COMP9313: Big Data Management

Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

Chapter 5.2: Spark IV

Part 1: Spark Structured APIs

A Brief Review of RDD

- The RDD is the most basic abstraction in Spark. There are three vital characteristics associated with an RDD:
  - Dependencies (lineage)
    - When necessary to reproduce results, Spark can recreate an RDD from the dependencies and replicate operations on it. This characteristic gives RDDs resiliency.
  - Partitions (with some locality information)
    - Partitions provide Spark the ability to split the work to parallelize computation on partitions across executors
    - Reading from HDFS—Spark will use locality information to send work to executors close to the data
  - Compute function: Partition => Iterator[T]
    - An RDD has a compute function that produces an Iterator[T] for the data that will be stored in the RDD.

Compute Average Values for Each Key

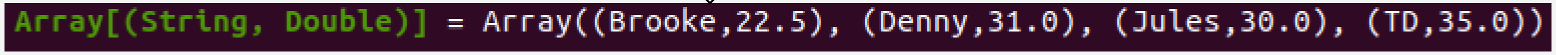
- Assume that we want to aggregate all the ages for each name, group by name, and then compute the average age for each name

```
val dataRDD = sc.parallelize(List(("Brooke", 20), ("Denny",
31), ("Jules", 30),("TD", 35), ("Brooke", 25)))
```

```
dataRDD.map(x=> (x._1, (x._2, 1)))
```

```
.reduceByKey((a, b)=> (a._1 + b._1, a._2 + b._2))
```

```
.map(x => (x._1, x._2._1.toDouble/x._2._2)))
```



```
Array[(String, Double)] = Array((Brooke,22.5), (Denny,31.0), (Jules,30.0), (TD,35.0))
```

Problems of RDD Computation Model

- The compute function (or computation) is opaque to Spark
  - Whether you are performing a join, filter, select, or aggregation, Spark only sees it as a lambda expression

```
dataRDD.map(x=> (x._1, (x._2, 1)))
```

- Spark has no way to optimize the expression, because it's unable to inspect the computation or expression in the function.
- Spark has no knowledge of the specific data type in RDD
  - To Spark it's an opaque object; it has no idea if you are accessing a column of a certain type within an object

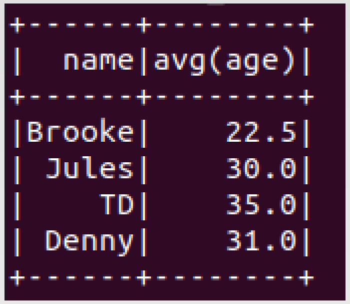
Spark's Structured APIs

- Spark 2.x introduced a few key schemes for structuring Spark,
- This specificity is further narrowed through the use of a set of common operators in a DSL (domain specific language), including the Dataset APIs and DataFrame APIs
  - These operators let you tell Spark what you wish to compute with your data
  - It can construct an efficient query plan for execution.
- Structure yields a number of benefits, including better performance and space efficiency across Spark components

Spark's Structured APIs

- E.g, for the average age problem, using the DataFrame APIs:

```
import spark.implicits._

val data_df = List(("Brooke", 20), ("Denny",
31), ("Jules", 30),("TD", 35), ("Brooke",
25)).toDF("name", "age")

data_df.groupBy("name").agg(avg("age")).show()
```



```
+------+--------+
|  name|avg(age)|
+------+--------+
|Brooke|    22.5|
| Jules|    30.0|
|    TD|    35.0|
| Denny|    31.0|
+------+--------+
```

- Spark now knows exactly what we wish to do: group people by their names, aggregate their ages, and then compute the average age of all people with the same name.
- Spark can inspect or parse this query and understand our intention, and thus it can optimize or arrange the operations for efficient execution.

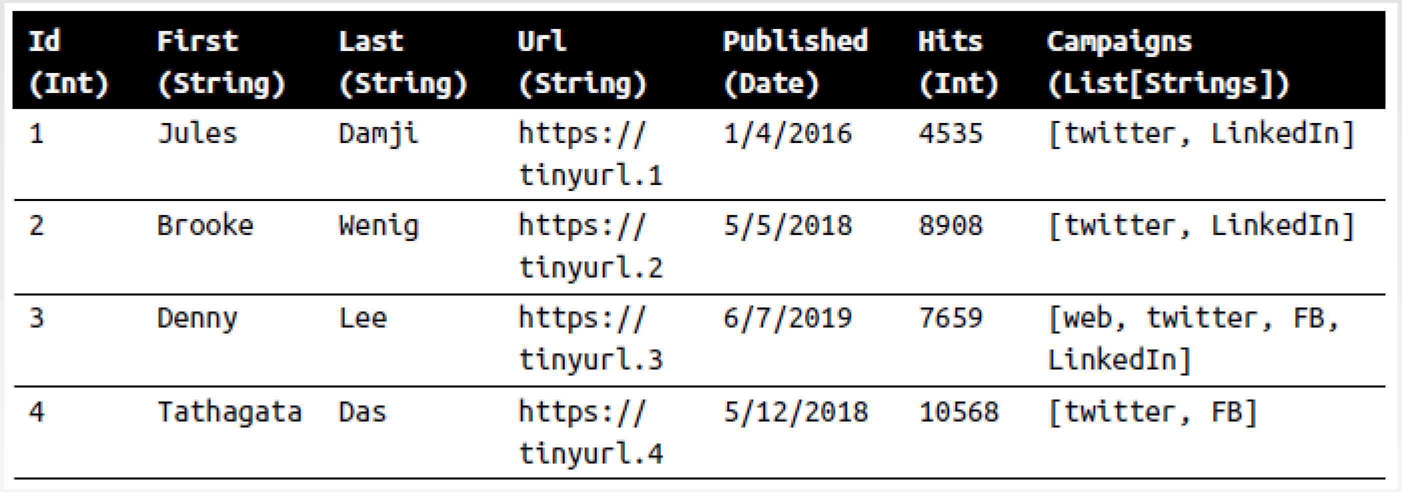
Datasets and DataFrames

- A *Dataset* is a distributed collection of data
  - provides the benefits of RDDs (e.g., strong typing) with the benefits of Spark SQL's optimized execution engine
  - A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, etc.)

- A *DataFrame* is a *Dataset* organized into named columns
  - It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations
  - An abstraction for selecting, filtering, aggregating and plotting structured data
  - A DataFrame can be represented by a Dataset of Rows
    - Scala: DataFrame is simply a type alias of Dataset[Row]
    - Java: use Dataset<Row> to represent a DataFrame

DataFrame API

- Spark DataFrames are like distributed in-memory tables with named columns and schemas, where each column has a specific data type.
- When data is visualized as a structured table, it’s not only easy to digest but also easy to work with

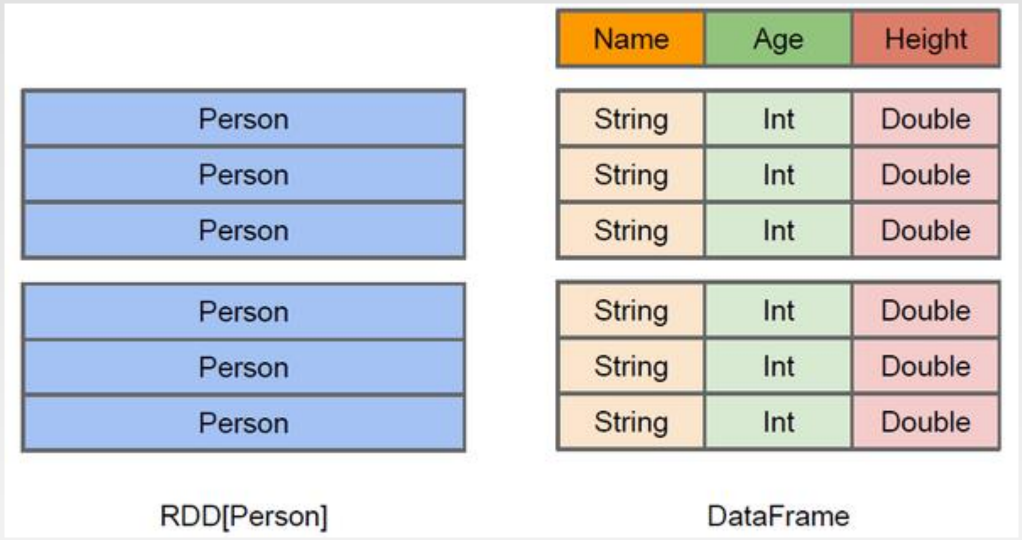


Id (Int)	First (String)	Last (String)	Url (String)	Published (Date)	Hits (Int)	Campaigns (List[Strings])
1	Jules	Damji	https:// tinyurl.1	1/4/2016	4535	[twitter, LinkedIn]
2	Brooke	Wenig	https:// tinyurl.2	5/5/2018	8908	[twitter, LinkedIn]
3	Denny	Lee	https:// tinyurl.3	6/7/2019	7659	[web, twitter, FB, LinkedIn]
4	Tathagata	Das	https:// tinyurl.4	5/12/2018	10568	[twitter, FB]

