**Introduction to Spark for Developers and Data Scientists**

**What is Spark?**

Spark is “a fast and general engine for large-scale data processing”. – <http://spark.apache.org/>

Spark is also one of the most popular open source frameworks for big data, based on number of contributors. Let us find out why this is the case.

**When do you use Spark?**

Suppose you would like to analyze a data set: perform ETL or data munging, then run SQL queries such as grouping and aggregations against the data, and maybe apply a machine learning algorithm. When the data size is small, everything will run quickly on a single machine, and you can use analysis tools like Pandas (Python), R, or Excel, or write your own scripts. But, for larger data sets, data processing will be too slow on a single machine, and then you will want to move to a cluster of machines. This is when you would use Spark.

You could probably benefit from Spark if:

* Your data is currently stored in Hadoop / HDFS.
* Your data set contains more than 100 million rows.
* Ad-hoc queries take longer than 5 minutes to complete.

**What Spark is Not, typical architecture**

Spark can be a central component to a big data system, but it is not the only component. It is not a distributed file system: you would typically store your data on HDFS or S3. Nor is Spark a NoSQL database: Cassandra or HBase would be a better place for horizontally scalable table storage. And, it is not a message queue: you would use Kafka or Flume to collect streaming event data. Spark is, however, a compute engine which can take input or send output to all of these other systems.

**How do you use Spark?**

Implemented in Scala, Spark can be programmed in Scala, Java, Python, SQL, and R. However, not all of the latest functionality is immediately available in all languages.

**What kind of operations does Spark support?**

**Spark SQL**. Spark supports batch operations involved in ETL and data munging, via the DataFrame API. It supports parsing different input formats, such as JSON or Parquet. Once the raw data is loaded, you can easily compute new columns from existing columns. You can slice and dice the data by filtering, grouping, aggregating, and joining with other tables. Spark supports relational queries, which you can express in SQL or through the DataFrame API.

**Spark Streaming**. Spark also provides scalable stream processing. Given an input data stream, for example, coming from Kafka, Spark allows you to perform operations on the streaming data, such as map, reduce, join, and window.

**MLlib**. Spark includes a machine learning library, with scalable algorithms for classification, regression, collaborative filtering, clustering, and more. If training your dataset on a single machine takes too long, you might consider cluster computing with Spark.

**GraphX**. Finally, GraphX is a component in Spark for scalable batch processing on graphs.

As you may have noticed by now, Spark processing is batch oriented. It works best when you want to perform the same operation on all of your data, or a large subset of your data. Even with Spark Streaming, you operate on small batches of the data stream, rather than one event at a time.

**Spark vs. Hadoop MapReduce, Hive, Impala**

How does Spark compare with other big data compute engines? Unlike Hadoop MapReduce, Spark caches data in memory for huge performance gains when you have ad-hoc queries or iterative workloads, which are common in machine learning algorithms. Hive and Impala both run SQL queries at scale; the advantage of Spark over these systems is (1) the convenience of writing both queries and UDFs in the same language, such as Scala, and (2) support for machine learning algorithms, streaming data, and graph processing within the same system.

**Conclusion**

This overview has explained what Spark is, when to use it, what kinds of operations it supports, and how it compares with other big data systems. To learn more, please take a look at the Spark website

### 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]]

**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>

**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" \l "org.apache.spark.package" \t "_blank).

**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" \l "org.apache.spark.sql.DataFrameReader" \t "_blank) 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" \l "org.apache.spark.sql.DataFrameStatFunctions" \t "_blank) 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")

**Continuation**

To read about untyped operations with DataFrames, continue onto [part 2](http://xinhstechblog.blogspot.com/2016/07/overview-of-spark-20-dataset-dataframe_29.html).

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

**Introduction**

In [Part 1](http://xinhstechblog.blogspot.com/2016/07/overview-of-spark-20-dataset-dataframe.html) of this series, we examined type-safe operations with Datasets. In Part 2, we will cover untyped operations with DataFrames. Being untyped, DataFrames are well-suited for data exploration, ad-hoc queries, and data munging.

**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.

**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" \l "org.apache.spark.sql.DataFrameReader" \t "_blank) 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" \l "org.apache.spark.sql.SQLContext" \t "_blank), its read() method returns a [DataFrameReader](http://spark.apache.org/docs/latest/api/scala/index.html" \l "org.apache.spark.sql.DataFrameReader" \t "_blank), 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" \l "org.apache.spark.sql.DataFrame" \t "_blank): **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" \l "org.apache.spark.sql.GroupedData" \t "_blank)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" \l "org.apache.spark.sql.DataFrame" \t "_blank):  
**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" \l "org.apache.spark.sql.DataFrameStatFunctions" \t "_blank) 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" \l "org.apache.spark.sql.expressions.WindowSpec" \t "_blank):

> 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" \l "org.apache.spark.sql.GroupedData" \t "_blank) 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).

### Storm vs. Spark Streaming: Side-by-side comparison

### Overview

Both Storm and Spark Streaming are open-source frameworks for distributed stream processing. But, there are important differences as you will see in the following side-by-side comparison.

#### Processing Model, Latency

Although both frameworks provide scalability and fault tolerance, they differ fundamentally in their processing model. Whereas Storm processes incoming events **one at a time**, Spark Streaming [**batches up events**](http://www.cs.berkeley.edu/~matei/papers/2012/hotcloud_spark_streaming.pdf) that arrive within a short time window before processing them. Thus, Storm can achieve **sub-second latency** of processing an event, while Spark Streaming has a latency of several seconds.

