DataFrames (1)

Big Data Analysis with Scala and Spark

Heather Miller

DataFrames

So far, we got an intuition of what `DataFrames` are, and we learned how to create them. We also saw that if we have a `DataFrame`, we use SQL syntax and do SQL queries on them.

DataFrames have their own APIs as well!

DataFrames

So far, we got an intuition of what `DataFrames` are, and we learned how to create them. We also saw that if we have a `DataFrame`, we use SQL syntax and do SQL queries on them.

DataFrames have their own APIs as well!

In this session we'll focus on the DataFrames API. We'll dig into:

- available DataFrame data types
- some basic operations on DataFrames
- aggregations on DataFrames

DataFrames: In a Nutshell

`DataFrames` are...

A relational API over Spark's RDDs

Because sometimes it is more convenient to use declarative relational APIs than functional APIs for analysis jobs.

DataFrames: In a Nutshell

`DataFrames` are...

A relational API over Spark's RDDs

Because sometimes it is more convenient to use declarative relational APIs than functional APIs for analysis jobs.

Able to be automatically aggressively optimized

Spark SQL applies years of research on relational optimizations in the databases community to Spark.

DataFrames: In a Nutshell

`DataFrames` are...

A relational API over Spark's RDDs

Because sometimes it is more convenient to use declarative relational APIs than functional APIs for analysis jobs.

Able to be automatically aggressively optimized

Spark SQL applies years of research on relational optimizations in the databases community to Spark.

Untyped!

The elements within DataFrames are `Rows`*, which are not parameterized by a type. Therefore, the Scala compiler cannot type check Spark SQL schemas in DataFrames.*

DataFrames Data Types

To enable optimization opportunities, Spark SQL's `DataFrames` operate on a restricted set of data types.

DataFrames Data Types

To enable optimization opportunities, Spark SQL's `DataFrames` operate on a restricted set of data types.

Basic Spark SQL Data Types:

Scala Type	*SQL Type*	*Details*
`Byte`	`ByteType`	1 byte signed integers (-128,127)
`Short`	`ShortType`	2 byte signed integers (-32768,32767)
`Int`	`IntegerType`	4 byte signed integers (-2147483648,2147483647)
`Long`	`LongType`	8 byte signed integers
`java.math.BigDecimal`	`DecimalType`	Arbitrary precision signed decimals
`Float`	`FloatType`	4 byte floating point number
`Double`	`DoubleType`	8 byte floating point number
`Array[Byte]`	`BinaryType`	Byte sequence values
`Boolean`	`BooleanType`	true/false
`Boolean`	`BooleanType`	true/false
`java.sql.Timestamp`	`TimestampType`	Date containing year, month, day, hour, minute, and second.
`java.sql.Date`	`DateType`	Date containing year, month, day.
`String`	`StringType`	Character string values (stored as UTF8)

DataFrames Data Types

To enable optimization opportunities, Spark SQL's `DataFrames` operate on a restricted set of data types.

Complex Spark SQL Data Types:

Scala Type	*SQL Type*
`Array[T]`	`ArrayType(elementType, containsNull)`
`Map[K, V]`	`MapType(keyType, valueType, valueContainsNull)`
`case class`	`StructType(List[StructFields])`

DataFrames Data Types

To enable optimization opportunities, Spark SQL's `DataFrames` operate on a restricted set of data types.

Complex Spark SQL Data Types:

Scala Type	*SQL Type*
`Array[T]`	`ArrayType(elementType, containsNull)`
`Map[K, V]`	`MapType(keyType, valueType, valueContainsNull)`
`case class`	`StructType(List[StructFields])`

Arrays

Array of only one type of element (`elementType`*).* `containsNull` *is set to true if the elements in* `ArrayType` *value can have null values.*

Example:

```
// Scala type
Array[String]

// SQL type
ArrayType(StringType, true)
```

DataFrames Data Types

To enable optimization opportunities, Spark SQL's `DataFrames` operate on a restricted set of data types.

