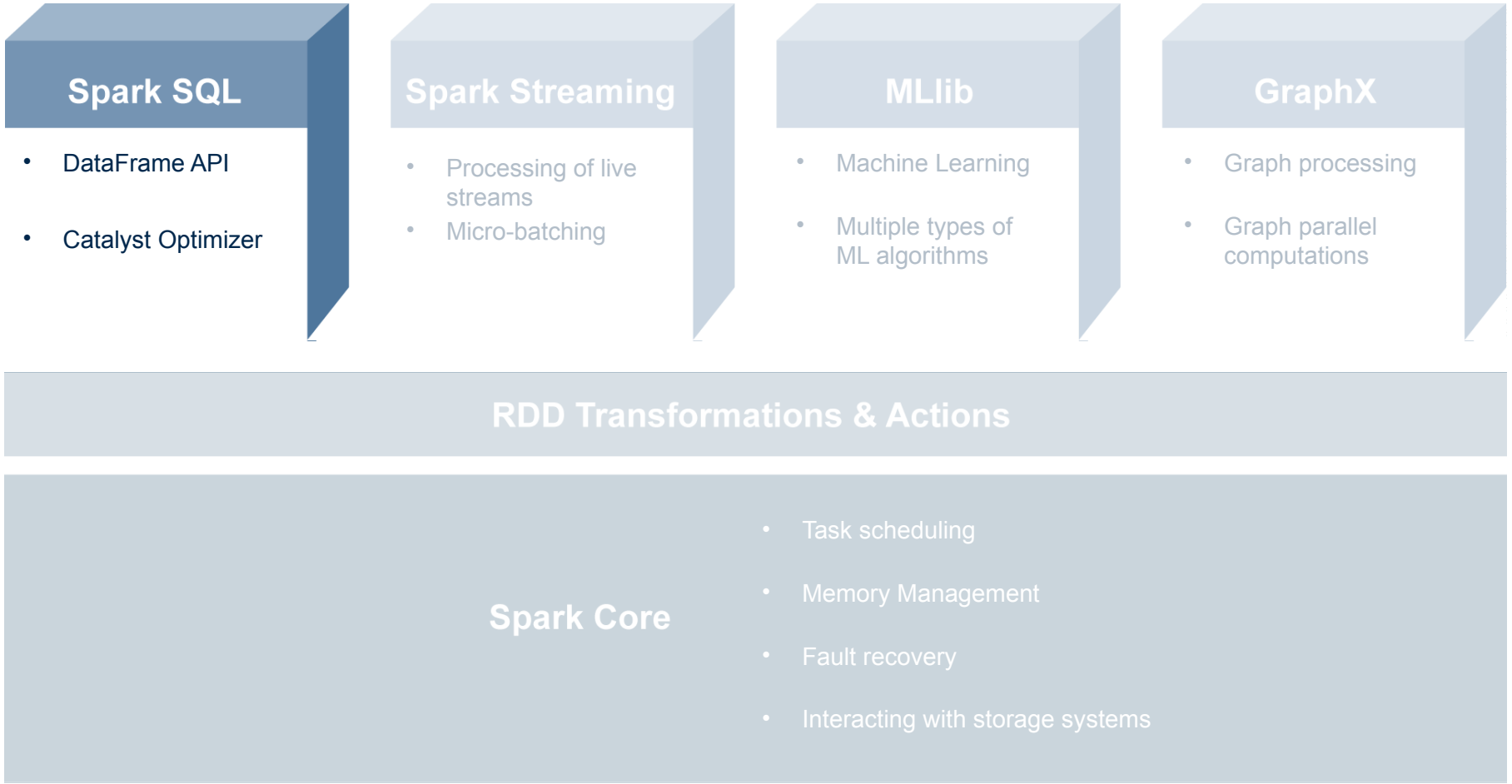
Spark Dataframe and Spark SQL

Learning Goals

1. **Create Apache Spark DataFrames**
2. Explore Data in DataFrames
3. Create User Defined Functions
4. Repartition DataFrames

