https://cdn.eventfinda.co.nz/uploads/events/transformed/1330836-590621-34.png

Lecture 2: **Spark RDD, DataFrame, ML Pipelines, and Parallelization**

Haiping Lu - Scalable ML 2020

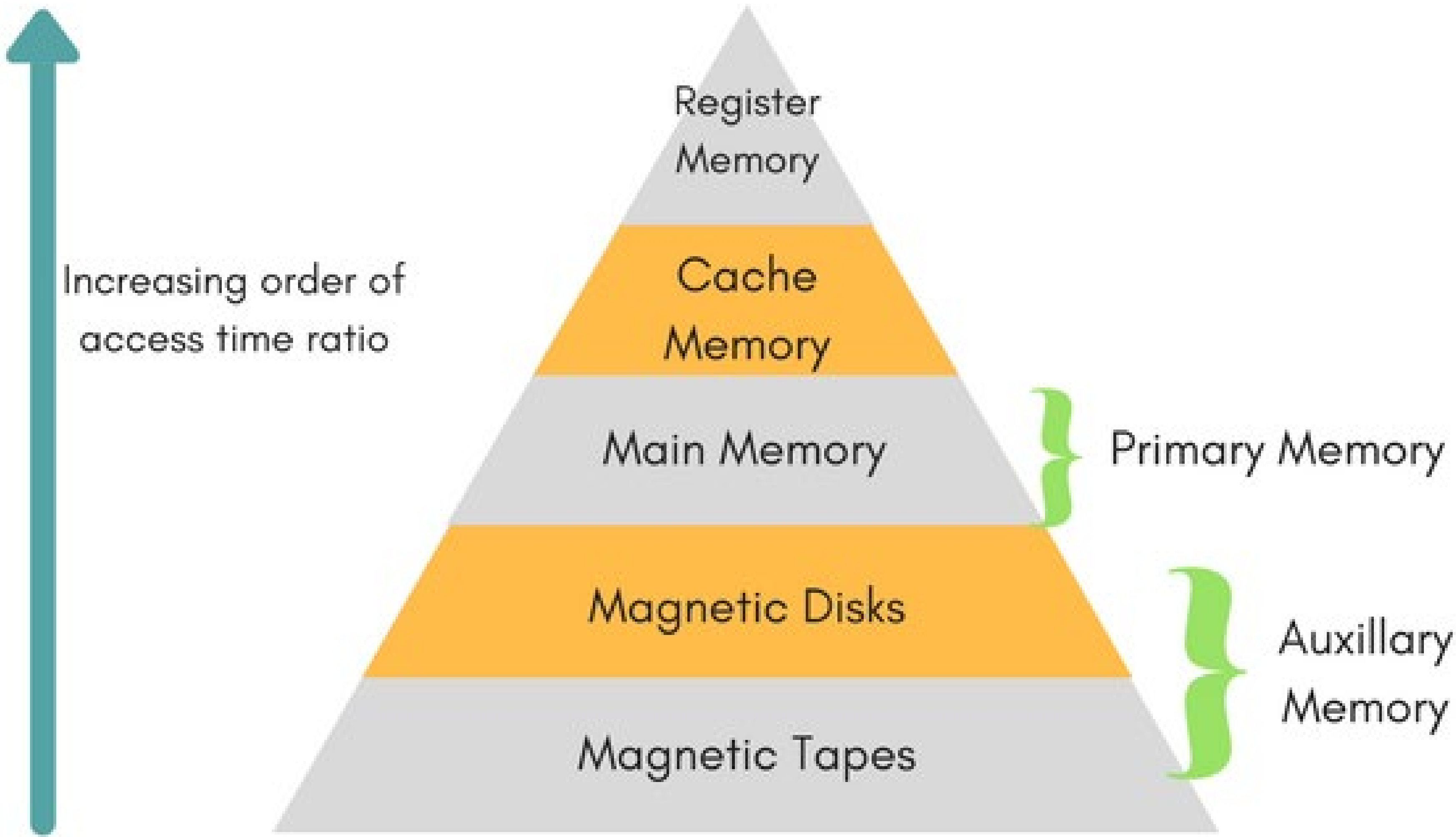
Week 2 Contents / Objectives

- Resilient Distributed Datasets: The Birth
- Spark DataFrame: Grow
- Spark ML Pipelines: Scalable ML
- Parallelization: How to Achieve Scalability?

Week 2 Contents / Objectives

- **Resilient Distributed Datasets: The Birth**
- Spark DataFrame: Grow
- Spark ML Pipelines: Scalable ML
- Parallelization: How to Achieve Scalability?

Computer Memory Organisation



https://www.studytonight.com/computer-architecture/memory-organization

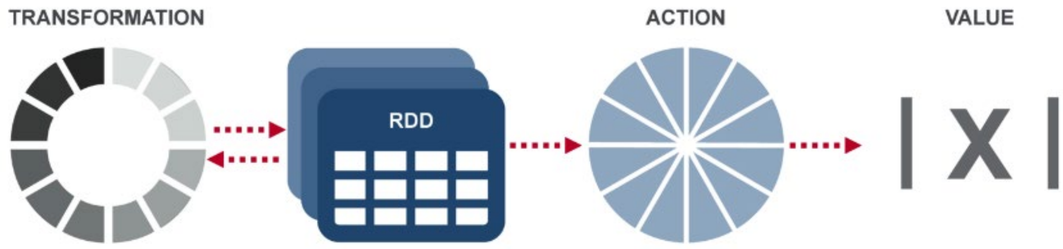
RDD

- Resilient Distributed Datasets:
  - A **distributed** memory abstraction that lets programmers perform **in-memory** computations on large clusters in a **fault-tolerant** manner.
- **Resilient**
  - Fault-tolerant, is able to recompute missing or damaged partitions due to node failures.
- **Distributed**
  - Data residing on multiple nodes in a cluster.
- **Dataset**
  - A collection of partitioned elements, e.g. tuples or other objects (that represent records of the data you work with).
- RDD is the primary data abstraction in Apache Spark and the core of Spark. It enables operations on collection of elements in parallel.

RDD Traits

- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
- **Immutable** or **Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
- **Parallel**, i.e. process data in parallel.
- **Typed**, i.e. values in a RDD have types, e.g. RDD[Long] or RDD[(Int, String)]. → **Dataset/DataFrame from 2.0**
- **Partitioned**, i.e. the data inside a RDD is partitioned (split into partitions) and then distributed across nodes in a cluster (one partition per JVM that may or may not correspond to a single node).

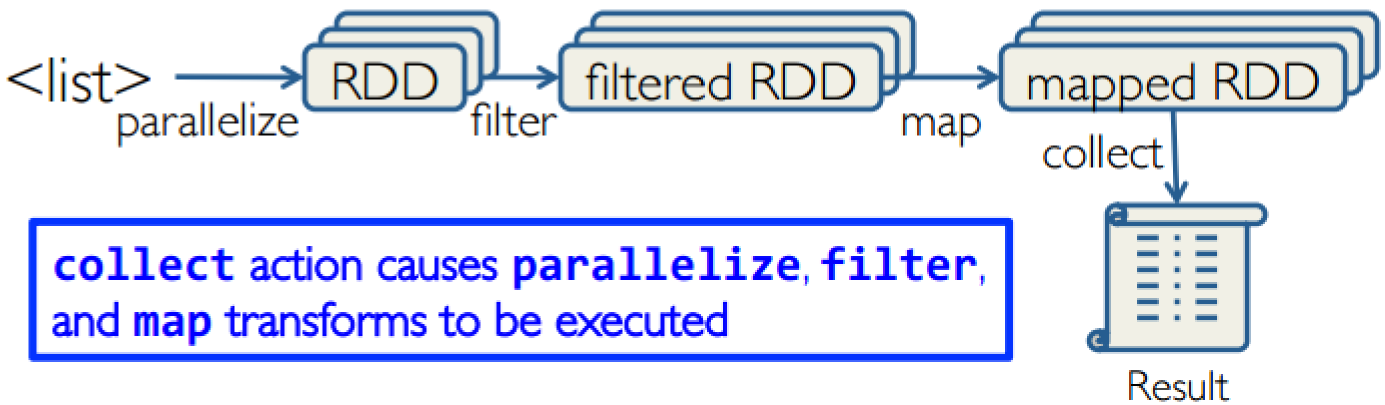
RDD Operations



- **Transformation:** returns a new RDD.
  - Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD.
  - Transformation functions include *map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, filter, join, etc.*
- **Action:** evaluates and returns a new value.
  - When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.
  - Action operations include *reduce, collect, count, first, take, countByKey, foreach, saveAsTextFile, etc.*

Working with RDDs

- Create an RDD from a data source
  - by parallelizing existing Python collections (lists)
  - by transforming an existing RDDs
  - from files in HDFS or any other storage system
- Apply transformations to an RDD: e.g., map, filter
- Apply actions to an RDD: e.g., collect, count



