

# Spark

Giovanni Simonini

*Slides partially taken from  
the Spark Summit, and Amp Camp:  
<http://spark-summit.org/2014/training>  
<http://ampcamp.berkeley.edu/>*

DBGroup  
Università di Modena e Reggio Emilia  
Dipartimento di Ingegneria 'Enzo Ferrari'

# SPARK INTRODUCTION

MapReduce let users write programs for parallel computations using a set of high-level operators:

- without having to worry about:
  - distribution
  - fault tolerance
- giving abstractions for accessing a cluster's computational resources
- but lacks abstractions for leveraging distributed memory
- between two MR jobs writes results to an external stable storage system, e.g., HDFS

! Inefficient for an important class of emerging applications:

- iterative algorithms
  - those that reuse intermediate results across multiple computations
  - e.g. Machine learning and graph algorithms
- interactive data mining
  - where a user runs multiple ad-hoc queries on the same subset of the data

Spark handles current computing frameworks' inefficiency (iterative algorithms and interactive data mining tools)

## How?

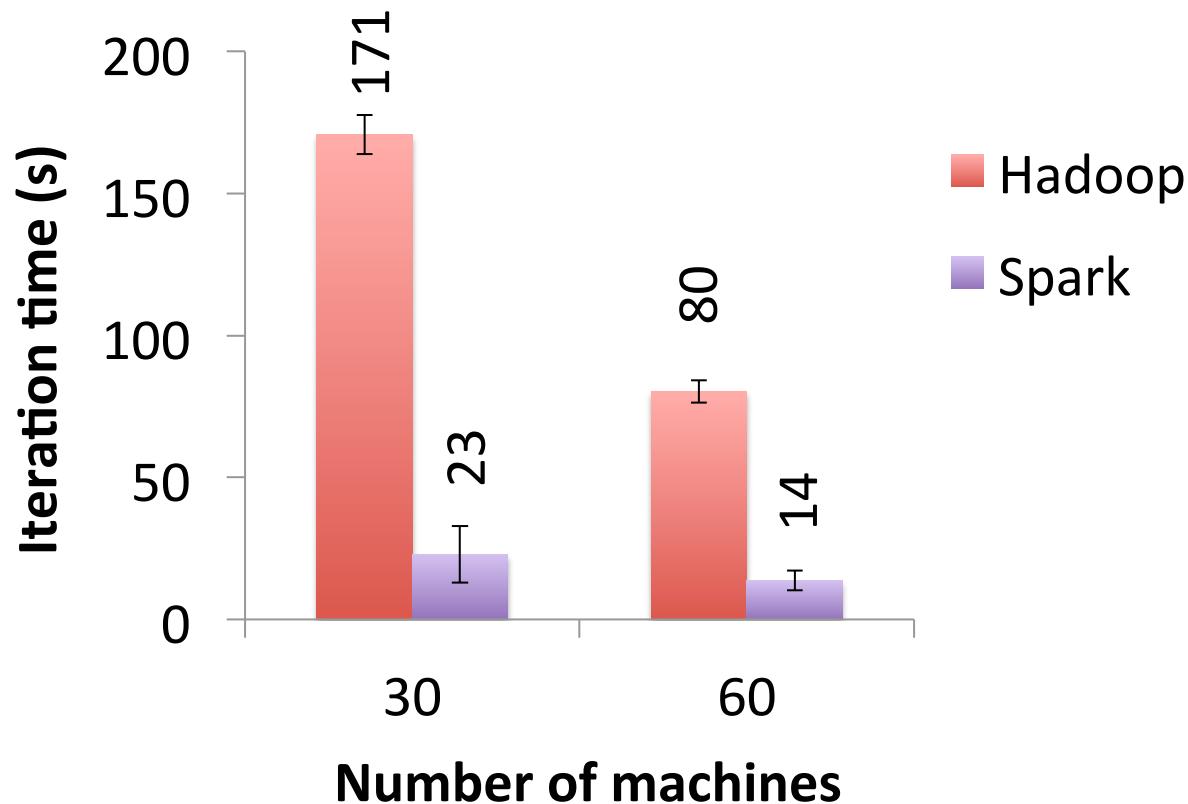
- keeping data in memory can improve performance by an order of magnitude
  - Resilient Distributed Datasets (RDDs)
- up to 20×/40x faster than Hadoop for iterative applications

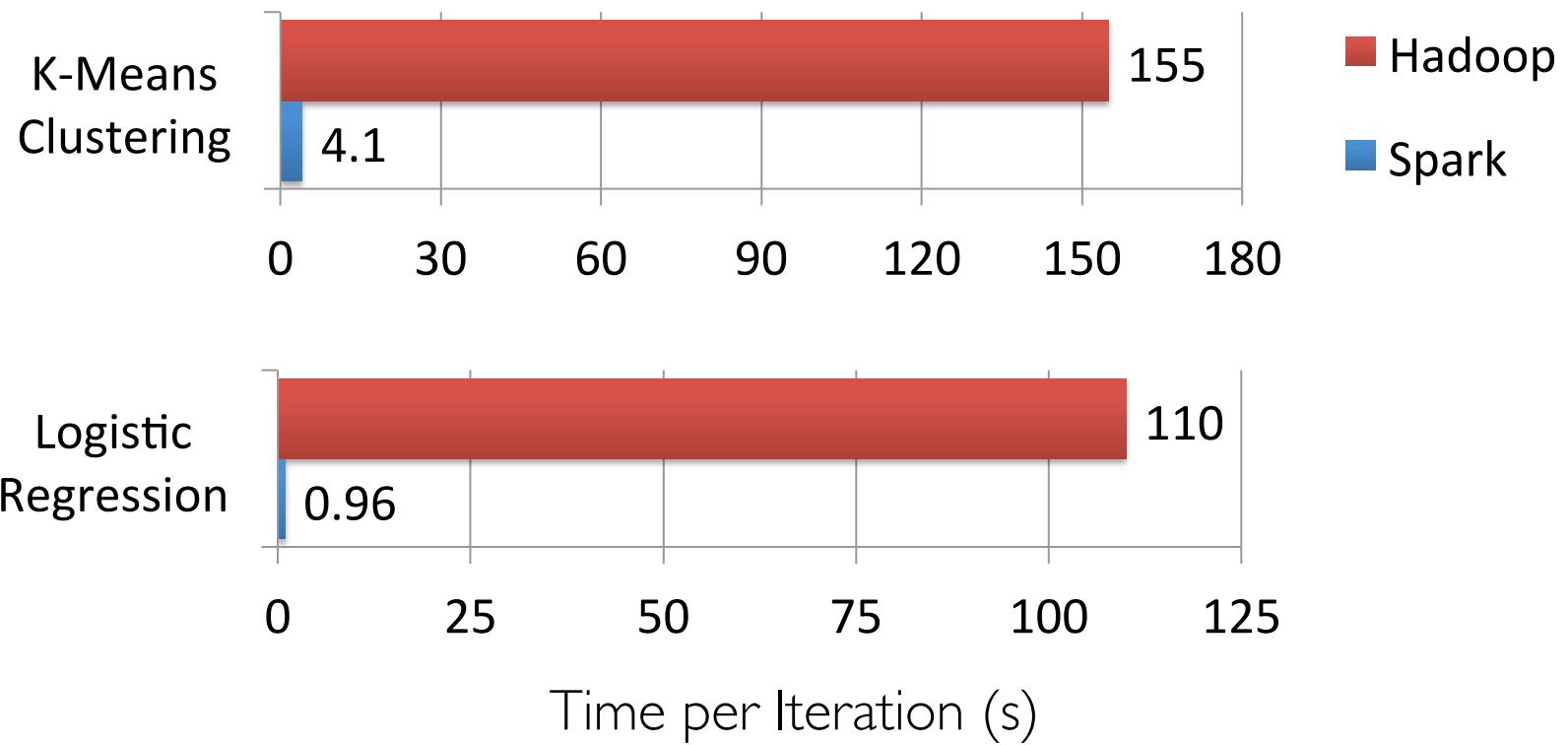
### RDDs



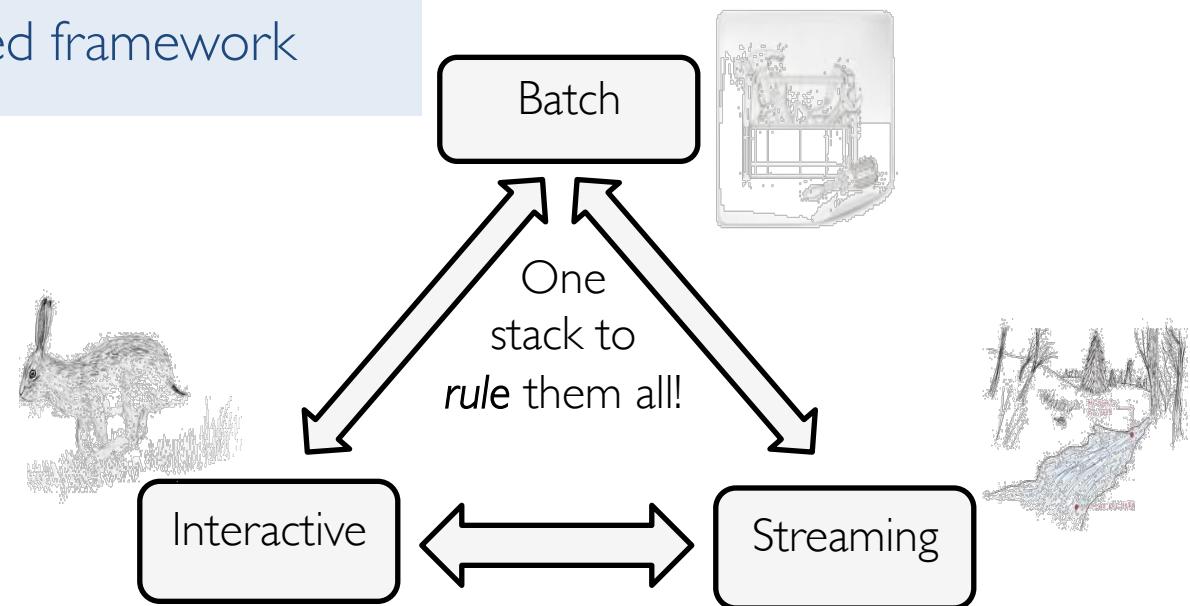
RDDs provide a restricted form of shared memory:

- based on coarse-grained transformations
- RDDs are expressive enough to capture a wide class of computations
  - including recent specialized programming models for iterative jobs, such as Pregel (Giraph)
  - and new applications that these models do not capture



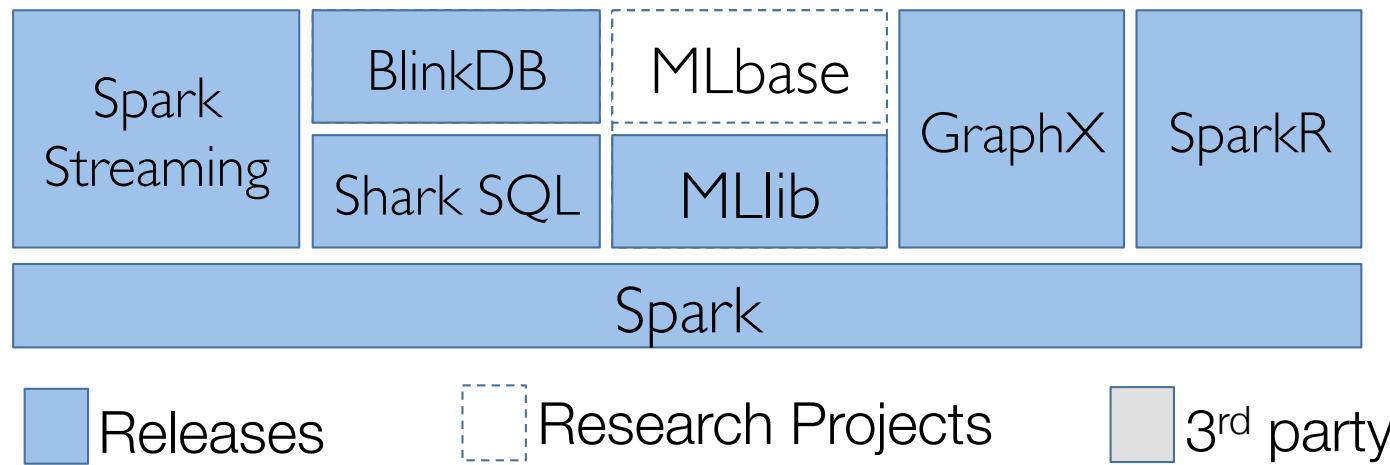


Support **batch**, **streaming**, and **interactive** computations in a unified framework



- Easy to combine batch, streaming, and interactive computations
- Easy to develop sophisticated algorithms
- Compatible with existing open source ecosystem (Hadoop/HDFS)

# Stack (Feb, 2014)



RDDs are fault-tolerant, parallel data structures:

- let users to explicitly:
  - persist intermediate results in memory
  - control their partitioning to optimize data placement
  - manipulate them using a rich set of operators
- RDDs provide an interface based on coarse-grained transformations (e.g., map, filter and join) that apply the same operation to many data items
  - This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its lineage)
- If a partition of an RDD is lost:
  - the RDD has enough information about how it was derived from other RDDs to re-compute just that partition