The table-like format of a DataFrame

Difference between DataFrame and RDD

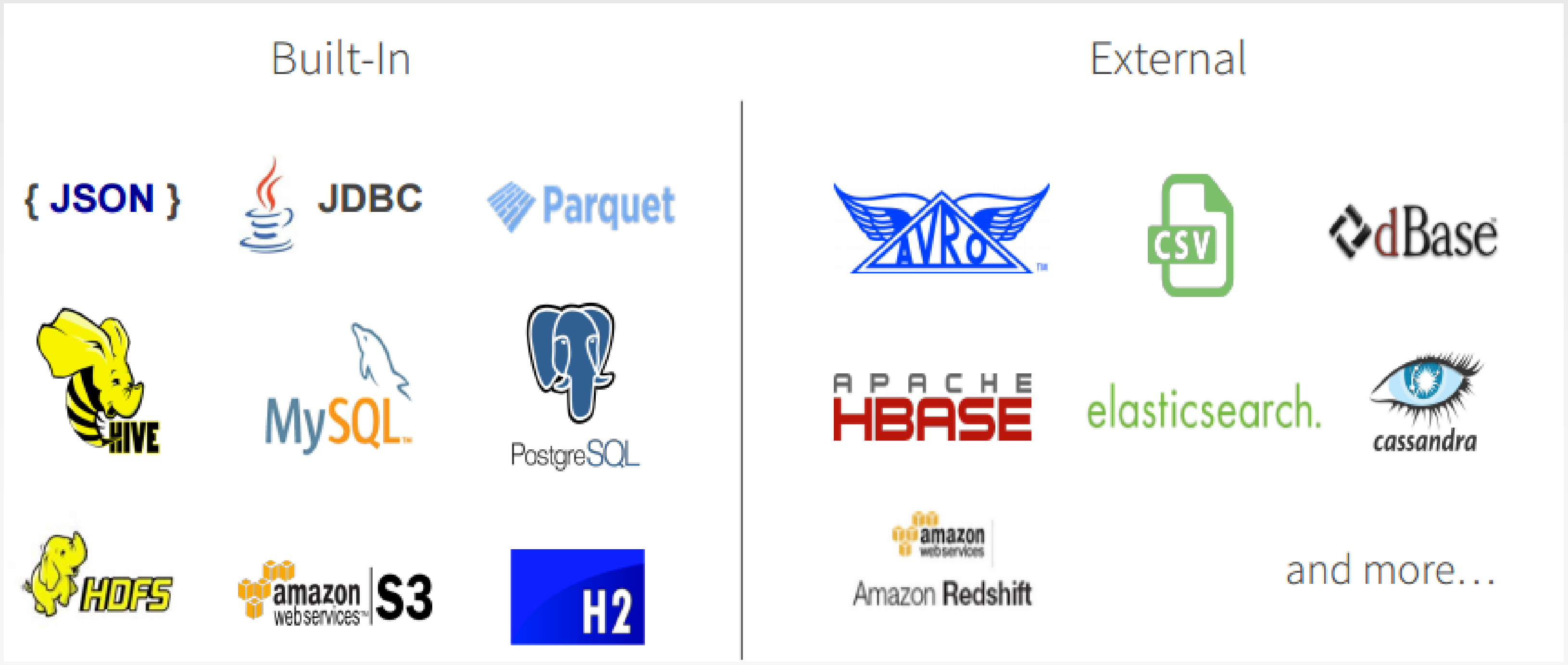
- DataFrame more like a traditional database of two-dimensional form, in addition to data, but also to grasp the structural information of the data, that is, schema



  - RDD[Person] although with Person for type parameters, but the Spark framework itself does not understand internal structure of Person class
  - DataFrame has provided a detailed structural information, making Spark SQL can clearly know what columns are included in the dataset, and what is the name and type of each column. Thus, Spark SQL query optimizer can target optimization

DataFrame Data Sources

- Spark SQL's Data Source API can read and write DataFrames using a variety of formats.
  - E.g., structured data files, tables in Hive, external databases, or existing RDDs
  - In the Scala API, DataFrame is simply a type alias of Dataset[Row]



Create DataFrames

- We first need to import “spark.implicits._”. The implicits object gives implicit conversions for converting Scala objects (incl. RDDs) into a Dataset or DataFrame

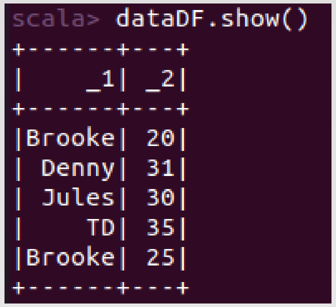
```
import spark.implicits._

// Given a list of pairs including names and ages
val data = List(("Brooke", 20), ("Denny", 31), ("Jules",
30),("TD", 35), ("Brooke", 25))

// Create DataFrame' from ‘RDD’ and the schema
val dataDF = spark.createDataFrame(data)
```

- You can also convert an RDD into a DataFrame

```
import spark.implicits._

// Given a list of pairs including name and age
val data = sc.parallelize(Seq(("Brooke", 20), ("Denny", 31),
("Jules", 30),("TD", 35), ("Brooke", 25)))

// Create DataFrame' from ‘RDD’ and the schema
val dataDF = spark.createDataFrame(data)
```

Create DataFrames

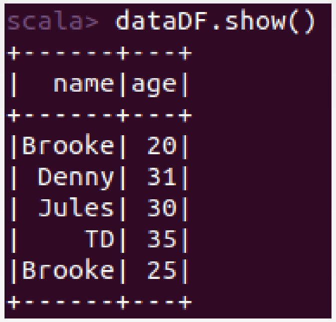
- Using the above method, we can get the DataFrame as below:



```
scala> dataDF.show()
+------+---+
|    _1| _2|
+------+---+
|Brooke| 20|
| Denny| 31|
| Jules| 30|
|    TD| 35|
|Brooke| 25|
+------+---+
```

- We can see that the schema is not defined, and the columns have no meaningful names. To define the names for columns, we can use the the toDF() method

```
val dataDF = spark.createDataFrame(data).toDF("name", "age")
```



```
scala> dataDF.show()
+------+---+
|  name|age|
+------+---+
|Brooke| 20|
| Denny| 31|
| Jules| 30|
|    TD| 35|
|Brooke| 25|
+------+---+
```

- We can also write (data could be a list or an RDD):

```
val dataDF = data.toDF("name", "age")
```

Schemas in Spark

- A schema in Spark defines the column names and associated data types for a DataFrame
- Defining a schema up front offers three benefits
  - You relieve Spark from the onus of inferring data types.
  - You prevent Spark from creating a separate job just to read a large portion of your file to ascertain the schema, which for a large data file can be expensive and time-consuming.
  - You can detect errors early if data doesn't match the schema.
- Define a DataFrame programmatically with three named columns, author, title, and pages

```
import org.apache.spark.sql.types._
val schema = StructType(Array(StructField("author", StringType, false),
StructField("title", StringType, false),
StructField("pages", IntegerType, false)))
```

Spark's Basic Data Types

- Spark supports basic internal data types, which can be declared in your Spark application or defined in your schema

Data type	Value assigned in Scala	API to instantiate
ByteType	Byte	DataTypes.ByteType
ShortType	Short	DataTypes.ShortType
IntegerType	Int	DataTypes.IntegerType
LongType	Long	DataTypes.LongType
FloatType	Float	DataTypes.FloatType
DoubleType	Double	DataTypes.DoubleType
StringType	String	DataTypes.StringType
BooleanType	Boolean	DataTypes.BooleanType
DecimalType	java.math.BigDecimal	DecimalType

Spark’s Structured and Complex Data Types

- For complex data analytics, you’ll need Spark to handle complex data types, such as maps, arrays, structs, dates, timestamps, fields, etc.

Data type	Value assigned in Scala	API to instantiate
BinaryType	Array[Byte]	DataTypes.BinaryType
Timestamp Type	java.sql.Timestamp	DataTypes.TimestampType
DateType	java.sql.Date	DataTypes.DateType
ArrayType	scala.collection.Seq	DataTypes.createArrayType(ElementType)
MapType	scala.collection.Map	DataTypes.createMapType(keyType, valueType)
StructType	org.apache.spark.sql.Row	StructType(ArrayType[fieldTypes])
StructField	A value type corresponding to the type of this field	StructField(name, dataType, [nullable])

Create DataFrames with Schema

- We can use spark.createDataFrame(data, schema) to create DataFrame, after the schema is defined for the data.
  - The first argument data must be of type *RDD[Row]*
  - The second argument schema must of type *StructType*

```
import org.apache.spark.sql.types._
import org.apache.spark.sql._
// Create the schema
val schema = StructType(Array(StructField("name", StringType,
false), StructField("age", IntegerType, false)))

// Given a list of pairs including names and ages
val data = List(("Brooke", 20), ("Denny", 31), ("Jules",
30),("TD", 35), ("Brooke", 25))

// Create 'Row' from 'Seq'
val row = Row.fromSeq(data)

// Create 'RDD' from 'Row'
val rdd = spark.sparkContext.makeRDD(List(row))

// Create DataFrame' from ‘RDD’ and the schema
val dataDF = spark.createDataFrame(rdd, schema)
```

Create DataFrames with Schema

- In order to convert the List to *RDD[Row]*, you can also do as below

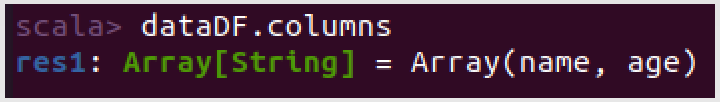
```
import org.apache.spark.sql.types._
import org.apache.spark.sql._

// Create the schema
val schema = StructType(Array(StructField("name", StringType,
false), StructField("age", IntegerType, false)))

// Given a list of pairs including names and ages
val data = List(("Brooke", 20), ("Denny", 31), ("Jules",
30),("TD", 35), ("Brooke", 25))

// Create 'RDD' from ‘List'
val rdd = spark.sparkContext.parallelize(data)

// Transform the pair (String, Integer) to a Row object
val rddRow = rdd.map(x => Row(x._1, x._2))

// Create DataFrame' from ‘RDD’ and the schema
val dataDF = spark.createDataFrame(rddRow, schema)
```

- You can also create a DataFrame from a json file:

```
val blogsDF = spark.read.schema(schema).json(jsonFile)
```

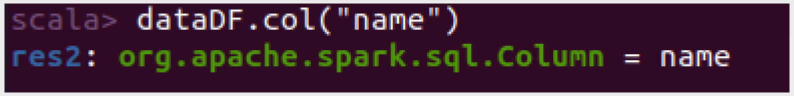
Columns

- Each column describe a type of field
- We can list all the columns by their names, and we can perform operations on their values using relational or computational expressions
  - List all the columns



