence
  - Avro
  - Parquet
  - ORC: Optimized Row Columnar

amnachphoto/*iStock/Getty Images Plus/Getty Images*

VHS
vs.
Beta

PoppyPixels/iStock/Getty Images Plus/Getty Images

Data Format: Text

- Usual formats: CSV/JSON
  - CSV
    timestamp, ip_address,
  - JSON
    {“timestamp”: 100, “ip_address”: “1.2.3.4”}
- Pros:
  - Human-readable
  - Compatible with tools (export/import from DB for example)
- Cons:
  - Not size-efficient to store
  - Not efficient to query
  - Does not support block compression

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Data Format: Binary/Sequence

- Binary key-value pairs
- Row-based

- Pros:
  - Well-supported within Hadoop ecosystem
  - Supports block compression

- Cons:
  - Not human-readable
  - Not much support outside Hadoop ecosystem

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Data Format: Avro

- Popular serializing format
- Binary
- Row-based
- Schema is stored as part of the data
  - Decoding is easy
  - No need for separate data-dictionaries
- Supports schema evolution or schema versioning
  - Version 1 has two attributes:
    name, email
  - Version 2 has an extra attribute:
    name, email, phone
  - They can co-exist

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Data Format: Parquet

- New hot format
- Came out of Twitter + Cloudera
- Column-based storage
- Binary
- Schema stored with data
- Very efficient for column-based queries

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Data Format: Optimized Row Columnar (ORC)

- Evolution of RCFile
- Hybrid row/columnar format
  - Stores rows
  - Within rows, data is stored in columnar format
- Can support basic stats (minimum/maximum, etc.) on columns

What Format to Choose?

- Depends on:
  - Workload
  - Other ETL/ingestion systems
- Is "human readability" a big deal?
  → Text: CSV, JSON
- Speed
  → Parquet/ORC
- DataFrames natively support
  - JSON
  - Parquet
  - Avro
- Parquet/ORC is preferred format currently

Converting Between Formats

- DataFrame API allows easy migration of data formats

```
// loading json data
dfJson = spark.read.json("data.json")

// save as parquet (faster queries)
dfJson.write.parquet("data-parquet/")

// save as ORC (faster queries)
dfJson.write.orc("data-orc/")


// in 1.6, use sqlContext
// df = sqlContext.read.json("/data/data.json")
```

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Lab: Data Formats

Lab

- **Overview**:
  In this lab, we'll evaluate JSON and Parquet data formats for DataFrames
- **Builds on previous labs**:
  None
- **Approximate time**:
  20-30 minutes
- **Instructions**:
  04-DataFrame/4.6-data-formats.md

Practice Labs End-of-Day

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Practice Labs

- **Do these at the end of day.**
  Usually day-2 of Spark class (after dataframe, API sections)

- Students are encouraged to work in pairs.

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Lab: Practice Lab 1

Lab

- **Overview**:
  Analyze spark commit log data

- **Builds on previous labs**: 4.1 , 4.2, 4.3

- **Approximate time**: 20-30 minutes

- **Instructions**
  - **Practice Lab 1**

- **Solution**
  - **spark/spark-solutions/practice-labs/practice1-analyze-spark-commits-solution.md**

Lab: Practice Lab 2

Lab

- **Overview**:
  Analyze clickstream data

- **Builds on previous labs**: 4.1 , 4.2, 4.3

- **Approximate time**: 20-30 minutes

- **Instructions**
  - **Practice Lab 2**

- **Solution**
  - **spark/spark-solutions/practice-labs/practice2-analyze-clickstream-solution.md**

Review Questions

- True or False? Spark only support text formats.
- What are JSON, Parquet, ORC?
- How is Dataset related to RDD and DataFrame?

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12