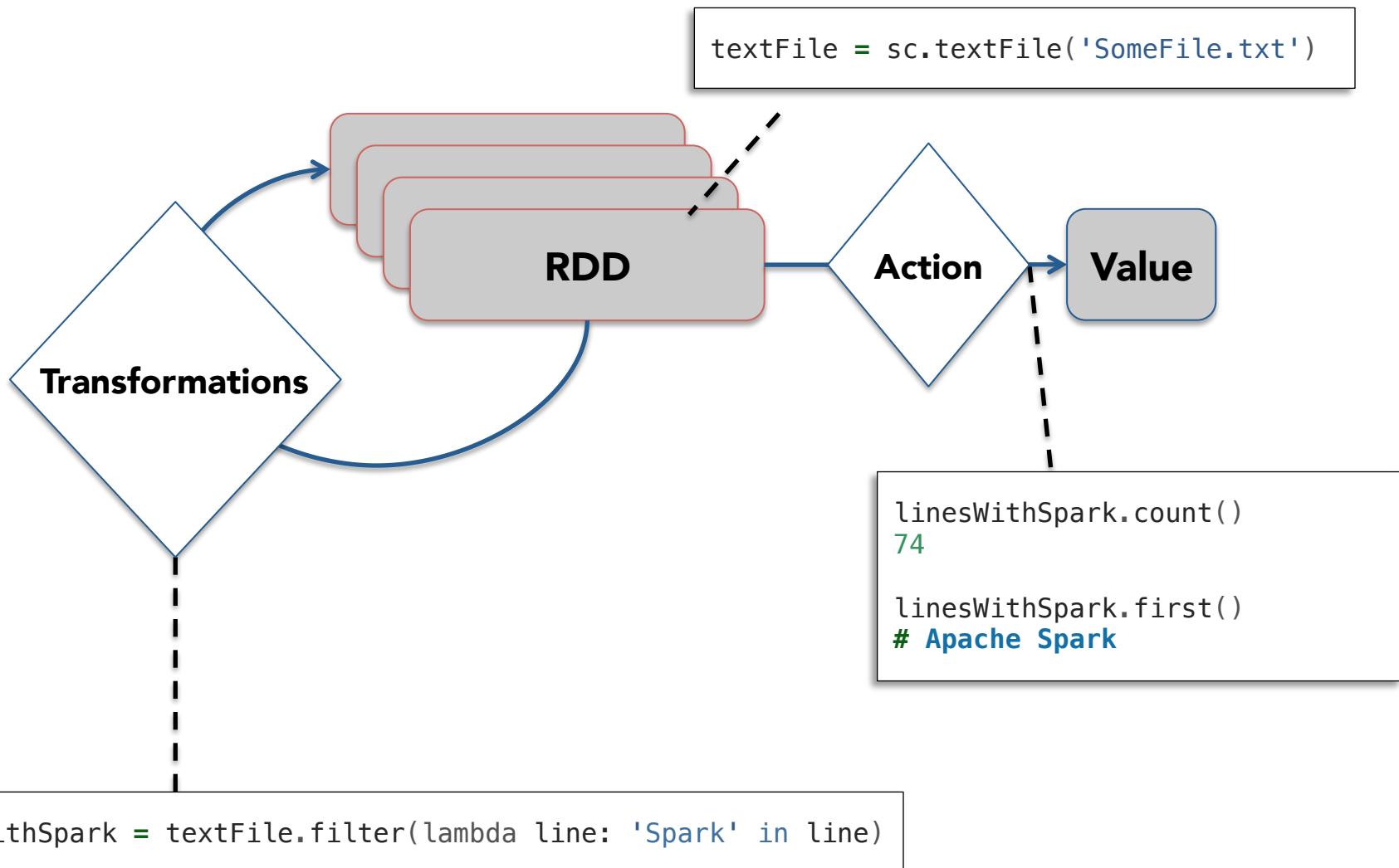
# Write programs in terms of transformations on distributed datasets

## Resilient Distributed Datasets

- Collections of objects spread across a cluster, stored in RAM or on Disk
- Built through parallel transformations
- Automatically rebuilt on failure

## Operations

- Transformations  
(e.g. map, filter, groupBy)
- Actions  
(e.g. count, collect, save)



Load error messages from a log into memory, then interactively search for various patterns

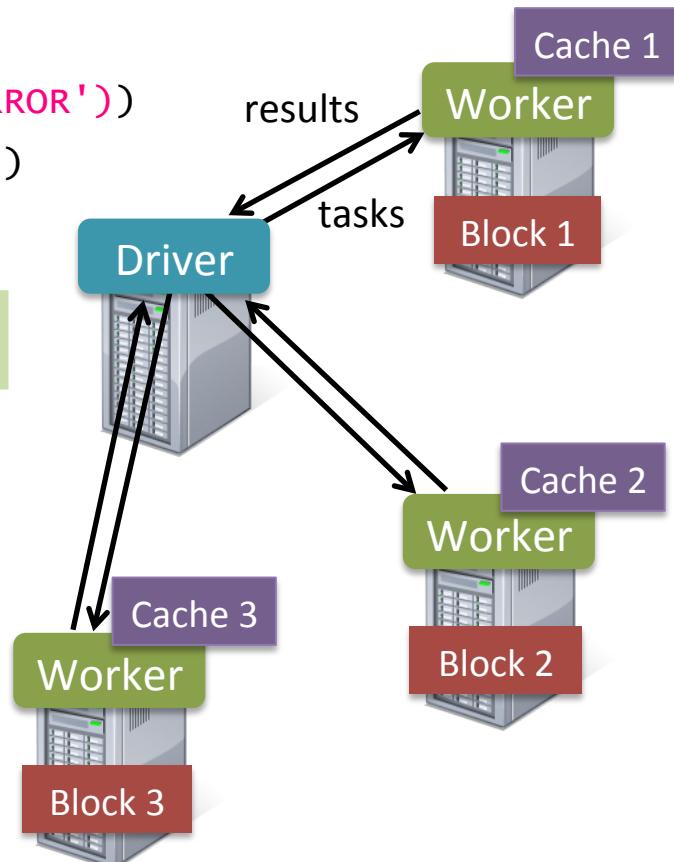
Base RDD

```
lines = spark.textFile('hdfs://...')  
errors = lines.filter(lambda s: s.startswith('ERROR'))  
messages = errors.map(lambda s: s.split('\t')[2])  
messages.cache()
```

Transformed RDD

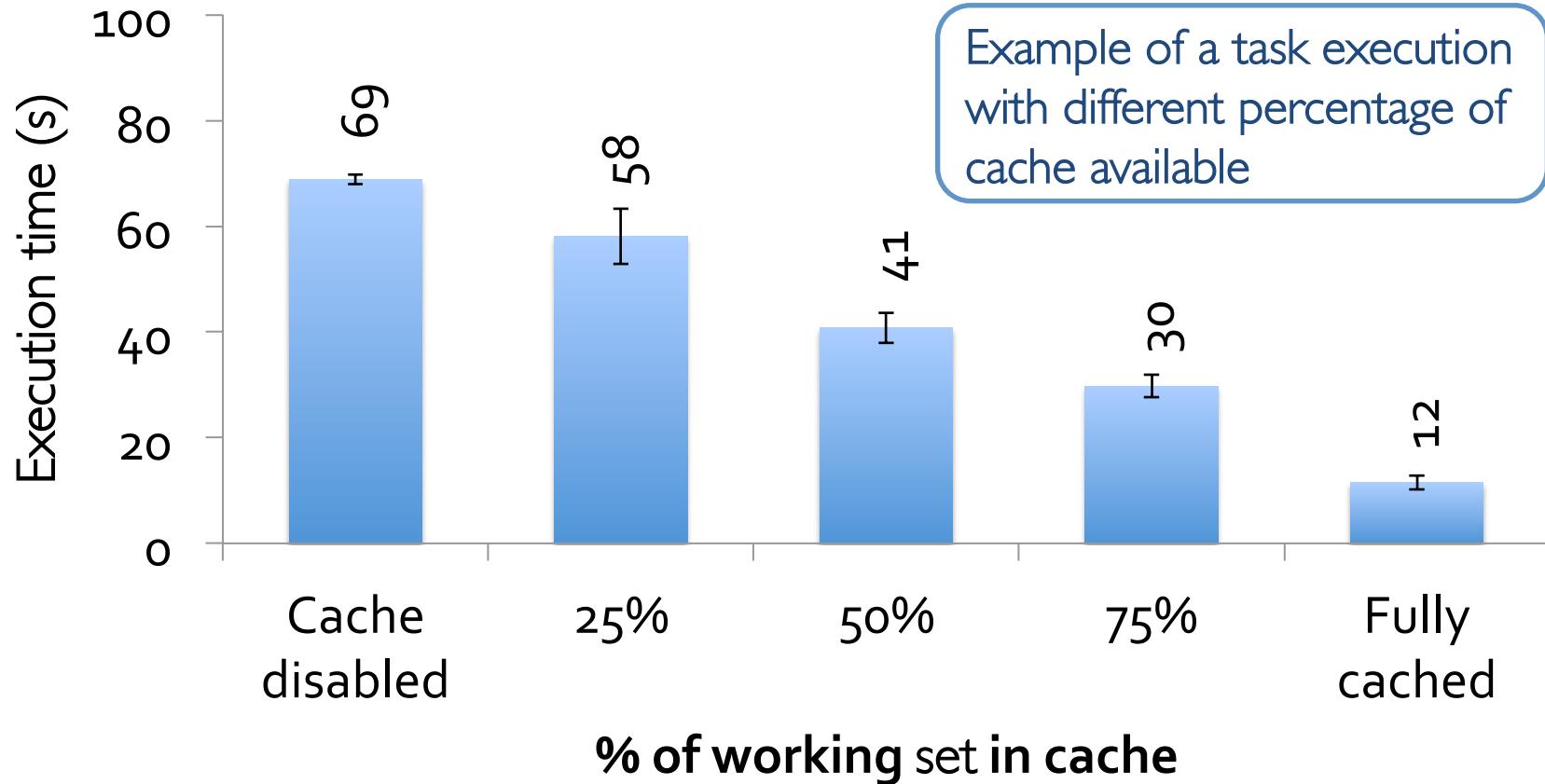
Action: here is launched the computation  
(Lazy Evaluation)

```
messages.filter(lambda s: 'mysql' in s).count()  
messages.filter(lambda s: 'php' in s).count()  
...
```



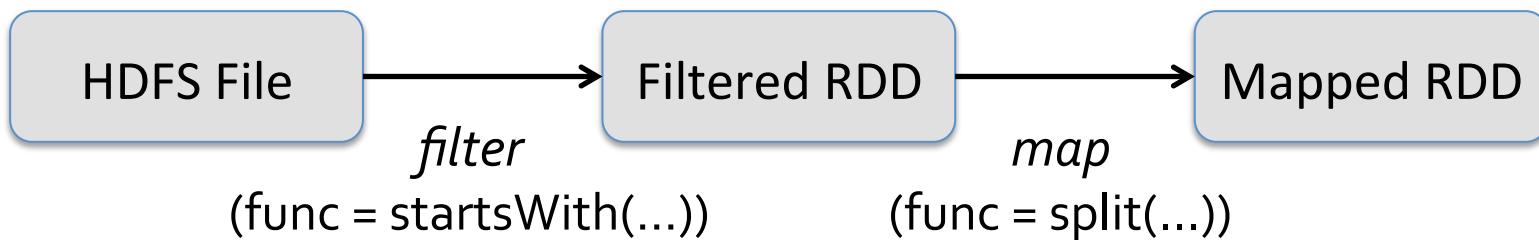
if you don't have enough memory, Spark *degrade gracefully*

- User can define custom policies to allocate memory to RDDs



RDDs track *lineage* information that can be used to efficiently re-compute lost data

```
msgs = textFile.filter(lambda s: s.startswith('ERROR'))  
    .map(lambda s: s.split('\t')[2])
```



## Python

```
lines = sc.textFile(...)  
lines.filter(lambda s: 'ERROR' in s).count()
```

## Scala

```
val lines = sc.textFile(...)  
lines.filter(x => x.contains('ERROR')).count()
```

## Java

```
JavaRDD<String> lines = sc.textFile(...);  
lines.filter(new Function<String, Boolean>() {  
    Boolean call(String s) {  
        return s.contains('error');  
    }  
}).count();
```

## Standalone Programs

- Python, Scala, & Java

## Interactive Shells

- Python & Scala

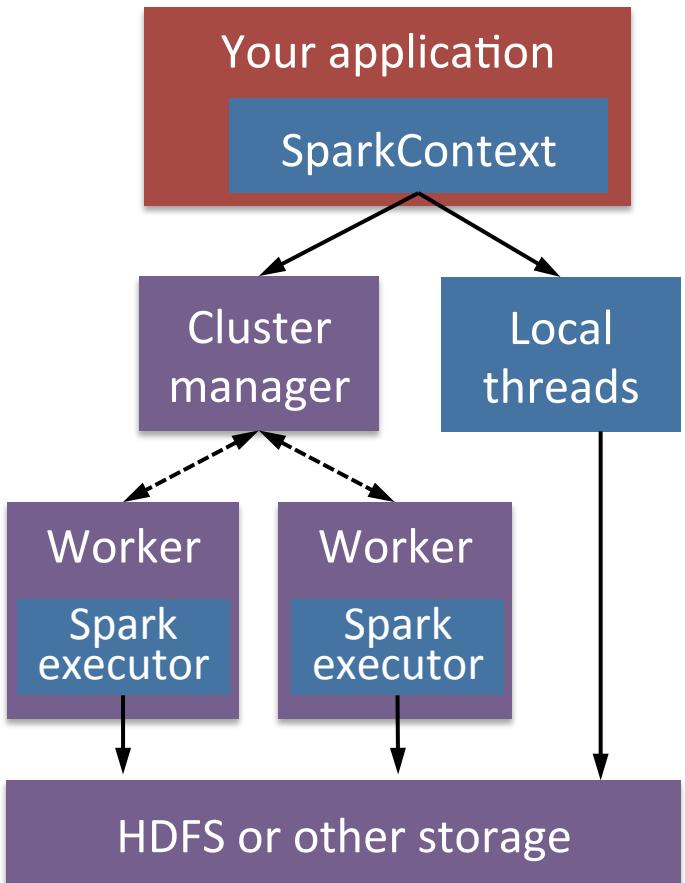
## Performance

- Java & Scala are faster due to static typing
- ...but Python is often fine

- The Fastest Way to Learn Spark
  - Available in Python and Scala
  - Runs as an application on an existing Spark Cluster...
  - OR Can run locally

