



Introduction to Apache Spark

HBD

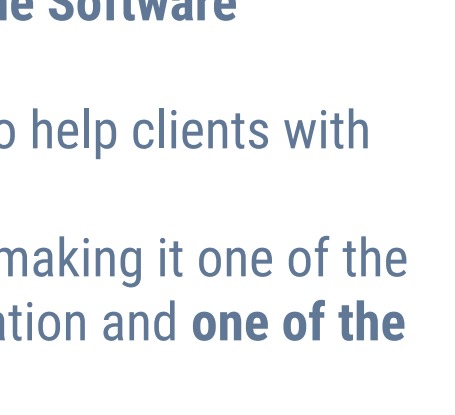


Sales Points





Apache Spark

- » **Apache Spark™** is a fast and general engine for large-scale data processing.
 - » **Started** in 2009 **as a research project** in the UC Berkeley RAD Lab, later to become the AMPLab.
 - » The Spark codebase was later **donated to the Apache Software Foundation**, which has maintained it since.
 - » In 2013 the creators of Spark **founded Databricks** to help clients with cloud-based big data processing using Spark
 - » Un 2015 Spark **had in excess of 1000 contributors** making it one of the most active projects in the Apache Software Foundation and **one of the most active open source big data projects.**
- 



Spark API

Unified API in Scala, Java, R or Python for multiple kinds of data processing:

- » Batch (high throughput).
- » Interactive (low latency)
- » Stream (continuous processing)
- » Iterative (multiple passes over the same data)

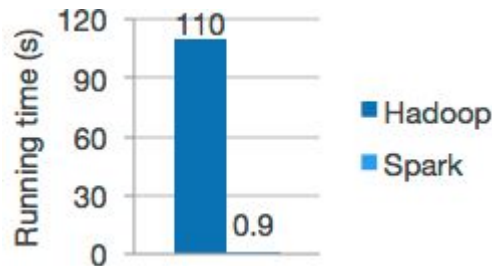
This provides **generality and ease of use**.



Spark Speed

Speed: Provides **in-memory primitives** which make it **faster than** competing technologies like **Hadoop** and Storm.

» Logistic regression: Spark vs Hadoop:



» Daytona Gray Sort Benchmark (sort 100TB of data)

	Machines	Cores	Time
Hadoop (2013)	2,100 (physical)	50,400	72 min
Spark (2014)	207 (AWS, virtualized)	6,592	23 min

3x faster using almost 10x less cores and more than **10x fewer machines**

Spark Environments

You can run Spark:

- » using its standalone cluster mode.
- » on EC2.
- » on Hadoop YARN.
- » on Apache Mesos.



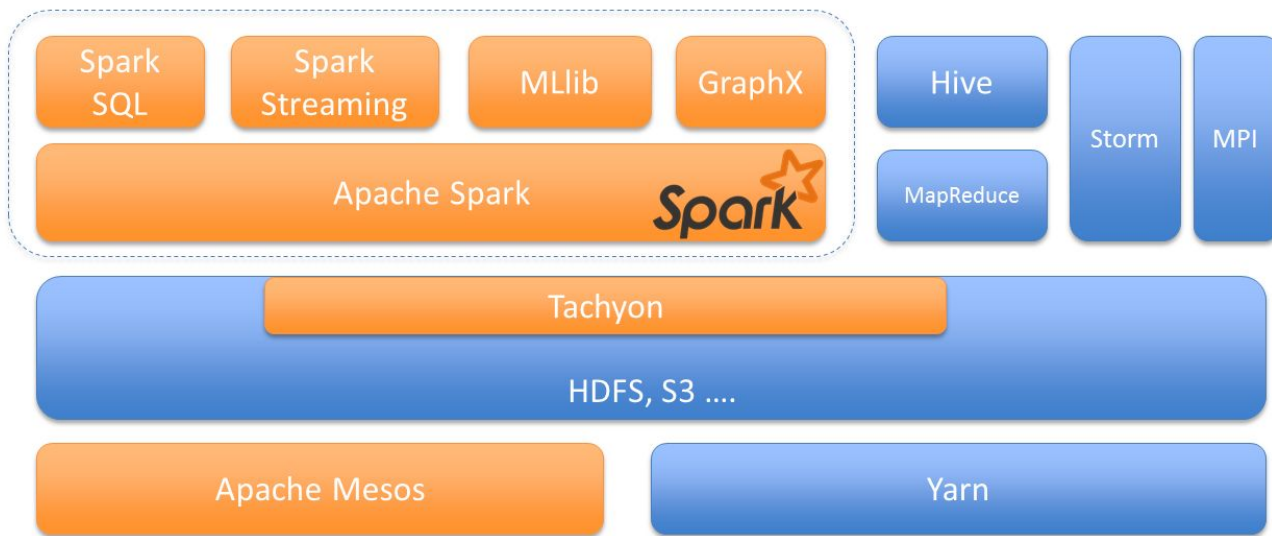
You can Access data in:

- » HDFS
- » Cassandra
- » HBase
- » Hive
- » Tachyon
- » any Hadoop data source.



Spark Ecosystem

APIs for R, Scala, Java and Python





Spark Core

Spark Word Count

```
val sc = new SparkContext(new SparkConf().setAppName("Spark  
Count"))
```

Context

Resilient distributed
datasets (RDDs)

```
// read in text file and  
val lines = sc.textFile("path")
```

```
// split each document into words  
val words = lines.flatMap(line => line.split(" ")).map((_, 1))
```

Transformation

```
// count the occurrence of each word
```


```
val wordCounts = words.reduceByKey(_ + _)
```

Action




Spark: Run modes

Spark can be run:

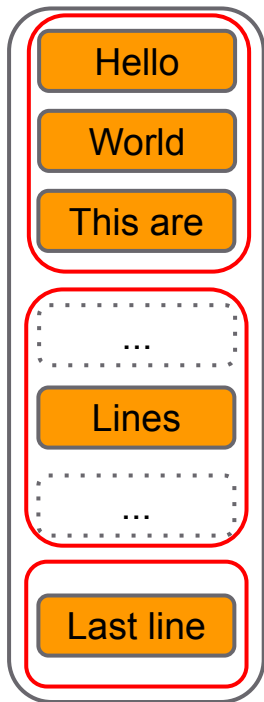
- » from an interactive shell: **./spark-shell**.
 - » as a script run from the shell: **./spark-shell -f script**.
 - » as a self contained application in any language.
 - » a module as part of a bigger application.
 - » Inside notebooks
- 



SparkContext and SparkConf

- » The first thing a Spark program must do is to create a **SparkContext object**, which **tells** Spark **how to access a cluster**.
 - » To create a SparkContext **you first need to build a SparkConf** object that **contains information about your application**.
 - » **Only one SparkContext may be active per JVM**. You must stop() the active SparkContext before creating a new one.
 - » A SparkContext is used to create RDDs and manage operations on the cluster.
- 

RDD: Resilient Distributed Dataset



Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark.

An RDD is:

- » **Immutable** collection of objects.
- » Partitioned and **distributed**.
- » Stored **in memory** (can spill to disk if permitted).
- » **Fault Tolerant:** Partitions recompute with failure.
- » **Provides lazy execution:** nothing done until action is invoked.



RDD: Creation

There are several ways to create an RDD:

- » Create from an scala array or scala sequence:

```
val rdd = sc.parallelize(1 to 1000)
```

- » Create from a text file:

```
val words = sc.textFile("README.md")
```

- » Create from other sources:

```
val names = sc.hadoopRDD("hdfs://data/names.avro")
```

- » Apply a transformation to an existing RDD:
- 



RDD Operations

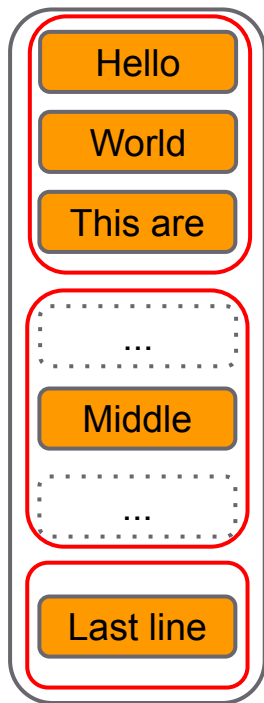
RDD provide 2 types of operations to process their data:

- » **Transformation:** return a **new RDD** with the values of applying . They are **lazy**. their result is not immediately computed.
- » **Actions:** Compute a result based on the RDD, and either returned or saved to an external storage system (e. g. HDFS). They are **eager**, their result is immediately computed.

