Lightning-fast cluster computing

# Spark SQL and DataFrames
# Spark GraphX
# Spark Mlib
# Spark Streaming

# Chaining transformations

- **What is Spark SQL?**
  - Spark module for structured data processing
  - Replaces Shark (a prior Spark module, now deprecated)
  - Built on top of core Spark

- **What does Spark SQL provide?**
  - The DataFrame API – a library for working with data as tables
    - Defines DataFrames containing Rows and Columns
    - DataFrames are the focus of this chapter!
  - Catalyst Optimizer – an extensible optimization framework
  - A SQL Engine and command line interface

# SQL context

- **The main Spark SQL entry point is a SQL Context object**
  - Requires a SparkContext
  - The SQL Context in Spark SQL is similar to Spark Context in core Spark
- **There are two implementations**
  - **`SQLContext`**
    - basic implementation
  - **`HiveContext`**
    - Reads and writes Hive/HCatalog tables directly
    - Supports full HiveQL language
    - Requires the Spark application be linked with Hive libraries
    - Recommended starting with Spark 1.5

# Creating a SQL context

- **SQLContext is created based on the SparkContext**

Python

```python
from pyspark.sql import SQLContext
sqlCtx = SQLContext(sc)
```

Scala

```scala
import org.apache.spark.sql.SQLContext
val sqlCtx = new SQLContext(sc)
import sqlCtx._
```

# DataFrames

- **DataFrames are the main abstraction in Spark SQL**
  - Analogous to RDDs in core Spark
  - A distributed collection of data organized into named columns
  - Built on a base RDD containing `Row` objects

- **Dat**

```python
# sc is an existing SparkContext.
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

# A JSON dataset is pointed to by path.
# The path can be either a single text file or a directory storing text files.
people = sqlContext.read.json("examples/src/main/resources/people.json")

# The inferred schema can be visualized using the printSchema() method.
people.printSchema()
# root
#  |-- age: integer (nullable = true)
#  |-- name: string (nullable = true)

# Register this DataFrame as a table.
people.registerTempTable("people")

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

# Alternatively, a DataFrame can be created for a JSON dataset represented by
# an RDD[String] storing one JSON object per string.
anotherPeopleRDD = sc.parallelize([
  '{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}'])
anotherPeople = sqlContext.jsonRDD(anotherPeopleRDD)
```

# Creating a DataFrame from Hive

Place your hive-site.xml, core-site.xml (for security configuration), hdfs-site.xml (for HDFS configuration) file in your spark conf/

```python
# sc is an existing SparkContext.
from pyspark.sql import HiveContext
sqlContext = HiveContext(sc)

sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

# Queries can be expressed in HiveQL.
results = sqlContext.sql("FROM src SELECT key, value").collect()
```

# Creating a DataFrame from MySQL

```
>>> dataframe_mysql = sqlContext.read.format("jdbc").option("url", "jdbc:mysq
l://localhost/uber").option("driver", "com.mysql.jdbc.Driver").option("dbtable",
 "trips").option("user", "root").option("password", "root").load()
```

```
>>> dataframe_mysql.show()
+--------------------+--------+---------------+-----+
|dispatching_base_number|    date|active_vehicles|trips|
+--------------------+--------+---------------+-----+
|              B02512|1/1/2015|            190| 1132|
|              B02765|1/1/2015|            225| 1765|
|              B02764|1/1/2015|           3427|29421|
|              B02682|1/1/2015|            945| 7679|
|              B02617|1/1/2015|           1228| 9537|
|              B02598|1/1/2015|            870| 6903|
|              B02598|1/2/2015|            785| 4768|
|              B02617|1/2/2015|           1137| 7065|
|              B02512|1/2/2015|            175|  875|
|              B02682|1/2/2015|            890| 5506|
|              B02765|1/2/2015|            196| 1001|
|              B02764|1/2/2015|           3147|19974|
|              B02765|1/3/2015|            201| 1526|
|              B02617|1/3/2015|           1188|10664|
|              B02598|1/3/2015|            818| 7432|
|              B02682|1/3/2015|            915| 8010|
|              B02512|1/3/2015|            173| 1088|
|              B02764|1/3/2015|           3215|29729|
|              B02512|1/4/2015|            147|  791|
|              B02682|1/4/2015|            812| 5621|
+--------------------+--------+---------------+-----+
```

# Creating a DataFrame from MySQL

```
>>> sqlContext.sql("select * from trips where dispatching_base_number like '%251
2%'").show()
+--------------------+---------+---------------+-----+
|dispatching_base_number|     date|active_vehicles|trips|
+--------------------+---------+---------------+-----+
|              B02512| 1/1/2015|            190| 1132|
|              B02512| 1/2/2015|            175|  875|
|              B02512| 1/3/2015|            173| 1088|
|              B02512| 1/4/2015|            147|  791|
|              B02512| 1/5/2015|            194|  984|
|              B02512| 1/6/2015|            218| 1314|
|              B02512| 1/7/2015|            217| 1446|
|              B02512| 1/8/2015|            238| 1772|
|              B02512| 1/9/2015|            224| 1560|
|              B02512|1/10/2015|            206| 1646|
|              B02512|1/11/2015|            162| 1104|
|              B02512|1/12/2015|            217| 1399|
|              B02512|1/13/2015|            234| 1652|
|              B02512|1/14/2015|            233| 1582|
|              B02512|1/15/2015|            237| 1636|
|              B02512|1/16/2015|            234| 1481|
|              B02512|1/17/2015|            201| 1281|
|              B02512|1/18/2015|            177| 1521|
|              B02512|1/19/2015|            168| 1025|
|              B02512|1/20/2015|            221| 1310|
+--------------------+---------+---------------+-----+
```

