CS 696 Intro to Big Data: Tools and Methods
Fall Semester, 2016
Doc 25 Spark 2
Nov 29, 2016

Transformations

```
JavaSparkContext sc = new JavaSparkContext();
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1,2,3,3));
```

rdd.map(x -> x + 1)	[2, 3, 4, 4]
rdd.distinct()	[1, 2, 3]
rdd.sample(false,0.5);	varies

Transformations

```
JavaSparkContext sc = new JavaSparkContext();
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3));
JavaRDD<Integer> other = sc.parallelize(Arrays.asList(3, 4, 5));
```

rdd.union(other)	[1, 2, 3, 3, 4, 5]
rdd.intersection(other)	[3]
rdd.subtract(other)	[1, 2]

Transformations & Output

```
JavaSparkContext sc = new JavaSparkContext();
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3));
JavaRDD<Integer> other = sc.parallelize(Arrays.asList(3, 4, 5));
JavaRDD<Integer> result = rdd.union(other);

result.saveAsTextFile(somePath);
```

Actions & Output

```
JavaSparkContext sc = new JavaSparkContext();
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3));
JavaRDD<Integer> other = sc.parallelize(Arrays.asList(3, 4, 5));
JavaRDD<Integer> result = rdd.union(other);

List onMaster = result.collect();
```

List is java.util.List - no spark method saveAsTextFile

Writing to HDFS

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.Path;

static void saveAsTextFile(String destination,  Object contents) throws Exception {
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(destination), conf);
    FSDataOutputStream out = fs.create(new Path(destination));
    out.writeChars(contents.toString());
    out.close();
  }
```

destination

- /full/path/to/file.txt
- local/path/file.txt
- s2://bucket/file.txt

Actions

```
JavaSparkContext sc = new JavaSparkContext();
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(3, 1, 2, 3));
```

rdd.collect()	[3,1, 2, 3]
rdd.reduce((a,b) -> a + b)	9
rdd.first	3
rdd.top(2)	[3, 1]
rdd.take(2)	[3, 1]
rdd.takeOrdered(2)	[1, 2]
rdd.count()	3
rdd.countByValue()	{3=2, 1=1, 2=1}

Actions on DoubleRDD

```
JavaSparkContext sc = new JavaSparkContext();
JavaDoubleRDD rdd = sc.parallelizeDoubles(Arrays.asList(5.0,1.0,2.0,3.0,4.0,5.0));
```

rdd.mean()
rdd.min()
rdd.sum()
rdd.variance()
rdd.stdev()
rdd.stats()
rdd.histogram(bucketCount)

JavaPairRDD

RDD on key-value pairs

```
JavaSparkContext sc = new JavaSparkContext();
JavaRDD a = sc.parallelize(Arrays.asList(1, 3, 3));
JavaRDD b = sc.parallelize(Arrays.asList(2, 4, 6));

JavaPairRDD<Integer, Integer> rdd = a.zip(b);

rdd.collect();              // [(1,2), (3,4), (3,6)]
```

Transformations on Pair RDD

rdd.collect(); // [(1,2), (3,4), (3,6)]

rdd.reduceByKey((x, y) -> x + y)	[(1, 2), (3, 10)]
rdd.groupByKey()	[(1, [2]), (3, [4, 6])]
rdd.mapValues(x -> x + 1)	
rdd.top(2)	[3, 1]
rdd.keys()	[1, 3, 6]
rdd.values()	[2, 4, 6]
rdd.sortByKey()	

Transformations on Pair RDD

rdd [(1,2), (3,4), (3,6)]

other [(3,9)]

rdd.subtractByKey(other)	[(1,2)]
rdd.join(other)	[(3, (4, 9)), (3, (6,9))]
rdd.cogroup(other)	

Shared Variables

BroadCast Variables

Read-only variable on each machine

Sent from master

```
Broadcast<int[]> broadcastVar = sc.broadcast(new int[] {1, 2, 3});
```

Accumulators

Write-only on each machine - using add()

Master can read

```
LongAccumulator accum = jsc.sc().longAccumulator();
```

Some Debugging help

```
40        JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1,2,3,3));
41        JavaRDD<Integer> result = rdd.map(x -> x + 2);
42        String test = result.toDebugString();
```

