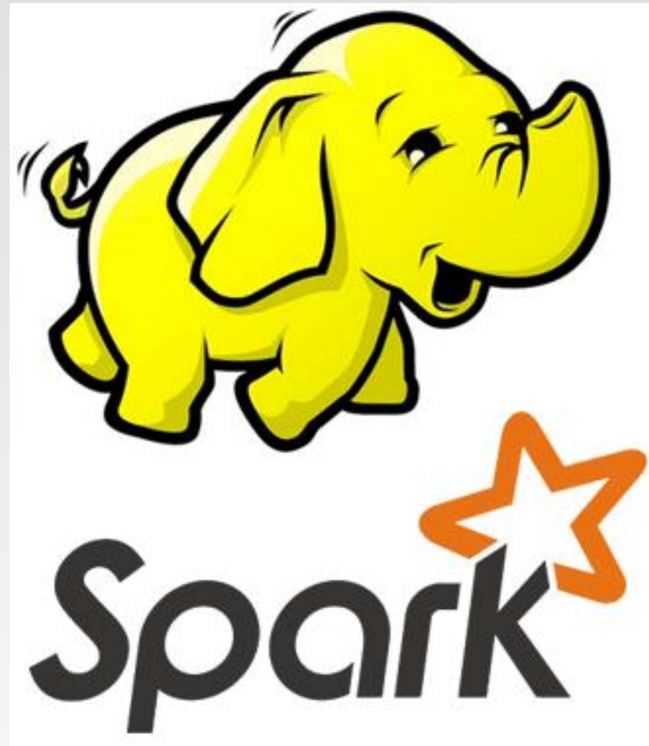
COMP9313: Big Data Management



Lecturer: Xin Cao

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

Chapter 5.1: Spark III

Part 1: Spark Structured APIs

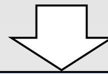
A Brief Review of RDD

- 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.

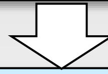
Compute Average Values for Each Key

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

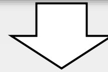
```
pairs = sc.parallelize([(1, 2), (3, 1), (3, 6), (4,2)])
```



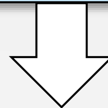
```
pairs1 = pairs.mapValues(lambda x: (x, 1))
```



```
pairs2 = pairs1.reduceByKey(lambda x, y: (x[0] + y[0], x[1]+y[1]))
```



```
avg = pairs2.mapValues(lambda x: x[0]/x[1]).collect()
```



```
[(1, 2.0), (3, 3.5), (4, 2.0)]
```

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

```
pairs1 = pairs.mapValues(lambda x: (x, 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:
  - Scala:

```
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|
+------+--------+
```

  - Python:

```
from pyspark.sql.functions import avg

data_df = spark.createDataFrame([("Brooke",
20), ("Denny", 31), ("Jules", 30),("TD", 35),
("Brooke", 25)], schema = 'name string, age
int')

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

Spark's Structured APIs

- 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

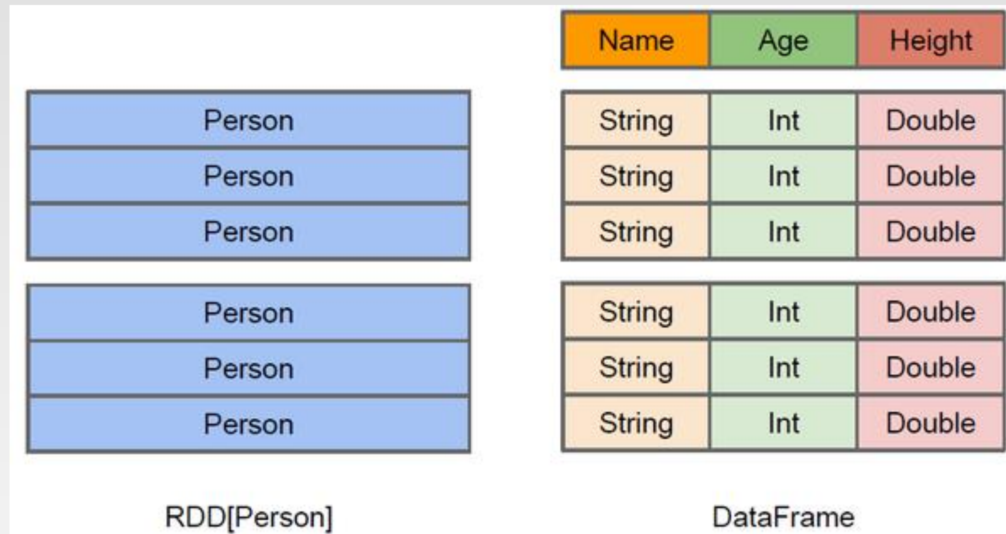
- Spark DataFrames are like distributed in-memory tables with named columns and schemas, where each column has a specific data type.
- When data is visualized as a structured table, it's not only easy to digest but also easy to work with