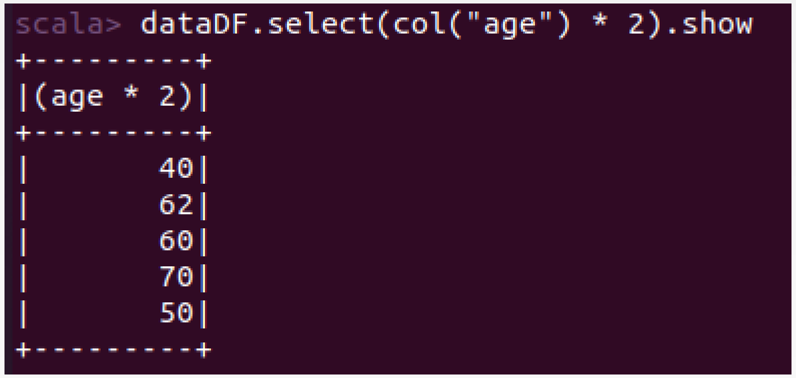
```
scala> dataDF.columns
res1: Array[String] = Array(name, age)
```

  - Access a particular column with col and it returns a Column type



```
scala> dataDF.col("name")
res2: org.apache.spark.sql.Column = name
```

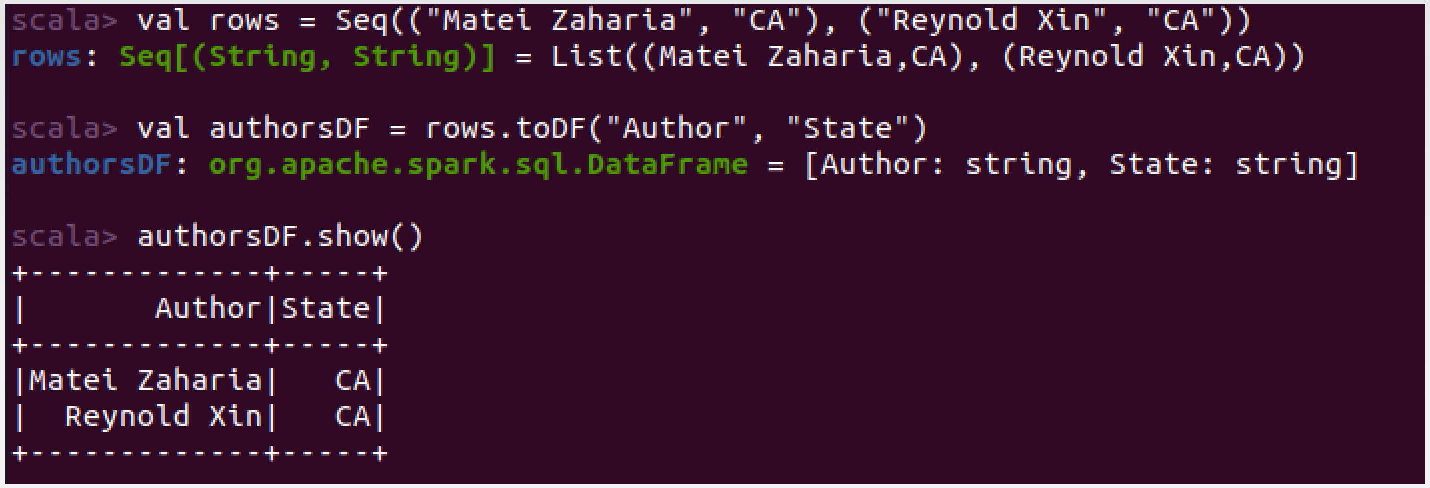
  - We can also use logical or mathematical expressions on columns



```
scala> dataDF.select(col("age") * 2).show
+---------+
|(age * 2)|
+---------+
|       40|
|       62|
|       60|
|       70|
|       50|
+---------+
```

Rows

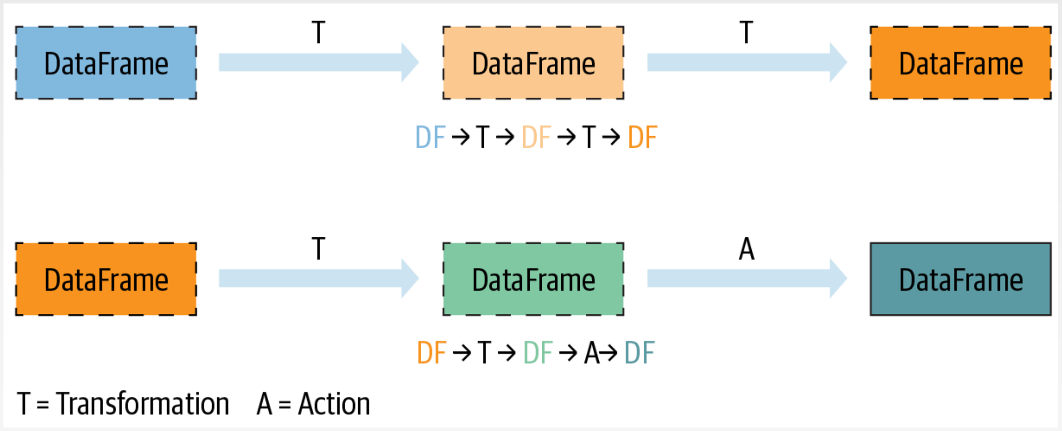
- A row in Spark is a generic Row object, containing one or more columns
- Row is an object in Spark and an ordered collection of fields, we can access its fields by an index starting at 0
- Row objects can be used to create DataFrames



```
scala> val rows = Seq(("Matei Zaharia", "CA"), ("Reynold Xin", "CA"))
rows: Seq[(String, String)] = List((Matei Zaharia,CA), (Reynold Xin,CA))

scala> val authorsDF = rows.toDF("Author", "State")
authorsDF: org.apache.spark.sql.DataFrame = [Author: string, State: string]

scala> authorsDF.show()
+-------------+-----+
|       Author|State|
+-------------+-----+
|Matei Zaharia|   CA|
|  Reynold Xin|   CA|
+-------------+-----+
```

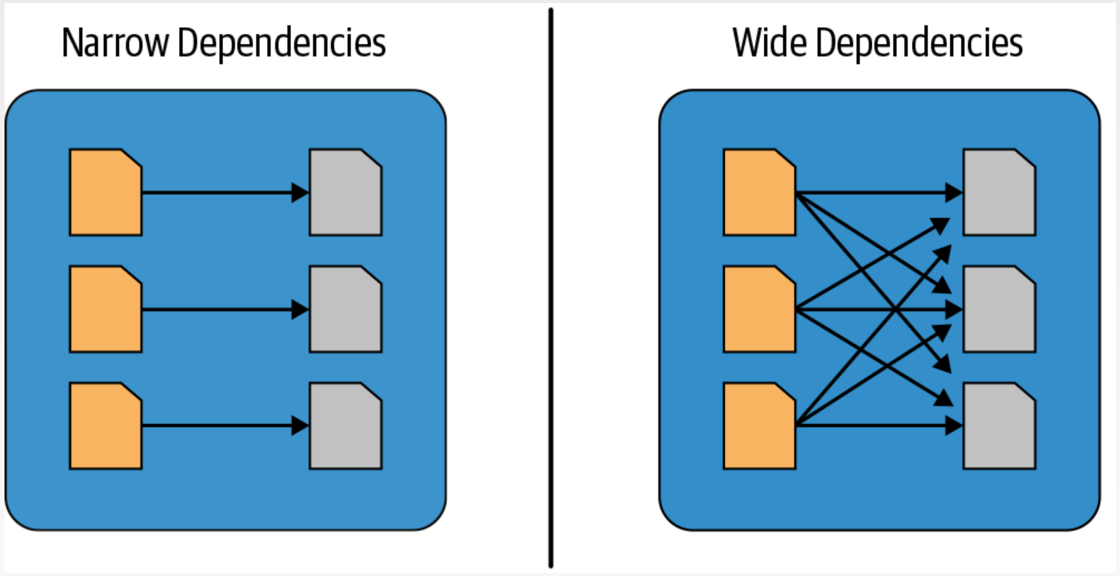
Transformations, Actions, and Lazy Evaluation

- Spark DataFrame operations can also be classified into two types: transformations and actions.
  - All transformations are evaluated lazily - their results are not computed immediately, but they are recorded or remembered as a lineage
  - An action triggers the lazy evaluation of all the recorded transformations



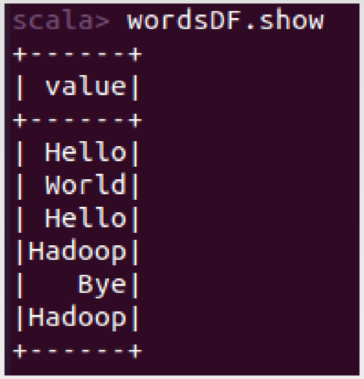
Narrow and Wide Transformations

- Transformations can be classified as having either narrow dependencies or wide dependencies
  - Any transformation where a single output partition can be computed from a single input partition is a narrow transformation, like filter()
  - Any transformation where data from other partitions is read in, combined, and written to disk is a wide transformation, like groupBy()



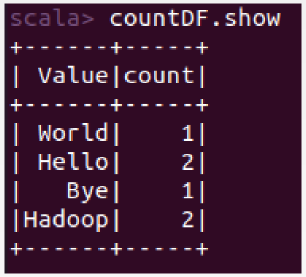
WordCount using DataFrame

```
val fileRDD =
spark.sparkContext.textFile("file:///home/comp9313/inputText")
val wordsDF = fileRDD.flatMap(_.split(" ")).toDF
```



```
scala> wordsDF.show
+------+
| value|
+------+
| Hello|
| World|
| Hello|
|Hadoop|
|   Bye|
|Hadoop|
+------+
```

```
val countDF = wordsDF.groupBy("Value").count()
```