# Transforming and querying DataFrames

- **Basic Operations deal with DataFrame metadata (rather than its data), e.g.**
  - **`schema`** – returns a Schema object describing the data
  - **`printSchema`** – displays the schema as a visual tree
  - **`cache`** / **`persist`** – persists the DataFrame to disk or memory
  - **`columns`** – returns an array containing the names of the columns
  - **`dtypes`** – returns an array of (column-name,type) pairs
  - **`explain`** – prints debug information about the DataFrame to the console

```
>>> from pyspark.sql import SQLContext
>>> sqlcontext = SQLContext(sc)
>>> people = sqlcontext.read.json('input/people.json')
>>> for item in people.dtypes:
...     print item
...
('age', 'bigint')
('name', 'string')
('pcode', 'string')
>>> 
```

https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#

# Working data in a DataFrame

- **Queries – create a new DataFrame**
  - DataFrames are immutable
  - Queries are analogous to RDD transformations
- **Actions – return data to the Driver**
  - Actions trigger "lazy" execution of queries

# Working data in a DataFrame

- **Some DataFrame actions**
  - **`collect`** – return all rows as an array of **`Row`** objects
  - **`take(n)`** – return the first **`n`** rows as an array of **`Row`** objects
  - **`count`** – return the number of rows
  - **`show(n)`** – display the first **`n`** rows (default=20)

```
>>> people.count()
3
>>> people.show(2)
+----+-----+-----+
| age| name|pcode|
+----+-----+-----+
|null|Alice|94304|
|  30|  Bob|94304|
+----+-----+-----+
only showing top 2 rows

>>> 
```

# DataFrame queries

- **DataFrame query methods return new DataFrames**
  - Queries can be chained like transformations
- **Some query methods**
  - **`distinct`** – returns a new DataFrame with distinct elements of this DF
  - **`join`** – joins this DataFrame with a second DataFrame
    - several variants for inside, outside, left, right, etc.
  - **`limit`** – a new DF with the first **`n`** rows of this DataFrame
  - **`select`** – a new DataFrame with data from one or more columns of the base DataFrame
  - **`filter`** – a new DataFrame with rows meeting a specified condition

# DataFrame queries

- **Some query operations take strings containing simple query expressions**
  - Such as **`select`** and **`where`**
- **Example: `select`**

| age | name | pcode |
|---|---|---|
| null | Alice | 94304 |
| 30 | Brayden | 94304 |
| 19 | Carla | 10036 |
| 46 | Diana | null |
| null | Étienne | 94104 |

```
peopleDF.
  select("age")
```

| age |
|---|
| null |
| 30 |
| 19 |
| 46 |
| null |

```
peopleDF.
  select("name","age")
```

| name | age |
|---|---|
| Alice | null |
| Brayden | 30 |
| Carla | 19 |
| Diana | 46 |
| Étienne | null |

# DataFrame queries

- **Example: `where`**

```
peopleDF.
   where("age > 21")
```

| age | name | pcode |
|---|---|---|
| null | Alice | 94304 |
| 30 | Brayden | 94304 |
| 19 | Carla | 10036 |
| 46 | Diana | null |
| null | Étienne | 94104 |

| age | name | pcode |
|---|---|---|
| 30 | Brayden | 94304 |
| 46 | Diana | null |

# Query DataFrame using columns

- **Some DF queries take one or more *columns* or *column expressions***
  - Required for more sophisticated operations
- **Some examples**
  - **`select`**
  - **`sort`**
  - **`join`**
  - **`where`**

# Query DataFrame using columns

```
>>> people.select(people.age).show()
+----+
| age|
+----+
|null|
|  30|
|  19|
+----+

>>> people.select(people.name,people.age+10).show()
+-------+----------+
|   name|(age + 10)|
+-------+----------+
|  Alice|      null|
|    Bob|        40|
|Charlie|        29|
+-------+----------+

>>> 
```

```
>>> people.sort(people.age.desc()).show()
+----+-------+-----+
| age|   name|pcode|
+----+-------+-----+
|  30|    Bob|94304|
|  19|Charlie|10036|
|null|  Alice|94304|
+----+-------+-----+

>>> 
```

# SQL queries

- **Spark SQL also supports the ability to perform SQL queries**
  - First, register the DataFrame as a “table” with the SQL Context

```
peopleDF.registerTempTable("people")
sqlCtx.sql("""SELECT * FROM people WHERE name LIKE "A%" """)
```


```
peopleDF.registerTempTable("people")
sqlCtx.sql("""SELECT * FROM people WHERE name LIKE "A%" """)
```


| age | name | pcode |
|---|---|---|
| null | Alice | 94304 |
| 30 | Brayden | 94304 |
| 19 | Carla | 10036 |
| 46 | Diana | null |
| null | Étienne | 94104 |

| age | name | pcode |
|---|---|---|
| null | Alice | 94304 |

# Saving DataFrames

- **Data in DataFrames can be saved to a data source**
  - Built in support for JDBC and Parquet File
    - **`createJDBCTable`** – create a new table in a database
    - **`insertInto`** – save to an existing table in a database
    - **`saveAsParquetFile`** – save as a Parquet file (including schema)
    - **`saveAsTable`** – save as a Hive table (HiveContext only)
  - Can also use third party and custom data sources
    - **`save`** – generic base function