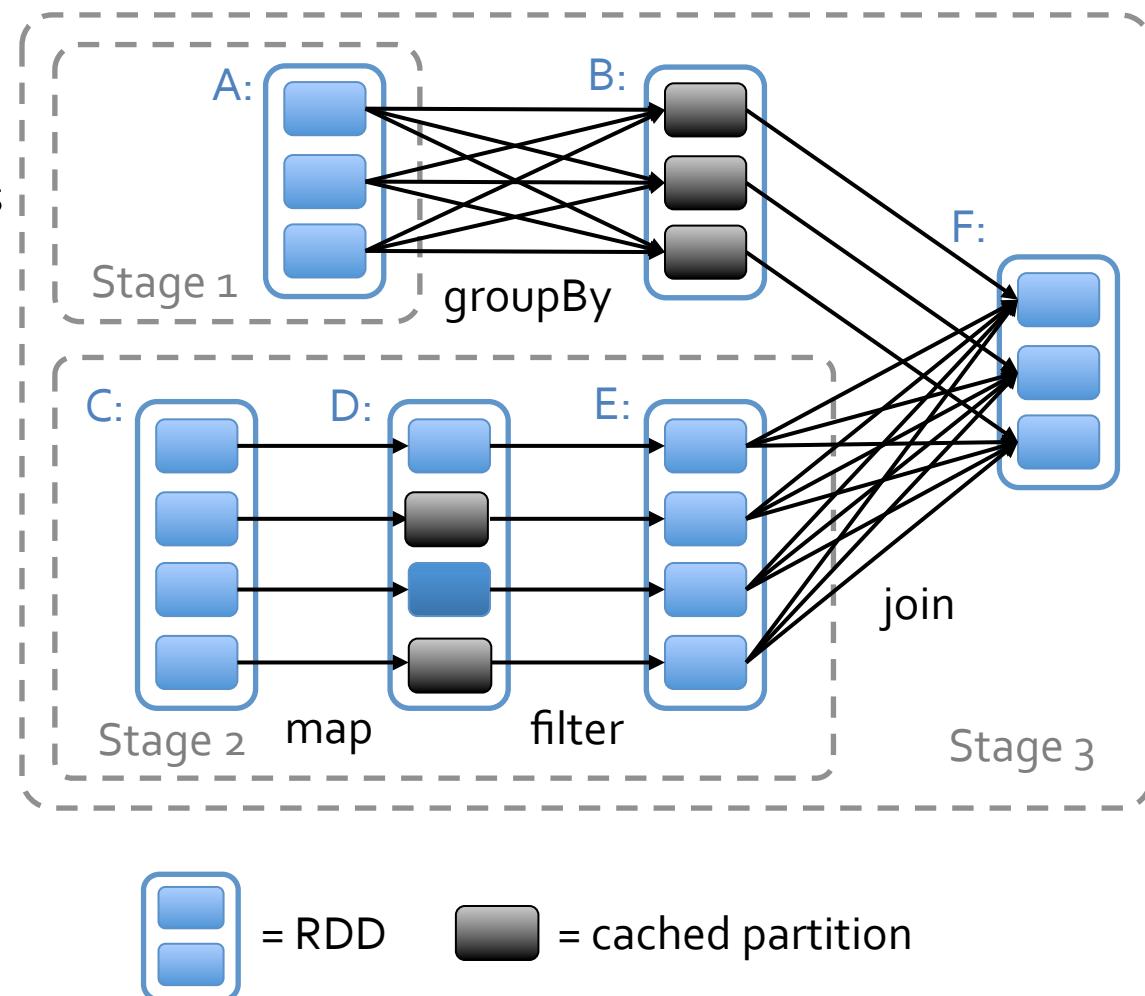
# JOB EXECUTION

- Spark runs as a library in your program (1 instance per app)
- Runs tasks locally or on cluster
  - Mesos, YARN or standalone mode
- Accesses storage systems via Hadoop InputFormat API
  - Can use HBase, HDFS, S3, ...



- General task graphs
- Automatically pipelines functions
- Data locality aware
- Partitioning aware to avoid shuffles

## NARROW/WIDE DEPENDENCIES



- Controllable partitioning
  - Speed up joins against a dataset
- Controllable storage formats
  - Keep data serialized for efficiency, replicate to multiple nodes, cache on disk
- Shared variables: broadcasts, accumulators

# SPARK INTERNALS

- Partitions:
  - set of partitions (e.g. one per block in HDFS)
- Dependencies:
  - dependencies on parent RDDs
- Iterator(/compute):
  - given a parent partitions, apply a function and return the new partition as iterator (e.g. read the input split of a HDFS block)
- PreferredLocations (optional):
  - define the preferred location for the partitions
- Partitioner (optional):
  - partition schema for the RDD



Lineage



Optimized  
Execution

- Just pass `local` or `local[k]` as master URL
- Debug using local debuggers
  - For Java / Scala, just run your program in a debugger
  - For Python, use an attachable debugger (e.g. PyDev)
- Great for development & unit tests

# WORKING WITH SPARK

## Launching:

```
spark-shell # scala  
pyspark      # python
```

## Modes:

```
MASTER=local    ./spark-shell  # local, 1 thread  
MASTER=local[2] ./spark-shell  # local, 2 threads  
MASTER=spark://host:port ./spark-shell # cluster
```

- Main entry point to Spark functionality
- Available in shell as variable `sc`
- In standalone programs, you'd make your own (see later for details)

```
# Turn a Python collection into an RDD
> sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or s3
> sc.textFile('file.txt')
> sc.textFile('directory/*.txt')
> sc.textFile('hdfs://namenode:9000/path/file')

# Use existing Hadoop InputFormat (Java/Scala only)
> sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

```
> nums = sc.parallelize([1, 2, 3])  
  
# Pass each element through a function  
> squares = nums.map(lambda x: x*x) # {1, 4, 9}  
  
# Keep elements passing a predicate  
> even = squares.filter(lambda x: x % 2 == 0) # {4}  
  
# Map each element to zero or more others  
> nums.flatMap(lambda x: range(x)) # {0, 0, 1, 0, 1, 2}  
  
# Lazy Evaluation!  
> even.collect()
```

Range object (sequence  
of numbers 0, 1, ..., x-1)

```
> nums = sc.parallelize([1, 2, 3])  
  
# Retrieve RDD contents as a local collection  
> nums.collect() # => [1, 2, 3]  
  
# Return first K elements  
> nums.take(2) # => [1, 2]  
  
# Count number of elements  
> nums.count() # => 3  
  
# Merge elements with an associative function  
> nums.reduce(lambda x, y: x + y) # => 6  
  
# Write elements to a text file  
> nums.saveAsTextFile('hdfs://file.txt')
```

# Working with Key-Value Pairs

Spark's 'distributed reduce' transformations operate on RDDs of key-value pairs:

Python: `pair = (a, b)`  
        `pair[0] # => a`  
        `pair[1] # => b`

Scala: `val pair = (a, b)`  
        `pair._1 // => a`  
        `pair._2 // => b`

Java: `Tuple2 pair = new Tuple2(a, b);`  
        `pair._1 // => a`  
        `pair._2 // => b`

## Some Key-Value Operations:

```
> pets = sc.parallelize([('cat', 1), ('dog', 1), ('cat', 2)])
> pets.reduceByKey(lambda x, y: x + y)  #{('cat', 3), ('dog', 1)}
> pets.groupByKey()      # {('cat', [1, 2]), ('dog', [1])}
> pets.sortByKey()      # {('cat', 1), ('cat', 2), ('dog', 1)}
```

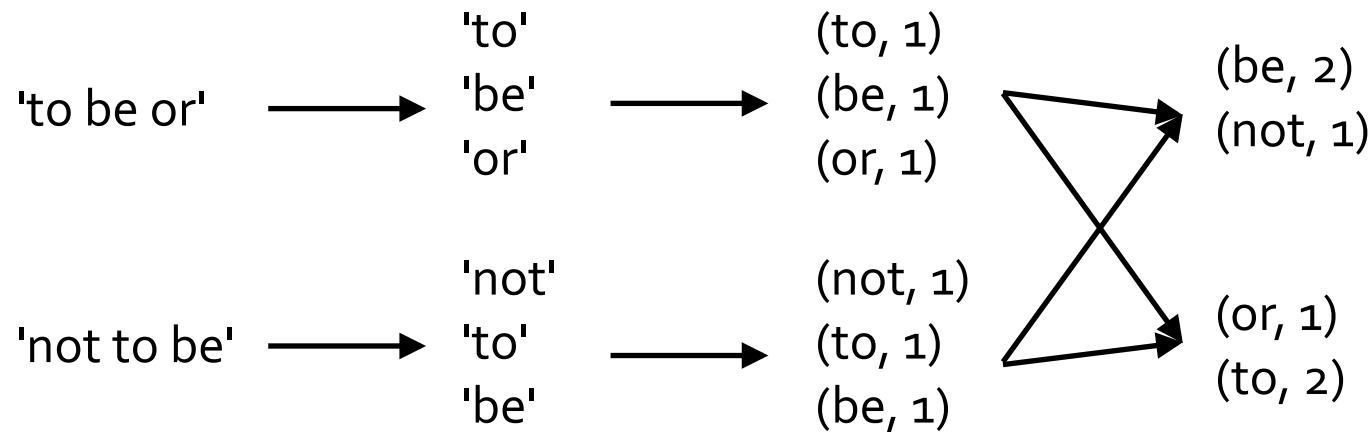
reduceByKey also automatically implements combiners on the map side

```

# create file 'hamlet.txt'
$ echo -e 'to be\nor not to be' > /usr/local/spark/hamlet.txt
$ IPYTHON=1 pyspark

lines = sc.textFile('file:///usr/local/spark/hamlet.txt')
words = lines.flatMap(lambda line: line.split(' '))
w_counts = words.map(lambda word: (word, 1))
counts = w_counts.reduceByKey(lambda x, y: x + y)
counts.collect()
# descending order:
counts.sortBy(lambda (word, count): count, ascending=False).take(3)

```



```
> visits = sc.parallelize([ ('index.html', '1.2.3.4'),
   & ('about.html', '3.4.5.6'),
   & ('index.html', '1.3.3.1') ])
```

```
> pageNames = sc.parallelize([ ('index.html', 'Home'),
   & ('about.html', 'About') ])
```

```
> visits.join(pageNames)
# ('index.html', ('1.2.3.4', 'Home'))
# ('index.html', ('1.3.3.1', 'Home'))
# ('about.html', ('3.4.5.6', 'About'))
```

```
> visits.cogroup(pageNames)
# ('index.html', ([1.2.3.4, '1.3.3.1'], ['Home']))
# ('about.html', ([3.4.5.6], ['About']))
```

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

```
> words.reduceByKey(lambda x, y: x + y, 5)  
> words.groupByKey(5)  
> visits.join(pageNames, 5)
```

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 / Pickle-able
- Don't use fields of an outer object (ships all of it!)

- map
- filter
- groupBy
- sort
- union
- join
- leftOuterJoin
- rightOuterJoin
- reduce
- count
- fold
- reduceByKey
- groupByKey
- cogroup
- cross
- zip
- sample
- take
- first
- partitionBy
- mapwith
- pipe
- save
- ...

# CREATING SPARK APPLICATIONS

- Scala / Java: add a Maven dependency on

```
groupId:          org.spark-project  
artifactId:spark-core_2.9.3  
version:         0.8.0
```

- Python: run program with pyspark script

```
import org.apache.spark.SparkContext  
import org.apache.spark.SparkContext._  
  
val sc = new SparkContext('url', 'name', 'sparkHome', Seq('app.jar'))
```

Cluster URL, or  
local / local[N]

App  
name

Spark install  
path on cluster

List of JARs with  
app code (to ship)

```
import org.apache.spark.api.java.JavaSparkContext;  
  
JavaSparkContext sc = new JavaSparkContext(  
    'masterUrl', 'name', 'sparkHome', new String[] {'app.jar'}));
```

```
from pyspark import SparkContext  
  
sc = SparkContext('masterUrl', 'name', 'sparkHome', ['library.py']))
```

```
import sys
from pyspark import SparkContext

if __name__ == '__main__':
    sc = SparkContext('local', 'wordCount', sys.argv[0], None)
    lines = sc.textFile(sys.argv[1])

    counts = lines.flatMap(lambda s: s.split(' '))
                  .map(lambda word: (word, 1))
                  .reduceByKey(lambda x, y: x + y)

    counts.saveAsTextFile(sys.argv[2])
```

# CONCLUSION

- Spark offers a rich API to make data analytics *fast*: both fast to write and fast to run
- Achieves 100x speedups in real applications
- Growing community with 25+ companies contributing



Hive on Spark, and more...

# SPARK SQL

- Tables: unit of data with the same schema
- Partitions: e.g. range-partition tables by date
- Data Types:
  - Primitive types
    - TINYINT, SMALLINT, INT, BIGINT
    - BOOLEAN
    - FLOAT, DOUBLE
    - STRING
    - TIMESTAMP
  - Complex types
    - Structs: STRUCT {a INT; b INT}
    - Arrays: ['a', 'b', 'c']
    - Maps (key-value pairs): M['key']

- Subset of SQL
  - Projection, selection
  - Group-by and aggregations
  - Sort by and order by
  - Joins
  - Sub-queries, unions
  - Supports custom map/reduce scripts (TRANSFORM)

