consorzio interuniversitario nazionale per l'informatica

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 issues

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

- without having to worry about:
  - distribution
  - fault tolerance
- 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

The Spark Solution

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

How?

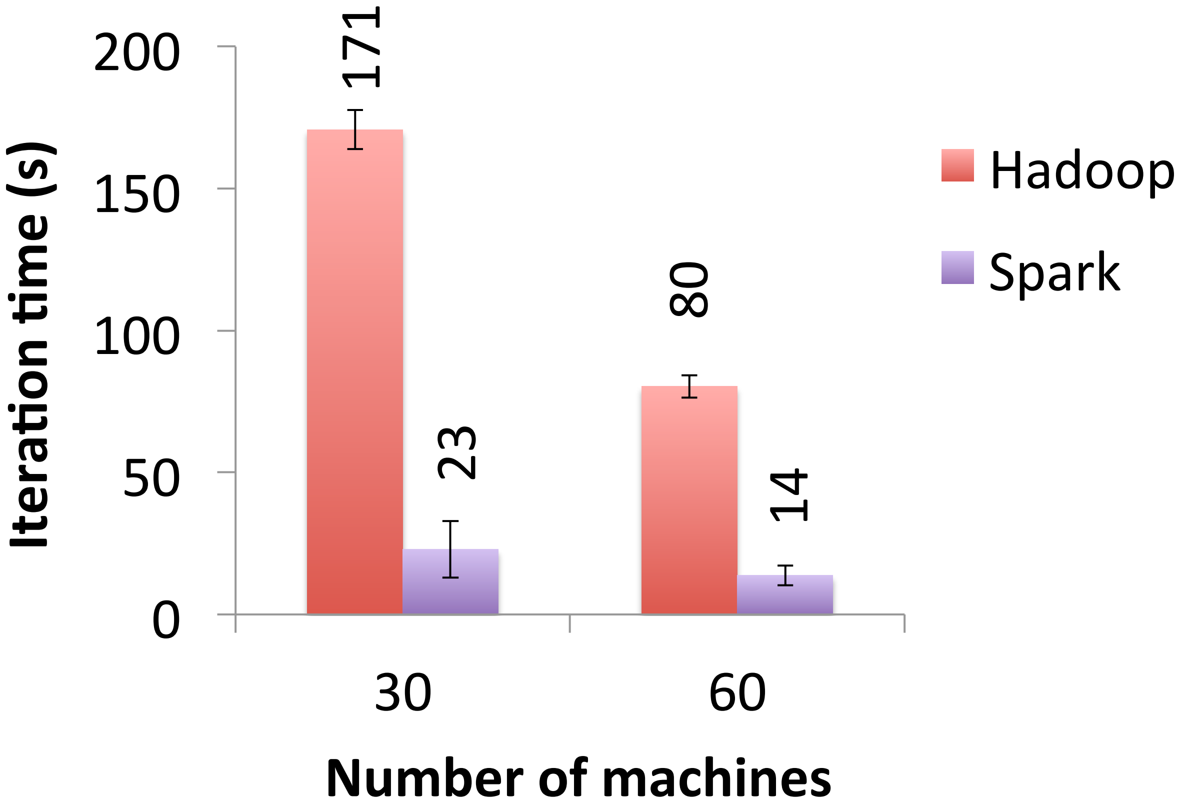
- keeping data in memory can improve performance by an order of magnitude
  - Resilient Distributed Datasets (RDDs)
- up to 20×/40x faster than Hadoop for iterative applications

RDDs

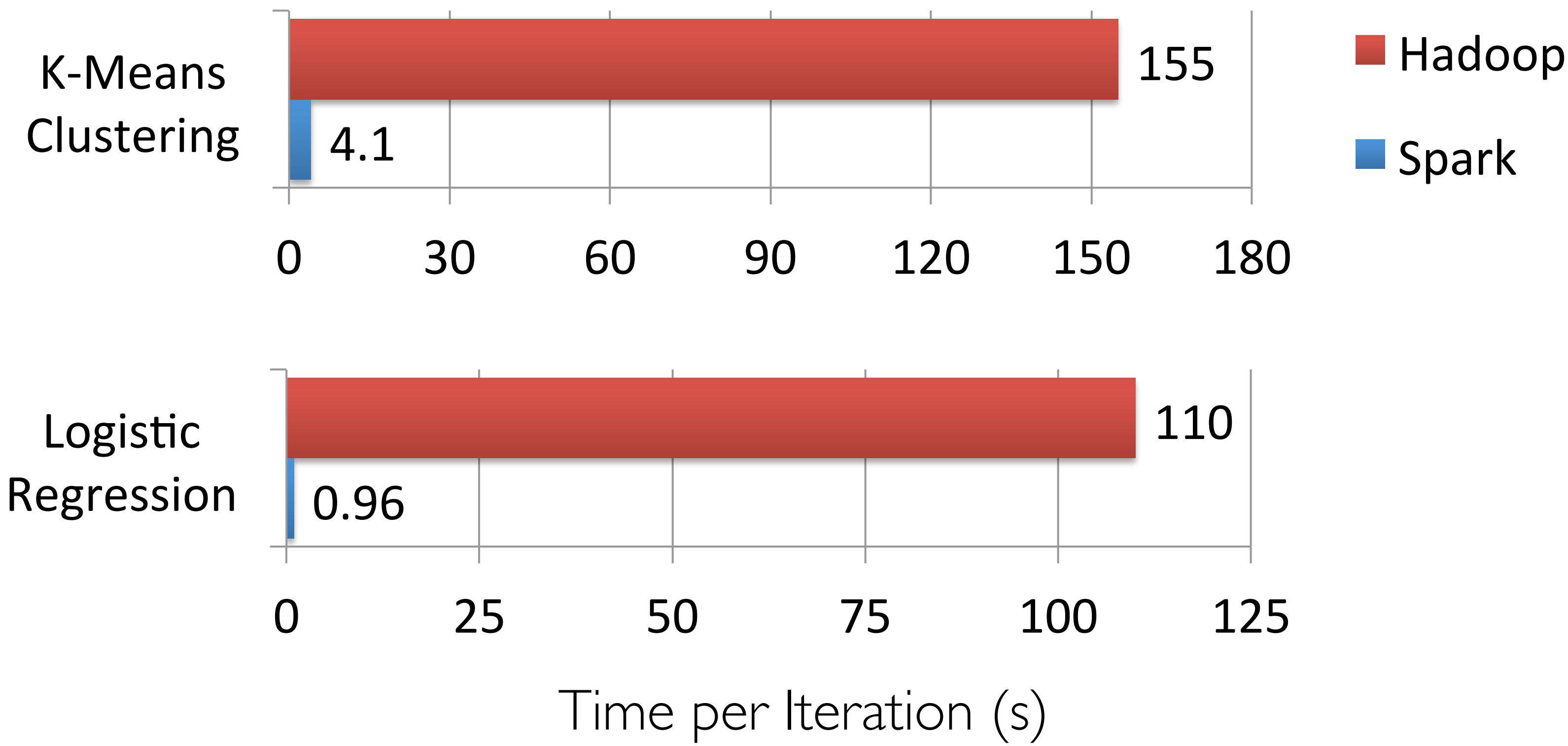
RDDs provide a restricted form of shared memory:

- based on coarse-grained transformations rather than fine-grained updates to shared state
- 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

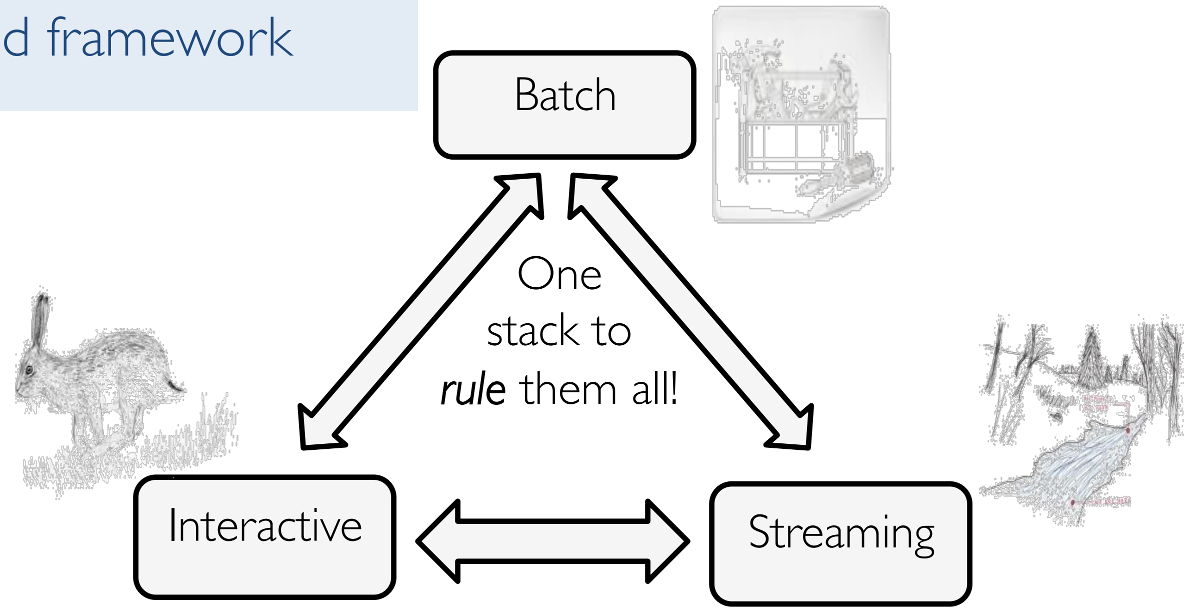
PageRank Performance



Other Iterative Algorithms

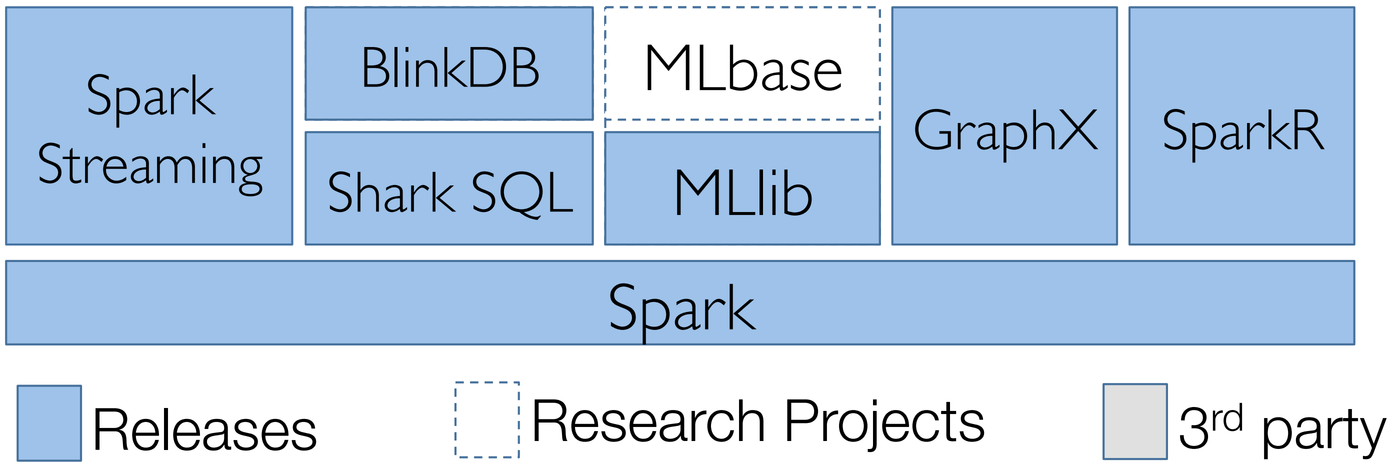


Support batch, streaming, and interactive computations in a unified framework



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

BDAS Stack (Feb, 2014)



RDDs

RDDs are fault-tolerant, parallel data structures:

