Spark2.x

<https://github.com/phatak-dev/spark2.0-examples>

Overview of Spark DataFrame API

**Introduction:**

Spark DataFrames were introduced in early 2015, in Spark 1.3. Since then, a lot of new functionality has been added in Spark 1.4, 1.5, and 1.6. More than a year later, Spark's DataFrame API provides a rich set of operations for data munging, SQL queries, and analytics. This post will give an overview of all the major features of Spark's DataFrame API, focusing on the Scala API in 1.6.1.

**Outline**

* Classes and Objects
* Reading Data
* Traditional Dataframe Operations
* Lazy Eval and collect()
* SQL (Relational) Operations
* Data Munging
* Analytics

**Classes and Objects:**

Let us start by reviewing the major classes and objects in the DataFrame API. The main ones are SQLContext, DataFrame, Column, and functions.

**SQLContext, DataFrame, and Column Classes**

* [SQLContext](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.SQLContext) is the main entry point for creating DataFrames.
* [DataFrame](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrame) is the main class representing the DataFrame data and operations.
* The [Column](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column) class represents an individual column of a DataFrame.

**Functions Object:**

The [functions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$) object contains functions for aggregation, math, and date/time and string manipulation that can be applied on DataFrame columns.

**Reading Data into a DataFrame:**

**JSON, Parquet, JDBC, Hive, CSV**

DataFrames can read from a large number of source data formats, such as JSON, Parquet, JDBC, and Hive. See the [DataFrameReader](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameReader) class for some of the natively supported formats and [Spark Packages](http://spark-packages.org/?q=tags%3A%22Data%20Sources%22) for packages available for other formats, such as [CSV](http://spark-packages.org/package/databricks/spark-csv) and many others.

**Reading JSON Example :**

Here is an example of reading JSON data into a DataFrame. The input file must contain one JSON object on each line:

> val df = sqlContext.**read**.json("/home/data.json")

df: org.apache.spark.sql.DataFrame = [col1: int, col2: int]

In the above example, sqlContext is of type [SQLContext](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.SQLContext), its read() method returns a [DataFrameReader](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameReader), and the reader's json() method reads the specified data file.

**Traditional Dataframe Operations:**

Spark DataFrames support traditional dataframe operations that you might expect from working with Pandas or R dataframes. You can select columns and rows, create new columns, and apply functions on column values.

**Selecting Columns:**

To select one or more columns:

> df.**select**("col1")

|col1| +----+ | 1| | 2|

**Selecting Rows:**

To select rows based on a boolean filter:

> df.**filter**(df("col1") > 1)

|col1|col2| +----+----+ | 2| 6|

In the above example, df("col1") is of type [Column](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column), and ">" is a method defined in the [Column](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column) class. Alternatively, a column can be specified with the $"col1" syntax.

More methods in class [Column](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column): **===**, **!==**, **isNaN**, **isNull**, **isin**, **like**, **startsWith**, **endsWith**

See also in class [DataFrame](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrame): **sample**

**Creating New Columns:**

To create a new column derived from existing ones, use the withColumn() method:

> df.**withColumn**("col3", df("col1") + df("col2"))

|col1|col2|col3| +----+----+----+ | 1| 5| 6| | 2| 6| 8|

More methods in class [Column](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column): **%**, **\***, **-**, **/**, **bitwiseAND**, **bitwiseOR**, **cast**, **&&**, **||**

**Math Functions on Columns:**

A number of math functions, defined in the [functions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$) object, such as sqrt(), can be applied to column values:

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

> df.select(df("col1"), **sqrt**(df("col1")))

|col1| SQRT(col1)|

+----+------------------+

| 1| 1.0| | 2|1.4142135623730951|

You can also define your only functions on columns, via UDFs. See the **"UDF"** section below. Here are some more predefined math functions (not comprehensive) in [functions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$):

**cos**, **sin**, **tan**, **exp**, **log**, **pow**, **cbrt**, **hypot**, **toDegrees**, **toRadians**, **ceil**, **floor**, **round**, **rint**, **pmod**,**shiftLeft**, **shiftRight**

**Displaying Data and Schema:**

To display a DataFrame:

> df.**show**()

|col1|col2|

+----+----+

| 1| 5| | 2| 6|

To display a DataFrame's column names and types:

> df.**printSchema**()

root |-- col1: integer (nullable = false)

|-- col2: integer (nullable = false)

See also: **head**, **take, count**

**Lazy Eval and collect()**

DataFrames are evaluated lazily, which means that no computation takes place until you perform an *action*. Any non-action method will thus return immediately, in most cases. An *action* is any method that produces output that is not a DataFrame, such as displaying data on the console, converting data into Scala Arrays, or saving data into a file or database.

To convert a DataFrame into an array, use the collect() method:

> df.**collect**()

res0: Array[org.apache.spark.sql.Row] = Array([1,5], [2,6])

To convert only the first n rows, use **head**or **take**.

**SQL (Relational) Operations**

DataFrames also support SQL (relational) operations, such as SELECT, WHERE, GROUP BY, Aggregate, and JOIN. You can also define UDFs (user-defined functions).

**SELECT, WHERE**

To do a SELECT with a WHERE clause:

> df.**select**("col1", "col2")

    .**where(**$"col1" === 1)

|col1|col2| +----+----+ | 1| 5|

Note that "===" is a Column method that tests for equality.

More methods in class [Column](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column):   
**>, <**, **!==**, **isNaN**, **isNull**, **isin**, **like**, **startsWith**, **endsWith**

**GROUP BY, Aggregate**

To do a GROUP BY and aggregation:

> df1.show()

|col1|col2| +----+----+ | 1| 5| | 2| 6| | 2| 7|

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

> df1.**groupBy**("col1")

     .**agg**(sum("col2").as("sum\_col2"))

|col1|sum\_col2| +----+--------+ | 1| 5| | 2| 13|

Note that the groupBy() method returns a [GroupedData](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.GroupedData)object, on which we call the agg() method to perform one or more aggregations. The sum() function is one of the aggregation functions defined in the [functions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$) object.

More aggregation functions in [functions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$): **approxCountDistinct**, **avg**, **corr**, **count**, **countDistinct**,**first**, **last**, **max**, **mean**, **min**, **skewness**, **stddev**, **sumDistinct**, **variance** (and more)

**JOIN**

To do a JOIN between two DataFrames:

> people.show()

| id| name|

+---+-----+

| 1|Alice| | 2| Bob|

> people.**join**(df, people("id") === df("col1"))

| id| name|col1|col2|

+---+-----+----+----+

| 1|Alice| 1| 5| | 2| Bob| 2| 6|

The above example shows an inner join; other join types, such as outer join, are also supported.

**More SQL operations**

See also in class [DataFrame](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrame):

**alias**, **as**, **cube**, **distinct**, **drop**, **dropDuplicates**, **intersect**, **limit**, **na**, **orderBy**, **repartition**, **rollup**,**selectExpr**, **sort**, **unionAll**, **withColumnRenamed**

**UDFs:**

To define a UDF, use the udf function:

> import org.apache.spark.sql.functions.udf

val myUdf = **udf** {(n: Int) => (n \* 2) + 1}

You can then apply the UDF on one or more Columns:

> df.select(df("col1"), myUdf(df("col1")))

|col1|UDF(col1)|

+----+---------+ | 1| 3| | 2| 5|

**Functions for Data Munging:**

There are a variety of functions to simplify data munging on date, timestamp, string, and nested data in DataFrames. These functions are defined in the [functions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$) object.

**Dates and Timestamps**

Here is an example of working with dates and timestamps. The date\_add() function adds or subtracts days from a given date. The unix\_timestamp() function returns a Unix timestamp corresponding to a timestamp string in a specified format:

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

> df6.withColumn("day\_before", **date\_add**(df6("date"), -1))

     .withColumn("unix\_time", **unix\_timestamp**(df6("date"), "yyyy-MM-dd"))

| date|day\_before| unix\_time|

+----------+----------+----------+

|2016-01-01|2015-12-31|1451606400| |2016-09-05|2016-09-04|1473033600|

See also (not comprehensive) in [functions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$): **current\_date**, **current\_timestamp**, **date\_sub**, **datediff**,**dayofmonth**, **dayofyear**, **from\_unixtime**, **hour**, **last\_day**, **minute**, **month**, **next\_day**, **quarter**,**second**, **trunc**, **weekofyear**, **year**

**Strings**

There are also a number of functions for working with strings. Here is one example, with the substring() function, which returns a substring given an input string, position, and length:

> df6.withColumn("month\_day", **substring**(df6("date"), 6, 5))

| date|month\_day|

+----------+---------+ |2016-01-01| 01-01| |2016-09-05| 09-05|

See also (not comprehensive) in [functions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$): **ascii**, **concat**, **decode**, **encode**, **format\_number**,**format\_string**, **length**, **levenshtein**, **lower**, **lpad**, **ltrim**, **regexp\_extract**, **regexp\_replace**, **repeat**,**rtrim**, **split**, **translate**, **trim**, **upper**

**Nested Data Structures**

With certain data formats, such as JSON, it is common to have nested arrays and structs in the schema. The [functions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$) object includes functions for working with nested columns. For example, if a column is of type Array, such as "col2" below, you can use the explode() function to flatten the data inside that column:

> df8

|col1| col2| +----+--------+ | 1|[1a, 1b]| | 2| [2a]|

> df8.select(df8("col1"), **explode**(df8("col2")).as("col2\_flat"))

|col1|col2\_flat| +----+---------+ | 1| 1a| | 1| 1b| | 2| 2a|

The new flattened column, "col2\_flat", can now be manipulated as an ordinary top-level column. For more about nested array data, please see [my post](http://xinhstechblog.blogspot.com/2016/05/reading-json-nested-array-in-spark.html) on the topic.