```
CREATE EXTERNAL TABLE wiki
(id BIGINT, title STRING, last_modified STRING, xml
STRING, text STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LOCATION 's3n://spark-data/wikipedia-sample/';

SELECT COUNT(*) FROM wiki WHERE TEXT LIKE '%Berkeley%';
```

- Creates a table cached in a cluster's memory using RDD.cache ()
- '\_cached' suffix is reserved from Spark, and guarantees caching of the table

```
CREATE TABLE mytable_cached AS SELECT *  
FROM mytable WHERE count > 10;
```

- Unified table naming:

```
CACHE mytable;  UNCACHE mytable;
```

From Scala:

```
val points = sc.runSql[Double, Double](
  'select latitude, longitude from historic_tweets')

val model = KMeans.train(points, 10)

sc.twitterStream(...)
  .map(t => (model.closestCenter(t.location), 1))
  .reduceByWindow('5s', _ + _)
```

From Spark SQL:

```
GENERATE KMeans(tweet_locations) AS TABLE tweet_clusters
// Scala table generating function (TGF):
object KMeans {
  @Schema(spec = 'x double, y double, cluster int')
  def apply(points: RDD[(Double, Double)]) = {
    ...
  }
}
```

- Shark relies on Spark to infer the number of map task
  - automatically based on input size
- Number of 'reduce' tasks needs to be specified
- Out of memory error on slaves if too small
- Automated process soon (?)

- Optimized storage format
  - Columnar memory store
- Various other optimizations
  - Fully distributed sort, data co-partitioning, partition pruning, etc.
- Extremely fast scheduling
  - ms in Spark vs secs in Hadoop MR
- Support for general DAGs
  - Each query is a 'job' rather than stages of jobs
- Partial DAG Execution (PDE – extension of Spark): Spark SQL can re-optimize a running query after running the first few stages of its task DAG, choosing better join strategies or the right degree of parallelism based on observed statistics
- Many more useful primitives
  - Higher level APIs
  - Broadcast variables
  - ...

- Column-oriented storage for in-memory tables
  - when we *cache* in spark, each element of an RDD is maintained in memory as java object
  - with column-store (spark sql) each column is serialized as a single byte array (single java object)
- Yahoo! contributed CPU-efficient compression
  - e.g. dictionary encoding, run-length encoding
- 3 – 20X reduction in data size

**Row Storage**

1	john	4.1
2	mike	3.5
3	sally	6.4

**Column Storage**

1	2	3
john	mike	sally
4.1	3.5	6.4

```
# Import SQLContext and data types
> from pyspark.sql import *

# sc is an existing SparkContext
> sqlContext = SQLContext(sc)

# Load a text file and convert each line in a tuple. 'file:///' for
local files
> fname = 'file:///usr/local/spark/examples/src/main/resources/people.txt'

> lines = sc.textFile(fname)

# Count number of elements
> parts = lines.map(lambda l: l.split(','))
> people = parts.map(lambda p: (p[0], p[1].strip()))

# The schema is encoded in a string
> schemaString = 'name age'

# Write elements to a text file
> fields = [StructField(field_name, StringType(), True) for
field_name in schemaString.split()]
```

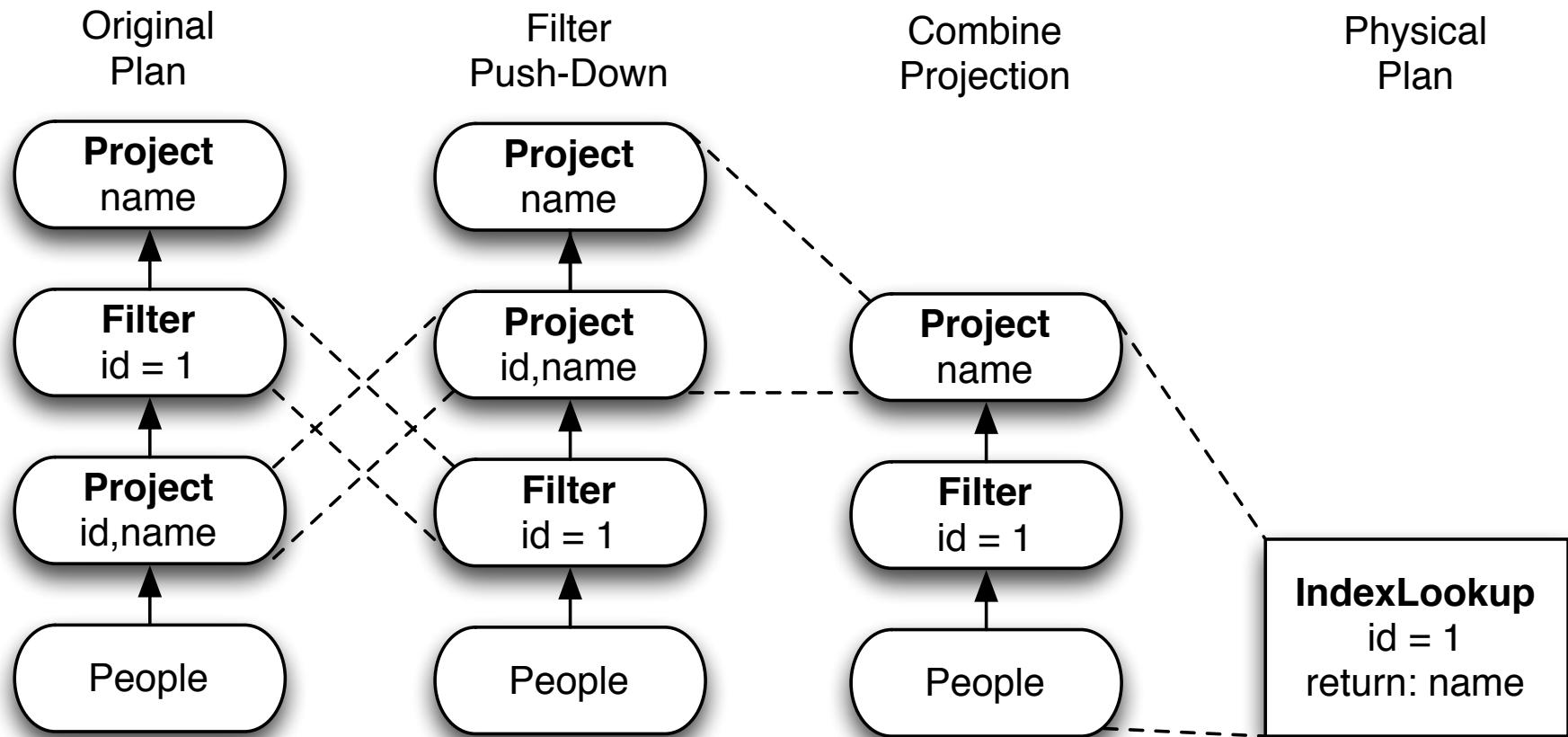
```
> schema = StructType(fields)
# Apply the schema to the RDD
> schemaPeople = sqlContext.applySchema(people, schema)

# Register the SchemaRDD as a table
> schemaPeople.registerTempTable('people')

# SQL can be run over SchemaRDDs that have been registered as a table
> results = sqlContext.sql('SELECT name FROM people')

# The results of SQL queries are RDDs and support all the normal RDD
operations
> results = sqlContext.sql('SELECT name FROM people') # return a RDD
> names = results.map(lambda p: 'Name: ' + p.name)

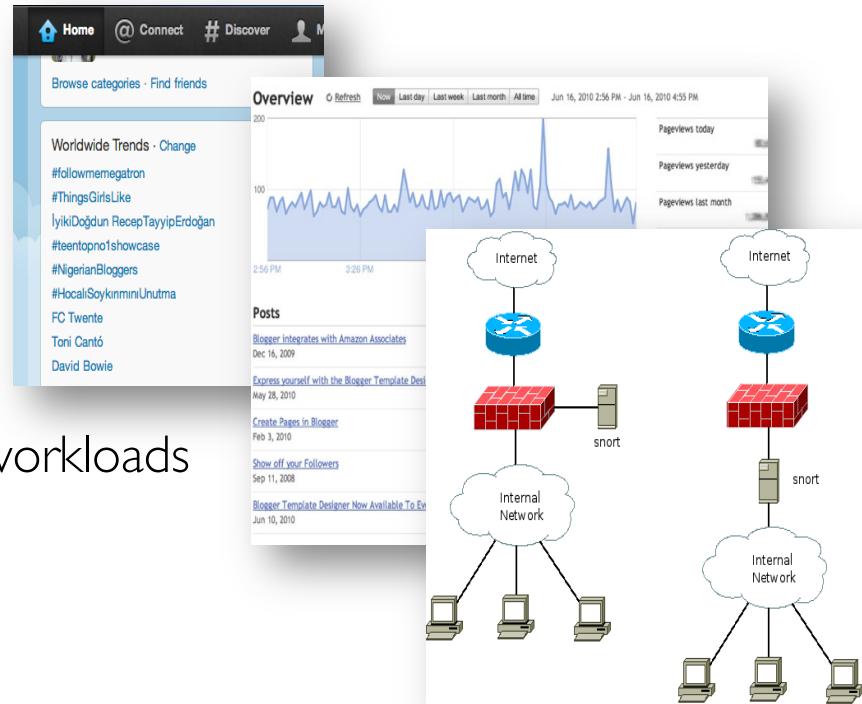
> for name in names.collect():
    print name
```



# SPARK STREAMING

- Framework for large scale stream processing
  - Scales to 100s of nodes
  - Can achieve second scale latencies
  - Integrates with Spark's batch and interactive processing
  - Provides a simple batch-like API for implementing complex algorithm
  - Can absorb live data streams from Kafka, Flume, ZeroMQ, etc.

- Many important applications must process large streams of live data and provide results in near-real-time
  - Social network trends
  - Website statistics
  - Intrusion detection systems
  - etc.
- Require large clusters to handle workloads
- Require latencies of few seconds



... for building such complex stream processing applications  
But what are the requirements from such a framework?

- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model

If you want to process live streaming data with current tools (e.g. MapReduce and Storm), you have this problem:

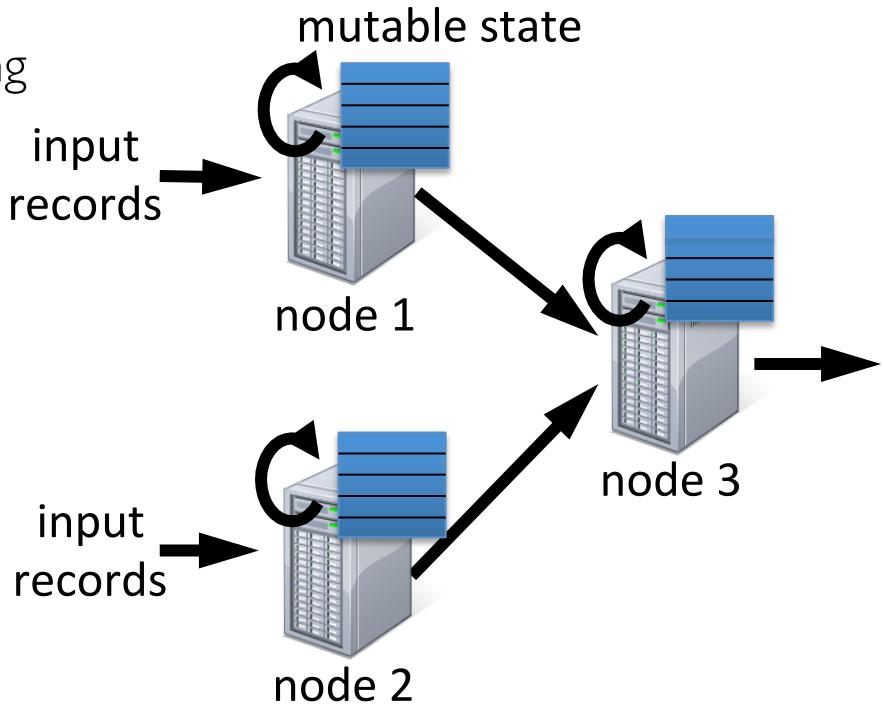
- **Twice** the effort to implement any new function
- **Twice** the number of bugs to solve
- **Twice** the headache

### New Requirement:

- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model
- **Integrated** with batch & interactive processing

- Traditional streaming systems have a event-driven **record-at-a-time** processing model
  - Each node has mutable state
  - For each record, update state & send new records
- State is lost if node dies!
- Making stateful stream processing be fault-tolerant is challenging

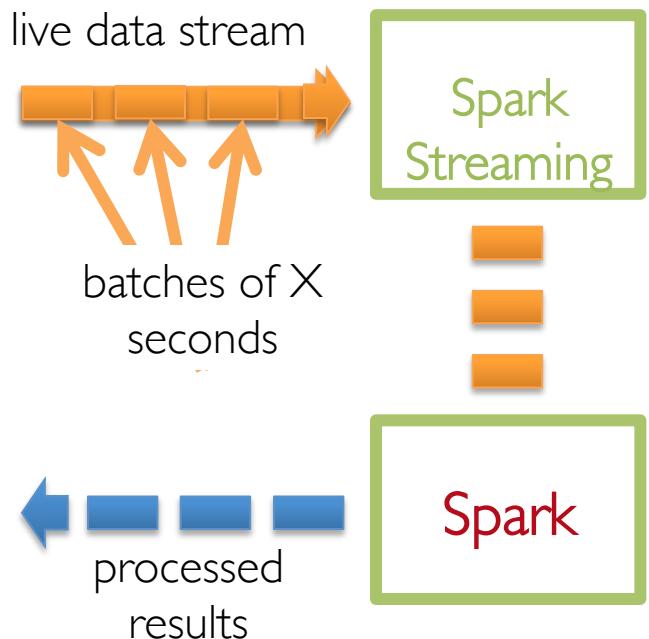
## Stateful Stream Processing



# Spark Streaming: Discretized Stream Processing (I)

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

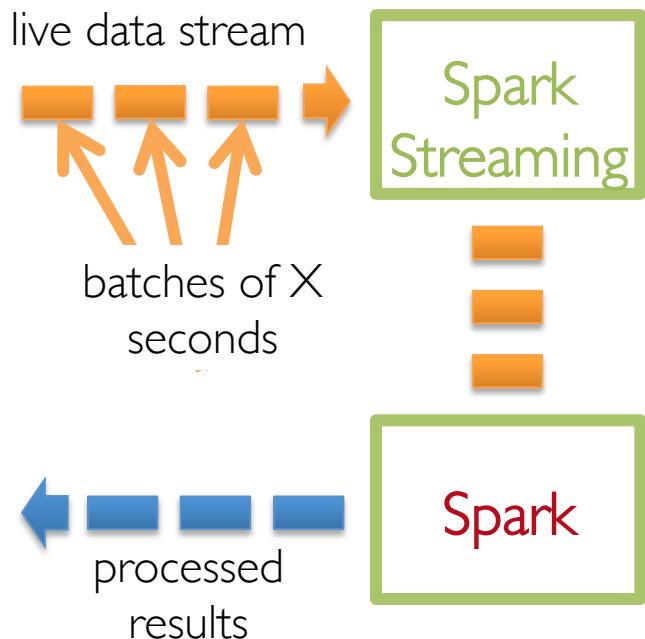
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



# Spark Streaming: Discretized Stream Processing (2)

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

- Batch sizes as low as  $\frac{1}{2}$  second, latency  $\sim 1$  second
- Potential for combining batch processing and streaming processing in the same system



# Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
```

