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

COM6012: Scalable ML by Haiping Lu

YouTube Playlist: https://www.youtube.com/c/HaipingLu/

Week 2 Contents / Objectives

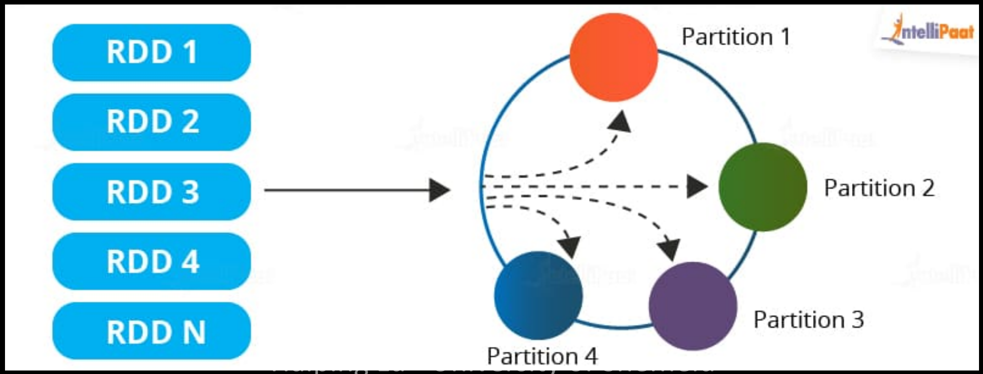
- Resilient Distributed Datasets
- DataFrames and Datasets
- Machine Learning Pipelines
- Execution Parallelization

Week 2 Contents / Objectives

- Resilient Distributed Datasets
- DataFrames and Datasets
- Machine Learning Pipelines
- Execution Parallelization

RDD

- Resilient Distributed Datasets
  - A distributed memory abstraction enabling in-memory computations on large clusters in a fault-tolerant manner
  - The primary data abstraction in Spark enabling operations on collection of elements in parallel
- **R**: recompute missing partitions due to node failures
- **D**: data distributed on multiple nodes in a cluster
- **D**: a collection of partitioned elements (datasets)

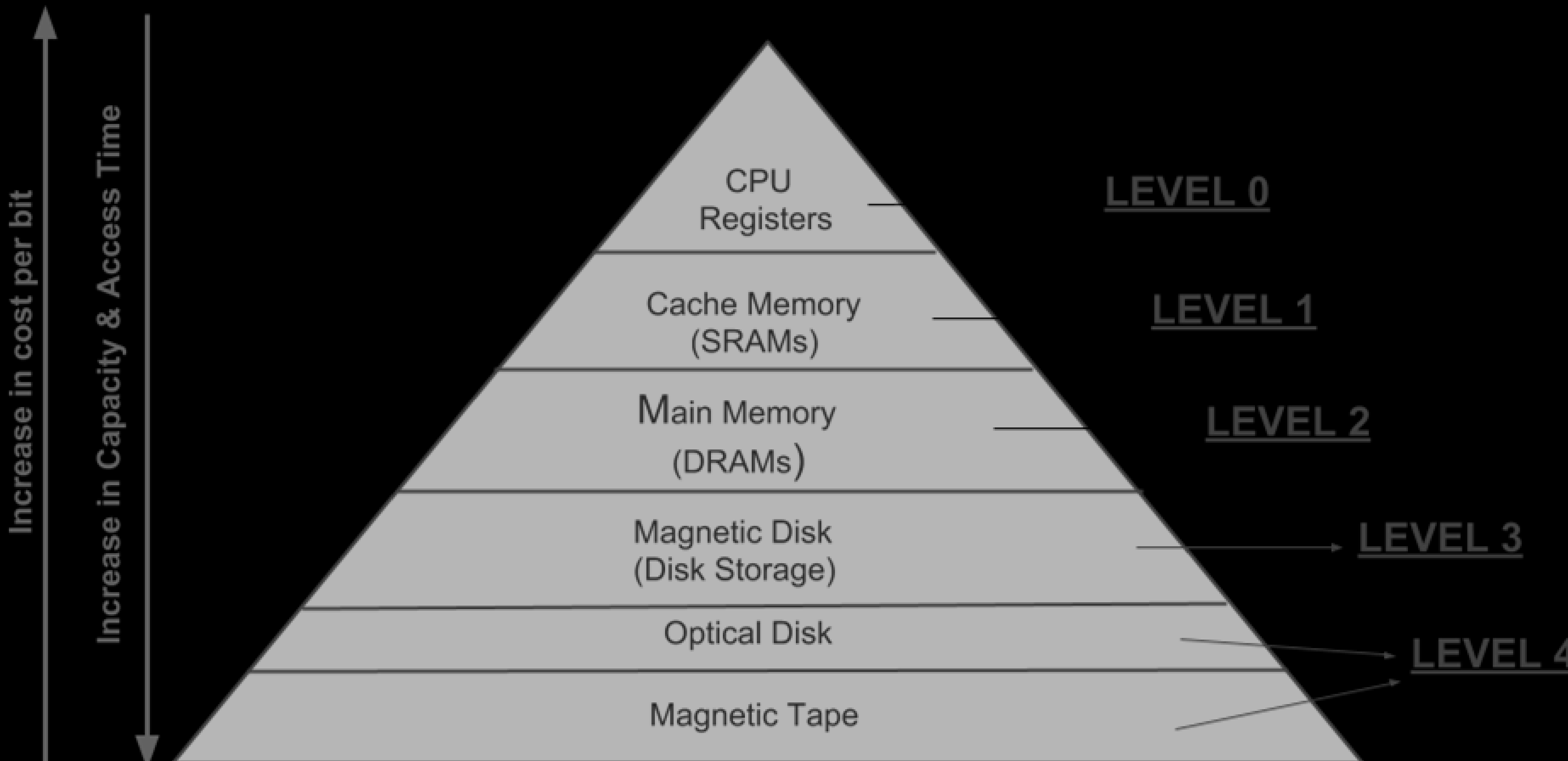


https://intellipaat.com/mediaFiles/2015/08/Resilient-Distributed-Datasets-RDDs.jpg

RDD Traits

- In-Memory: data inside RDD is stored in memory as much (size) and long (time) as possible
- Immutable (read-only): no change after creation, only transformed using transformations to new RDDs
- Lazily evaluated: RDD data not available/transformed until an action is executed that triggers the execution
- Parallel: process data in parallel
- Partitioned: the data in a RDD is partitioned and then distributed across nodes in a cluster
- Cacheable: hold all the data in a persistent "storage" like memory (the most preferred) or disk (the least preferred)

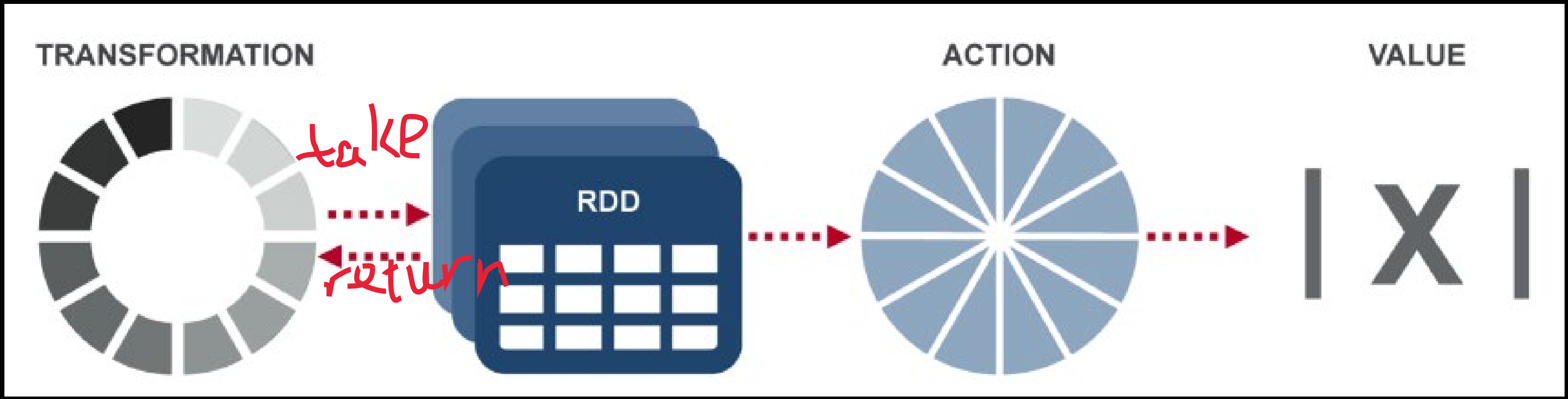
Computer Memory Hierarchy



[Memory Hierarchy Design and its Characteristics - GeeksforGeeks]