```
test

(2) MapPartitionsRDD[1] at map at JavaWordCount.java:41 []
 |  ParallelCollectionRDD[0] at parallelize at JavaWordCount.java:40 []
```

Spark vs Hadoop MapReduce

Hadoop MapReduce

Implements two common functional programming functions is OO manner

Spark

Uses set of common functional programming functions in a functional way

Spark SQL

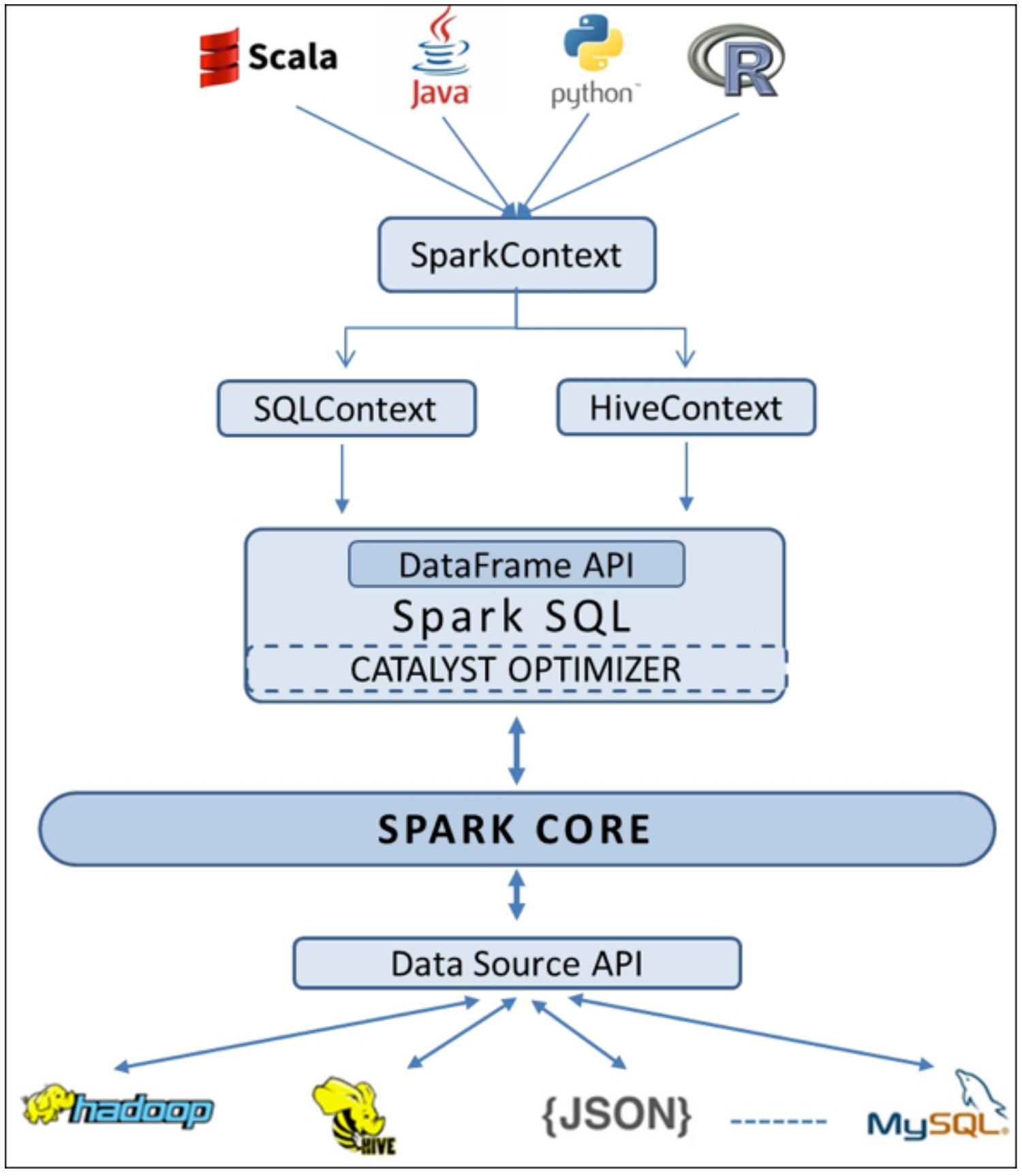
Spark SQL

Structured data processing

Structure allow extra optimizations via Spark SQL optimized catalyst engine

Structured data
- SQL
- Datasets & Dataframes

Data Sources
- Hive (distributed data warehouse)
- Parquet (compressed column data)
- SQL databases
- JSON, CSV, text files
- RDD


Scala
Java
python
R
SparkContext
SQLContext
HiveContext
DataFrame API
Spark SQL
CATALYST OPTIMIZER
SPARK CORE
Data Source API
hadoop
HIVE
{JSON}
MySQL

Datasets & DataFrames

Dataset

- Distributed collection of structured data
- Like RDD but uses Spark SQL optimized execution engine
- Scala, Java, JVM languages only (no python or R)

DataFrame

- Dataset organized into named columns
- Like SQL table or dataframe in Julia

Dataset<Row> in Java

Creating DataFrames