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

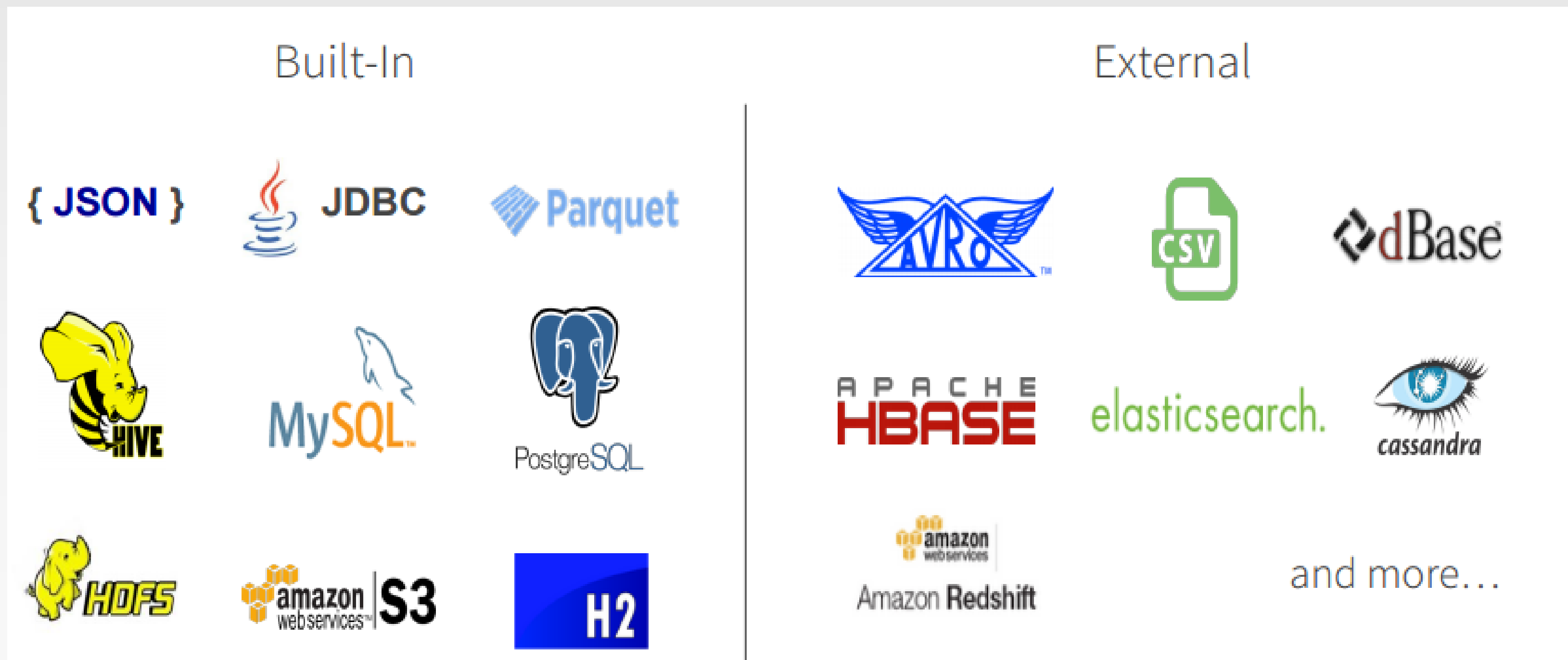
- DataFrame more like a traditional database of two-dimensional form, in addition to data, but also to grasp the structural information of the data, that is, schema



  - 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 (Scala)

- You can create a DataFrame from a Scala object

```
// 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 a list
val dataDF = spark.createDataFrame(data)
```

- You can also convert an RDD into a DataFrame

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

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

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

Create DataFrames (Scala)

- 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 (Scala)

- 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

Structured and Complex Data Types (Scala)

- 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 (Scala)

- 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 (Scala)

- 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)
```

Spark's Basic Data Types (Python)

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

Data type	Value assigned in Python	API to instantiate
ByteType	int	DataTypes.ByteType
ShortType	int	DataTypes.ShortType
IntegerType	int	DataTypes.IntegerType
LongType	int	DataTypes.LongType
FloatType	float	DataTypes.FloatType
DoubleType	float	DataTypes.DoubleType
StringType	str	DataTypes.StringType
BooleanType	bool	DataTypes.BooleanType
DecimalType	decimal.Decimal	DecimalType

Structured and Complex Data Types (Python)

- 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 Python	API to instantiate
BinaryType	bytearray	BinaryType()
Timestamp Type	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	List, tuple, or array	ArrayType(dataType, [nullable])
MapType	dict	MapType(keyType, valueType, [nullable])
StructType	List or tuple	StructType([fields])
StructField	A value type corresponding to the type of this field	StructField(name, dataType, [nullable])

Create DataFrames with Schema (Python)

- A PySpark DataFrame can be created via pyspark.sql.SparkSession.createDataFrame typically by passing a list of lists, tuples, dictionaries and pyspark.sql.Rows, a pandas DataFrame and an RDD consisting of such a list.
- You can Create a PySpark DataFrame with an explicit schema.

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

// Create DataFrame' from a list
dataDF = spark.createDataFrame(data, schema = "name string, age
int")
```

- You can create a PySpark DataFrame from an RDD consisting of a list of tuples.

```
// Given a pair RDD including name and age
data = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules",
30),("TD", 35), ("Brooke", 25)])

// Create DataFrame' from an RDD
dataDF = spark.createDataFrame(data, schema=["name","age"])
```

Create DataFrames with Schema (Python)

- You can also create a PySpark DataFrame from a list of rows

```
from pyspark.sql import Row

// Given a list of rows containing names and ages
data = [Row(name = "Brooke", age = 20), Row(name = "Denny", age =
31), Row(name = "Jules", age = 30), Row(name = "TD", age = 35),
Row(name = "Brooke", age = 25)]

// Create DataFrame' from a list of Rows
dataDF = spark.createDataFrame(data)
```

