Crash Course on Spark

Amir H. Payberah
payberah@kth.se
07/08/2019

Big Data


BIG DATA
BIG DATA EVERYWHERE
makeameme.org

What is Big Data?

Big Data

Big data is the data characterized by 4 key attributes: volume, variety, velocity and value.

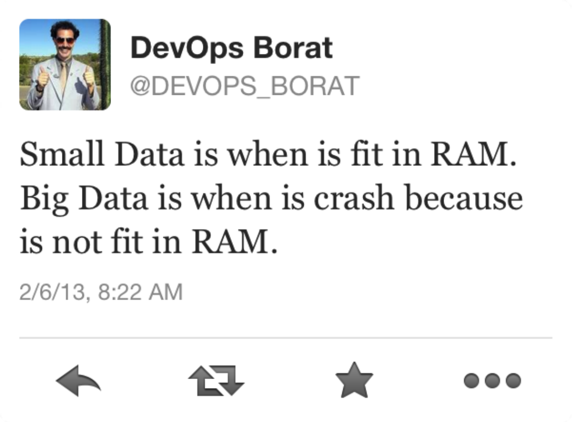
ORACLE®

Big Data

Big data is the data characterized by 4 key attributes: volume, variety, velocity and value.

Buzzwords

Big Data


DevOps Borat
@DEVOPS_BORAT
Small Data is when is fit in RAM. Big Data is when is crash because is not fit in RAM.
2/6/13, 8:22 AM

How To Store and Process Big Data?

Problem

- Traditional platforms fail to show the expected performance.
- Need new systems to store and process large-scale data

Scale Up vs. Scale Out

- Scale up or scale vertically: adding resources to a single node in a system.
- Scale out or scale horizontally: adding more nodes to a system.

KTH
VETENSKAP OCH KONST

APACHE HBASE

mahout

GraphX

hadoop

kafka

Storm

Spark

Dato

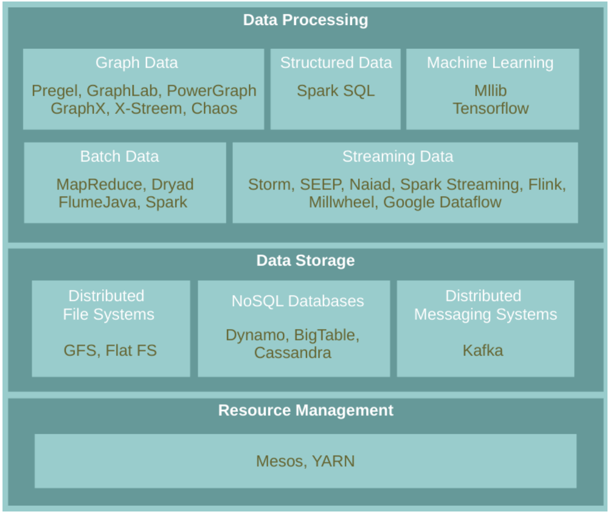
HIVE

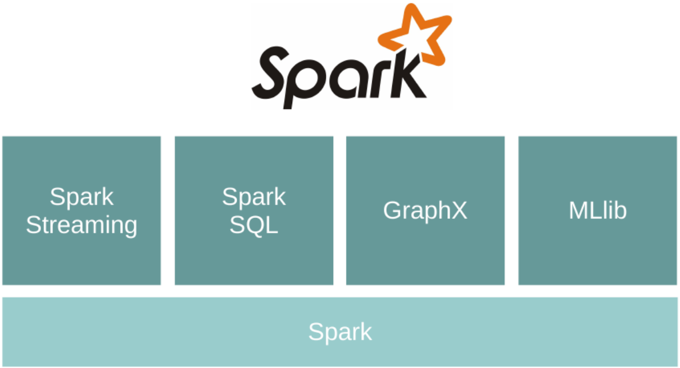
S4 distributed stream computing platform

cassandra

Google Cloud Platform

Big Data Stack




Spark
Spark Streaming
Spark SQL
GraphX
MLlib
Spark

Scala

Scala

- Scala: scalable language
- A blend of object-oriented and functional programming
- Runs on the Java Virtual Machine
- Designed by Martin Odersky at EPFL

Functional Programming Languages

- Functions are first-class citizens:
  - Defined anywhere (including inside other functions).
  - Passed as parameters to functions and returned as results.
  - Operators to compose functions.

Scala Variables

- Values: immutable
- Variables: mutable

```scala
var myVar: Int = 0
val myVal: Int = 1
```

- Scala data types:
  - Boolean, Byte, Short, Char, Int, Long, Float, Double, String

If ... Else

```
var x = 30;

if (x == 10) {
  println("Value of X is 10");
} else if (x == 20) {
  println("Value of X is 20");
} else {
  println("This is else statement");
}
```

Loop

```scala
var a = 0
var b = 0
for (a <- 1 to 3; b <- 1 until 3) {
  println("Value of a: " + a + ", b: " + b )
}
```

```scala
// loop with collections
val numList = List(1, 2, 3, 4, 5, 6)
for (a <- numList) {
  println("Value of a: " + a)
}
```

Functions

```
def functionName([list of parameters]): [return type] = {
  function body
  return [expr]
}

def addInt(a: Int, b: Int): Int = {
  var sum: Int = 0
  sum = a + b
  sum
}

println("Returned Value: " + addInt(5, 7))
```

Anonymous Functions

- Lightweight syntax for defining functions.

```
var mul = (x: Int, y: Int) => x * y
println(mul(3, 4))
```

Higher-Order Functions


```scala
def apply(f: Int => String, v: Int) = f(v)

def layout(x: Int) = "[" + x.toString() + "]"

println(apply(layout, 10))
```

Collections (1/2)

- Array: fixed-size sequential collection of elements of the same type

```
val t = Array("zero", "one", "two")
val b = t(0) // b = zero
```

Collections (1/2)

- Array: fixed-size sequential collection of elements of the same type

```
val t = Array("zero", "one", "two")
val b = t(0) // b = zero
```

- List: sequential collection of elements of the same type

```
val t = List("zero", "one", "two")
val b = t(0) // b = zero
```

Collections (1/2)

- Array: fixed-size sequential collection of elements of the same type

```
val t = Array("zero", "one", "two")
val b = t(0) // b = zero
```

- List: sequential collection of elements of the same type

```
val t = List("zero", "one", "two")
val b = t(0) // b = zero
```

- Set: sequential collection of elements of the same type without duplicates

```
val t = Set("zero", "one", "two")
val t.contains("zero")
```

Collections (2/2)

- Map: collection of key/value pairs

```
val m = Map(1 -> "sics", 2 -> "kth")
val b = m(1) // b = sics
```

Collections (2/2)

- Map: collection of key/value pairs

```
val m = Map(1 -> "sics", 2 -> "kth")
val b = m(1) // b = sics
```

- Tuple: A fixed number of items of different types together

```
val t = (1, "hello")
val b = t._1 // b = 1
val c = t._2 // c = hello
```

Functional Combinators

- map: applies a function over each element in the list

```
val numbers = List(1, 2, 3, 4)
numbers.map(i => i * 2) // List(2, 4, 6, 8)
```

- flatten: it collapses one level of nested structure

```
List(List(1, 2), List(3, 4)).flatten // List(1, 2, 3, 4)
```

- flatMap: map + flatten
- foreach: it is like map but returns nothing

Classes and Objects

```scala
class Calculator {
  val brand: String = "HP"
  def add(m: Int, n: Int): Int = m + n
}

val calc = new Calculator
calc.add(1, 2)
println(calc.brand)
```

Classes and Objects

```scala
class Calculator {
  val brand: String = "HP"
  def add(m: Int, n: Int): Int = m + n
}

val calc = new Calculator
calc.add(1, 2)
println(calc.brand)
```

```scala
object Test {
  def main(args: Array[String]) { ... }
}

Test.main(null)
```

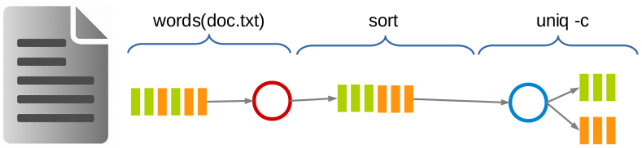
```scala
case class Calc(brand: String, model: String)
```

Data Intensive Computing


SAY "WORD COUNT"
ONE MORE TIME...

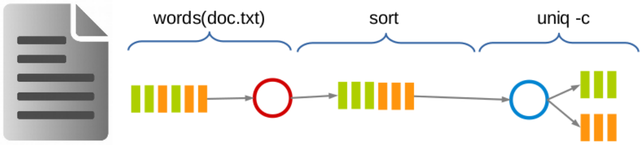
Word Count

- Count the number of times each distinct word appears in the file
- If the file fits in memory: `words(doc.txt) | sort | uniq -c`



Word Count

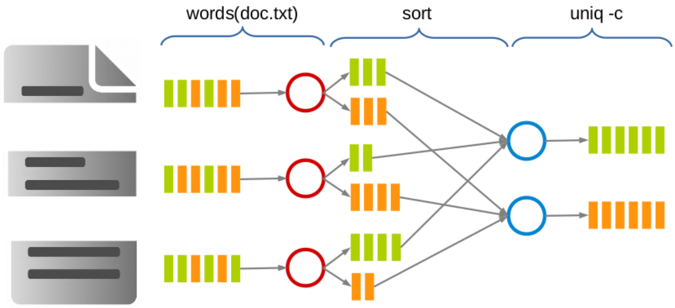
- Count the number of times each distinct word appears in the file
- If the file fits in memory: `words(doc.txt) | sort | uniq -c`



- If not?