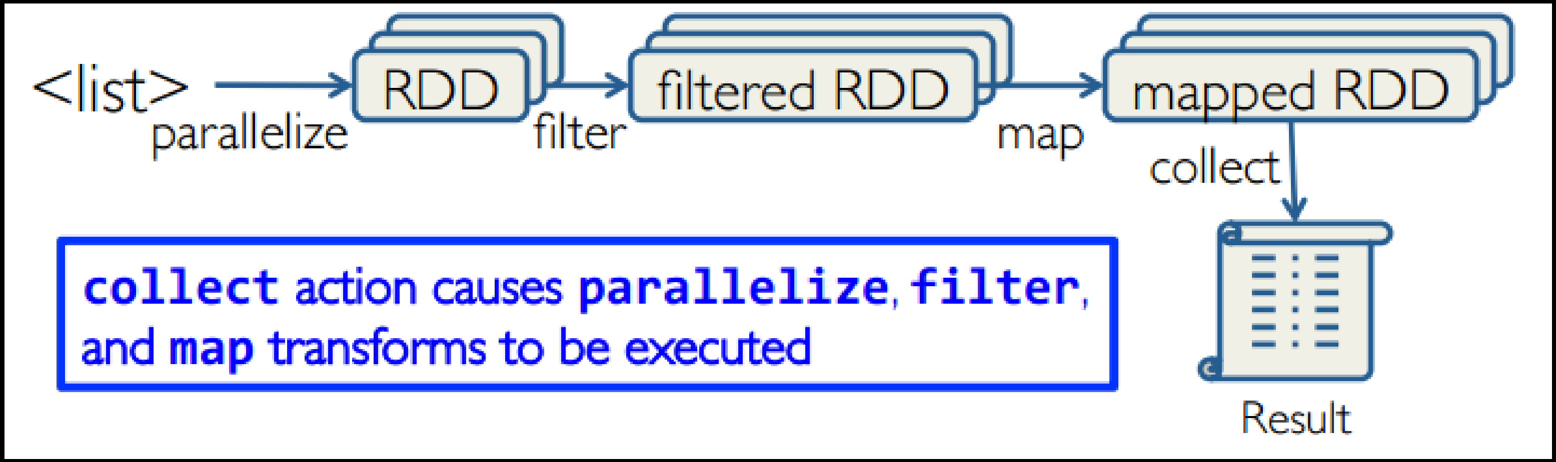
RDD Operations

- Transformation: takes an RDD and returns a new RDD but nothing gets evaluated / computed
- Action: all the data processing queries are computed (evaluated) and the result value is returned



RDD Workflow

- Create an RDD from a data source, e.g. RDD or file
- Apply transformations to an RDD, e.g., map, filter
- Apply actions to an RDD, e.g., collect, count
- Users to control 1) persistence, 2) partitioning



Creating RDDs

- Parallelize existing Python collections (lists)
- Transform existing RDDs
- Create from (HDFS, text, Amazon S3) files
- [sc APIs](): sc.parallelize, sc.hadoopFile, sc.textFile

Spark Transformations

- Create new datasets from an existing one
- Lazy evaluation: just remember transformations applied to the base dataset (results not computed)
  - Spark optimises the required calculations
  - Spark recovers from failures

Transformation	Meaning
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.
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).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.

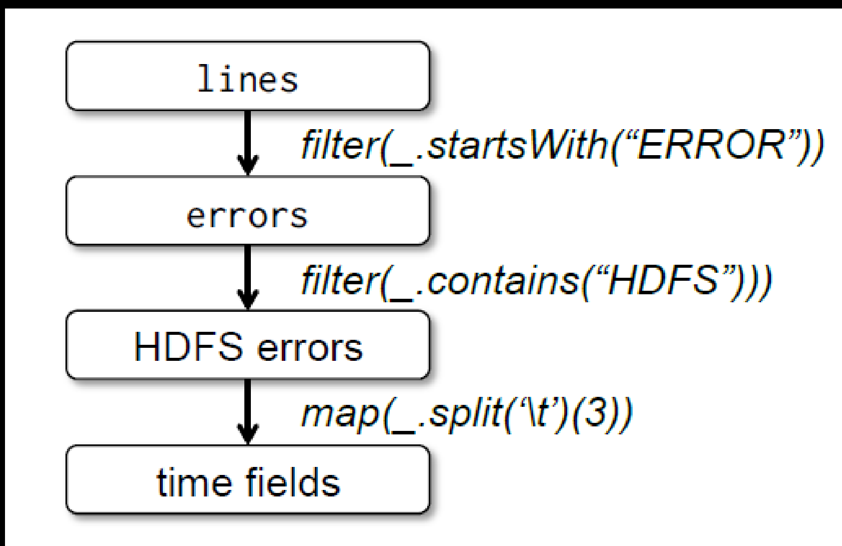
Spark Actions

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

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.

Example from the Spark Paper (2012)

- Web service is experiencing errors. Operators want to search terabytes of logs in the Hadoop file system to find the cause.