See also:

**array\_contains**, **size**, **sort\_array**, **struct**, **array** in [functions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$).

**Analytics**

The DataFrame API includes functionality for analytics, namely, summary statistics, window functions, and pivot tables.

**Summary Statistics**

The describe() method computes summary statistics for numerical columns and is meant for exploratory data analysis:

> df.**describe**()

|summary| col1| col2| +-------+------------------+------------------+ | count| 2| 2| | mean| 1.5| 5.5| | stddev|0.7071067811865476|0.7071067811865476| | min| 1| 5| | max| 2| 6|

The **stat**() method returns a [DataFrameStatFunctions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameStatFunctions) object, which provides additional statistics functions such as:

**corr**, **cov**, **crosstab**, **freqItems**, **sampleBy**

**Window Functions**

Window functions allow you to perform calculations over a moving window of rows, and are the basis for calculating a moving average or cumulative sum. You can apply window functions on DataFrames by defining a [WindowSpec](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.expressions.WindowSpec):

> import org.apache.spark.sql.expressions.Window

> val wSpec2 = **Window**.partitionBy("name").orderBy("date").rowsBetween(-1, 1)

The above window spec for a moving average consists of three components: (1) partition by, (2) order by, and (3) a frame. To use this WindowSpec in a DataFrame, you would apply a window function or aggregation function, such as avg() over this WindowSpec:

> customers.withColumn("movingAvg", **avg**(customers("amountSpent")).**over**(wSpec2))

| name| date|amountSpent|movingAvg| +-----+----------+-----------+---------+ |Alice|2016-05-01| 50.0| 47.5| |Alice|2016-05-03| 45.0| 50.0| |Alice|2016-05-04| 55.0| 50.0| | Bob|2016-05-01| 25.0| 27.0| | Bob|2016-05-04| 29.0| 27.0| | Bob|2016-05-06| 27.0| 28.0|

For a list of all the possible window functions and aggregation functions, please see the [functions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$) object. For more examples of using window functions, please see my [blog post](http://xinhstechblog.blogspot.com/2016/04/spark-window-functions-for-dataframes.html) on the topic.

For more window functions, see in [functions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$): **cume\_dist**, **lag**, **lead**, **ntile**, **percent\_rank**, **rank**,**row\_number**(and more)

**Pivot Tables**

You can create pivot tables with the pivot() method:

> df7.groupBy("col1").**pivot**("col2").avg("col3")

|col1| A| B| +----+----+----+ | 1|10.0|21.0| | 2|12.0|20.0|

In the above example, the DataFrame is grouped by col1, pivoted along col2, which contains the values "A" and "B", and computes the average of col3 in each group. The pivot() method is defined in the [GroupedData](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.GroupedData) class. For more information about pivoting, please see this Databricks article: [Reshaping Data with Pivot in Apache Spark](https://databricks.com/blog/2016/02/09/reshaping-data-with-pivot-in-apache-spark.html).

**Summary**

By now, you should have a good feel for what is possible with the Spark DataFrame Scala API. From data munging, to SQL, to analytics, this API provides a broad range of functionality for working with big data. For more information about DataFrames, see the [Spark programming guide](http://spark.apache.org/docs/latest/sql-programming-guide.html).

Reading JSON Nested Array in Spark DataFrames:

In a previous post on JSON data, I showed how to read nested JSON arrays with Spark DataFrames. Now that I am more familiar with the API, I can describe an easier way to access such data, using the *explode()* function. All of the example code is in Scala, on Spark 1.6.

**Loading JSON data**

Suppose you have a file with JSON data, with one JSON object per line:

{"name":"Michael", "schools":[{"sname":"stanford", "year":2010}, {"sname":"berkeley", "year":2012}]}

{"name":"Andy", "schools":[{"sname":"ucsb", "year":2011}]}

You can read it into a DataFrame with the SqlContext *read()* method:

>> val people = sqlContext.read.json("people.json")

people: org.apache.spark.sql.DataFrame

>> people.show()

+-------+--------------------+ | name| schools| +-------+--------------------+ |Michael|[[stanford,2010],...| | Andy| [[ucsb,2011]]| +-------+--------------------+

Notice that the second column "schools", is an Array type, and each element of the array is a Struct:  
  
>> people.printSchema()  
root |-- name: string (nullable = true) |-- schools: array (nullable = true) | |-- element: struct (containsNull = true) | | |-- sname: string (nullable = true) | | |-- year: long (nullable = true)

**Nested Array of Struct**

**Flatten / Explode an Array**

If your JSON object contains nested arrays of structs, how will you access the elements of an array? One way is by flattening it. For instance, in the example above, each JSON object contains a "schools" array. We can simply flatten "schools" with the *explode()*function.  
  
>> import org.apache.spark.sql.functions.\_  
val flattened = people.select($"name", explode($"schools").as("schools\_flat"))  
flattened: org.apache.spark.sql.DataFrame  
  
>> flattened.show()  
+-------+---------------+ | name| schools\_flat| +-------+---------------+ |Michael|[stanford,2010]| |Michael|[berkeley,2012]| | Andy| [ucsb,2011]| +-------+---------------+  
  
Now each school is on a separate row. The new column "schools\_flat" is of type Struct.

**Select into Struct**

Now you can select, for instance, all the school names within each struct, by using the DataFrame *select()* method. The struct has two fields: "sname" and "year". We will select only the school name, "sname":

>> val schools = flattened.select("name", "schools\_flat.sname")  
schools: org.apache.spark.sql.DataFrame = [sname: string]  
  
>> schools.show()  
+-------+--------+  
| name| sname| +-------+--------+ |Michael|stanford| |Michael|berkeley| | Andy| ucsb| +-------+--------+

There you have it! We have taken data that was nested as structs inside an array column and bubbled it up to a first-level column in a DataFrame. You can now manipulate that column with the standard DataFrame methods.

**References**

1. The DataFrame API: <http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrame>
2. The *explode()*function: <http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$>

Spark Window Functions for DataFrames and SQL

Introduced in Spark 1.4, Spark window functions improved the expressiveness of Spark DataFrames and Spark SQL. With window functions, you can easily calculate a moving average or cumulative sum, or reference a value in a previous row of a table. Window functions allow you to do many common calculations with DataFrames, without having to resort to RDD manipulation.

**Aggregates, UDFs vs. Window functions**

Window functions are complementary to existing DataFrame operations: aggregates, such as *sum*and *avg*, and UDFs. To review, aggregates calculate one result, a sum or average, for each group of rows, whereas UDFs calculate one result for each row based on only data in that row. In contrast, window functions calculate one result for each row based on a window of rows. For example, in a moving average, you calculate for each row the average of the rows surrounding the current row; this can be done with window functions.

**Moving Average Example**

Let us dive right into the moving average example. In this example dataset, there are two customers who have spent different amounts of money each day.

// Building the customer DataFrame. All examples are written in Scala with Spark 1.6.1, but the same can be done in Python or SQL.

val customers = sc.parallelize(List(("Alice", "2016-05-01", 50.00),

                                    ("Alice", "2016-05-03", 45.00),

                                    ("Alice", "2016-05-04", 55.00),

                                    ("Bob", "2016-05-01", 25.00),

                                    ("Bob", "2016-05-04", 29.00),

                                    ("Bob", "2016-05-06", 27.00))).

                               toDF("name", "date", "amountSpent")

// Import the window functions.

import org.apache.spark.sql.expressions.Window

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

// Create a window spec.

val wSpec1 = Window.partitionBy("name").orderBy("date").rowsBetween(-1, 1)

In this window spec, the data is partitioned by customer. Each customer’s data is ordered by date. And, the window frame is defined as starting from -1 (one row before the current row) and ending at 1 (one row after the current row), for a total of 3 rows in the sliding window.