Complex Spark SQL Data Types:

Scala Type	*SQL Type*
`Array[T]`	`ArrayType(elementType, containsNull)`
`Map[K, V]`	`MapType(keyType, valueType, valueContainsNull)`
`case class`	`StructType(List[StructFields])`

Maps

Map of key/value pairs with two types of elements. `valuecontainsNull` *is set to true if the elements in* `MapType` *value can have null values.*

Example:

```
// Scala type          // SQL type
Map[Int,String]     MapType(IntegerType, StringType, true)
```

DataFrames Data Types

To enable optimization opportunities, Spark SQL's `DataFrames` operate on a restricted set of data types.

Complex Spark SQL Data Types:

Scala Type	*SQL Type*
`Array[T]`	`ArrayType(elementType, containsNull)`
`Map[K, V]`	`MapType(keyType, valueType, valueContainsNull)`
`case class`	`StructType(List[StructFields])`

Structs

Struct type with list of possible fields of different types. `containsNull` *is set to true if the elements in* `StructFields` *can have null values.*

Example:

```
// Scala type
case class Person(name: String, age: Int)
```

```
// SQL type
StructType(List(StructField("name", StringType, true),
                StructField("age", IntegerType, true)))
```

Complex Data Types Can Be Combined!

It's possible to arbitrarily nest complex data types! For example:

```scala
// Scala type
case class Account(
  balance: Double,
  employees:
  Array[Employee])

case class Employee(
  id: Int,
  name: String,
  jobTitle: String)

case class Project(
  title: String,
  team: Array[Employee],
  acct: Account)
```

```
// SQL type
StructType(
  StructField(title,StringType,true),
  StructField(
    team,
    ArrayType(
      StructType(StructField(id,IntegerType,true),
      StructField(name,StringType,true),
      StructField(jobTitle,StringType,true)),
      true),
    true),
    StructField(
      acct,
      StructType(
        StructField(balance,DoubleType,true),
        StructField(
          employees,
          ArrayType(
            StructType(StructField(id,IntegerType,true),
            StructField(name,StringType,true),
            StructField(jobTitle,StringType,true)),
            true),
          true)
        ),
      true)
    )
```

Accessing Spark SQL Types

Important.

In order to access *any* of these data types, either basic or complex, you must first import Spark SQL types!

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

DataFrames Operations Are More Structured!

When introduced, the DataFrames API introduced a number of relational operations.

The main difference between the RDD API and the DataFrames API was that DataFrame APIs accept Spark SQL expressions, instead of arbitrary user-defined function literals like we were used to on RDDs. This allows the optimizer to understand what the computation represents, and for example with filter, it can often be used to skip reading unnecessary records.

DataFrames API: Similar-looking to SQL. Example methods include:

- `select`
- `where`
- `limit`
- `orderBy`
- `groupBy`
- `join`

Getting a look at your data

Before we get into transformations and actions on `DataFrames`, let's first look at the ways we can have a look at our data set.

show() pretty-prints DataFrame in tabular form. Shows first 20 elements.

Getting a look at your data

Before we get into transformations and actions on `DataFrames`, let's first look at the ways we can have a look at our data set.

show() pretty-prints DataFrame in tabular form. Shows first 20 elements.

Example:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF
employeeDF.show()
// +---+-----+-------+---+--------+
// | id|fname|  lname|age|    city|
// +---+-----+-------+---+--------+
// | 12|  Joe|  Smith| 38|New York|
// |563|Sally|  Owens| 48|New York|
// |645|Slate|Markham| 28|  Sydney|
// |221|David| Walker| 21|  Sydney|
// +---+-----+-------+---+--------+
```

Getting a look at your data

Before we get into transformations and actions on `DataFrames`, let's first look at the ways we can have a look at our data set.

printSchema() prints the schema of your DataFrame in a tree format.

Getting a look at your data

Before we get into transformations and actions on `DataFrames`, let's first look at the ways we can have a look at our data set.

printSchema() prints the schema of your DataFrame in a tree format.

Example:

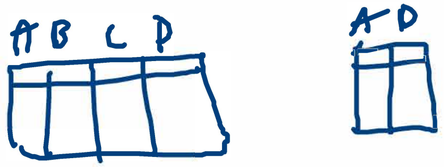
```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF
employeeDF.printschema()