Apache SparkSQL



Apache Spark DataFrames

- Programming abstraction in SparkSQL
- Distributed collection of data organized into named columns
- Supports wide array of data formats & storage systems
- Works in Scala, Python, Java & R

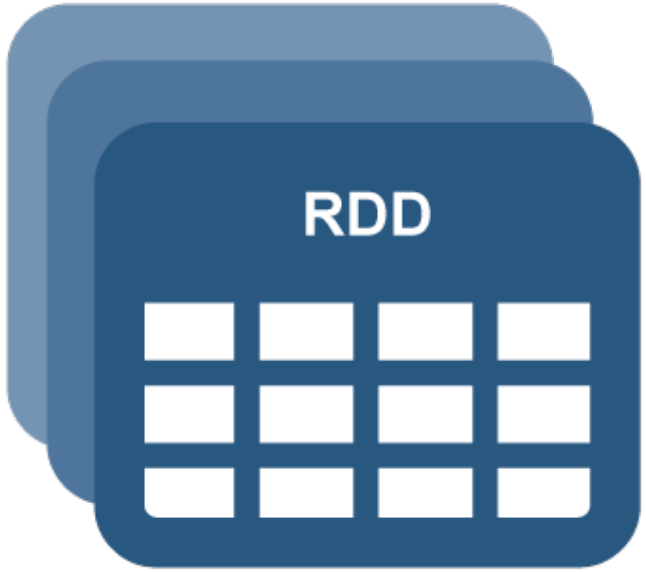
Spark DataFrames vs Spark RDD

Spark DataFrames	Spark RDD
Collection with schema	Opaque collection of objects with no information of underlying data type
Can query data using SQL	Cannot query using SQL

Discussion

What is the starting point for creating DataFrames?

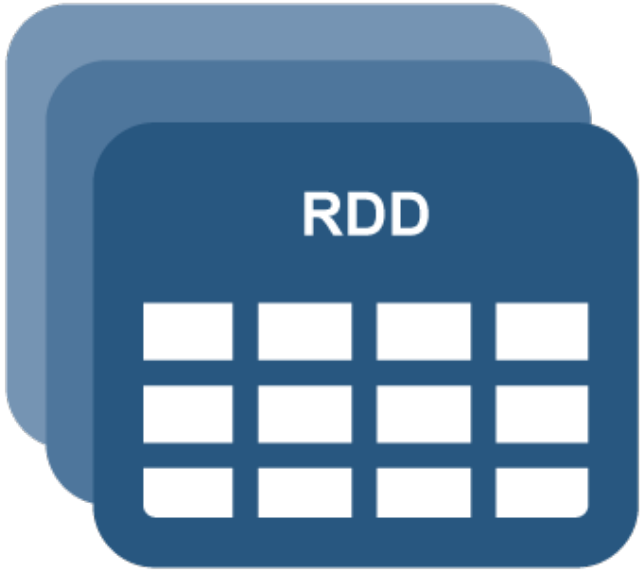
Creating DataFrames



Data Sources

- Parquet
- JSON
- Hive tables
- Relational Databases

Creating DataFrames from Existing RDD



1. **Infer schema by reflection**
   - Convert RDD containing case classes
   - Use when schema is known
2. **Construct schema programmatically**
   - Use to construct DataFrames when columns & their types not known until runtime

Infer Schema by Reflection: Case Class

- **Defines table schema**
  - Names of arguments to case class read using reflection
  - Names become name of column
- **Can be**
  - Nested
  - Contain complex data (Sequences or Arrays)

Infer Schema by Reflection

1. Import necessary classes
2. Create RDD
3. Define case class
4. Convert RDD into RDD of case objects
5. Implicitly convert resulting RDD of case objects into DataFrame
   - Apply DataFrame operations & functions to DataFrame
6. Register DataFrame as table
   - Run SQL queries on table

Example: Infer Schema by Reflection

1. **Import necessary classes**

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

2. **Create RDD**

```
val sfpdRDD =
sc.textFile("/path/to/file/sfpd.csv”).map(inc=>inc.split(","))
```