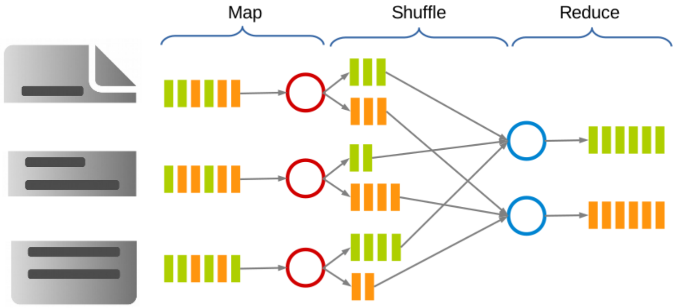
Data-Parallel Processing (1/2)

- Parallelize the data and process.



Data-Parallel Processing (2/2)

- MapReduce

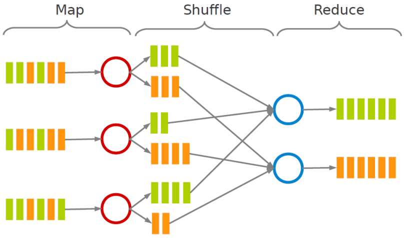


MapReduce Programming Model (1/2)

- `words(doc.txt) | sort | uniq -c`

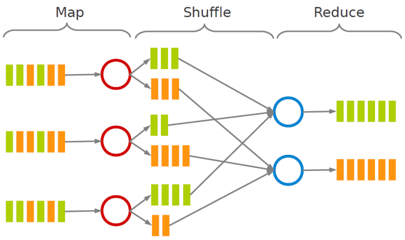
MapReduce Programming Model (1/2)

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.



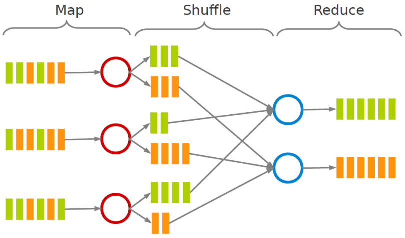
MapReduce Programming Model (1/2)

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.



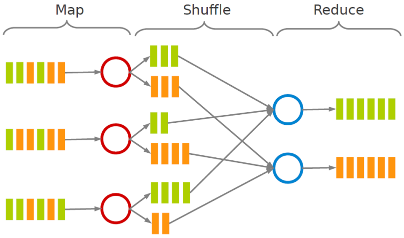
MapReduce Programming Model (1/2)

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.



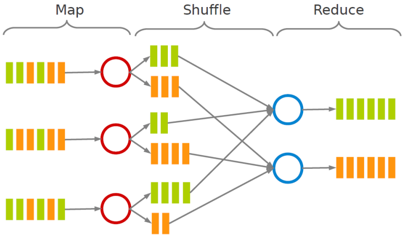
MapReduce Programming Model (1/2)

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.
- Reduce: aggregate, summarize, filter or transform.



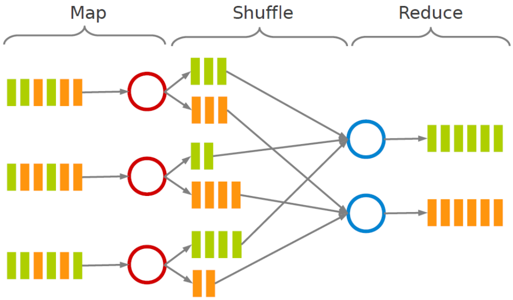
MapReduce Programming Model (1/2)

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.
- Reduce: aggregate, summarize, filter or transform.
- Write the result.



MapReduce Programming Model (2/2)

- map function: processes data and generates a set of intermediate key/value pairs.
- reduce function: merges all intermediate values associated with the same intermediate key.



Word Count in MapReduce

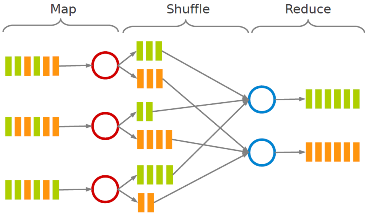
- Consider doing a word count of the following file using MapReduce:

```
Hello World Bye World
Hello Hadoop Goodbye Hadoop
```

Word Count in MapReduce - map

- The map function reads in words one a time and outputs (word, 1) for each parsed input word.
- The map function output is:

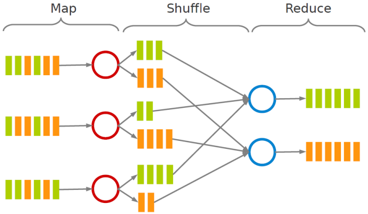
```
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
```



Word Count in MapReduce - shuffle

- The shuffle phase between map and reduce phase creates a list of values associated with each key.
- The reduce function input is:

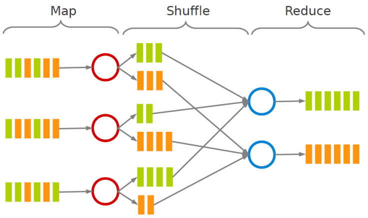
```
(Bye, (1))
(Goodbye, (1))
(Hadoop, (1, 1))
(Hello, (1, 1))
(World, (1, 1))
```



Word Count in MapReduce - reduce

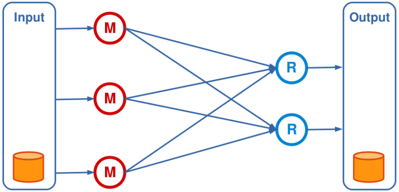
- The reduce function sums the numbers in the list for each key and outputs (word, count) pairs.
- The output of the reduce function is the output of the MapReduce job:

```
(Bye, 1)
(Goodbye, 1)
(Hadoop, 2)
(Hello, 2)
(World, 2)
```



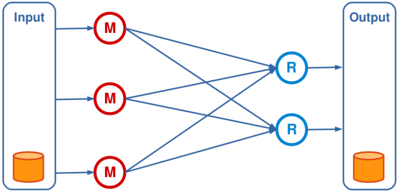
Dataflow Programming Model

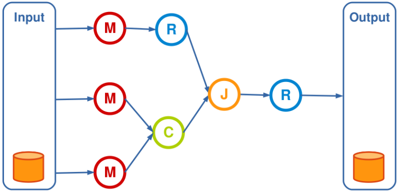
- Acyclic data flow from stable storage to stable storage.



Dataflow Programming Model

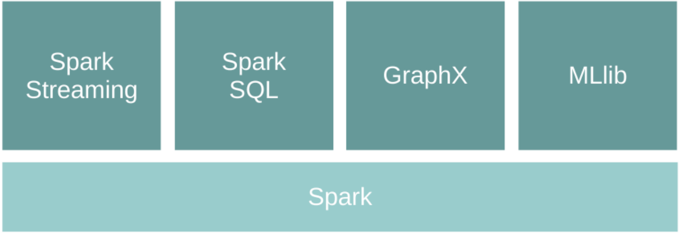
- Acyclic data flow from stable storage to stable storage.





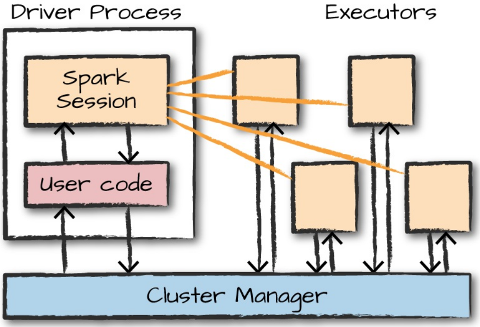
Spark

Spark



Spark Applications Architecture

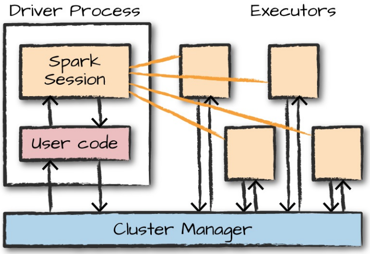
- Spark applications consist of
  - A driver process
  - A set of executor processes



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

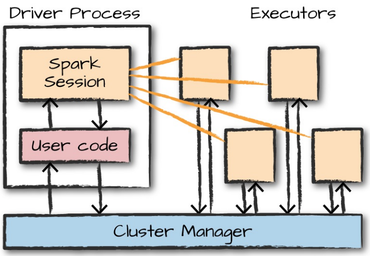
Driver Process

- The heart of a Spark application
- Sits on a node in the cluster
- Runs the `main()` function



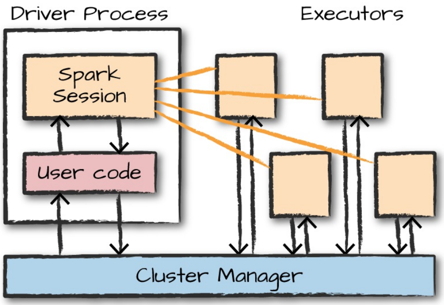
Driver Process

- The heart of a Spark application
- Sits on a node in the cluster
- Runs the `main()` function
- Responsible for three things:
  - Maintaining information about the Spark application
  - Responding to a user's program or input
  - Analyzing, distributing, and scheduling work across the executors



Executors

- Responsible for two things:
  - Executing code assigned to it by the driver
  - Reporting the state of the computation on that executor back to the driver



SparkSession and SparkContext

- Main entry point to Spark functionality.
- SparkSession is available in console shell as spark.
- SparkContext is available in console shell as sc.

```
// spark session
spark = SparkSession.builder.master(master).appName(appName).getOrCreate()

// spark context
val conf = new SparkConf().setMaster(master).setAppName(appName)
sc = new SparkContext(conf)
```

SparkSession vs. SparkContext

- Prior to Spark 2.0.0, a the spark driver program uses `SparkContext` to connect to the cluster.
- In order to use APIs of SQL, Hive and streaming, separate `SparkContexts` should to be created.