// Calculate the moving average

customers.withColumn( "movingAvg",

                                             avg(customers("amountSpent")).over(wSpec1)  ).show()

This code adds a new column, “movingAvg”, by applying the *avg* function on the sliding window defined in the window spec:

|  |  |  |  |
| --- | --- | --- | --- |
| name | date | amountSpent | movingAvg |
| Alice | 5/1/2016 | 50 | 47.5 |
| Alice | 5/3/2016 | 45 | 50 |
| Alice | 5/4/2016 | 55 | 50 |
| Bob | 5/1/2016 | 25 | 27 |
| Bob | 5/4/2016 | 29 | 27 |
| Bob | 5/6/2016 | 27 | 28 |

**Window function and Window Spec definition**

As shown in the above example, there are two parts to applying a window function: (1) specifying the window function, such as *avg* in the example, and (2) specifying the window spec, or *wSpec1* in the example. For (1), you can find a full list of the window functions here:  
<https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$>  
 You can use functions listed under “Aggregate Functions” and “Window Functions”.

For (2) specifying a window spec, there are three components: partition by, order by, and frame.

1. “Partition by” defines how the data is grouped; in the above example, it was by customer. You have to specify a reasonable grouping because all data within a group will be collected to the same machine. Ideally, the DataFrame has already been partitioned by the desired grouping.
2. “Order by” defines how rows are ordered within a group; in the above example, it was by date.
3. “Frame” defines the boundaries of the window with respect to the current row; in the above example, the window ranged between the previous row and the next row.

**Cumulative Sum**

Next, let us calculate the cumulative sum of the amount spent per customer.

// Window spec: the frame ranges from the beginning (Long.MinValue) to the current row (0).

val wSpec2 = Window.partitionBy("name").orderBy("date").rowsBetween(Long.MinValue, 0)

// Create a new column which calculates the sum over the defined window frame.

customers.withColumn( "cumSum",

  sum(customers("amountSpent")).over(wSpec2)  ).show()

|  |  |  |  |
| --- | --- | --- | --- |
| name | date | amountSpent | cumSum |
| Alice | 5/1/2016 | 50 | 50 |
| Alice | 5/3/2016 | 45 | 95 |
| Alice | 5/4/2016 | 55 | 150 |
| Bob | 5/1/2016 | 25 | 25 |
| Bob | 5/4/2016 | 29 | 54 |
| Bob | 5/6/2016 | 27 | 81 |

**Data from previous row**

In the next example, we want to see the amount spent by the customer in their previous visit.

// Window spec. No need to specify a frame in this case.

val wSpec3 = Window.partitionBy("name").orderBy("date")

// Use the *lag* function to look backwards by one row.

customers.withColumn("prevAmountSpent",

 lag(customers("amountSpent"), 1).over(wSpec3) ).show()

|  |  |  |  |
| --- | --- | --- | --- |
| name | date | amountSpent | prevAmountSpent |
| Alice | 5/1/2016 | 50 | null |
| Alice | 5/3/2016 | 45 | 50 |
| Alice | 5/4/2016 | 55 | 45 |
| Bob | 5/1/2016 | 25 | null |
| Bob | 5/4/2016 | 29 | 25 |
| Bob | 5/6/2016 | 27 | 29 |

**Rank**

In this example, we want to know the order of a customer’s visit (whether this is their first, second, or third visit).

// The *rank* function returns what we want.

customers.withColumn( "rank", rank().over(wSpec3) ).show()

|  |  |  |  |
| --- | --- | --- | --- |
| name | date | amountSpent | rank |
| Alice | 5/1/2016 | 50 | 1 |
| Alice | 5/3/2016 | 45 | 2 |
| Alice | 5/4/2016 | 55 | 3 |
| Bob | 5/1/2016 | 25 | 1 |
| Bob | 5/4/2016 | 29 | 2 |
| Bob | 5/6/2016 | 27 | 3 |

**Conclusion**

I hope these examples have helped you understand Spark’s window functions. There is more functionality that was not covered here. To learn more, please see the Databricks article on this topic: <https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html>

**Spark 1.6 Datasets API: Example Usage**

**Overview**

Spark 1.6 introduced a new Datasets API. It is an extension of Dataframes that supports functional processing on a collection of objects. Let's take a look at some examples of how to use them. First we'll read a JSON file and a text file into Datasets. We will apply functional transformations to parse the data. Then we will run relational queries against a Dataset.

**Creating a Dataset from a JSON file**

Suppose you have JSON formatted data which you would like to read into a Dataset. Here is an example JSON file:

Contents of "students.json" --

{"name":"Alice", "dept":"Math"}  
{"name":"Bob", "dept":"CS"}  
{"name":"Carl", "dept":"Math"}

To create a Dataset from this JSON file:

// Define the Student row type.  
> case class Student(name: String, dept: String)  
// Read JSON objects into a Dataset[Student].  
> val studentsFromJSON = sqlContext.read.json("students.json").as[Student]

**Creating a Dataset from a Text file**

Suppose instead you have data in a text file, in tab-separated (.tsv) format:

Alice<tab>Math<tab>18  
Bob<tab>CS<tab>19  
Carl<tab>Math<tab>21

To create a Dataset from this text file:

// Read the lines of the file into a Dataset[String].  
> val studentsFromText = sqlContext.read.text("students.tsv").as[String]

(result) studentsFromText: org.apache.spark.sql.Dataset[String] = [value: string]

// We want a Dataset of type "Student".  
case class Student(name: String, dept: String, age:Int)

// Functional programming to parse the lines into a Dataset[Student].

val students = studentsFromText.  
  map(line => {

    val cols = line.split("\t") // parse each line

    Student(cols(0), cols(1), cols(2).toInt)

  })

(result) students: org.apache.spark.sql.Dataset[Student] = [name: string, dept: string, age: int]

// Show the contents of the Dataset.

> students.show()

| name|dept|age|

+-----+----+---+

|Alice|Math| 18|

|  Bob|  CS| 19|

| Carl|Math| 21|

**Relational queries**

Datasets support relational queries, with operations such as: select, filter, group by, count, avg, join.

**SELECT, FILTER**

Get the names of students in the Math department.

// Select two columns and filter on one column.  
// Each argument of "select" must be a "TypedColumn".

> students.select($"name".as[String], $"dept".as[String]).

    filter(\_.\_2 == "Math").  // Filter on \_2, the second selected column

    collect()

(result) Array((Alice,Math), (Carl,Math))

**GROUP BY, COUNT**

Count the number of students in each department.

// Group by department and count each group.

> students.groupBy(\_.dept).count().collect()

(result) Array((CS,1), (Math,2))

**GROUP BY, AVG**

Average age in each department.

// Import the "avg" function.

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

// Group and aggregate in each group.

> students.groupBy(\_.dept).  
    agg(avg($"age").as[Double]).  
    collect()

(result) Array((CS,19.0), (Math,19.5))

**JOIN**

Suppose we have a separate table with deparment information. We would like to join the department information into our student table.

First, create the department Dataset.

// The Department type.

> case class Department(abbrevName: String, fullName: String)

// Initialize a Seq and convert to a Dataset.

> val depts = Seq(Department("CS", "Computer Science"), Department("Math", "Mathematics")).toDS()

// Show the contents of the Dataset.

> depts.show()

|abbrevName|        fullName|

+----------+----------------+

|        CS|Computer Science|

|      Math|     Mathematics|

Join the students Dataset with the departments Dataset.

// Join two datasets with "joinWith".

> val joined = students.joinWith(depts, $"dept" === $"abbrevName")

// Show the contents of the joined Dataset.  
// Note that the original objects are nested into tuples under the \_1 and \_2 columns.

> joined.show()

|             \_1|                  \_2|

+---------------+--------------------+

|[Alice,Math,18]|  [Math,Mathematics]|

|    [Bob,CS,19]|[CS,Computer Scie...|

| [Carl,Math,21]|  [Math,Mathematics]|

Select two columns from the joined Dataset.

// Use "map" to select from the joined Dataset.   
// Notice that the original Dataset types are preserved.

> joined.map(s => (s.\_1.name, s.\_2.fullName)).show()

|   \_1|              \_2|

+-----+----------------+

|Alice|     Mathematics|

|  Bob|Computer Science|

| Carl|     Mathematics|

**EXPLAIN**

"Explain" prints the query's physical plan for debugging.