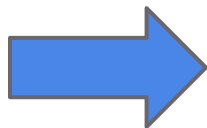
Lazy/eagerness is how we can limit network communication using the programming model.



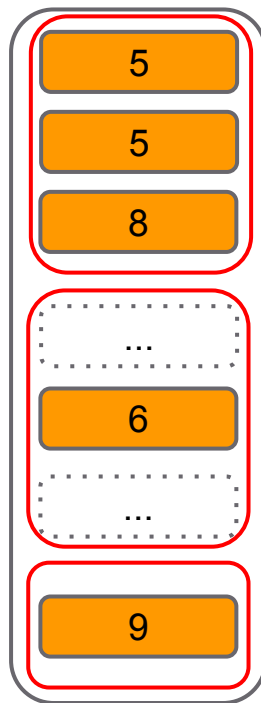
RDD Operations



Transformation



`map(_.length())`



Action



`reduce(_+_)`

33



RDD Transformation

Transformation

Meaning

map

Apply function to each element in the RDD.

filter

Return a new RDD containing only the elements that satisfy a predicate.

flatMap

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

mapPartitions

Similar to map, but runs separately on each partition (block) of the RDD.

sample

Return a sampled subset of this RDD.

distinct

return a new RDD with duplicates removed

[RDD scaladoc](#)






RDD Transformation

There are some transformation that apply to 2 RDDs.

Transformation	Meaning
union	Return the union of this RDD and another one.
subtract	Return an RDD with the elements from this that are not in other.
cartesian	Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of elements (a, b) where a is in this and b is in other.





RDD Actions

Actions

Meaning

carefull

collect

Return an array that contains all of the elements in this RDD.

count

Return the number of elements in the RDD.

take

Take the first num elements of the RDD.

reduce

Reduces the elements of this RDD using the specified commutative and associative binary operator.

foreach

Applies a function to all elements of this RDD.

takeSample

Returns an array of K elements randomly selected.

takeOrdered

Returns the first K elements from this RDD as defined by the specified Order.





RDD Save Actions

Some actions are to store the RDD content.

Actions

Meaning

saveAsTextFile

Save this RDD as a compressed text file, using string representations of elements.

saveAsObjectFile

Save this RDD as a SequenceFile of serialized objects.

Save action generate a file for each partition in the RDD.





Pair RDDs

- » Often when working with distributed data, it's useful to organize data into **key-values pairs**.
- » In Spark distributed key-value pairs are "**Pair RDDs**".
- » PairRDDs allow to work with each key in parallel or regroup data on the network.
- » PairRDDs are usually created as a result of a map operation on a "normal RDD".

```
val pairRdd = rdd.map(user => (user.id, user.name))
```

- » PairRDD provide specific operations to process.
- 



Pair RDD Transformation

For efficiency
prefer *reduce*
before *group*

Transformation

Meaning

groupByKey

Group the values for each key in the RDD into a single sequence.

reduceByKey

Merge the values for each key using an associative and commutative reduce function.

mapValues


Pass each value in the key-value pair RDD through a map function without changing the keys; this also retains the original RDD's partitioning.

keys

Return an RDD with the keys of each tuple.

join

Return an RDD containing all pairs of elements with matching keys in this and other.





Pair RDD Actions

Actions

countByKey

Meaning

Group the values for each key in the RDD into a single sequence.


And all the reductions of a “normal” RDD





RDD Persistence and Fault Tolerance


One of the **most important capabilities** in Spark is **persisting (or caching)** a dataset in memory across operations.

- » **Each node stores any partitions** of it that it computes in memory **and reuses** them in other actions on that dataset (or datasets derived from it).
 - » You can mark an **RDD to be persisted using the `persist()` or `cache()`** methods on it.
 - » **If any partition of an RDD is lost, it will automatically be recomputed** using the transformations that originally created it.
- 



Spark Operations Example

```
val sc = new SparkContext("spark://...", "MyJob", home,  
jars)  
  
val file = sc.textFile("hdfs://...")  
  
val errors = file.filter(line => line.contains("ERROR"))  
  
errors.cache()  
  
errors.count()
```





Spark core internals

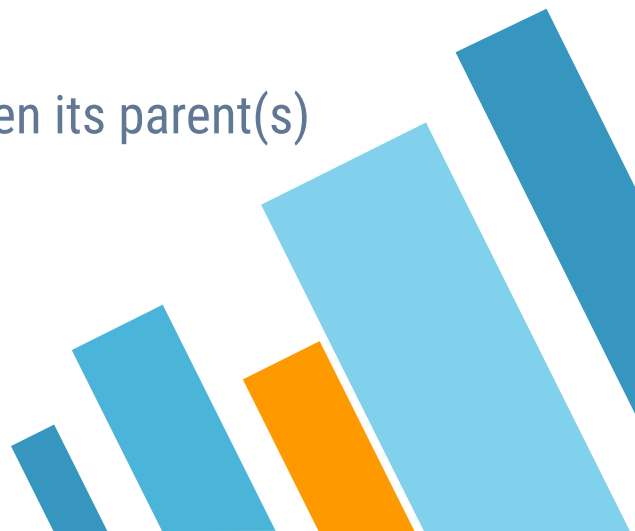
Or how it works under the hood





RDD: Internals

RDD is an INTERFACE: specific implementation depends on each backend.
Internally an RDD is composed of the following components:

- » Set of partitions (“splits” in Hadoop)
 - » List of dependencies on parent RDDs
 - » Function to compute a partition (as an Iterator) given its parent(s)
 - » (Optional) partitioner (hash, range)
 - » (Optional) preferred location(s) for each partition
- 

RDD: Dependencies and Partitions

Dataset-level view:

file:

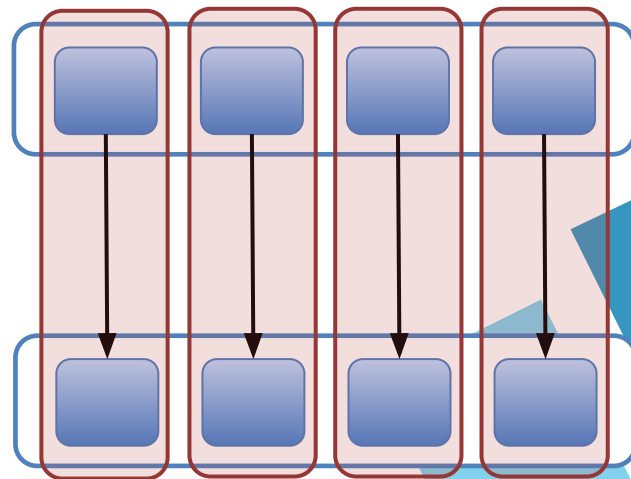
HadoopRDD
path = hdfs://...



errors:

FilteredRDD
func = `_.contains(...)`
shouldCache = true

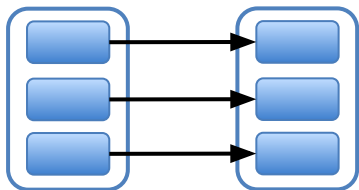
Partition-level view:



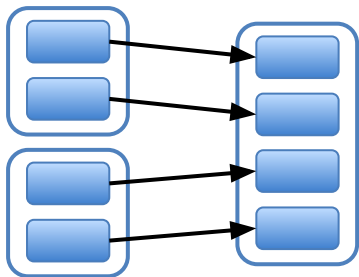
Task 1 Task 2 ...