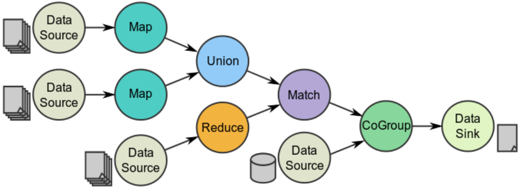
SparkSession vs. SparkContext

- Prior to Spark 2.0.0, a the spark driver program uses SparkContext to connect to the cluster.
- In order to use APIs of SQL, Hive and streaming, separate SparkContexts should to be created.
- SparkSession provides access to all the spark functionalities that SparkContext does, e.g., SQL, Hive and streaming.
- SparkSession internally has a SparkContext for actual computation.

Programming Model

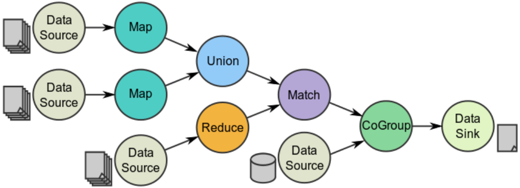
Spark Programming Model

- Job is described based on directed acyclic graphs (DAG) data flow.



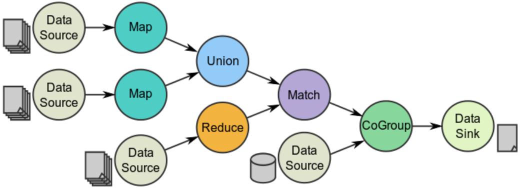
Spark Programming Model

- Job is described based on directed acyclic graphs (DAG) data flow.
- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.



Spark Programming Model

- Job is described based on directed acyclic graphs (DAG) data flow.
- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.
- Parallelizable operators



Resilient Distributed Datasets (RDD) (1/3)

- A distributed memory abstraction.
- Immutable collections of objects spread across a cluster.
  - Like a `LinkedList <MyObjects>`

Resilient Distributed Datasets (RDD) (2/3)

- An RDD is divided into a number of partitions, which are atomic pieces of information.
- Partitions of an RDD can be stored on different nodes of a cluster.

Resilient Distributed Datasets (RDD) (3/3)

- RDDs were the primary API in the Spark 1.x series.
- They are not commonly used in the Spark 2.x series.
- Virtually all Spark code you run, compiles down to an RDD.

Types of RDDs

- Two types of RDDs:
  - Generic RDD
  - Key-value RDD
- Both represent a collection of objects.
- Key-value RDDs have special operations, such as aggregation, and a concept of custom partitioning by key.

Creating RDDs

Creating RDDs - Parallelized Collections

- Use the `parallelize` method on a `SparkContext`.
- This turns a single node collection into a parallel collection.
- You can also explicitly state the number of partitions.
- In the console shell, you can either use `sc` or `spark.sparkContext`

```
val numsCollection = Array(1, 2, 3)
val nums = sc.parallelize(numsCollection)

val wordsCollection = "take it easy, this is a test".split(" ")
val words = spark.sparkContext.parallelize(wordsCollection, 2)
```

Creating RDDs - External Datasets

- Create RDD from an external storage.
  - E.g., local file system, HDFS, Cassandra, HBase, Amazon S3, etc.

- Text file RDDs can be created using `textFile` method.

```
val myFile1 = sc.textFile("file.txt")
val myFile2 = sc.textFile("hdfs://namenode:9000/path/file")
```

RDD Operations

RDD Operations

- RDDs support two types of operations:
  - Transformations: allow us to build the logical plan
  - Actions: allow us to trigger the computation

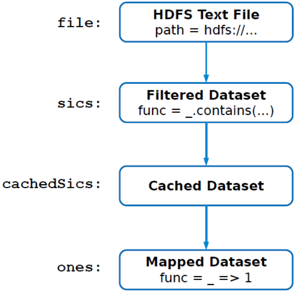
Transformations

Transformations

- Create a new RDD from an existing one.
- All transformations are lazy.
  - Not compute their results right away.
  - Remember the transformations applied to the base dataset (lineage).
  - They are only computed when an action requires a result to be returned to the driver program.

Lineage

- Lineage: transformations used to build an RDD.
- RDDs are stored as a chain of objects capturing the lineage of each RDD.




```scala
val file = sc.textFile("hdfs://...")
val sics = file.filter(_.contains("SICS"))
val cachedSics = sics.cache()
val ones = cachedSics.map(_ => 1)
val count = ones.reduce(_+_)
```

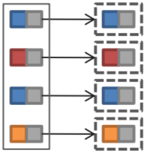
Generic RDD Transformations (1/3)

- `distinct` removes duplicates from the RDD.
- `filter` returns the RDD records that match some predicate function.

```
val words = sc.parallelize("this it easy, this is a test".split(" "))
val distinctWords = words.distinct()
// a, this, is, easy,, test, it

val nums = sc.parallelize(Array(1, 2, 3))
val even = nums.filter(x => x % 2 == 0)
// 2

def startsWithT(individual:String) = { individual.startsWith("t") }
val tWordList = words.filter(word => startsWithT(word))
// take, this, test
```

Generic RDD Transformations (2/3)

- `map` and `flatMap` apply a given function on each RDD record independently.



```
val nums = sc.parallelize(Array(1, 2, 3))
val squares = nums.map(x => x * x)
// 1, 4, 9

val words = sc.parallelize("take it easy, this is a test".split(" "))
val tWords = words.map(word => (word, word.startsWith("t")))
// (take,true), (it,false), (easy,,false), (this,true), (is,false), (a,false), (test,true)

val chars = words.flatMap(word => word.toSeq)
// t, a, k, e, i, t, e, a, s, y, ,, t, h, i, s, i, s, a, t, e, s, t
```

Generic RDD Transformations (3/3)

- `sortBy` sorts an RDD records.

```
val words = sc.parallelize("take it easy, this is a test".split(" "))
val sortedWords = words.sortBy(word => word.length())
// a, it, is, take, this, test, easy,
```

Key-Value RDD Transformations - Basics (1/2)

- In a (k, v) pairs, k is is the key, and v is the value.
- To make a key-value RDD:
  - `map` over your current RDD to a basic key-value structure.
  - Use the `keyBy` to create a key from the current value.
  - Use the `zip` to zip together two RDD.

```scala
val numRange = sc.parallelize(0 to 6)
val words = sc.parallelize("take it easy, this is a test".split(" "))

val keyword1 = words.map(word => (word.toLowerCase, 1))
// (take,1), (it,1), (easy,,1), (this,1), (is,1), (a,1), (test,1)

val keyword2 = words.keyBy(word => word.toLowerCase.toSeq(0).toString)
// (t,take), (i,it), (e,easy,), (t,this), (i,is), (a,a), (t,test)

val keyword3 = words.zip(numRange)
// (take,0), (it,1), (easy,,2), (this,3), (is,4), (a,5), (test,6)
```

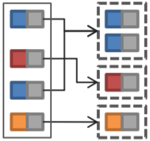
Key-Value RDD Transformations - Basics (2/2)

- `keys` and `values` extract keys and values, respectively.
- `lookup` looks up the values for a particular key with an RDD.
- `mapValues` maps over values.

```scala
val words = sc.parallelize("take it easy, this is a test".split(" "))

val keyword = words.keyBy(word => word.toSeq(0).toString)
// (t,take), (i,it), (e,easy,), (t,this), (i,is), (a,a), (t,test)

val k = keyword.keys
val v = keyword.values

val tValues = keyword.lookup("t")
// take, this, test

val mapV = keyword.mapValues(word => word.toUpperCase)
// (t,TAKE), (i,IT), (e,EASY,), (t,THIS), (i,IS), (a,A), (t,TEST)
```

Key-Value RDD Transformations - Aggregation

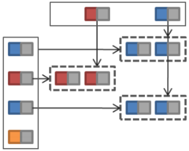
- Aggregate the values associated with each key.



```scala
val words = sc.parallelize("take it easy, this is a test".split(" "))
val chars = words.flatMap(word => word.toSeq)
val kvChars = chars.map(letter => (letter, 1))
// (t,1), (a,1), (k,1), (e,1), (i,1), (t,1), (e,1), (a,1), (s,1), (y,1), (,,1), ...

def addFunc(left:Int, right:Int) = left + right

val grpChar = kvChars.groupByKey().map(row => (row._1, row._2.reduce(addFunc)))
// (t,5), (h,1), (,,1), (e,3), (a,3), (i,3), (y,1), (s,4), (k,1))

val redChar = kvChars.reduceByKey(addFunc)
// (t,5), (h,1), (,,1), (e,3), (a,3), (i,3), (y,1), (s,4), (k,1))
```

Key-Value RDD Transformations - Join

- `join` performs an inner-join on the key.
- `fullOtherJoin`, `leftOuterJoin`, `rightOuterJoin`, and `cartesian`.



```scala
val words = sc.parallelize("take it easy, this is a test".split(" "))
val chars = words.flatMap(word => word.toSeq)
val distinctChars = chars.distinct

val keyedChars = distinctChars.map(c => (c, new Random().nextInt(10)))
// (t,4), (h,6), (,,9), (e,8), (a,3), (i,5), (y,2), (s,7), (k,0)
val kvChars = chars.map(letter => (letter, 1))
// (t,1), (a,1), (k,1), (e,1), (i,1), (t,1), (e,1), (a,1), (s,1), (y,1), (,,1), ...
val joinedChars = kvChars.join(keyedChars)
// (t,(1,4)), (t,(1,4)), (t,(1,4)), (t,(1,4)), (t,(1,4)), (h,(1,6)), (,,(1,9)), (e,(1,8)), ...
```

Actions

Actions

- Transformations allow us to build up our logical transformation plan.
- We run an action to trigger the computation.
  - Instructs Spark to compute a result from a series of transformations.

