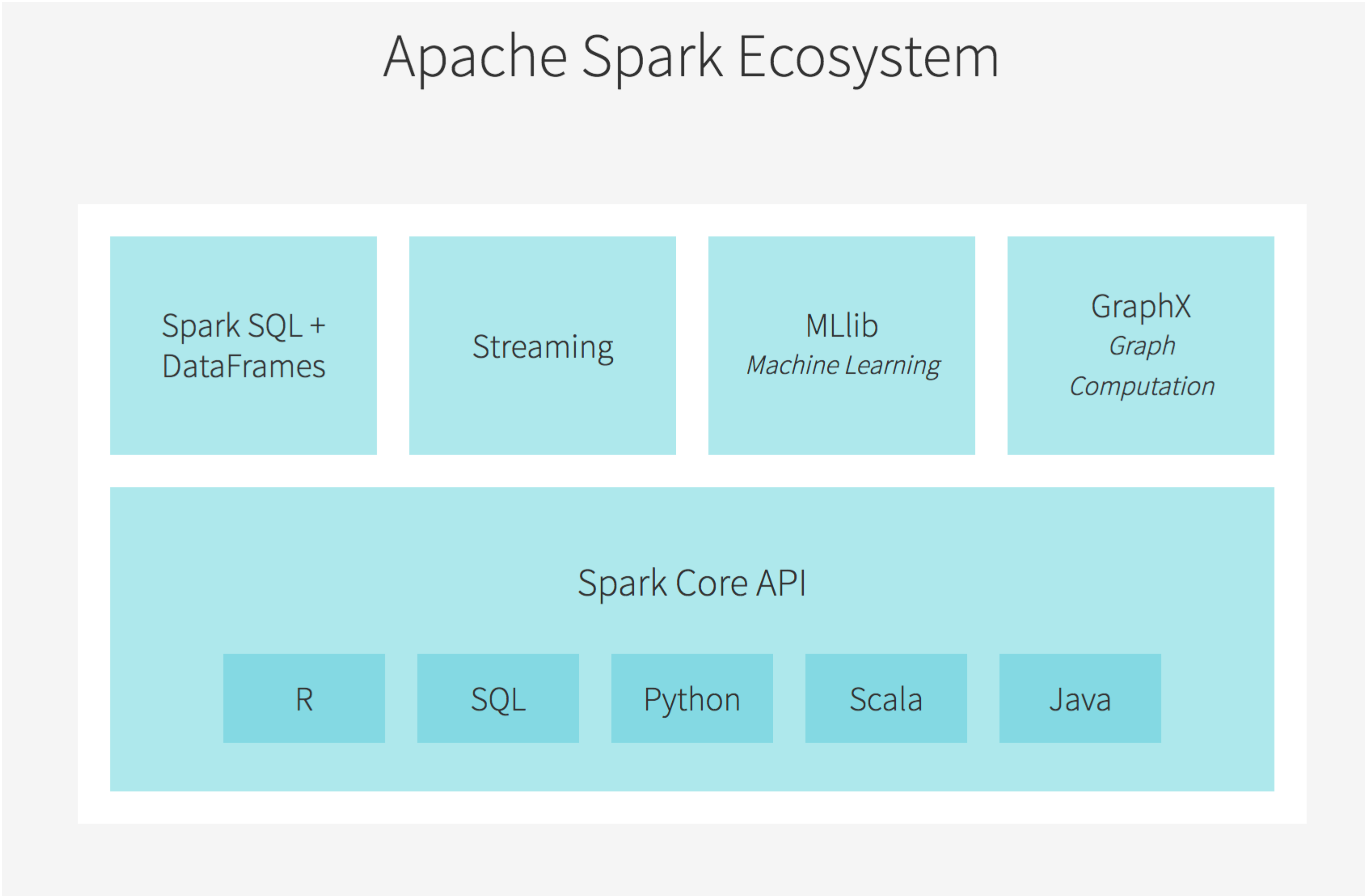
Class 8: SparkSQL

New York University

Summer 2017

Agenda

1. Spark SQL and the SQL Context
2. Creating DataFrames
3. Transforming and Querying DataFrames
4. Saving DataFrames
5. DataFrames and RDDs
6. Comparing Spark SQL, Impala and Hive-on-Spark



https://databricks.com/spark/about

- **What is Spark SQL?**
  - Spark module for structured data processing
  - Replaces Shark (a prior Spark module, now deprecated)
  - Built on top of core Spark