DStream: a sequence of RDD representing a stream of data

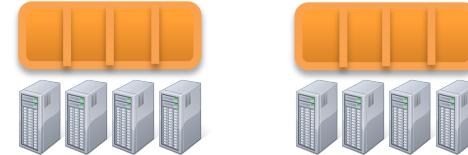
Twitter Streaming API

batch @ t      batch @ t+1

batch @ t+2



tweets DStream



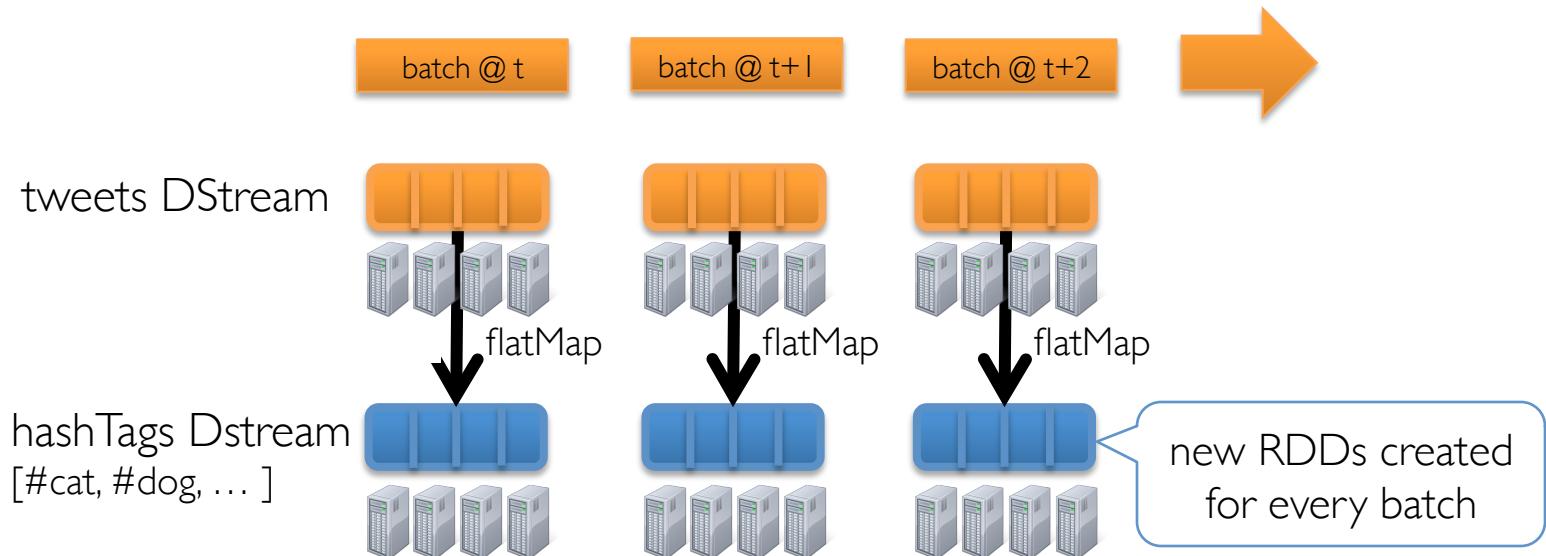
stored in memory as an RDD  
(immutable, distributed)

# Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)  
val hashTags = tweets.flatMap (status => getTags(status))
```

new DStream

transformation: modify data in one  
Dstream to create another DStream

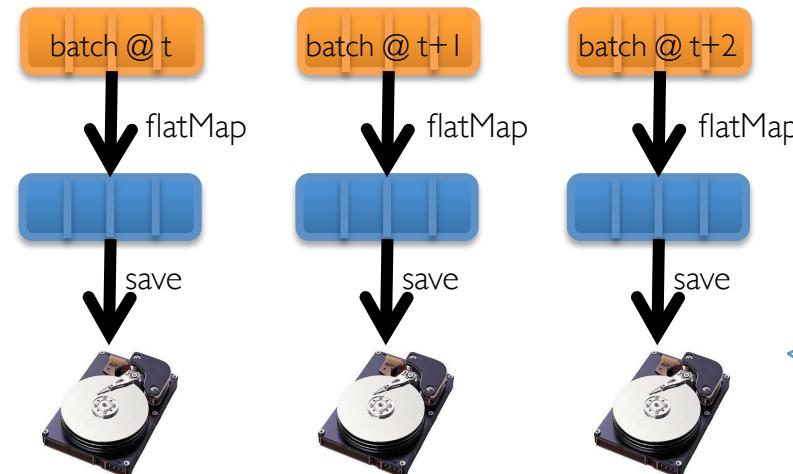


# Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

output operation: to push data to external storage

tweets DStream



hashTags DStream

every batch saved  
to HDFS

## Scala

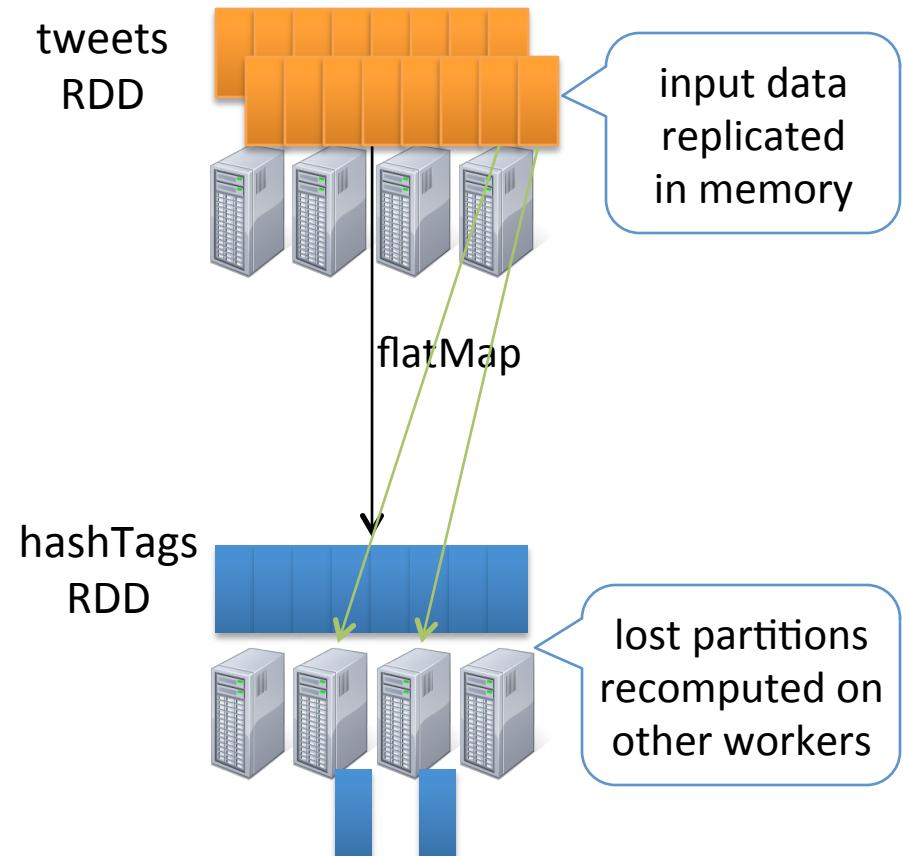
```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

## Java

```
JavaDStream<Status> tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
JavaDstream<String> hashTags = tweets.flatMap(new Function<...> { })
hashTags.saveAsHadoopFiles("hdfs://...")
```

Function object to define the transformation

- RDDs remember the sequence of operations that created it from the original fault-tolerant input data
- Batches of input data are replicated in memory of multiple worker nodes, therefore fault-tolerant
- Data lost due to worker failure, can be recomputed from input data