```
SparkSession spark = SparkSession
        .builder()
        .appName("JavaWordCount")
        .getOrCreate();

Dataset<Row> df = spark.read().json(“people.json”);
df.show();
```

```
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

people.json

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

Reading CSV

people.csv

```
name,age
Andy,30
Justin,19
Michael,
```

```
SparkSession spark = SparkSession
        .builder()
        .appName("JavaWordCount")
        .getOrCreate();

DataFrameReader reader = spark.read();
reader.option("header",true);
reader.option("inferSchema",true);
Dataset<Row> df = reader.csv(args[0]);
df.show();
df.printSchema();
```

```
+-------+----+
|   name| age|
+-------+----+
|   Andy|  30|
| Justin|  19|
|Michael|null|
+-------+----+
```

```
root
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
```

Types We Can Read

csv
jdbc
json
orc
parquet
text

Some CSV options

encoding
sep (erator)
header
inferSchema
ignoreLeadingWhiteSpace
nullValue
dateFormat
timeStampFormat

mode
- PERMISSIVE - sets record field on corrupt record
- DROPMALFORMED - ignores whole corrupt records
- FAILFAST - throw exception on corrupt record

Dataset Functions common with RDDs

collect
count
distinct
filter
flatmap
foreach
groupBy
map
reduce

We can work with columns

people.csv

```
name,age
Andy,30
Justin,19
Michael,
```

```
import static org.apache.spark.sql.functions.col;

Dataset<Row> df = reader.csv(“people.csv”);
df.select("name").show();
df.select(col("name")).show();
```

```
+-------+
|   name|
+-------+
|   Andy|
| Justin|
|Michael|
+-------+
```

```
// Select everybody, but increment the age by 1
df.select(col("name"), col("age").plus(1)).show();
```

```
+-------+---------+
|   name|(age + 1)|
+-------+---------+
|   Andy|       31|
| Justin|       20|
|Michael|     null|
+-------+---------+
```

```
// Select people older than 21
df.filter(col("age").gt(21)).show();
```

```
+----+---+
|name|age|
+----+---+
|Andy|  30|
+----+---+
```

```
// Count people by age
df.groupBy("age").count().show();
```

```
+----+-----+
| age|count|
+----+-----+
|null|    1|
|  19|    1|
|  30|    1|
+----+-----+
```

Using SQL

```
df.createOrReplaceTempView("people");

Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
sqlDF.show();
```

```
+-------+----+
|   name| age|
+-------+----+
|   Andy|  30|
| Justin|  19|
|Michael|null|
+-------+----+
```

Saving DataFrames

```
df.write().format("json").save("people2.json");
```

```
{"name":"Andy","age":30}
{"name":"Justin","age":19}
{"name":"Michael"}
```

Formats

json, parquet, jdbc, orc, libsvm, csv, text

DataFrames & Objects

Can define encoders/decoders so can have dataframes/datasets of objects

Efficient transmission on network

Efficient in memory allocation

DataFrames & RDDs

Can convert RDDs to dataframes/datasets -
But need schema for dataFrame

Converting DataFame to RDD

```
fd.toJavaRDD()
```

More Statistics

DataFrameStatFunctions stat = df.stat();

approxQuantile	
bloomFilter	
corr	(Pearson Correlation Coeffient)
cov	(Covariance)
freqItems	find frequent items in column(s)

Spark vs Hadoop MapReduce

Hadoop MapReduce

- Implements two common functional programming functions is OO manner

Spark

- Uses set of common functional programming functions in a functional way
- Reads more data formats
- SQL
- More queries
- Interacts with SQL databases directly
- 10 to 100 times faster

But there is more - machine learning!

MLib

Linear Algebra

Statistics

- Correlations, Hypothesis testing, Sampling, density estimation

Classification & regression

- Linear, logistic, isotonic, & ridge regression
- Decision trees, naive Bayes

Collaborative filtering (recommender systems)

Clustering

Dimensionality reduction

Feature extraction

Frequent pattern mining