RDD: Dependency Types

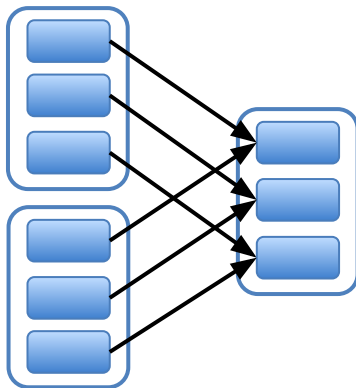
“Narrow” (pipeline-able):



map, filter

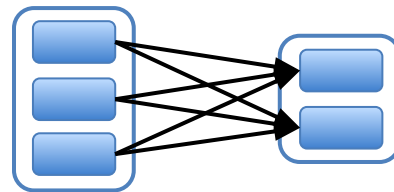


union

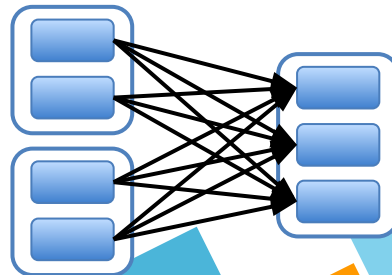


join with inputs
co-partitioned

“Wide” (shuffle):



groupByKey



join with inputs not
co-partitioned

Example: RDDs Components

HadoopRDD	
partitions	one per HDFS block
dependencies	none
compute	read corresponding block
partitioner	none
preferredLocations	HDFS block location

FilteredRDD	
partitions	same as parent RDD
dependencies	"one-to-one" on parent
compute	compute parent and filter it
partitioner	none
preferredLocations	none (ask parent)

Example: RDDs Components

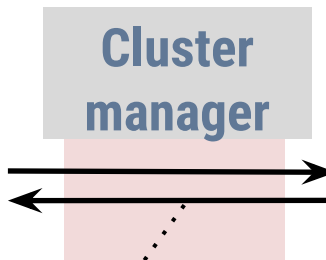
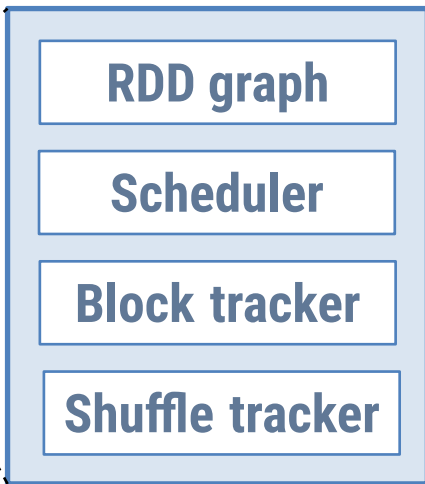
JoinedRDD	
partitions	one per reduce task
dependencies	"shuffle" on each parent
compute	read and join shuffled data
partitioner	HashPartitioner(numTasks)
preferredLocations	none

Spark Application

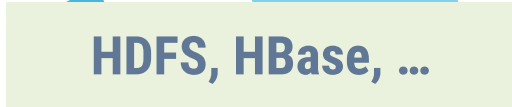
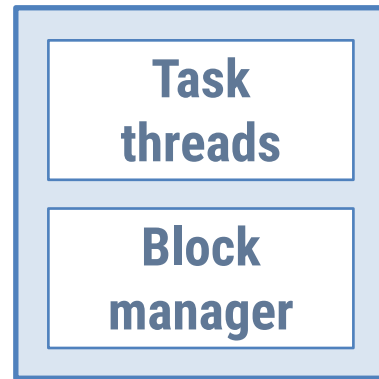
Spark client
(app master)

Your program

```
sc = new SparkContext  
f = sc.textFile(...)   
f.filter(...) .count()  
...
```



Spark worker



Code is serialized and sent to workers to execute on partitions.

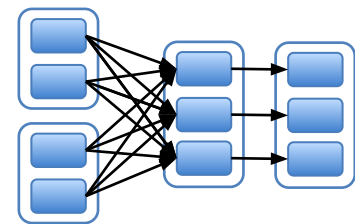
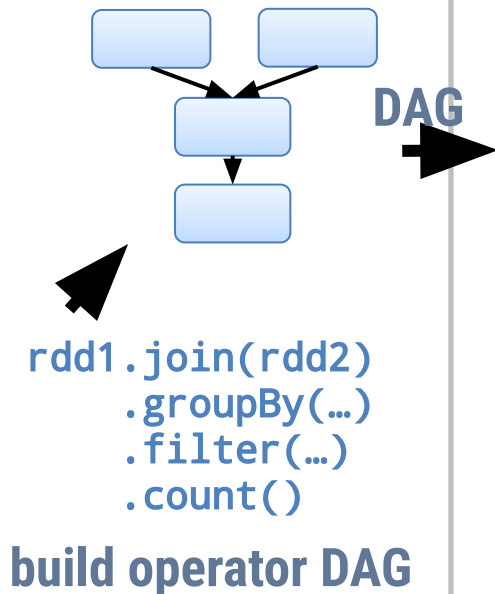
Job Scheduling Process

RDD Objects

DAG Scheduler

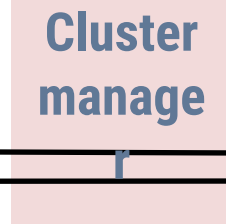
Task Scheduler

Worker



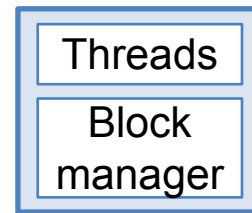
split graph into
stages of tasks
submit each
stage as ready

TaskSet



launch tasks via
cluster manager
retry failed or
straggling tasks

Task



execute tasks
store and serve
blocks




DAG Scheduler

Operation:

- » **Input:** RDD and partitions to compute
- » **Output:** Output from actions on those partitions

Roles:

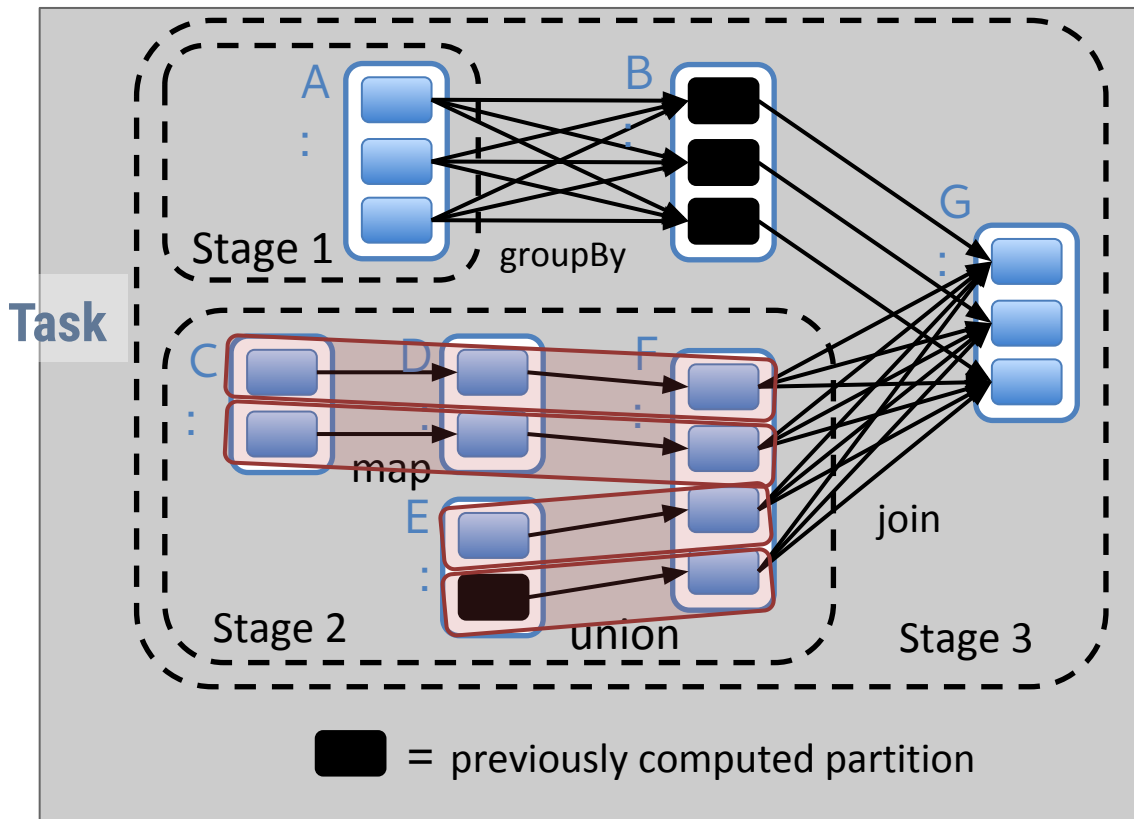
- » Build stages of tasks.
 - » Submit them to lower level scheduler (e.g. YARN, Mesos, Standalone) as ready.
 - » Lower level scheduler will schedule data based on locality.
 - » Resubmit failed stages if outputs are lost.
- 

Scheduler Optimizations

Pipelines operations within a stage.