Actions

- Transformations allow us to build up our logical transformation plan.
- We run an action to trigger the computation.
  - Instructs Spark to compute a result from a series of transformations.
- There are three kinds of actions:
  - Actions to view data in the console
  - Actions to collect data to native objects in the respective language
  - Actions to write to output data sources

RDD Actions (1/6)

- `collect` returns all the elements of the RDD as an array at the driver.
- `first` returns the first value in the RDD.

```
val nums = sc.parallelize(Array(1, 2, 3))

nums.collect()
// Array(1, 2, 3)

nums.first()
// 1
```

RDD Actions (2/6)

- take returns an array with the first n elements of the RDD.
- Variations on this function: takeOrdered and takeSample.

```
val words = sc.parallelize("take it easy, this is a test".split(" "))

words.take(5)
// Array(take, it, easy,, this, is)

words.takeOrdered(5)
// Array(a, easy,, is, it, take)

val withReplacement = true
val numberToTake = 6
val randomSeed = 100L
words.takeSample(withReplacement, numberToTake, randomSeed)
// Array(take, it, test, this, test, take)
```

RDD Actions (3/6)

- `count` returns the number of elements in the dataset.
- `countByValue` counts the number of values in a given RDD.
- `countByKey` returns a hashmap of (`K`, `Int`) pairs with the count of each key.
  - Only available on key-valye RDDs, i.e., (`K`, `V`)

```scala
val words = sc.parallelize("take it easy, this is a test, take it easy".split(" "))

words.count()
// 10

words.countByValue()
// Map(this -> 1, is -> 1, it -> 2, a -> 1, easy, -> 1, test, -> 1, take -> 2, easy -> 1)
```

RDD Actions (4/6)

- `max` and `min` return the maximum and minimum values, respectively.

```
val nums = sc.parallelize(1 to 20)

val maxValue = nums.max()
// 20

val minValue = nums.min()
// 1
```

RDD Actions (5/6)

- `reduce` aggregates the elements of the dataset using a given function.
- The given function should be commutative and associative so that it can be computed correctly in parallel.

```scala
sc.parallelize(1 to 20).reduce(_ + _)
// 210

def wordLengthReducer(leftWord:String, rightWord:String): String = {
  if (leftWord.length > rightWord.length)
    return leftWord
  else
    return rightWord
}

words.reduce(wordLengthReducer)
// easy,
```

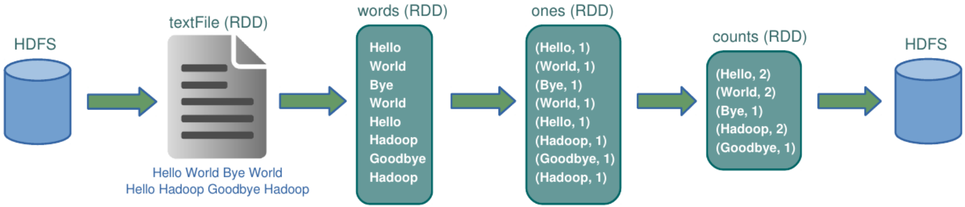
RDD Actions (6/6)

- `saveAsTextFile` writes the elements of an RDD as a text file.
  - Local filesystem, HDFS or any other Hadoop-supported file system.
- `saveAsObjectFile` explicitly writes key-value pairs.

```
val words = sc.parallelize("take it easy, this is a test".split(" "))

words.saveAsTextFile("file:/tmp/words")
```

Example

```
val textFile = sc.textFile("hdfs://...")

val words = textFile.flatMap(line => line.split(" "))
val ones = words.map(word => (word, 1))
val counts = ones.reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```



Cache and Checkpoints

Caching

- When you cache an RDD, each node stores any partitions of it that it computes in memory.
- An RDD that is not cached is re-evaluated each time an action is invoked on that RDD.
- A node reuses the cached RDD in other actions on that dataset.

Caching

- When you cache an RDD, each node stores any partitions of it that it computes in memory.
- An RDD that is not cached is re-evaluated each time an action is invoked on that RDD.
- A node reuses the cached RDD in other actions on that dataset.
- There are two functions for caching an RDD:
  - `cache` caches the RDD into memory
  - `persist(level)` can cache in memory, on disk, or off-heap memory

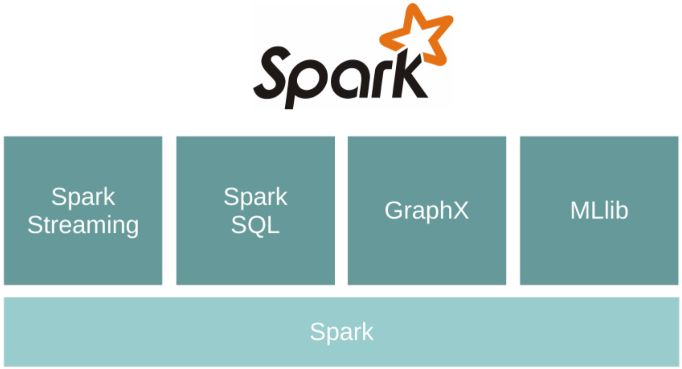
```
val words = sc.parallelize("take it easy, this is a test".split(" "))

words.cache()
```

Checkpointing

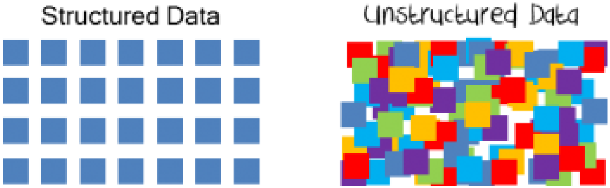
- `checkpoint` saves an RDD to disk.
- Checkpointed data is not removed after `SparkContext` is destroyed.
- When we reference a checkpointed RDD, it will derive from the checkpoint instead of the source data.

```
val words = sc.parallelize("take it easy, this is a test".split(" "))

sc.setCheckpointDir("/path/checkpointing")
words.checkpoint()
```

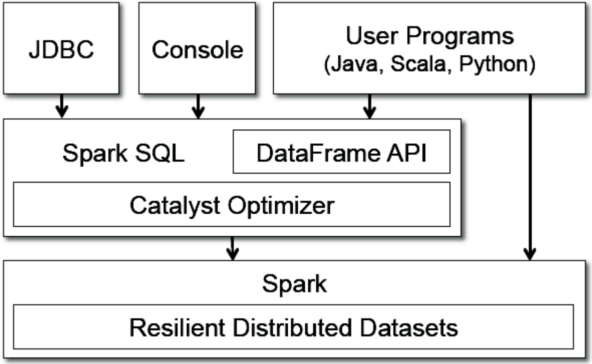
Spark SQL


Spark
Spark Streaming
Spark SQL
GraphX
MLlib
Spark

Motivation



Spark and Spark SQL



Structured Data vs. RDD (1/2)

- `case class Account(name: String, balance: Double, risk: Boolean)`

Structured Data vs. RDD (1/2)

- `case class Account(name: String, balance: Double, risk: Boolean)`
- `RDD[Account]`

Structured Data vs. RDD (1/2)

- `case class Account(name: String, balance: Double, risk: Boolean)`
- `RDD[Account]`
- RDDs don't know anything about the schema of the data it's dealing with.

Structured Data vs. RDD (2/2)

- `case class Account(name: String, balance: Double, risk: Boolean)`
- `RDD[Account]`
- A database/Hive sees it as a columns of named and typed values.

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

DataFrames and DataSets

- Spark has two notions of structured collections:
  - DataFrames
  - Datasets
- They are distributed table-like collections with well-defined rows and columns.

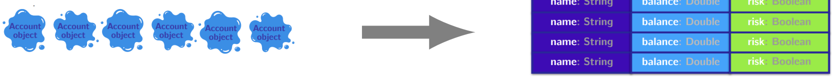
DataFrames and DataSets

- Spark has two notions of structured collections:
  - DataFrames
  - Datasets
- They are distributed table-like collections with well-defined rows and columns.
- They represent immutable lazily evaluated plans.
- When an action is performed on them, Spark performs the actual transformations and return the result.

DataFrame

DataFrame

- Consists of a series of rows and a number of columns.
- Equivalent to a table in a relational database.
- Spark + RDD: functional transformations on partitioned collections of objects.
- SQL + DataFrame: declarative transformations on partitioned collections of tuples.



Creating a DataFrame

- Two ways to create a DataFrame:
  1. From an RDD
  2. From raw data sources

Creating a DataFrame - From an RDD

- The schema automatically inferred.

Creating a DataFrame - From an RDD

- The schema automatically inferred.
- You can use `toDF` to convert an RDD to DataFrame.

```
val tupleRDD = sc.parallelize(Array(("seif", 65, 0), ("amir", 40, 1)))
val tupleDF = tupleRDD.toDF("name", "age", "id")
```

Creating a DataFrame - From an RDD

- The schema automatically inferred.
- You can use `toDF` to convert an RDD to DataFrame.

```
val tupleRDD = sc.parallelize(Array(("seif", 65, 0), ("amir", 40, 1)))
val tupleDF = tupleRDD.toDF("name", "age", "id")
```

- If RDD contains `case` class instances, Spark infers the attributes from it.