- let users 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

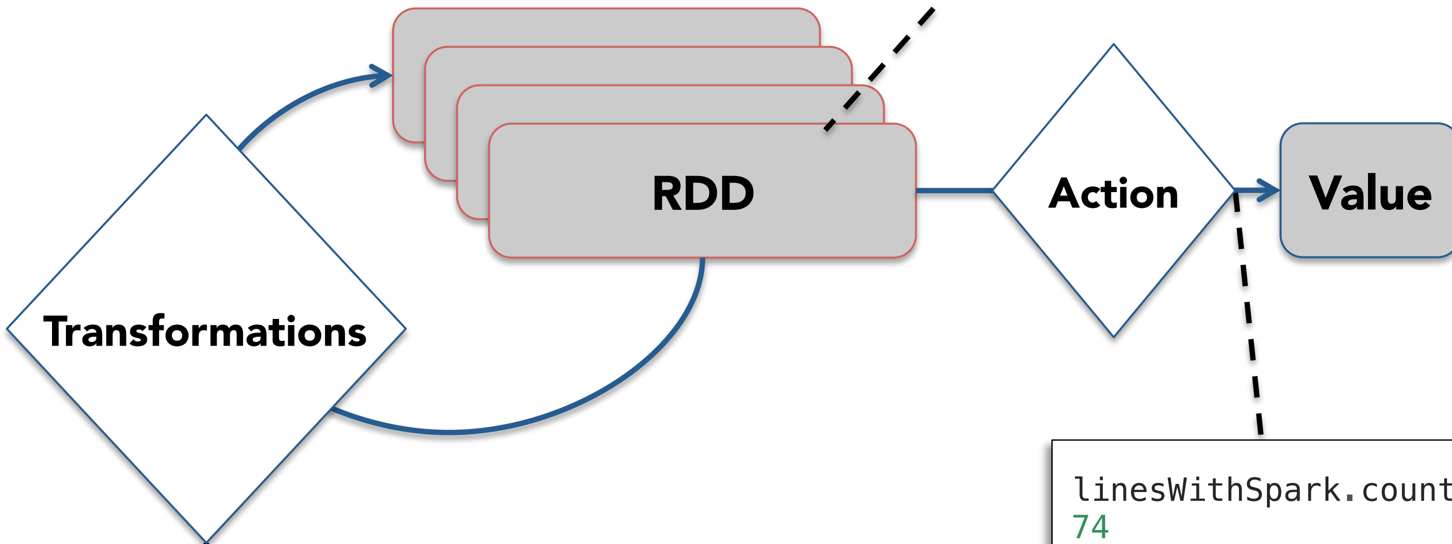
Resilient Distributed Datasets

- 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)

Working With RDDs




```
textFile = sc.textFile('SomeFile.txt')
```

```
linesWithSpark = textFile.filter(lambda line: 'Spark' in line)
```

```
linesWithSpark.count()
74

linesWithSpark.first()
# Apache Spark
```

Example: Log Mining

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

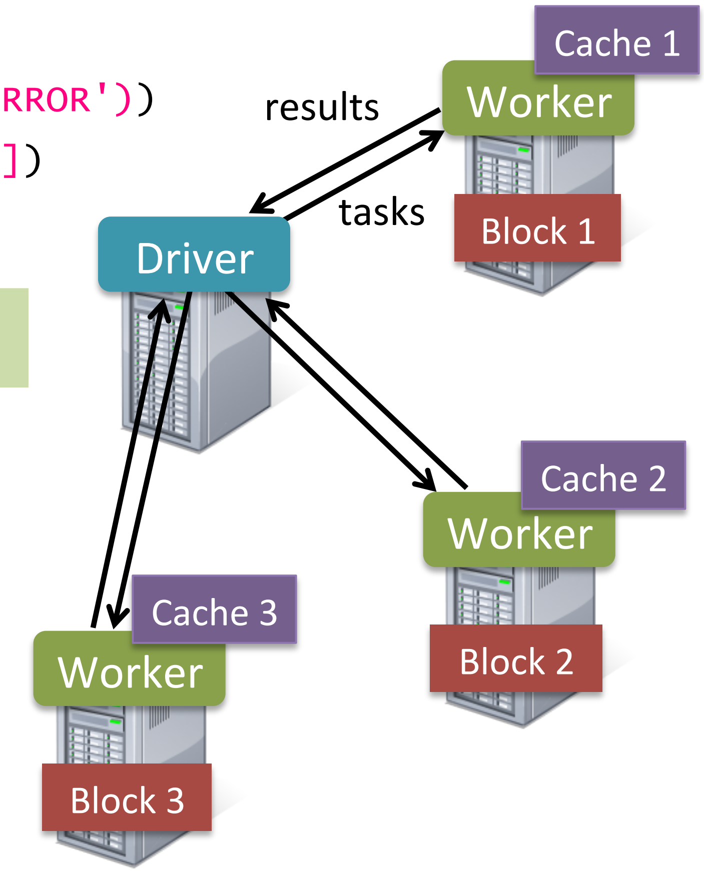
Base RDD

```
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 Evaluaziont)

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

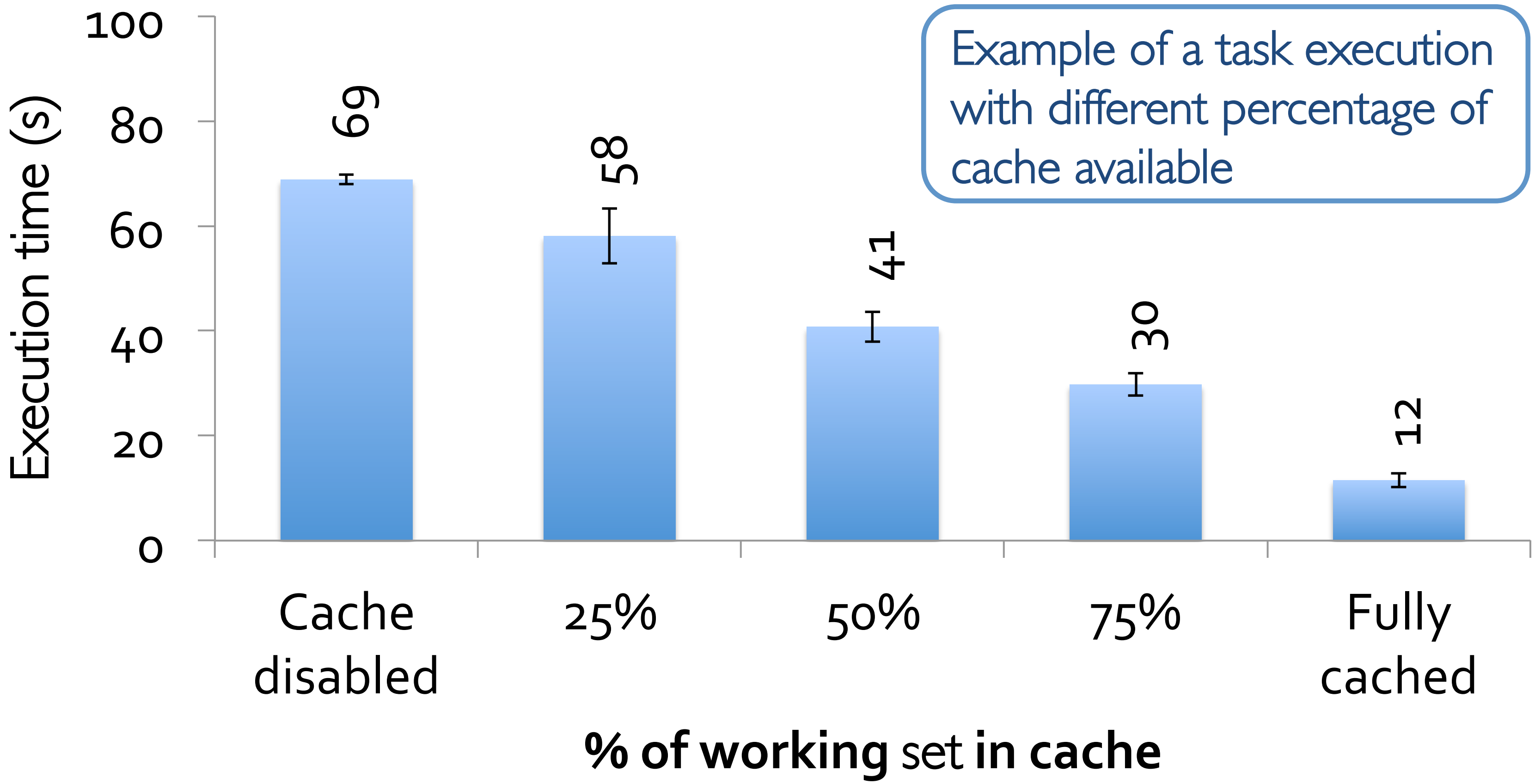


Note:

Scaling Down

Degrade Gracefully, if you don't have enough memory

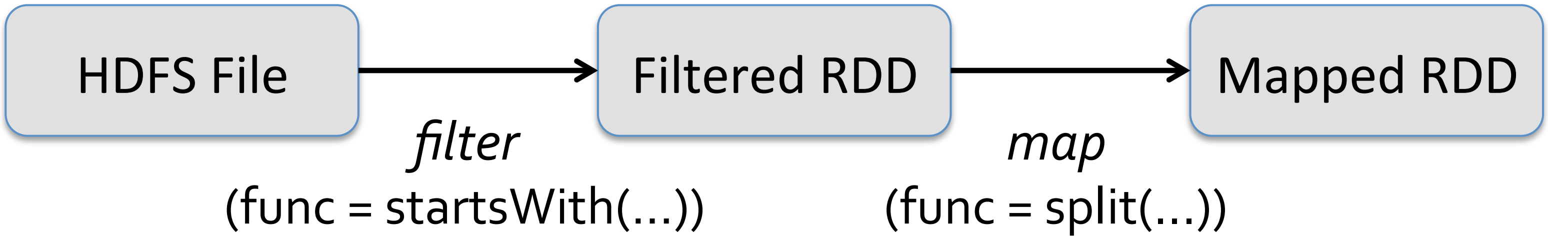
- User can define custom policies to allocate memory to RDDs



Fault Recovery

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])
```



Language Support

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

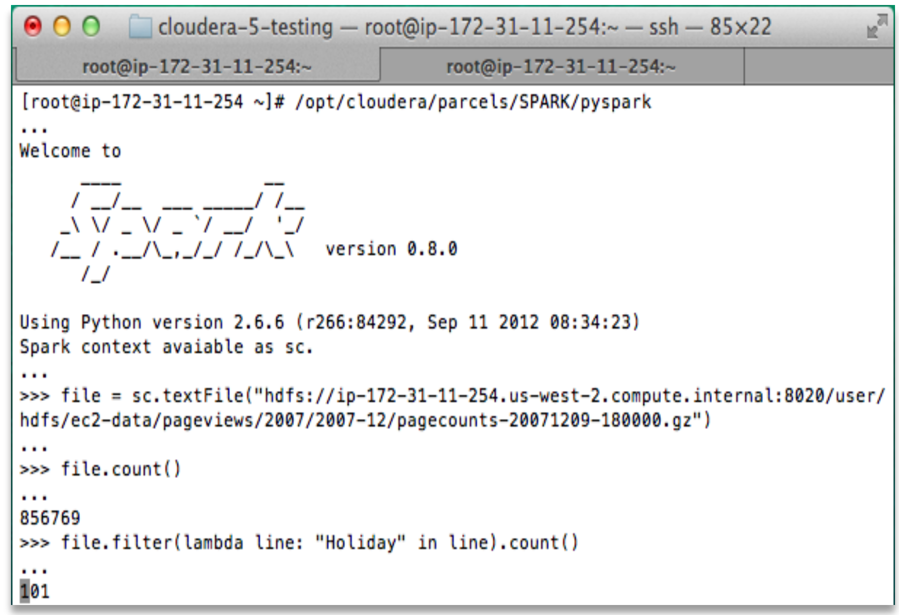
- Python & Scala

Performance

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

Interactive Shell

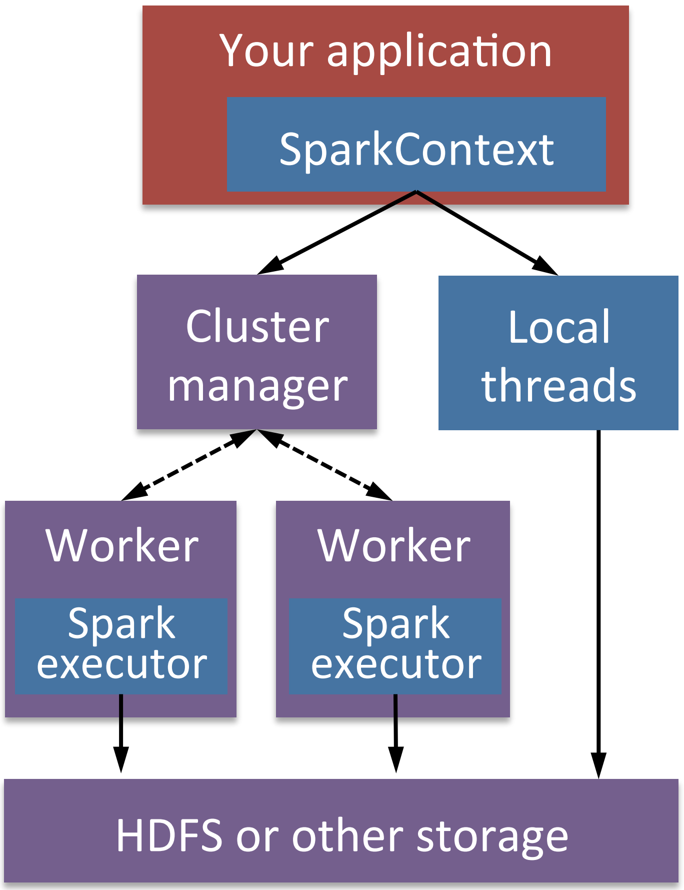
- The Fastest Way to Learn Spark
- Available in Python and Scala
- Runs as an application on an existing Spark Cluster…
- OR Can run locally