# DataFrames and RDDs

- **DataFrames are built on RDDs**
  - Base RDDs contain **Row** objects
  - Use **rdd** to get the underlying RDD

```
peopleRDD = peopleDF.rdd
```

**peopleDF**

| age | name | pcode |
|---|---|---|
| null | Alice | 94304 |
| 30 | Brayden | 94304 |
| 19 | Carla | 10036 |
| 46 | Diana | null |
| null | Étienne | 94104 |

**peopleRDD**

| |
|---|
| Row[null,Alice,94304] |
| Row[30,Brayden,94304] |
| Row[19,Carla,10036] |
| Row[46,Diana,null] |
| Row[null,Étienne,94104] |

# DataFrames and RDDs

- **Row RDDs have all the standard Spark actions and transformations**
  - Actions – **`collect`**, **`take`**, **`count`**, etc.
  - Transformations – **`map`**, **`flatMap`**, **`filter`**, etc.
- **Row RDDs can be transformed into PairRDDs to use map-reduce methods**

# Working with Row objects

- **The syntax for extracting data from Rows depends on language**
- **Python**
  - Column names are object attributes
    - **`row.age`** – return age column value from row
- **Scala**
  - Use Array-like syntax
    - **`row(0)`** – returns element in the first column
    - **`row(1)`** – return element in the second column
    - etc.
  - Use type-specific **`get`** methods to return typed values
    - **`row.getString(n)`** – returns $n^{th}$ column as a String
    - **`row.getInt(n)`** – returns $n^{th}$ column as an Integer
    - etc.

# Extracting data from rows

- **Extract data from Rows**

```
peopleRDD = peopleDF.rdd
peopleByPCode = peopleRDD \
   .map(lambda row(row.pcode,row.name)) \
   .groupByKey()
```

```
val peopleRDD = peopleDF.rdd
peopleByPCode = peopleRDD.
  map(row => (row(2),row(1))).
  groupByKey())
```

| |
|---|
| Row[null,Alice,94304] |
| Row[30,Brayden,94304] |
| Row[19,Carla,10036] |
| Row[46,Diana,null] |
| Row[null,Étienne,94104] |

| |
|---|
| (94304,Alice) |
| (94304,Brayden) |
| (10036,Carla) |
| (null,Diana) |
| (94104,Étienne) |

| |
|---|
| (null,[Diana]) |
| (94304,[Alice,Brayden]) |
| (10036,[Carla]) |
| (94104,[Étienne]) |

# Covert RDD to DataFrame

- You can also create a DF from an RDD
  - `sqlCtx.createDataFrame(rdd)`

# ML and GraphX in Spark

# Common spark use case

- **Spark is especially useful when working with any combination of:**
  - Large amounts of data
    - Distributed storage
  - Intensive computations
    - Distributed computing
  - Iterative algorithms
    - In-memory processing and pipelining

# Common spark use case

- **Examples**
  - Risk analysis
    - "How likely is this borrower to pay back a loan?"
  - Recommendations
    - "Which products will this customer enjoy?"
  - Predictions
    - "How can we prevent service outages instead of simply reacting to them?"
  - Classification
    - "How can we tell which mail is spam and which is legitimate?"

# Spark examples

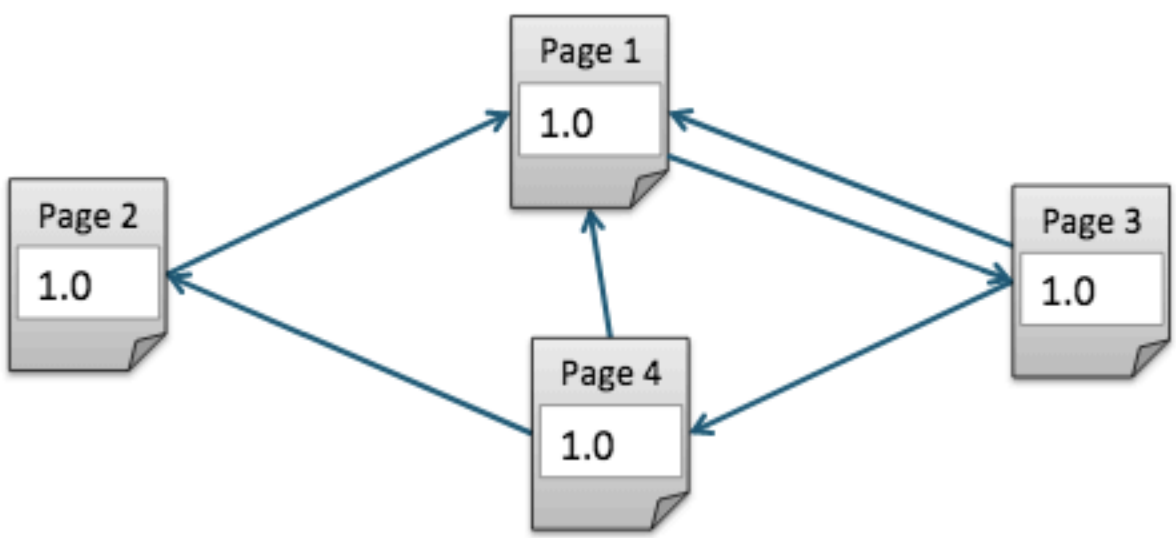
- **Spark includes many example programs that demonstrate some common Spark programming patterns and algorithms**
  - k-means
  - Logistic regression
  - Calculate pi
  - Alternating least squares (ALS)
  - Querying Apache web logs
  - Processing Twitter feeds