Lineage Graph

Code in Scala

```
//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: create RDD from an HDFS file (but NOT loaded in memory)
- Line3: ask for errors to persist in memory (when loaded)

Lineage Graph → Fault-Tolerance

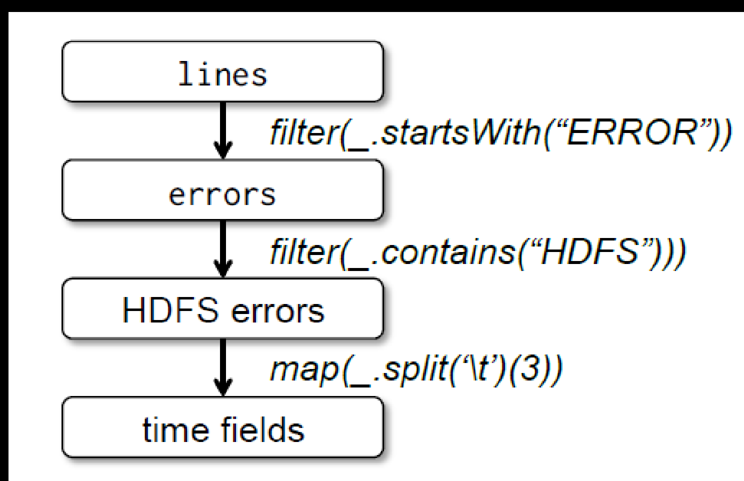
- RDDs keep track of **lineage** → how it was derived from
how
to compute its partitions from data in **stable** storage
- A partition of errors is lost → rebuild it by applying a filter on **only** the corresponding partition of lines → partitions can be recomputed in parallel on different nodes without rolling back the **whole** program

RDD1

RDD2

RDD3

RDD4

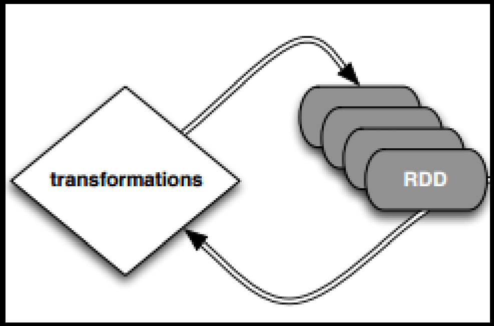


assuming time is field number 3 in a tab-separated format

Operations – Step by Step

//base RDD

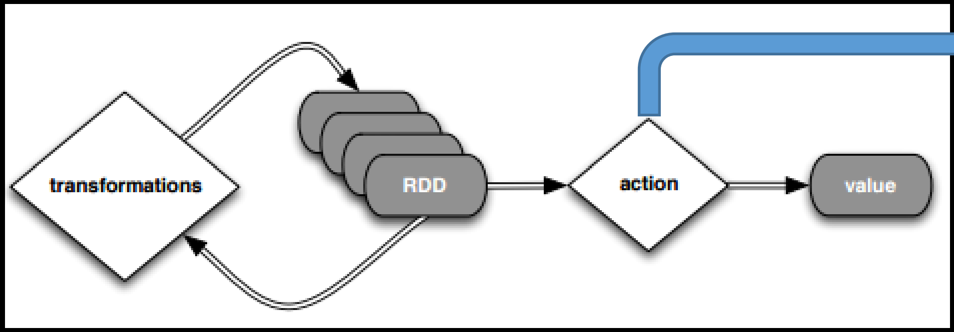
val lines = sc.textFile("hdfs://...")



//Transformed RDD

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

errors.persist()



errors.count()

count() causes Spark to:

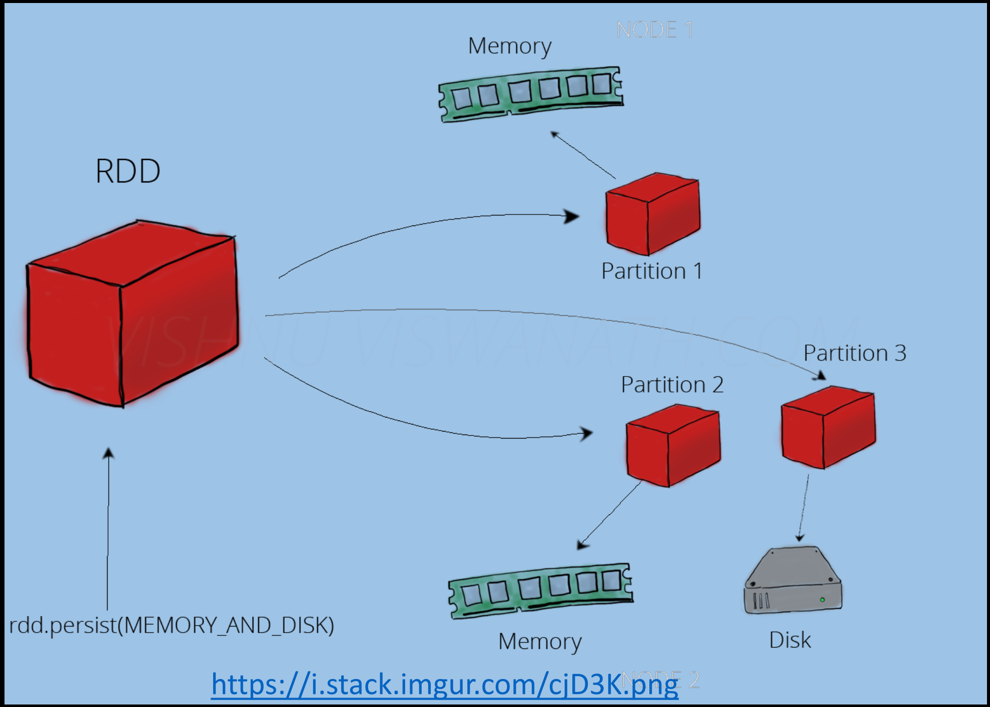
1) read data;
2) sum within partitions;
3) combine sums in driver

Put transform and action together:

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

RDD Persistence

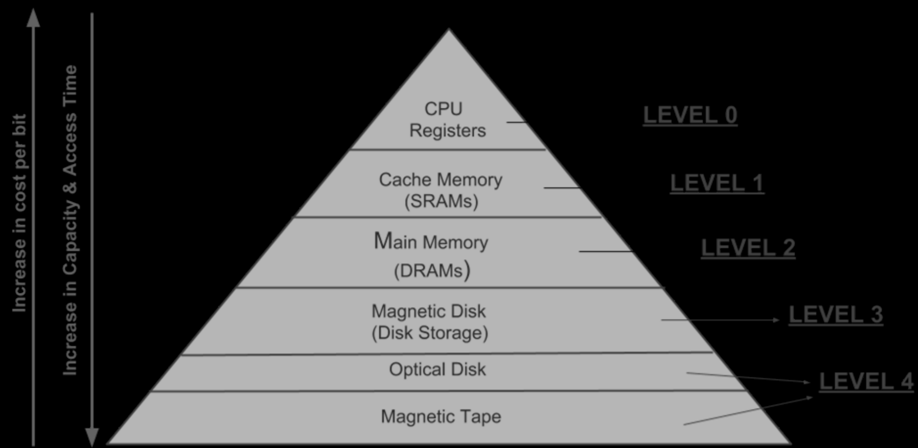
- Nodes store partitions for reuse in other actions on that dataset
- Storage levels for each persisted RDD
  - MEMORY_ONLY (default)
  - MEMORY_AND_DISK: if unfit in memory, store the unfit partitions on disk
  - Unfit partitions: to be recomputed when needed
- cache() = persist(StorageLevel.MEMORY_ONLY)



https://i.stack.imgur.com/cjD3K.png

Why Persisting RDD?