JOB EXECUTION

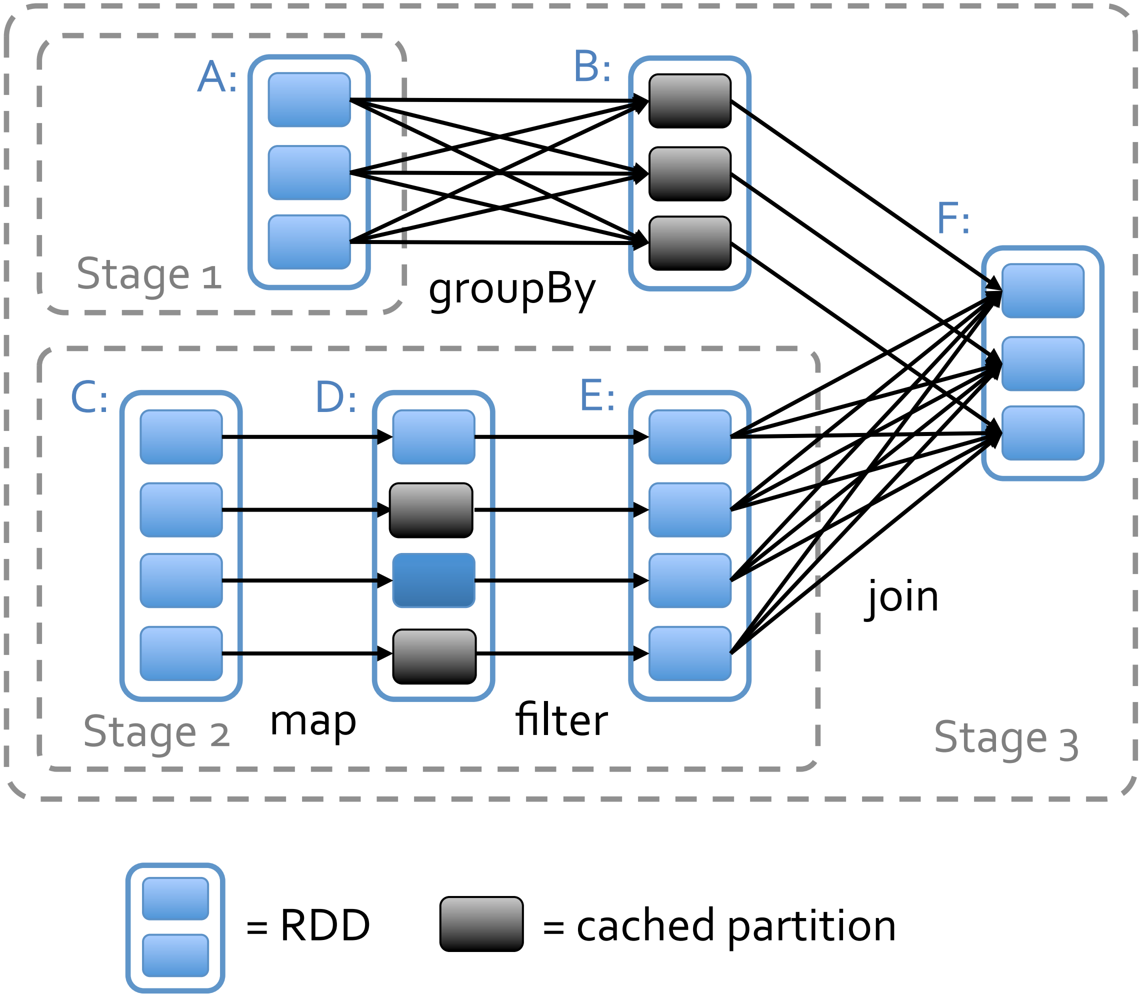
Software Components

- 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, …



Task Scheduler

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



Advanced Features

- 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
- See online docs for details!

Local Execution

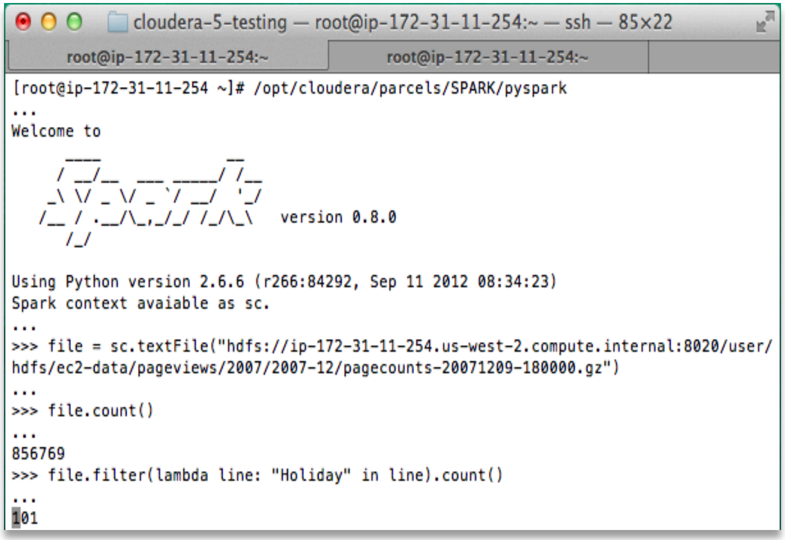
- Just pass `local` or `local[k]` as master URL
- Debug using local debuggers
  - For Java / Scala, just run your program in a debugger
  - For Python, use an attachable debugger (e.g. PyDev)
- Great for development & unit tests

WORKING WITH SPARK

Using the Shell

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
```

SparkContext

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

Creating RDDs

```
# 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)
```

Basic Transformations

```
> 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}

# Fuzzy Evaluation!
> even.collect()
```

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

Basic Actions

```
> 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
```

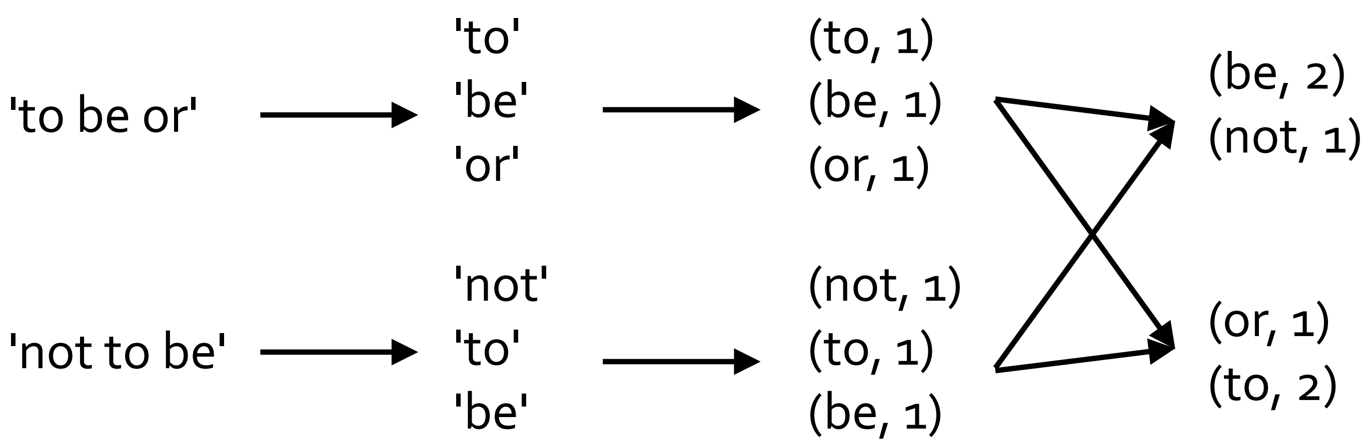
Some Key-Value Operations:

```
> 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

Example: Word Count

```
# 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)
```



Other Key-Value Operations

```
> 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']))
```

Setting the Level of Parallelism

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)
```

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

More RDD Operators

- `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

Add Spark to Your Project

- 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

Create a SparkContext

Scala

```scala
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)

Java

```java
import org.apache.spark.api.java.JavaSparkContext;

JavaSparkContext sc = new JavaSparkContext(
    'masterUrl', 'name', 'sparkHome', new String[] {'app.jar'}));
```

Python

```python
from pyspark import SparkContext

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

Complete App

```python
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

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

Data Model

- 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']

Hive QL

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

```
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%';
```

Caching Data in Shark

- 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 (in Shark 0.8.1):