- You can print out the schema of a DataFrame

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

- The top rows of a DataFrame can be displayed using DataFrame.show().
- All the DataFrame APIs are listed here: https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/dataframe.html

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)
```

```
>>> dataDF.columns
['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
```

```
>>> dataDF.name
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|
+---------+
```

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

Columns (Python)

- withColumn() returns a new DataFrame by adding a column or replacing the existing column that has the same name

```
>>> from pyspark.sql.functions import upper
>>> dataDF.withColumn("name_upper", upper(dataDF.name)).show()
+------+---+----------+
|  name|age|name_upper|
+------+---+----------+
|Brooke| 20|    BROOKE|
| Denny| 31|     DENNY|
| Jules| 30|     JULES|
|    TD| 35|        TD|
|Brooke| 25|    BROOKE|
+------+---+----------+
```

```
>>> dataDF.withColumn("name", upper(dataDF.name)).show()
+------+---+
|  name|age|
+------+---+
|BROOKE| 20|
| DENNY| 31|
| JULES| 30|
|    TD| 35|
|BROOKE| 25|
+------+---+
```

- All the Columns APIs are listed here: https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/column.html

Rows (Scala)

- 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|
+-------------+-----+
```

Grouping Data

- DataFrame also provides a way of handling grouped data by using the common approach, split-apply-combine strategy. It groups the data by a certain condition applies a function to each group and then combines them back to the DataFrame.

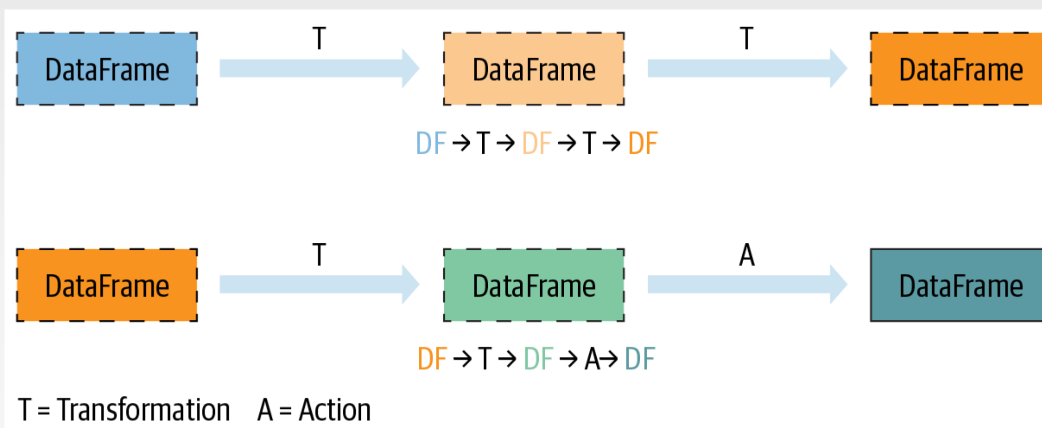
```
>>> df.show()
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

  - Grouping and then applying the avg() function to the resulting groups

```
>>> df.groupby('color').avg().show()
+-----+-------+-------+
|color|avg(v1)|avg(v2)|
+-----+-------+-------+
|  red|    4.8|   48.0|
|black|    6.0|   60.0|
| blue|    3.0|   30.0|
+-----+-------+-------+
```

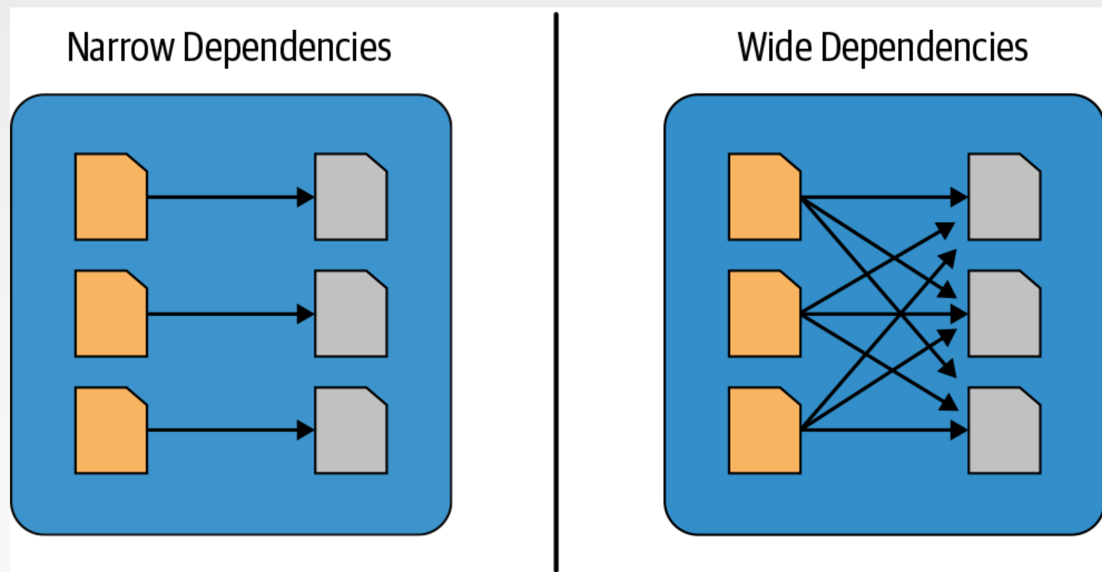
Transformations, Actions, and Lazy Evaluation

- 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 (Scala)