Picks join algorithms based on partitioning (to minimize shuffles).

Reuses previously computed data.





Task

- » **Unit of work** to execute in an executor.
 - » Unlike MR, there is **no “map reduce”** task.
 - » Sparks **runs tasks in threads in the executor** (contains a thread pool). MapReduce runs task as standalone processes.
 - » Each task either:
 - partitions its output for “shuffle”.
 - send the output back to the driver.
- 

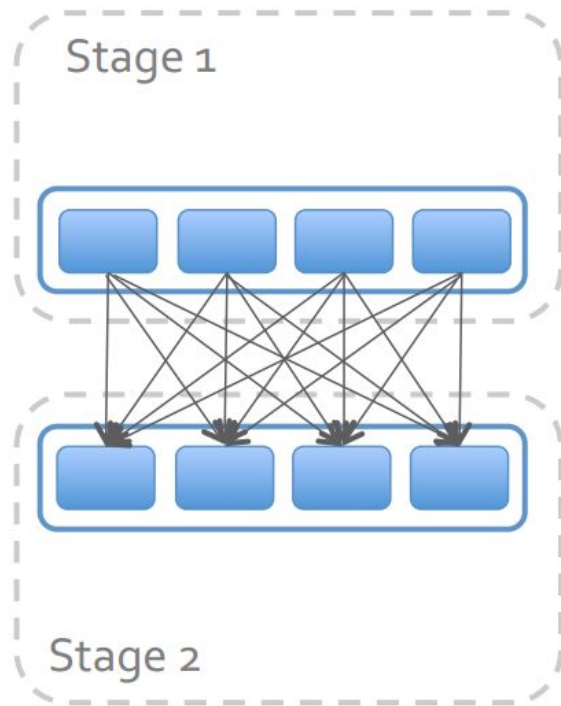
Shuffle

Redistributes data among partitions

Partition keys into buckets (user-defined partitioner)

Optimizations:

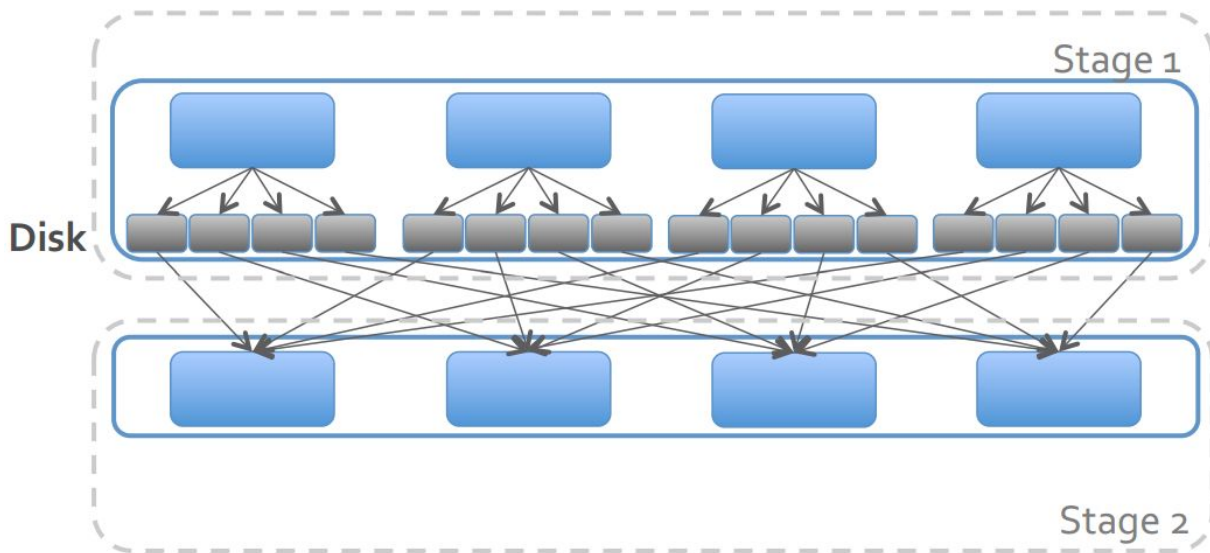
- » Avoided when possible, if data is already properly partitioned
- » Partial aggregation reduces data movement



Shuffle: Internals

Write intermediate files to disk


Fetches by the next stage of tasks (“reduce” in MR).





RDD Persistence

Storage Level	Description
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. Partitions are recomputed if they do not fit in memory.(default level)
MEMORY_AND_DISK	Store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects. More space-efficient than deserialized objects, but more CPU-intensive to read
MEMORY_AND_DISK_SER	Spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.





Performance Considerations

Main issue: too few partitions

- » Less concurrency.
- » More susceptible to data skew.
- » Increased memory pressure for group by

Secondary issue: too many partitions

- » Task take longer to schedule than to run

Reasonable number of partitions

- » Between 100 and 10,000 usually (small clusters near 100, big 10,000)
- » Lower bound: at least 2x the number of cores in the cluster
- » Upper bound: ensure each task takes at least 100ms

Minimize data shuffle.

Know the Spark standard API





Performance Considerations

- » **Split long-running jobs into batches** and write intermediate results to disk if cached data is filling memory.
- » Use **broadcast variables** (read-only variables that are cached in-memory locally on each machine) **to do efficient joins between large and small RDDs** or store a lookup table in memory that provides efficient retrieval.

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))
```

- » Use **accumulators** as a way to efficiently update a variable in parallel during execution.

```
val accum = sc.accumulator(0, "My Accumulator")
```

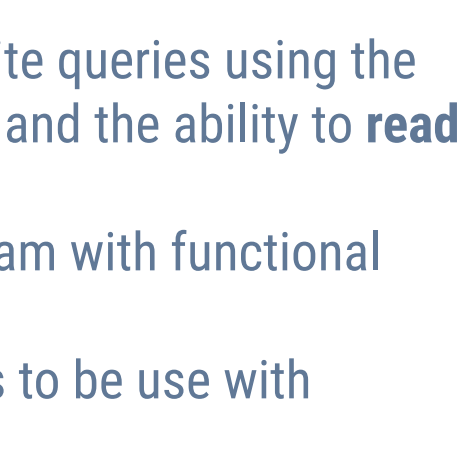
- » Configure Kryo serialization for your objects. (much quicker and efficient than default Java serialization).
- 



Spark SQL



Spark SQL

- » **Spark SQL is a Spark module for structured data processing.**
 - » It **provides** a 2 programming abstractions called **Datasets and DataFrames** and can also act as distributed SQL query engine.
 - » **The entry point** into all functionality in Spark SQL **is the SparkSession** (or SQLContext in Spark 1.X).
 - » **SparkSession** (HiveContext in 1.X) also provides write queries using the more complete **HiveQL parser, access to Hive UDFs**, and the ability to **read data from Hive tables**.
 - » Spark comes packaged **with spark-sql, a shell** program with functional similarities to Hive's CLI.
 - » Spark SQL also provides ODBC and JDBC connectors to be use with RDBMS.
- 

SparkSession

```
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._
```



DataFrames

- » A DataFrame is a **distributed collection of data organized into named columns**.
 - » It is **conceptually equivalent to a table in a relational database** or a data-frame in R/Python.
 - » DataFrames **can be constructed from a wide array of sources** such as: structured data files, tables in Hive, external databases, or existing RDDs.
 - » Before any computation on a DataFrame starts, **the Catalyst optimizer compiles the operations** that were used to build the DataFrame **into a physical plan for execution**.
 - » The **optimizer can push filter predicates** down into the data source.
 - » Catalyst generates JVM bytecode for those plans that is **often more optimized than hand-written code**.
- 

DataFrames

Data Frames can be created from files with semi-structured data (like json).

```
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|
```

```
// +---+-----+
```

DataFrames - Operations

DataFrames provide an [API](#) to make queries over the structured data.