```
case class Person(name: String, age: Int, id: Int)
val peopleRDD = sc.parallelize(Array(Person("seif", 65, 0), Person("amir", 40, 1)))
val peopleDF = peopleRDD.toDF()
```

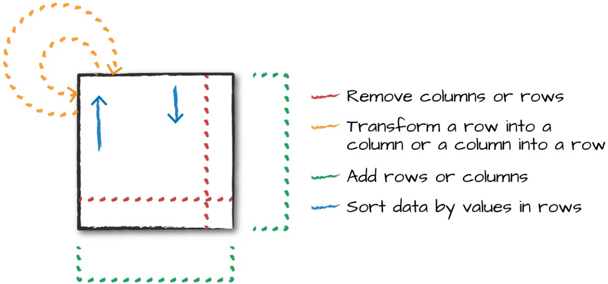
Creating a DataFrame - From Data Source

- Data sources supported by Spark.
  - CSV, JSON, Parquet, ORC, JDBC/ODBC connections, Plain-text files
  - Cassandra, HBase, MongoDB, AWS Redshift, XML, etc.

```
val peopleJson = spark.read.format("json").load("people.json")

val peopleCsv = spark.read.format("csv")
  .option("sep", ";")
  .option("inferSchema", "true")
  .option("header", "true")
  .load("people.csv")
```

DataFrame Transformations (1/4)

- Add and remove rows or columns
- Transform a row into a column (or vice versa)
- Change the order of rows based on the values in columns



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

DataFrame Transformations (2/4)

- `select` and `selectExpr` allow to do the DataFrame equivalent of SQL queries on a table of data.

```
// select
people.select("name", "age", "id").show(2)
people.select(col("name"), expr("age + 3")).show()
people.select(expr("name AS username")).show(2)

// selectExpr
people.selectExpr("*", "(age < 20) as teenager").show()
people.selectExpr("avg(age)", "count(distinct(name))", "sum(id)").show()
```

DataFrame Transformations (3/4)

- `filter` and `where` both filter rows.
- `distinct` can be used to extract unique rows.

```
people.filter(col("age") < 20).show()

people.where("age < 20").show()

people.select("name").distinct().count()
```

DataFrame Transformations (4/4)

- `withColumn` adds a new column to a DataFrame.
- `withColumnRenamed` renames a column.
- `drop` removes a column.

```
// withColumn
people.withColumn("teenager", expr("age < 20")).show()

// withColumnRenamed
people.withColumnRenamed("name", "username").columns

// drop
people.drop("name").columns
```

DataFrame Actions

- Like RDDs, DataFrames also have their own set of actions.
- `collect`: returns an array that contains all of rows in this DataFrame.
- `count`: returns the number of rows in this DataFrame.
- `first` and `head`: returns the first row of the DataFrame.
- `show`: displays the top 20 rows of the DataFrame in a tabular form.
- `take`: returns the first n rows of the DataFrame.

Aggregation

Aggregation

- In an aggregation you specify
  - A key or grouping
  - An aggregation function
- The given function must produce one result for each group.

Grouping Types

- Summarizing a complete DataFrame
- Group by
- Windowing

Grouping Types

- Summarizing a complete DataFrame
- Group by
- Windowing

Summarizing a Complete DataFrame Functions (1/2)

- count returns the total number of values.
- countDistinct returns the number of unique groups.
- first and last return the first and last value of a DataFrame.

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

val people = spark.read.format("json").load("people.json")

people.selectExpr(count("age")).show()

people.select(countDistinct("name")).show()

people.select(first("name"), last("age")).show()
```

Summarizing a Complete DataFrame Functions (2/2)

- min and max extract the minimum and maximum values from a DataFrame.
- sum adds all the values in a column.
- avg calculates the average.

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

val people = spark.read.format("json").load("people.json")

people.select(min("name"), max("age"), max("id")).show()

people.select(sum("age")).show()

people.select(avg("age")).show()
```

Grouping Types

- Summarizing a complete DataFrame
- Group by
- Windowing

Group By (1/3)

- Perform aggregations on groups in the data.
- Typically on categorical data.
- We do this grouping in two phases:
  1. Specify the column(s) on which we would like to group.
  2. Specify the aggregation(s).

Group By (2/3)

- Grouping with expressions
  - Rather than passing that function as an expression into a `select` statement, we specify it as within agg.

```
val people = spark.read.format("json").load("people.json")

people.groupBy("name").agg(count("age").alias("ageagg")).show()
```

Group By (3/3)

- Grouping with Maps
  - Specify transformations as a series of Maps
  - The key is the column, and the value is the aggregation function (as a string).

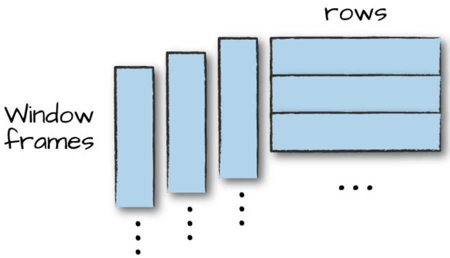
```
val people = spark.read.format("json").load("people.json")

people.groupBy("name").agg("age" -> "count", "age" -> "avg", "id" -> "max").show()
```

Grouping Types

- Summarizing a complete DataFrame
- Group by
- Windowing

Windowing (1/2)

- Computing some aggregation on a specific window of data.
- The window determines which rows will be passed in to this function.
- You define them by using a reference to the current data.
- A group of rows is called a frame.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

Windowing (2/2)

- Unlike grouping, here each row can fall into one or more frames.

```
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions.col

val people = spark.read.format("json").load("people.json")

val windowSpec = Window.rowsBetween(-1, 1)
val avgAge = avg(col("age")).over(windowSpec)
people.select(col("name"), col("age"), avgAge.alias("avg_age")).show
```

Joins

Joins

- A join goes throught the following steps:
  - Compares the value of one or more keys of the left and right datasets.
  - Evaluates the result of a join expression.
  - Determines whether Spark should bring together the left set of data with the right set of data.
- Different join types: inner join, outer join, left outer join, right outer join, left semi join, left anti join, cross join

Joins Example

```scala
val person = Seq(
  (0, "Seif", 0),
  (1, "Amir", 1),
  (2, "Sarunas", 1))
  .toDF("id", "name", "group_id")

val group = Seq(
  (0, "SICS/KTH"),
  (1, "KTH"),
  (2, "SICS"))
  .toDF("id", "department")
```

Joins Example - Inner

```
val joinExpression = person.col("group_id") === group.col("id")

var joinType = "inner"

person.join(group, joinExpression, joinType).show()
```

```
+---+-------+--------+---+----------+
| id|   name|group_id| id|department|
+---+-------+--------+---+----------+
|  0|   Seif|       0|  0|  SICS/KTH|
|  1|   Amir|       1|  1|       KTH|
|  2|Sarunas|       1|  1|       KTH|
+---+-------+--------+---+----------+
```

Joins Example - Outer

```scala
val joinExpression = person.col("group_id") === group.col("id")

var joinType = "outer"

person.join(group, joinExpression, joinType).show()
```

```
+----+-------+--------+---+----------+
|  id|   name|group_id| id|department|
+----+-------+--------+---+----------+
|   1|   Amir|       1|  1|       KTH|
|   2|Sarunas|       1|  1|       KTH|
|null|   null|    null|  2|      SICS|
|   0|   Seif|       0|  0|  SICS/KTH|
+----+-------+--------+---+----------+
```

Joins Example - Right Outer

```
val joinExpression = person.col("group_id") === group.col("id")

var joinType = "right_outer"

person.join(group, joinExpression, joinType).show()
```

```
+----+-------+--------+---+----------+
|  id|   name|group_id| id|department|
+----+-------+--------+---+----------+
|   0|   Seif|       0|  0|  SICS/KTH|
|   2|Sarunas|       1|  1|       KTH|
|   1|   Amir|       1|  1|       KTH|
|null|   null|    null|  2|      SICS|
+----+-------+--------+---+----------+
```

SQL

SQL

- You can run SQL queries on views/tables via the method `sql` on the `SparkSession` object.

```
spark.sql("SELECT * from people_view").show()
```

```
+---+---+-------+
|age| id|   name|
+---+---+-------+
| 15| 12|Michael|
| 30| 15|   Andy|
| 19| 20| Justin|
| 12| 15|   Andy|
| 19| 20|    Jim|
| 12| 10|   Andy|
+---+---+-------+
```

Temporary View

- createOrReplaceTempView creates (or replaces) a lazily evaluated view.
- You can use it like a table in Spark SQL.
- It does not persist to memory unless you cache it.

```
val people = spark.read.format("json").load("people.json")

people.createOrReplaceTempView("people_view")

val teenagersDF = spark.sql("SELECT name, age FROM people_view WHERE age BETWEEN 13 AND 19").show()
```

DataSet

Untyped API with DataFrame

- DataFrames elements are `Row`s, which are generic untyped JVM objects.
- Scala compiler cannot type check Spark SQL schemas in DataFrames.

Untyped API with DataFrame

- DataFrames elements are `Rows`, which are generic untyped JVM objects.
- Scala compiler cannot type check Spark SQL schemas in DataFrames.
- The following code compiles, but you get a runtime exception.
  - `id_num` is not in the DataFrame columns `[name, age, id]`

```scala
// people columns: ("name", "age", "id")
val people = spark.read.format("json").load("people.json")

people.filter("id_num < 20") // runtime exception
```

Why DataSet?

- Assume the following example

```
case class Person(name: String, age: BigInt, id: BigInt)
val peopleRDD = sc.parallelize(Array(Person("seif", 65, 0), Person("amir", 40, 1)))
val peopleDF = peopleRDD.toDF
```

Why DataSet?

- Assume the following example

```scala
case class Person(name: String, age: BigInt, id: BigInt)
val peopleRDD = sc.parallelize(Array(Person("seif", 65, 0), Person("amir", 40, 1)))
val peopleDF = peopleRDD.toDF
```

- Now, let's use `collect` to bring back it to the master.