Count the (e.g. most 10 popular) hashtags over last 10 mins

1. Count HashTags from a stream
2. Count HashTags in a time windows from a stream

# Count the hashtags over last 10 mins (I)

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```

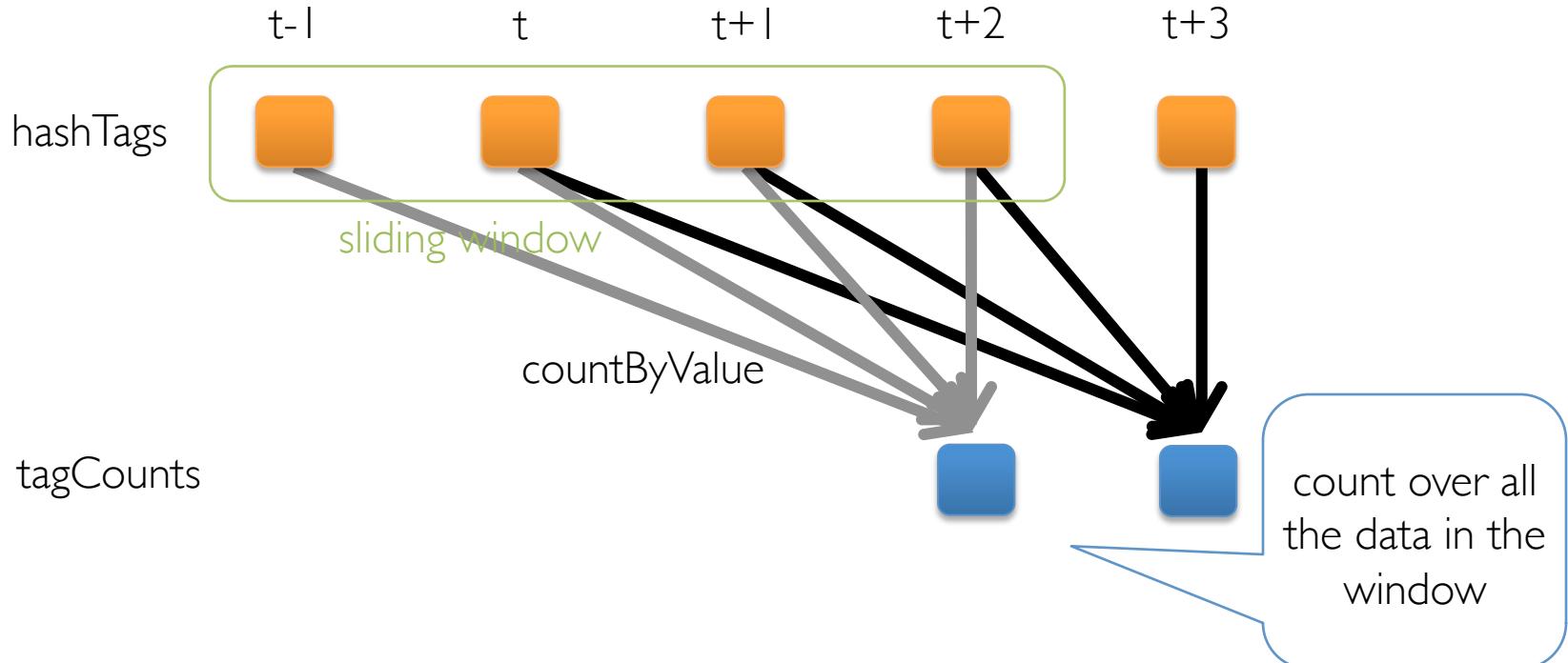
sliding window  
operation

window length

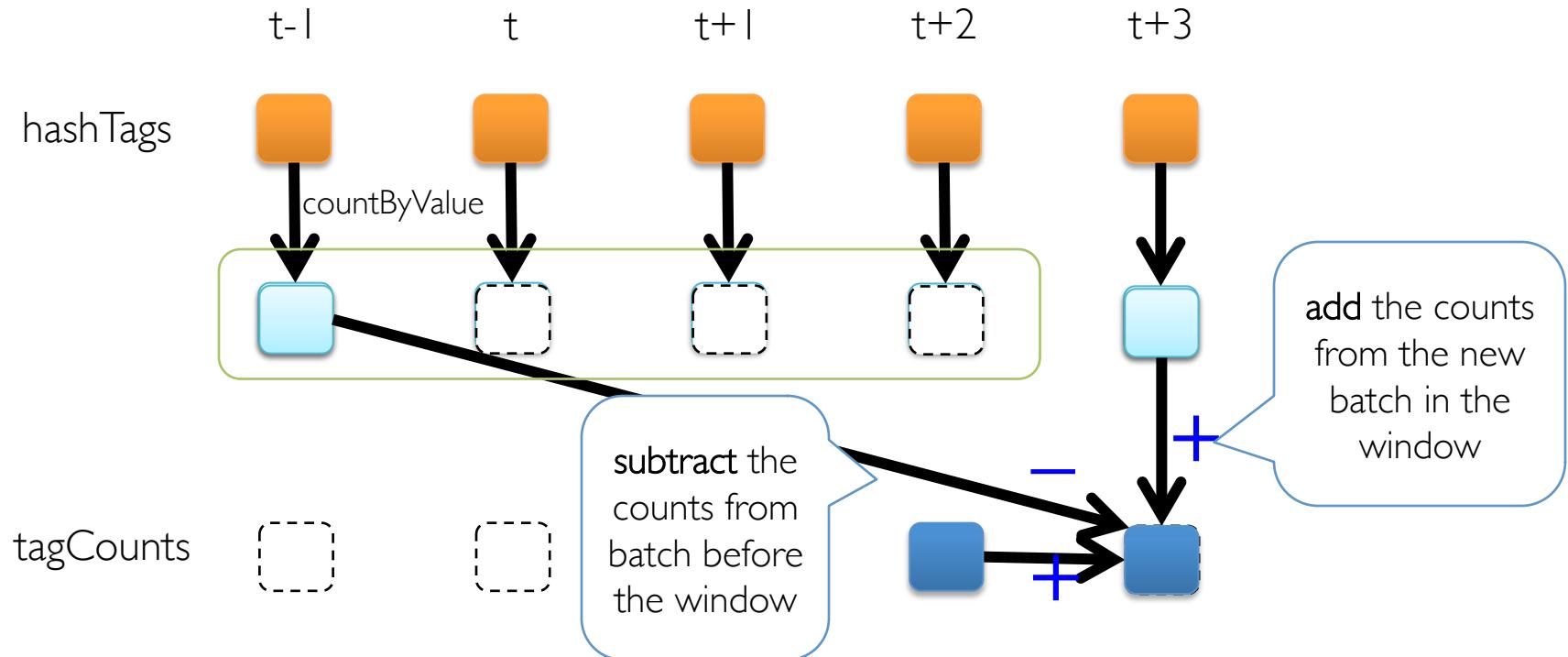
sliding interval

## Example – Count the hashtags over last 10 mins (2)

```
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



```
val tagCounts = hashtags.countByValueAndWindow(Minutes(10), Seconds(1))
```

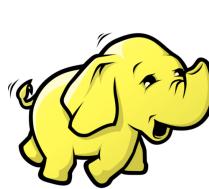


- Stream processing framework that is ...
  - Scalable to large clusters
  - Achieves second-scale latencies
  - Has simple programming model
  - Integrates with batch & interactive workloads
  - Ensures efficient fault-tolerance in stateful computations
- For more information, checkout the paper:  
[www.cs.berkeley.edu/~matei/papers/2012/hotcloud\\_spark\\_streaming.pdf](http://www.cs.berkeley.edu/~matei/papers/2012/hotcloud_spark_streaming.pdf)

GRAPHX

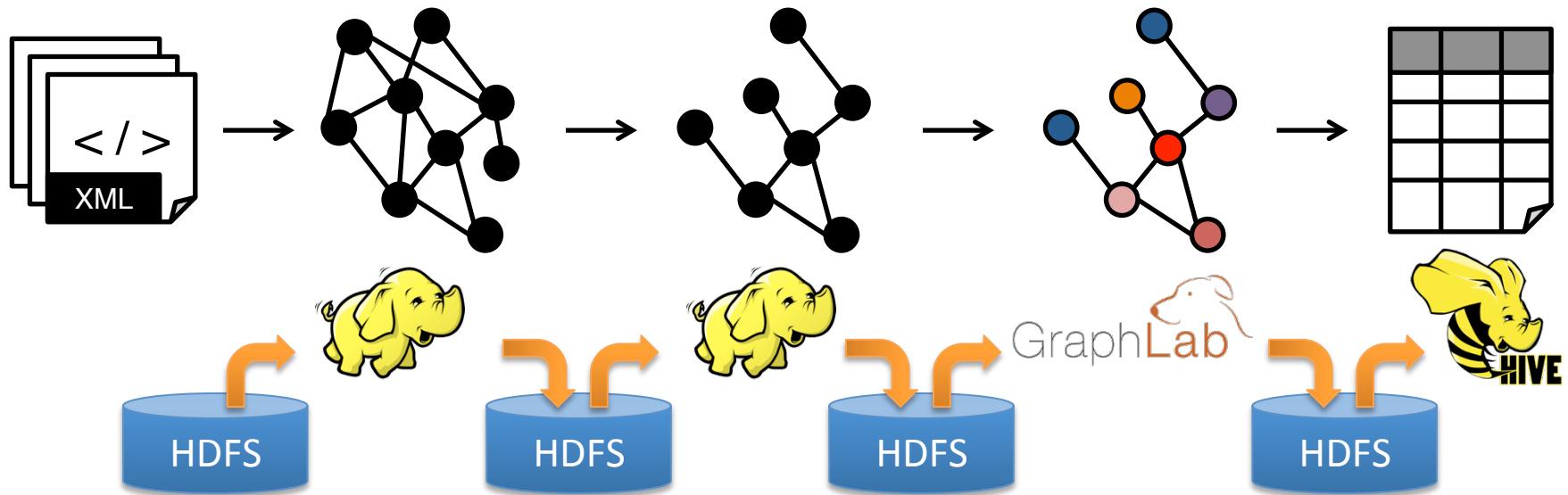
## Difficult to Program and Use

- Having separate systems for each view is:
  - difficult to use
  - inefficient
- Users must Learn, Deploy, and Manage multiple systems



Leads to brittle and often  
complex interfaces

Extensive **data movement and duplication** across the network and file system

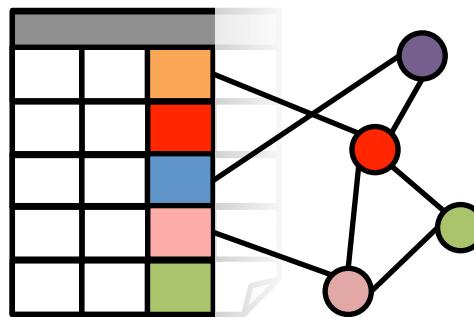


Limited reuse internal data-structures  
across stages

# Solution: The GraphX Unified Approach

## New API

Blurs the distinction between  
Tables and Graphs



## New System

Combines Data-Parallel Graph-  
Parallel Systems



Enabling users to **easily** and **efficiently** express  
the entire graph analytics pipeline

Tables and Graphs are **composable views** of the same *physical* data

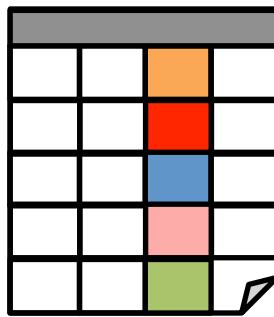
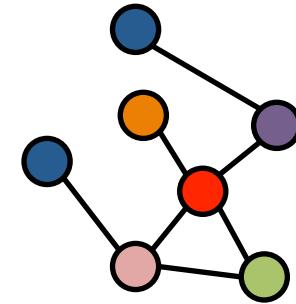
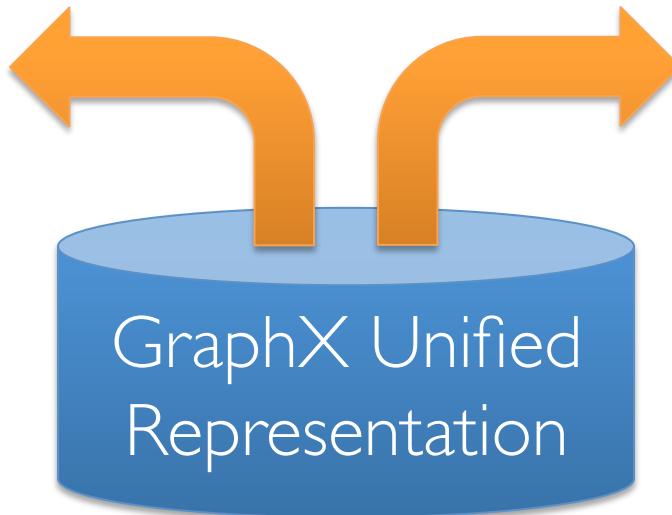


Table View



Graph View

Each view has its own **operators** that  
**exploit the semantics** of the view  
to achieve **efficient execution**

Machine Learning on Spark

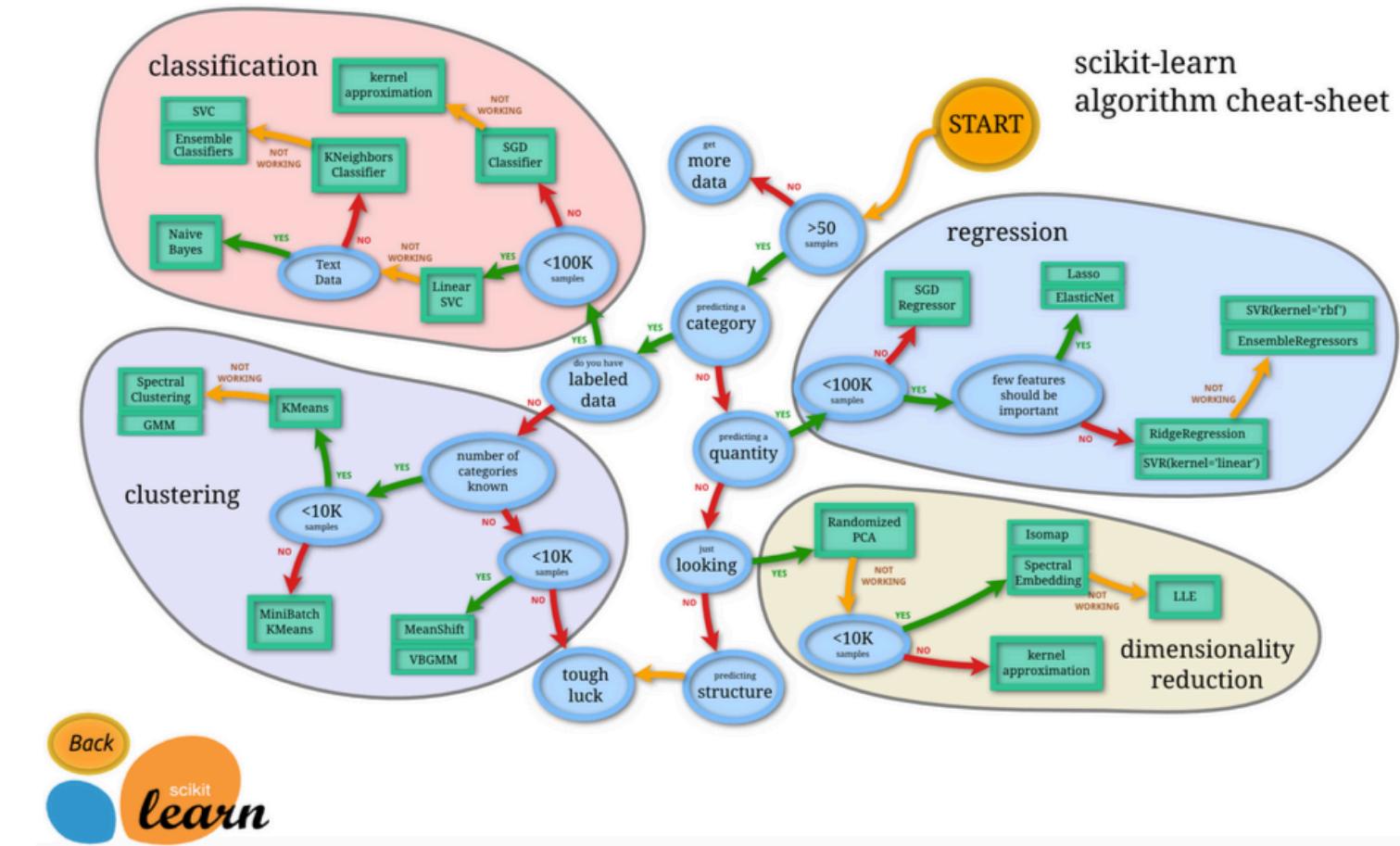
**MLLIB**

- Classification
  - Identifying to which category an object belongs to.
  - Applications: Spam detection, Image recognition.
- Regression
  - Predicting a continuous-valued attribute associated with an object.
  - Applications: Drug response, Stock prices.
- Clustering
  - Automatic grouping of similar objects into sets.
  - Applications: Customer segmentation, Grouping experiment outcomes
- Dimensionality reduction
  - Reducing the number of random variables to consider.
  - Applications: Visualization, Increased efficiency

- Model selection
  - Comparing, validating and choosing parameters and models.
  - Goal: Improved accuracy via parameter tuning
- Preprocessing
  - Feature extraction and normalization.
  - Application: Transforming input data such as text for use with machine learning algorithms.

# (How to) Learning Machine Learning

- MOOC:
  - <https://www.coursera.org/course/ml> (Stanford) - General
  - <https://www.edx.org/course/scalable-machine-learning-uc-berkeleyx-cs190-1x#.VSbqlxOUdp8> (berkley) – Spark
- Tools
  - <http://scikit-learn.org/stable/> (Python)
  - <http://www.cs.waikato.ac.nz/ml/weka/> (Java)



[http://scikit-learn.org/stable/tutorial/machine\\_learning\\_map/](http://scikit-learn.org/stable/tutorial/machine_learning_map/)

- Algorithms:
- classification: logistic regression, linear support vector machine
- (SVM), naive Bayes, classification tree
- regression: generalized linear models (GLMs), regression tree
- collaborative filtering: alternating least squares (ALS)
- clustering: k-means
- decomposition: singular value decomposition (SVD), principal component analysis (PCA)

## Algorithms

Mlib 1.1 contains the following algorithms:

- linear SVM and logistic regression
- classification and regression tree
- k-means clustering
- recommendation via alternating least squares
- singular value decomposition
- linear regression with L1- and L2-regularization
- multinomial naive Bayes
- basic statistics
- feature transformations

Usable in Java, Scala and Python

Mlib fits into Spark's APIs and interoperates with NumPy in Python