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

Spark Integration

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)]) = {
    ...
  }
}
```

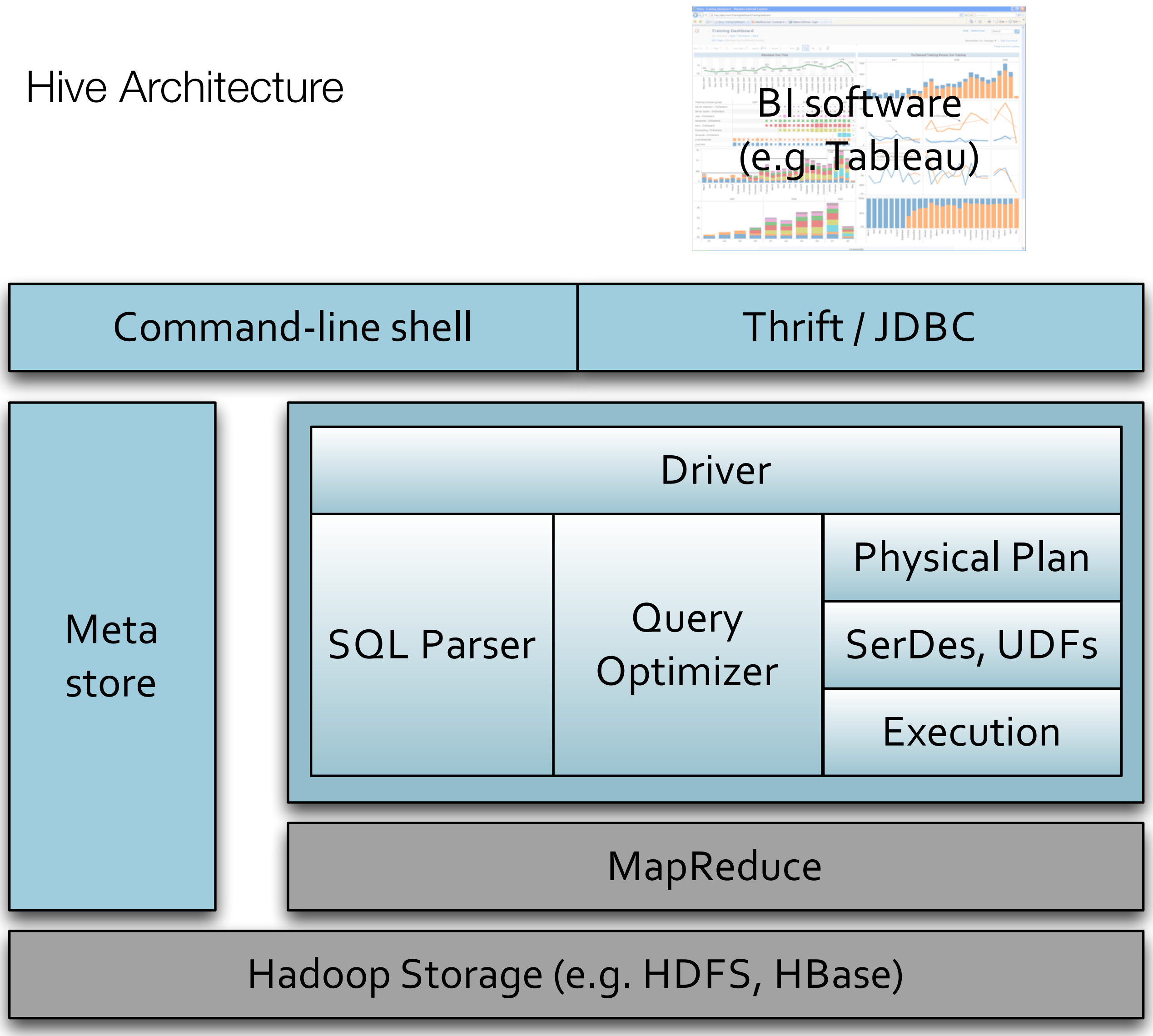
Tuning Degree of Parallelism

- 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 (?)

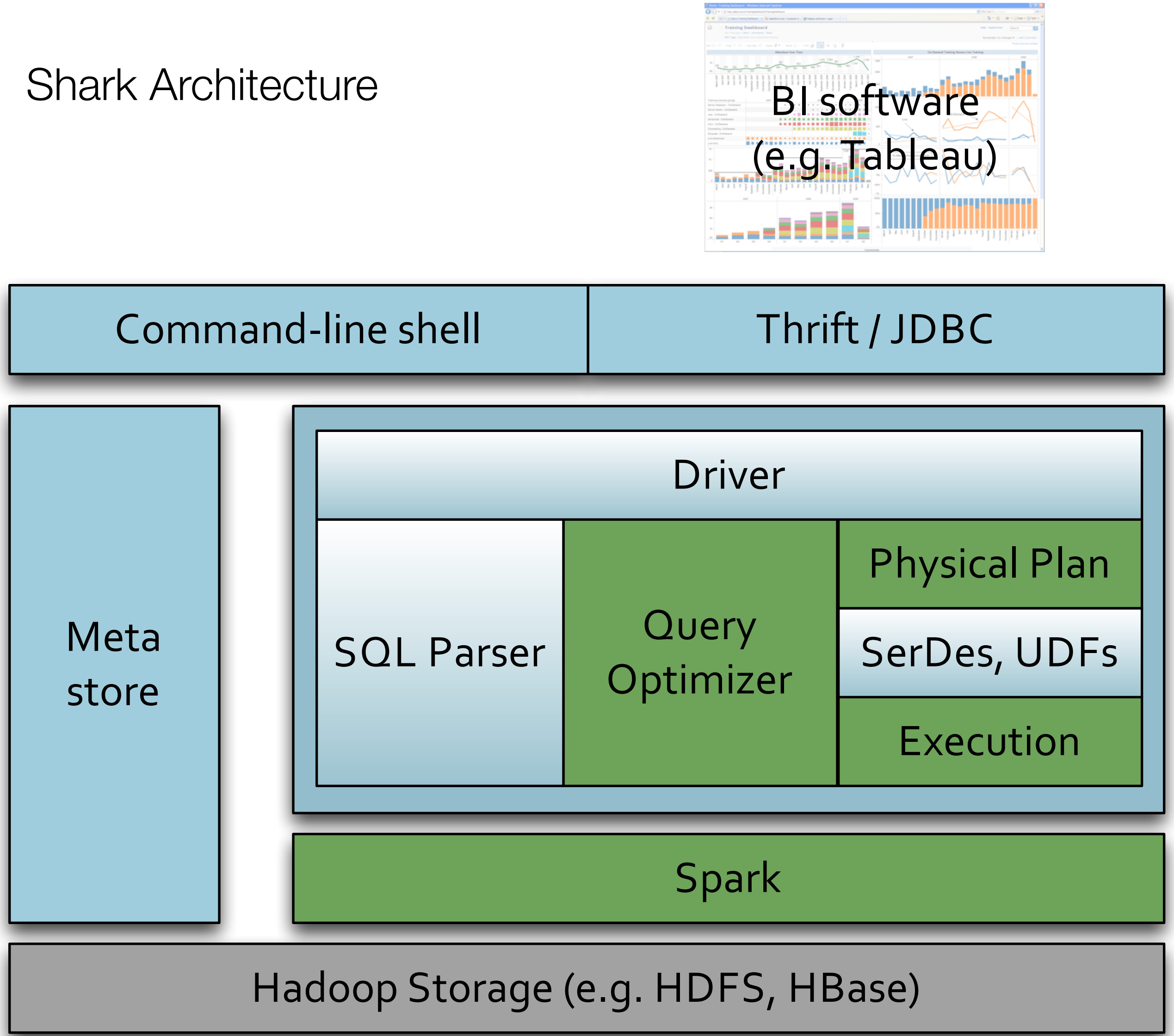
Under the hood

- A better execution engine
  - Hadoop MR is ill-suited for short running SQL
- 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
  - …

Hive Architecture

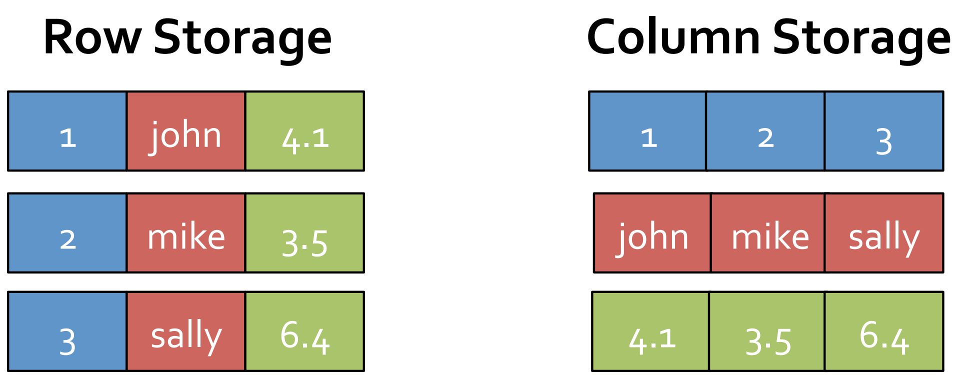


Shark Architecture



Columnar Memory Store

- Column-oriented storage for in-memory tables
  - when we *chache* 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



Spark SQL example (I)

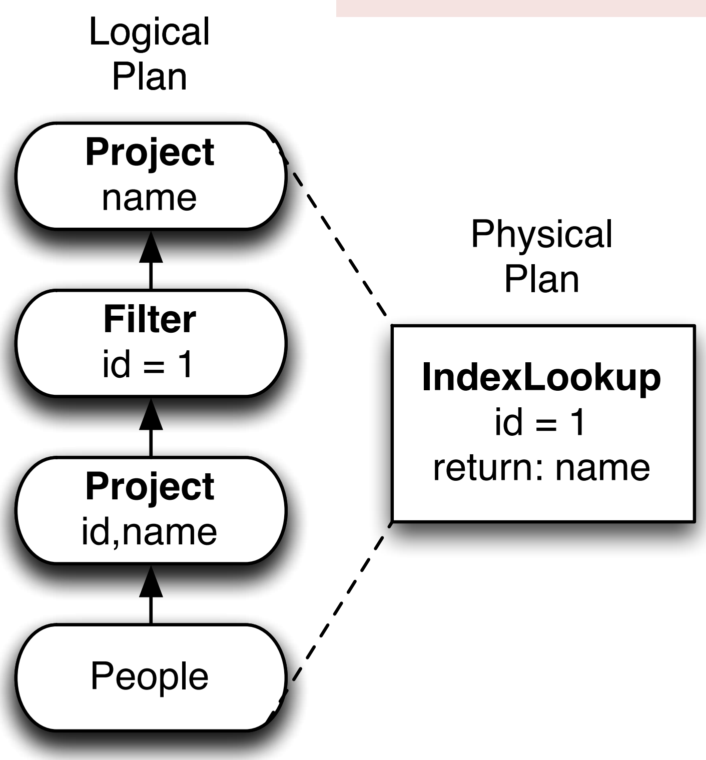
```
# Import SQLContext and data types
> from pyspark.sql import *