// Explain how the join is computed.  
// Note that a BroadcastJoin is planned.  
> joined.explain()

== Physical Plan ==

Project [struct(name#168163,dept#168164,age#168165) AS \_1#168203,struct(abbrevName#168200,fullName#168201) AS \_2#168204]

+- **BroadcastHashJoin** [dept#168164], [abbrevName#168200], BuildRight

   :- ConvertToUnsafe

   :  +- !MapPartitions <function1>, class[value[0]: string], class[name[0]: string, dept[0]: string, age[0]: int], [name#168163,dept#168164,age#168165]

   :     +- ConvertToSafe

   :        +- Scan TextRelation[value#168157] InputPaths: /students.tsv

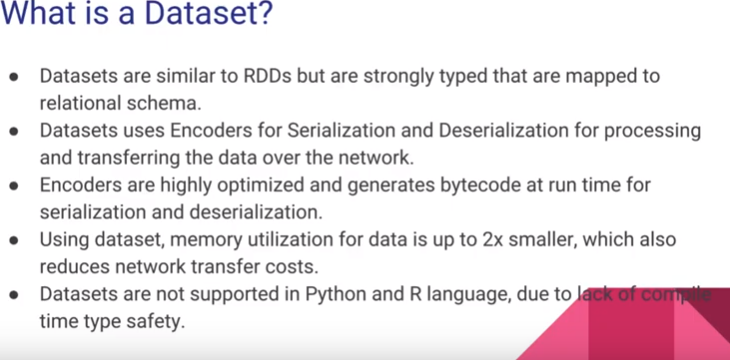
   +- ConvertToUnsafe

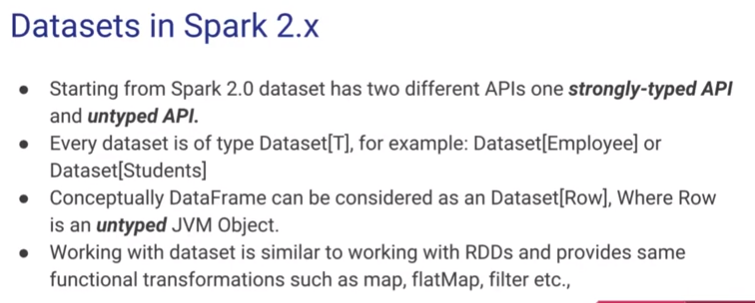
      +- LocalTableScan [abbrevName#168200,fullName#168201], [[0,1800000002,2000000010,5343,72657475706d6f43,65636e6569635320],[0,1800000004,200000000b,6874614d,74616d656874614d,736369]]

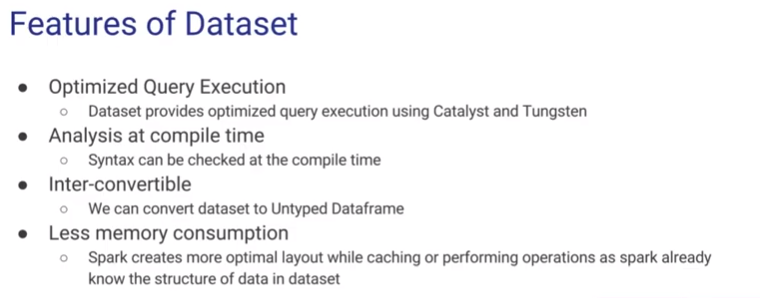
**References**

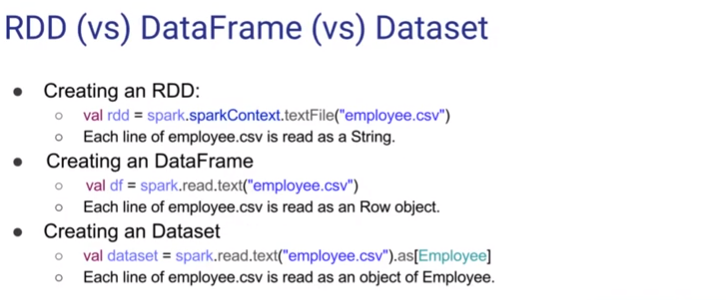
Spark API docs: <http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset>

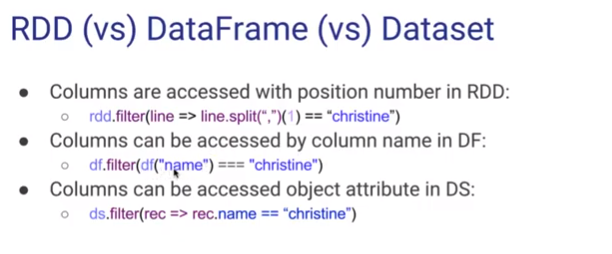
Spark Programming Guide: <http://spark.apache.org/docs/latest/sql-programming-guide.html>  
  
Introducing Spark Datasets: <https://databricks.com/blog/2016/01/04/introducing-spark-datasets.html>

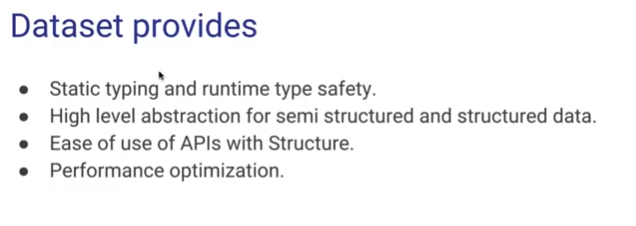


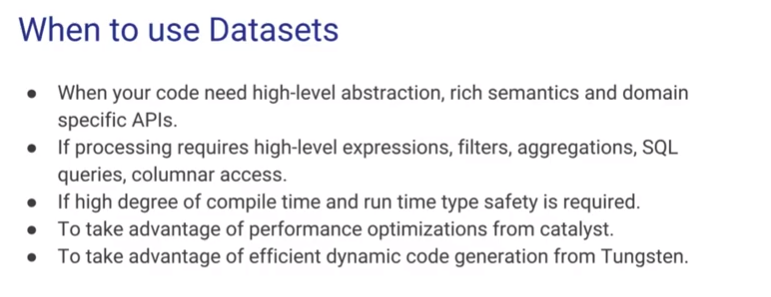




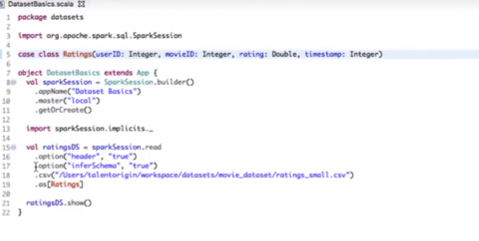




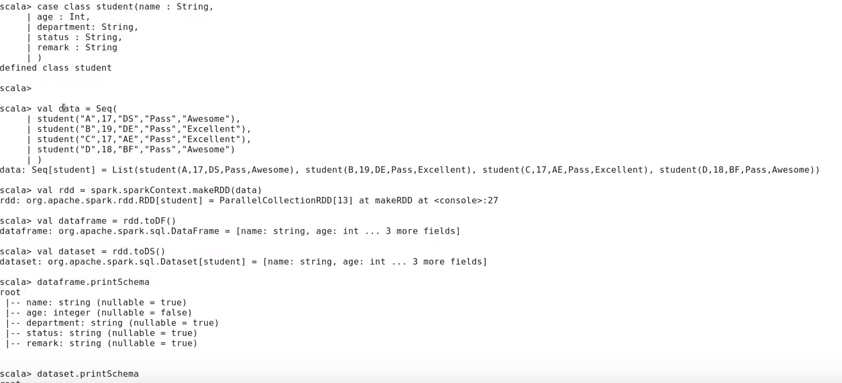




Example:



We have to use inferSchema else we will get error



### Overview of Spark 2.0 Dataset / DataFrame API, Part 1

## Introduction

Spark 2.0 features a new [Dataset API](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset). Now that Datasets support a full range of operations, you can avoid working with low-level RDDs in most cases. In 2.0, DataFrames no longer exist as a separate class; instead, DataFrame is defined as a special case of Dataset. Here is some example code to get you started with Spark 2.0 Datasets / DataFrames. Part 1 focuses on type-safe operations with Datasets, which provide compile time type safety. Part 2 focuses on DataFrames, which have untyped operations.  
  
Part 1: Datasets: Type-safe operations. (This blog post)  
Part 2: DataFrame: Untyped operations. ([Next blog post](http://xinhstechblog.blogspot.com/2016/07/overview-of-spark-20-dataset-dataframe_29.html))

**Dataset vs. DataFrame**

A Dataset[T] is a parameterized type, where the type T is specified by the user and is associated with each element of the Dataset. A DataFrame, on the other hand, has no explicit type associated with it at compile time, from the user's point of view. Internally, a DataFrame is defined as a Dataset[Row], where Row is a generic row type defined by Spark SQL.

**Language**

This blog post refers to the [Scala API](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package).

**Outline**

* Reading Data In
* Data Exploration
* Statistics
* Functional Transformations
* Caching
* Getting Data Out

**Reading Data In**

Spark supports a number of input formats, including Hive, JDBC, Parquet, CSV, and JSON. Below is an example of reading JSON data into a Dataset.

**JSON example**

Suppose you have this example JSON data, with one object per line:

{"name":"Alice", "dept":"Math", "age":21}  
{"name":"Bob", "dept":"CS", "age":23}  
{"name":"Carl", "dept":"Math", "age":25}

To read a JSON data file, first use the SparkSession object as an entry point, and access its [DataFrameReader](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameReader) to read data into a DataFrame:

> val df = spark.read.**json**("/path/to/file.json") // "spark" is a SparkSession object

df1: org.apache.spark.sql.DataFrame

Then convert the DataFrame into Dataset[Student]:

> case class Student(name: String, dept: String, age: Long)

> val ds = df.**as**[Student]

ds: org.apache.spark.sql.Dataset[Student]

**Data Exploration**

When you first look into a new data set, you can explore its contents by printing out the schema, counting the number of rows, and displaying some of those rows.