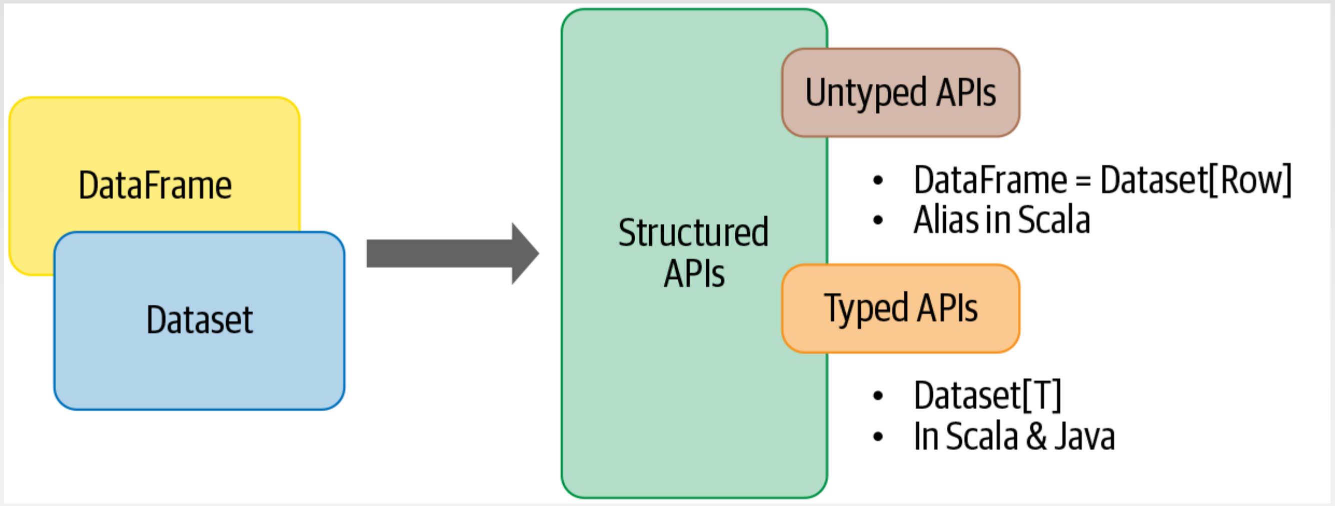
```
scala> countDF.show
+------+-----+
| Value|count|
+------+-----+
| World|    1|
| Hello|    2|
|   Bye|    1|
|Hadoop|    2|
+------+-----+
```

```
countDF.collect.foreach(println)
```

```
countDF.write.format(“csv”).save("file:///home/comp9313/output")
```

DataSet

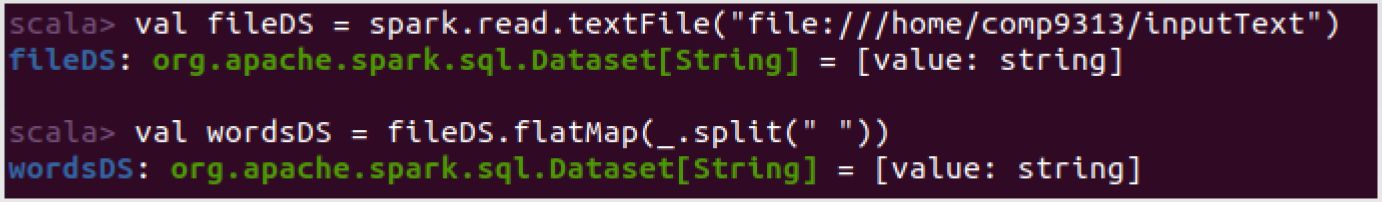
- Spark 2.0 unified the DataFrame and Dataset APIs as Structured APIs with similar interfaces
- Datasets take on two characteristics: typed and untyped APIs



- Conceptually, you can think of a DataFrame in Scala as an alias for Dataset[Row]

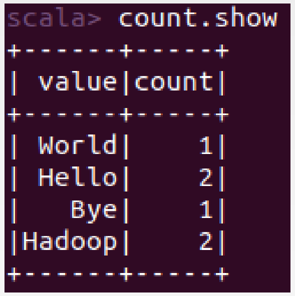
WordCount using DataSet

```
val fileDS = spark.read.textFile("file:///home/comp9313/inputText")
val wordsDS = fileDS.flatMap(_.split(" "))
```



```
scala> val fileDS = spark.read.textFile("file:///home/comp9313/inputText")
fileDS: org.apache.spark.sql.Dataset[String] = [value: string]

scala> val wordsDS = fileDS.flatMap(_.split(" "))
wordsDS: org.apache.spark.sql.Dataset[String] = [value: string]
```

```
val countDF = wordsDS.groupBy("Value").count()
```



```
scala> count.show
+------+-----+
| value|count|
+------+-----+
| World|    1|
| Hello|    2|
|   Bye|    1|
|Hadoop|    2|
+------+-----+
```

```
countDF.collect.foreach(println)
```

```
countDF.write.format(“csv”).save("file:///home/comp9313/output")
```

DataFrames Versus Datasets

- If you want to tell Spark what to do, not how to do it, use DataFrames or Datasets.
- If you want rich semantics, high-level abstractions, and DSL operators, use DataFrames or Datasets.
- If your processing demands high-level expressions, filters, maps, aggregations, computing averages or sums, SQL queries, columnar access, or use of relational operators on semi-structured data, use DataFrames or Datasets.
- If your processing dictates relational transformations similar to SQL-like queries, use DataFrames.
- If you want unification, code optimization, and simplification of APIs across Spark components, use DataFrames.
- If you want space and speed efficiency, use DataFrames.

- More examples of DataFrame usage could be found at: https://github.com/databricks/LearningSparkV2

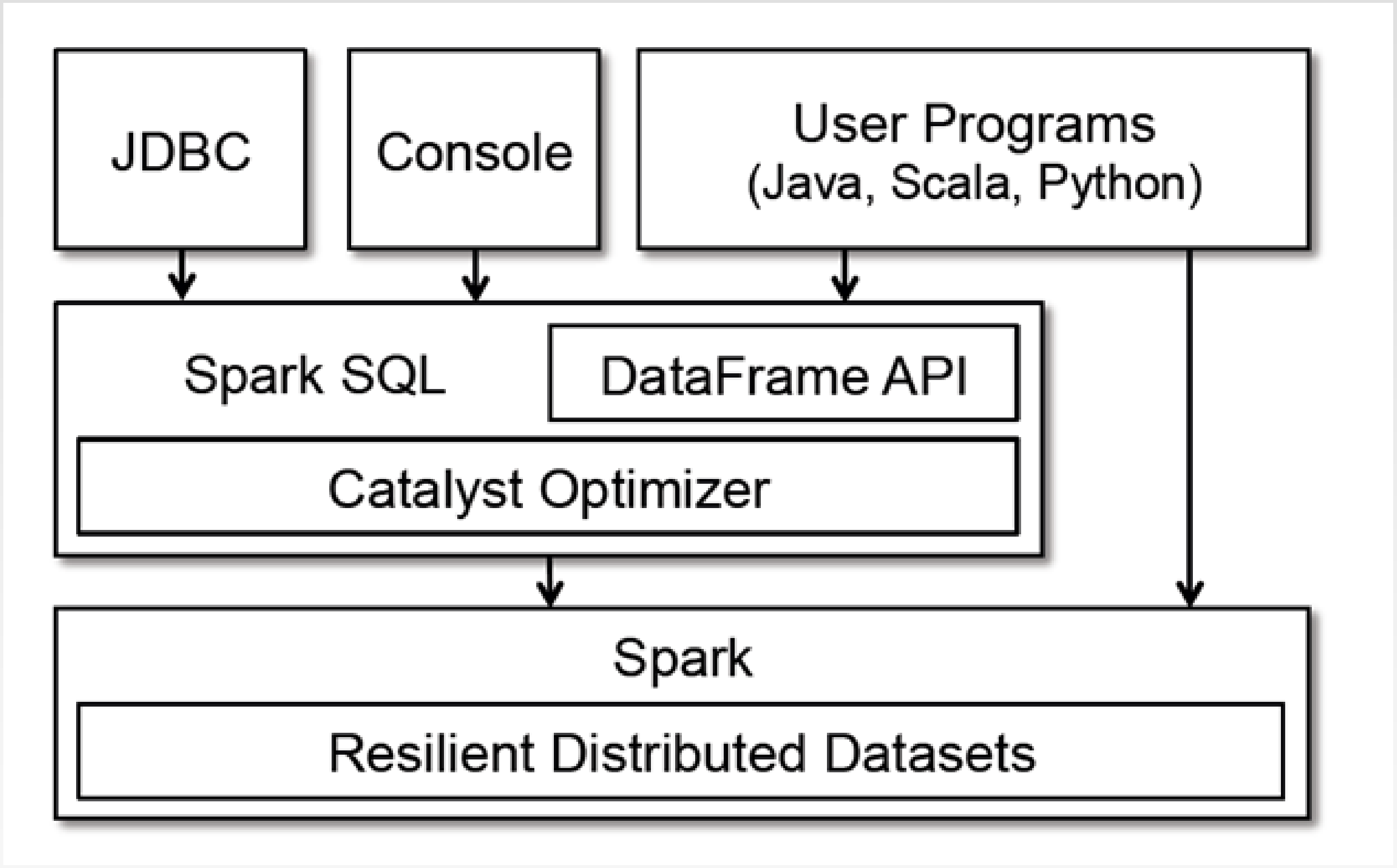
Part 1: Spark SQL

Spark SQL Overview

- Part of the core distribution since Spark 1.0, Transform RDDs using SQL in early versions (April 2014)
- Tightly integrated way to work with structured data (tables with rows/columns)
- Data source integration: Hive, Parquet, JSON, and more

- Spark SQL is **not** about SQL.
  - Aims to Create and Run Spark Programs Faster:

Spark Programming Interface


JDBC
Console
User Programs (Java, Scala, Python)
Spark SQL
DataFrame API
Catalyst Optimizer
Spark
Resilient Distributed Datasets

Starting Point: SparkSession

- The entry point into all functionality in Spark is the SparkSession class.