# 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()]
```

Spark SQL example (2)

```
> 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
```

Optimized Execution

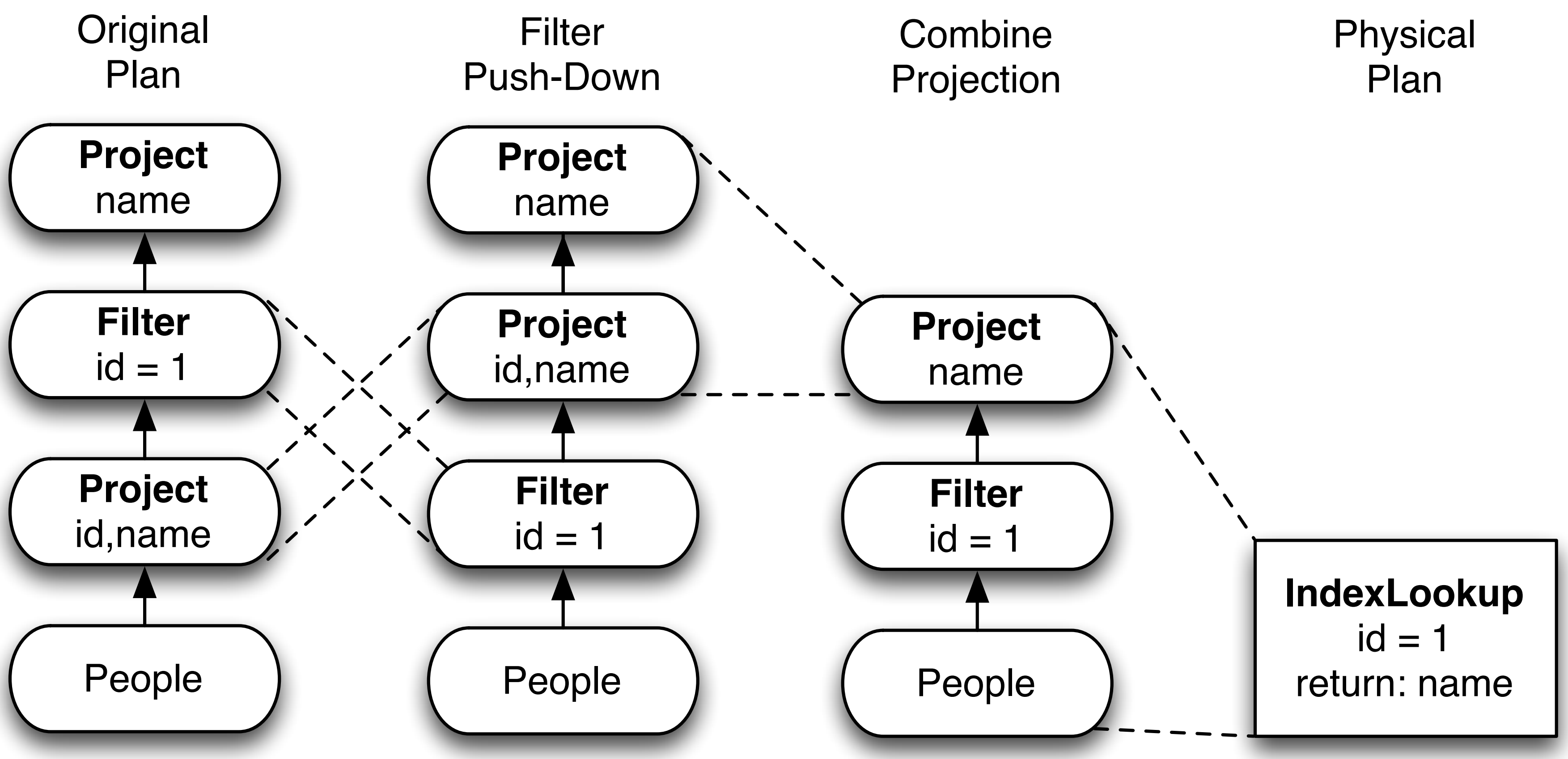
Writing imperative code to optimize such patterns generally is hard.



Instead write simple rules:

- Each rule makes one small change
- Many rules together to fixed point.

Optimizing with Rules



Ongoing Work

- Code generation for query plan (Intel)
- BlinkDB integration (UCB)
- Bloom-filter based pruning (Yahoo!)
- More intelligent optimizer

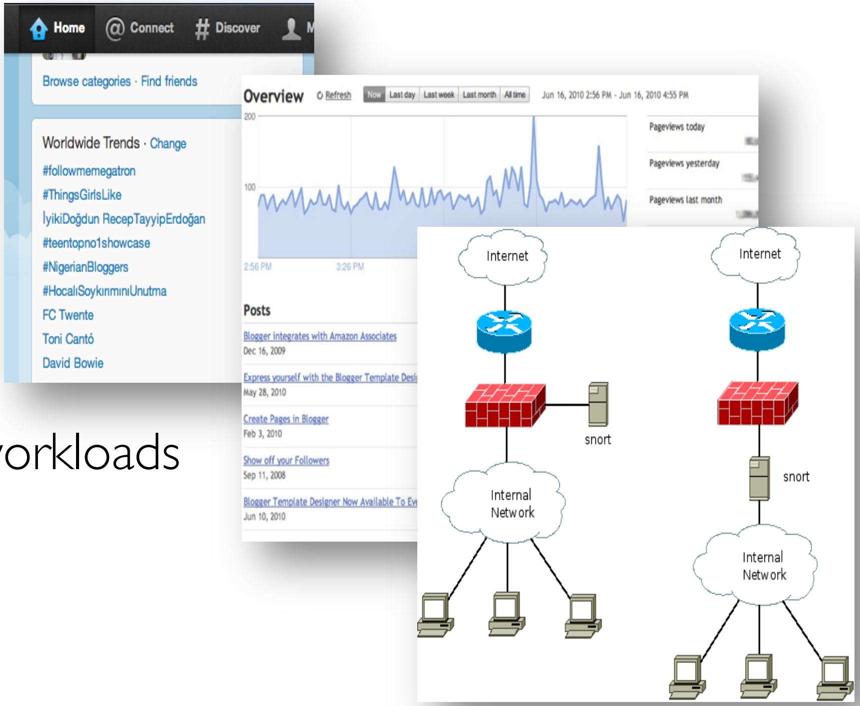
SPARK STREAMING

What is 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.

Motivation

- Many important applications must process large streams of live data and provide results in near-real-time
  - Social network trends
  - Website statistics
  - Intrustion detection systems
  - etc.



- Require large clusters to handle workloads
- Require latencies of few seconds

Need for a framework …

… 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

Case study: XYZ, Inc.

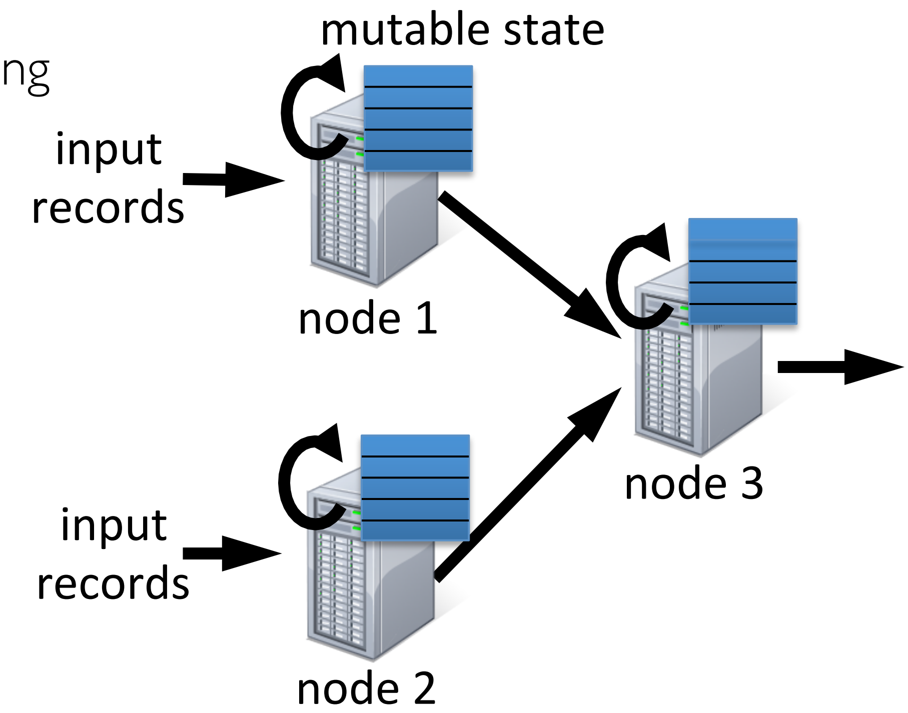
- Any company who wants to process live streaming data has this problem
- Twice the effort to implement any new function
- Twice the number of bugs to solve
- Twice the headache

New Requirement:

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

Stateful Stream Processing

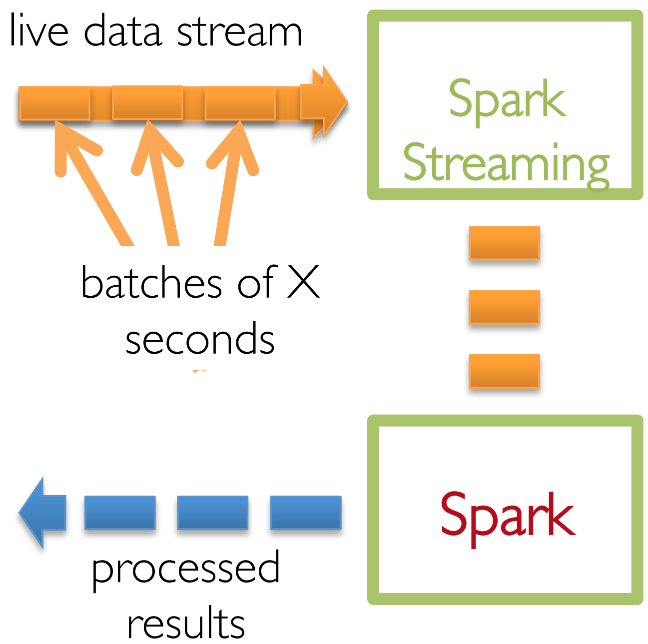
- Traditional streaming systems have a event-driven **record-at-a-time** processing model
  - Each node has mutable state
  - For each record, update state & send new records
- State is lost if node dies!
- Making stateful stream processing be fault-tolerant is challenging



Spark Streaming: Discretized Stream Processing (1)

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

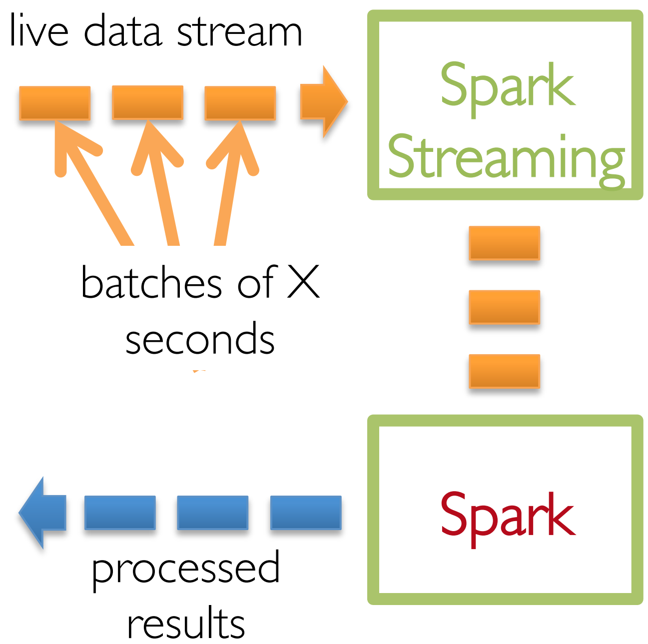
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



Spark Streaming: Discretized Stream Processing (2)

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

- Batch sizes as low as ½ second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system



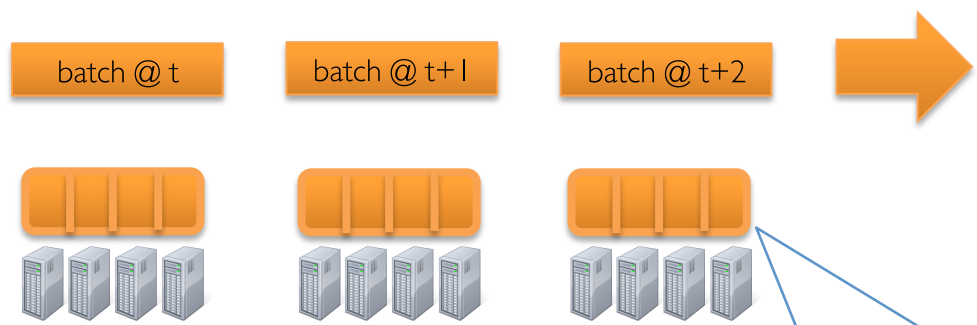
Example 1 – Get hashtags from Twitter

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

DStream: a sequence of RDD representing a stream of data

Twitter Streaming API

tweets DStream



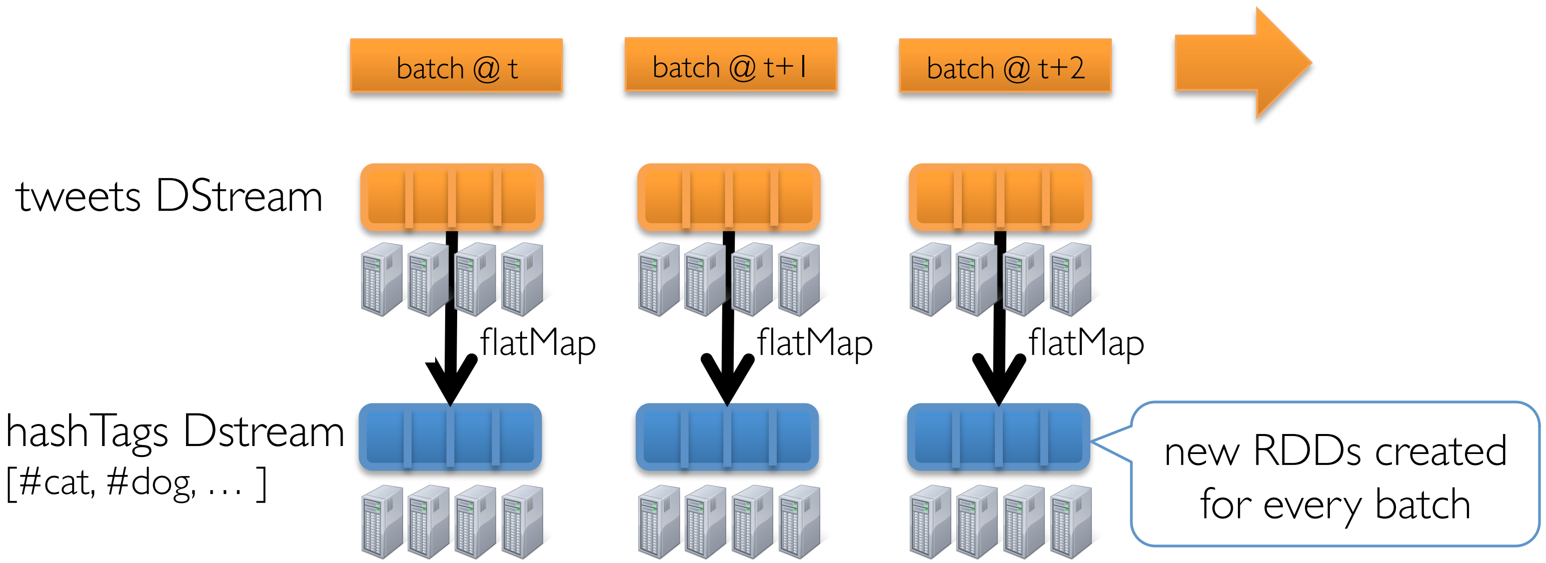
stored in memory as an RDD (immutable, distributed)

Example 1 – Get hashtags from Twitter

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

new DStream

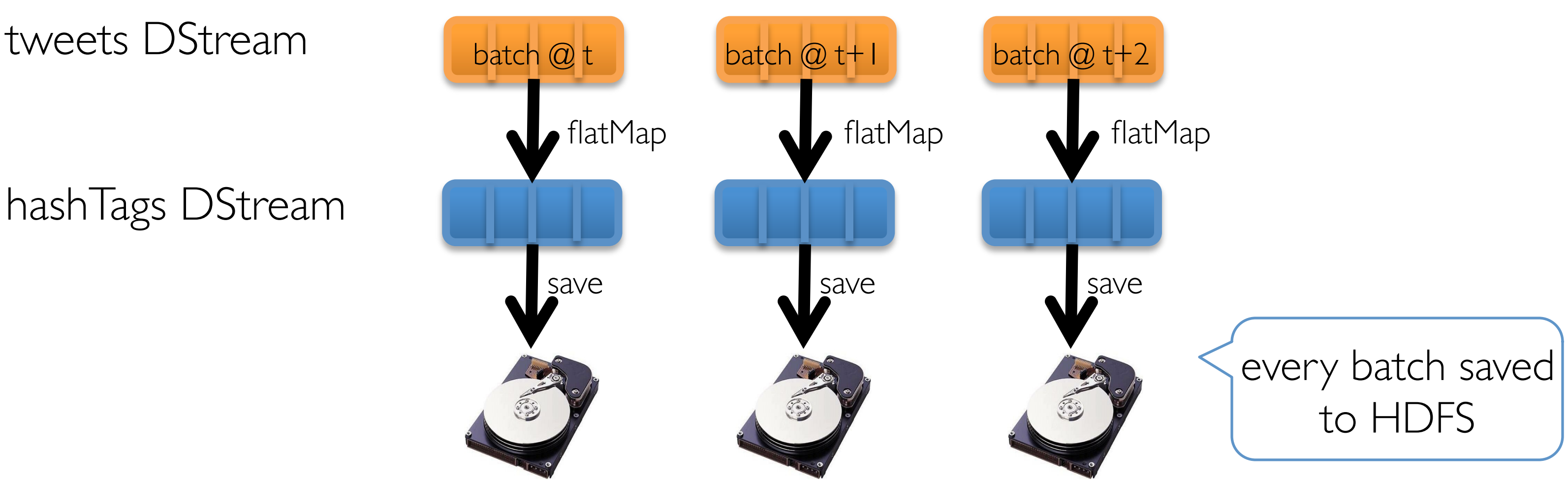
transformation: modify data in one Dstream to create another DStream



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



Java Example

Scala

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

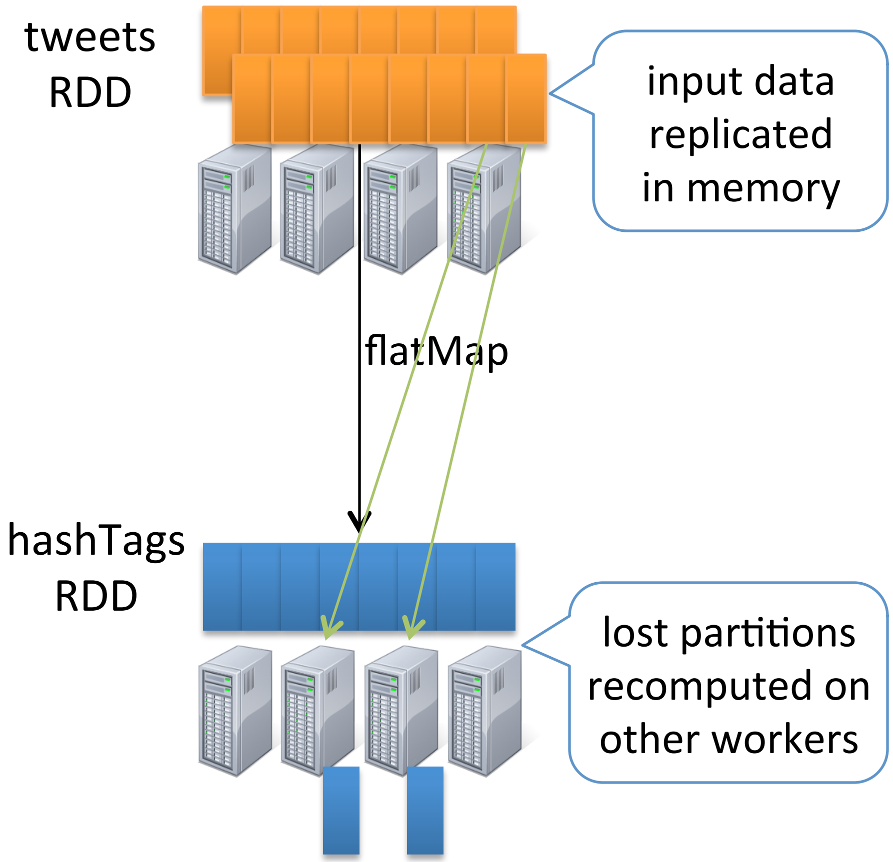
Java