**Print Schema**

To explore what is in this Dataset, you can print out the schema:

> ds.**printSchema**()

root |-- age: long (nullable = true) |-- dept: string (nullable = true) |-- name: string (nullable = true)

**Count Rows**

To count the number of rows:

> ds.**count**()

res2: Long = 3

**Display Rows**

To display the first few rows in tabular format:

> ds.**show**()

|age|dept| name| +---+----+-----+ | 21|Math|Alice| | 23| CS| Bob| | 25|Math| Carl|

**Sample Rows**

To get a sample of the data:

> val sample = ds.**sample**(withReplacement=false, fraction=0.3)

sample: org.apache.spark.sql.Dataset[Student]

|age|dept|name| +---+----+----+ | 25|Math|Carl|

**Statistics**

A number of statistics functions are available for Datasets.

**Summary Statistics**

To get summary statistics on numerical fields, call "describe":  
  
> val summary = ds.**describe**()  
summary: org.apache.spark.sql.DataFrame  
|summary| age|

+-------+----+ | count| 3| | mean|23.0| | stddev| 2.0| | min| 21| | max| 25|

**Additional Statistical Functions, Approximate Frequent Items**

The "stat" method returns a [DataFrameStatFunctions](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameStatFunctions) object for statistical functions:  
  
> ds.**stat**  
res11: org.apache.spark.sql.DataFrameStatFunctions  
  
For example, "stat.freqItems" returns approximate frequent items for the given columns:  
  
> val approxFreqItems = ds.**stat**.**freqItems**(Seq("dept"))

approxFreqItems: org.apache.spark.sql.DataFrame  
|dept\_freqItems|

+--------------+ | [CS, Math]|

**Functional Transformations**

The Dataset API supports functional transformations, such as "filter" and "map", much like the RDD API. These operators transform one Dataset[T] into another Dataset[U], where T and U are user-specified types. These operations have compile-time type safety, in the sense that each row is associated with a Scala object of a fixed type T (or U). This is in contrast to DataFrames, which are untyped. "Reduce" is an action that reduces the elements of a Dataset into a scalar value.

**Filter**

To filter for rows that satisfy a given predicate:

> val youngStudents = ds.**filter**($"age" < 22)  
youngStudents: org.apache.spark.sql.Dataset[Student]  
|age|dept| name|

+---+----+-----+ | 21|Math|Alice|

**Map**

To map over rows with a given lambda function:

> val names = ds.**map**{\_.name}

names: org.apache.spark.sql.Dataset[String]  
|value|  
+-----+ |Alice| | Bob| | Carl|

**Reduce**

To reduce the elements of a Dataset with a given reducer function:

> val totalAge = ds.map(\_.age).**reduce**(\_ + \_)

totalAge: Long = 69

**Join**

You can join two Datasets. Suppose you want to join the "Students" Dataset with a new "Department" Dataset:

> case class Department(name: String, building: Int)  
> val depts = Seq(Department("Math", 125), Department("CS", 110)).toDS()  
|name|building|  
+----+--------+ |Math| 125| | CS| 110|

To join the Students" Dataset with the new "Department" Dataset:

> val joined = ds.**joinWith**(depts, ds("dept") === depts("name"))  
joined: org.apache.spark.sql.Dataset[(Student, Department)]  
| \_1| \_2|  
+---------------+----------+ |[21,Math,Alice]|[Math,125]| | [23,CS,Bob]| [CS,110]| | [25,Math,Carl]|[Math,125]|

**GroupByKey, Aggregation**

To group elements of a Dataset and aggregate within each group:  
  
> val deptSizes = ds.**groupByKey**(\_.dept).count()  
deptSizes: org.apache.spark.sql.Dataset[(String, Long)]  
|value|count(1)|  
+-----+--------+ | Math| 2| | CS| 1|  
  
Additional aggregation functions are available in the ["functions" object](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$). The "avg" function calculates an average for each group:  
  
> import org.apache.spark.sql.**functions**.\_  
> val avgAge = ds.**groupByKey**(\_.dept)  
                             .**agg**(avg($"age").as[Double])  
avgAge: org.apache.spark.sql.Dataset[(String, Double)]  
|value|avg(age)|  
+-----+--------+ | Math| 23.0| | CS| 23.0|

**OrderBy**

To order by a given set of fields:

> val ordered = ds.**orderBy**("dept", "name")  
ordered: org.apache.spark.sql.Dataset[Student]  
|age|dept| name|  
+---+----+-----+ | 23| CS| Bob| | 21|Math|Alice| | 25|Math| Carl|

**Caching**

To persist a Dataset at the default storage level (memory and disk):

> ds.**cache**()

**Getting Data Out**

**Into an Array**

To collect data into a Scala Array, use "collect". Note that this will collect all rows into the Driver node, and thus could potentially be a memory- and IO- intensive operation.

> val studentArr = ds.**collect**()  
studentArr: Array[Student] = Array(Student(Alice,Math,21), Student(Bob,CS,23), Student(Carl,Math,25))  
  
To collect only the first few rows into a Scala Array:  
> val firstTwo = ds.**head**(2)  
firstTwo: Array[Student] = Array(Student(Alice,Math,21), Student(Bob,CS,23))

**Into an RDD**

To convert into an RDD:  
  
> val studentRdd = ds.**rdd**  
studentRdd: org.apache.spark.rdd.RDD[Student]

**Into a File**

To write a Dataset into a file, use "write". A number of output formats are supported. Here is an example of writing in JSON format:

> ds.**write**.**json**("/path/to/file.json")

**Overview of Spark 2.0 Dataset / DataFrame API, Part 2**

**DataFrame**

DataFrames are still available in Spark 2.0, and remain mostly unchanged. The biggest change is that they have been merged with the new [Dataset API](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset). The DataFrame class no longer exists on its own; instead, it is defined as a specific type of Dataset: type DataFrame = Dataset[Row]. However, all of the functionality from 1.6 is still there.

**Outline**

* Example Data
* DataFrames: Untyped Language-Integrated SQL Queries
* DataFrames: SQL Queries in SQL
* DataFrames: Adding Columns, Data Munging

**Example Data**

We will continue with the example data from Part 1. We have defined a "Student" class as:

> case class Student(name: String, dept: String, age: Long)

The example data has been read into a Dataset[Student]:

> ds

ds: org.apache.spark.sql.Dataset[Student]  
|age|dept| name|

+---+----+-----+ | 21|Math|Alice| | 23| CS| Bob| | 25|Math| Carl|

**DataFrames: Untyped Language-Integrated SQL Queries**

DataFrames supports language-integrated SQL queries, such as "select", "where", and "group by".

**Convert to DataFrame**

To convert a Dataset into a DataFrame:  
  
> val df = ds.**toDF**()  
df: org.apache.spark.sql.DataFrame

**Select, Where**

To select columns and specify a "where" clause:

> val selected = df.select("name", "age")  
                             .where($"age" === 21)  
selected: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]  
| name|age| +-----+---+ |Alice| 21|

**Count**

To count the number of rows:

> df.**count**()  
res1: Long = 3

**GroupBy, Aggregate**

To perform a "group by" and aggregate within each group, use "groupBy" and "agg". A number of aggregation functions, such as "avg", are available in the ["functions" object](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$). To group by a column and compute the average in each group:

> import org.apache.spark.sql.functions.\_  
> val avgAge2 = df.**groupBy**("dept")  
                               .**agg**(avg($"age"))  
|dept|avg(age)| +----+--------+ |Math| 23.0| | CS| 23.0|

**Join**

You can join two DataFrames with "join". To create a second DataFrame with department info:

> case class Department(deptName: String, building: Int)  
> val depts = Seq(Department("Math", 125), Department("CS", 110)).toDF()  
|deptName|building| +--------+--------+ | Math| 125| | CS| 110|  
  
Then, to join the students DataFrame with the new department DataFrame:  
  
> val joined2 = df.**join**(depts, df("dept") === depts("deptName"))  
|age|dept| name|deptName|building| +---+----+-----+--------+--------+ | 21|Math|Alice| Math| 125| | 23| CS| Bob| CS| 110| | 25|Math| Carl| Math| 125|

**Explain**

To examine the query plan used to compute a DataFrame:

> joined2.**explain**()

== Physical Plan == \*BroadcastHashJoin [dept#134], [deptName#384], Inner, BuildRight :- \*Filter isnotnull(dept#134) : +- Scan ExistingRDD[age#133L,dept#134,name#135] +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, false])) +- \*Filter isnotnull(deptName#384) +- LocalTableScan [deptName#384, building#385]

**DataFrames: SQL Queries in SQL**

You can also query DataFrames with SQL. First create a temp view and then specify SQL queries against that view:

> df.**createTempView**("StudentTable")  
  
> val sqlResults = spark.**sql**("SELECT name, dept FROM StudentTable") // "spark" is a SparkSession object  
| name|dept| +-----+----+ |Alice|Math| | Bob| CS| | Carl|Math|

**DataFrames: Adding Columns, Data Munging**

DataFrames support creating new columns and data munging. To add a column, use "withColumn" to specify a new column name and an expression for column values. The [Column class](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column) defines column operations, such as the minus operator shown below. The ["functions" object](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$) also contains convenient functions for working with columns, such as math, string, and date / time functions.