// root
//  |-- id: integer (nullable = true)
//  |-- fname: string (nullable = true)
//  |-- lname: string (nullable = true)
//  |-- age: integer (nullable = true)
//  |-- city: string (nullable = true)
```

Common DataFrame Transformations

Like on RDDs, transformations on `DataFrames` are (1) operations which return a `DataFrame` as a result, and (2) are lazily evaluated.

Common DataFrame Transformations

Like on RDDs, transformations on `DataFrames` are (1) operations which return a `DataFrame` as a result, and (2) are lazily evaluated.



Some common transformations include:

```
def select(col: String, cols: String*): DataFrame
// selects a set of named columns and returns a new DataFrame with these
// columns as a result.

def agg(expr: Column, exprs: Column*): DataFrame
// performs aggregations on a series of columns and returns a new DataFrame
// with the calculated output.

def groupBy(col1: String, cols: String*): DataFrame // simplified
// groups the DataFrame using the specified columns. Intended to be used before an aggregation.

def join(right: DataFrame): DataFrame // simplified
// inner join with another DataFrame
```

Common DataFrame Transformations

Like on RDDs, transformations on `DataFrames` are (1) operations which return a `DataFrame` as a result, and (2) are lazily evaluated.

Some common transformations include:

```
def select(col: String, cols: String*): DataFrame
// selects a set of named columns and returns a new DataFrame with these
// columns as a result.

def agg(expr: Column, exprs: Column*): DataFrame
// performs aggregations on a series of columns and returns a new DataFrame
// with the calculated output.

def groupBy(col1: String, cols: String*): DataFrame // simplified
// groups the DataFrame using the specified columns. Intended to be used before an aggregation.

def join(right: DataFrame): DataFrame // simplified
// inner join with another DataFrame
```

Other transformations include: `filter`, `limit`, `orderBy`, `where`, `as`, `sort`, `union`, `drop`, amongst others.

Specifying Columns

As you might have observed from the previous slide, most methods take a parameter of type `Column` or `String`, always referring to some attribute/column in the data set.

Most methods on `DataFrames` tend to some well-understood, pre-defined operation on a column of the data set

You can select and work with columns in three ways:

1. **Using $-notation**

```
// $-notation requires: import spark.implicits._
df.filter($"age" > 18)
```

2. **Referring to the Dataframe**

```
df.filter(df("age") > 18))
```

3. **Using SQL query string**

```
df.filter("age > 18")
```

sql(" ... ")

DataFrame Transformations: Example

Example:
Recall the example SQL query that we did in the previous session on a data set of employees. Rather than using SQL syntax, let's convert our example to use the DataFrame API.

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF
```

We'd like to obtain just the IDs and last names of employees working in a specific city, say, Sydney, Australia. Let's sort our result in order of increasing employee ID.

How could we solve this with the DataFrame API?

DataFrame Transformations: Example

Example:
We'd like to obtain just the IDs and last names of employees working in a specific city, say, Sydney, Australia. Let's sort in order of increasing employee ID.

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF

val sydneyEmployeesDF = employeeDF.select("id", "lname")
                                  .where("city == 'Sydney'")
                                  .orderBy("id")
```

DataFrame Transformations: Example

Example:
We'd like to obtain just the IDs and last names of employees working in a specific city, say, Sydney, Australia. Let's sort in order of increasing employee ID.

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF

val sydneyEmployeesDF = employeeDF.select("id", "lname")
                                  .where("city == 'Sydney'")
                                  .orderBy("id")
// employeeDF:
// +---+-----+-------+---+--------+
// | id|fname|  lname|age|    city|
// +---+-----+-------+---+--------+
// | 12|  Joe|  Smith| 38|New York|
// |563|Sally|  Owens| 48|New York|
// |645|Slate|Markham| 28|  Sydney|
// |221|David| Walker| 21|  Sydney|
// +---+-----+-------+---+--------+
```

DataFrame Transformations: Example

Example:
We'd like to obtain just the IDs and last names of employees working in a specific city, say, Sydney, Australia. Let's sort in order of increasing employee ID.