```scala
val collectedPeople = peopleDF.collect()
// collectedPeople: Array[org.apache.spark.sql.Row]
```

Why DataSet?

- Assume the following example

```
case class Person(name: String, age: BigInt, id: BigInt)
val peopleRDD = sc.parallelize(Array(Person("seif", 65, 0), Person("amir", 40, 1)))
val peopleDF = peopleRDD.toDF
```

- Now, let's use `collect` to bring back it to the master.

```
val collectedPeople = peopleDF.collect()
// collectedPeople: Array[org.apache.spark.sql.Row]
```

- What is in `Row`?

Why DataSet?

- To be able to work with the collected values, we should cast the `Rows`.
  - How many columns?
  - What types?

```
// Person(name: Sting, age: BigInt, id: BigInt)

val collectedList = collectedPeople.map {
  row => (row(0).asInstanceOf[String], row(1).asInstanceOf[Int], row(2).asInstanceOf[Int])
}
```

Why DataSet?

- To be able to work with the collected values, we should cast the `Rows`.
  - How many columns?
  - What types?

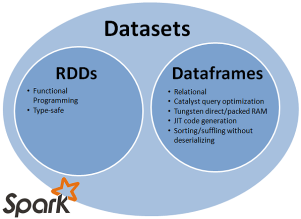
```
// Person(name: Sting, age: BigInt, id: BigInt)

val collectedList = collectedPeople.map {
  row => (row(0).asInstanceOf[String], row(1).asInstanceOf[Int], row(2).asInstanceOf[Int])
}
```

- But, what if we cast the types wrong?
- Wouldn't it be nice if we could have both Spark SQL optimizations and typesafety?

DataSet

- Datasets can be thought of as typed distributed collections of data.
- Dataset API unifies the DataFrame and RDD APIs.
- You can consider a `DataFrame` as an alias for `Dataset[Row]`, where a `Row` is a generic untyped JVM object.

```
type DataFrame = Dataset[Row]
```



[http://why-not-learn-something.blogspot.com/2016/07/apache-spark-rdd-vs-dataframe-vs-dataset.html]

Creating DataSets

- To convert a sequence or an RDD to a Dataset, we can use `toDS()`.
- You can call `as[SomeCaseClass]` to convert the DataFrame to a Dataset.

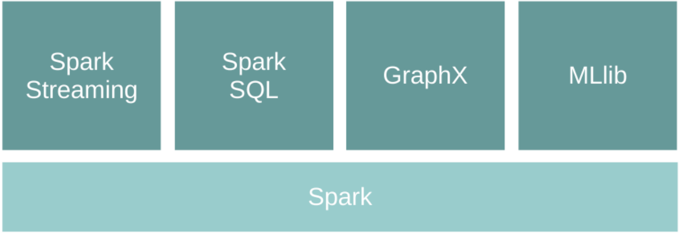
```
case class Person(name: String, age: BigInt, id: BigInt)

val personSeq = Seq(Person("Max", 33, 0), Person("Adam", 32, 1))

val ds1 = personSeq.toDS()

val ds2 = sc.parallelize(personSeq).toDS

val ds3 = spark.read.format("json").load("people.json").as[Person]
```

DataSet Transformations

- Transformations on Datasets are the same as those that we had on DataFrames.
- Datasets allow us to specify more complex and strongly typed transformations.

```
case class Person(name: String, age: BigInt, id: BigInt)

val people = spark.read.format("json").load("people.json").as[Person]

people.filter(x => x.age < 40).show()

people.map(x => (x.name, x.age + 5, x.id)).show()
```

GraphX


Spark Streaming
Spark SQL
GraphX
MLlib
Spark

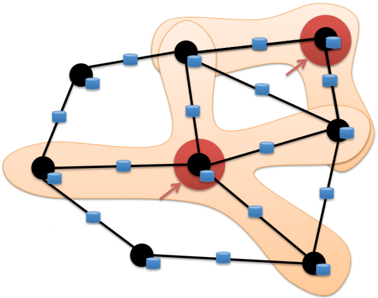

SOUNDCLOUD


You
Tube

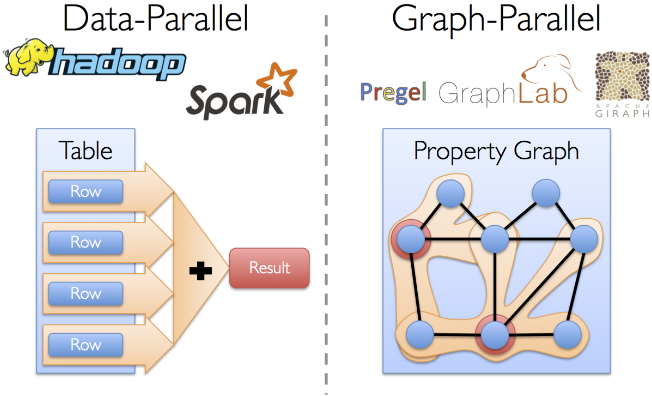
Graph Algorithms Challenges

- Difficult to extract parallelism based on partitioning of the data.
- Difficult to express parallelism based on partitioning of computation.

Graph-Parallel Processing Model



Data-Parallel vs. Graph-Parallel Computation

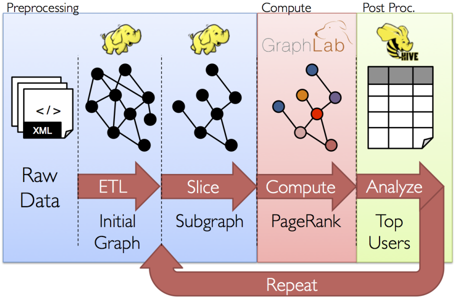


Motivation (2/3)

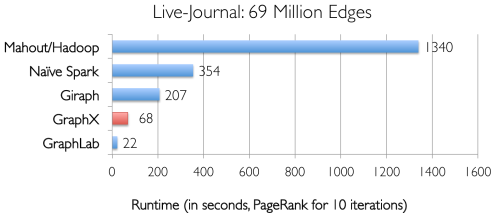
- Graph-parallel computation: restricting the types of computation to achieve performance.

Motivation (2/3)

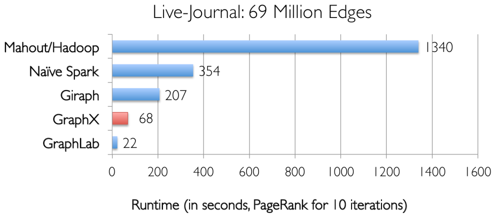
- Graph-parallel computation: restricting the types of computation to achieve performance.
- The same restrictions make it difficult and inefficient to express many stages in a typical graph-analytics pipeline.

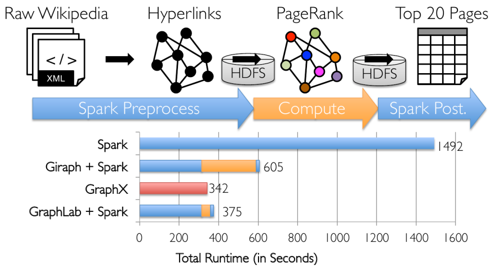


Motivation (3/3)

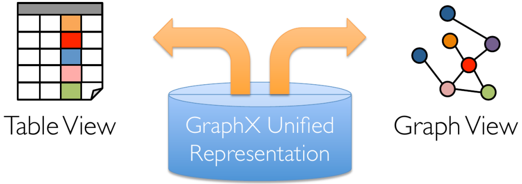


Motivation (3/3)



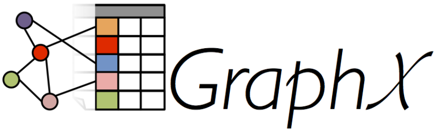


- Unifies data-parallel and graph-parallel systems.
- Tables and Graphs are composable views of the same physical data.



GraphX

- GraphX is the library to perform graph-parallel processing in Spark.
- In-memory caching.
- Lineage-based fault tolerance.

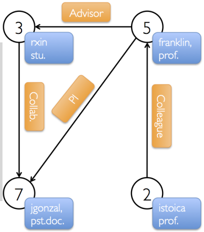


The Property Graph Data Model

- Spark represent graph structured data as a property graph.
- It is logically represented as a pair of vertex and edge property collections.
    - `VertexRDD` and `EdgeRDD`

```scala
// VD: the type of the vertex attribute
// ED: the type of the edge attribute
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

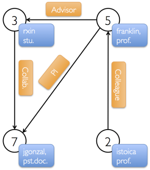
SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

The Vertex Collection

- `VertexRDD`: contains the vertex properties keyed by the vertex ID.

```scala
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}

// VD: the type of the vertex attribute
abstract class VertexRDD[VD] extends RDD[(VertexId, VD)]
```

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

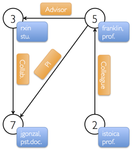
SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Vertices:

The Edge Collection

- EdgeRDD: contains the edge properties keyed by the source and destination vertex IDs.