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

  - SparkSession since Spark 2.0 provides built-in support for Hive features including the ability to write queries using HiveQL, access to Hive UDFs, and the ability to read data from Hive tables

Creating DataFrames

- With a SparkSession, applications can create DataFrames from *an existing RDD*, from *a Hive table*, or from *Spark data sources*.
  - creates a DataFrame based on the content of a JSON file:

```
val df =
spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout

df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

DataFrame Operations

- DataFrames are just Dataset of Rows in Scala and Java API.
  - These operations are also referred as “untyped transformations” in contrast to “typed transformations” come with strongly typed Scala/Java Datasets

```
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+
```

DataFrame Operations

```
// Select everybody, but increment the age by 1
df.select($"name", $"age" + 1).show()
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+

// Select people older than 21
df.filter($"age" > 21).show()
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+

// Count people by age
df.groupBy("age").count().show()
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

Running SQL Queries Programmatically

- The sql function on a SparkSession enables applications to run SQL queries programmatically and returns the result as a DataFrame.

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Global Temporary View

- Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates
- Global temporary view: a temporary view that is shared among all sessions and keep alive until the Spark application terminates
- Global temporary view is tied to a system preserved database global_temp, and we must use the qualified name to refer it, e.g. SELECT * FROM global_temp.view1

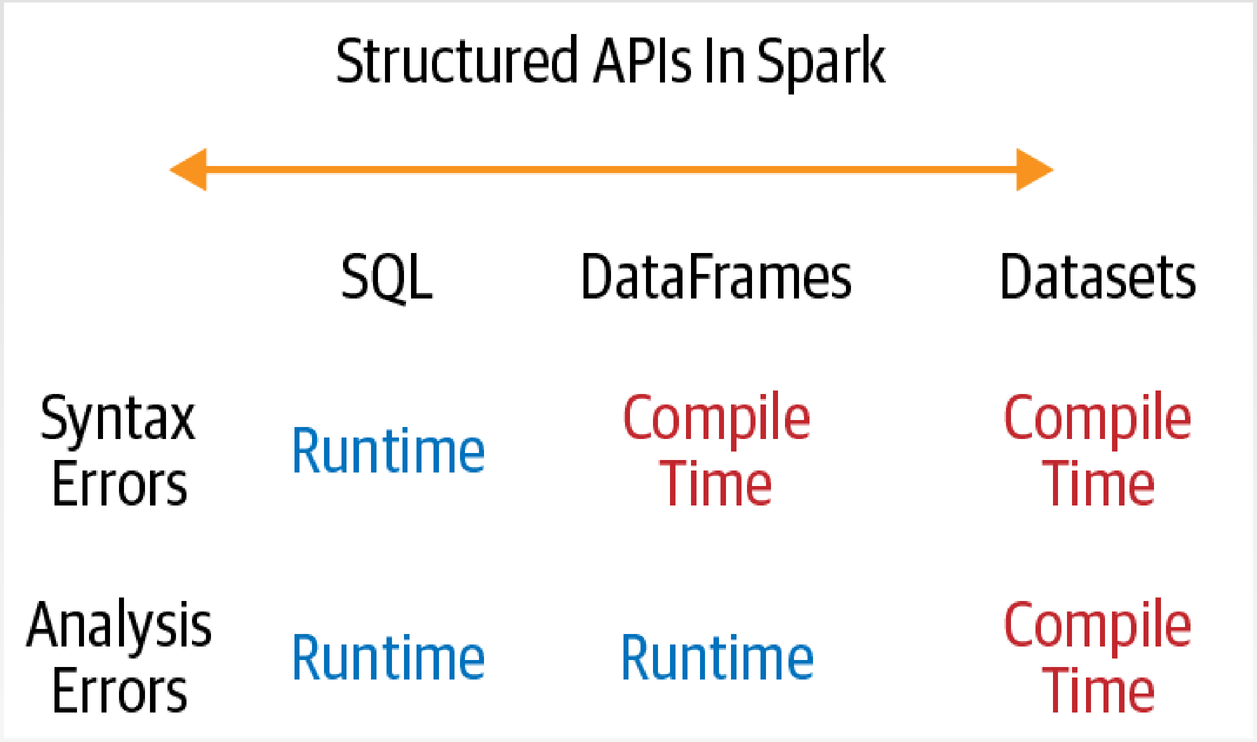
Global Temporary View Example

```
// Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Find full example code at
https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala

Error Detection of Structured APIs

- If you want errors caught during compilation rather than at runtime, choose the appropriate API



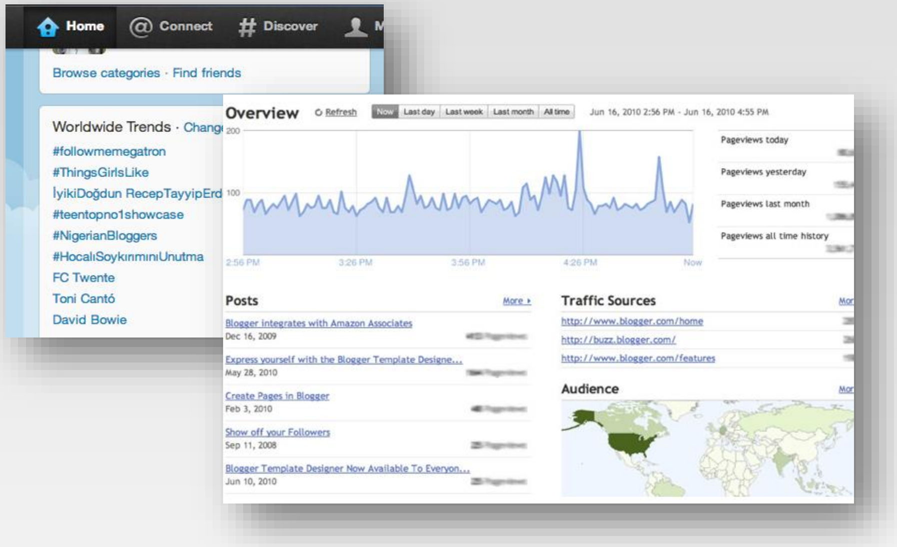
Structured APIs In Spark

	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

Part 2: Spark Streaming

Motivation

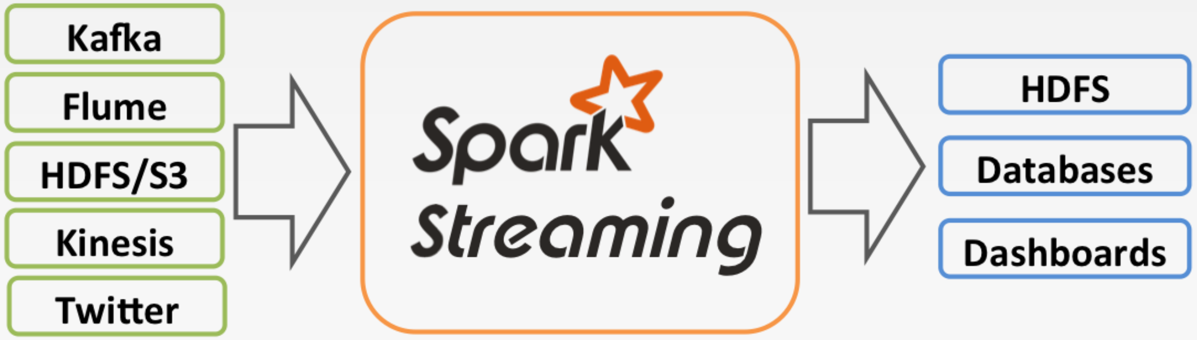
- Many important applications must process large streams of live data and provide results in near-real-time
  - Social network trends
  - Website statistics
  - Ad impressions
  - … …



- Distributed stream processing framework is required to
  - Scale to large clusters (100s of machines)
  - Achieve low latency (few seconds)

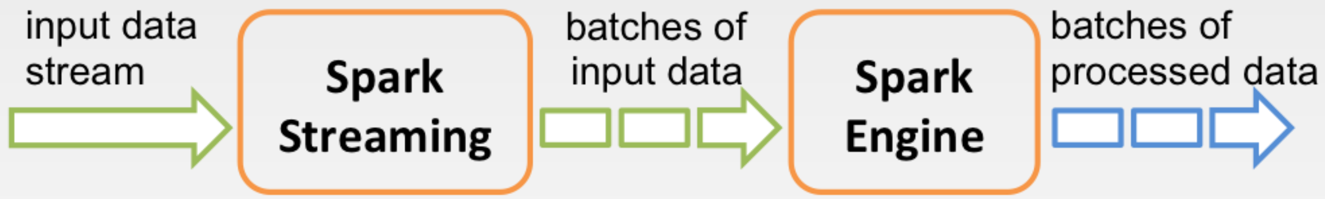
What is Spark Streaming

- Spark Streaming is an extension of the core Spark API that enables **scalable**, **high-throughput**, **fault-tolerant** stream processing of live data streams
- Receive data streams from input sources, process them in a cluster, push out to filesystems, databases, and live dashboards
  - Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets
  - Data can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window
  - Processed data can be pushed out to filesystems, databases, and live dashboards



How does Spark Streaming Work

- Run a streaming computation as a series of very small, deterministic batch jobs
  - Chop up the live stream into batches of X seconds
  - Spark treats each batch of data as RDDs and processes them using RDD operations
  - Finally, the processed results of the RDD operations are returned in batches



Spark Streaming Programming Model

- Spark Streaming provides a high-level abstraction called *discretized stream* (*Dstream*)
  - Represents a continuous stream of data.
  - DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams.
  - Internally, a DStream is represented as a sequence of RDDs.

- DStreams API very similar to RDD API
  - Functional APIs in Scala, Java
  - Create input DStreams from different sources
  - Apply parallel operations

An Example: Streaming WordCount

- Use StreamingContext, rather then SparkContext

```
import org.apache.spark._
import org.apache.spark.streaming._

object NetworkWordCount {
        val conf = new
        SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
        val ssc = new StreamingContext(conf, Seconds(10))

        val lines = ssc.socketTextStream("localhost", 9999)
        val words = lines.flatMap(_.split(" "))
        val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
        wordCounts.print()
        ssc.start()
        ssc.awaitTermination()
}
```

Streaming WordCount

- Linking with Apache Spark
  - The first step is to explicitly import the required spark classes into your Spark program

```
import org.apache.spark._
import org.apache.spark.streaming._
```

- Create a local StreamingContext with two working thread and batch interval of 10 second.