- Users can control two other aspects:
  - Persistence
  - Partitioning

Creating RDDs

- From HDFS, text files, Amazon S3, Apache HBase, SequenceFiles, any other Hadoop InputFormat
  - sc.parallelize()
  - sc.hadoopFile()

- Creating an RDD from a File
  - val inputfile = sc.textFile("...", 4) *
    - RDD distributed in 4 partitions
    - Elements are lines of input
    - Lazy evaluation means no execution happens now

***Note: Spark is written in Scala. We will use some Scala code in lecture though lab will be purely Python.**

Spark Transformations

- Create new datasets from an existing one
- Use lazy evaluation: results not computed right away – instead **remember** set of transformations applied to base dataset
  - Spark optimizes the required calculations
  - Spark recovers from failures
- Some transformation functions

Transformation	Description
`map(`*func*`)`	return a new distributed dataset formed by passing each element of the source through a function *func*
`filter(`*func*`)`	return a new dataset formed by selecting those elements of the source on which *func* returns true
`distinct([`*numTasks*`]))`	return a new dataset that contains the distinct elements of the source dataset
`flatMap(`*func*`)`	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)

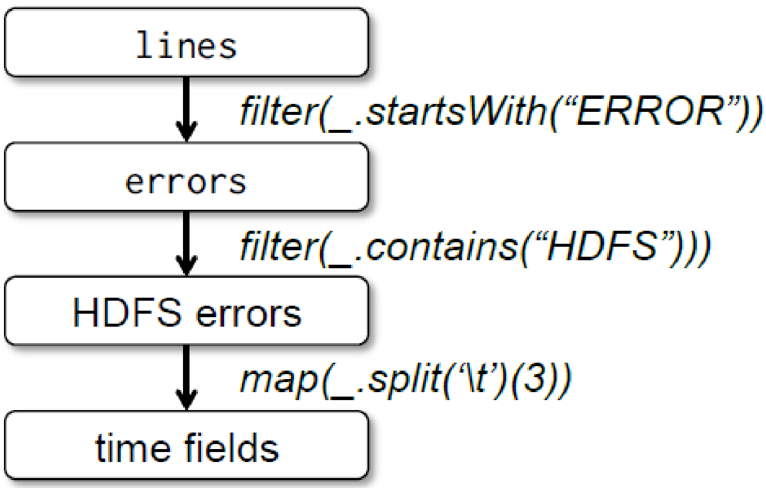
Spark Actions

- Cause Spark to execute recipe to transform source
- Mechanism for getting results out of Spark
- Some action functions

Action	Description
`reduce(func)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(n)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(n, key=func)`	return n elements ordered in ascending order or as specified by the optional key function

Example

- Web service is experiencing errors and an operators want to search terabytes of logs in the Hadoop file system to find the cause. https://amplab.cs.berkeley.edu/wp-content/uploads/2012/01/nsdi_spark.pdf

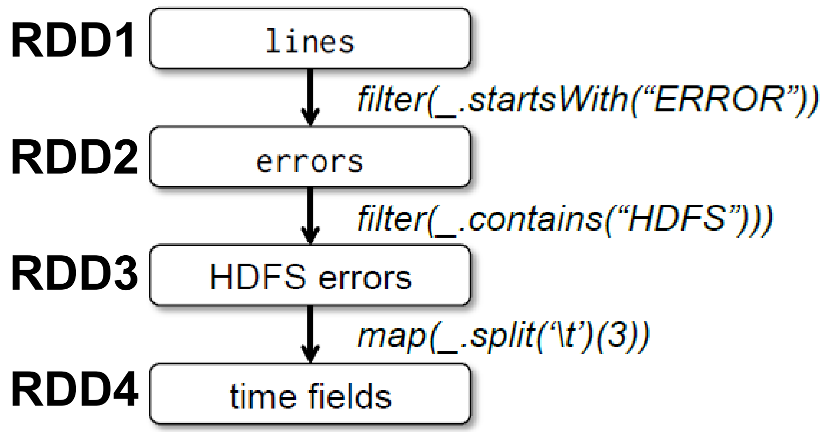



```
//base RDD
val lines = sc.textFile("hdfs://…")
//Transformed RDD
val errors = lines.filter(_.startsWith("Error"))
errors.persist()     //or .cache()
errors.count()
errors.filter(_.contains("HDFS"))
          .map(_split('\t')(3))
          .collect()
```

- Line1: RDD backed by an HDFS file (base RDD lines not loaded in memory)
- Line3: Asks for errors to persist in memory (errors are in RAM)

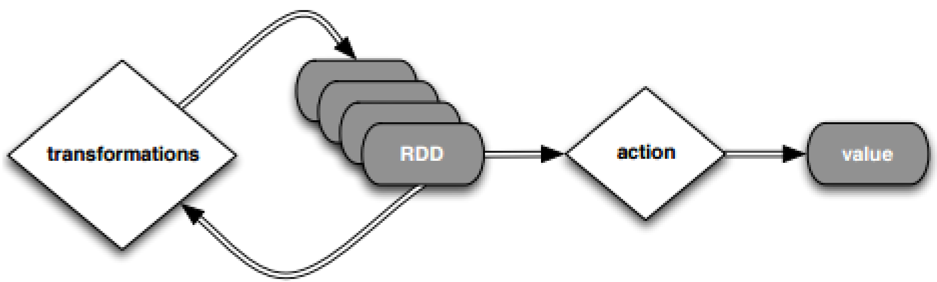
Lineage Graph

- RDDs keep track of *lineage*
- RDD has enough information about how it was derived from to compute its partitions from data in stable storage.



- Example:
- If a partition of errors is lost, Spark rebuilds it by applying a filter on only the corresponding partition of lines.
- Partitions can be recomputed in parallel on different nodes, without having to roll back the whole program.

Operations

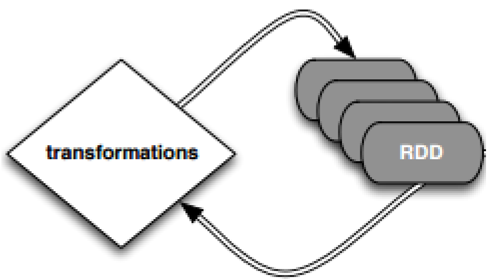



```
//base RDD
val lines = sc.textFile("hdfs://…")
//Transformed RDD
val errors = lines.filter(_.startsWith("Error"))
errors.persist()
errors.count()
errors.filter(_.contains("HDFS"))
        .map(_split('\t')(3))
        .collect()
```

Operations – Step by Step

//base RDD

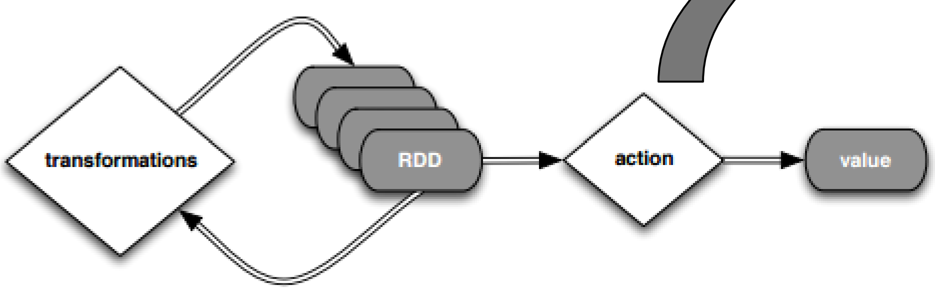
val lines = sc.textFile("hdfs://…")



//Transformed RDD

val errors = lines.filter(_.startsWith("Error"))

errors.persist()



errors.count()

count() causes Spark to: 1) read data; 2) sum within partitions; 3) combine sums in driver

Put transform and action together:

errors.filter(_.contains("HDFS")).map(_split('\t')(3)).collect()

SparkContext

- SparkContext is the entry point to Spark for a Spark application
  - The entry point becomes **SparkSession** since 2.0
- Once a SparkContext instance is created you can use it to
  - Create RDDs
  - Create accumulators
  - Create broadcast variables
  - access Spark services and run jobs
- A Spark context is essentially a client of Spark's execution environment and acts as the *master of your Spark application*
- The first thing a Spark program must do is to create a SparkContext object, which tells Spark how to access a cluster
- In the Spark shell, a special interpreter-aware SparkContext is already created for you, in the variable called *sc*

RDD Persistence: Cache/Persist

- One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations.
- When you persist an RDD, each node stores any partitions of it. You can reuse it in other actions on that dataset
- Each persisted RDD can be stored using a different *storage level,* e.g.
  - MEMORY_ONLY:
    - Store RDD as deserialized Java objects in the JVM.
    - If the RDD does not fit in memory, some partitions will not be cached and will be recomputed when they're needed.
    - This is the default level.
  - MEMORY_AND_DISK:
    - If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
- cache() = persist(StorageLevel.MEMORY_ONLY)

Why Persisting RDD?

val lines = sc.textFile(“hdfs://...”)

val errors = lines.filter(_.startsWith(“Error”))

errors.persist()

errors.count()

•If you do errors.count() again, the file will be loaded again and computed again.