```
points = spark.textFile("hdfs://...")  
        .map(parsePoint)  
  
model = KMeans.train(points, k=10)
```

[spark.apache.org/mllib/](http://spark.apache.org/mllib/)

In theory, Mahout is a project open to implementations of all kinds of machine learning techniques

In practice, it's a project that focuses on three key areas of machine learning at the moment. These are recommender engines (collaborative filtering), clustering, and classification

## Recommendation

- For a given set of input, make a recommendation
- Rank the best out of many possibilities

## Clustering

- Finding similar groups (based on a definition of similarity)
- Algorithms do not require training
- Stopping condition: iterate until close enough

## Classification

- identifying to which of a set of (predefined) categories a new observation belongs
- Algorithms do require training

## Mahout News

### 25 April 2014 - Goodbye MapReduce

The Mahout community decided to move its codebase onto modern data processing systems that offer a richer programming model and more efficient execution than Hadoop MapReduce. **Mahout will therefore reject new MapReduce algorithm implementations from now on.** We will however keep our widely used MapReduce algorithms in the codebase and maintain them.

We are building our future implementations on top of a [DSL for linear algebraic operations](#) which has been developed over the last months. Programs written in this DSL are automatically optimized and executed in parallel on [Apache Spark](#).

Furthermore, there is an experimental contribution undergoing which aims to [integrate the h2o platform into Mahout](#).

### Scala & Spark Bindings for Mahout:

- Scala DSL and algebraic optimizer
  - The main idea is that a scientist writing algebraic expressions cannot care less of distributed operation plans and works entirely on the logical level just like he or she would do with R.
  - Another idea is decoupling logical expression from distributed back-end. As more back-ends are added, this implies "write once, run everywhere".

	Single Machine	MapReduce	Spark
<b>Collaborative Filtering with CLI Drivers</b>			
User-Based Collaborative Filtering	x		x
Item-Based Collaborative Filtering	x	x	x
Matrix Factorization with ALS	x	x	
Matrix Factorization with ALS on Implicit Feedback	x	x	
Weighted Matrix Factorization, SVD++	x		
<b>Classification with CLI Drivers</b>			
Logistic Regression - trained via SGD	x		
Naive Bayes / Complementary Naive Bayes	x		<i>in development</i>
Random Forest	x		
Hidden Markov Models	x		
Multilayer Perceptron	x		
<b>Clustering with CLI Drivers</b>			
Canopy Clustering	<i>deprecated</i>	<i>deprecated</i>	
k-Means Clustering	x	x	
Fuzzy k-Means	x	x	
Streaming k-Means	x	x	
Spectral Clustering		x	

<http://mahout.apache.org/users/basics/algorithms.html>

	Single Machine	MapReduce	Spark
<b>Dimensionality Reduction with CLI Drivers</b>			
- note: most scala-based dimensionality reduction algorithms are available through the Mahout Math-Scala Core Library for all engines			
Singular Value Decomposition	x	x	
Lanczos Algorithm	x	x	
Stochastic SVD	x	x	
PCA (via Stochastic SVD)	x	x	
QR Decomposition	x	x	
<b>Topic Models</b>			
Latent Dirichlet Allocation	x	x	
<b>Miscellaneous</b>			
RowSimilarityJob	x	x	
ConcatMatrices	x		
Collocations	x		
Sparse TF-IDF Vectors from Text	x		
XML Parsing	x		
Email Archive Parsing	x		
Lucene Integration	x		
Evolutionary Processes	x		

<http://mahout.apache.org/users/basics/algorithms.html>

	Single Machine	MapReduce	Spark
<b>Collaborative Filtering with CLI Drivers</b>			
User-Based Collaborative Filtering	x		x
Item-Based Collaborative Filtering	x	x	x
Matrix Factorization with ALS	x	x	
Matrix Factorization with ALS on Implicit Feedback	x	x	
Weighted Matrix Factorization, SVD++	x		
<b>Classification with CLI Drivers</b>			
Logistic Regression - trained via SGD	x		
Naive Bayes / Complementary Naive Bayes	x		<i>in development</i>
Random Forest	x		
Hidden Markov Models	x		
Multilayer Perceptron	x		
<b>Clustering with CLI Drivers</b>			
Canopy Clustering	<i>deprecated</i>	<i>deprecated</i>	
k-Means Clustering	x	x	
Fuzzy k-Means	x	x	
Streaming k-Means	x	x	
Spectral Clustering		x	

<http://mahout.apache.org/users/basics/algorithms.html>

# Machine Learning library for Giraph

- Collaborative Filtering
  - Alternating Least Squares (ALS)
  - Bayesian Personalized Ranking (BPR) –beta-
  - Collaborative Less-is-More Filtering (CLiMF) –beta-
  - Singular Value Decomposition (SVD++)
  - Stochastic Gradient Descent (SGD)
- Graph Analytics
  - Graph partitioning
  - Similarity
  - SybilRank
- Clustering
  - Kmeans



<http://grafos.ml/#Okapi>

# SPARK REAL CASES APPLICATIONS

# Thunder: Neural Data Analysis in Spark

**thunder**

0.4.1

Tutorials

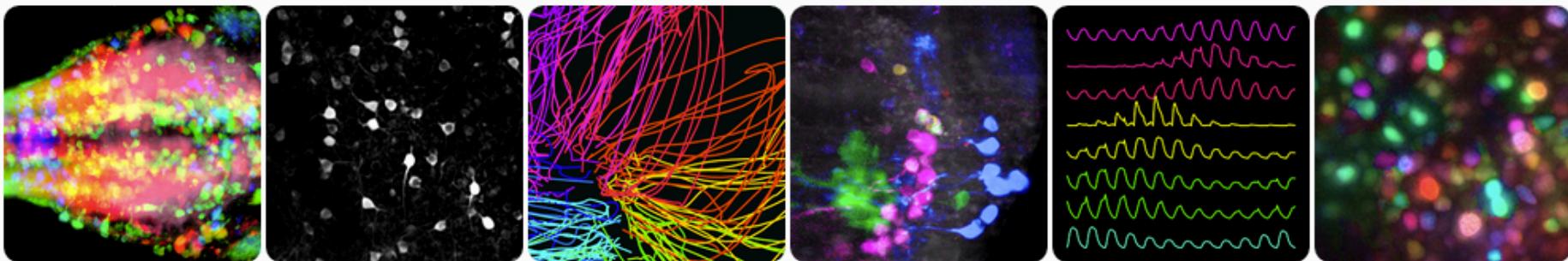
API

Site ▾

Page ▾

Search

## thunder: neural data analysis in spark



Thunder is a library for analyzing large-scale neural data. It's fast to run, easy to develop for, and can be used interactively. It is built on Spark, a new framework for cluster computing.

Thunder includes utilities for loading and saving different formats, classes for working with distributed spatial and temporal data, and modular functions for time series analysis, factorization, and model fitting. Analyses can easily be scripted or combined. It is written in Spark's Python API (Pyspark), making use of scipy, numpy, and scikit-learn.