# Iterative algorithms in Spark: PageRank

- **PageRank gives web pages a ranking score based on links from other pages**
  - Higher scores given for more links, and links from other high ranking pages
- **Why do we care?**
  - PageRank is a classic example of big data analysis (like WordCount)
    - Lots of data – needs an algorithm that is distributable and scalable
    - Iterative – the more iterations, the better than answer

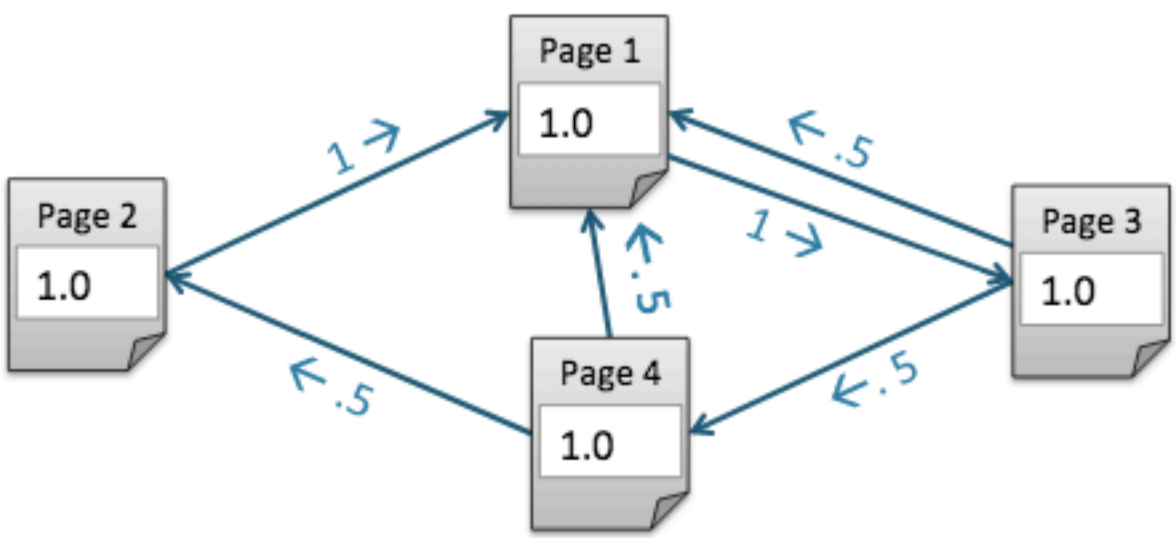
# PageRank algorithm

1. **Start each page with a rank of 1.0**



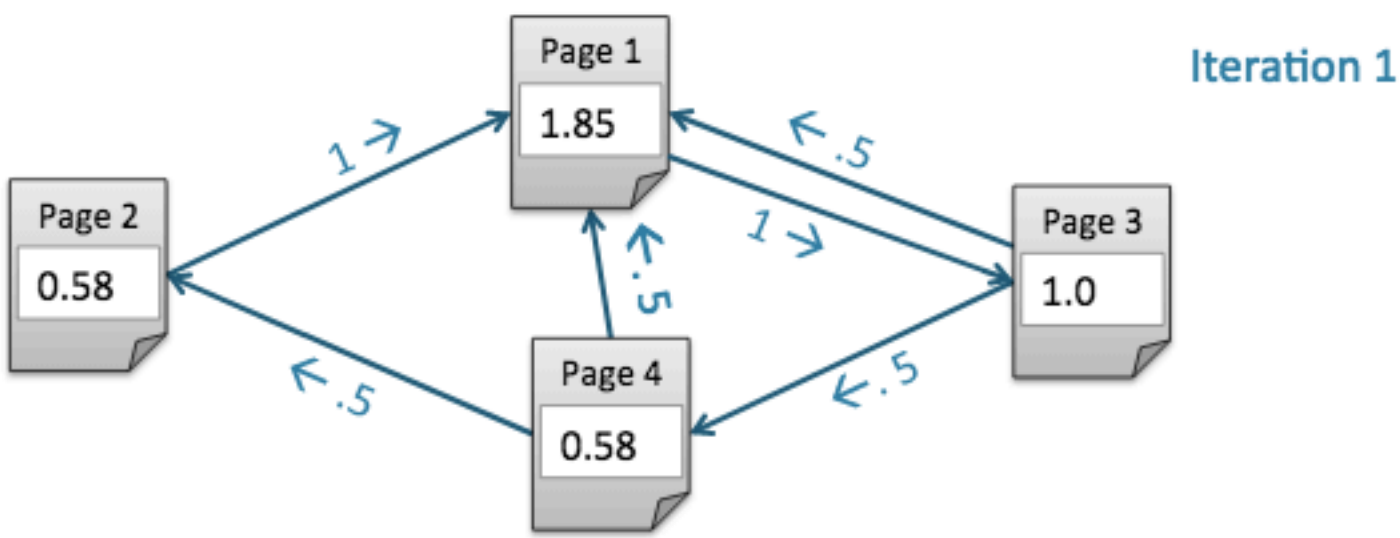