```
val conf = new
SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(10))
```

  - A **StreamingContext** object has to be created which is the main entry point of all Spark Streaming functionality.
  - At least two local threads must be used (two cores)
  - Do the count for each 10 seconds
    - The batch interval must be set based on the latency requirements of your application and available cluster resources

Streaming WordCount

- After a streaming context is defined, you have to do the following:
  - Define the input sources by creating input DStreams.
  - Define the streaming computations by applying transformation and output operations to DStreams.
  - Start receiving data and processing it using streamingContext.start().
  - Wait for the processing to be stopped (manually or due to any error) using streamingContext.awaitTermination().
  - The processing can be manually stopped using streamingContext.stop().

Streaming WordCount

Using this context, we can create a DStream that represents streaming data from a TCP source, specified as hostname (e.g. localhost) and port (e.g. 9999).

```
val lines = ssc.socketTextStream("localhost", 9999)
```

- This lines DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text.

- Split the lines by space characters into words and do the count

```
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
```

- No real processing has started yet now. To start the processing after all the transformations have been setup, we finally call

```
ssc.start()
ssc.awaitTermination()
```

- The complete code can be found in the Spark Streaming example NetworkWordCount.

Linking the Application

- Add the following dependency to your SBT configuration:
  - libraryDependencies += "org.apache.spark" %% "spark-streaming_2.12" % “3.1.2"

- For data sources like Kafka, Flume, and Kinesis that are not present in the Spark Streaming core API, you will have to add the corresponding artifact spark-streaming-xyz_2.12 to the dependencies

Kafka	spark-streaming-kafka-0-10_2.12
Kinesis	spark-streaming-kinesis-asl_2.12 [Amazon Software License]

Run Streaming WordCount

- First need to run Netcat (a small utility found in most Unix-like systems) as a data server by using:

```
$ nc -lk 9999
```

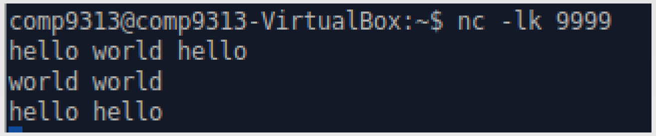
- sbt configuration file:

```
name := “Network WordCount"
version := "1.0"
scalaVersion := "2.12.10"
libraryDependencies += "org.apache.spark" %% "spark-streaming" % “3.1.2"
```

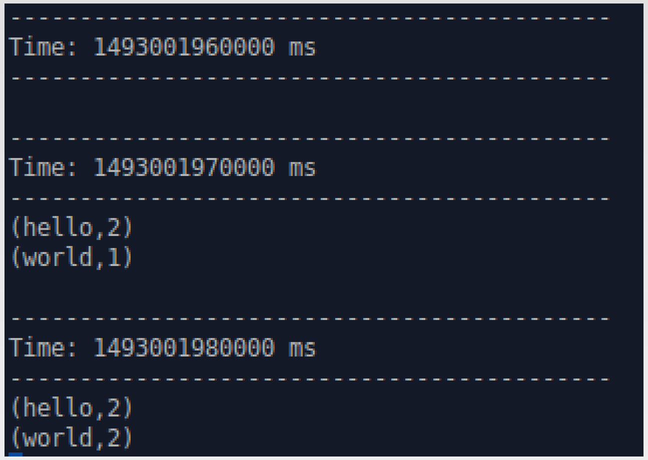
- Then, in a different terminal, you can start the example by using

```
$ spark-submit --class NetworkWordCount ~/sparkapp/target/scala-2.12/network-wordcount_2.12-1.0.jar
```

Results of Streaming WordCount



```
comp9313@comp9313-VirtualBox:~$ nc -lk 9999
hello world hello
world world
hello hello
```



```
-------------------------------------------
Time: 1493001960000 ms
-------------------------------------------

-------------------------------------------
Time: 1493001970000 ms
-------------------------------------------
(hello,2)
(world,1)

-------------------------------------------
Time: 1493001980000 ms
-------------------------------------------
(hello,2)
(world,2)
```

- The first 10 seconds, receives no data
- The next 10 seconds, receives one line
- The last 10 seconds, receives two lines

Get Streaming Data From Files

- Spark streaming can also get streaming data from a specified file folder
- It can only ingest files that have been moved to the directory
- Create a local StreamingContext with two working thread and batch interval of 10 second.

```
val conf = new
SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(10))
```

- Using this context, we can create a DStream that represents streaming data from files in a folder

```
val lines = ssc.textFileStream(“file:///home/comp9313/logs”)
```

- Next do similarly as in NetworkWordCount

Get Streaming Data From Files

- The first 10 seconds, receives no data
- The next 10 seconds, do:

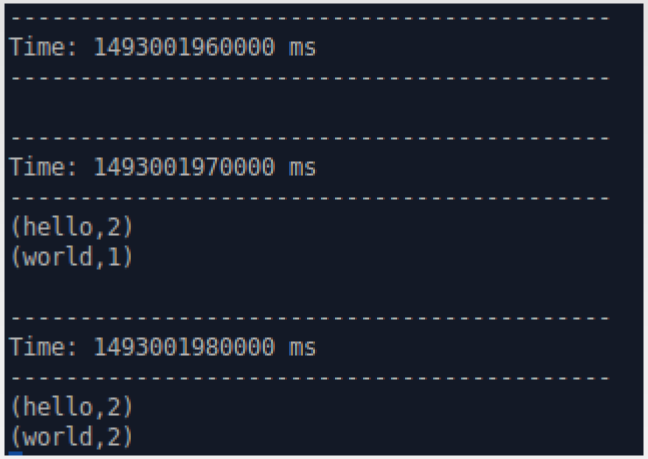
  echo “hello hello world” > log1

  mv log1 ~/logs
- The next 10 seconds, do:

  echo “hello world” > log2

  echo “hello world” > log3

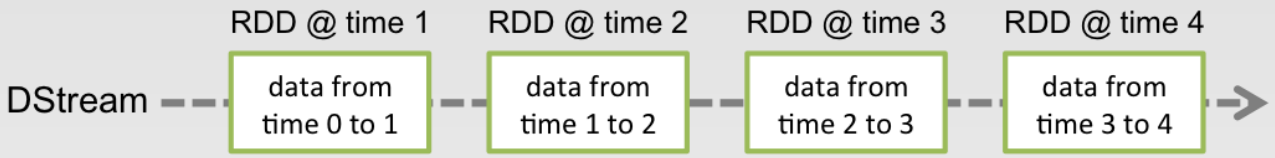
  mv log2 log3 ~/logs



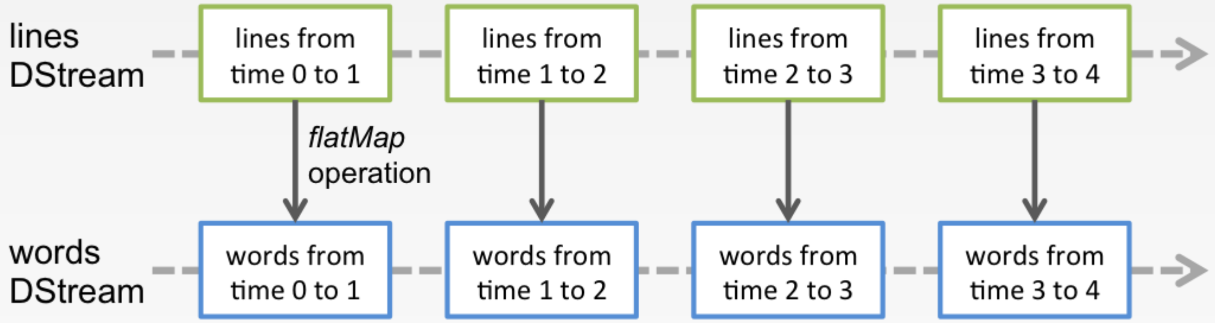
```
-------------------------------------------
Time: 1493001960000 ms
-------------------------------------------

-------------------------------------------
Time: 1493001970000 ms
-------------------------------------------
(hello,2)
(world,1)

-------------------------------------------
Time: 1493001980000 ms
-------------------------------------------
(hello,2)
(world,2)
```

Discretized Streams (DStreams)

- A DStream is represented by a continuous series of RDDs
  - Each RDD in a DStream contains data from a certain interval, as shown in the following figure.



- Any operation applied on a DStream translates to operations on the underlying RDDs.
  - in the earlier example of converting a stream of **lines** to **words**, the flatMap operation is applied on each RDD in the **lines** DStream to generate the RDDs of the **words** DStream



Input DStreams and Receivers

- Input DStreams are DStreams representing the stream of input data received from streaming sources.
  - E.g., **lines** was an input DStream as it represented the stream of data received from the netcat server
- Every input DStream is associated with a **Receiver** object which receives the data from a source and stores it in Spark's memory for processing
- Spark Streaming provides two categories of built-in streaming sources.
  - *Basic sources*: Sources directly available in the StreamingContext API. Examples: file systems, and socket connections.
  - *Advanced sources*: Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. These require linking against extra dependencies

Input DStreams and Receivers

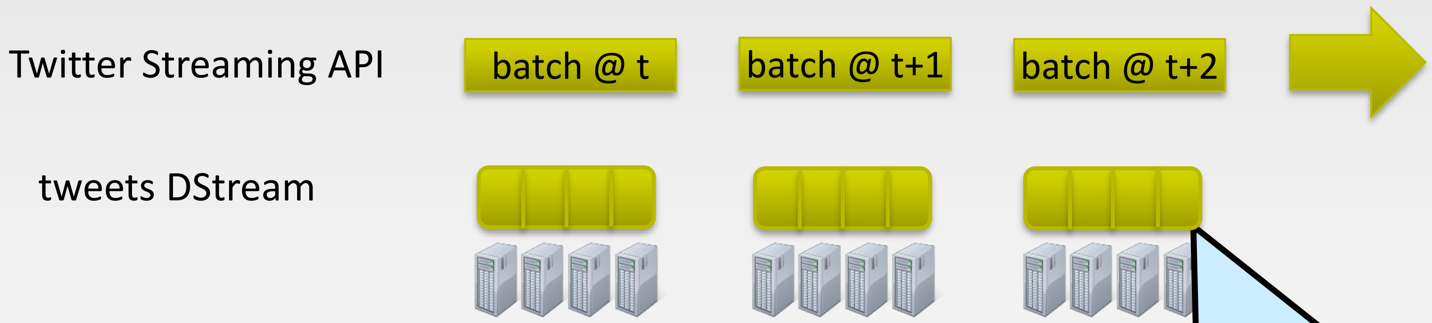
- When running a Spark Streaming program locally, do not use “local” or “local[1]” as the master URL
  - If you are using an input DStream based on a receiver (e.g., sockets), then the single thread will be used to run the receiver, leaving no thread for processing the received data
  - When running locally, always use “local[*n*]” as the master URL, where *n* > number of receivers to run

- The number of cores allocated to the Spark Streaming application must be more than the number of receivers. Otherwise the system will only receive data, but not be able to process it

Example – Get hashtags from Twitter