```
val lines = sc.textFile("hdfs://...")
val errors = lines.filter(_.startsWith("Error"))
errors.persist()
errors.count()
```



- errors.count() again → file reload and re-computation
- Persist → cache the data in memory → reduce the data loading cost for further actions on the same data
- erros.persist(): do nothing (a lazy operation, telling "*read this file and then cache the contents*"). An action will trigger computation and data caching.

Spark Key-Value RDDs

- Spark supports key-value pairs

groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a 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. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.

Week 2 Contents / Objectives

- Resilient Distributed Datasets
- DataFrames and Datasets
- Machine Learning Pipelines
- Execution Parallelization

Why DataFrame?

- Challenges
  - ETL to/from various semi/unstructured data sources
  - Advanced analytics (e.g. machine learning) are hard to express in relational systems
- Solutions
  - A DataFrame API to perform relational operations on both external data sources and Spark's built-in RDDs
  - A highly extensible optimizer Catalyst *to* use Scala features to add composable rule, control code generation, and define extensions

DataFrame-based API for MLlib

- In v2.0, the DataFrame-based API became the primary API for MLlib
  - Voted by the community
  - org.apache.spark.ml, pyspark.ml
- The RDD-based API entered the maintenance mode
  - Still maintained with bug fixes, but no new features
  - org.apache.spark.mllib, pyspark.mllib

Announcement: DataFrame-based API is primary API

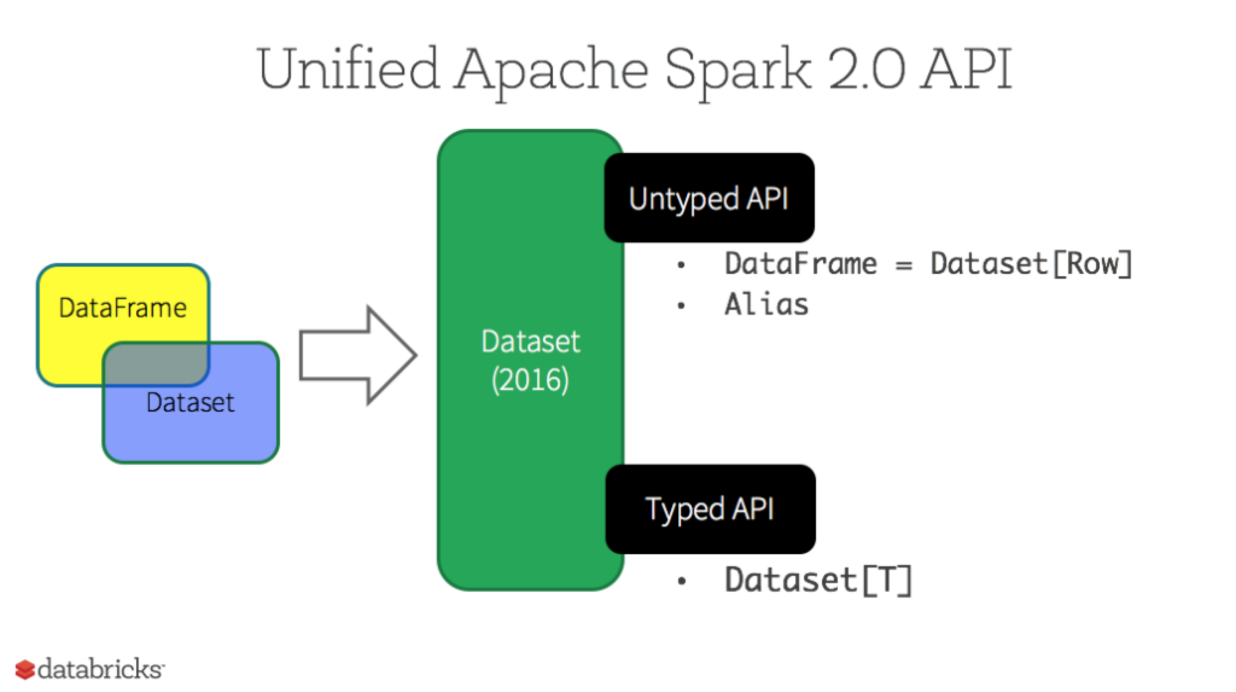
The MLlib RDD-based API is now in maintenance mode.

As of Spark 2.0, the RDD-based APIs in the `spark.mllib` package have entered maintenance mode. The primary Machine Learning API for Spark is now the DataFrame-based API in the `spark.ml` package.

What are the implications?

DataFrames and Datasets

- DataFrame: schema, generic untyped (like a table)
- Dataset: static typing, strongly-typed
- DataFrame = Dataset[Row] (Row: generic untyped)
  - Dataset organised into named columns



Typed and Un-typed APIs

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

* 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. CSV
- Ease-of-use of APIs with structure
  - Rich semantics and domain specific operations
- Performance and optimization
  - SQL Catalyst

DataFrame

- 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

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

Data grouped into named columns

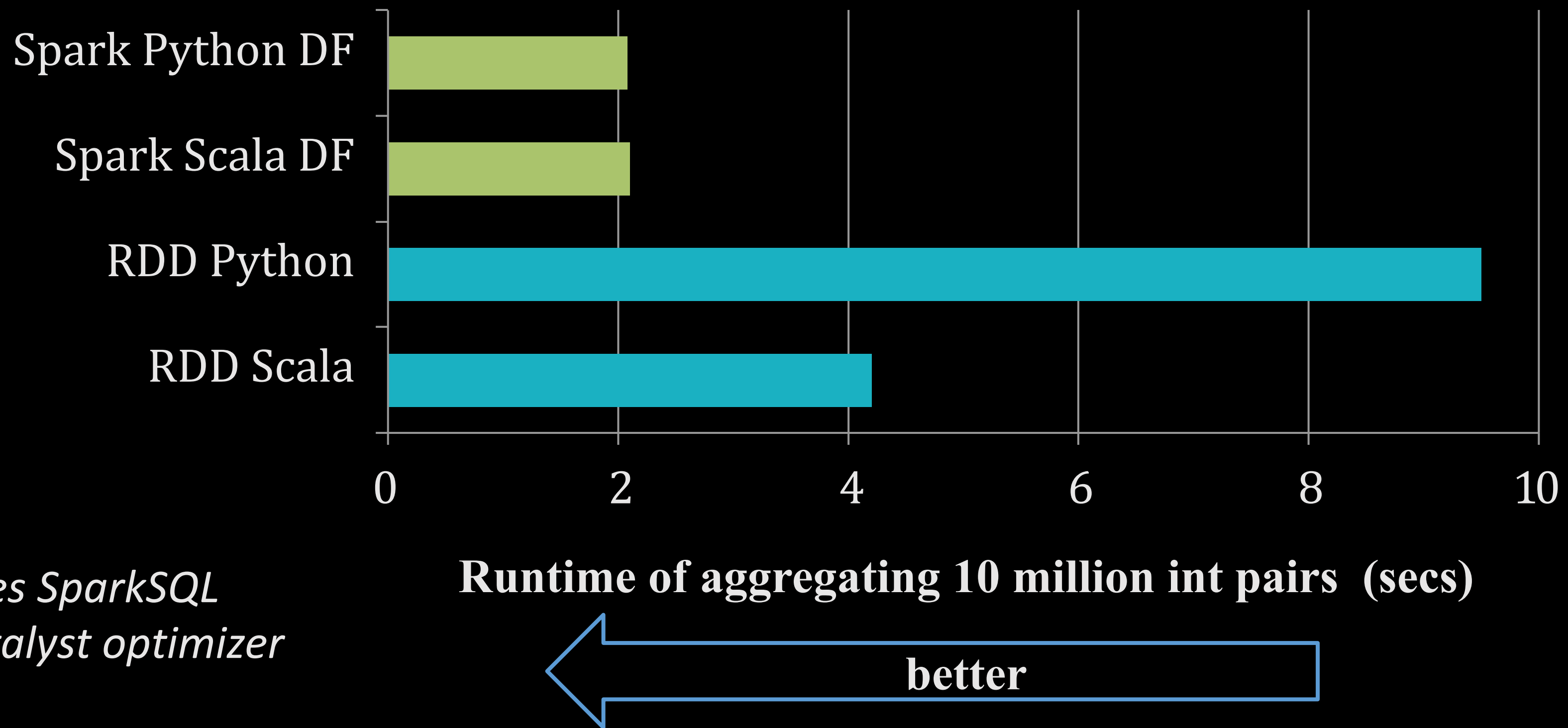
RDD API

```
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

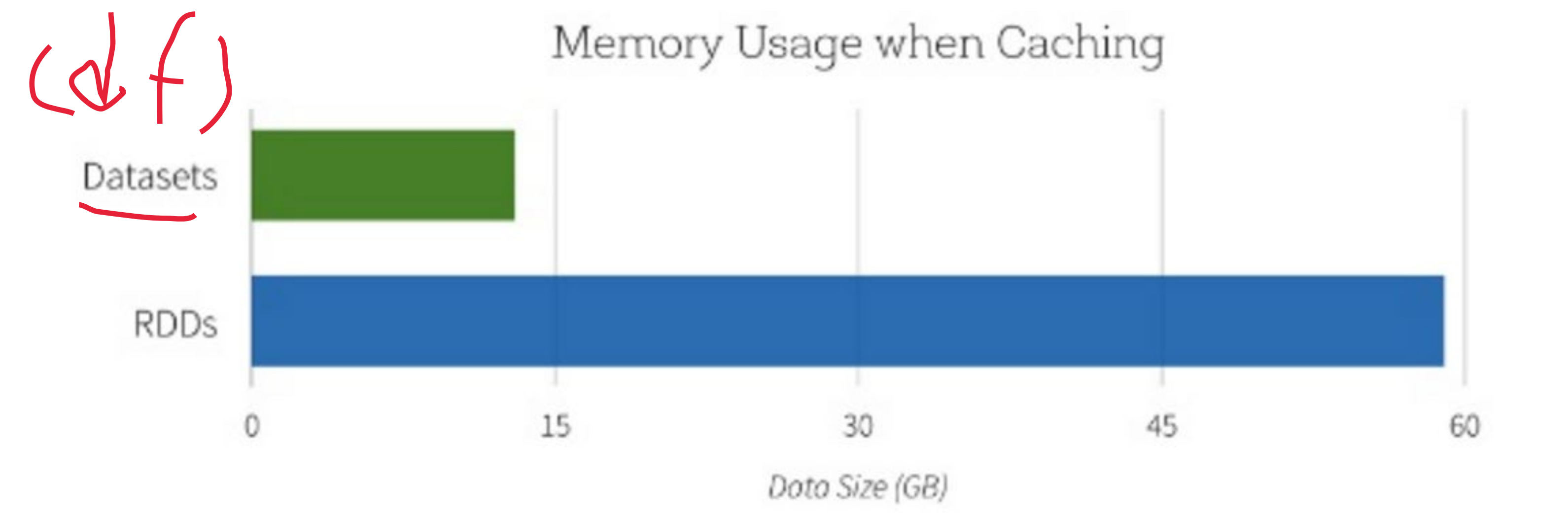
```
data.groupBy("dept").avg("age")
```

Spark DataFrames are Fast



Uses SparkSQL Catalyst optimizer

Space Efficiency



Week 2 Contents / Objectives

- Resilient Distributed Datasets
- DataFrames and Datasets
- Machine Learning Pipelines
- Execution Parallelization