# PageRank algorithm

1. **Start each page with a rank of 1.0**
2. **On each iteration:**
   1. each page contributes to its neighbors its own rank divided by the number of its neighbors: $contrib_p = rank_p / neighbors_p$



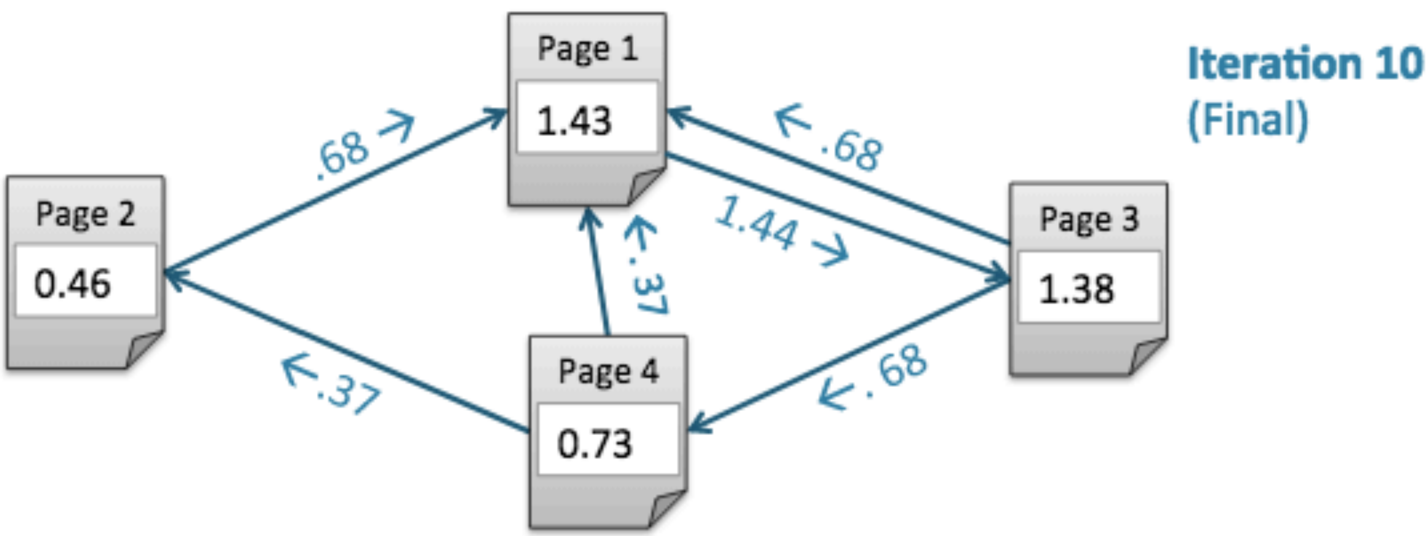
# PageRank algorithm

1. **Start each page with a rank of 1.0**
2. **On each iteration:**
   1. each page contributes to its neighbors its own rank divided by the number of its neighbors: $contrib_p = rank_p / neighbors_p$
   2. Set each page's new rank based on the sum of its neighbors contribution: $new\text{-}rank = \Sigma contribs * .85 + .15$