```
val ssc = new StreamingContext(conf, Seconds(10))
val tweets :DStream[Status] = TwitterUtils.createStream(ssc, None)
```

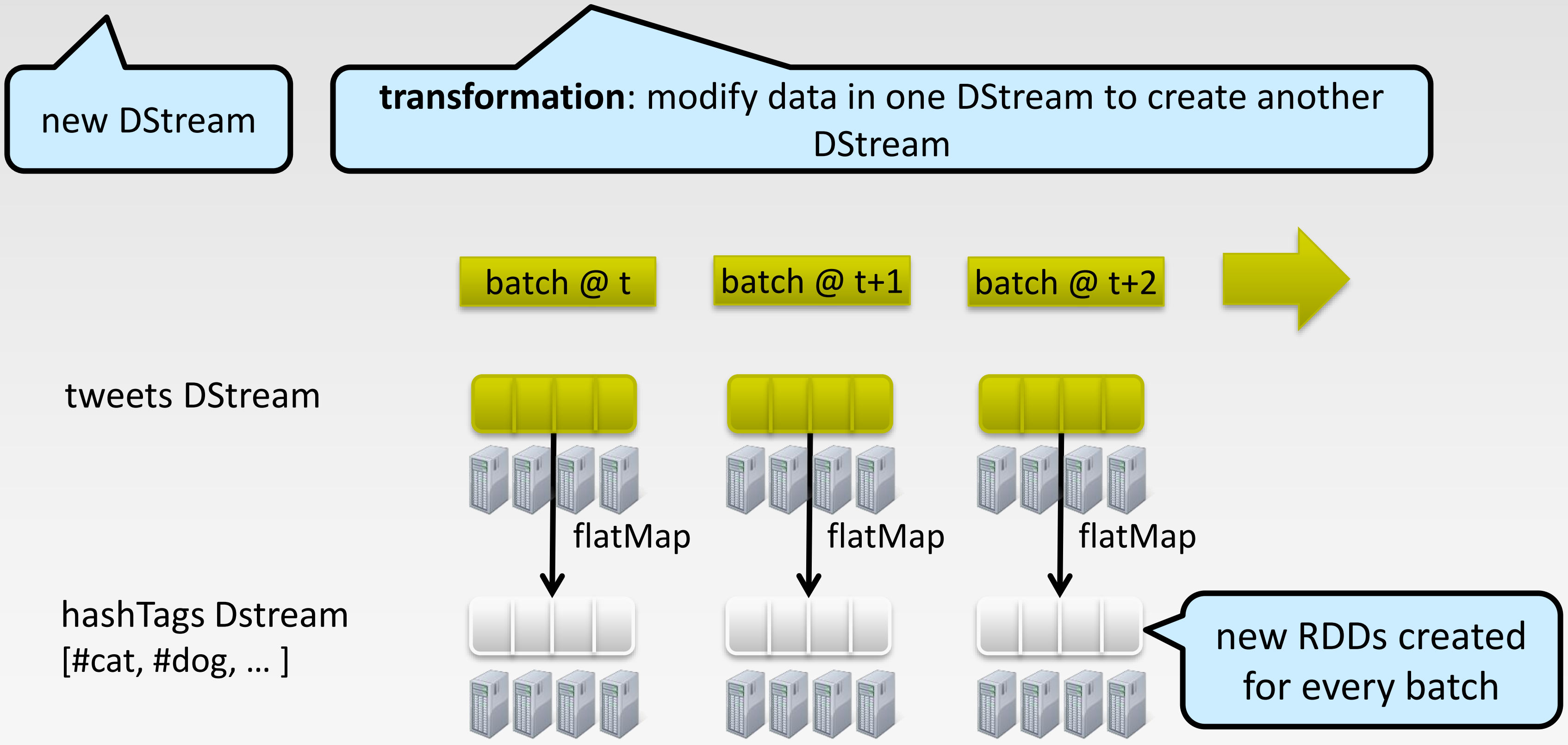
DStream: a sequence of RDDs representing a stream of data



stored in memory as an RDD (immutable, distributed)

Example – Get hashtags from Twitter

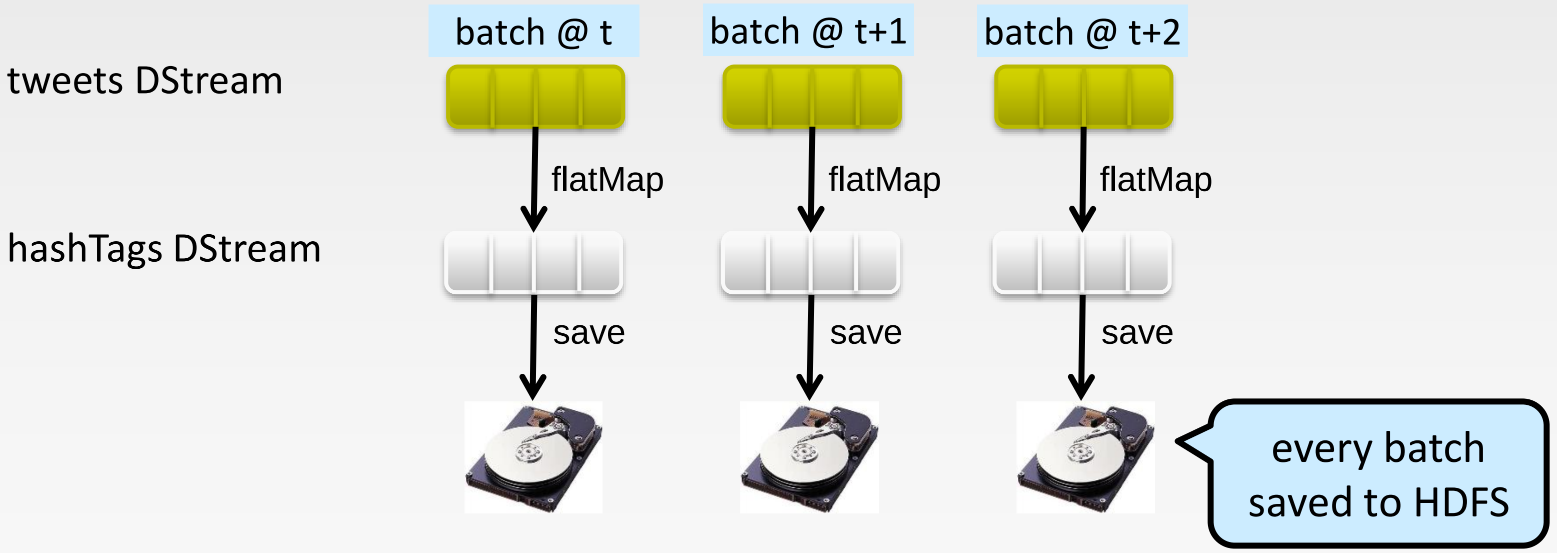
```
val hashTags = tweets.flatMap (status => getTags(status))
```



Example – Get hashtags from Twitter

```
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

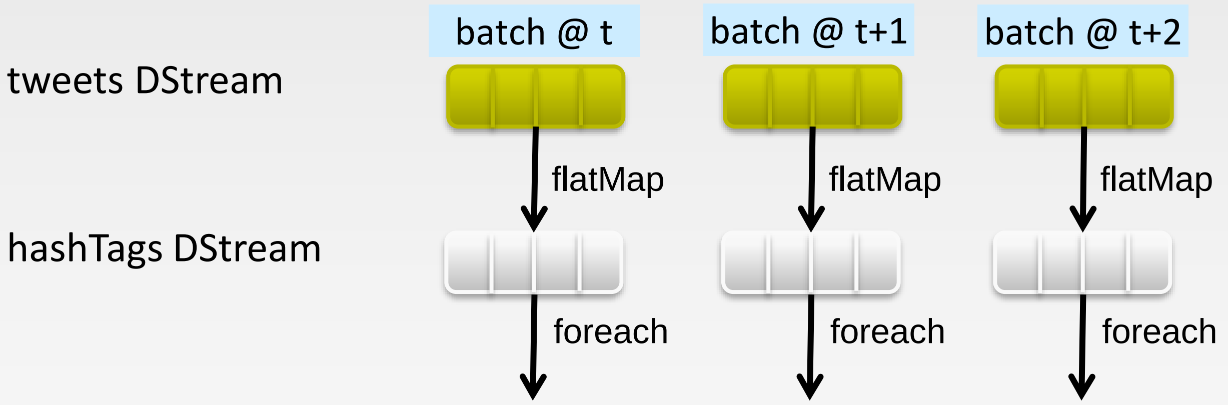
output operation: to push data to external storage



Example – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

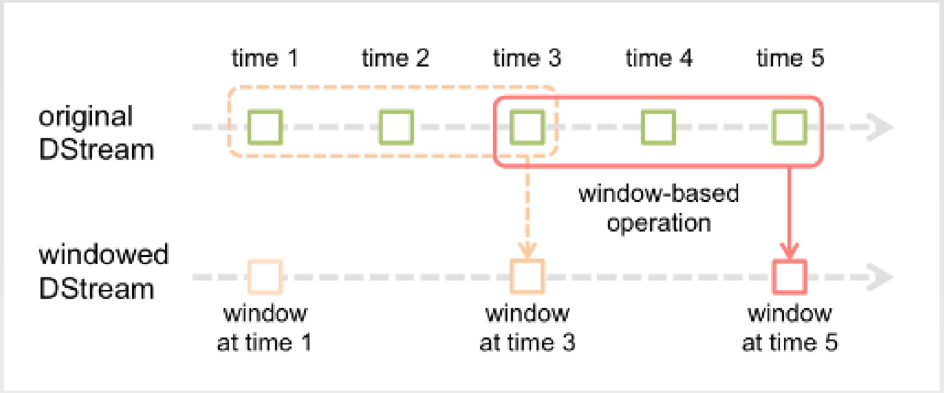
foreach: do whatever you want with the processed data



Write to database, update analytics
UI, do whatever you want

Window Operations

- Spark Streaming also provides *windowed computations*, which allow you to apply transformations over a sliding window of data



- Every time the window *slides* over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream.
  - E.g., the operation is applied over the last 3 time units of data, and slides by 2 time units
  - Any window operation needs to specify two parameters
    - *window length* - The duration of the window (3 in the figure).
    - *sliding interval* - The interval at which the window operation is performed (2 in the figure).