•Persist will tell Spark to cache the data in memory, to reduce the data loading cost for further actions on the same data

•erros.persist() will do nothing. It is a lazy operation. But now the RDD says "read this file and then cache the contents". The action will trigger computation and data caching.

Spark Key-Value RDDs

- Similar to Map Reduce, Spark supports Key-Value pairs
- Each element of a *Pair RDD* is a pair tuple
- Some Key-Value transformation functions:

Key-Value Transformation	Description
`reduceByKey(`*func*`)`	return a new distributed dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) ➔ V
`sortByKey()`	return a new dataset (K,V) pairs sorted by keys in ascending order
`groupByKey()`	return a new dataset of (K, Iterable<V>) pairs

Week 2 Contents / Objectives

- Resilient Distributed Datasets: The Birth
- **Spark DataFrame: Grow**
- Spark ML Pipelines: Scalable ML
- Parallelization: How to Achieve Scalability?

Challenges & Solutions

- Perform ETL to and from various (semi- or unstructured) data sources
- Perform advanced analytics (e.g. machine learning, graph processing) that are hard to express in relational systems

- A *DataFrame* API that can perform relational operations on both external data sources and Spark's built-in RDDs.
- A highly extensible optimizer, *Catalyst*, that uses features of Scala to add composable rule, control code gen., and define extensions.

DataFrame-based API for MLlib

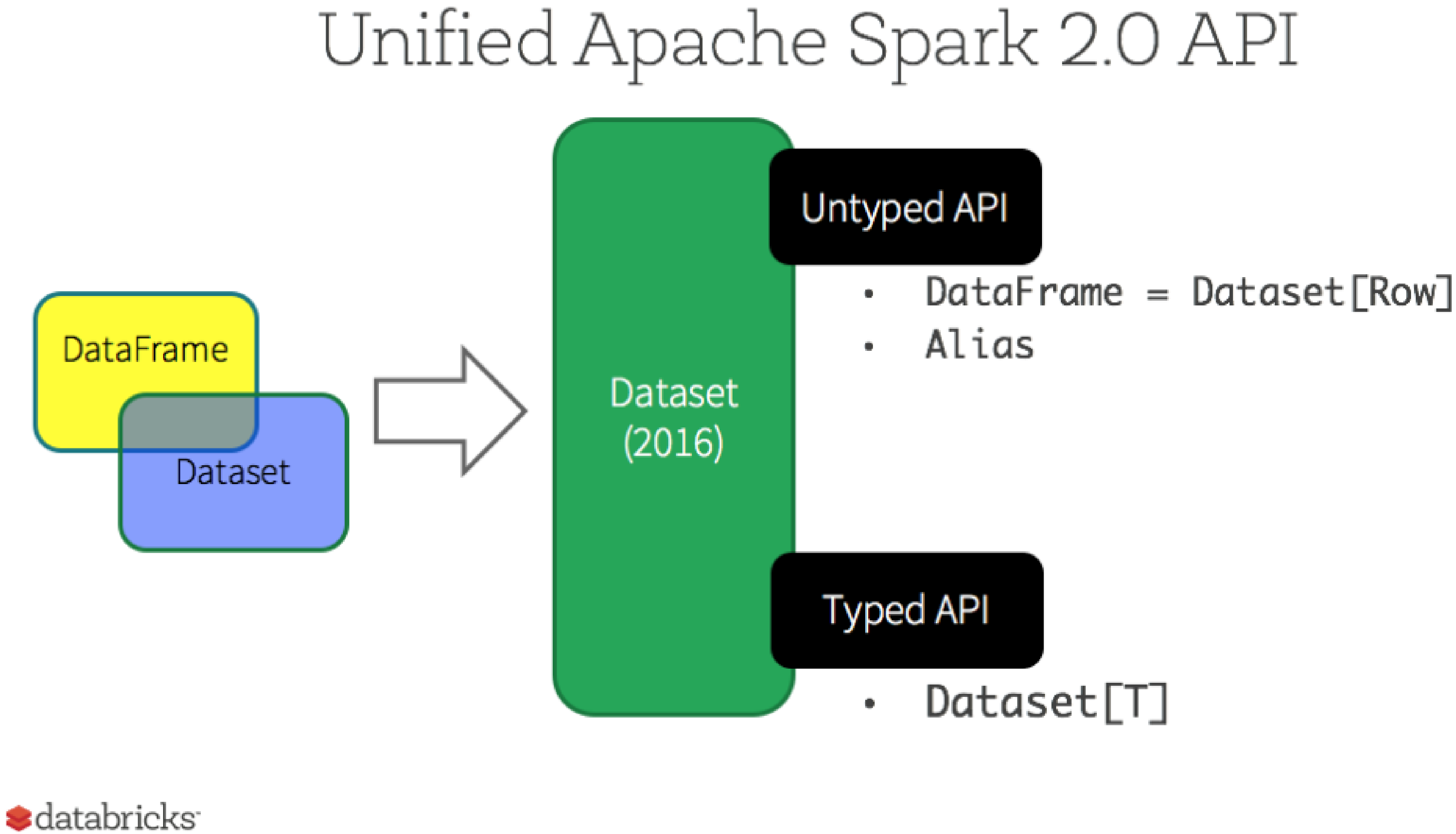
- a.k.a. “Pipelines” API, with utilities for constructing ML Pipelines
- In 2.0, the DataFrame-based API became the primary API for MLlib
  - Voted by community
  - org.apache.spark.**ml**, pyspark.**ml**
- The RDD-based API will entermaintenance mode
  - Still maintained with bug fixes, but no new features
  - org.apache.spark.**mllib**, pyspark.**mllib**

Structuring Spark: DataFrames and Datasets

- DataFrames and Datasets
  - DataFrame (schema, generic untyped)
    - Index access, named columns (like a table)
  - Dataset (static typing, strongly-typed)
    - Object access
  - DataFrame = Dataset[Row] (row: generic untyped)
  - Unified in Apache Spark 2.0
- RDDs are low-level (like assembler), DataFrames & Datasets are built on top of RDDs
- New libraries: built on Datasets and DataFrames

A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets

Unified API


Unified Apache Spark 2.0 API
DataFrame
Dataset
Dataset
(2016)
Untyped API
• DataFrame = Dataset[Row]
• Alias
Typed API
• Dataset[T]
databricks

Typed and Un-typed APIs

Language	**Main Abstraction**
Scala	Dataset[T] & DataFrame (alias for Dataset[Row])
Java	Dataset[T]
Python*	DataFrame
R*	DataFrame

Note: *Since Python and R have no compile-time type-safety, we only have untyped APIs, namely DataFrames.*

Benefits of Dataset APIs

- Static-typing and runtime type-safety
  - SQL least restrictive, no syntax error until runtime
  - DF/DS: syntax error detected at compile time
- High-level abstraction and custom view into structured and semi-structured data, e.g. JSON
- Ease-of-use of APIs with structure
  - Rich semantics and domain specific operations
- Performance and Optimization
  - SQL Catalyst

DataFrame

```
ctx = new HiveContext()
users = ctx.table("users")
young = users.where(users("age") < 21)
println(young.count())
```

- A distributed collection of rows with the same schema
- Can be constructed from external data sources or RDDs into essentially an RDD of Row objects
- Supports relational operators (e.g. *where*, *groupby*) as well as Spark operations.
- Evaluated lazily → unmaterialized *logical* plan

DataFrames

dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	43	B Jones
Chem	61	M Kennedy

Data grouped into named columns

RDD API

```
pdata.map(lambda x: (x.dept, [x.age, 1])) \
  .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
  .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
  .collect()
```