Example: Infer Schema by Reflection:

3. Define case class

```
case class Incidents(incidentnum:String,
category:String, description:String, dayofweek:String,
date:String, time:String, pddistrict:String,
resolution:String, address:String, X:Float, Y:Float,
pdid:String)
```

4. Convert RDD into RDD of case objects

```
val sfpdCase = sfpdRDD.map(inc=>Incidents(inc(0), inc(1),
inc(2), inc(3), inc(4),inc(5), inc(6), inc(7), inc(8),
inc(9).toFloat, inc(10).toFloat, inc(11)))
```

Example: Infer Schema by Reflection

5. **Implicitly convert resulting RDD of case objects into DataFrame**

```
val sfpdDF = sfpdCase.toDF()
```

6. **Register DataFrame as table**

```
sfpdDF.registerTempTable("sfpd")
```

Construct Schema Programmatically

1. Create a Row RDD from original RDD
2. Create schema separately using:
   - StructType → table
   - StructField → field
3. Create DataFrame by applying schema to Row RDD

Note:
Used when:
- Case class cannot be defined ahead of time
- More than 22 fields as case classes (Scala)

Example: Construct Schema Programmatically

Sample Data

150599321 Thursday **7/9/15** 23:45 **CENTRAL**

156168837 Thursday **7/9/15** 23:45 **CENTRAL**

150599321 Thursday **7/9/15** 23:45 **CENTRAL**

Import Classes

```
import sqlContext.implicits._
import org.apache.spark.sql._
import org.apache.spark.sql.types._
```

Example: Construct Schema Programmatically

1. **Create Row RDD from input RDD**

```
val rowRDD = sc.textFile("/user/user01/data/test.txt")
.map(x=>x.split(" "))
.map(p=>Row(p(0),p(2),p(4)))
```

Example: Construct Schema Programmatically

2. Create schema separately

```
    val testsch = StructType(Array(StructField("IncNum",
StringType,true), \ StructField("Date",StringType,true), \
StructField("District",StringType,true)))
```

Example: Construct Schema Programmatically

3. Create DataFrame

```
val testDF = sqlContext.createDataFrame(rowRDD,schema)
```

Register the DataFrame as a table

```
testDF.registerTempTable("test")

val incs = sql(“SELECT * FROM test”)
```

Create DataFrames from Data Sources

Operate on variety of data sources through DataFrame interface

- Parquet
- JSON
- Hive tables
- Relational Databases (in Spark 1.4)

Generic “load” Method

- **Generic load method**

```
sqlContext.load("/path/to/sfpd.parquet")
```

- **Specify format manually**

```
sqlContext.load("/path/to/sfpdjson", "json")
```

Methods to Load Specific Data Sources

DataFrame from database table:

```
sqlContext.jdbc
```

DataFrame from JSON file:

```
sqlContext.jsonFile
```

DataFrame from RDD containing JSON objects:

```
sqlContext.jsonRDD
```

DataFrame from parquet file:

```
sqlContext.parquetFile
```

Methods to Load Data Sources: Spark 1.4 Onwards

```
sqlContext.read.load("path/to/file/filename.parquet")

sqlContext.read.format("json").load(”/path/to/file/filename.json)
```

Learning Goals

1. Create Apache Spark DataFrames
2. **Explore Data in DataFrames**
3. Create User Defined Functions
4. Repartition DataFrames

Exploring the Data

- What are the top five addresses with most incidents?
- What are the top five districts with most incidents?
- What are the top 10 resolutions?
- What are the top 10 categories of incidents?