```
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")
```

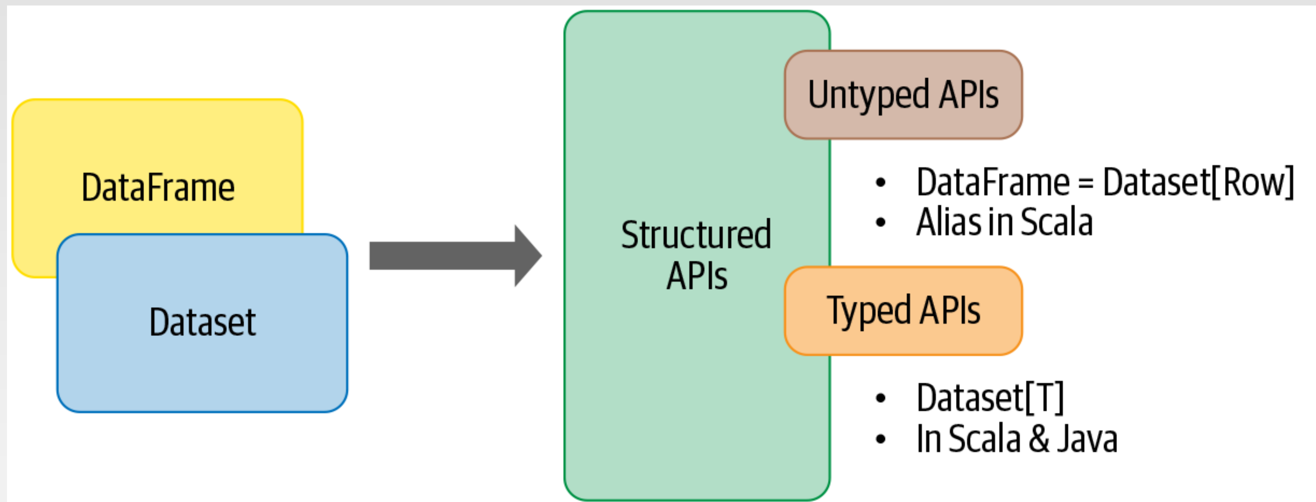
WordCount using DataFrame (Python)

```
fileRDD =
spark.sparkContext.textFile("file:///home/comp9313/inputText")

wordsDF = fileRDD.flatMap(lambda x: x.split(" ")).map(lambda x:
(x, )).toDF("word string")



#or
# from pyspark.sql.types import StringType
# wordsDF = spark.createDataFrame(fileRDD.flatMap(lambda x:
x.split(" ")), StringType()).withColumnRenamed("value", "word")

countDF = wordsDF.groupBy("word").count()

countDF.show()
```

```
>>> countDF = wordsDF.groupBy(wordsDF.word).count()
>>> countDF.show()
+------+-----+
|  word|count|
+------+-----+
| World|    1|
| Hello|    2|
|   Bye|    1|
|Hadoop|    2|
+------+-----+
```

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]
- The Datasets are strong typed, and so the typed errors can be detected during compile-time

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 and https://sparkbyexamples.com/pyspark-tutorial/

Standalone Appliation (Scala)

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

object WordCount {
  def main(args: Array[String]) {
    val inputFile = args(0)
    val outputFolder = args(1)
    val spark =
SparkSession.builder.appName("WordCount").getOrCreate()

    import spark.implicits._
    val fileRDD = spark.sparkContext.textFile(inputFile)

    val wordsDF = fileRDD.flatMap(_.split(" ")).toDF

    val countDF = wordsDF.groupBy("value").count()

    countDF.write.format("csv").save(outputFolder)
    spark.stop()
  }
}
```