DataFrame API

```
data.groupBy("dept").avg("age")
```

DataFrames

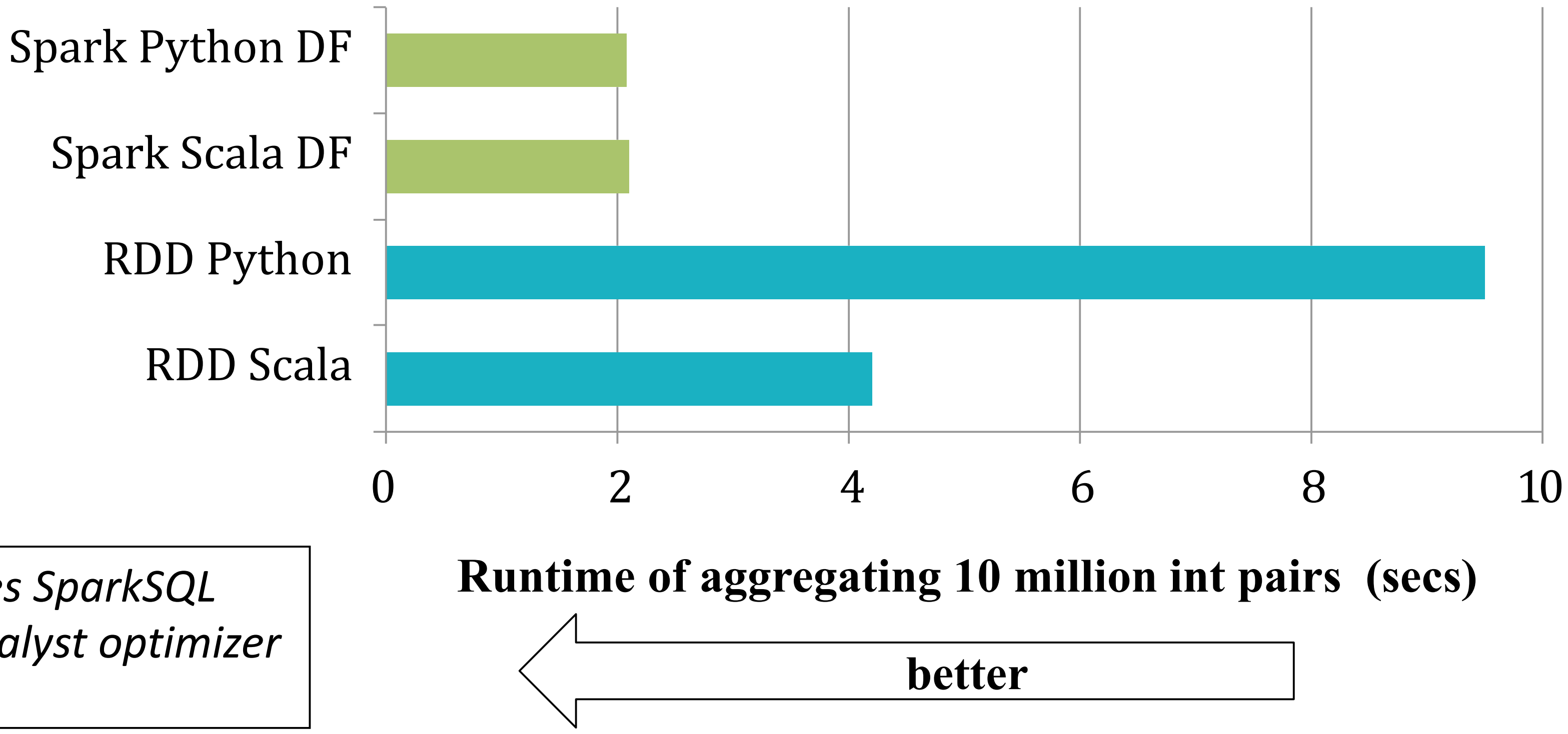
dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	43	B Jones
Chem	61	M Kennedy

Data grouped into named columns

DSL for common tasks
- Project, filter, aggregate, join, …
- Metadata
- UDFs

```
data.groupBy("dept").avg("age")
```

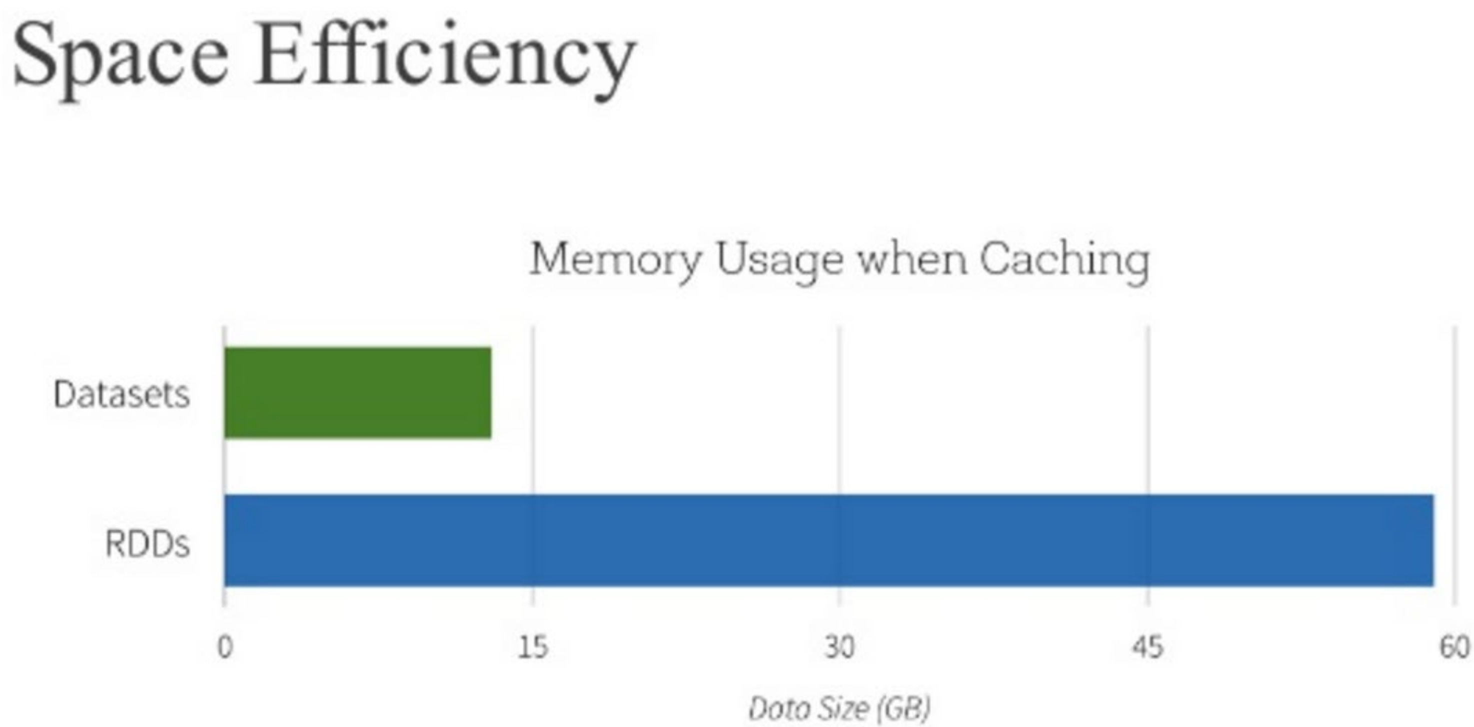
Spark DataFrames are *fast*



Uses SparkSQL Catalyst optimizer

30

Space Efficiency



Week 2 Contents / Objectives

- Resilient Distributed Datasets: The Birth

- Spark DataFrame: Grow

- **Spark ML Pipelines: Scalable ML**

- Parallelization: How to Achieve Scalability?

Machine Learning Library (MLlib)

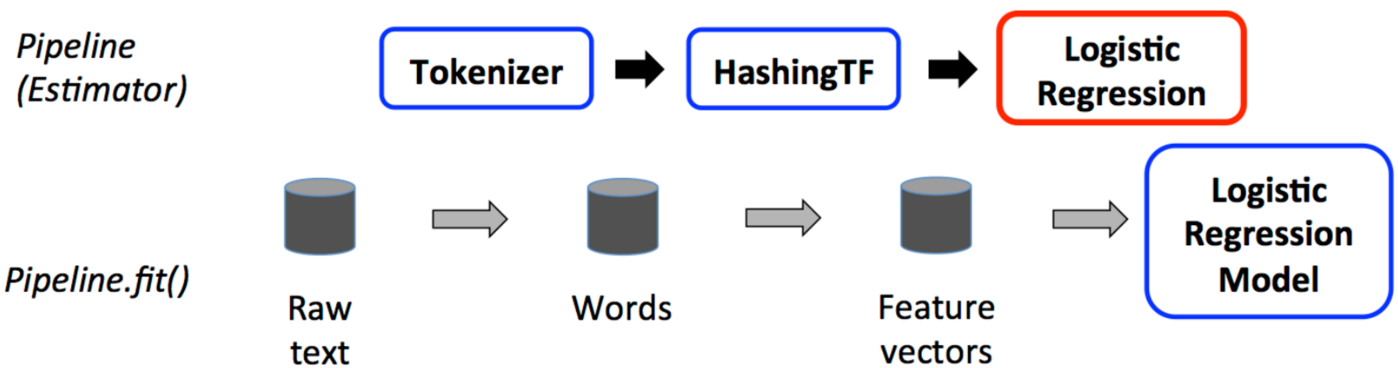
- ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
- Featurization: feature extraction, transformation, dimensionality reduction, and selection
- Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
- Persistence: saving and load algorithms, models, and Pipelines
- Utilities: linear algebra, statistics, data handling, etc

Main Concepts in Pipelines

- DataFrame: an ML dataset, which can hold a variety of data types. E.g., different columns storing text, feature vectors, true labels, and predictions.
- Transformer: algorithm transforming one DataFrame into another DataFrame. E.g., ML model→ features into predictions.
- Estimator: algorithm fit on a DataFrame to produce a Transformer. E.g., ML algorithm DataFrame →model
- Pipeline: chains multiple Transformers and Estimators together to specify an ML workflow.
- Parameter: All Transformers and Estimators now share a common API for specifying parameters