DataFrame Operations

- DataFrame Actions
- DataFrame Functions
- Language Integrated Queries

DataFrame Actions

Action	Description
`collect()`	Returns array containing all rows in DataFrame
`count()`	Returns number of rows in DataFrame
`describe(cols:String*)`	Computes statistics for numeric columns (count, mean, stddev, min and max)
`first()`	Returns the first row
`head()`	Returns the first row
`show()`	Displays first 20 rows
`take(n:int)`	Returns the first n rows

DataFrame Functions

Function	Description
`cache()`	Cache this DataFrame
`columns`	Returns an array of all column names
`printSchema()`	Prints schema to console in tree format
`persist()`	Persist this DataFrame
`explain()`	Prints physical plan (Catalyst optimizer) to console

Language Integrated Queries

L I Query	Description
`agg(expr, exprs)`	Aggregates on entire DataFrame
`distinct`	Returns new DataFrame with unique rows
`except(other)`	Returns new DataFrame with rows from this DataFrame not in other DataFrame
`filter(expr); where(condition)`	Filter based on the SQL expression or condition
`groupBy(cols: Columns)`	Groups DataFrame using specified columns
`join (DataFrame, joinExpr)`	Joins with another DataFrame using given join expression
`sort(sortcol)`	Returns new DataFrame sorted by specified column
`select(col)`	Selects set of columns

Top Five Addresses with Most Incidents

1. `val incByAdd=sfpdDF.groupBy("address”)`

2. `val numAdd=incByAdd.count`

3. `val numAddDesc=numAdd.sort($"count".desc)`

4. `val top5Add=numAddDesc.show(5)`

Top Five Addresses with Most Incidents

```
val incByAdd = sfpdDF.groupBy("address")
                     .count
                       .sort($"count".desc)
                        .show(5)
```

address	count
800_Block_of_BRYA...	10852
800_Block_of_MARK...	3671
1000_Block_of_POT...	2027
2000_Block_of_MIS...	1585
16TH_ST/MISSION_ST	1512

Top Five Addresses with Most Incidents: SQL

```
val top5Addresses = sqlContext.sql("SELECT address,
count(incidentnum) AS inccount FROM sfpd GROUP BY address ORDER
BY inccount DESC LIMIT 5”).show
```

address	inccount
800_Block_of_BRYA...	10852
800_Block_of_MARK...	3671
1000_Block_of_POT...	2027
2000_Block_of_MIS...	1585
16TH_ST/MISSION_ST	1512

Exploring the Data

- ? What are the top five addresses with most incidents?
- What are the top five districts with most incidents?
- What are the top 10 resolutions?
- What are the top 10 categories of incidents?

Output Operations: `save()`

Operation	Description
`Save` (source, mode, options)	Saves contents of DataFrame based on given data sources, savemode and set of options
`insertIntoJDBC` (url,name, overwrite)	Saves contents of DataFrame to JDBC at url under table name table
`saveAsParquetFile` (path)	Saves contents of DataFrame as parquet file
`saveAsTable` (tablename, source, mode, options)	Creates a table from contents of DataFrame using data source, options & mode

Output Operations: `save()`

To save contents of top5Addresses DataFrame:

```
top5Addresses.toJSON.saveAsTextFile("/user/user01/test")
```

```
{"address":"800_Block_of_BRYANT_ST","inccount":10852}
{"address":"800_Block_of_MARKET_ST","inccount":3671}
{"address":"1000_Block_of_POTRERO_AV","inccount":2027}
{"address":"2000_Block_of_MISSION_ST","inccount":1585}
{"address":"16TH_ST/MISSION_ST","inccount":1512}
```

Output Operations – Spark 1.4 Onwards

To save contents of top5Addresses DataFrame:

```
top5Addresses.write.format("json").mode("overw
rite").save("/user/user01/test")
```

Learning Goals

1. Create Apache Spark DataFrames
2. Explore data in DataFrames
3. **Create User Defined Functions**
4. Repartition DataFrames

User Defined Functions (UDF)

- UDFs allow developers to define custom functions
- In Spark, can define UDF inline
- No complicated registration or packaging process
- Two types of UDF:
  - To use with Scala DSL (DataFrame operations)
  - To use with SQL

User Defined Functions (Scala DSL)

- Inline creation
  - Use `udf()`
- Function can be used with DF operations

```
val func1 =  udf((arguments)=>{function definition})
```