In this example, the "lit" function, defined in "functions", returns a Column populated with a literal value. The Column class minus operator performs subtraction. The "$" method returns the Column associated with the given column name:

> import org.apache.spark.sql.**functions**.\_

> val withCol = df.**withColumn**("birthYear", lit(2016) - $"age")  
|age|dept| name|birthYear| +---+----+-----+---------+ | 21|Math|Alice| 1995| | 23| CS| Bob| 1993| | 25|Math| Carl| 1991|  
  
In the next example, the "round" function, defined in "functions", rounds values to the nearest tens digit:  
  
> val rounded = df.**withColumn**("roundedAge", round($"age", -1))  
|age|dept| name|roundedAge| +---+----+-----+----------+ | 21|Math|Alice| 20| | 23| CS| Bob| 20| | 25|Math| Carl| 30|

**Summary**

In Spark 2.0, DataFrames have been merged into the DataSet API. DataFrame is a special type of Dataset that has untyped operations. DataFrames support convenient ways to query data, either through language-integrated queries or SQL. DataFrames are also useful in creating new columns and data munging.

Introduction to Spark 2.0 - Part 1 : Spark Session API

### **Dataset - New Abstraction of Spark**

For long, RDD was the standard abstraction of Spark. But from Spark 2.0, Dataset will become the new abstraction layer for spark. Though RDD API will be available, it will become low level API, used mostly for runtime and library development. All user land code will be written against the Dataset abstraction and it’s subset Dataframe API.

Dataset is a superset of Dataframe API which is released in Spark 1.3. Dataset together with Dataframe API brings better performance and flexibility to the platform compared to RDD API. Dataset will be also replacing RDD as an abstraction for streaming in future releases.

### **SparkSession - New entry point of Spark**

In earlier versions of spark, spark context was entry point for Spark. As RDD was main API, it was created and manipulated using context API’s. For every other API,we needed to use different contexts.For streaming, we needed StreamingContext, for SQL sqlContext and for hive HiveContext. But as DataSet and Dataframe API’s are becoming new standard API’s we need an entry point build for them. So in Spark 2.0, we have a new entry point for DataSet and Dataframe API’s called as Spark Session.

SparkSession is essentially combination of SQLContext, HiveContext and future StreamingContext. All the API’s available on those contexts are available on spark session also. Spark session internally has a spark context for actual computation.

So in rest of our post, we will discuss how to create and interact with Spark session.

### **Creating SparkSession**

SparkSession follows builder factory design pattern. The below is the code to create a spark session.

**val** sparkSession **=** **SparkSession.**builder**.**

master**(**"local"**)**

**.**appName**(**"spark session example"**)**

**.**getOrCreate**()**

The above is similar to creating an SparkContext with local and creating an SQLContext wrapping it. If you need to create, hive context you can use below code to create spark session with hive support.

**val** sparkSession **=** **SparkSession.**builder**.**

master**(**"local"**)**

**.**appName**(**"spark session example"**)**

**.**enableHiveSupport**()**

**.**getOrCreate**()**

**enableHiveSupport** on factory enables hive support which is similiar to HiveContext.

Once we have created spark session, we can use it to read the data.

## **Read data using Spark Session**

The below code is reading data from csv using spark session.

**val** df **=** sparkSession**.**read**.**option**(**"header"**,**"true"**).**

csv**(**"src/main/resources/sales.csv"**)**

It looks like exactly like reading using SQLContext. You can easily replace all your code of SQLContext with SparkSession now.

You can access complete code [here](https://github.com/phatak-dev/spark2.0-examples/blob/master/src/main/scala/com/madhukaraphatak/examples/sparktwo/SparkSessionExample.scala).

### **Is SQLContext and HiveContext going away?**

Not really. Spark is big on backward compatibility. So these API’s are still available. Though from documentation it’s clear that they want developers to use SparkSession going forward.

So in this post, we learnt about new spark API called SparkSession. This will be our new entry point of spark code in future.

## **References**

Apache Spark 2.0 presented by Databricks co-founder Reynold Xin - <https://www.brighttalk.com/webcast/12891/202021>

# Introduction to Spark 2.0 - Part 2 : Wordcount in Dataset API

### **Introduction to Dataset**

Dataset is new abstraction in Spark introduced as alpha API in Spark 1.6. It’s becoming stable API in spark 2.0. It’s new single abstraction for all user land code in Spark.

From Definition, ” A ***Dataset*** is a strongly typed collection of domain-specific objects that can be transformed in parallel using functional or relational operations. Each dataset also has an untyped view called a ***DataFrame***, which is a Dataset of ***Row***. “

which sounds similar to RDD definition

” RDD represents an immutable,partitioned collection of elements that can be operated on in parallel “

The major difference is, dataset is collection of domain specific objects where as RDD is collection of any object. Domain object part of definition signifies the schema part of dataset. So dataset API is always strongly typed and optimized using schema where RDD is not.

Dataset definition also talks about Dataframes API. Dataframe is special dataset where there is no compilation checks for schema. So this makes dataSet new single abstraction replacing RDD from earlier versions of spark.

Once we understood the dataset abstraction, in rest of post we will see how to work with this abstraction.

## **Dataset Wordcount example**

As with any new API, we will learn API using how to use in WordCount example. The below is the code for wordcount in dataset API.

### **Step 1 : Create SparkSession**

As we discussed in last blog, we use spark session as entry point for dataset API.

**val** sparkSession **=** **SparkSession.**builder**.**

master**(**"local"**)**

**.**appName**(**"example"**)**

**.**getOrCreate**()**

### **Step 2 : Read data and convert to Dataset**

We read data using read.text API which is similar to textFile API of RDD. The as[String] part of code assigns the needed schema for dataset.

**import** sparkSession.implicits.\_

**val** data **=** sparkSession**.**read**.**text**(**"src/main/resources/data.txt"**).**as**[String]**

Here data will be of the type of DataSet[String]. Remember to import **sparkSession.implicits.\_** for all schema conversion magic.

### **Step 3 : Split and group by word**

Dataset mimics lot of RDD API’s like map, groupByKey etc. The below code we are splitting lines to get words and group them by words.

**val** words **=** data**.**flatMap**(**value **=>** value**.**split**(**"\\s+"**))**

**val** groupedWords **=** words**.**groupByKey**(\_.**toLowerCase**)**

One thing you may observed we don’t create a key/value pair. The reason is unlike RDD, dataset works in row level abstraction. Each value is treated a row with multiple columns and any column can act as key for grouping like in database.

### **Step 4 : Count**

Once we have grouped, we can count each word using count method. It’s similar to reduceByKey of RDD.

**val** counts **=** groupedWords**.**count**()**

### **Step 5 : Print results**

Finally once we count, we need to print the result. As with RDD, all the above API’s are lazy. We need to call an action to trigger computation. In dataset, show is one of those actions. It’s show first 20 results. If you want complete result, you can use collect API.

counts**.**show**()**

You can access complete code [here](https://github.com/phatak-dev/spark2.0-examples/blob/master/src/main/scala/com/madhukaraphatak/examples/sparktwo/DataSetWordCount.scala).

Now we have written our first example in dataset abstraction. We will explore more about dataset API in future posts.

# Introduction to Spark 2.0 - Part 3 : Porting Code from RDD API to Dataset API

## **RDD to Dataset**

Dataset API combines best of RDD and DataFrame API’s in one API. Many API’s in Dataset mimic the RDD API though they differ a lot in the implementation. So most of RDD code can be easily ported to Dataset API. In this post, I will be sharing few code snippets to show how a given code in RDD API can be written in Dataset API.