```
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

Fault-tolerance

- RDDs are 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



Example – Counting HashTags

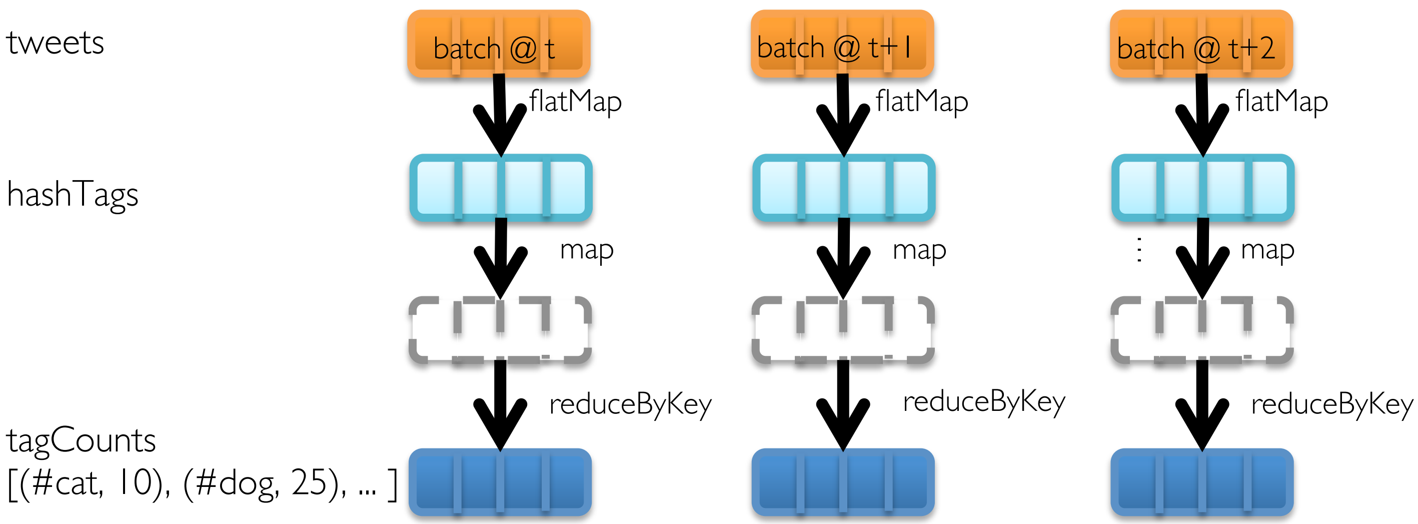
Count the (e.g. most 10 popular) hashtags over last 10 mins

1. Count HashTags from a stream

2. Count HashTags in a time windows from a stream

Count the hashtags

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



Count the hashtags over last 10 mins (1)

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

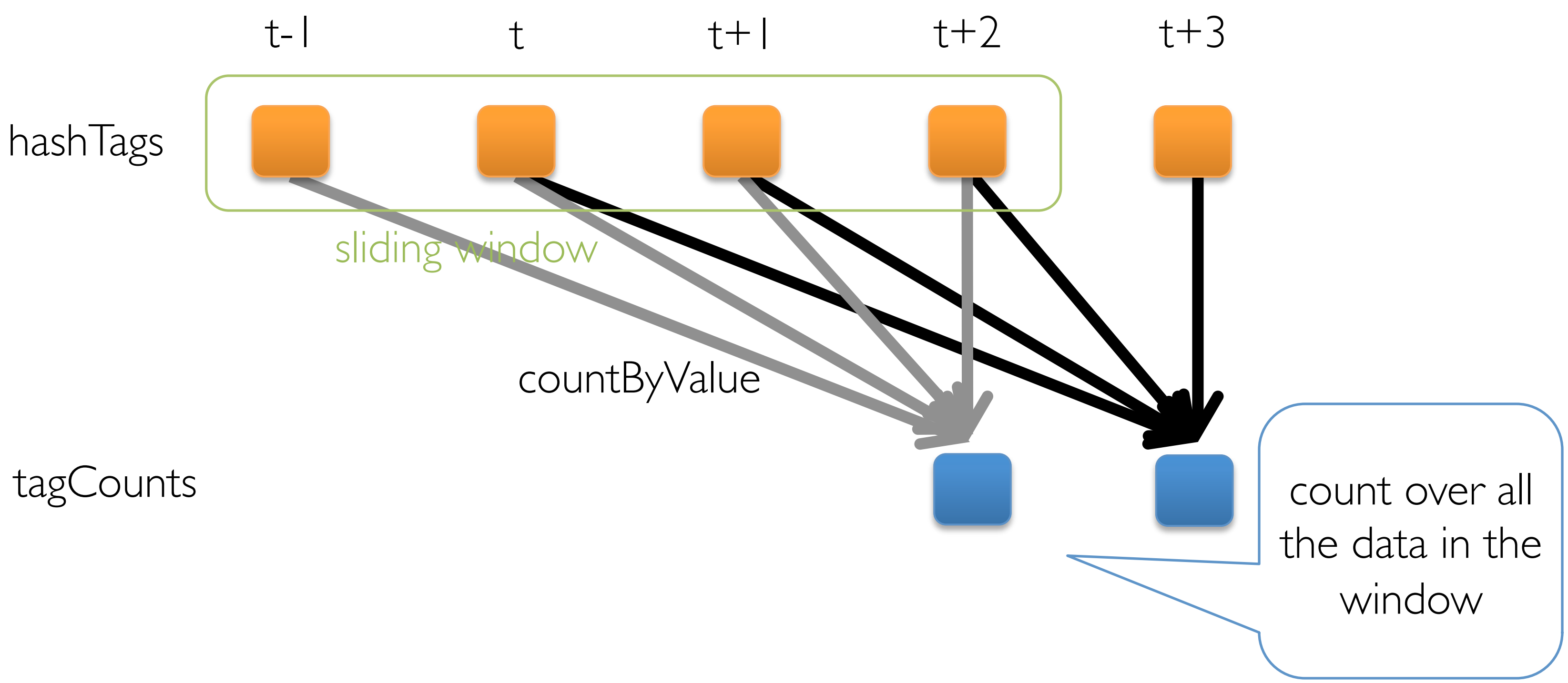
sliding window operation

window length

sliding interval

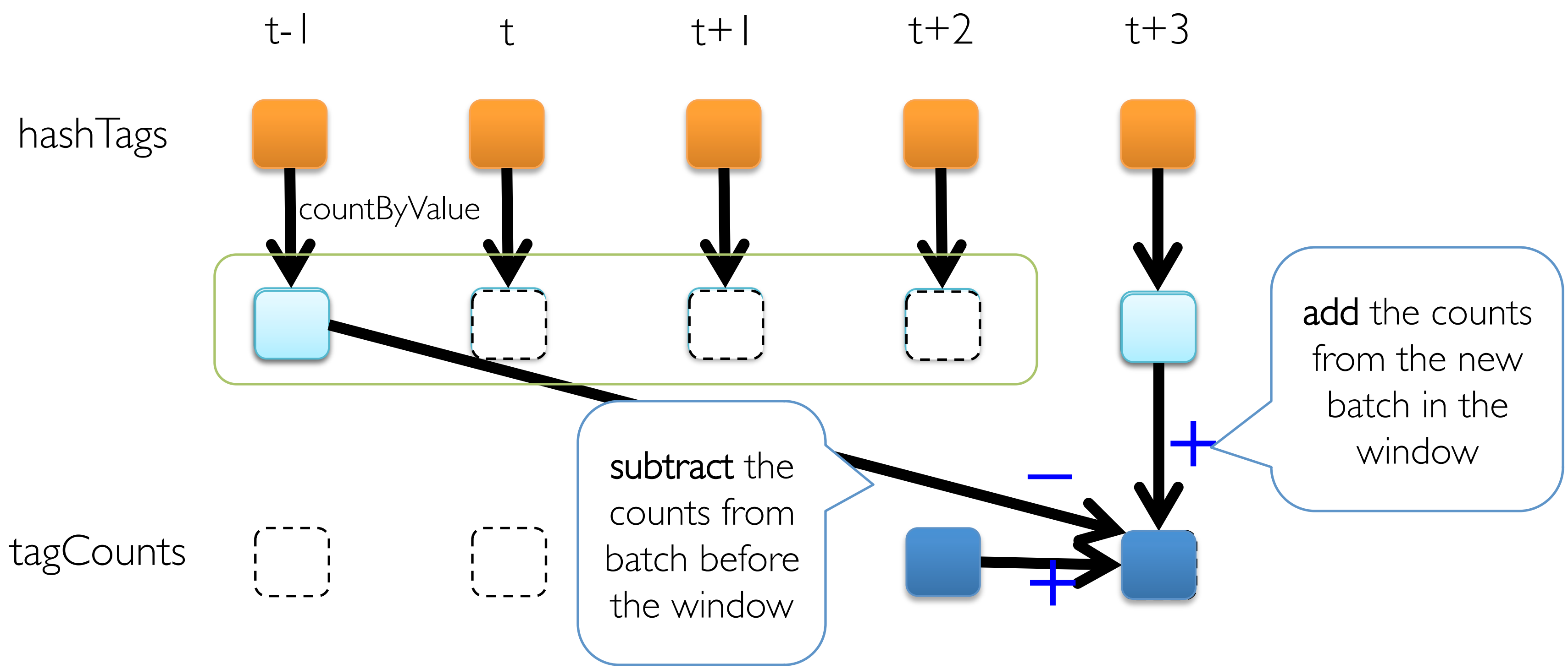
Example – Count the hashtags over last 10 mins (2)

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



Smart window-based countByValue

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



Spark program vs Spark Streaming program

Spark Streaming program on Twitter stream

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

Spark program on Twitter log file

```
val tweets = sc.hadoopFile("hdfs://...")
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFile("hdfs://...")
```