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}

// ED: the type of the edge attribute
case class Edge[ED](srcId: VertexId, dstId: VertexId, attr: ED)
abstract class EdgeRDD[ED] extends RDD[Edge[ED]]
```

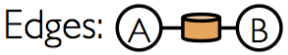


Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

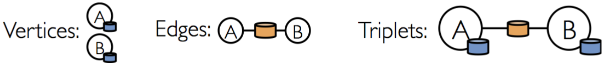


The Triplet Collection

- The triplets collection consists of each edge and its corresponding source and destination vertex properties.

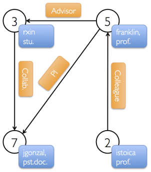
- It logically joins the vertex and edge properties: `RDD[EdgeTriplet[VD, ED]]`.

- The `EdgeTriplet` class extends the `Edge` class by adding the `srcAttr` and `dstAttr` members, which contain the source and destination properties respectively.



Building a Property Graph

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

```scala
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

val users: RDD[(VertexId, (String, String))] = sc.parallelize(Array((3L, ("rxin", "student")),
  (7L, ("jgonzal", "postdoc")), (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

val relationships: RDD[Edge[String]] = sc.parallelize(Array(Edge(3L, 7L, "collab"),
  Edge(5L, 3L, "advisor"), Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi"), Edge(5L, 1L, "-")))

val defaultUser = ("John Doe", "Missing")

val graph: Graph[(String, String), String] = Graph(users, relationships, defaultUser)
```

Graph Operators

- Information about the graph
- Property operators
- Structural operators
- Joins
- Aggregation
- ...

Information About The Graph (1/2)

- Information about the graph

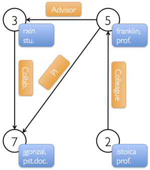
```scala
val numEdges: Long
val numVertices: Long
val inDegrees: VertexRDD[Int]
val outDegrees: VertexRDD[Int]
val degrees: VertexRDD[Int]
```

- Views of the graph as collections

```scala
val vertices: VertexRDD[VD]
val edges: EdgeRDD[ED]
val triplets: RDD[EdgeTriplet[VD, ED]]
```

Information About The Graph (2/2)

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

```scala
// Constructed from above
val graph: Graph[(String, String), String]

// Count all users which are postdocs
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count

// Count all the edges where src > dst
graph.edges.filter(e => e.srcId > e.dstId).count
```

Property Operators

- Transform vertex and edge attributes
- Each of these operators yields a new graph with the vertex or edge properties modified by the user defined `map` function.

```
def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
```

Property Operators

- Transform vertex and edge attributes
- Each of these operators yields a new graph with the vertex or edge properties modified by the user defined `map` function.

```
def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
```

```
val relations: RDD[String] = graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1)
relations.collect.foreach(println)
```

Property Operators

- Transform vertex and edge attributes
- Each of these operators yields a new graph with the vertex or edge properties modified by the user defined `map` function.

```
def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
```

```
val relations: RDD[String] = graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1)
relations.collect.foreach(println)
```

```
val newGraph = graph.mapTriplets(triplet =>
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1)
newGraph.edges.collect.foreach(println)
```

Structural Operators

- `reverse` returns a new graph with all the edge directions reversed.

```
def reverse: Graph[VD, ED]
def subgraph(epred: EdgeTriplet[VD, ED] => Boolean, vpred: (VertexId, VD) => Boolean):
    Graph[VD, ED]
def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
```

Structural Operators

- `reverse` returns a new graph with all the edge directions reversed.
- `subgraph` takes vertex/edge predicates and returns the graph containing only the vertices/edges that satisfy the given predicate.

```
def reverse: Graph[VD, ED]
def subgraph(epred: EdgeTriplet[VD, ED] => Boolean, vpred: (VertexId, VD) => Boolean):
    Graph[VD, ED]
def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
```

```
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
graph.vertices.collect.foreach(println)
validGraph.vertices.collect.foreach(println)

// Restrict the answer to the valid subgraph
val validUserGraph = graph.mask(validGraph)
```

Structural Operators

- `reverse` returns a new graph with all the edge directions reversed.
- `subgraph` takes vertex/edge predicates and returns the graph containing only the vertices/edges that satisfy the given predicate.
- `mask` constructs a subgraph of the input graph.

```scala
def reverse: Graph[VD, ED]
def subgraph(epred: EdgeTriplet[VD, ED] => Boolean, vpred: (VertexId, VD) => Boolean):
    Graph[VD, ED]
def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
```

```scala
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
graph.vertices.collect.foreach(println)
validGraph.vertices.collect.foreach(println)

// Restrict the answer to the valid subgraph
val validUserGraph = graph.mask(validGraph)
```

Join Operators

- `joinVertices` joins the vertices with the input RDD.
  - Returns a new graph with the vertex properties obtained by applying the user defined `map` function to the result of the joined vertices.
  - Vertices without a matching value in the RDD retain their original value.

```
def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD): Graph[VD, ED]
```

Join Operators

- `joinVertices` joins the vertices with the input RDD.
  - Returns a new graph with the vertex properties obtained by applying the user defined `map` function to the result of the joined vertices.
  - Vertices without a matching value in the RDD retain their original value.

```
def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD): Graph[VD, ED]
```

```
val rdd: RDD[(VertexId, String)] = sc.parallelize(Array((3L, "phd")))
val joinedGraph = graph.joinVertices(rdd)((id, user, role) => (user._1, role + " " + user._2))
joinedGraph.vertices.collect.foreach(println)
```

Aggregation (1/2)

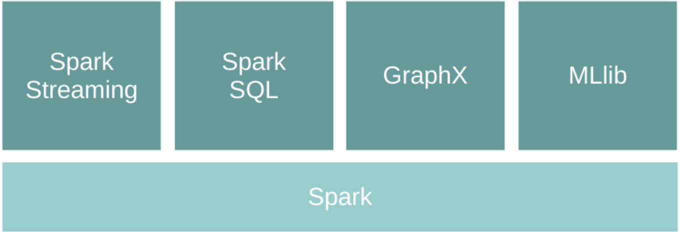
- aggregateMessages applies a user defined sendMsg function to each edge triplet in the graph and then uses the mergeMsg function to aggregate those messages at their destination vertex.

```
def aggregateMessages[Msg: ClassTag](
  sendMsg: EdgeContext[VD, ED, Msg] => Unit, // map
  mergeMsg: (Msg, Msg) => Msg, // reduce
  tripletFields: TripletFields = TripletFields.All):
  VertexRDD[Msg]
```

Aggregation (2/2)

```
// count and list the name of friends of each user
val profs: VertexRDD[(Int, String)] = validUserGraph.aggregateMessages[(Int, String)](
  // map
  triplet => {
    triplet.sendToDst((1, triplet.srcAttr._1))
  },
  // reduce
  (a, b) => (a._1 + b._1, a._2 + " " + b._2)
)

profs.collect.foreach(println)
```

Spark Streaming


Spark Streaming
Spark SQL
GraphX
MLlib
Spark

Stream Processing (1/4)

- Stream processing is the act of continuously incorporating new data to compute a result.

Stream Processing (2/4)

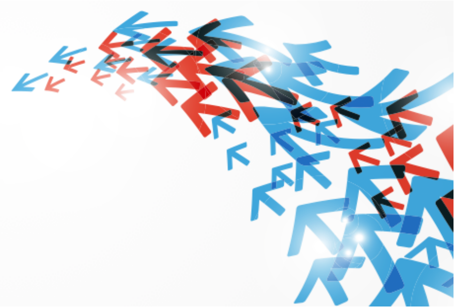
- The input data is unbounded.
  - A series of events, no predetermined beginning or end.

Stream Processing (2/4)

- The input data is unbounded.
  - A series of events, no predetermined beginning or end.
  - E.g., credit card transactions, clicks on a website, or sensor readings from IoT devices.

Stream Processing (3/4)

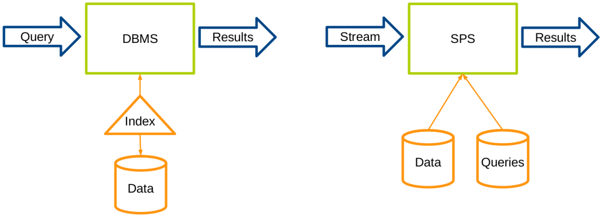
- User applications can then compute various queries over this stream of events.
  - E.g., tracking a running count of each type of event or aggregating them into hourly windows



Stream Processing (4/4)

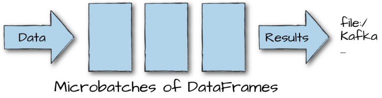
- Database Management Systems (DBMS): data-at-rest analytics
  - Store and index data before processing it.
  - Process data only when explicitly asked by the users.

- Stream Processing Systems (SPS): data-in-motion analytics
  - Processing information as it flows, without storing them persistently.



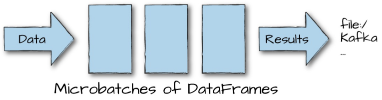
Streaming Data Processing Patterns

- Micro-batch systems
  - Batch engines
  - Slicing up the unbounded data into a sets of bounded data, then process each batch.

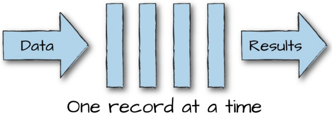


Streaming Data Processing Patterns

- Micro-batch systems
  - Batch engines
  - Slicing up the unbounded data into a sets of bounded data, then process each batch.



- Continuous processing-based systems
  - Each node in the system continually listens to messages from other nodes and outputs new updates to its child nodes.



Spark Streaming

- Run a streaming computation as a series of very small, deterministic batch jobs.

Spark Streaming

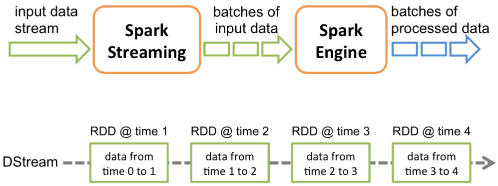
- Run a streaming computation as a series of very small, deterministic batch jobs.
    - Chops up the live stream into batches of X seconds.
    - Treats each batch as RDDs and processes them using RDD operations.

Spark Streaming

- Run a streaming computation as a series of very small, deterministic batch jobs.
  - Chops up the live stream into batches of X seconds.
  - Treats each batch as RDDs and processes them using RDD operations.
  - Discretized Stream Processing (DStream)

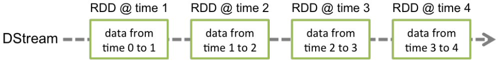
DStream (1/2)

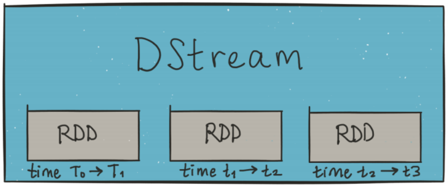
- DStream: sequence of RDDs representing a stream of data.



DStream (1/2)

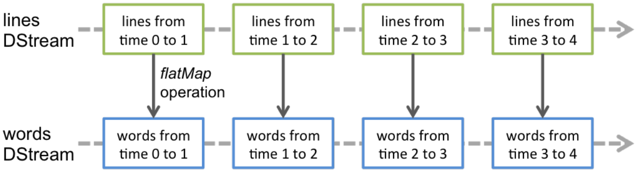
- DStream: sequence of RDDs representing a stream of data.





DStream (2/2)

- Any operation applied on a DStream translates to operations on the underlying RDDs.



StreamingContext

- StreamingContext is the main entry point of all Spark Streaming functionality.
- The second parameter, Seconds(1), represents the time interval at which streaming data will be divided into batches.

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

StreamingContext

- StreamingContext is the main entry point of all Spark Streaming functionality.
- The second parameter, Seconds(1), represents the time interval at which streaming data will be divided into batches.

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

- It can also be created from an existing SparkContext object.