- **What does Spark SQL provide?**
  - The DataFrame API – a library for working with data as tables
    - DataFrames contain data organized as Rows and Columns
  - A SQL Engine and command line interface

- **The main Spark SQL entry point is a SQL Context object**
  - Built on Spark Context
  - Akin to Spark Context in core Spark
- **There are two implementations**
  - **`SQLContext`**
  - **`HiveContext`**
    - Reads and writes Hive tables directly
    - Supports HiveQL
    - Enables unmodified Hadoop Hive queries to run up to 100x faster on existing deployments and data

- Spark SQL is used for processing *structured data*
  - Useful for exploring data interactively
  - Used by data analysts, data scientists, business intelligence (BI) users
  - Spark SQL is a Spark module that provides a convenient programming abstraction - DataFrames
  - Integrated with Spark ecosystem of tools, e.g. Spark core and MLlib

- **SQLContext is created based on the SparkContext**

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

Agenda

1. Spark SQL and the SQL Context
2. Creating DataFrames
3. Transforming and Querying DataFrames
4. Saving DataFrames
5. DataFrames and RDDs
6. Comparing Spark SQL, Impala and Hive-on-Spark

- **DataFrames are the main abstraction in Spark SQL**
  - Analogous to RDDs in core Spark
  - A distributed collection of data
  - Data is organized into named columns
  - Built on a base RDD containing `Row` objects

- **DataFrames can be created**
  - From an existing structured data source (Parquet file, JSON file, etc.)
  - From an existing RDD
  - By performing an operation or query on another DataFrame
  - By programmatically defining a schema

```
val sqlCtx = new SQLContext(sc)
import sqlCtx._
val peopleDF = sqlCtx.jsonFile("people.json")
```

File: people.json

```
{"name":"Alice", "pcode":"94304"}
{"name":"Brayden", "age":30, "pcode":"94304"}
{"name":"Carla", "age":19, "pcode":"10036"}
{"name":"Diana", "age":46}
{"name":"Étienne", "pcode":"94104"}
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

- **Methods on the SQLContext object**
- **Convenience functions**
  - `jsonFile(filename)`
  - `parquetFile(filename)`
- **Generic base function: `load`**
  - `load(filename,source)` – load `filename` of type `source` (default Parquet)
  - `load(source,options…)` – load from `source` using options
- **Convenience functions are implemented by calling `load`**

```
jsonFile("people.json") = load("people.json", "json")
```

- **Spark SQL includes data source types such as**
  - JSON
  - Parquet
  - JDBC
- **Spark can also access data from third party data source libraries, such as**
  - Amazon Redshift
  - Amazon S3
  - Avro
  - Azure storage services
  - Cassandra
  - Couchbase
  - CSV
  - ElasticSearch
  - HBase
  - HIVE Tables
  - MongoDB
  - Oracle
  - Avro Files
  - CSV Files
  - Reading LZO Compressed Files
  - Redis
  - Riak Time Series
  - Zip Files

https://docs.databricks.com/spark/latest/data-sources/index.html

Example: Loading from a MySQL database

```
val accountsDF = sqlCtx.load("jdbc",
   Map("url"-> "jdbc:mysql://dbhost/dbname?user=…&password=…",
       "dbtable" -> "accounts"))
```

Warning: Avoid direct access to databases in production environments, which may overload the DB or be interpreted as service attacks

- Use Apache Sqoop to import instead

- **You can also use custom or third party data sources**
- **Example: Read from an Avro file using the `avro` source in the Databricks Spark Avro package**