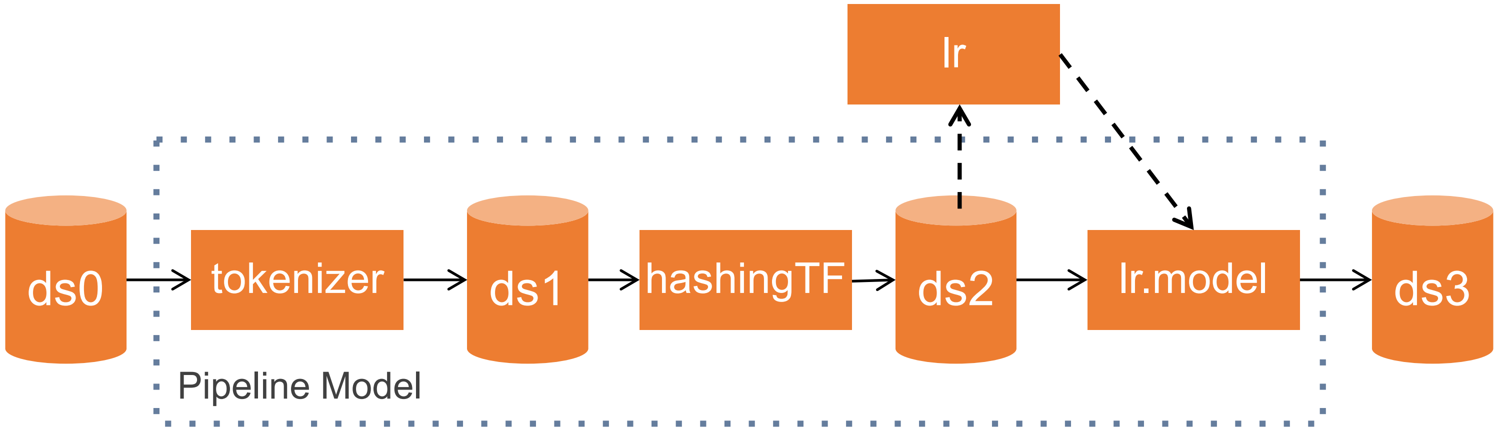
ML Pipelines

- ML Pipelines: high-level APIs to create and tune machine learning pipelines.
- Spark DataFrame: distributed collection of data organized into named columns.
  - Table in a relational database
  - Data frame in R or Python



Spark MLlib Pipelines

```
tokenizer = Tokenizer(inputCol="text", outputCol="words”)
hashingTF = HashingTF(inputCol="words", outputCol="features”)
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

df = sqlCtx.load("/path/to/data")
model = pipeline.fit(df)
```



Example: Text Classification

<u>Goal</u>: Given a text document, predict its topic.

Features

```
Subject: Re: Lexan Polish?
Suggest McQuires #1 plastic
polish.  It will help somewhat
but nothing will remove deep
scratches without making it
worse than it already is.
McQuires will do
something...
```

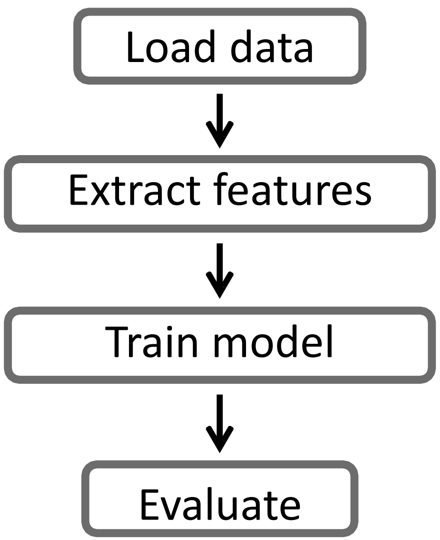
⟹

Label

1: about science
0: not about science

Dataset: "20 Newsgroups"
From UCI KDD Archive

ML Workflow



Load Data

Load data

↓

Extract features

↓

Train model

↓

Evaluate

Data sources for DataFrames

built-in

Parquet

JDBC

{ JSON }

PostgreSQL

HIVE

MySQL

HDFS

amazon web services S3

H2

external

AVRO

CSV

dBase

APACHE HBASE

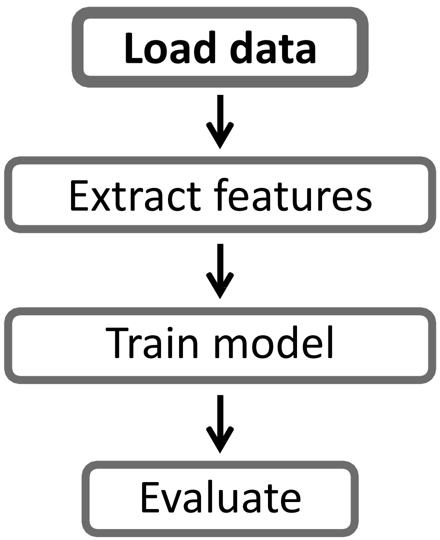
elasticsearch.

cassandra

amazon web services Amazon Redshift

and more …

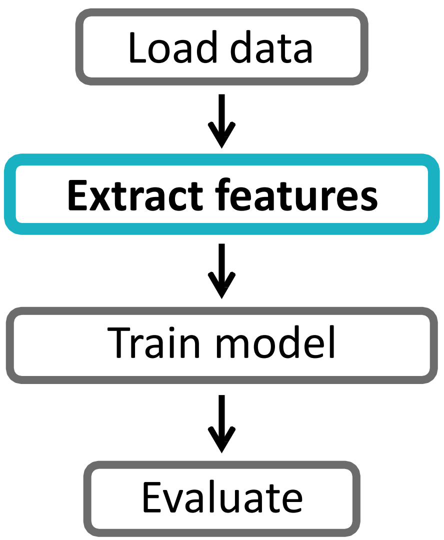
Load Data



<u>Current data schema</u>

```
label: Int
text: String
```

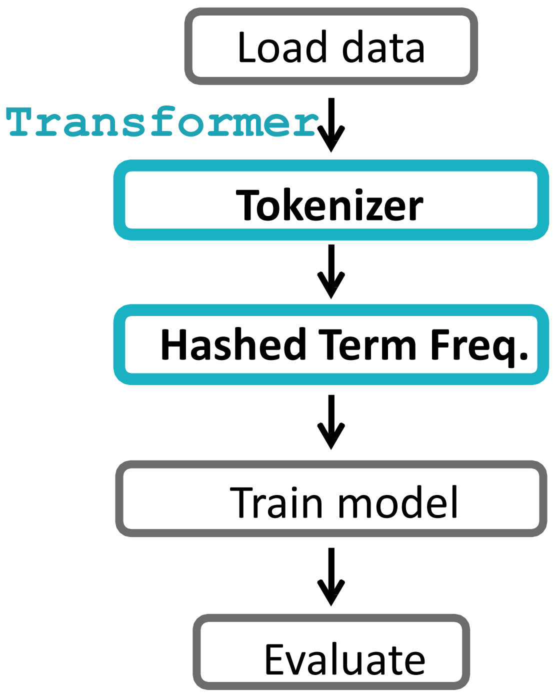
Extract Features



<u>Current data schema</u>

```
label: Int
text: String
```

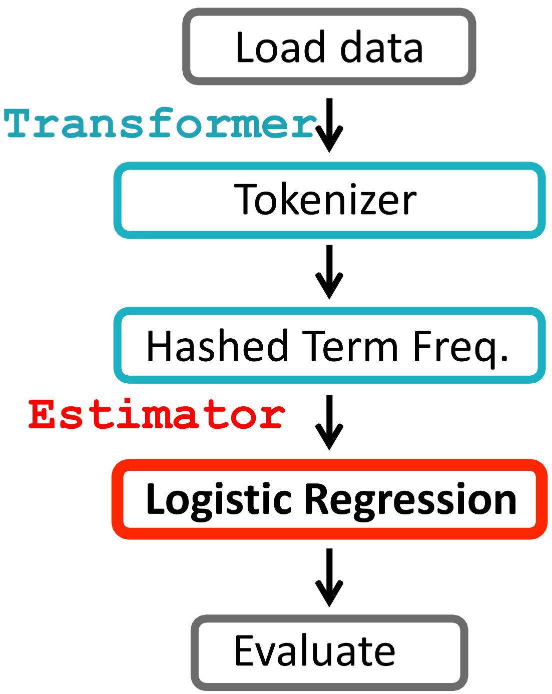
Extract Features



<u>Current data schema</u>

```
label: Int
text: String

words: Seq[String]

features: Vector
```