```
import spark.implicit._ // This import is needed to use the $-notation
```

```
df.printSchema() // Print the schema in a tree format
```

```
df.select("name").show() // Select only the "name" column
```

```
df.select($"name", $"age" + 1).show() // Select everybody, but increment the age by 1
```

```
df.filter($"age" > 21).show() // Select people older than 21
```

```
df.groupBy("age").count().show() // Count people by age
```

```
// +-----+-----+
// |  name|(age + 1)|
// +-----+-----+
// |Michael|    null|
// |  Andy|     31|
// |  Justin|    20|
// +-----+-----+
```

DataFrames - SQL

DataFrames allow us to register a dataframe as a table and then run SQL queries programatically.

```
// 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|
```

```
// +---+-----+
```

DataFrame Inferring Schema

DataFrames can be created inferring the schema from case classes.

```
case class Person(name: String, age: Long) // you can use custom classes
```

```
// Create an RDD of Person objects from a text file, convert it to a Dataframe
```

```
val peopleDF = spark.sparkContext  
  .textFile("examples/src/main/resources/people.txt")  
  .map(_._split(","))  
  .map(attributes => Person(attributes(0), attributes(1).trim.toInt))  
  .toDF()
```

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

```
// SQL statements can be run by using the sql methods provided by Spark
```

```
val teenagersDF = spark.sql("SELECT name, age FROM people WHERE age BETWEEN 13 AND 19")  
teenagersDF.map(teenager => "Name: " + teenager(0)).show() // The columns of a row in the result can be accessed by field index  
teenagersDF.map(teenager => "Name: " + teenager.getAs[String]("name")).show() // or by field name
```


Programmatically Specifying the Schema

When case classes cannot be defined a schema can be created programmatically.

```
val peopleRDD = spark.sparkContext.textFile("examples/src/main/resources/people.txt") // Create an RDD
```

```
val schemaString = "name age" // The schema is encoded in a string
```

```
// Generate the schema based on the string of schema
```

```
val fields = schemaString.split(" ")
```

```
  .map(fieldName => StructField(fieldName, StringType, nullable = true))
```

```
val schema = StructType(fields)
```

```
// Convert records of the RDD (people) to Rows
```

```
val rowRDD = peopleRDD
```

```
  .map(_>split(","))
```

```
  .map(attributes => Row(attributes(0), attributes(1).trim))
```

```
// Apply the schema to the RDD
```

```
val peopleDF = spark.createDataFrame(rowRDD, schema)
```

DataFrames - Example (Spark pre V1.6)

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// this is used to implicitly convert an RDD to a DataFrame.
import sqlContext.implicits._

// Define the schema using a case class.
// Note: Case classes in Scala 2.10 can support only up to 22 fields. To work around this limit,
// you can use custom classes that implement the Product interface.
case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile("examples/src/main/resources/people.txt").map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt)).toDF()
people.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

// The results of SQL queries are DataFrames and support all the normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```



Dataset

- » A Dataset is a distributed **collection of typed data**.
- » **Dataset** is an interface added in Spark 1.6 that **provides the benefits of RDDs** (strong typing, ability to use powerful lambda functions) **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, filter, etc.) **or the SQL API**.
- » **DataFrames are actually an implementation of Datasets, this means that they share most of their API**

DataFrames = Dataset[Row]



Dataset Creation

Datasets can be created from Typed Collections

```
case class Person(name: String, age: Long) // you can use custom classes
```

```
val caseClassDS = Seq(Person("Andy", 32)).toDS() // Encoders are created for case classes
```

```
// Encoders for most common types are automatically provided by importing spark.implicits._
```

```
val primitiveDS = Seq(1, 2, 3).toDS()
```

```
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)
```

```
// DataFrames can be converted to a Dataset by providing a class.
```

```
// Mapping will be done by name
```

```
val path = "examples/src/main/resources/people.json"
```

```
val peopleDS = spark.read.json(path).as[Person]
```

```
peopleDS.show()
```

```
// +----+-----+
```

```
// | age|  name|
```

```
// +----+-----+
```

```
// |null|Michael|
```

```
// | 30|  Andy|
```

```
// | 19| Justin|
```

Datasets used with RDD operations

```
peopleDF.createOrReplaceTempView("people")
```

```
// SQL statements can be run by using the sql methods provided by Spark
```

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

```
teenagersDF.map(teenager => "Name: " + teenager(0)).show()
```

```
// or by field name
```

```
teenagersDF.map(teenager => "Name: " + teenager.getAs[String]("name")).show()
```

```
// No pre-defined encoders for Dataset[Map[K,V]], define explicitly
```

```
implicit val mapEncoder = org.apache.spark.sql.Encoders.kryo[Map[String, Any]]
```

```
// Primitive types and case classes can be also defined as
```

```
// implicit val stringIntMapEncoder: Encoder[Map[String, Any]] = ExpressionEncoder()
```

```
// row.getValuesMap[T] retrieves multiple columns at once into a Map[String, T]
```

```
teenagersDF.map(teenager => teenager.getValuesMap[Any](List("name", "age"))).collect()
```

```
// Array(Map("name" -> "Justin", "age" -> 19))
```

Load and Save Functions

Generic Load/Save Functions (Parquet is default)

```
val usersDF = spark.read.load("examples/src/main/resources/users.parquet")
usersDF.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

Manually Specifying Options

```
val peopleDF = spark.read.format("json").load("examples/src/main/resources/people.json")
peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```


Run SQL on files directly

```
val sqlDF = spark.sql("SELECT * FROM parquet.`examples/src/main/resources/users.parquet`")
```



Save Modes

SaveMode	Description
ErrorIfExists (Default)	If data already exists, an exception is expected to be thrown.
Append	If data/table already exists, contents are appended to existing data.
Overwrite	Existing data is expected to be overwritten by new data.
Ignore	If data already exists, the save operation is expected to not save the contents of the DataFrame and to not change the existing data.



Datasources API

Provides a single interface for loading and storing data using Spark SQL (since Spark 1.3).

Built-In

{ JSON }




External



and more...



Interacting with Hive Tables

- » Spark SQL also **supports reading and writing** data stored in **Apache Hive**.
 - » Since Hive has a large number of dependencies, **it is not included in the default Spark assembly**. (need to build with -Phive and -Phive-thriftserver flags)
 - » Configuration is done by placing your *hive-site.xml* file in Spark's conf/ dir.
 - » One must construct a HiveContext, which inherits from SQLContext, and adds support for finding tables in the MetaStore and writing queries using HiveQL.
- 

Interacting with Hive Tables

```
val warehouseLocation = "spark-warehouse" // warehouseLocation points to the default location for managed databases
```