Spark Streaming Conclusion

- 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

GRAPHX

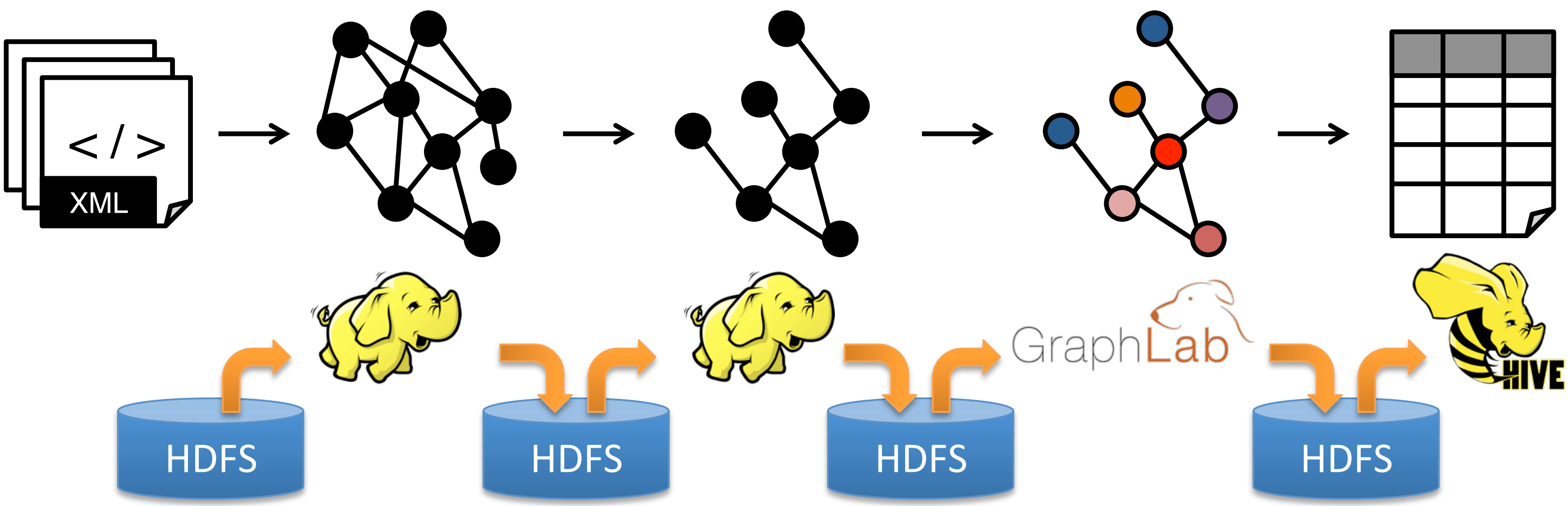
Difficult to Program and Use

- 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

Inefficient

Extensive data movement and duplication across the network and file system



Limited reuse internal data-structures across stages

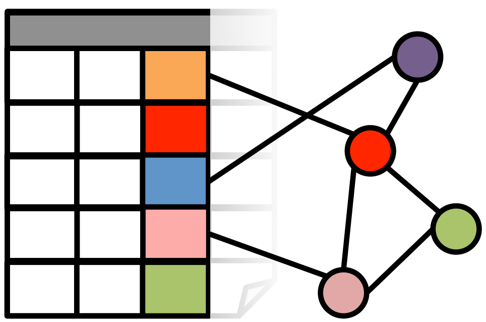
Solution: The GraphX Unified Approach

New API

Blurs the distinction between Tables and Graphs

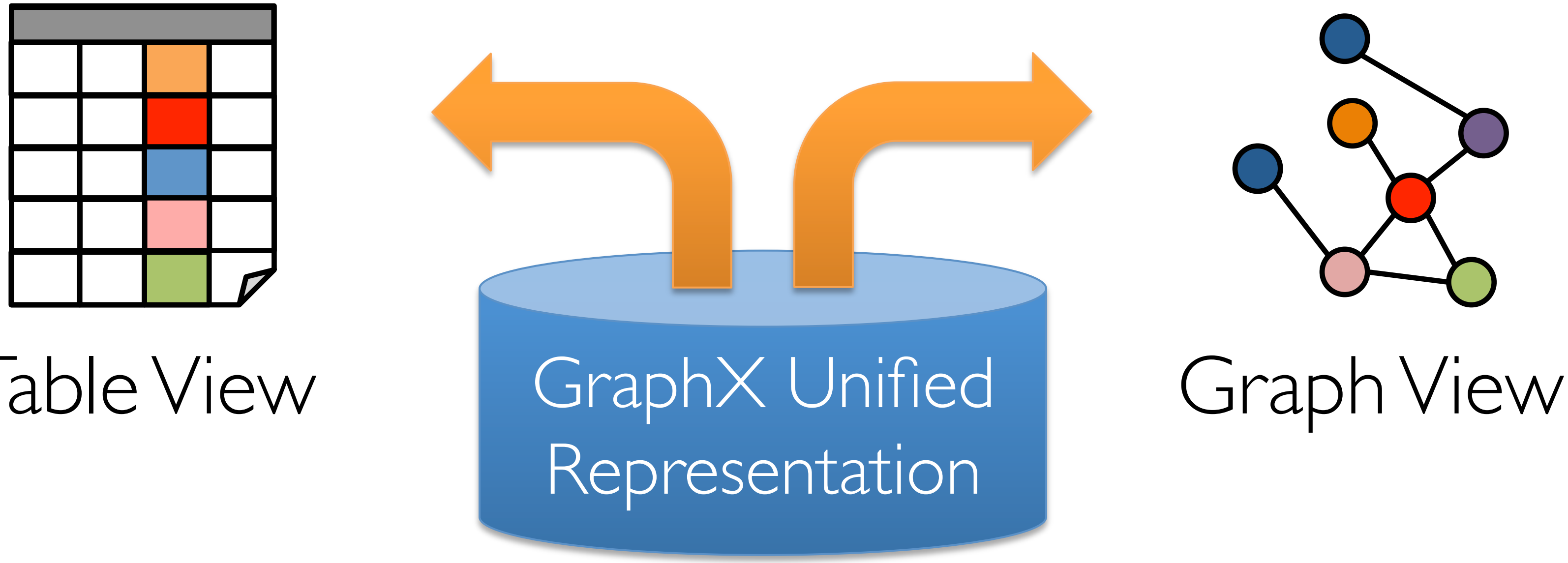
New System

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



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

MLLIB

Machine Learning on Spark

Algorithms

MLlib 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

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

```
points = spark.textFile("hdfs://...")
              .map(parsePoint)

model = KMeans.train(points, k=10)
```

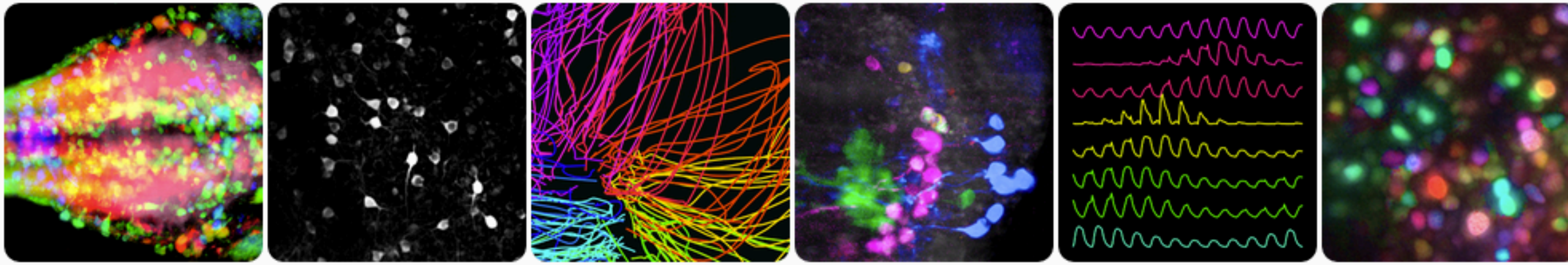
spark.apache.org/mllib/

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 utilties 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/
Youtube: www.youtube.com/watch?v=Gg_5fWIlfgA&list=UURzsq7k4-kT-h3TDUBQ82-w

Big Data Genomics

Big Data Genomics | Blog | Archives | Projects | Mailing List | CLAs

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

At the moment, we a

Variant Call Format

From Wikipedia, the free encyclopedia

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.

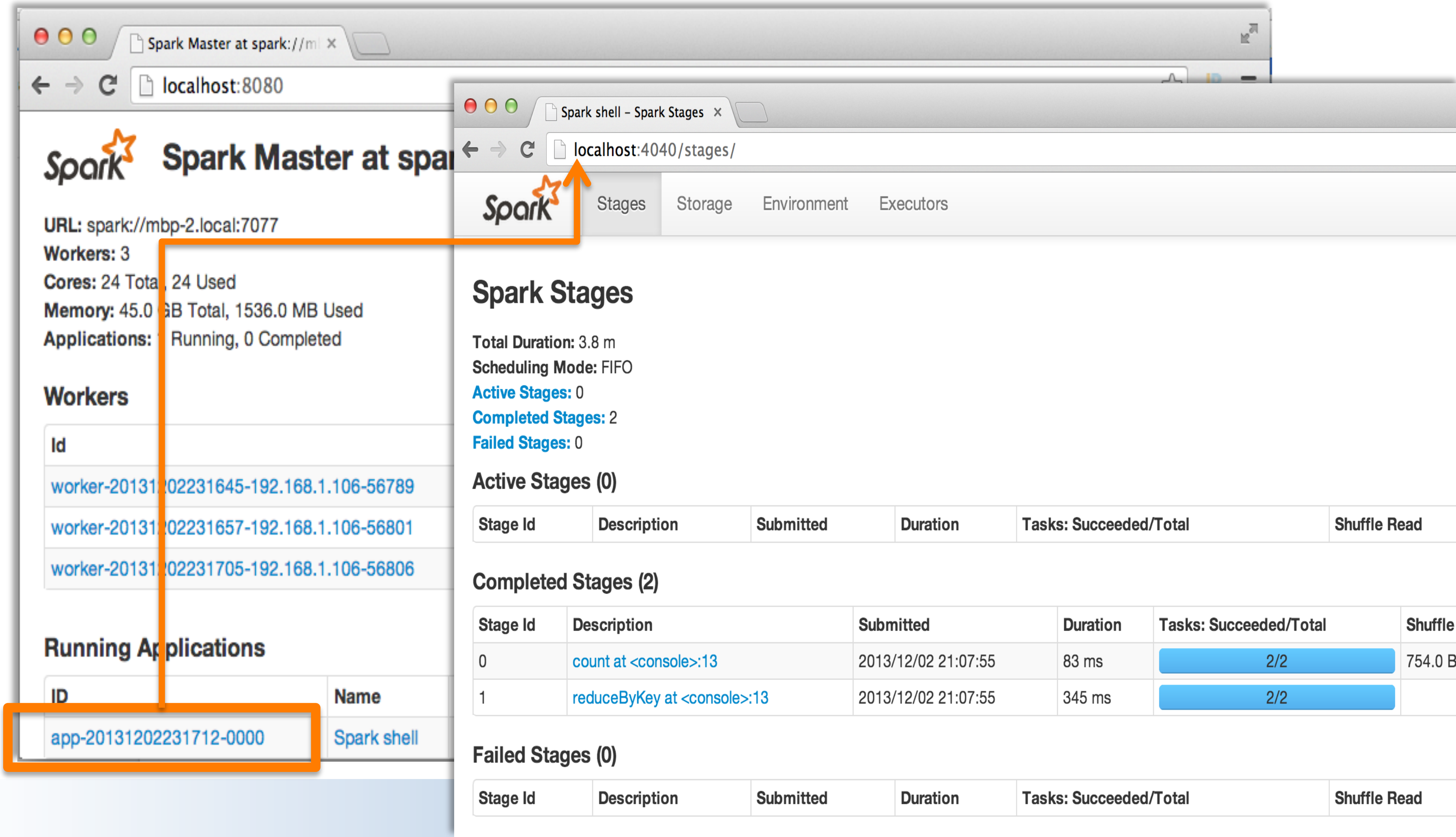
Project Homepage: Homepage: http://bdgenomics.org/projects/
Youtube: www.youtube.com/watch?v=RwyEEMw-NR8&list=UURzsq7k4-kT-h3TDUBQ82-w

Spark

ADDENDUM

Administrative GUIs

http://<Standalone Master>:8080 (by default)



