CS 378
Big Data Programming

Lecture 27

DataFrames and Datasets

Review

- Assignment 11: User sessions in Spark

DataFrames

- Early Spark versions had a SchemaRDD
  - As the name implies, the RDD understands the fields of the contained data (named columns), and their types
- Elements of the RDD: instances of Row
  - Equivalent to a data base table row
- DataFrames replaced SchemaRDDs
  - Can query data using SQL in code
  - Can access data using standard DB connectors
  - And offer all the Spark RDD capabilities
  - Equivalent to a DB table, or R or Python data frame

DataFrames

- DataFrames can be created from:
  - External sources
  - Results of queries
  - From other RDDs

- In addition to the utility of the table paradigm

- Spark implementors have used the schema data for various runtime optimizations
  - Faster performance (optimized code)
  - Efficiently serialize/deserialize
  - Smaller memory requirements
    - Operate on binary or storage optimized data

Datasets

- Spark 2.0 features Datasets
  - Specific Datasets API for Java, Scala
  - Offers compile-time checking of syntax and semantic errors

- No Python API for Datasets
  - DataFrame = Datasets[Row]
  - Essentially an “untyped” Dataset
  - Since Python not strongly typed, you don’t get the compile time checking (there is no compile step)

Datasets as RDDs

- Datasets have the characteristics of RDDs
  - Immutable – you don't modify an RDD, you transform it
  - Distributed – data can be partitioned across multiple machines
  - Distributed – computation distributed across multiple machines
  - Lazy evaluation

- Dataset Java documentation link:

  http://spark.apache.org/docs/latest/api/java/index.html

Dataset/Dataframe Creation

- Read a CSV file with header line:

- Python

```
sqlContext.read.format('com.databricks.spark.csv')
    .options(header='true', inferSchema='true')
    .load('filename')
```

  - Example

- Java

```
sparkSession.read().csv(inputFilename)
```

Dataframe Creation (Python)

- From an RDD – given an RDD containing lists:

```
df = rdd.toDF([‘field1’, ‘field2’, … ])
```

Or from an RDD of lists:

```
df = sqlContext.createDataFrame(rdd,
    [‘field1’, ‘field2’, … ])
```

Or for an RDD of Rows, namedtuple, or dictionary:

```
df = sqlContext.createDataFrame(rdd)
```

Or first create RDD of Rows, then make the DataFrame

```
rowRDD = rdd.map(lambda x: Row(field1=x[0],
    field2=x[1], ...))
df = sqlContext.createDataFrame(rowRDD)
```

Dataframe Creation

- Previous examples in Python used sqlContext

- In Java, replace sqlContext with an instance of class `SparkSession`

- `SparkSession` documentation:

  http://spark.apache.org/docs/2.0.0-preview/api/java/org/apache/spark/sql/SparkSession.html

Row (Python)

- Create a Row using named arguments:

```
row = Row(make = "Ford", model="F-150")
```

Fields sorted by name

- Fields in a Row can be accessed by:

```
row['make']
```

Or

```
row.make
```

- Ask if a field is in a Row object

```
'make' in row
```

Row (Python)

- Convert a Row to a dictionary (Python):

```
row = Row(vin=ABC,
          desc=Row(make="Ford", model="F-150"))
```

- Convert top level

```
row.asDict()
```

Only converts the outer-most Row

- Convert recursive

```
row.asDict(True)
```

Converts Row objects recursively

SQL Query against a Dataframe

- Register a table name (Python)

```
sqlContext.registerDataFrameAsTable(df, "table1")
```

Or

```
df.registerTempTable("table1")
```

- Query against the table

```
df2 = sqlContext.sql(
  "SELECT field1 as f1, field2 as f2 FROM table1")
```

- Resulting DataFrame (`df2`) has Rows with fields:

```
f1, f2
```

SQL Query against a Dataframe

- Register a table name (Java)

```
sparkSession.createTempView("table1", df)
```

- Query against the table

```
df2 = sparkSession.sql(
  "SELECT field1 as f1, field2 as f2 FROM table1")
```

- Resulting DataFrame (`df2`) has Rows with fields:

```
f1, f2
```

Save a Dataframe to CSV

- Output to a CSV file (Python)

```
(df.repartition(1)
    .write
    .format("csv")
    .save(outputLocation))
```

- Output to a CSV file (Java)

```
df.repartition(1)
    .write()
    .format("csv")
    .save(outputLocation)
```