Train a Model

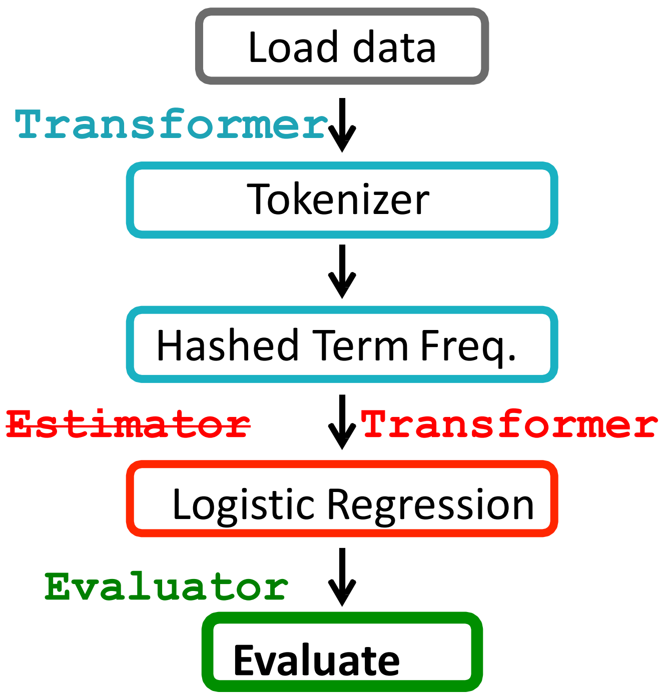


Current data schema

```
label: Int
text: String

words: Seq[String]

features: Vector
```

`model parameters`

Evaluate the Model



Current data schema

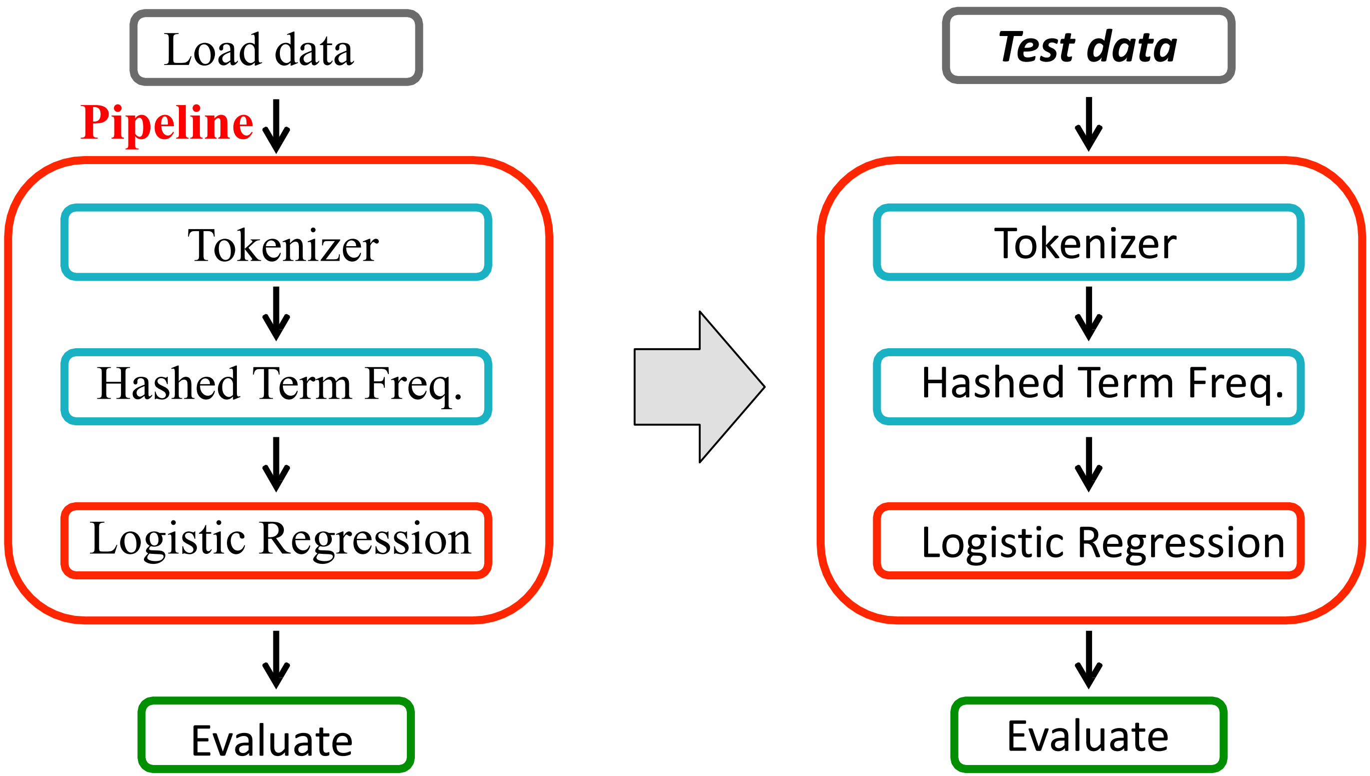
```
label: Int
text: String

words: Seq[String]

features: Vector

prediction: Int
```

By default, always append new columns

→ Can go back & inspect intermediate results

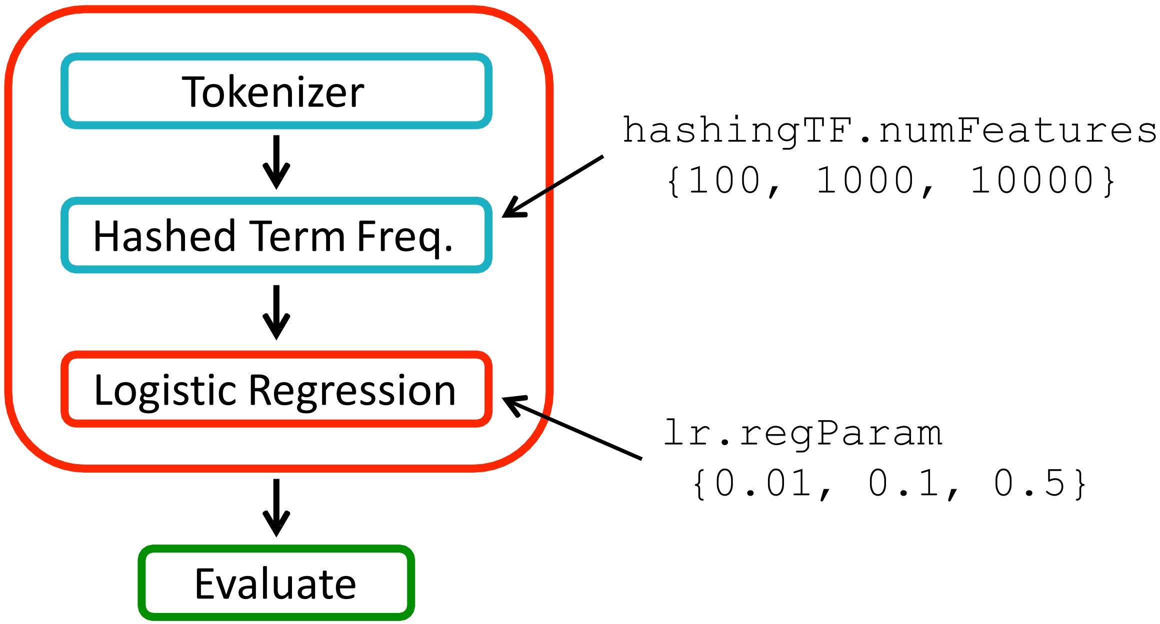
→ Made efficient by DataFrame optimizations

ML Pipelines



Re--run exactly the same way

Parameter Tuning



CrossValidator

Given:

- Estimator
- Parameter grid
- Evaluator

Find best parameters

Week 2 Contents / Objectives

- Resilient Distributed Datasets: The Birth
- Spark DataFrame: Grow
- Spark ML Pipelines: Scalable ML
- **Parallelization: How to Achieve Scalability?**

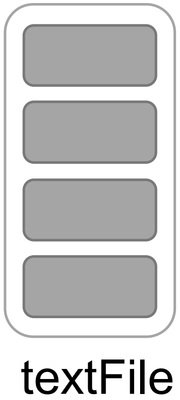
How Spark Works

- User application create RDDs, transform them, and run actions.
- This results in a DAG (Directed Acyclic Graph) of operators.
- DAG is compiled into stages
- Each stage is executed as a series of Task (one Task for each Partition).