User Defined Functions in SQL Query

```
def funcname
sqlContext.udf.register("funcname", funcname _)
```

partially applied function in Scala

Inline registration and creation:

```
sqlContext.udf.register("funcname", func def)
```

Example: Want to Find Incidents by Year

- Date in the format: “dd/mm/yy”
- Need to extract string after last slash
- Can then compute incidents by year

Example: Want to Find Incidents by Year

Note:
This is
Scala DSL

- **Defining UDF**

```
val getStr = udf((s:String)=>{
val lastS = s.substring(s.lastIndexOf('/')+1)
lastS
})
```

- **Use UDF in DataFrame Operations**

```
val yy = sfpdDF.groupBy(getStr(sfpdDF("date")))
            .count
            .show
```

Example: Want to Find Incidents by Year

Note:
This is
Scala DSL

scalaUDF(date)	count
13	152830
14	150185
15	80760

Example: Want to Find Incidents by Year

Note:
This is SQL

- Define UDF

```
def getStr(s:String) = {val
strAfter=s.substring(s.lastIndexOf('/')+1)
strafter}
```

- Register UDF

```
sqlContext.udf.register("getStr", getStr _)
```

Example: Want to Find Incidents by Year

Note:
This is SQL

- **Define & register UDF**

```
sqlContext.udf.register("getStr", (s:String)=>{
val strAfter=s.substring(s.lastIndexOf('/')+1)
strafter
})
```

- **Use in SQL statement**

```
    val numIncByYear = sqlContext.sql("SELECT getStr(date),
count(incidentnum) AS countbyyear
    FROM sfpd GROUP BY getStr(date)
    ORDER BY countbyyear DESC
    LIMIT 5")
```

Example: Want to Find Incidents by Year

Note:
This is SQL

```
numIncByYear.foreach(println)
[13,152830]



[14,150185]
[15,80760]
```

Learning Goals

1. Create Apache Spark DataFrames
2. Explore data in DataFrames
3. Create User Defined Functions
4. **Repartition DataFrames**

Partition DataFrames

DataFrame with 4 Partitions

P1			P2			P3			P4		
Incnum (Str)	Category (Str)	PdDistrict (Str)	Incnum (Str)	Category (Str)	PdDistrict (Str)	Incnum (Str)	Category (Str)	PdDistrict (Str)	Incnum (Str)	Category (Str)	PdDistrict (Str)
150598981	ASSAULT	CENTRAL	150599183	ASSAULT	SOUTHERN	150597701	ASSAULT	MISSION	150597400	ROBBERY	TARAVAL
150599161	BURGLARY	PARK	150599246	ASSAULT	CENTRAL	150597701	ROBBERY	INGLESIDE	150596468	FRAUD	SOUTHERN
150599127	SUSPICIOUS	SOUTHERN	150599246	WARRANTS	CENTRAL	150597701	ASSAULT	SOUTHERN	150597234	BURGLARY	SOUTHERN
150603455	VANDALISM	NORTHERN	150599246	WARRANTS	CENTRAL	150597591	ROBBERY	SOUTHERN	150596468	FRAUD	TARAVAL

Partition DataFrames

- Sets number of partitions in DataFrame after shuffle:
  `spark.sql.shuffle.partitions`
- Default value set to 200
- Can change parameter using:
  `sqlContext.setConf(key, value)`

Why Repartition

- Internally SparkSQL partitions data for joins and aggregations

- If applying other RDD operations on result of DataFrame operations, can manually control partitioning

repartition(numPartitions)

- To repartition, use
  - `df.repartition(numPartitions)`
- To determine current number of partitions:
  - `df.rdd.partitions.size`

Best Practices

- Specifying the number of partitions:
- Want each partition to be 50 MB – 200 MB
- Small dataset → few partitions
- Large cluster with 100 nodes → at least 100 partitions
- Example:
  - 100 nodes with 10 slots in each executor
  - Want 1000 partitions to use all executor slots