## **1. Loading Text Data**

### **RDD**

**val** rdd **=** sparkContext**.**textFile**(**"src/main/resources/data.txt"**)**

### **Dataset**

**val** ds **=** sparkSession**.**read**.**text**(**"src/main/resources/data.txt"**)**

## **2. Calculating count**

### **RDD**

rdd**.**count**()**

### **Dataset**

ds**.**count**()**

## **3. WordCount Example**

### **RDD**

**val** wordsRDD **=** rdd**.**flatMap**(**value **=>** value**.**split**(**"\\s+"**))**

**val** wordsPair **=** wordsRDD**.**map**(**word **=>** **(**word**,**1**))**

**val** wordCount **=** wordsPair**.**reduceByKey**(\_+\_)**

### **Dataset**

**import** sparkSession.implicits.\_

**val** wordsDs **=** ds**.**flatMap**(**value **=>** value**.**split**(**"\\s+"**))**

**val** wordsPairDs **=** wordsDs**.**groupByKey**(**value **=>** value**)**

**val** wordCountDs **=** wordsPairDs**.**count**()**

## **4. Caching**

### **RDD**

rdd**.**cache**()**

### **Dataset**

ds**.**cache**()**

## **5. Filter**

### **RDD**

**val** filteredRDD **=** wordsRDD**.**filter**(**value **=>** value **==**"hello"**)**

### **Dataset**

**val** filteredDS **=** wordsDs**.**filter**(**value **=>** value **==**"hello"**)**

## **6. Map Partitions**

### **RDD**

**val** mapPartitionsRDD **=** rdd**.**mapPartitions**(**iterator **=>** **List(**iterator**.**count**(**value **=>** **true)).**iterator**)**

### **Dataset**

**val** mapPartitionsDs **=** ds**.**mapPartitions**(**iterator **=>** **List(**iterator**.**count**(**value **=>** **true)).**iterator**)**

## **7. reduceByKey**

### **RDD**

**val** reduceCountByRDD **=** wordsPair**.**reduceByKey**(\_+\_)**

### **Dataset**

**val** reduceCountByDs **=** wordsPairDs**.**mapGroups**((**key**,**values**)** **=>(**key**,**values**.**length**))**

## **7. Conversions**

### **RDD**

**val** dsToRDD **=** ds**.**rdd

### **Dataset**

Converting a RDD to dataframe is little bit work as we need to specify the schema. Here we are showing how to convert RDD[String] to DataFrame[String].

**val** rddStringToRowRDD **=** rdd**.**map**(**value **=>** **Row(**value**))**

**val** dfschema **=** **StructType(Array(StructField(**"value"**,StringType)))**

**val** rddToDF **=** sparkSession**.**createDataFrame**(**rddStringToRowRDD**,**dfschema**)**

**val** rDDToDataSet **=** rddToDF**.**as**[String]**

## **8. Double Based Operations**

### **RDD**

**val** doubleRDD **=** sparkContext**.**makeRDD**(List(**1.0**,**5.0**,**8.9**,**9.0**))**

**val** rddSum **=**doubleRDD**.**sum**()**

**val** rddMean **=** doubleRDD**.**mean**()**

### **Dataset**

**val** rowRDD **=** doubleRDD**.**map**(**value **=>** **Row.**fromSeq**(List(**value**)))**

**val** schema **=** **StructType(Array(StructField(**"value"**,DoubleType)))**

**val** doubleDS **=** sparkSession**.**createDataFrame**(**rowRDD**,**schema**)**

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

doubleDS**.**agg**(**sum**(**"value"**))**

doubleDS**.**agg**(**mean**(**"value"**))**

## **9. Reduce API**

### **RDD**

**val** rddReduce **=** doubleRDD**.**reduce**((**a**,**b**)** **=>** a **+**b**)**

### **Dataset**

**val** dsReduce **=** doubleDS**.**reduce**((**row1**,**row2**)** **=>Row(**row1**.**getDouble**(**0**)** **+** row2**.**getDouble**(**0**)))**

You can access complete code [here](https://github.com/phatak-dev/spark2.0-examples/blob/master/src/main/scala/com/madhukaraphatak/examples/sparktwo/RDDToDataSet.scala).

The above code samples show how to move your RDD based code base to new Dataset API. Though it doesn’t cover complete RDD API, it should give you fair idea about how RDD and Dataframe API’s are related.

# Introduction to Spark 2.0 - Part 4 : Introduction to Catalog API

## **Catalog API**

DataSet with Dataframe API supports structured data analysis in spark. One of the important aspects of structured data analysis is managing metadata. It may be temporary metadata like temp table, registered udfs on SQL context or permanent metadata like Hive meta store or HCatalog.

In earlier versions of spark, there was no standard API to access this metadata. Users used to use queries like show tables and others to query this metadata. These queries often needed raw string manipulation and used to differ depending upon the underneath meta store.

But it’s changing in Spark 2.0.In Spark 2.0, spark has added a standard API called catalog for accessing metadata in spark SQL. This works both for spark sql and hive metadata.

In this post I will be discussing about how to work with catalog API.

## **Accessing Catalog**

Catalog is available on spark session. The following code shows how to access catalog.

**val** catalog **=** sparkSession**.**catalog

## **Querying the databases**

Once we have access to catalog, we can use it to query the databases. All the API’s on catalog returns a dataset.

catalog**.**listDatabases**().**select**(**"name"**).**show**()**

On catalog, listDatabases gives all the databases.By default, you will have only one database called default.In case of hive, it also access databases from the metastore. As the listDatabases returns a dataset, we can use all the operation available on dataset to query the metadata.

## **Registering Dataframe with createTempView**

In earlier versions of spark, we used to register a dataframe using registerTempTable. But in spark 2.0, this API is deprecated. The registerTempleTable API was one of the source of confusion as users used think it materializes the dataframe and saves as a temporary table which was not the case. So this API is replaced with createTempView.

createTempView can be used as follows.

df**.**createTempView**(**"sales"**)**

Once we have registered a view, we can query it using listTables.

## **Querying the tables**

As we can query databases, we can query tables. It lists all the temporary table registered in case of spark sql. In hive case, it lists all the tables in the underneath metadata store.

catalog**.**listTables**().**select**(**"name"**).**show**()**

## **Checking is table cached or not**

Catalog not only is used for querying. It can be used to check state of individual tables. Given a table, we can check is it cache or not. It’s useful in scenarios to make sure we cache the tables which are accessed frequently.

catalog**.**isCached**(**"sales"**)**

You will get false as by default no table will be cache. Now we cache the table and query again.

df**.**cache**()**

catalog**.**isCached**(**"sales"**)**

Now it will print true.

## **Drop view**

We can use catalog to drop views. In spark sql case, it will deregister the view. In case of hive, it will drops from the metadata store.

catalog**.**dropTempView**(**"sales"**)**

## **Query registered functions**

Catalog API not only allow us to interact with tables, it also allows us to interact with udf’s. The below code shows how to query all functions registered on spark session. They also include all built in functions.

catalog**.**listFunctions**().**

select**(**"name"**,**"description"**,**"className"**,**"isTemporary"**).**show**(**100**)**

You can access complete code [here](https://github.com/phatak-dev/spark2.0-examples/blob/master/src/main/scala/com/madhukaraphatak/examples/sparktwo/CatalogExample.scala).

Catalog is new API in spark 2.0 which allows us to interact with metadata of spark sql. This is a much better interface to metadata compared to earlier versions of spark.

## **References**

Jira for catalog API <https://issues.apache.org/jira/browse/SPARK-13477>.

# Introduction to Spark 2.0 - Part 5 : Time Window in Spark SQL

## **Window API in Spark SQL**

Spark introduced window API in 1.4 version to support smarter grouping functionalities. They are very useful for people coming from SQL background. One of the missing window API was ability to create windows using time. Time plays an important role in many industries like finance, telecommunication where understanding the data depending upon the time becomes crucial.

In Spark 2.0, framework has introduced built in support for time windows. These behave very similar to time windows in spark-streaming. In this blog post, I will be discussing about how to use this time window API.

## **Time Series Data**

Before we start doing time window, we need to have access to a time series data. For my example, I will be using data of Apple stock from 1980 to 2016. You can access the data [here](https://raw.githubusercontent.com/phatak-dev/spark2.0-examples/master/src/main/scala/com/madhukaraphatak/examples/sparktwo/TimeWindowExample.scala). The original source of data is [yahoo finance](https://in.finance.yahoo.com/q/hp?s=AAPL).

The data has six columns. Out of those six, we are only interested in Date, which signifies the date of trade and Close which signifies end of the day value.

## **Importing time series data to DataFrame**

Once we have time series data, we need to import it to dataframe. All the time window API’s need a column with type timestamp. Luckily spark-csv package can automatically infer the date formats from data and create schema accordingly. The below code is for importing with schema inference.

**val** stocksDF **=** sparkSession**.**read**.**option**(**"header"**,**"true"**).**

option**(**"inferSchema"**,**"true"**)**

**.**csv**(**"src/main/resources/applestock.csv"**)**

## **Find weekly average in 2016**

Once we have data is represented as dataframe, we can start doing time window analysis. In our analysis, we want to find weekly average of the stock for 2016. The below are the steps to do that.

### **Step 1 : Filter data for 2016**

As we are interested only in 2016, we need to filter the data for 2016. The below code show how to filter data on time.

**val** stocks2016 **=** stocksDF**.**filter**(**"year(Date)==2016"**)**

We can use builtin function year, as Date is already represented as a timestamp.

### **Step 2 : Tumbling window to calculate average**

Once we have filtered data, we need to create window for every 1 week. This kind of discretization of data is called as a tumbling window.

**val** tumblingWindowDS **=** stocks2016

**.**groupBy**(**window**(**stocks2016**.**col**(**"Date"**),**"1 week"**))**

**.**agg**(**avg**(**"Close"**).**as**(**"weekly\_average"**))**

The above code show how to use time window API. Window is normally used inside a group by. The first parameter signifies which column needs to be treated as time. Second parameter signifies the window duration. Window duration can be seconds, minutes, hours, days or weeks.