Word Count in Spark

val file = sc.textFile("hdfs://…", 4)

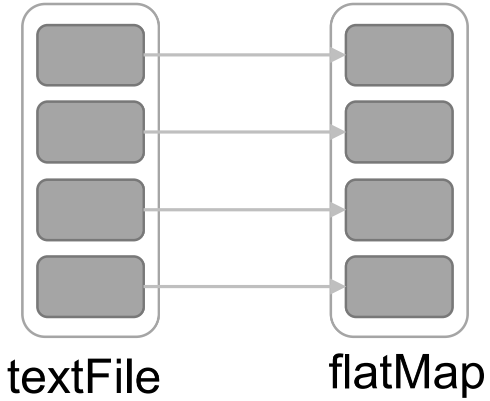
RDD[String]



Word Count in Spark

```
val file = sc.textFile("hdfs://…", 4)
val words = file.flatMap(line =>
        line.split("\t"))
```

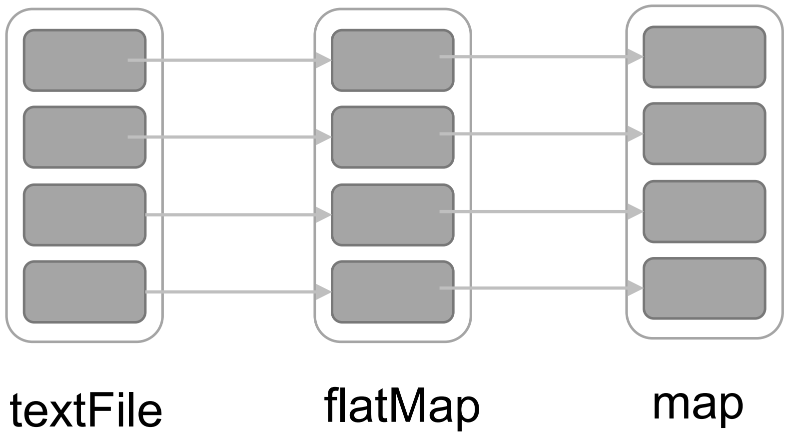
RDD[String]
RDD[List[String]]



Word Count in Spark

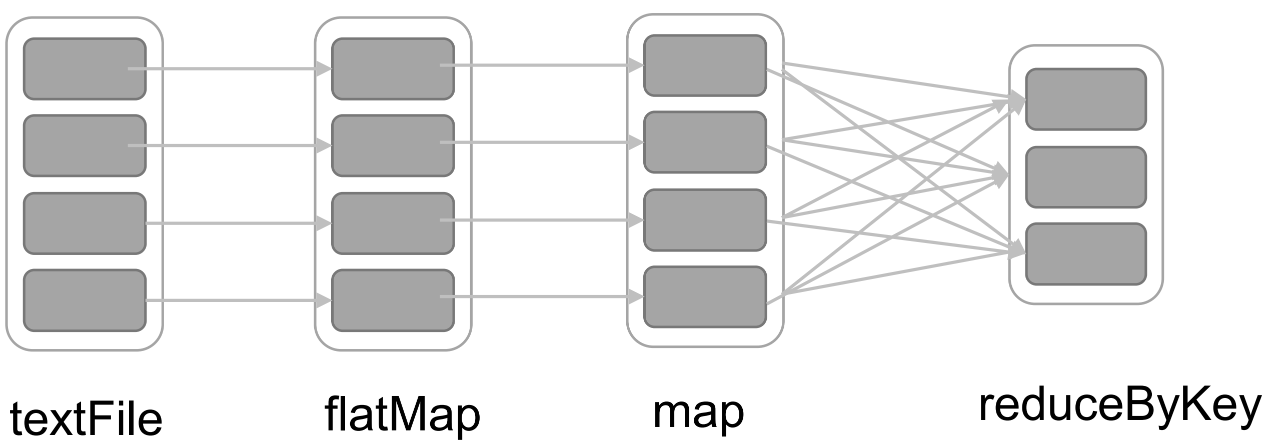
```
val file = sc.textFile("hdfs://...", 4)
val words = file.flatMap(line =>
        line.split("\t"))
val pairs = words.map(t => (t, 1))
```

RDD[String]
RDD[List[String]]
RDD[(String, Int)]



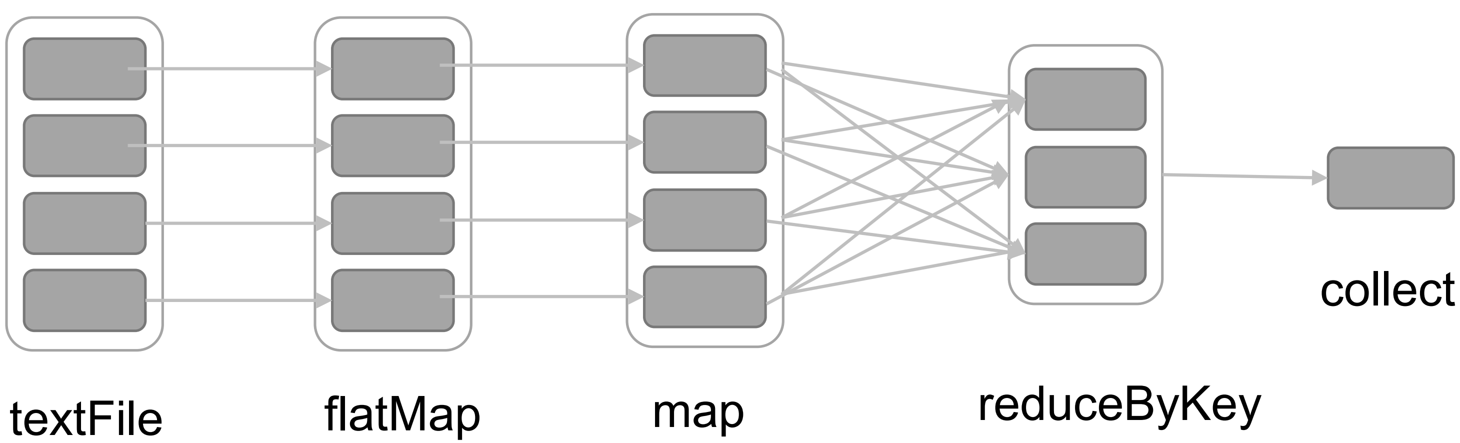
Word Count in Spark

```
val file = sc.textFile("hdfs://...", 4)          RDD[String]
val words = file.flatMap(line =>                 RDD[List[String]]
        line.split("\t"))
val pairs = words.map(t => (t, 1))               RDD[(String, Int)]
val count = pairs. reduceByKey(_+_)              RDD[(String, Int)]
```

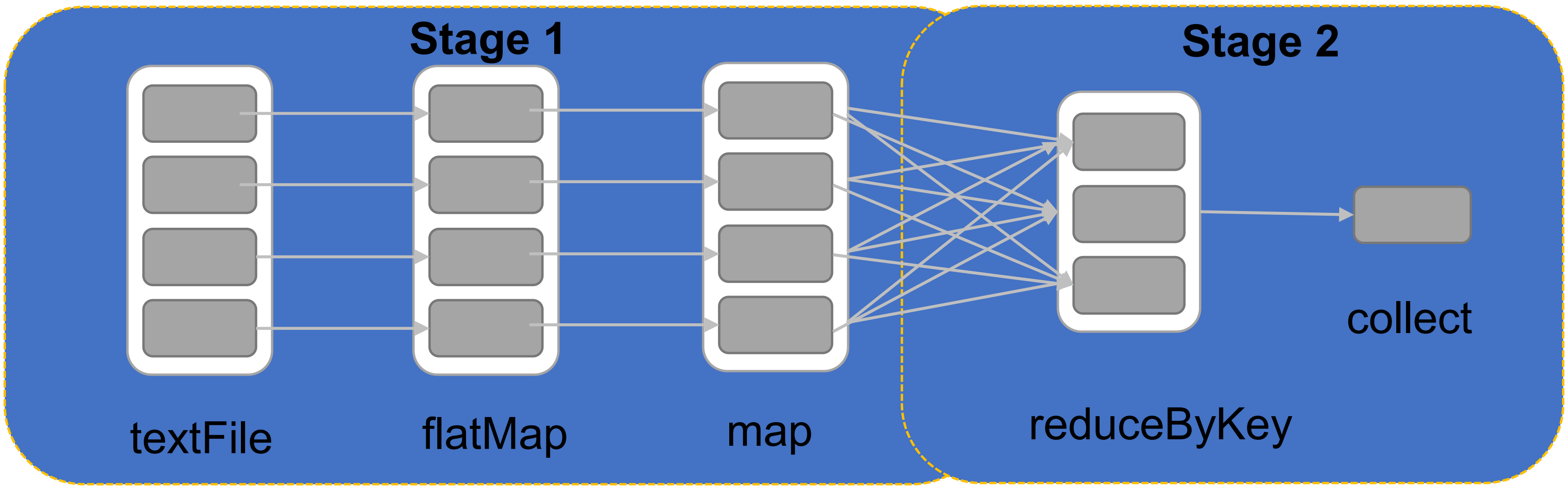


Word Count in Spark

```
val file = sc.textFile("hdfs://...", 4)          RDD[String]
val words = file.flatMap(line =>                 RDD[List[String]]
        line.split("\t"))
val pairs = words.map(t => (t, 1))               RDD[(String, Int)]
val count = pairs. reduceByKey(_+_)              RDD[(String, Int)]
count.collect()                                  Array[(String, Int)]
```