Machine Learning Library (MLlib)

- ML algorithms: common ML algorithms for regression, classification, clustering, and collaborative filtering

PCA

- Featurization: feature extraction, transformation, dimensionality reduction, and selection
- Pipelines: tools for constructing, evaluating, and tuning ML pipelines
- Persistence: save/load algorithms, models, & pipelines
- Utilities: linear algebra, statistics, data handling, …

Main Concepts in Pipelines

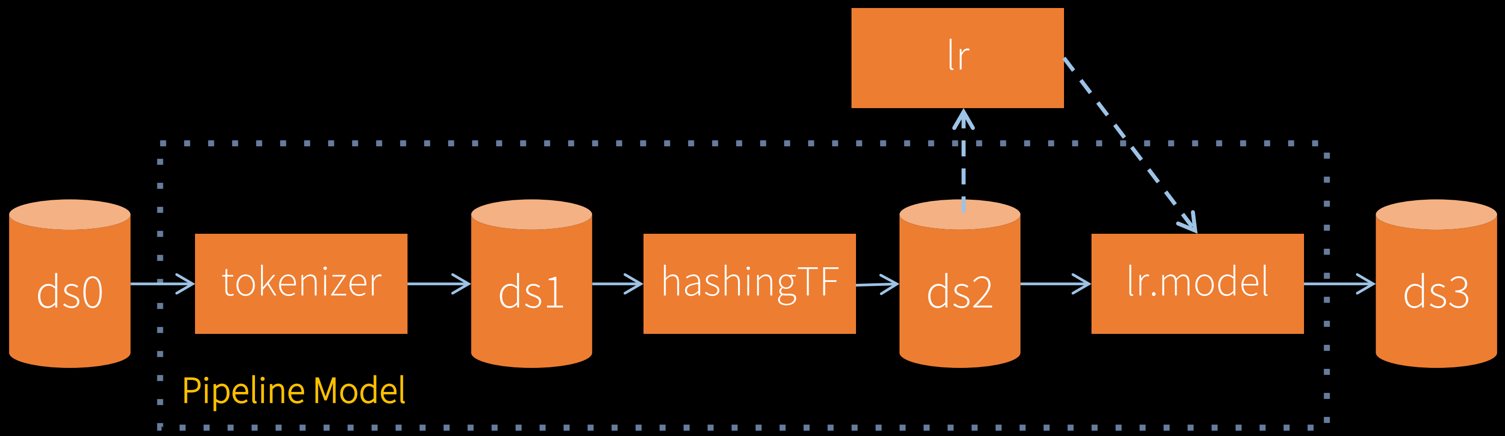
- DataFrame: an ML dataset holding various data types, e.g. columns for text, feature vectors, true labels, & predictions
- Transformer: algorithm transforming one DataFrame into another, e.g. features → ML model → predictions
- Estimator: algorithm fitting on a DataFrame to produce a Transformer, e.g. training data→ ML algorithm → ML 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

- High-level APIs to create and tune ML 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 = spark.read.load("/path/to/data")
model = pipeline.fit(df)
```



Example: Text Classification

Goal: Given a text document, predict its topic.

<u>Features</u>

```
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...
```

⟹

<u>Label</u>

1: about science
0: not about science

Dataset: "20 Newsgroups"
From UCI KDD Archive

ML Workflow

Load data

↓

Extract features

↓

Train model

↓

Evaluate

Load Data

Load data
↓
Extract features
↓
Train model
↓
Evaluate

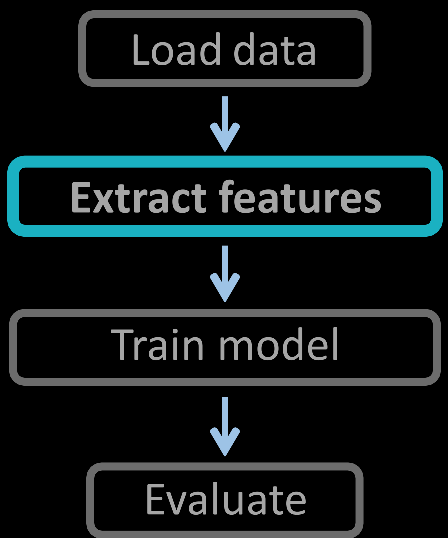
Current data schema

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

Data sources for DataFrames

built-in	external
Parquet, MySQL, JSON, amazon web services S3, HIVE, HDFS	AVRO, CSV, Apache HBASE, cassandra, Amazon Redshift, elasticsearch.

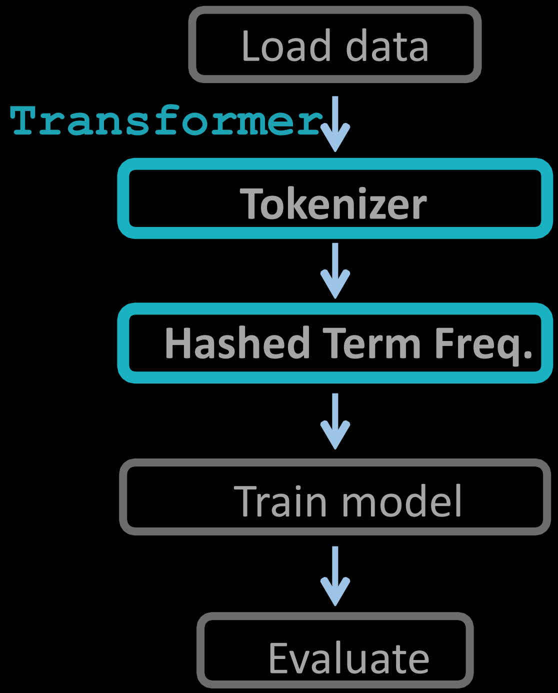
Extract Features



<u>Current data schema</u>

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

Extract Features



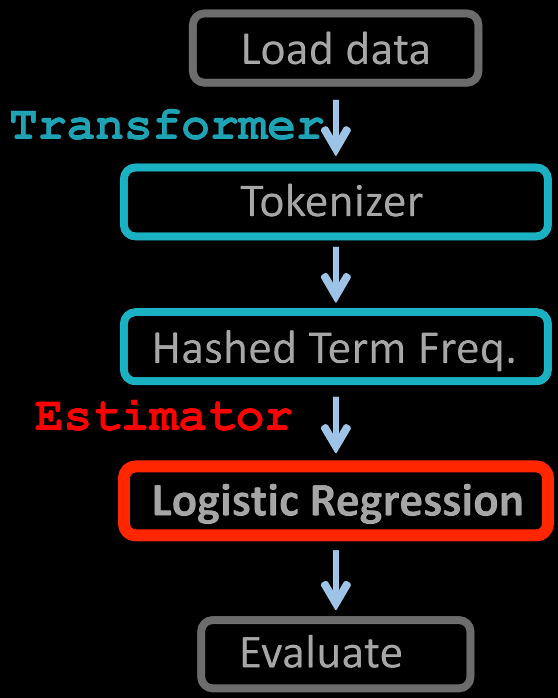
<u>Current data schema</u>