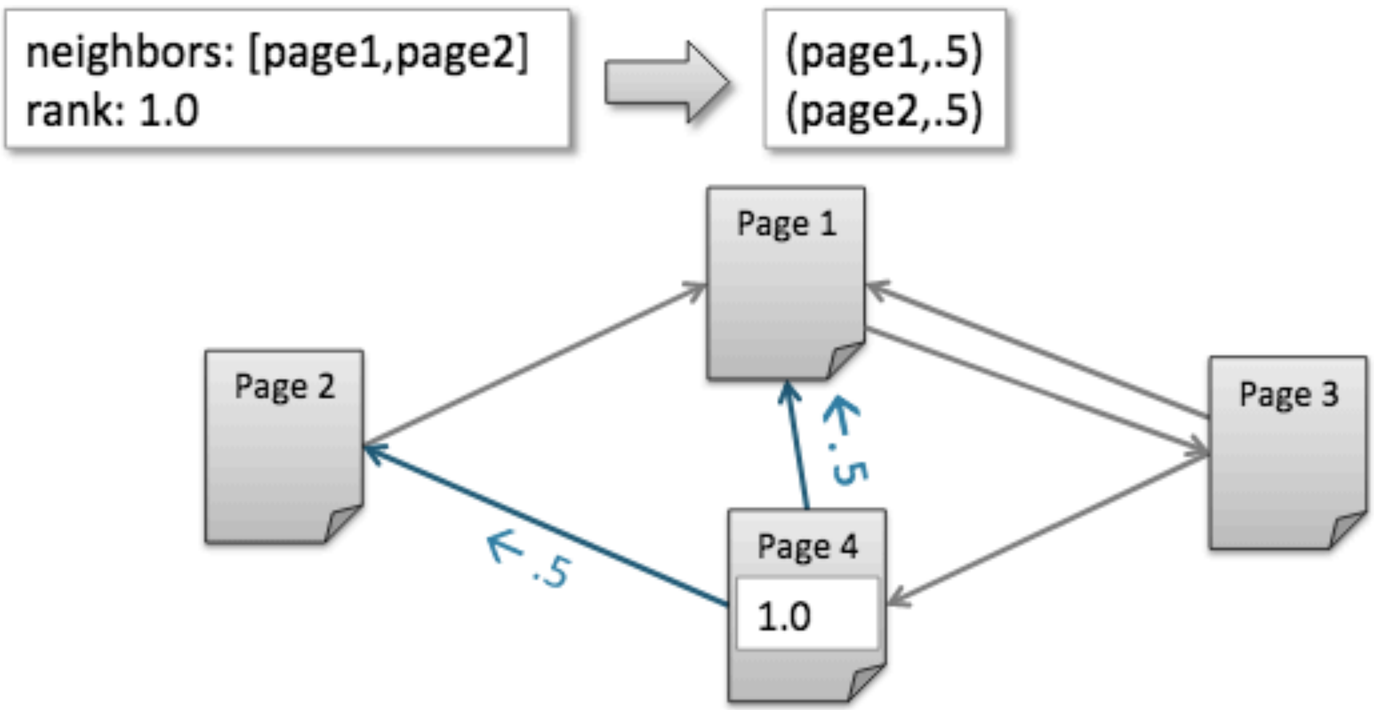
# PageRank algorithm

1. **Start each page with a rank of 1.0**
2. **On each iteration:**
   1. each page contributes to its neighbors its own rank divided by the number of its neighbors: $contrib_p = rank_p / neighbors_p$
   2. Set each page's new rank based on the sum of its neighbors contribution: $new\text{-}rank = \Sigma contribs * .85 + .15$
3. **Each iteration incrementally improves the page ranking**



# Neighbor contribution function

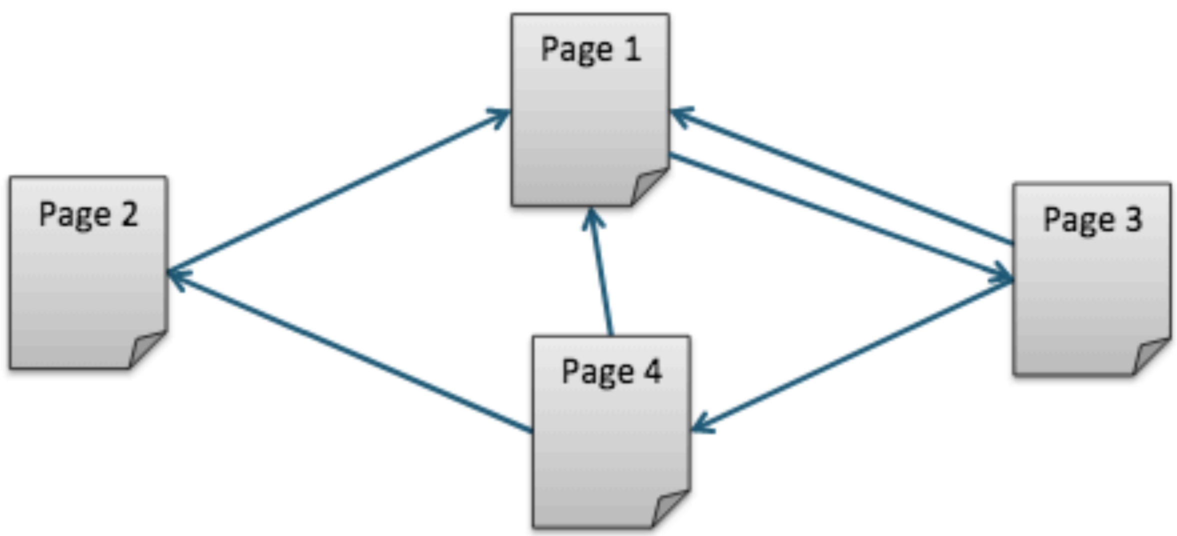
```
def computeContribs(neighbors, rank):
    for neighbor in neighbors: yield(neighbor, rank/len(neighbors))
```



# Input data

Data Format:
***source-page destination-page***
...

```
page1 page3
page2 page1
page4 page1
page3 page1
page4 page2
page3 page4
```