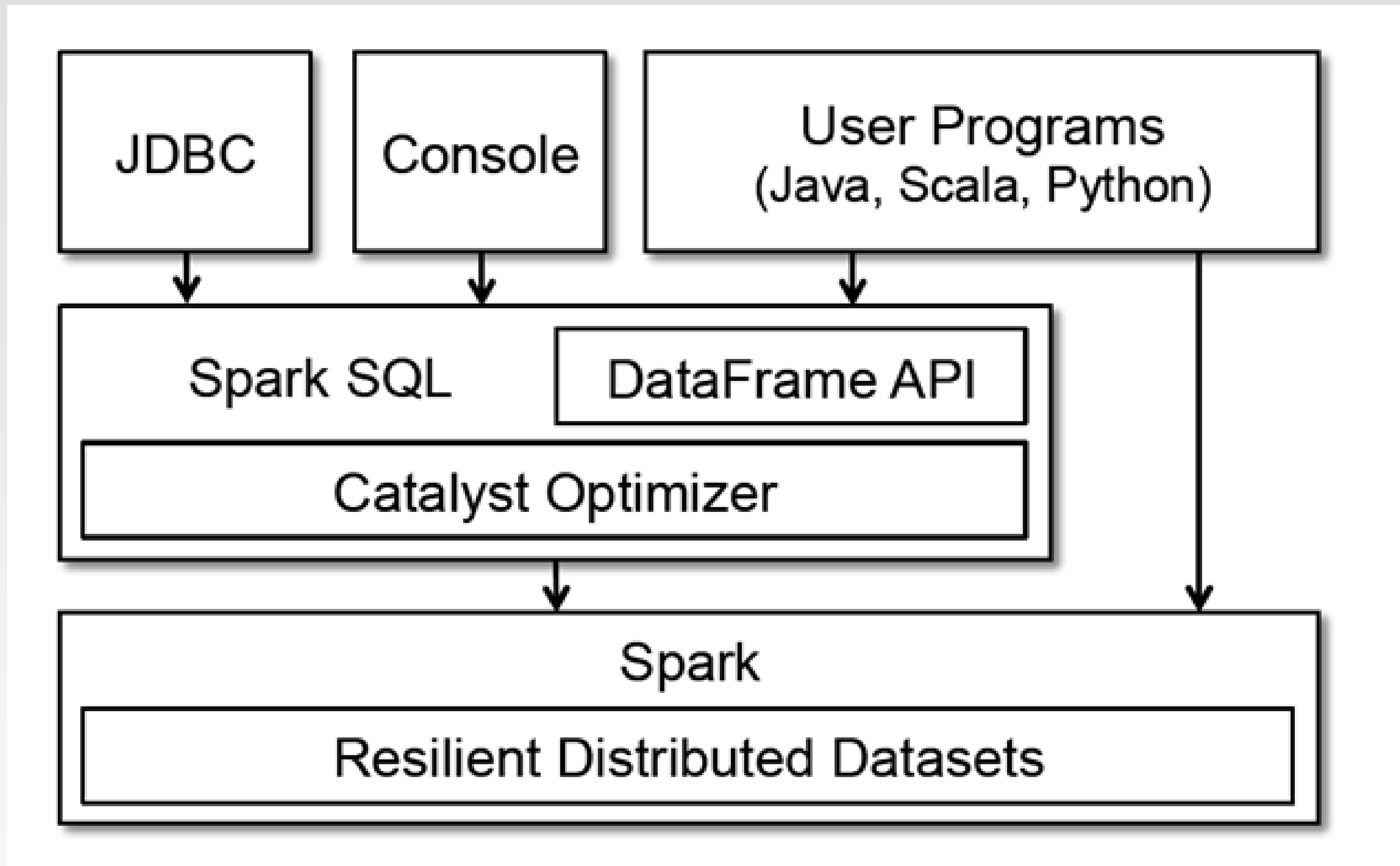
Part 2: Spark SQL

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



Starting Point: SparkSession

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

```
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._
```

  - Python

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local").appName("Spark SQL
basic example").getOrCreate()
```

  - 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 from JSON

- With a SparkSession, applications can create DataFrames 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|
// +----+-------+
```

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

Spark SQL Built-in Functions

- Spark SQL provides several built-in standard functions org.apache.spark.sql.functions to work with DataFrame/Dataset and SQL queries. All these Spark SQL Functions return org.apache.spark.sql.Column type.
  - String Functions
  - Date & Time Functions
  - Collection Functions
  - Math Functions
  - Aggregate Functions
  - Window Functions
  - You can check the examples of these functions at: https://spark.apache.org/docs/latest/api/sql/index.html
- In order to use these SQL Standard Functions, you need to import below packing into your application.

```
from pyspark.sql.functions import *
```

WordCount using Spark SQL (Python)

```
fileDF = spark.read.text("file:///home/comp9313/inputText")

from pyspark.sql.functions import *
wordsDF = fileDF.select(explode(split(fileDF.value, '
')).alias("word"))
```

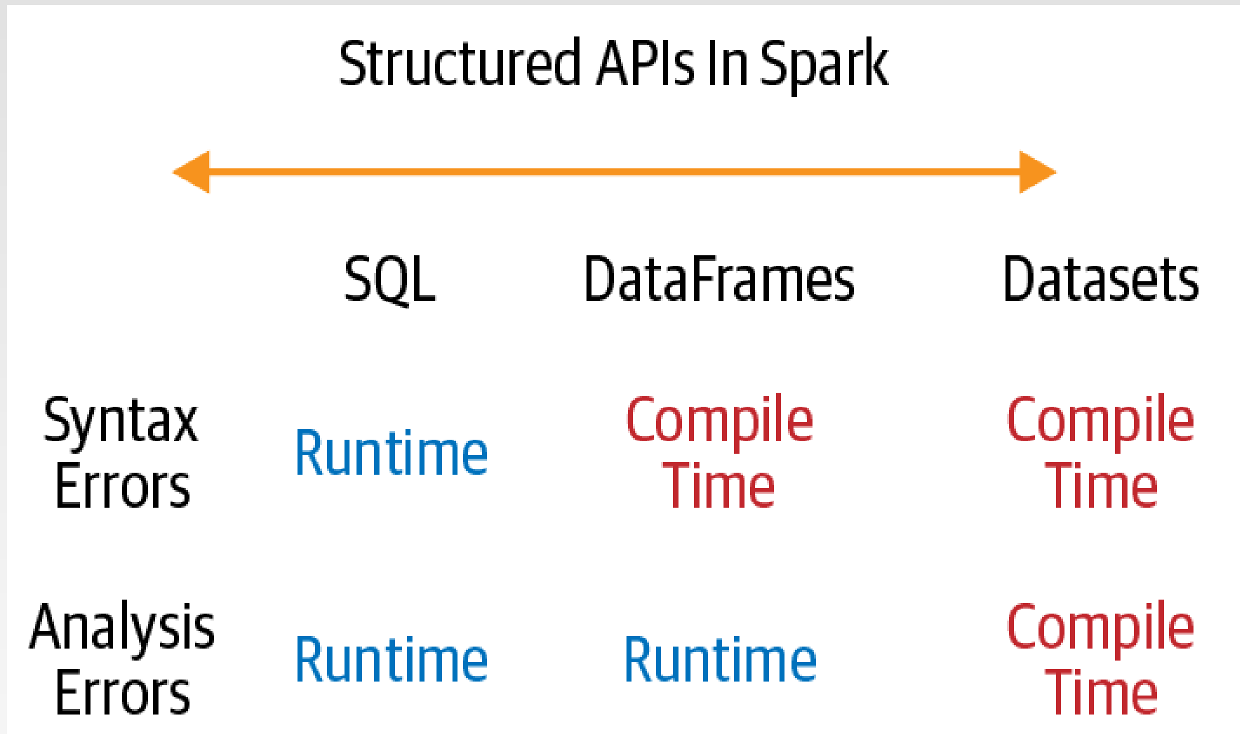
```
>>> wordsDF.show()
+------+
|  word|
+------+
| Hello|
| World|
| Hello|
|Hadoop|
|   Bye|
|Hadoop|
+------+
```

```
countDF = wordsDF.groupBy(wordsDF.word).count()
```