```
label: Int
text: String

words: Seq[String]

features: Vector
```

Train the Model



<u>Current data schema</u>

`label: Int`

`text: String`

`words: Seq[String]`

`features: Vector`

`model parameters (not in DF)`

Evaluate the Model

Load data

↓ Transformer

Tokenizer

↓

Hashed Term Freq.

↓ ~~Estimator~~ Transformer

Logistic Regression

↓ Evaluator

Evaluate

<u>Current data schema</u>

```
label: Int
text: String

words: Seq[String]

features: Vector

prediction: Int
```

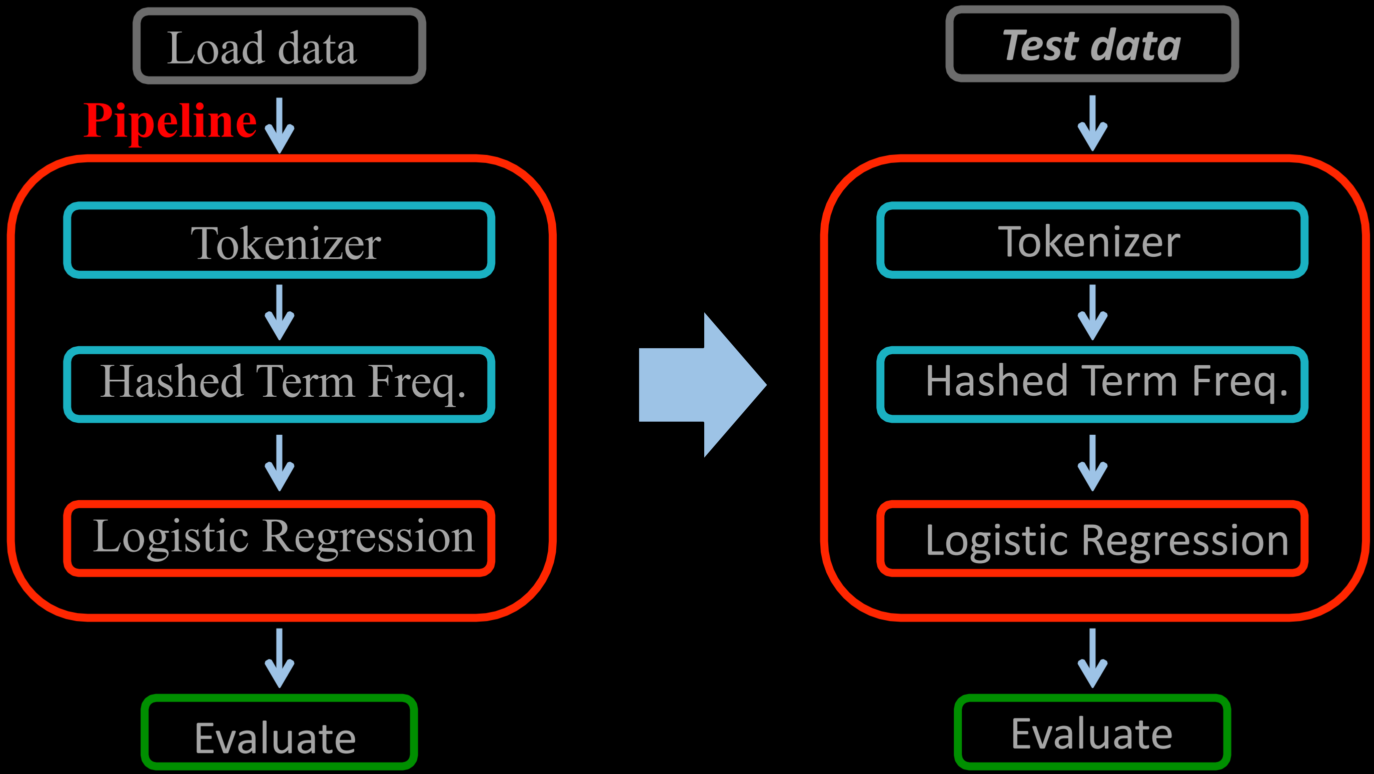
spark

By default, always append new columns

→ Can go back & inspect intermediate results

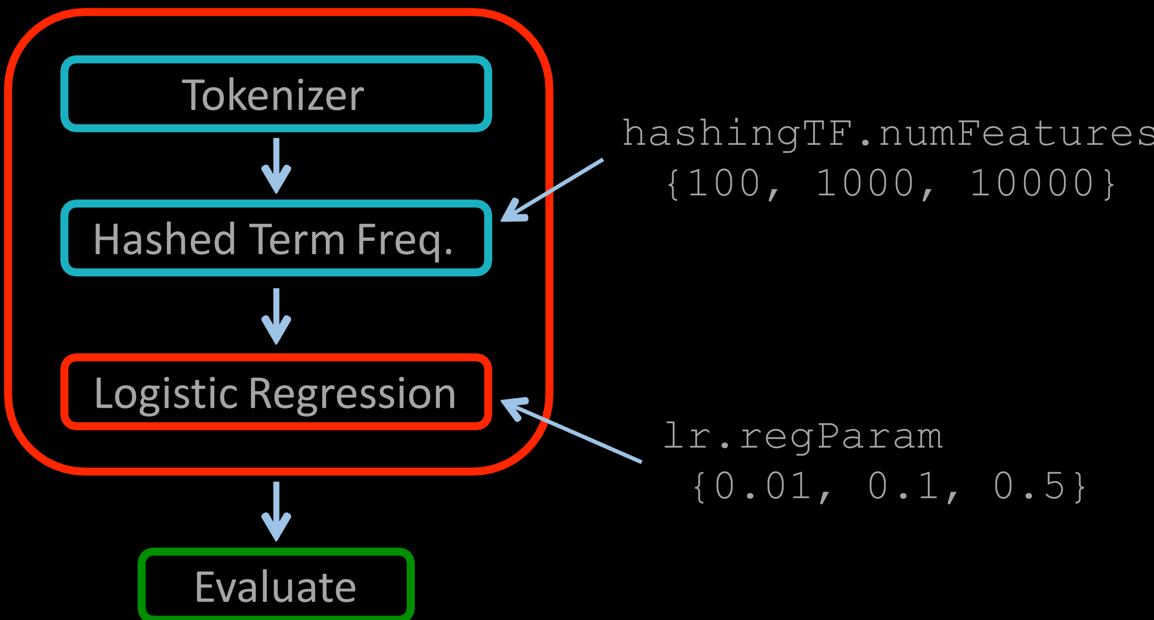
→ Made efficient by DataFrame optimizations

ML Pipelines



Re--run exactly the same way

Parameter Tuning



CrossValidator

Given:

- Estimator
- Parameter grid
- Evaluator

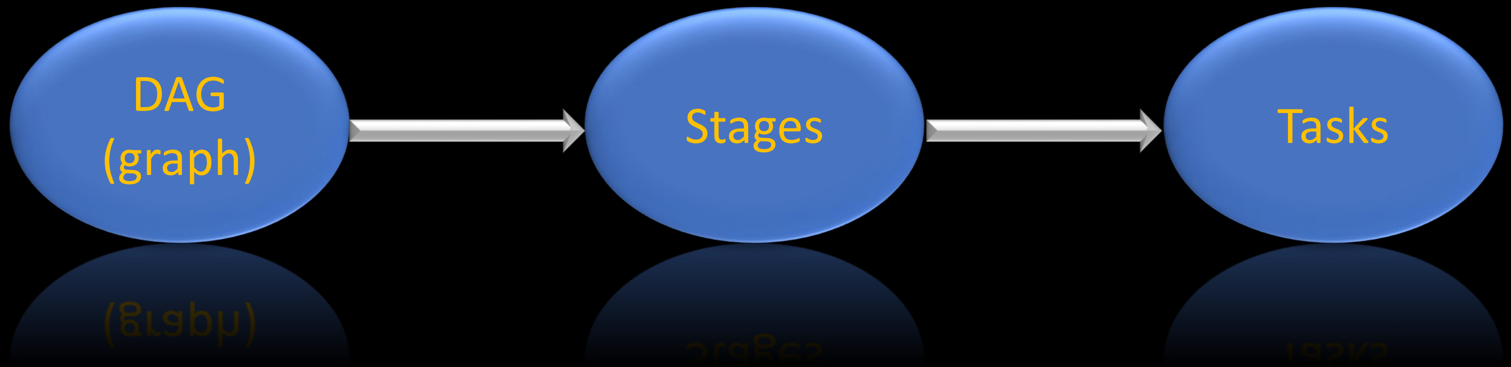
Find best parameters

Week 2 Contents / Objectives