```
$ spark-shell --packages com.databricks:spark-avro_2.10:1.0.0
> …
> val myDF =
sqlCtx.load("myfile.avro","com.databricks.spark.avro")
```

Agenda

1. Spark SQL and the SQL Context
2. Creating DataFrames
3. Transforming and Querying DataFrames
4. Saving DataFrames
5. DataFrames and RDDs
6. Comparing Spark SQL, Impala and Hive-on-Spark

- **Operations for dealing with DataFrame metadata (rather than its data)**
  - **`schema`** – returns a Schema object describing the data
  - **`printSchema`** – displays the schema as a visual tree
  - **`cache`** / **`persist`** – persists the DataFrame to disk or memory
  - **`columns`** – returns an array containing the names of the columns
  - **`dtypes`** – returns an array of (column-name,type) pairs
  - **`explain`** – prints debug information about the DataFrame to the console

- **Example: Displaying column data types using `dtypes`**

```
> val peopleDF = sqlCtx.jsonFile("people.json")
> peopleDF.dtypes.foreach(println)
(age,LongType)
(name,StringType)
(pcode,StringType)
```

- **Queries - create a new DataFrame**
  - DataFrames are immutable
  - Queries are analogous to RDD *transformations*
- **Actions - return data to the Driver**
  - Actions trigger "lazy" execution of queries

- Some DataFrame *actions*
  - **`collect`** – return all rows as an array of **`Row`** objects
  - **`take(`*`n`*`)`** – return the first **`n`** rows as an array of **`Row`** objects
  - **`count`** – return the number of rows
  - **`show(`*`n`*`)`** – display the first **`n`** rows (default=20)

```
> peopleDF.count()
res7: Long = 5

> peopleDF.show(3)
age  name    pcode
null Alice   94304
30   Brayden 94304
19   Carla   10036
```

- **DataFrame query methods return new DataFrames**
  - Queries can be chained like transformations
- **Some query methods**
  - **`join`** – joins this DataFrame with a second DataFrame
    - several variants for inside, outside, left, right, etc.
  - **`limit`** – a new DF with the first **`n`** rows of this DataFrame
  - **`select`** – a new DataFrame with data from one or more columns of the base DataFrame
  - **`filter`** – a new DataFrame with rows meeting a specified condition

▪ Example: A basic query with limit

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

```
> peopleDF.limit(3).show()
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036

Output of **show**

```
age  name    pcode
null Alice   94304
30   Brayden 94304
19   Carla   10036
```

- **Some query operations take strings containing simple query expressions**
  - Such as **`select`** and **`where`**
- **Example: `select`**

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

```
peopleDF.
  select("age")
```

age
null
30
19
46
null

```
peopleDF.
  select("name","age")
```

name	age
Alice	null
Brayden	30
Carla	19
Diana	46
Étienne	null

▪ Proving constraints with `where`

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

```
peopleDF.
  where("age > 21")
```

age	name	pcode
30	Brayden	94304
46	Diana	null

- **Some DF queries take one or more *columns* or *column expressions***
  - Required for more sophisticated operations
- **Some examples**
  - **`select`**
  - **`sort`**
  - **`join`**
  - **`where`**

- **In Scala, columns can be referenced in two ways**

```
val ageDF = peopleDF.select($"age")
```

– *OR*

```
val ageDF = peopleDF.select(peopleDF("age"))
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

age
null
30
19
46
null

- **Column references can also be *column expressions***

```
peopleDF.select(peopleDF("name"),peopleDF("age")+10)
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

name	age+10
Alice	null
Brayden	40
Carla	29
Diana	56
Étienne	null

Example: Sorting in by columns (descending)

```
peopleDF.sort(peopleDF("age").desc)
```

`.asc` and `.desc` are column expression methods used with `sort`

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

age	name	pcode
46	Diana	null
30	Brayden	94304
19	Carla	10036
null	Alice	94304
null	Étienne	94104