# Pairs of page links

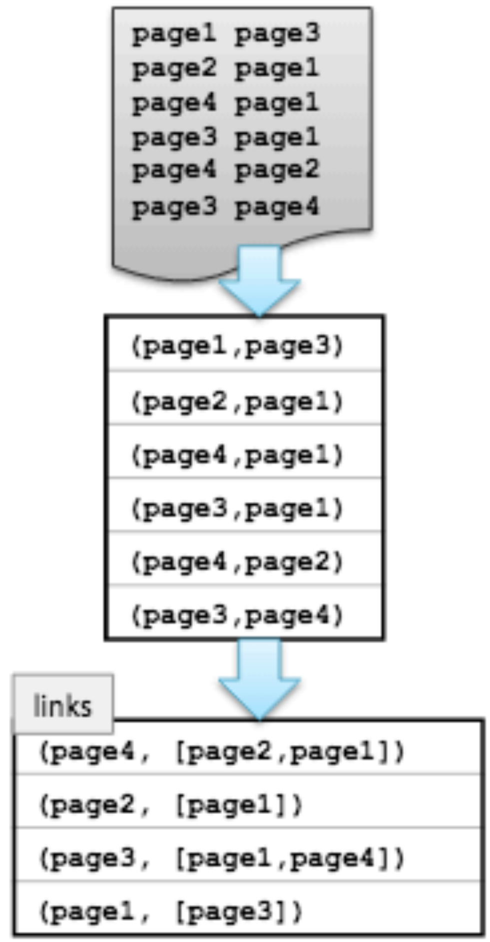
```
def computeContribs(neighbors, rank):…

links = sc.textFile(file)\
    .map(lambda line: line.split())\
    .map(lambda pages: (pages[0],pages[1]))\
    .distinct()
```

```
page1 page3
page2 page1
page4 page1
page3 page1
page4 page2
page3 page4
```

| |
|---|
| (page1,page3) |
| (page2,page1) |
| (page4,page1) |
| (page3,page1) |
| (page4,page2) |
| (page3,page4) |

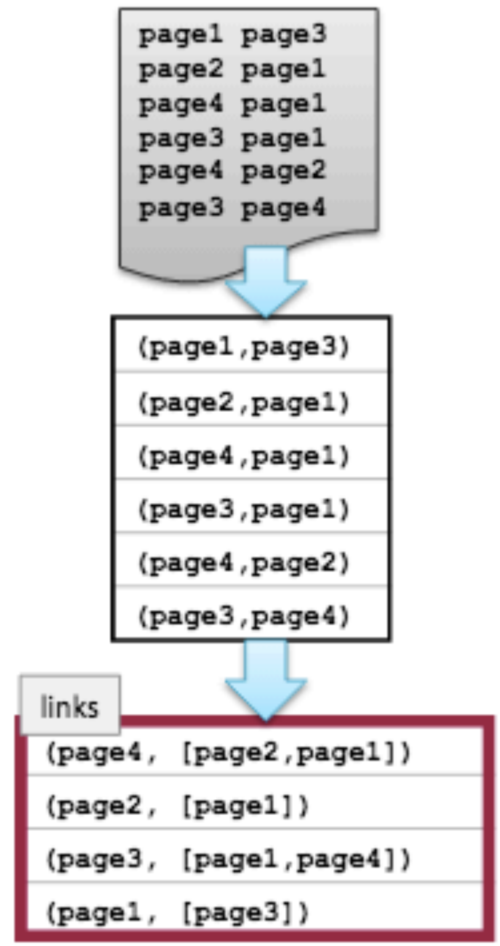
# Page links grouped by source page

```
def computeContribs(neighbors, rank):…

links = sc.textFile(file)\
    .map(lambda line: line.split())\
    .map(lambda pages: (pages[0],pages[1]))\
    .distinct()\
    .groupByKey()
```



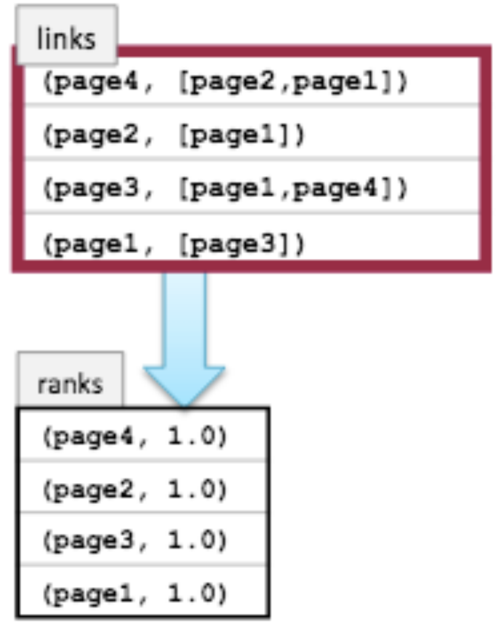
# Persisting the link pair RDD

```
def computeContribs(neighbors, rank):…

links = sc.textFile(file)\
    .map(lambda line: line.split())\
    .map(lambda pages: (pages[0],pages[1]))\
    .distinct()\
    .groupByKey()\
    .persist()
```