- Resilient Distributed Datasets
- DataFrames and Datasets
- Machine Learning Pipelines
- Execution Parallelization

How Spark Works

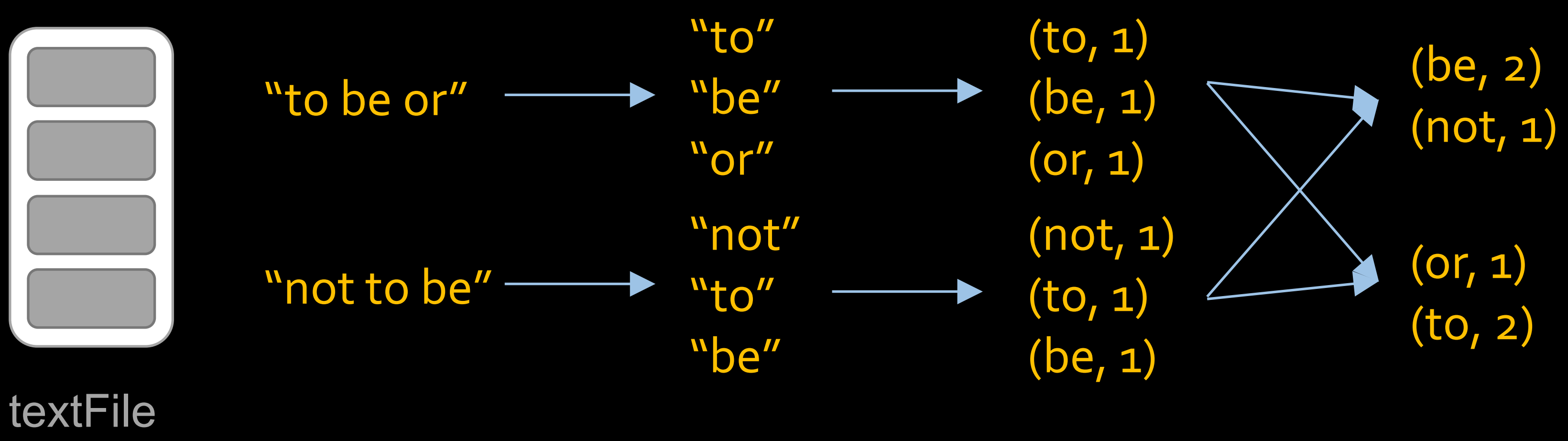
- User applications create RDDs/DFs, transform them, and run actions
- This results in a DAG (Directed Acyclic Graph) of operators
- DAG is compiled into stages
- Each stage is executed as a series of tasks



Word Count in Spark

val file = sc.textFile("hdfs://...", 4)

RDD[String]

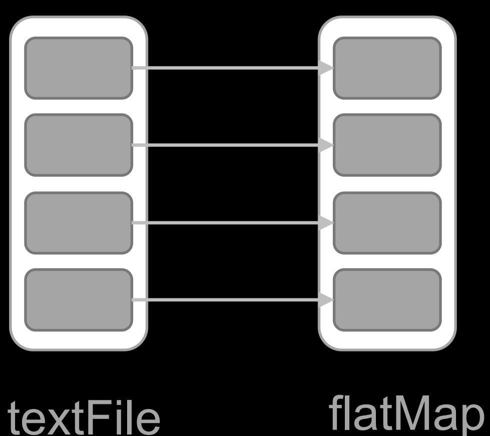


Word Count in Spark

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

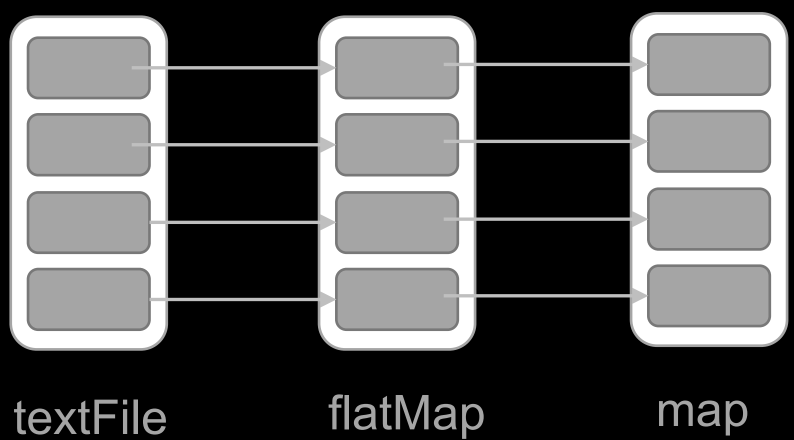
RDD[String]

RDD[List[String]]



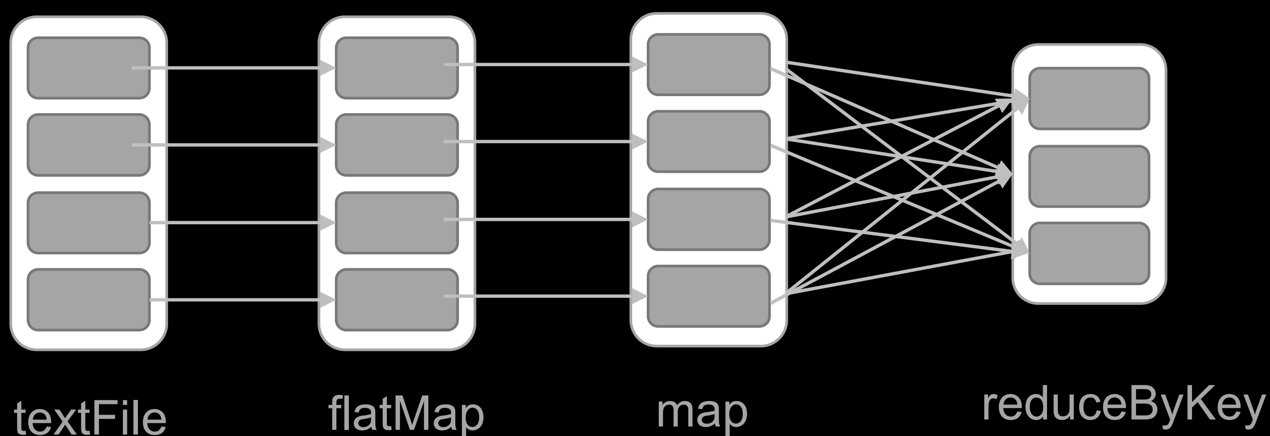
Word Count in Spark

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



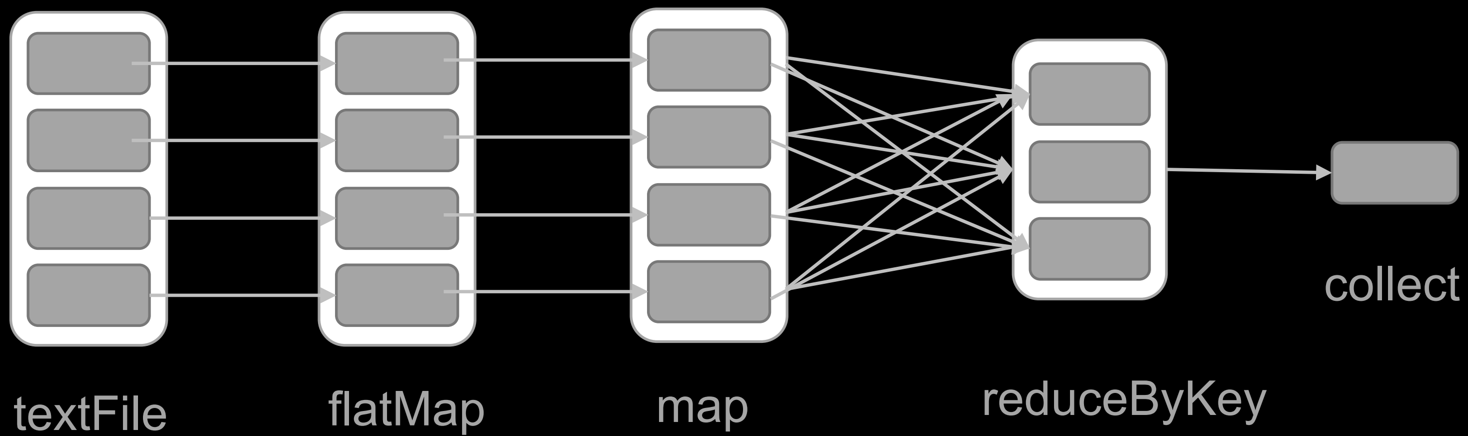
Word Count in Spark

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

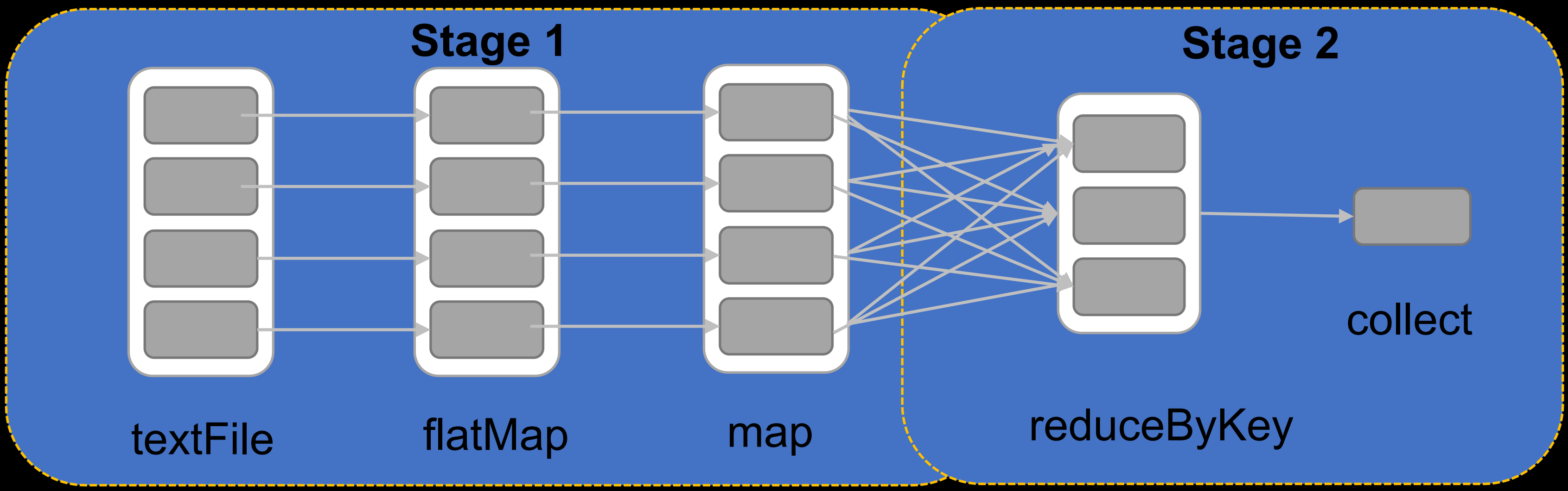


Word Count in Spark