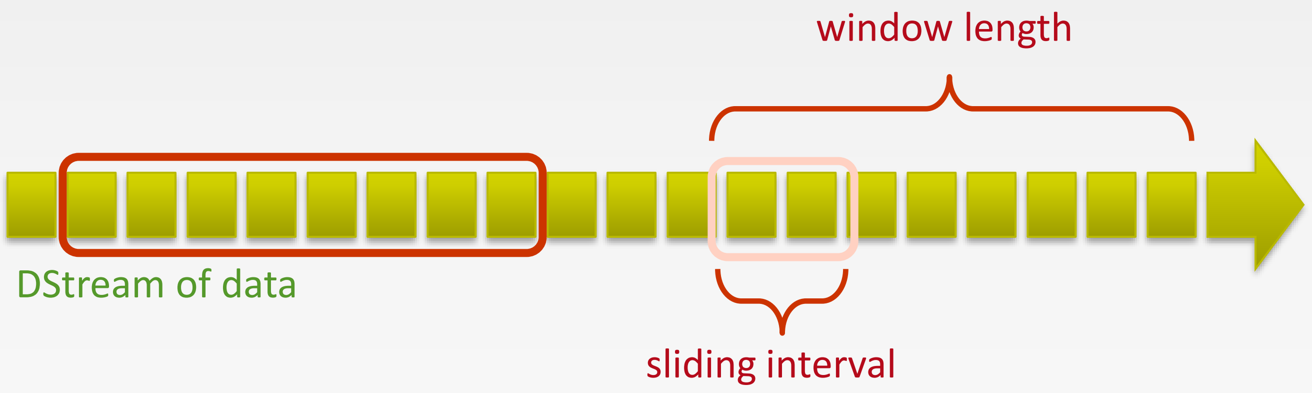
Window-based Transformations

```
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(1), Seconds(5)).countByValue()
```

sliding window operation

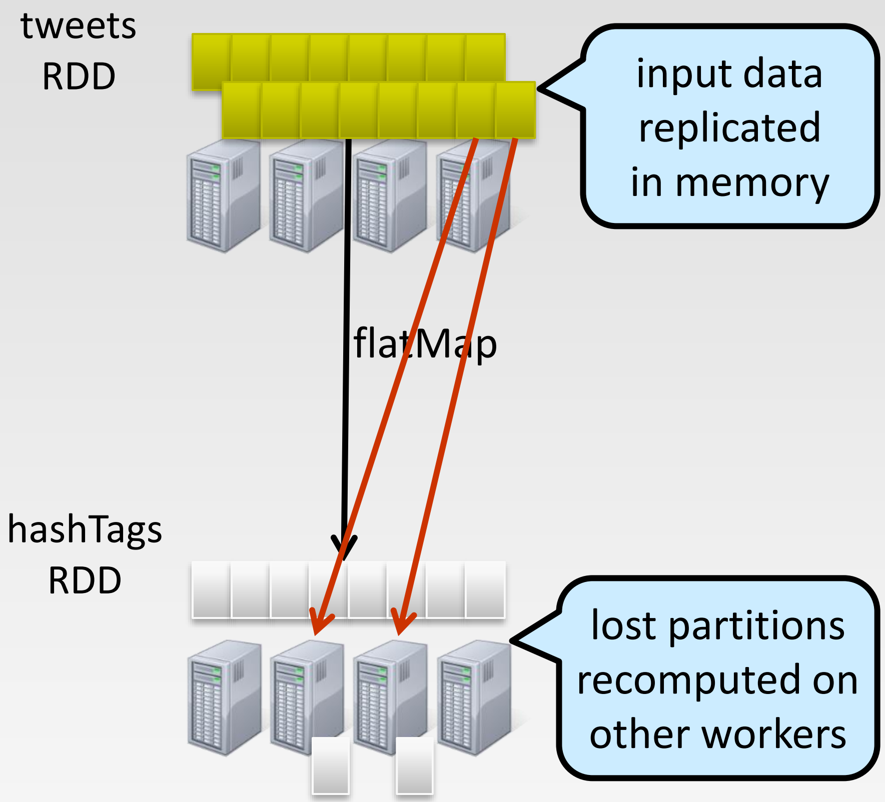
window length

sliding interval



Fault-tolerance: Worker

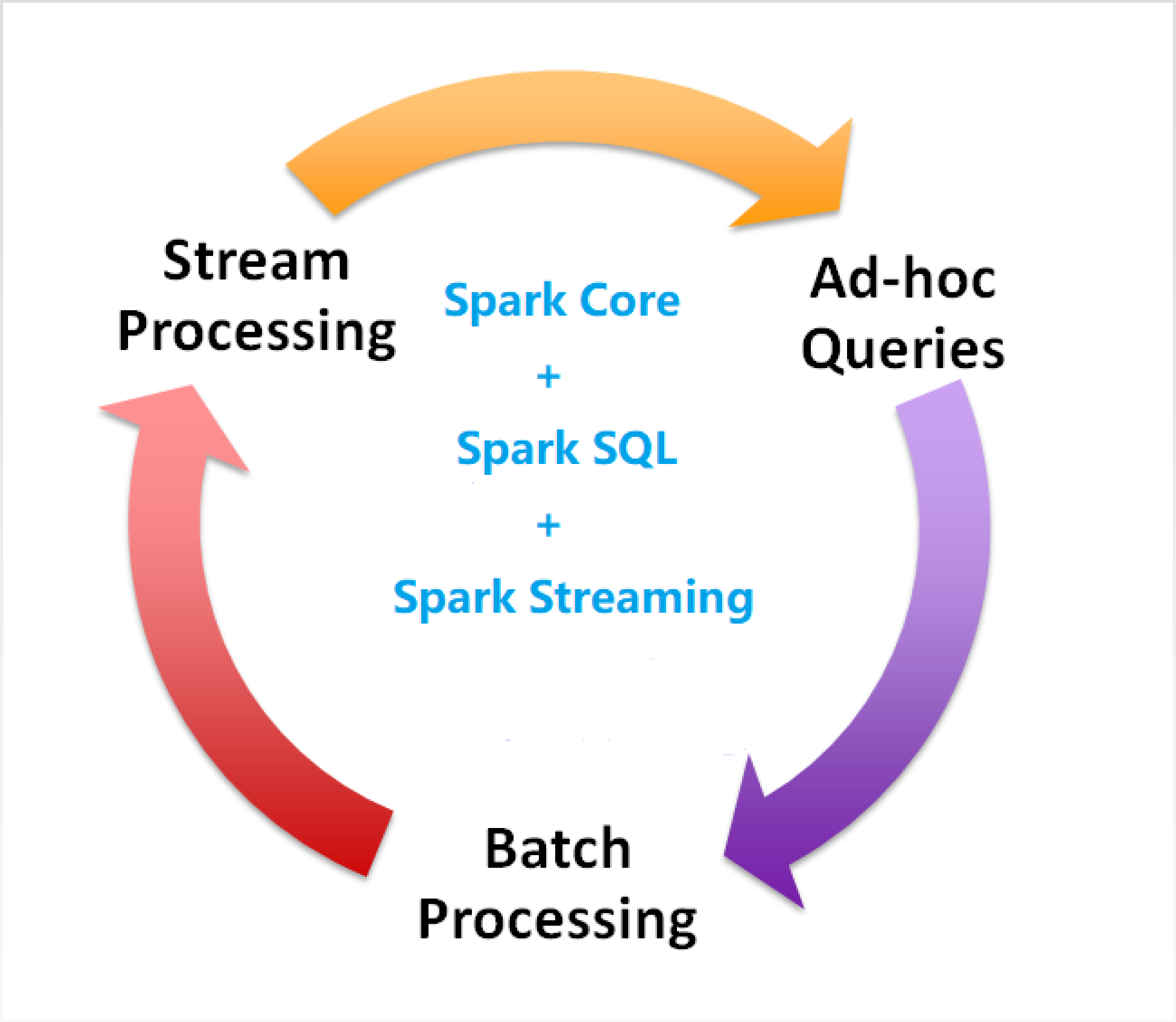
- RDDs remember the operations that created them
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data
- All transformed data is fault-tolerant, and exactly-once transformations


tweets RDD
input data replicated in memory
flatMap
hashTags RDD
lost partitions recomputed on other workers

Fault-tolerance: Master

- Master saves the state of the DStreams to a checkpoint file
  - Checkpoint file saved to HDFS periodically
- If master fails, it can be restarted using the checkpoint file
- More information in the Spark Streaming guide
- Automated master fault recovery coming soon

Vision - one stack to rule them all



References

- http://spark.apache.org/docs/latest/index.html
- Spark SQL guide: http://spark.apache.org/docs/latest/sql-programming-guide.html
- Spark Streaming guide: http://spark.apache.org/docs/latest/streaming-programming-guide.html
- Learning Spark. 2nd edition

End of Chapter 5.2