```
val spark = SparkSession
```

```
.builder()
```

```
.appName("Spark Hive Example")
```

```
.config("spark.sql.warehouse.dir", warehouseLocation)
```

```
.enableHiveSupport()
```

```
.getOrCreate()
```

```
import spark.implicit._
```

```
import spark.sql
```

```
sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
```

```
sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")
```

```
// Queries are expressed in HiveQL
```

```
sql("SELECT * FROM src").show()
```



Spark SQL Optimizations

Catalyst: Query Optimizer

- » by knowing and understanding all data types
- » knowing the schema of our data.
- » and the computations we'd like to do


Catalyst can:

- » Reorder operations.
 - » Reduce the amount of data we must read.
 - » Pruning unneeded partitioning.
- 



Spark SQL Optimizations

Tungsten: off-heap data encoder

- » **uses highly-specialized data encoders:** from the schema it can pack serialized data into memory. This means faster ser/des
 - » **column-based:** allows faster aggregations, sorts and groups
 - » **off-heap:** avoids garbage collections overheads.
- 



Spark Streaming




Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.





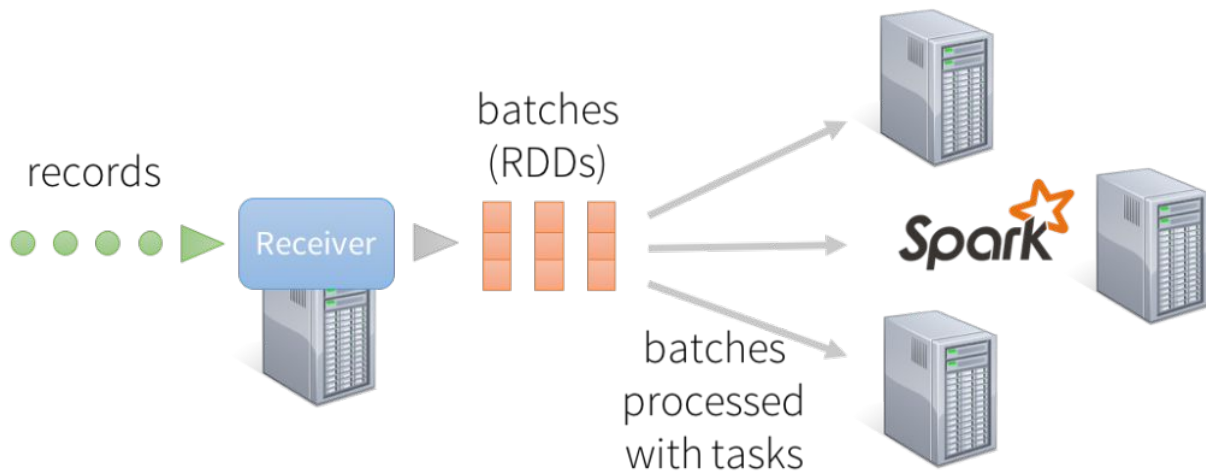
Spark Streaming Concepts

- » Spark Streaming **provides a high-level abstraction called discretized stream or DStream.**
 - » A DStream is represented as **a sequence of RDDs.**
 - » Each RDD in a DStream **contains data from a certain interval.**
 - » **Every input DStream is associated with a Receiver** object which receives the data from a source and stores it in Spark's memory for processing.
 - **Basic sources:** directly available to StreamingContext API. Example: file systems, socket.
 - **Advanced sources:** Sources like Kafka, Flume, Kinesis, Twitter, etc. are available through extra utility classes.
- 

Execution Model

Spark Streaming

discretized stream processing

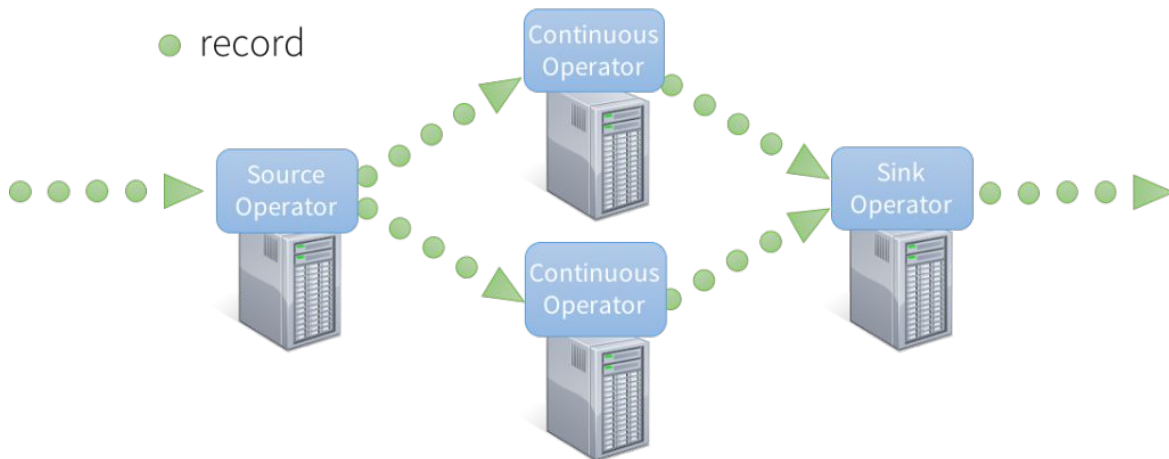


records processed in batches with short tasks
each batch is a RDD (partitioned dataset)

Execution Model

Traditional stream processing systems

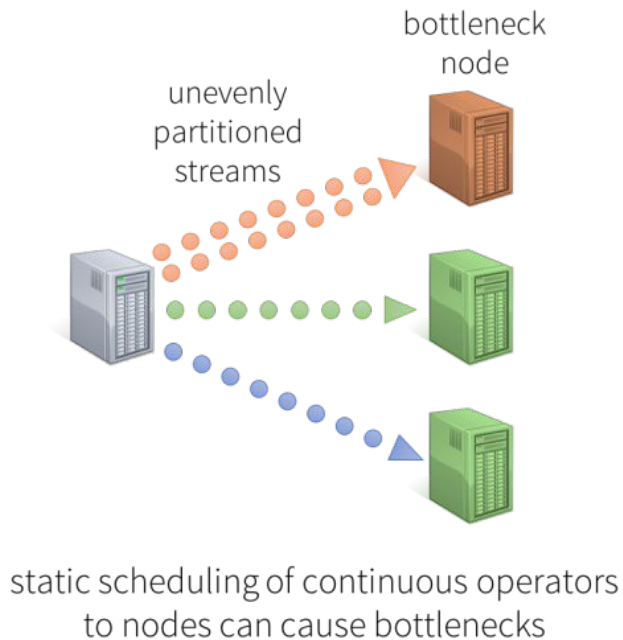
continuous operator model



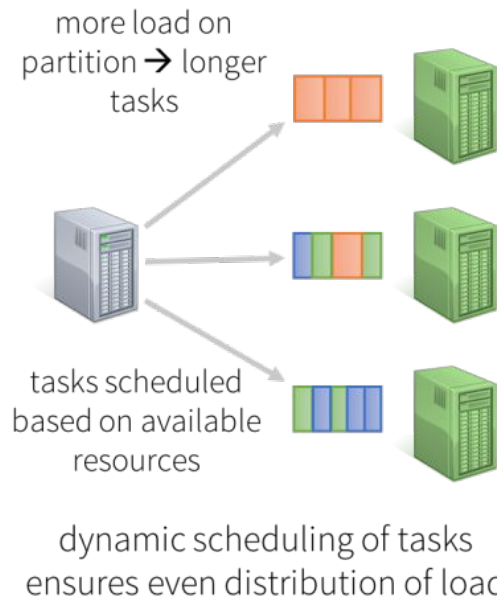
records processed one at a time

Execution Model

Traditional systems



Spark Streaming





Spark Streaming Concepts

Stateful Transformations:

- » Some of the data of previous batch is used to generate the results of the current batch.
- » Stateful operations like **updateStateByKey** and **reduceByKeyAndWindow** require a place to store this data.
- » Enabling checkpointing in the Spark Streaming Context provides a place to store this state information.

Windowed Operations:

- » The window duration controls how many previous batches of data are considered
- » Used for calculations like trending hashtags in Twitter.

Spark Streaming Example

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3

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

// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)

val words = lines.flatMap(_.split(" ")) // Split each line into words

// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print() // Print the first ten elements of each RDD generated in this DStream to the console