```
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)]
```



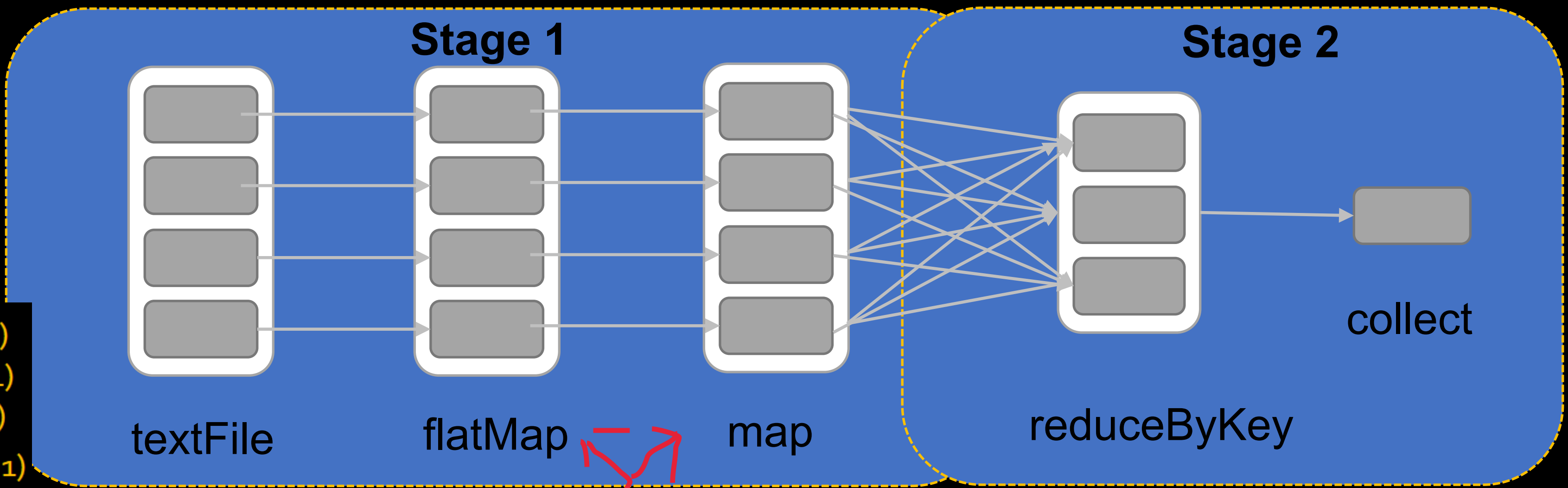
Execution Plan



- 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

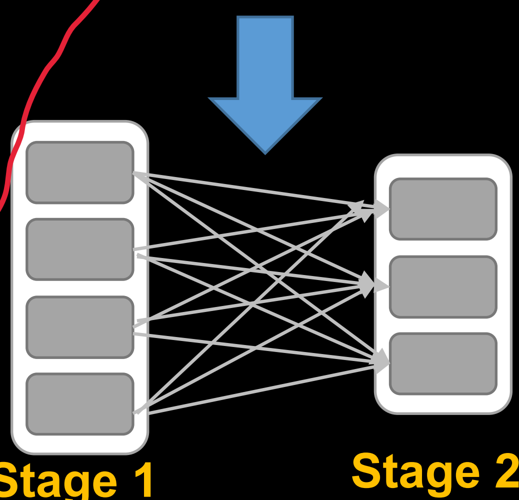
Execution Plan

(to, 1)
(be, 1)
(or, 1)
(not, 1)
(to, 1)
(be, 1)



and

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

Execution of Tasks

- Create a task for each partition in the new RDD
- Compute the task's closure (those variables and methods that must be visible to the worker)
- Serialize the task's closure
- Schedule and ship tasks (closures) to workers

Setting the Level of Parallelism

- Many transformations take an optional parameter numPartitions for number of tasks

distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a 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. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean

Shared Variables (for Cluster)

- Variables are distributed to workers via closures
- When a function is executed on a cluster node, it works on separate copies of those variables that are not shared across workers
- Iterative or single jobs with large global variables
  - Problem: inefficient to send large data with each iteration
  - Solution: Broadcast variables (keep rather than ship)
- Counting events that occur during job execution
  - Problem: Closures are one way driver → worker
  - Solution: Accumulators (only “added” to, e.g. sums/counters)

Recommended Reading

- A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets
- Sections 2.4.2 and 2.4.3 of the MMDS book (3rd edition)
- Hyperlinks in slides
- Suggested reading in Lab 2