Low-Level Spark: RDDs

Objective

Resilient Distributed Datasets

Essential proficiency

- Read from external sources
- Convert to/from DataFrames and Datasets
- Difference between RDDs/DataFrames/Datasets

RDDs

Distributed typed collections of JVM objects

The "first citizens" of Spark: all higher-level APIs reduce to RDDs

Pros: can be highly optimized

- partitioning can be controlled
- order of elements can be controlled
- order of operations matters for performance

Cons: hard to work with

- for complex operations, need to know the internals of Spark
- poor APIs for quick data processing

For 99% of operations, use the DataFrame/Dataset APIs

RDDs vs ~~DataFrames~~ Datasets

DataFrame == Dataset[Row]

In common

- collection API: *map, flatMap, filter, take, reduce* etc
- union, count, distinct
- groupBy, sortBy

RDDs over Datasets

- partition control: *repartition, coalesce, partitioner, zipPartitions, mapPartitions*
- operation control: *checkpoint, isCheckpointed, localCheckpoint, cache*
- storage control: *cache, getStorageLevel, persist*

Datasets over RDDs

- select and join!
- Spark planning/optimization before running code

For 99% of operations, use the DataFrame/Dataset APIs

Takeaways

Turn a regular collection into an RDD

```
val sc = spark.sparkContext
```

```
val numbersRDD = sc.parallelize(regularCollection)
```

Read RDD from file

```
val numbersRDD = sc.textFile("path/to/your/file").map(...)
```

need to process lines

Dataset to RDD

```
val stocksRDD = stocksDS.rdd
```

all Datasets can access underlying RDDs

RDD to high-level

```
val stocksDF = stocksRDD.toDF("symbol", "date", "price")
val stocksDS = spark.createDataset(stocksRDD)
```

RDDs vs ~~DataFrames~~ Datasets

Spark rocks