Project Homepage: [thefreemanlab.com/thunder/docs/](http://thefreemanlab.com/thunder/docs/)

Youtube: [www.youtube.com/watch?v=Gg\\_5fWllfgA&list=UURzsq7k4-kT-h3TDUBQ82-w](https://www.youtube.com/watch?v=Gg_5fWllfgA&list=UURzsq7k4-kT-h3TDUBQ82-w)

 MAR 4TH, 2014

# Projects

Thanks to advances in both the cost and speed of sequencing technology, the amount of genomic data available for processing is growing exponentially. As a project, our goal is to build scalable pipelines for processing genomic data on top of high performance distributed computing frameworks.

## Projects

### Variant Call Format

From Wikipedia, the free encyclopedia

At the moment, we a

The Variant Call Format (VCF) specifies the format of a text file used in bioinformatics for storing gene sequence variations.

- [ADAM](#): A scalable API & CLI for genome processing
- [bdg-formats](#): Schemas for genomic data
- [avocado](#): A Variant Caller, Distributed

The source for these projects is available at [Github](#).



Project Homepage: [Homepage: http://bdgenomics.org/projects/](http://bdgenomics.org/projects/)

Youtube: [www.youtube.com/watch?v=RwyEEMw-NR8&list=UURzsq7k4-kT-h3TDUBQ82-w](https://www.youtube.com/watch?v=RwyEEMw-NR8&list=UURzsq7k4-kT-h3TDUBQ82-w)

Spark

# ADDENDUM

# Administrative GUIs

[`http://<Standalone Master>:8080 \(by default\)`](http://<Standalone Master>:8080 (by default))

**Spark Master at spark**

URL: `spark://mbp-2.local:7077`  
Workers: 3  
Cores: 24 Total, 24 Used  
Memory: 45.0 GB Total, 1536.0 MB Used  
Applications: Running, 0 Completed

**Workers**

ID
<code>worker-20131202231645-192.168.1.106-56789</code>
<code>worker-20131202231657-192.168.1.106-56801</code>
<code>worker-20131202231705-192.168.1.106-56806</code>

**Running Applications**

ID	Name
<code>app-20131202231712-0000</code>	Spark shell

**Spark shell - Spark Stages**

Total Duration: 3.8 m  
Scheduling Mode: FIFO  
Active Stages: 0  
Completed Stages: 2  
Failed Stages: 0

**Active Stages (0)**

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read
----------	-------------	-----------	----------	------------------------	--------------

**Completed Stages (2)**

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle
0	<code>count at &lt;console&gt;:13</code>	2013/12/02 21:07:55	83 ms	2/2	754.0 B
1	<code>reduceByKey at &lt;console&gt;:13</code>	2013/12/02 21:07:55	345 ms	2/2	

**Failed Stages (0)**

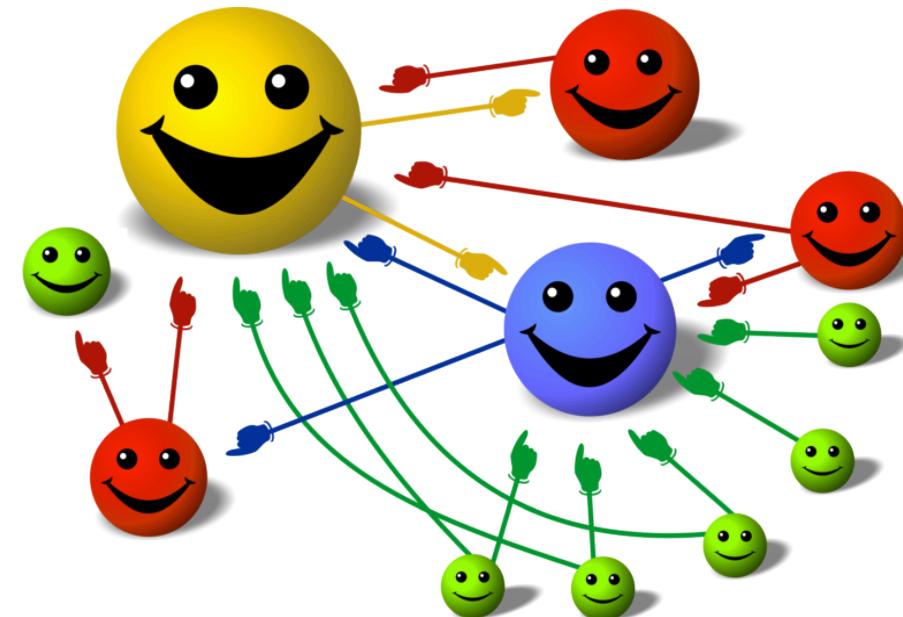
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read
----------	-------------	-----------	----------	------------------------	--------------

# EXAMPLE APPLICATION: PAGERANK

- Good example of a more complex algorithm
  - Multiple stages of map & reduce
- Benefits from Spark's in-memory caching
  - Multiple iterations over the same data

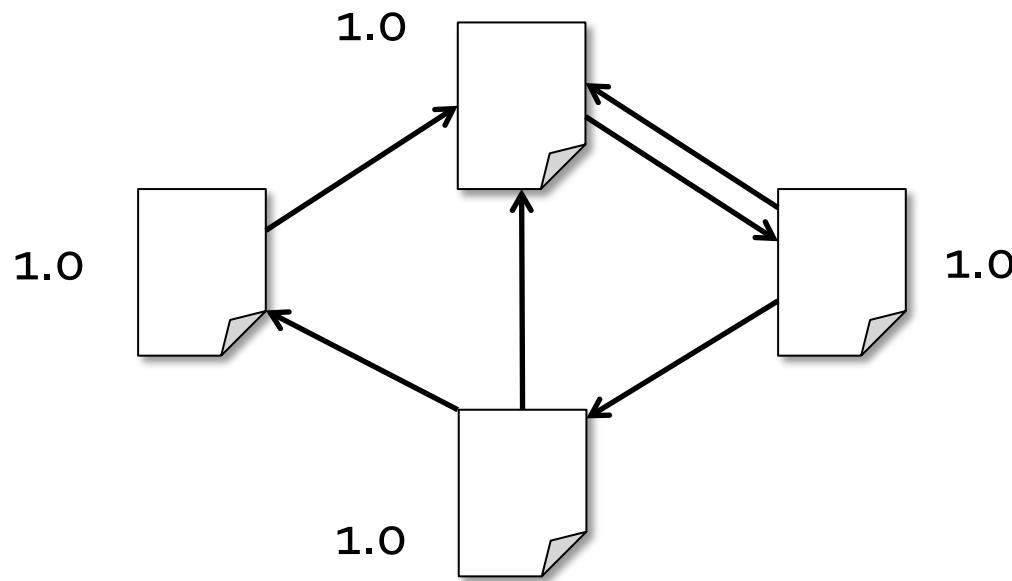
Give pages ranks (scores) based on links to them

- Links from many pages → high rank
- Link from a high-rank page → high rank



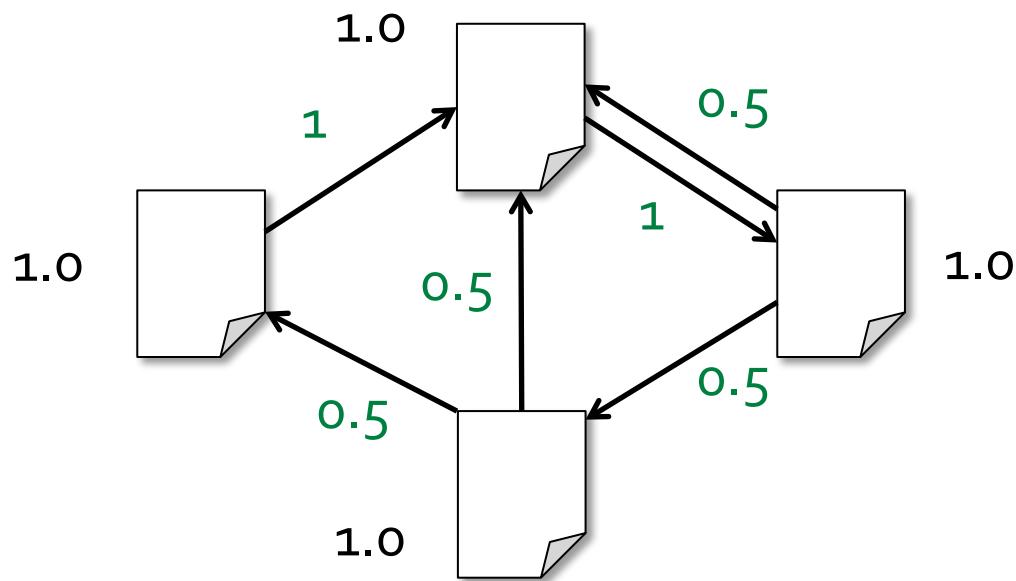
## Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page  $p$  contribute  $\text{rank}_p / |\text{neighbors}_p|$  to its neighbors
3. Set each page's rank to  $0.15 + 0.85 \times \text{contribs}$



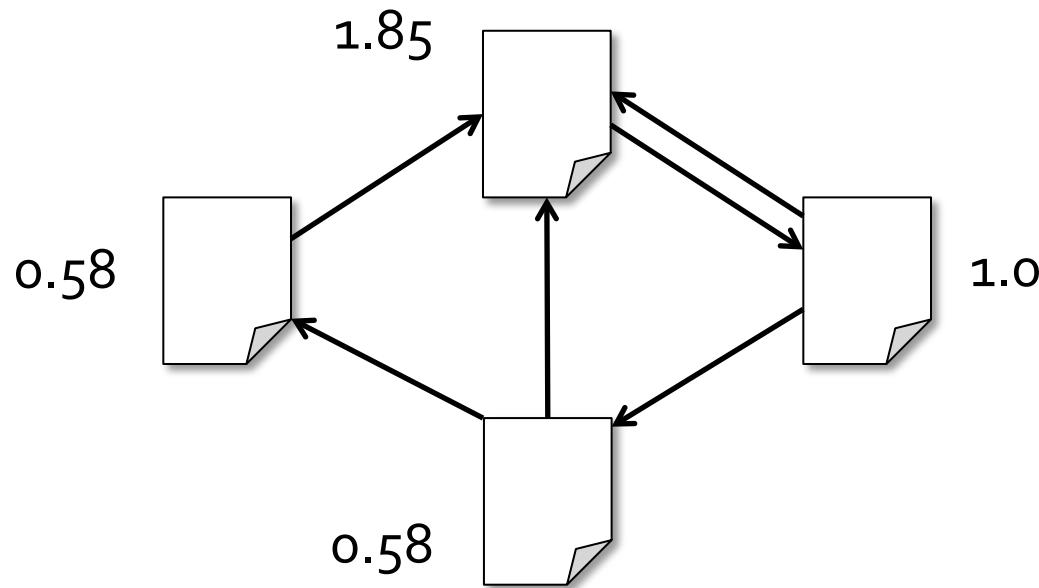
## Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page  $p$  contribute  $\text{rank}_p / |\text{neighbors}_p|$  to its neighbors
3. Set each page's rank to  $0.15 + 0.85 \times \text{contribs}$



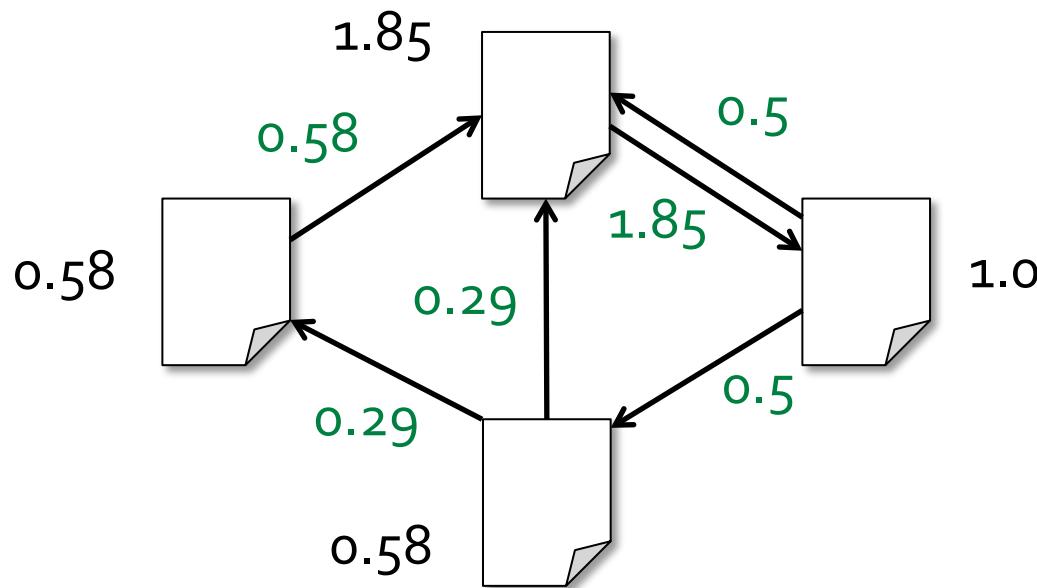
## Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page  $p$  contribute  $\text{rank}_p / |\text{neighbors}_p|$  to its neighbors
3. Set each page's rank to  $0.15 + 0.85 \times \text{contribs}$



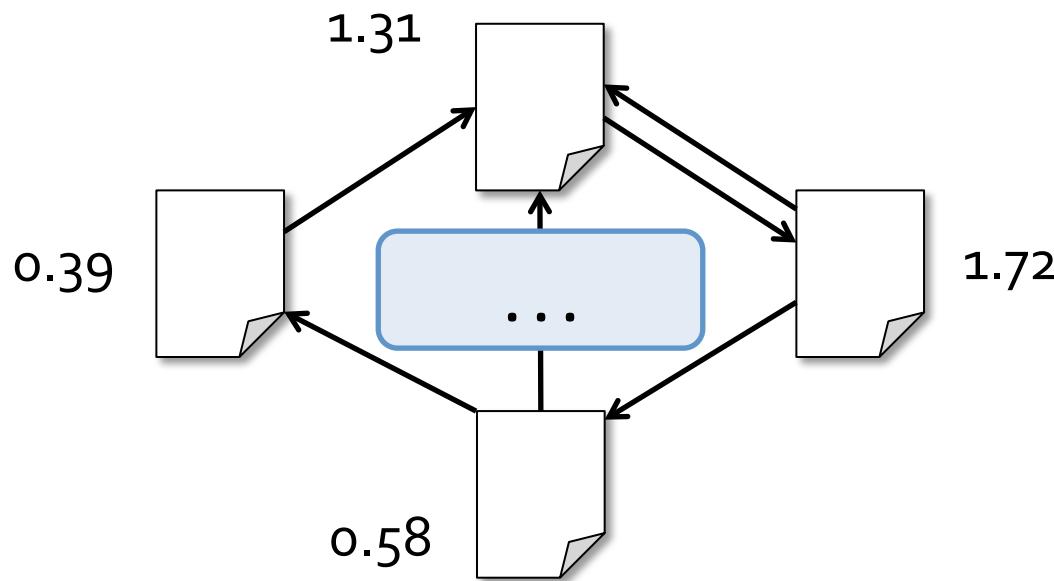
## Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page  $p$  contribute  $\text{rank}_p / |\text{neighbors}_p|$  to its neighbors
3. Set each page's rank to  $0.15 + 0.85 \times \text{contribs}$



## Algorithm

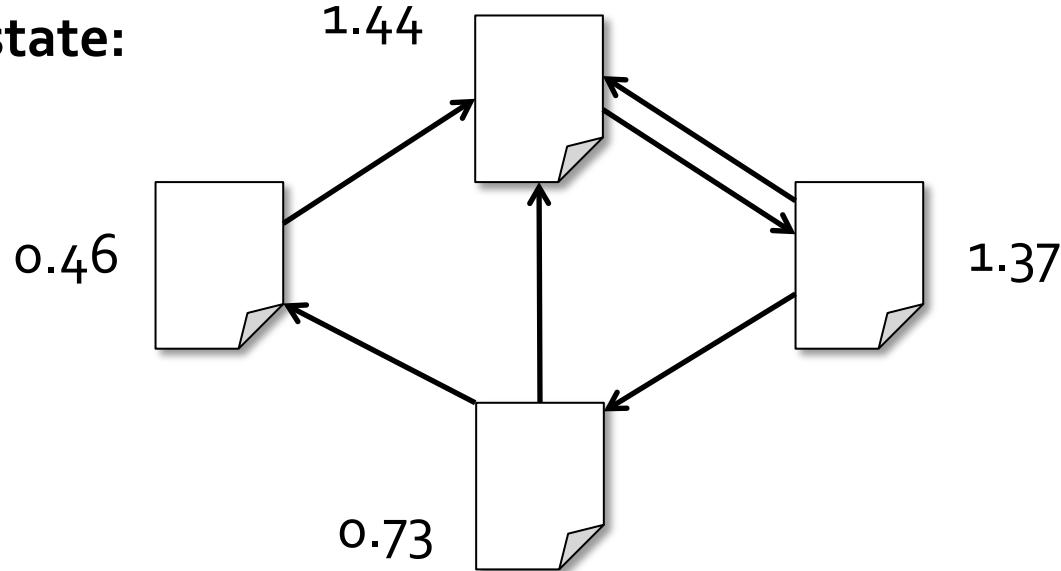
1. Start each page at a rank of 1
2. On each iteration, have page  $p$  contribute  $\text{rank}_p / |\text{neighbors}_p|$  to its neighbors
3. Set each page's rank to  $0.15 + 0.85 \times \text{contribs}$



## Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page  $p$  contribute  $\text{rank}_p / |\text{neighbors}_p|$  to its neighbors
3. Set each page's rank to  $0.15 + 0.85 \times \text{contribs}$

**Final state:**



```
val links = // load RDD of (url, neighbors) pairs
var ranks = // load RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
    val contribs = links.join(ranks).flatMap {
        case (url, (links, rank)) =>
            links.map(dest => (dest, rank/links.size))
    }
    ranks = contribs.reduceByKey(_ + _)
        .mapValues(0.15 + 0.85 * _)
}
ranks.saveAsTextFile(...)
```

- Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.
- Xin, Reynold S., et al. "Shark: SQL and rich analytics at scale." Proceedings of the 2013 international conference on Management of data. ACM, 2013.
- <https://spark.apache.org/>
- <http://spark-summit.org/2014/training>
- <http://ampcamp.berkeley.edu/>