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



DSTREAM Transformation

Transformation

Meaning

map

Return a new DStream by passing each element of the source DStream through a function.

filter

Return a new DStream by selecting only the records of the source DStream on which a function returns true.

flatMap

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

count

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

countByValue

When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.

reduceByKey

When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function



[DStream scaladoc](#)



DSTREAM Transformation

Transformation

Meaning

map

Return a new DStream by passing each element of the source DStream through a function.

filter

Return a new DStream by selecting only the records of the source DStream on which a function returns true.

flatMap

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

count

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

countByValue

When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.

reduceByKey

When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function



[DStream scaladoc](#)



Other Spark Resources



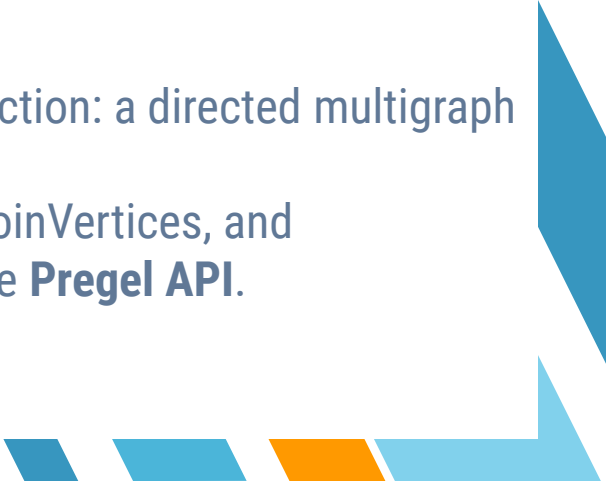


Machine Learning & Graph Processing

MLlib:

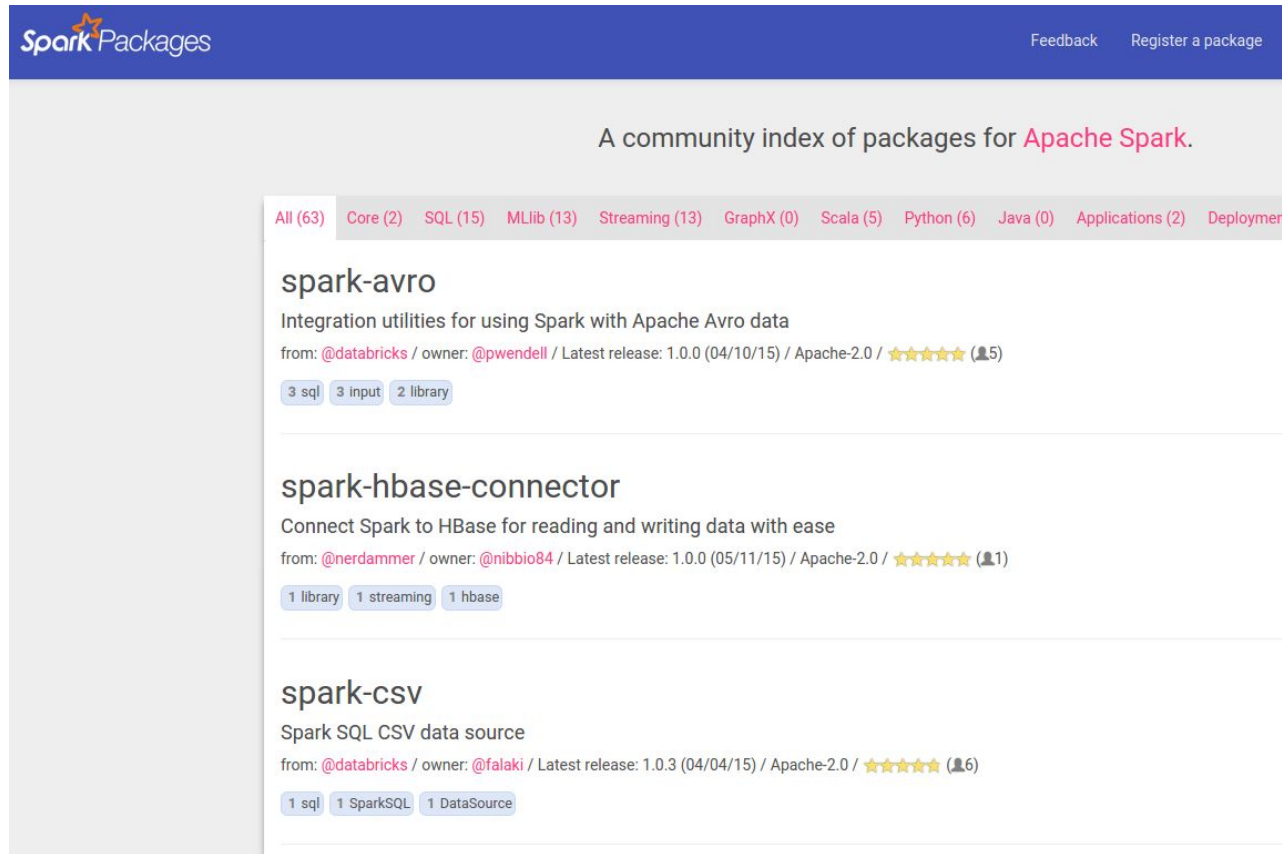
- » Scalable **machine learning library** consisting of **common learning algorithms and utilities**, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as **underlying optimization primitives**.

GraphX: scalable graphs and graph-parallel computation.

- » **Extends the Spark RDD** by introducing a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge.
 - » Exposes a set of **fundamental operators** (e.g., subgraph, joinVertices, and aggregateMessages) as well as an optimized variant of the **Pregel API**.
- 

spark-packages.org

A page to obtain
community provided
packages and
libraries to
complement a Spark
application.



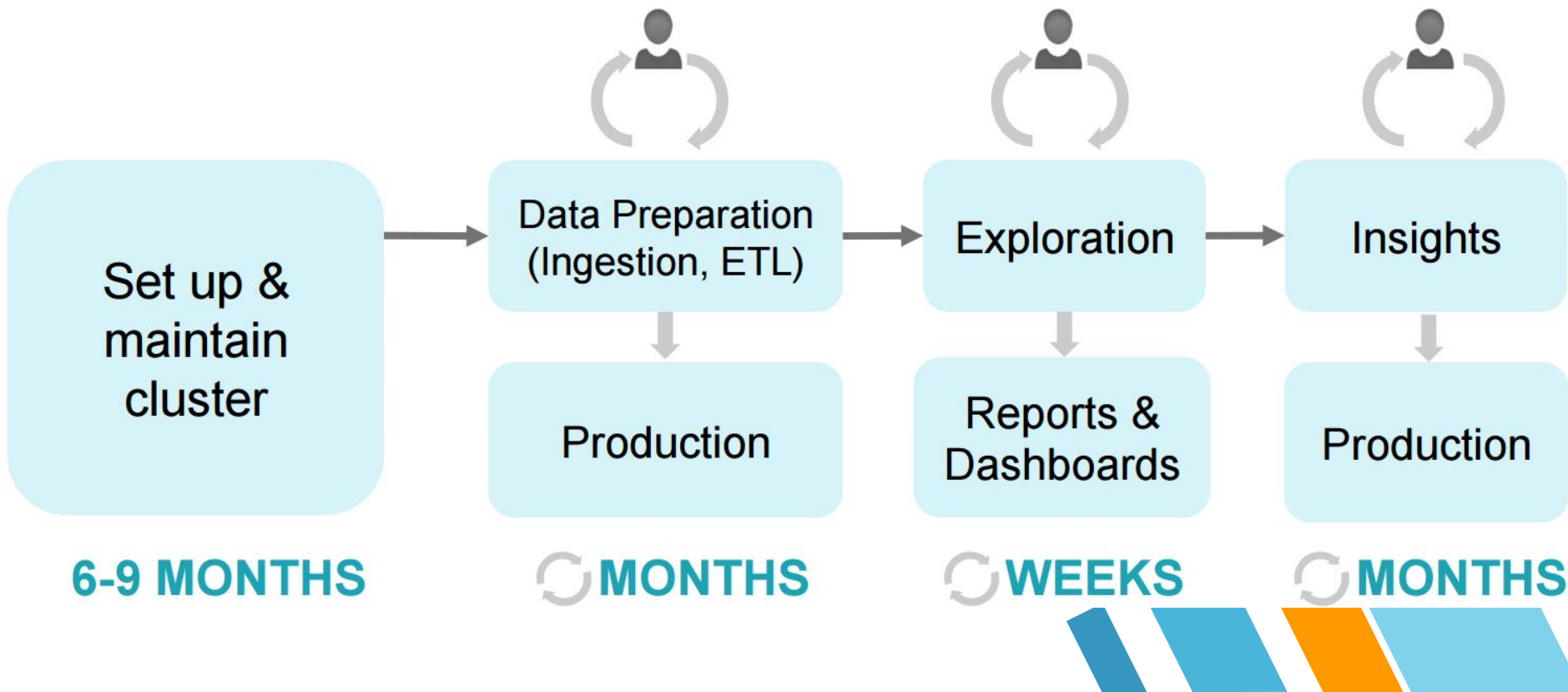
The screenshot shows the homepage of spark-packages.org. The header is dark blue with the 'Spark Packages' logo on the left and 'Feedback' and 'Register a package' links on the right. Below the header, a grey banner states 'A community index of packages for Apache Spark.' A horizontal navigation bar lists various categories with counts: All (63), Core (2), SQL (15), MLlib (13), Streaming (13), GraphX (0), Scala (5), Python (6), Java (0), Applications (2), and Deployment (0). The main content area displays three package cards. Each card includes the package name, a brief description, the source and owner, the latest release version and date, the Apache version, a star rating, and a user count. At the bottom of each card are tags for the package's components.

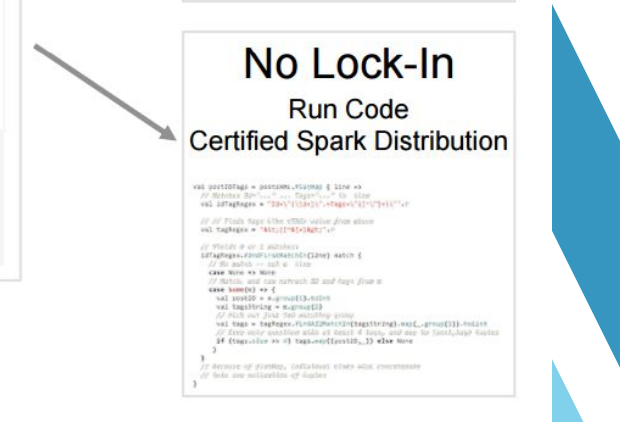
spark-avro
Integration utilities for using Spark with Apache Avro data
from: @databricks / owner: @pwendell / Latest release: 1.0.0 (04/10/15) / Apache-2.0 / ★★★★★ (15)
3 sql 3 input 2 library

spark-hbase-connector
Connect Spark to HBase for reading and writing data with ease
from: @nerdammer / owner: @nibbio84 / Latest release: 1.0.0 (05/11/15) / Apache-2.0 / ★★★★★ (1)