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF

val sydneyEmployeesDF = employeeDF.select("id", "lname")
                                  .where("city == 'Sydney'")
                                  .orderBy("id")
// employeeDF:
// +---+-----+-------+---+--------+     sydneyEmployeesDF:
// | id|fname|  lname|age|    city|     +---+-------+
// +---+-----+-------+---+--------+     | id|  lname|
// | 12|  Joe|  Smith| 38|New York|     +---+-------+
// |563|Sally|  Owens| 48|New York|     |221| Walker|
// |645|Slate|Markham| 28|  Sydney|     |645|Markham|
// |221|David| Walker| 21|  Sydney|     +---+-------+
// +---+-----+-------+---+--------+
```

Filtering in Spark SQL

The DataFrame API makes two methods available for filtering: **`filter`** and **`where`** (from SQL). *They are equivalent!*

```
val over30 = employeeDF.filter("age > 30").show()
// +---+-----+-----+---+--------+
// | id|fname|lname|age|    city|
// +---+-----+-----+---+--------+
// | 12|  Joe|Smith| 38|New York|
// |563|Sally|Owens| 48|New York|
// +---+-----+-----+---+--------+
```

```
val over30 = employeeDF.where("age > 30").show()
// +---+-----+-----+---+--------+
// | id|fname|lname|age|    city|
// +---+-----+-----+---+--------+
// | 12|  Joe|Smith| 38|New York|
// |563|Sally|Owens| 48|New York|
// +---+-----+-----+---+--------+
```

Filtering in Spark SQL

The DataFrame API makes two methods available for filtering: **`filter`** and **`where`** (from SQL). *They are equivalent!*

```
val over30 = employeeDF.filter("age > 30").show()
// +---+-----+-----+---+--------+
// | id|fname|lname|age|    city|
// +---+-----+-----+---+--------+
// | 12|  Joe|Smith| 38|New York|
// |563|Sally|Owens| 48|New York|
// +---+-----+-----+---+--------+
```

```
val over30 = employeeDF.where("age > 30").show()
// +---+-----+-----+---+--------+
// | id|fname|lname|age|    city|
// +---+-----+-----+---+--------+
// | 12|  Joe|Smith| 38|New York|
// |563|Sally|Owens| 48|New York|
// +---+-----+-----+---+--------+
```

Filters can be more complex too:

We can compare results between attributes/columns. Though can be more difficult to optimize.

```
employeeDF.filter(($"age" > 25) && ($"city" === "Sydney")).show()
// +---+-----+-------+---+------+
// | id|fname|  lname|age|  city|
// +---+-----+-------+---+------+
// |645|Slate|Markham| 28|Sydney|
// +---+-----+-------+---+------+
```

Grouping and Aggregating on DataFrames

One of the most common tasks on tables is to (1) group data by a certain attribute, and then (2) do some kind of aggregation on it like a count.

Grouping and Aggregating on DataFrames

One of the most common tasks on tables is to (1) group data by a certain attribute, and then (2) do some kind of aggregation on it like a count.

For grouping & aggregating, Spark SQL provides:

- a `groupBy` function which returns a `RelationalGroupedDataset`
- which has several standard aggregation functions defined on it like `count`, `sum`, `max`, `min`, and `avg`.

Grouping and Aggregating on DataFrames

One of the most common tasks on tables is to (1) group data by a certain attribute, and then (2) do some kind of aggregation on it like a count.

For grouping & aggregating, Spark SQL provides:

- a `groupBy` function which returns a `RelationalGroupedDataset`
- which has several standard aggregation functions defined on it like `count`, `sum`, `max`, `min`, and `avg`.

How to group and aggregate?

- Just call `groupBy` on specific attribute/column(s) of a `DataFrame`,
- followed by a call to a method on `RelationalGroupedDataset` like `count`, `max`, or `agg` (for `agg`, also specify which attribute/column(s) subsequent `spark.sql.functions` like `count`, `sum`, `max`, etc, should be called upon.)