```
val sc = ... // existing SparkContext
val ssc = new StreamingContext(sc, Seconds(1))
```

Input Operations

- Every input DStream is associated with a `Receiver` object.
  - It receives the data from a source and stores it in Spark's memory for processing.

Input Operations

- Every input DStream is associated with a `Receiver` object.
  - It receives the data from a source and stores it in Spark's memory for processing.
- Three categories of streaming sources:
  1. Basic sources directly available in the `StreamingContext` API, e.g., file systems, socket connections.
  2. Advanced sources, e.g., Kafka, Flume, Kinesis, Twitter.
  3. Custom sources, e.g., user-provided sources.

Input Operations - Basic Sources

- Socket connection
  - Creates a DStream from text data received over a TCP socket connection.

```
ssc.socketTextStream("localhost", 9999)
```

Input Operations - Basic Sources

- Socket connection
  - Creates a DStream from text data received over a TCP socket connection.

```
ssc.socketTextStream("localhost", 9999)
```

- File stream
  - Reads data from files.

```
streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)

streamingContext.textFileStream(dataDirectory)
```

Input Operations - Advanced Sources

- Connectors with external sources
- Twitter, Kafka, Flume, Kinesis, ...

```
TwitterUtils.createStream(ssc, None)

KafkaUtils.createStream(ssc, [ZK quorum], [consumer group id], [number of partitions])
```

Transformations (1/4)

- Transformations on DStreams are still lazy!
- Now instead, computation is kicked off explicitly by a call to the `start()` method.
- DStreams support many of the transformations available on normal Spark RDDs.

Transformations (2/4)

- `map`
  - Returns a new DStream by passing each element of the source DStream through a given function.

Transformations (2/4)

- `map`
  - Returns a new DStream by passing each element of the source DStream through a given function.

- `flatMap`
  - Similar to `map`, but each input item can be mapped to 0 or more output items.

Transformations (2/4)

- `map`
  - Returns a new DStream by passing each element of the source DStream through a given function.

- `flatMap`
  - Similar to `map`, but each input item can be mapped to 0 or more output items.

- `filter`
  - Returns a new DStream by selecting only the records of the source DStream on which func returns true.

Transformations (3/4)

- `count`
  - Returns a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.

Transformations (3/4)

- `count`
  - Returns a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
- `union`
  - Returns a new DStream that contains the union of the elements in two DStreams.

Transformations (4/4)

- `reduce`
  - Returns a new DStream of single-element RDDs by aggregating the elements in each RDD using a given function.

Transformations (4/4)

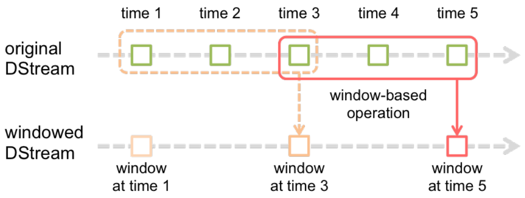
- `reduce`
  - Returns a new DStream of single-element RDDs by aggregating the elements in each RDD using a given function.
- `reduceByKey`
  - Returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.

Transformations (4/4)

- `reduce`
  - Returns a new DStream of single-element RDDs by aggregating the elements in each RDD using a given function.
- `reduceByKey`
  - Returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.
- `countByValue`
  - Returns a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.

Window Operations (1/3)

- Spark provides a set of transformations that apply to a over a sliding window of data.
- A window is defined by two parameters: window length and slide interval.
- A tumbling window effect can be achieved by making slide interval = window length



Window Operations (2/3)

- `window(windowLength, slideInterval)`
  - Returns a new DStream which is computed based on windowed batches.

Window Operations (2/3)

- `window(windowLength, slideInterval)`
  - Returns a new DStream which is computed based on windowed batches.
- `countByWindow(windowLength, slideInterval)`
  - Returns a sliding window count of elements in the stream.

Window Operations (2/3)

- `window(windowLength, slideInterval)`
  - Returns a new DStream which is computed based on windowed batches.
- `countByWindow(windowLength, slideInterval)`
  - Returns a sliding window count of elements in the stream.
- `reduceByWindow(func, windowLength, slideInterval)`
  - Returns a new single-element DStream, created by aggregating elements in the stream over a sliding interval using func.

Window Operations (3/3)

- `reduceByKeyAndWindow(func, windowLength, slideInterval)`
  - Called on a DStream of (K, V) pairs.
  - Returns a new DStream of (K, V) pairs where the values for each key are aggregated using function func over batches in a sliding window.

Window Operations (3/3)

- `reduceByKeyAndWindow(func, windowLength, slideInterval)`
  - Called on a DStream of (K, V) pairs.
  - Returns a new DStream of (K, V) pairs where the values for each key are aggregated using function func over batches in a sliding window.

- `countByValueAndWindow(windowLength, slideInterval)`
  - Called on a DStream of (K, V) pairs.
  - Returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window.

Word Count in Spark Streaming

Word Count in Spark Streaming (1/6)

- First we create a `StreamingContex`

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

// Create a local StreamingContext with two working threads and batch interval of 1 second.
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

Word Count in Spark Streaming (2/6)

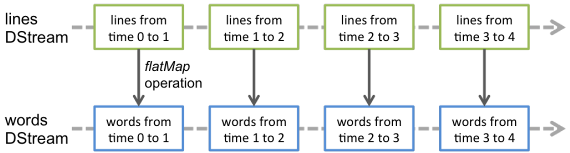
- Create a DStream that represents streaming data from a TCP source.
- Specified as hostname (e.g., localhost) and port (e.g., 9999).

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

Word Count in Spark Streaming (3/6)

- Use `flatMap` on the stream to split the records text to words.
- It creates a new DStream.

```
val words = lines.flatMap(_.split(" "))
```



Word Count in Spark Streaming (4/6)

- Map the `words` DStream to a DStream of `(word, 1)`.
- Get the frequency of words in each batch of data.
- Finally, print the result.

```
val pairs = words.map(word => (word, 1))

val wordCounts = pairs.reduceByKey(_ + _)

wordCounts.print()
```

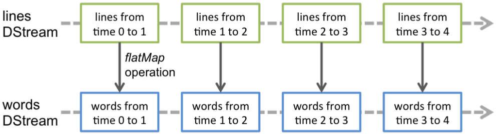
Word Count in Spark Streaming (5/6)

- Start the computation and wait for it to terminate.

```
// Start the computation
ssc.start()

// Wait for the computation to terminate
ssc.awaitTermination()
```

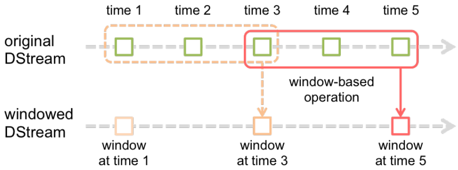
Word Count in Spark Streaming (6/6)

```
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))

val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()

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

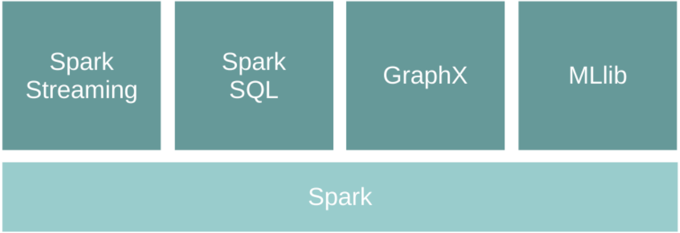


Word Count with Window

```
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))

val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val windowedWordCounts = pairs.reduceByKeyAndWindow(_ + _, Seconds(30), Seconds(10))
windowedWordCounts.print()

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



Summary


Spark Streaming
Spark SQL
GraphX
MLlib
Spark

Questions?