1 library 1 streaming 1 hbase

spark-csv
Spark SQL CSV data source
from: @databricks / owner: @falaki / Latest release: 1.0.3 (04/04/15) / Apache-2.0 / ★★★★★ (6)
1 sql 1 SparkSQL 1 DataSource

Steps of a Self-Service Data Project








References and additional resources






Online courses

- » <https://www.edx.org/course/introduction-big-data-apache-spark-uc-berkeleyx-cs100-1x>
 - » <https://www.coursera.org/learn/scala-spark-big-data/>
 - » <https://databricks.com/training>
- 



References: General

- » <https://spark.apache.org/docs/latest/quick-start.html>
 - » <https://spark.apache.org/docs/latest/programming-guide.html>
 - » <https://spark.apache.org/docs/latest/sql-programming-guide.html>
 - » <https://spark.apache.org/docs/latest/streaming-programming-guide.html>
 - » <https://sparkhub.databricks.com/>
 - » <https://databricks.com/blog/>
 - » <https://databricks.com/>
 - » <https://spark-summit.org/>
- 

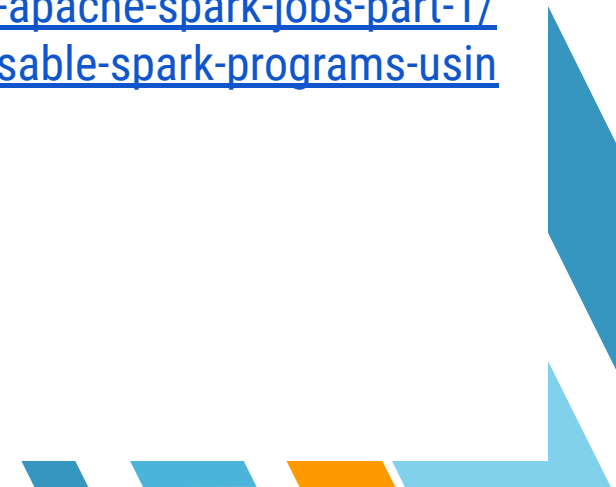


References: Databricks Blog

- » <https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html>
 - » <https://databricks.com/blog/2015/04/24/recent-performance-improvements-in-apache-spark-sql-python-dataframes-and-more.html>
 - » <https://databricks.com/blog/2015/01/09/spark-sql-data-sources-api-unified-data-access-for-the-spark-platform.html>
 - » <https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html>
 - » <https://databricks.com/blog/2015/07/08/new-visualizations-for-understanding-spark-streaming-applications.html>
 - » <https://databricks.com/blog/2015/07/30/diving-into-spark-streamings-execution-model.html>
- 



References: Various

- » <http://es.slideshare.net/frodriguezolivera/apache-spark-streaming>
 - » <http://arjon.es/2014/11/28/wisit2014-clasificando-tweets-en-realtime-con-apache-spark/>
 - » <http://blog.cloudera.com/blog/2015/05/working-with-apache-spark-or-how-i-learned-to-stop-worrying-and-love-the-shuffle/>
 - » <http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/>
 - » <http://blog.cloudera.com/blog/2015/03/how-to-build-re-usable-spark-programs-using-spark-shell-and-maven/>
- 

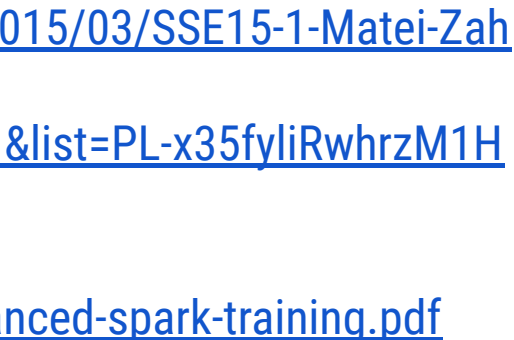


References: Spark Summit

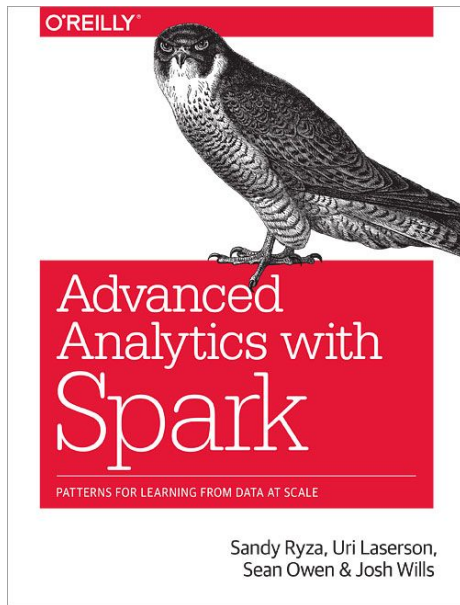
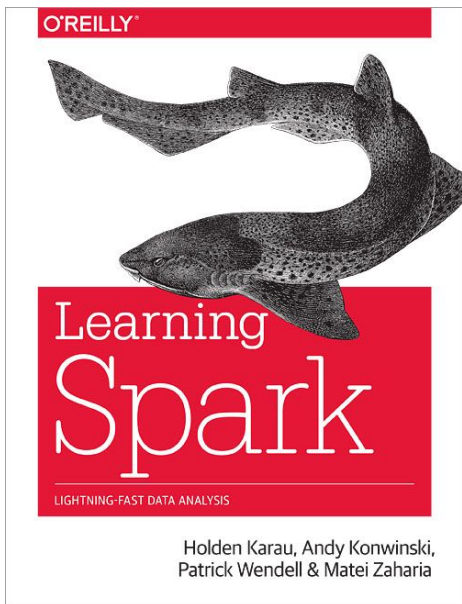
2014:

- » http://training.databricks.com/workshop/itas_workshop.pdf
- » <https://spark-summit.org/2014/wp-content/uploads/2014/07/A-Deeper-Understanding-of-Spark-Internals-Aaron-Davidson.pdf>

2015:

- » <https://spark-summit.org/2015-east/wp-content/uploads/2015/03/SSE15-1-Matei-Zaharia.pdf>
 - » <https://www.youtube.com/watch?v=EuWDz2Vb1Io&index=1&list=PL-x35fyliRwhrzM1Hq62WX4UellEqw3SU>
 - » <http://training.databricks.com/workshop/sparkcamp.pdf>
 - » <https://databricks-training.s3.amazonaws.com/slides/advanced-spark-training.pdf>
- 

References: Books





CREDITS

Content of the slides:

- » Big Data Tools - ITBA

Images:

- » Big Data Tools - ITBA
- » obtained from: commons.wikimedia.org

Special thanks to all the people who made and released these awesome resources for free:

- » Presentation template by [SlidesCarnival](https://slidescarnival.com)
 - » Photographs by [Unsplash](https://unsplash.com)
- 