#### Fault Tolerance, Data Guarantees

However, the tradeoff is in the fault tolerance data guarantees. Spark Streaming provides better support for **stateful computation** that is fault tolerant. In Storm, each individual record has to be tracked as it moves through the system, so Storm only guarantees that each record will be processed ***at least once***, but allows duplicates to appear during recovery from a fault. That means mutable state may be incorrectly updated twice.

Spark Streaming, on the other hand, need only track processing at the batch level, so it can efficiently guarantee that each mini-batch will be processed ***exactly once***, even if a fault such as a node failure occurs. [Actually, Storm's [Trident library](http://storm.incubator.apache.org/documentation/Trident-tutorial.html) also provides exactly once processing. But, it relies on transactions to update state, which is slower and often has to be implemented by the user.]

|  |
| --- |
| <http://4.bp.blogspot.com/-PRwOafv3eq8/U5dxChy-wlI/AAAAAAAAAE0/RSTposbv5hE/s1600/Screen+Shot+2014-06-10+at+1.54.54+PM.png> |
| Storm vs. Spark Streaming comparison. |

#### Summary

In short, **Storm is a good choice if you need sub-second latency and no data loss**. **Spark Streaming is better if you need stateful computation, with the guarantee that each event is processed exactly once**. Spark Streaming programming logic may also be easier because it is similar to batch programming, in that you are working with batches (albeit very small ones).

### Implementation, Programming API

#### Implementation

Storm is primarily [**implemented in Clojure**](https://github.com/apache/incubator-storm), while Spark Streaming is [implemented in Scala](https://github.com/apache/spark/tree/master/streaming). This is something to keep in mind if you want to look into the code to see how each system works or to make your own customizations. Storm was developed at BackType and Twitter; Spark Streaming was developed at UC Berkeley.

#### Programming API

Storm comes with a Java API, as well as support for other languages. Spark Streaming can be programmed in Scala as well as Java.

#### Batch Framework Integration

One nice feature of Spark Streaming is that it runs on Spark. Thus, **you can use the same (or very similar) code that you write for batch processing** and/or interactive queries in Spark, on Spark Streaming. This reduces the need to write separate code to process streaming data and historical data.

|  |
| --- |
| <http://1.bp.blogspot.com/-VMH2q0HElzg/U5ioS8EXsWI/AAAAAAAAAFE/c_1NMsXqjDY/s1600/Screen+Shot+2014-06-11+at+12.04.03+PM.png> |
| Storm vs. Spark Streaming: implementation and programming API. |

#### Summary

**Two advantages of Spark Streaming are that (1) it is not implemented in Clojure :) and (2) it is well integrated with the Spark batch computation framework**.

### Production, Support

#### Production Use

Storm has been around for several years and has run in production at Twitter since 2011, as well as at [many other companies](https://github.com/nathanmarz/storm/wiki/Powered-By). Meanwhile, Spark Streaming is a newer project; its only production deployment (that I am aware of) has been at [Sharethrough](http://blog.cloudera.com/blog/2014/03/letting-it-flow-with-spark-streaming/" \t "_blank) since 2013.

#### Hadoop Distribution, Support

Storm is the streaming solution in the [Hortonworks Hadoop data platform](http://hortonworks.com/hdp/" \t "_blank), whereas Spark Streaming is in both [MapR's distribution](http://www.mapr.com/products/apache-spark" \t "_blank) and [Cloudera's Enterprise data platform](http://www.cloudera.com/content/cloudera/en/products-and-services/cloudera-enterprise.html" \t "_blank). In addition, [Databricks](http://databricks.com/" \t "_blank) is a company that provides support for the Spark stack, including Spark Streaming.

#### Cluster Manager Integration

Although both systems can run on their own clusters, Storm also [runs on Mesos](https://github.com/mesosphere/storm-mesos%5d), while Spark Streaming runs on both YARN and Mesos.

|  |
| --- |
| <http://3.bp.blogspot.com/-9N32eR-SPH4/U79Fr2NsWbI/AAAAAAAAAFs/i4vv5esEtZU/s1600/Screen+Shot+2014-07-10+at+6.58.35+PM.png> |
| Storm vs. Spark Streaming: production and support. |

#### Summary

**Storm has run in production much longer than Spark Streaming. However, Spark Streaming has the advantages that (1) it has a company dedicated to supporting it (Databricks), and (2) it is compatible with YARN.**

### Further Reading

For an overview of Storm, see these [slides](http://www.slideshare.net/nathanmarz/storm-distributed-and-faulttolerant-realtime-computation).  
  
For a good overview of Spark Streaming, see the [slides](http://spark.incubator.apache.org/talks/strata_spark_streaming.ppt) to a Strata Conference talk. A more detailed description can be found in this [research paper](http://www.cs.berkeley.edu/~matei/papers/2012/hotcloud_spark_streaming.pdf).  
  
**Update**: A couple of readers have mentioned this other [comparison of Storm and Spark Streaming](http://www.slideshare.net/ptgoetz/apache-storm-vs-spark-streaming) from Hortonworks, written in defense of Storm's features and performance.  
  
**April, 2015**: Closing off comments now, since I don't have time to answer questions or keep this doc up-to-date.