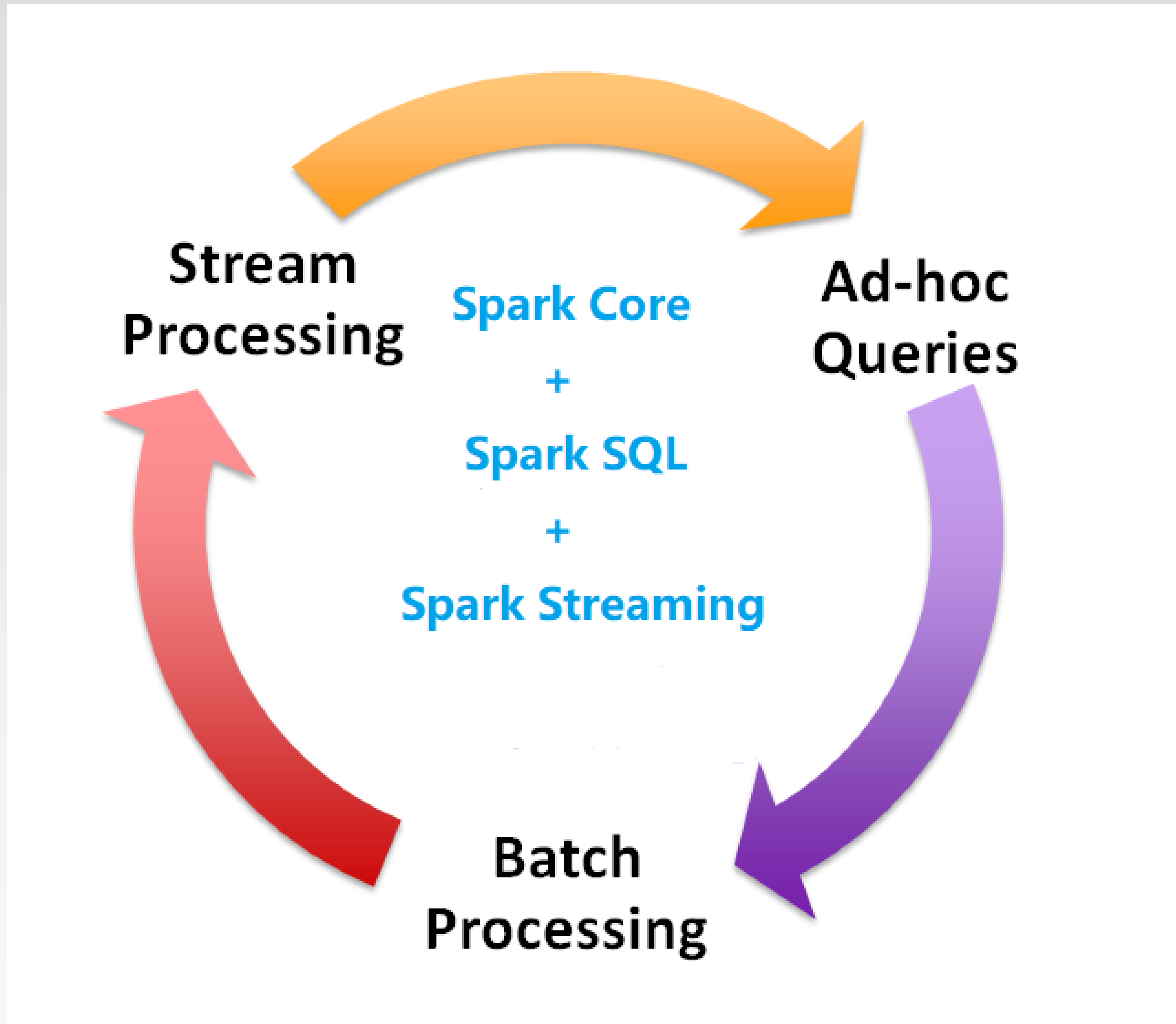
```
>>> countDF = wordsDF.groupBy(wordsDF.word).count()
>>> countDF.show()
+------+-----+
|  word|count|
+------+-----+
| World|    1|
| Hello|    2|
|   Bye|    1|
|Hadoop|    2|
+------+-----+
```