Once we have created window, we can run an aggregation like average as shown in the code.

### **Step 3 : Printing the window values**

Once we calculated the time window, we want to see the result.

printWindow**(**tumblingWindowDS**,**"weekly\_average"**)**

The above code uses a helper function called printWindow which takes aggregated window dataframe and aggregated column name. The helper function looks as follows.

**def** printWindow**(**windowDF**:DataFrame,** aggCol**:String)** **={**

windowDF**.**sort**(**"window.start"**).**

select**(**"window.start"**,**"window.end"**,**s"$aggCol"**).**

show**(**truncate **=** **false)**

**}**

In above function, we are sorting dataframe using window.start. This column signifies the start time of window. This sorting helps us to understand the output better. Once we have sorted, we print start,end, aggregated value. As the timestamp can be long, we tell the show not to truncate results for better display.

When you run the example, we see the below result.

+---------------------+---------------------+------------------+

|start |end |weekly\_average |

+---------------------+---------------------+------------------+

|2015-12-31 05:30:00.0|2016-01-07 05:30:00.0|101.30249774999999|

|2016-01-07 05:30:00.0|2016-01-14 05:30:00.0|98.47199859999999 |

|2016-01-14 05:30:00.0|2016-01-21 05:30:00.0|96.72000125000001 |

|2016-01-21 05:30:00.0|2016-01-28 05:30:00.0|97.6719984 |

One thing you may observe is the date is started from 31st and first week is considered till 7. But if you go through the data, the first entry for 2016 start from 2016-01-04. The reason is there was no trading on 1st as it’s new year, 2 and 3 as they are weekend.

We can fix this by specifying the start time for window, which signifies the offset from which window should start.

## **Time window with start time**

In earlier code, we used a tumbling window. In order to specify start time we need to use a sliding window. As of now, there is no API which combines tumbling window with start time. We can create tumbling window effect by keeping both window duration and slide duration same.

**val** windowWithStartTime **=** stocks2016**.**groupBy**(**window**(**stocks2016**.**col**(**"Date"**),**

"1 week"**,**"1 week"**,** "4 days"**))**

**.**agg**(**avg**(**"Close"**).**as**(**"weekly\_average"**))**

In above code, we specify “4 days” which is a offset for start time. The first two parameters specify window duration and slide duration.When we run this code, we observe the below result

+---------------------+---------------------+------------------+

|start |end |weekly\_average |

+---------------------+---------------------+------------------+

|2015-12-28 05:30:00.0|2016-01-04 05:30:00.0|105.349998 |

|2016-01-04 05:30:00.0|2016-01-11 05:30:00.0|99.0699982 |

|2016-01-11 05:30:00.0|2016-01-18 05:30:00.0|98.49999799999999 |

|2016-01-18 05:30:00.0|2016-01-25 05:30:00.0|98.1220016 |

Now we have a week starting from 2016-01-04. Still we have initial row which is take from 2015. The reason is, as our start time is 4 days, it creates a window till that time from last seven days.We can remove this row easily using filter as below.

**val** filteredWindow **=** windowWithStartTime**.**filter**(**"year(window.start)=2016"**)**

Now we will see the expected result.

+---------------------+---------------------+------------------+

|start |end |weekly\_average |

+---------------------+---------------------+------------------+

|2016-01-04 05:30:00.0|2016-01-11 05:30:00.0|99.0699982 |

|2016-01-11 05:30:00.0|2016-01-18 05:30:00.0|98.49999799999999 |

|2016-01-18 05:30:00.0|2016-01-25 05:30:00.0|98.1220016 |

|2016-01-25 05:30:00.0|2016-02-01 05:30:00.0|96.2539976 |

|2016-02-01 05:30:00.0|2016-02-08 05:30:00.0|95.29199960000001 |

You can access complete code [here](https://github.com/phatak-dev/spark2.0-examples/blob/master/src/main/scala/com/madhukaraphatak/examples/sparktwo/TimeWindowExample.scala).

So now we know how to use time windows in Spark 2.0. This is one of the powerful feature which helps in wide variety analysis in big data.

# Introduction to Spark 2.0 - Part 6 : Custom Optimizers in Spark SQL

## **Catalyst optimizer**

Spark SQL uses an optimizer called catalyst to optimize all the queries written both in spark sql and dataframe dsl. This optimizer makes queries run much faster than their RDD counterparts. Spark keeps on improving this optimizer every version in order to improve performance without changing user code.

Catalyst is a modular library which is build as a rule based system. Each rule in the the framework focuses on the specific optimization. For example, rule like ConstantFolding focuses on removing constant expression from the query. For more information catalyst, you can refer to my earlier talk on [anatomy of dataframe](http://blog.madhukaraphatak.com/anatomy-of-spark-dataframe-api).

In earlier versions of spark, if we wanted add our own optimizations, we need to change the source code of spark. This is not preferable in many cases where optimizations are only applicable to the domain or user specific problems. So developer community wanted to have a pluggable way to add their optimizations to the catalyst in runtime.

In Spark 2.0, we have an experimental API for adding user defined custom optimizations. In the rest of the blog I will be discussing about how to write an optimization rule and add it to catalyst.

## **Optimized plan for a dataframe**

Before we write our optimization rule, let’s understand how to access the optimized plan in spark. The below code shows a simple example

**val** df **=** sparkSession**.**read**.**option**(**"header"**,**"true"**).**csv**(**"src/main/resources/sales.csv"**)**

**val** multipliedDF **=** df**.**selectExpr**(**"amountPaid \* 1"**)**

println**(**multipliedDF**.**queryExecution**.**optimizedPlan**.**numberedTreeString**)**

In above code, we have loaded a csv file and multiplied one to one of the column. We can look at the optimized plan for that dataframe using optimizedPlan object on queryExecution. queryExecution allows us to access all the information related execution of the query. Optimized plan is one of them.

Every plan in spark is represented as a tree. So numberedTreeString method pretty prints the optimized plan. When we run this code we get below result.

00 Project [(cast(amountPaid#3 as double) \* 1.0) AS (amountPaid \* 1)#5]

01 +- Relation[transactionId#0,customerId#1,itemId#2,amountPaid#3] csv

All plans are read bottom to top. The below are the two nodes of tree

* 01 Relation - Signifies the dataframe we created from csv file
* 00 Project - Signifies the projection

You can observe some of the casts added by the spark for correct results.

## **Writing an optimizer rule**

From the above plan, it’s clear that its going to multiply 1.0 to each of the value of column. But it’s not optimal plan. Whenever we see 1 in multiplication, we know it’s going to return exact same value. We can use this knowledge to write a rule and add smartness to the optimizer.

The following code show how to write such a rule.

**object** **MultiplyOptimizationRule** **extends** **Rule[LogicalPlan]** **{**

**def** apply**(**plan**:** **LogicalPlan):** **LogicalPlan** **=** plan transformAllExpressions **{**

**case** **Multiply(**left**,**right**)** **if** right**.**isInstanceOf**[Literal]** **&&**

right**.**asInstanceOf**[Literal].**value**.**asInstanceOf**[Double]** **==** 1.0 **=>**

println**(**"optimization of one applied"**)**

left

**}**

**}**

Here we are extending from Rule which operates on logical plan. Most of the rules are written as pattern matching in scala. In code, we are checking is the right operand is literal and it’s value is 1.0. Here we are very specific about where value 1 should appear. If it appears on the left it will not optimize. As it’s for example, for brevity I have not included checking for left also. But you can easily add.

So whenever we have right value as 1, we will skip the right expression altogether and return left.

## **Integrating our optimizer rule**

Once we have our rule, next step is to add to the optimizer. The below code shows that.

sparkSession**.**experimental**.**extraOptimizations **=** **Seq(MultiplyOptimizationRule)**

On spark session, we have an experimental object which exposes all the experimental API’s. Using this API, you can add list of custom rules to catalyst with extraOptimizations.

## **Using the custom optimization**

Once we have our rule added, we need to check it is applied or not. We will do same manipulation again as below code.

**val** multipliedDFWithOptimization **=** df**.**selectExpr**(**"amountPaid \* 1"**)**

println**(**"after optimization"**)**

println**(**multipliedDFWithOptimization**.**queryExecution**.**

optimizedPlan**.**numberedTreeString**)**

If we observe the output now,

00 Project [cast(amountPaid#3 as double) AS (amountPaid \* 1)#7]

01 +- Relation[transactionId#0,customerId#1,itemId#2,amountPaid#3] csv

You can observe now that multiplication is gone. This denotes the our optimization is applied. You can access complete code [here](https://github.com/phatak-dev/spark2.0-examples/blob/master/src/main/scala/com/madhukaraphatak/examples/sparktwo/CustomOptimizationExample.scala).

In Spark 2.0 users can add their own custom rules to catalyst to optimize their code. This makes spark more developer friendly and powerful generic engine.

## **References**

Catalyst: Allow adding custom optimizers - <https://issues.apache.org/jira/browse/SPARK-9843>