EXAMPLE APPLICATION: PAGERANK

Example: 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

Basic Idea

Give pages ranks (scores) based on links to them

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

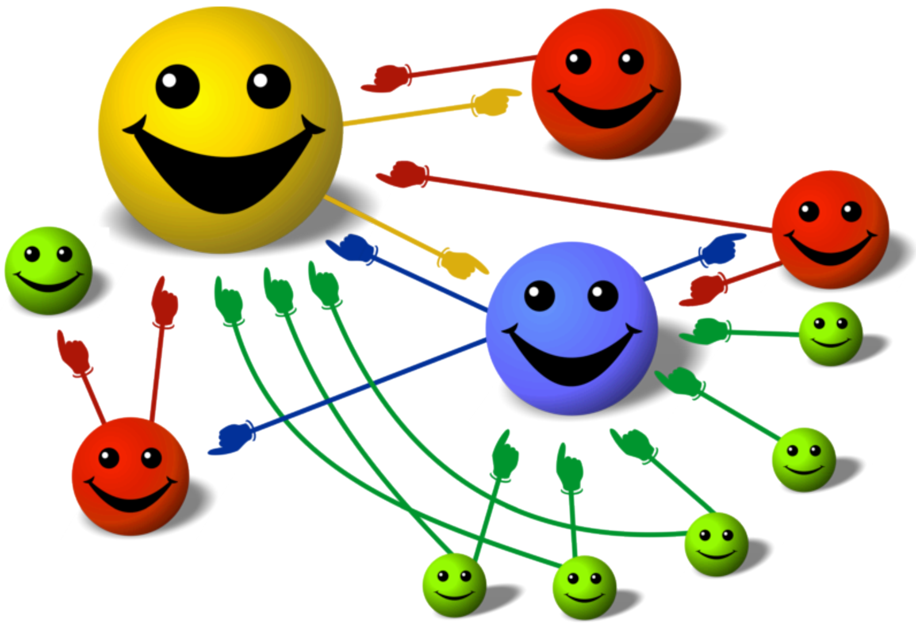
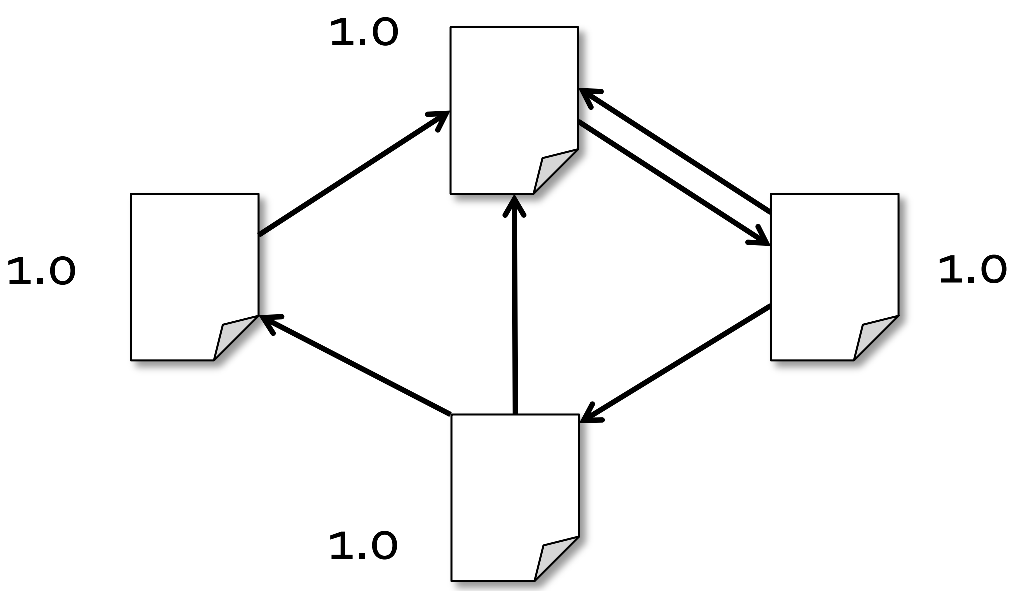


Image: en.wikipedia.org/wiki/File:PageRank-hi-res-2.png

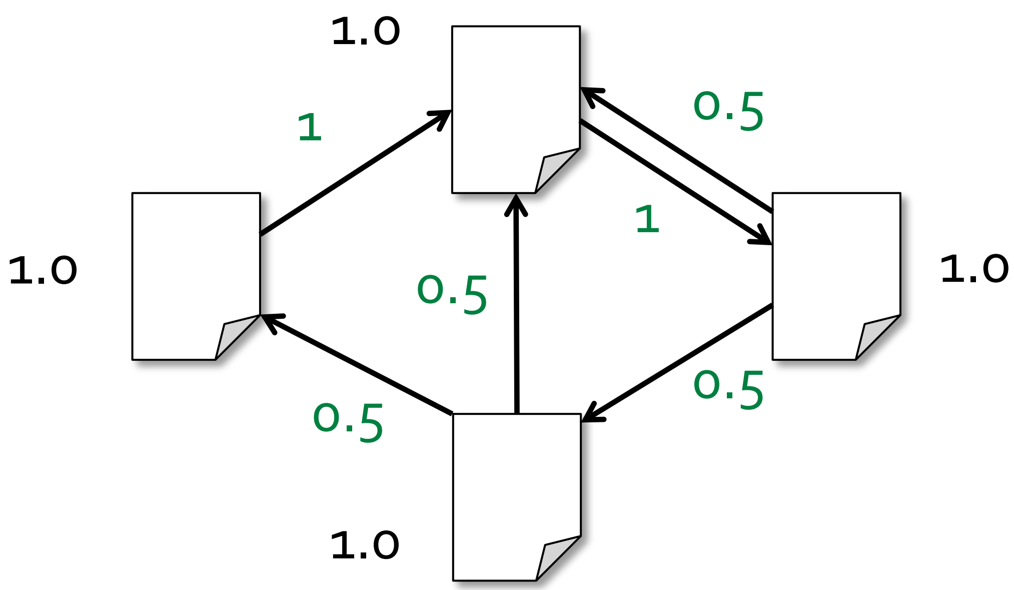
Algorithm

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



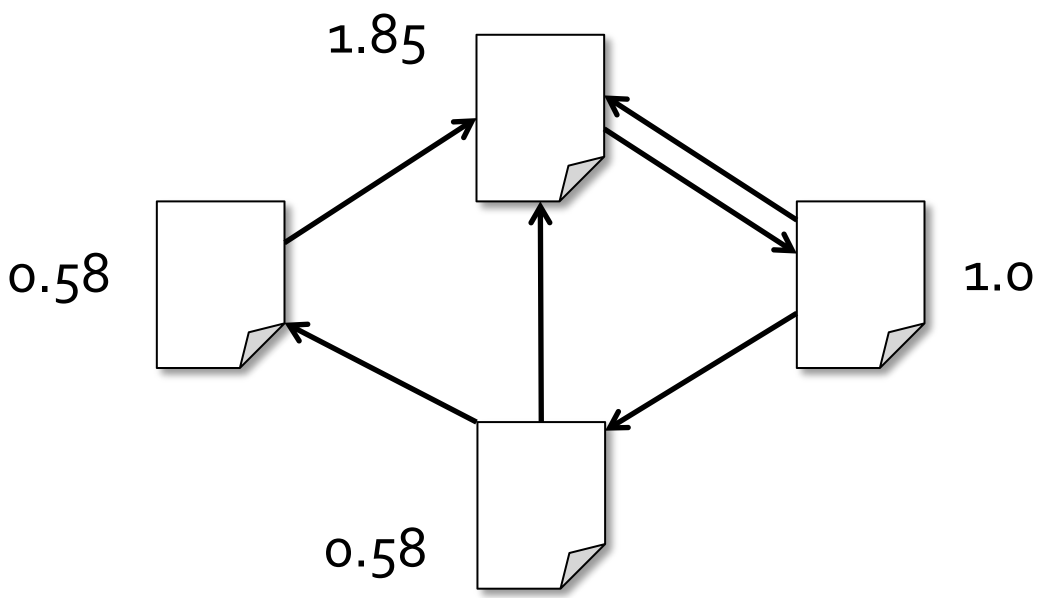
Algorithm

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



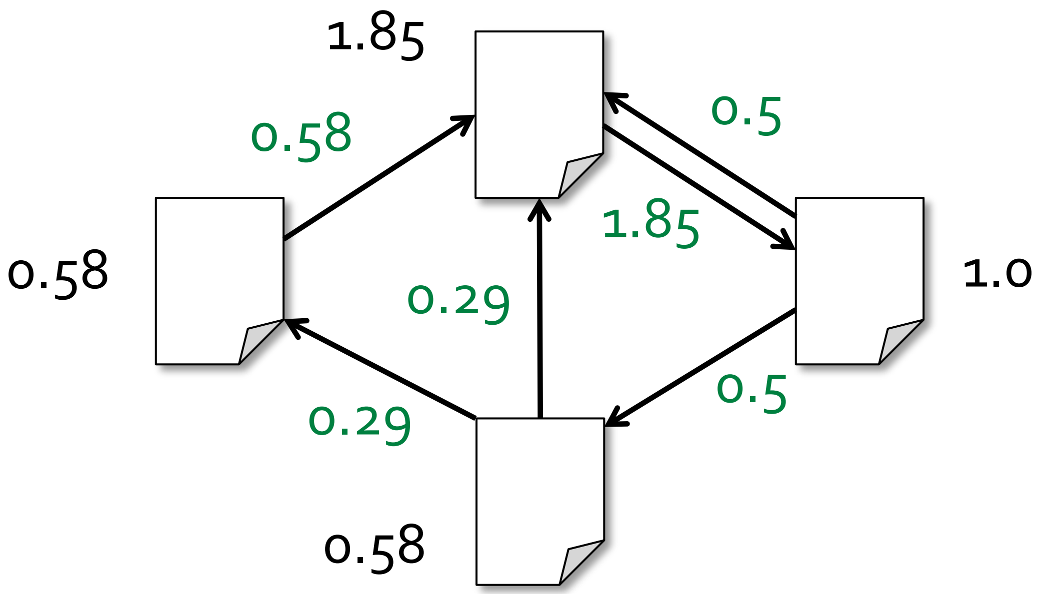
Algorithm

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



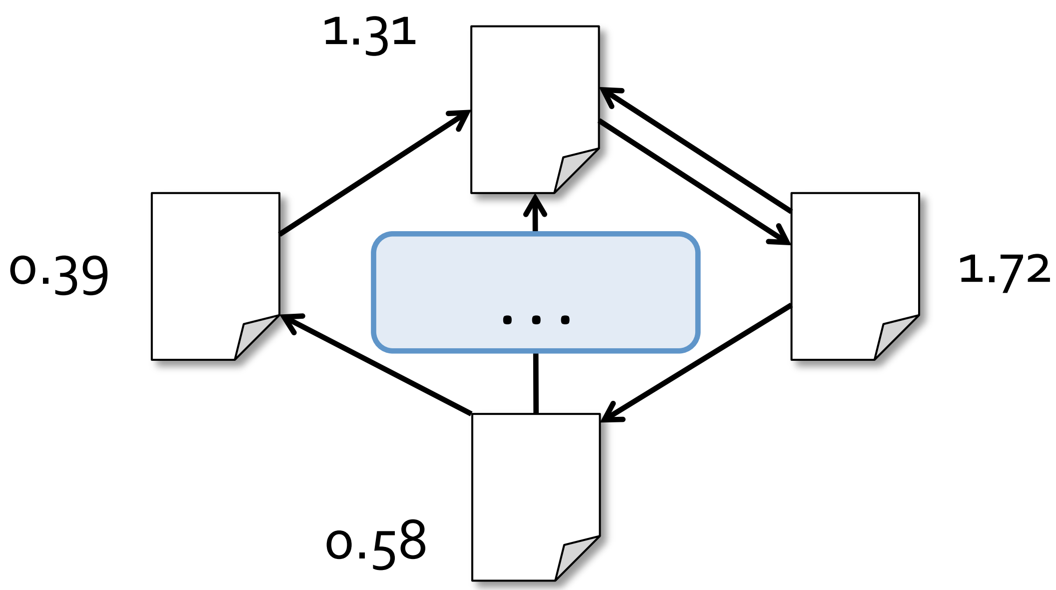
Algorithm

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



Algorithm

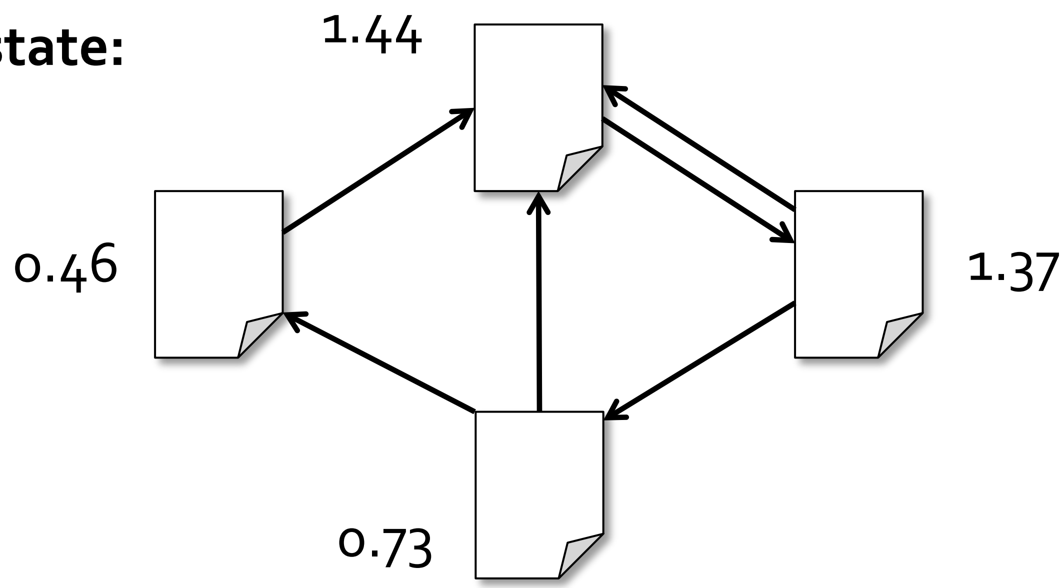
1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times$ contribs



Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to 0.15 + 0.85 × contribs

Final state:



Scala Implementation

```
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(...)
```

References

- 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/