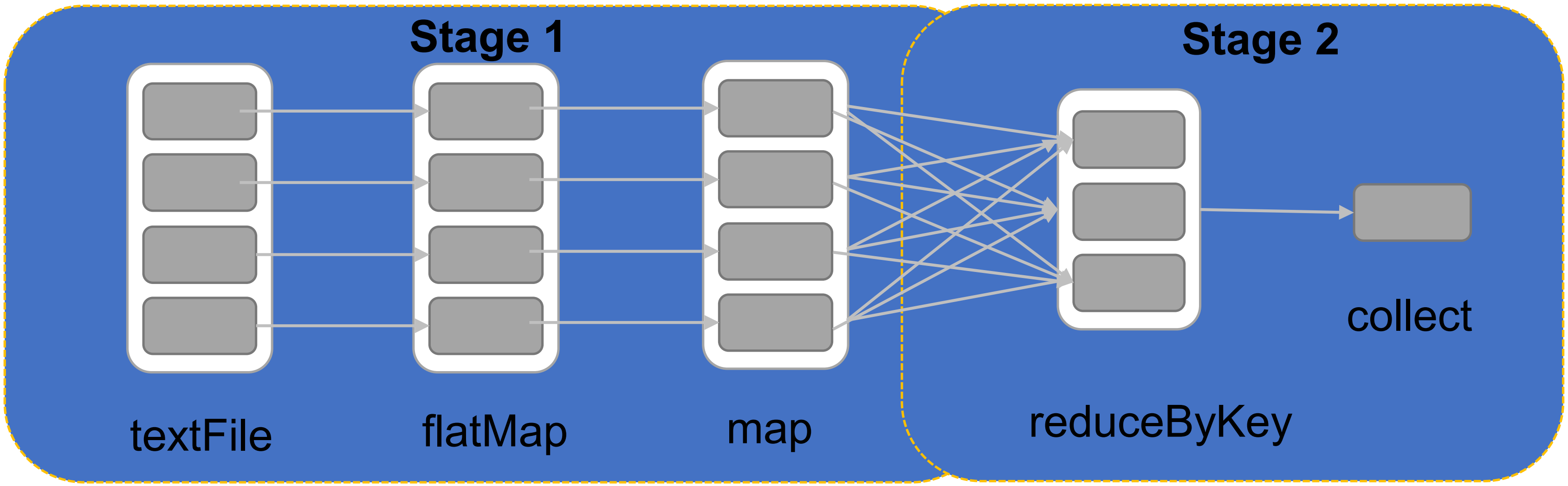


Execution **Plan**

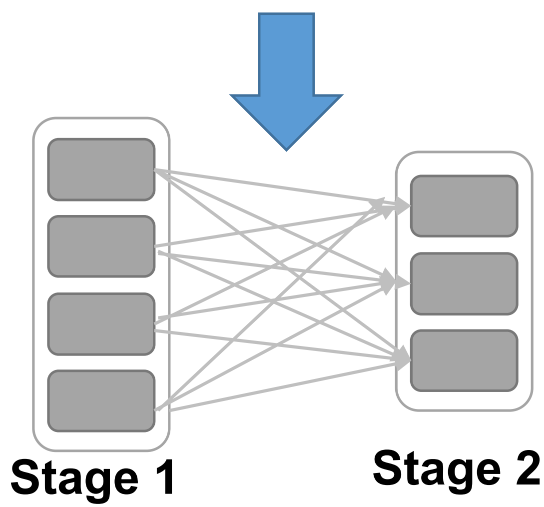


- The scheduler examines the RDD's lineage graph to build a DAG of stages.
- Stages are sequences of RDDs, that don't have a Shuffle in between
- The boundaries are the shuffle stages.

Execution **Plan**



1. Read HDFS split
2. Apply both the maps
3. Start Partial reduce
4. Write shuffle data



1. Read shuffle data
2. Final reduce
3. Send result to driver program

Stage Execution

- Create a task for each Partition in the new RDD
- Serialize the Task
- Schedule and ship Tasks to Slaves

- All this happens internally

Word Count in Spark (As a Whole View)

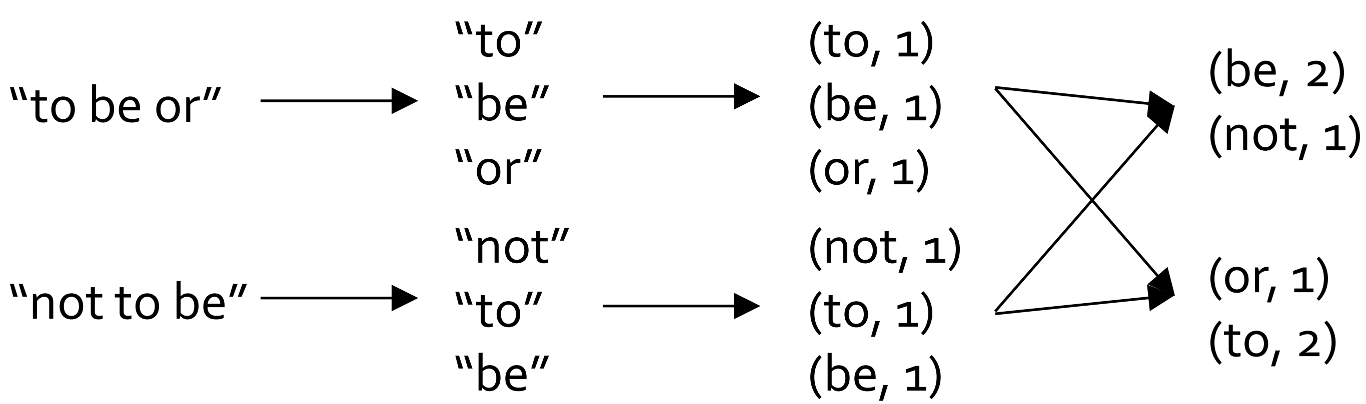
- Word Count using Scala in Spark

Transformation

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

val counts = file.flatMap(line => line.split(" "))
    .map(word => (word,1))
    .reduceByKey(_ + _)
```

Action

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



RDD Operations

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$join()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
Actions	$count()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Long}$
	$collect()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]}$
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{T}$
	$lookup(k : \mathrm{K})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]}$ (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.,* HDFS

More Examples on Pair RDD

- Create a pair RDD from existing RDDs

```
val pairs = sc.parallelize( List( (“This”, 2), (“is”, 3), (“Spark”, 5), (“is”, 3) ) )
pairs.collect().foreach(println)
```

- reduceByKey() function: reduce key-value pairs by key using give *func*

```
val pair1 = pairs.reduceByKey((x,y) => x + y)
pairs1.collect().foreach(println)
```

- mapValues() function: work on values only

```
val pair2 = pairs.mapValues( x => x -1 )
pairs2.collect().foreach(println)
```

- groupByKey() function: When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs

```
pairs.groupByKey().collect().foreach(println)
```

Setting the Level of Parallelism

- All the pair RDD operations take an optional second parameter for number of tasks

```
> words.reduceByKey((x,y) => x + y, 5)
> words.groupByKey(5)
```

Using Local Variables

- Any external variables you use in a closure will automatically be shipped to the cluster:

```
> query = sys.stdin.readline()
> pages.filter(lambda x: query in x).count()
```

- Some caveats:
  - Each task gets a new copy (updates aren't sent back)
  - Variable must be Serializable

Shared Variables

- When you perform transformations and actions that use functions (e.g., map(f: T=>U)), Spark will automatically push a closure containing that function to the workers so that it can run at the workers.
- Any variable or data within a closure or data structure will be distributed to the worker nodes along with the closure
- When a function (such as map or reduce) is executed on a cluster node, it works on **separate** copies of all the variables used in it.
- Usually these variables are just constants but they cannot be shared across workers efficiently.

Shared Variables

- Consider These Use Cases
  - Iterative or single jobs with large global variables
    - Sending large read-only lookup table to workers
    - Sending large feature vector in a ML algorithm to workers
    - Problems? Inefficient to send large data to each worker with each iteration
    - Solution: Broadcast variables
  - Counting events that occur during job execution
    - How many input lines were blank?
    - How many input records were corrupt?
    - Problems? Closures are one way: driver -> worker
    - Solution: Accumulators

Broadcast Variables

- Allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.
  - For example, to give every node a copy of a large input dataset efficiently
- Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost
- Broadcast variables are created from a variable **v** by calling **SparkContext.broadcast(v)**. Its value can be accessed by calling the **value** method.

```
scala > val broadcastVar =sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
scala > broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

- The broadcast variable should be used instead of the value **v** in any functions run on the cluster, so that **v** is not shipped to the nodes more than once.

Accumulators

- Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel.
- Be used to implement counters (as in MapReduce) or sums.
- Spark natively supports accumulators of numeric types, and programmers can add support for new types.
- Only driver can read an accumulator’s value, not tasks
- An accumulator is created from an initial value **v** by calling **SparkContext.accumulator(v)**.

```
scala> val accum = sc.longAccumulator("My Accumulator") accum:
org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name:
Some(My Accumulator), value: 0)
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))
... 10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
scala> accum.value
res2: Long = 10
```

Recommended Reading

- Sections 2.4.2 and 2.4.3 of the MMDS book (3rd edition)

  http://i.stanford.edu/~ullman/mmds/ch2n.pdf

- The full book
  http://i.stanford.edu/~ullman/mmds/book0n.pdf