Error Detection of Structured APIs

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



Vision - one stack to rule them all



Part 3: DataFrame and Spark SQL Practices

RDD

- Problem: Given a collection of documents, compute the average length of words starting with each letter.

```
textFile = sc.textFile(inputFile)
words = textFile.flatMap(lambda line: line.split(" ")).map(lambda x: x.lower())

counts = words.filter(lambda x: len(x) >=1 and x[0]<='z' and x[0]>='a').map(lambda x:
(x[0], (len(x), 1)))

avgLen = counts.reduceByKey(lambda a, b: (a[0]+b[0], a[1]+b[1])).map(lambda x:
(x[0], x[1][0]/x[1][1]))

avgLen.foreach(lambda x: print(x[0], x[1]))
```

DataFrame

- Problem: Given a collection of documents, compute the average length of words starting with each letter.

```
textFile = spark.sparkContext.textFile(inputFile)

wordsDF = textFile.flatMap(lambda x: x.split(" ")).map(lambda x: (x, )).toDF("word
string").withColumn("word", lower(col("word")))

wordsDF = wordsDF.filter(length(col("word")) >=1).filter((col("word").substr(0,1)<= 'z')
& (col("word").substr(0,1)>='a'))

pairDF = wordsDF.select(wordsDF.word.substr(0, 1),
length(wordsDF.word)).toDF("letter", "length")

countsDF = pairDF.groupBy("letter").agg(count("letter").alias("totalCount"),
sum("length").alias("totalLength"))

avgDF = countsDF.withColumn("ratio",
countsDF.totalLength/countsDF.totalCount).select("letter", "ratio").orderBy("letter")

avgDF.write.format("csv").save(outputFolder)
```

Spark SQL

- Problem: Given a collection of documents, compute the average length of words starting with each letter.

```
fileDF = spark.read.text(inputFile)

fileDF.selectExpr("explode(split(value, ' ')) as
word").createOrReplaceTempView("words")

spark.sql("select * from words where length(word)>=1 and substr(word, 0, 1)>='a' and
substr(word, 0, 1)<='z' ").createOrReplaceTempView("filteredwords")

spark.sql("select substr(word, 0, 1) as letter, length(word) as length from
filteredwords").createOrReplaceTempView("pair")

spark.sql("select letter, sum(length) as totalLength, count(*) as totalCount from pair
group by letter").createOrReplaceTempView("count")

avgDF = spark.sql("select letter, totalLength/totalCount as ratio from count order by
letter")

avgDF.write.format("csv").save(outputFolder)
```

PySpark Standalone Code (RDD)

```
from pyspark import SparkContext, SparkConf
import sys

class WordCount:
    def run(self, inputPath, outputPath):
        conf = SparkConf().setAppName("word count").setMaster("local[3]")
        sc = SparkContext(conf=conf)

        fileRDD = sc.textFile(inputPath)
        wordsRDD = fileRDD.flatMap(lambda line: line.lower().split())
        pairsRDD = wordsRDD.map(lambda word: (word, 1))
        countRDD = pairsRDD.reduceByKey(lambda a, b: a+b)

        countRDD.saveAsTextFile(outputPath)
        sc.stop()

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Wrong inputs")
        sys.exit(-1)
    WordCount().run(sys.argv[1], sys.argv[2])
```

PySpark Standalone Code (RDD)

- We first need to create a SparkContext object sc
  - You can utilize the SparkConf object to configure your task
  - You can also write sc = SparkContext("local", "WordCount")

- Parameters for setMaster:
  - local(default) - run locally with only one worker thread (no parallel)
  - local[k] - run locally with k worker threads
  - spark://HOST:PORT - connect to Spark standalone cluster URL
  - mesos://HOST:PORT - connect to Mesos cluster URL
  - yarn - connect to Yarn cluster URL
    - Specified in SPARK_HOME/conf/yarn-site.xml

PySpark Standalone Code (RDD)

- The input and output could be on HDFS or on your local file system
  - We receive them from the command line
    - spark-submit wordcount.py file:///home/comp9313/pg100.txt file:///home/comp9313/outuput
    - spark-submit wordcount.py hdfs:///localhost:9000/user/comp9313/input and hdfs:///localhost:9000/user/comp9313/output
- We can use the RDD textFile() operation to read the data into an RDD
- We can use RDD saveAsTextFile() operation to write the data to disk
  - The result contains parentheses by default

```
('school', 51)
('20210119,nt', 1)
('law', 19)
('darwin', 22)
('20160326,body', 1)
('believed', 4)
('nurse', 9)
```

  - You can format the output and then save to file
- Remember to release the resources by sc.stop() finally

PySpark Standalone Code (DataFrame)

```
from pyspark.sql.session import SparkSession
from pyspark.sql.functions import *
import sys

class WordCount:
    def run(self, inputPath, outputPath):
        spark = SparkSession.builder.master("local").appName("word
count").getOrCreate()
        fileDF = spark.read.text(inputPath)
        wordsDF = fileDF.selectExpr("explode(split(value, ' ')) as
word").withColumn("word", lower(col("word")))

        countDF = wordsDF.groupBy("word").count()
        countDF.write.format("csv").save(outputPath)
        #resDF = countDF.select(concat(col("word"), lit(","), col("count")))
        #resDF.write.text(outputPath)
        spark.stop()

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Wrong inputs")
        sys.exit(-1)
    WordCount().run(sys.argv[1], sys.argv[2])
```

PySpark Standalone Code (DataFrame)