- Spark SQL also supports the ability to perform SQL queries
  - First, register the DataFrame as a "table" with the SQL Context

```
peopleDF.registerTempTable("people")
sqlCtx.sql("""SELECT * FROM people WHERE name LIKE "A%" """)
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

age	name	pcode
null	Alice	94304

Agenda

1. Spark SQL and the SQL Context
2. Creating DataFrames
3. Transforming and Querying DataFrames
4. Saving DataFrames
5. DataFrames and RDDs
6. Comparing Spark SQL, Impala and Hive-on-Spark

- **Data in DataFrames can be saved to a data source**
  - Built in support for JDBC and Parquet File
    - **`createJDBCTable`** – create a new table in a database
    - **`insertInto`** – save to an existing table in a database
    - **`saveAsParquetFile`** – save as a Parquet file (including schema)
    - **`saveAsTable`** – save as a Hive table (HiveContext only)
  - Can also use third party and custom data sources
    - **`save`** – generic base function

Agenda

1. Spark SQL and the SQL Context
2. Creating DataFrames
3. Transforming and Querying DataFrames
4. Saving DataFrames
5. DataFrames and RDDs
6. Comparing Spark SQL, Impala and Hive-on-Spark

- **DataFrames are built on RDDs**
  - Base RDDs contain **`Row`** objects
  - Use **`rdd`** to get the underlying RDD

```
peopleRDD = peopleDF.rdd
```

peopleDF

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

peopleRDD

Row[null,Alice,94304]
Row[30,Brayden,94304]
Row[19,Carla,10036]
Row[46,Diana,null]
Row[null,Étienne,94104]

- **Row RDDs have all the standard Spark actions and transformations**
  - Actions – **`collect`**, **`take`**, **`count`**, etc.
  - Transformations – **`map`**, **`flatMap`**, **`filter`**, etc.
- **Row RDDs can be transformed into PairRDDs to use map-reduce methods**

- **The syntax for extracting data from Rows depends on language**
- **Scala**
  - Use Array-like syntax
    - **`row(0)`** – returns element in the first column
    - **`row(1)`** – return element in the second column
    - etc.
  - Use type-specific **`get`** methods to return typed values
    - **`row.getString(`*`n`*`)`** – returns n^{th} column as a String
    - **`row.getInt(n)`** – returns n^{th} column as an Integer
    - etc.

- **Extract data from Rows**

```
val peopleRDD = peopleDF.rdd
peopleByPCode = peopleRDD.
  map(row => (row(2),row(1))).
  groupByKey())
```

Row[null,Alice,94304]
Row[30,Brayden,94304]
Row[19,Carla,10036]
Row[46,Diana,null]
Row[null,Étienne,94104]

(94304,Alice)
(94304,Brayden)
(10036,Carla)
(null,Diana)
(94104,Étienne)

(null,[Diana])
(94304,[Alice,Brayden])
(10036,[Carla])
(94104,[Étienne])

- **You can also create a DF from an RDD**
  - **`sqlCtx.createDataFrame(`*`rdd`*`)`**

Agenda

1. Spark SQL and the SQL Context
2. Creating DataFrames
3. Transforming and Querying DataFrames
4. Saving DataFrames
5. DataFrames and RDDs
6. Comparing Spark SQL, Impala and Hive-on-Spark

- **Spark SQL is built on Spark, a *general purpose* processing engine**
  - Provides convenient SQL-like access to structured data in a Spark application
- **Impala is a *specialized* SQL engine**
  - Much better performance for querying
  - Robust security via Sentry
- **Impala is better for**
  - Interactive queries
  - Data analysis
- **Use Spark SQL for**
  - ETL
  - Access to structured data required by a Spark application

- **Spark SQL**
  - Provides the DataFrame API to allow structured data processing *in a Spark application*
  - Programmers can mix SQL with procedural processing
- **Hive-on-Spark**
  - Hive provides a SQL abstraction layer over MapReduce or Spark
    - Allows non-programmers to analyze data using familiar SQL
  - Hive-on-Spark replaces MapReduce as the engine underlying Hive
    - Does not affect the user experience of Hive
    - Except many times faster queries!

Homework

See the homework packet for details.