```
df.groupBy($"attribute1")
  .agg(sum($"attribute2"))
```

```
df.groupBy($"attribute1")
  .count($"attribute2")
```

Grouping and Aggregating on DataFrames: Example

Example:
Let's assume that we have a dataset of homes currently for sale in an entire US state. Let's calculate the most expensive, and least expensive homes for sale per zip code.

```
case class Listing(street: String, zip: Int, price: Int)

val listingsDF = ... // DataFrame of Listings
```

How could we do this with DataFrames?

Grouping and Aggregating on DataFrames: Example

Example:

Let's assume that we have a dataset of homes currently for sale in an entire US state. Let's calculate the most expensive, and least expensive homes for sale per zip code.

```
case class Listing(street: String, zip: Int, price: Int)

val listingsDF = ... // DataFrame of Listings

import org.apache.spark.sql.functions._

val mostExpensiveDF = listingsDF.groupBy($"zip")
                                .max("price")

val leastExpensiveDF = listingsDF.groupBy($"zip")
                                 .min("price")
```

Grouping and Aggregating on DataFrames: Harder Example

Example:
Let's assume we have the following data set representing all of the posts in a busy open source community's Discourse forum.

```
case class Post(authorID: Int, subforum: String, likes: Int, date: String)

val postsDF = ... // DataFrame of Posts
```

Let's say we would like to tally up each authors' posts per subforum, and then rank the authors with the most posts per subforum.

How could we do this with DataFrames?

Grouping and Aggregating on DataFrames: Harder Example

Example:
Let's assume we have the following data set representing all of the posts in a busy open source community's Discourse forum.

```
case class Post(authorID: Int, subforum: String, likes: Int, date: String)

val postsDF = ... // DataFrame of Posts
```

Let's say we would like to tally up each authors' posts per subforum, and then rank the authors with the most posts per subforum.

```
import org.apache.spark.sql.functions._

val rankedDF =
  postsDF.groupBy($"authorID",$"subforum")
         .agg(count($"authorID")) // new DF with columns authorID, subforum, count(authorID)
         .orderBy($"subforum",$"count(authorID)".desc)
```

Grouping and Aggregating on DataFrames: Harder Example

Example: Let's say we would like to tally up each authors' posts per subforum, and then rank the authors with the most posts per subforums.

```
val rankedDF = postsDF.groupBy($"authorID",$"subforum")
                      .agg(count($"authorID"))
                      .orderBy($"subforum",$"count(authorID)".desc)
```

```
// postsDF:
// +--------+--------+-----+----+
// |authorID|subforum|likes|date|
// +--------+--------+-----+----+
// |       1|  design|    2|    |
// |       1|  debate|    0|    |
// |       2|  debate|    0|    |
// |       3|  debate|   23|    |
// |       1|  design|    1|    |
// |       1|  design|    0|    |
// |       2|  design|    0|    |
// |       2|  debate|    0|    |
// +--------+--------+-----+----+
```

```
rankedDF:
+--------+--------+---------------+
|authorID|subforum|count(authorID)|
+--------+--------+---------------+
|       2|  debate|              2|
|       1|  debate|              1|
|       3|  debate|              1|
|       1|  design|              3|
|       2|  design|              1|
+--------+--------+---------------+
```

Grouping and Aggregating on DataFrames

After calling `groupBy`, methods on `RelationalGroupedDataset`:

To see a list of all operations you
can call following a `groupBy`, see the API docs for `RelationalGroupedDataset`.
http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.RelationalGroupedDataset

Methods within `agg`:

Examples include: `min`, `max`, `sum`, `mean`, `stddev`, `count`, `avg`, `first`, `last`. To see a list of all operations you can call within an `agg`, see the API docs for `org.apache.spark.sql.functions`.
http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$