- We first need to create a SparkSession object spark
  - You can utilize the SparkSession.builder to configure your task: spark = SparkSession.builder.master("local").appName("word count").getOrCreate()
- The input and output could be on HDFS or on your local file system
- Remember to release the resources by spark.stop() finally
- pyspark.sql.DataFrame.selectExpr projects a set of SQL expressions and returns a new DataFrame

```
>>>df = spark.createDataFrame([
    (2, "Alice"), (5, "Bob")], schema=["age", "name"])
>>>df.selectExpr("age * 2", "abs(age)").show()
+---------+--------+
|(age * 2)|abs(age)|
+---------+--------+
|        4|       2|
|       10|       5|
+---------+--------+
```

PySpark Standalone Code (DataFrame)

- We can use the pyspark.sql.DataFrameReader.text() operation to read the text data into a DataFrame
  - pyspark.sql.DataFrameReader.csv()
  - pyspark.sql.DataFrameReader.json()
  - https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/io.html

- We can use pyspark.sql.DataFrameWriter.text() to write a DataFrame **with a single column of string type** to a file
- We can use pyspark.sql.DataFrameWriter.csv() or pyspark.sql.DataFrameWriter.format("csv").save() to store the data as a csv file
  - You can also use other formats such as json
  - You can also use pyspark.sql.DataFrameWriter.format("text").save(), but it also requires a DataFrame **with a single column of string type**

Pass a Function to RDD Operations

- For a given text file, find the longest word from each line.

```
from pyspark import SparkContext, SparkConf
import sys

def findlongest(termList):
    maxTerm = ""
    for t in termList:
        if len(t) > len(maxTerm):
            maxTerm = t
    return maxTerm

class WordCount:
    def run(self, inputPath, outputPath):
        sc = SparkContext("local", "longest")
        fileRDD = sc.textFile(inputPath)
        wordsRDD = fileRDD.map(lambda line: line.lower().split())
        longestRDD = wordsRDD.map(lambda termList: findlongest(termList))
        longestRDD.saveAsTextFile(outputPath)
        sc.stop()
if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Wrong inputs")
        sys.exit(-1)
    WordCount().run(sys.argv[1], sys.argv[2])
```

Define Your Own Function with UDF

- PySpark UDF (a.k.a User Defined Function) is the most useful feature of Spark SQL & DataFrame that is used to extend the PySpark build in capabilities.

- PySpark UDF's are similar to UDF on traditional databases. In PySpark, you create a function in a Python syntax and wrap it with PySpark SQL udf() or register it as udf and use it on DataFrame and SQL respectively.

- https://sparkbyexamples.com/pyspark/pyspark-udf-user-defined-function

Pass a Function to DataFrame Operations

- For a given text file, find the longest word from each line.

```
from pyspark.sql.session import SparkSession
from pyspark.sql.functions import *
import sys

def findlongest(termList):
    maxTerm = ""
    for t in termList:
        if len(t) > len(maxTerm):
            maxTerm = t
    return maxTerm

class WordCount:
    def run(self, inputPath, outputPath):
        spark = SparkSession.builder.master("local").appName("longest").getOrCreate()
        fileDF = spark.read.text(inputPath)
        wordsDF = fileDF.select(split(fileDF.value, ' ').alias('termlist'))
        longestUDF = udf(lambda termList: findlongest(termList))
        resDF = wordsDF.withColumn('longest', longestUDF('termlist')).select('longest')
        resDF.write.text(outputPath)
        spark.stop()

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Wrong inputs")
        sys.exit(-1)
    WordCount().run(sys.argv[1], sys.argv[2])
```

Project 2

- we are still going to use the dataset of Australian news from ABC. Your task is to find out the top-k most frequent co-occurring term pairs in each year. The co-occurrence of (w, u) is defined as: u and w appear in the same article headline (i.e., (w, u) and (u, w) are treated equally).

```
20030219,council chief executive fails to secure position
20030219,council welcomes ambulance levy decision
20030219,council welcomes insurance breakthrough
20030219,fed opp to re introduce national insurance
20040501,cowboys survive eels comeback
20040501,cowboys withstand eels fightback
20040502,castro vows cuban socialism to survive bush
20200401,coronanomics things learnt about how coronavirus economy
20200401,coronavirus at home test kits selling in the chinese community
20200401,coronavirus campbell remess streams bear making classes
20201015,coronavirus pacific economy foriegn aid china
20201016,china builds pig apartment blocks to guard against swine flu
```

- You need to ignore the stop words such as “to”, “the”, and “in”. A stop words list will be provided.

Project 2

- Please get the terms from the dataset as below:
  - Split the headlines using the space character
  - Ignore the stop words such as “to”, “the”, and “in”.
  - Ignore terms starting with non-alphabetical characters, i.e., only consider terms starting with “a” to “z”.

- Your output should be in format of (No. of years * k) lines
  - Sort the results first by years in ascending order
  - Within each year, sort the pairs by their frequencies first, then by the pairs in alphabetical order

Project 2

- Key evaluation points:
  - Co-occurrence counting
  - Pass a function to RDD/DataFrame operations
  - Efficient top-k computation
  - Data sorting order
  - Data output format

References

- http://spark.apache.org/docs/latest/index.html
- Spark SQL guide: http://spark.apache.org/docs/latest/sql-programming-guide.html
- https://spark.apache.org/docs/3.3.0/api/scala/org/apache/spark/index.html
- https://spark.apache.org/docs/3.3.0/api/python/getting_started/index.html
- Learning Spark. 2nd edition
- Spark SQL Functions: https://sparkbyexamples.com/spark/spark-sql-functions/

End of Chapter 5.1