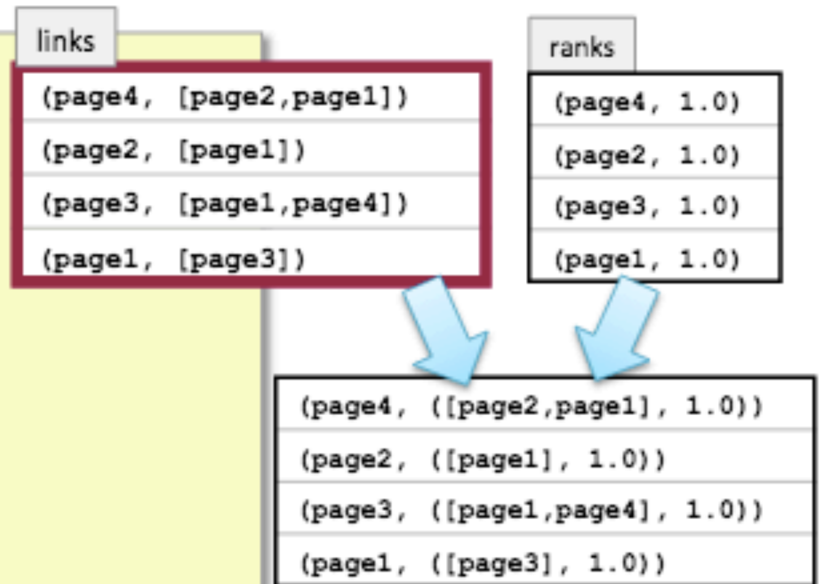
# Set initial ranks

```
def computeContribs(neighbors, rank):…

links = sc.textFile(file)\
    .map(lambda line: line.split())\
    .map(lambda pages: (pages[0],pages[1]))\
    .distinct()\
    .groupByKey()\
    .persist()

ranks=links.map(lambda (page,neighbors): (page,1.0))
```



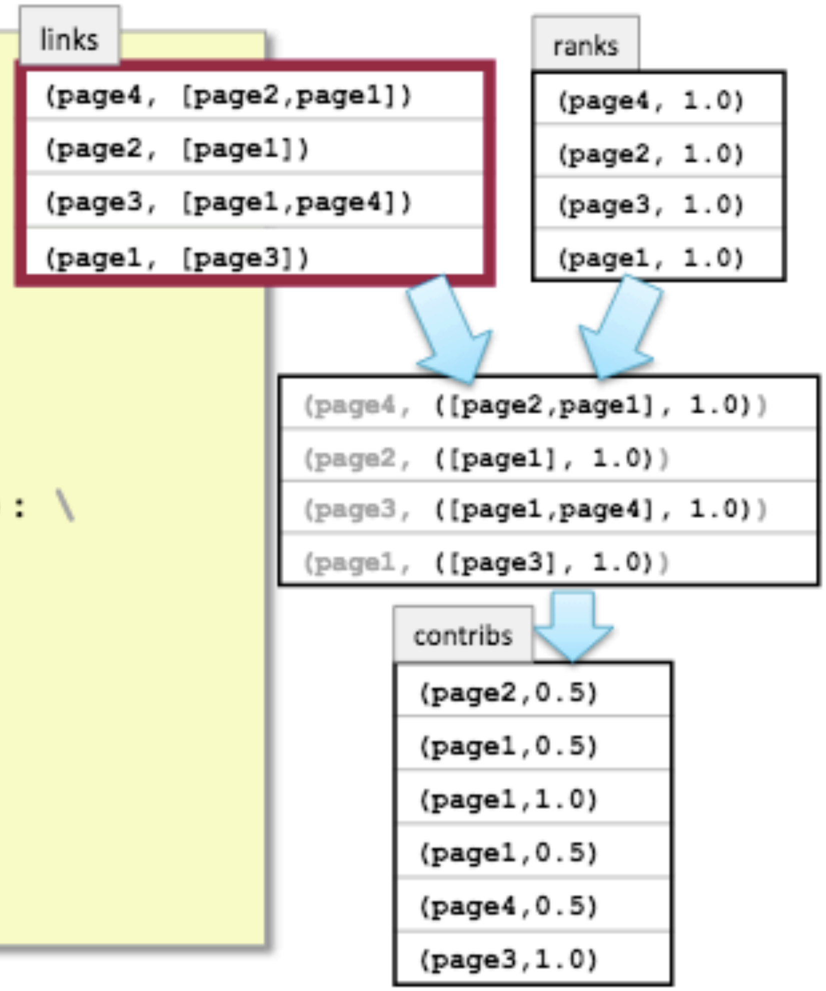
# First iteration

```
def computeContribs(neighbors, rank):...

links = ...

ranks = ...

for x in xrange(10):
  contribs=links\
    .join(ranks)
```



# First iteration

```
def computeContribs(neighbors, rank):...

links = ...

ranks = ...

for x in xrange(10):
  contribs=links\
    .join(ranks)\
    .flatMap(lambda (page,(neighbors,rank)): \
       computeContribs(neighbors,rank))
```



# First iteration

```
def computeContribs(neighbors, rank):…

links = …

ranks = …

for x in xrange(10):
  contribs=links\
    .join(ranks)\
    .flatMap(lambda (page,(neighbors,rank)): \
       computeContribs(neighbors,rank))
  ranks=contribs\
    .reduceByKey(lambda v1,v2: v1+v2)
```

| contribs |
|---|
| (page2,0.5) |
| (page1,0.5) |
| (page1,1.0) |
| (page1,0.5) |
| (page4,0.5) |
| (page3,1.0) |

| |
|---|
| (page4,0.5) |
| (page2,0.5) |
| (page3,1.0) |
| (page1,2.0) |

# First iteration

```
def computeContribs(neighbors, rank):…

links = …

ranks = …

for x in xrange(10):
  contribs=links\
    .join(ranks)\
    .flatMap(lambda (page,(neighbors,rank)): \
       computeContribs(neighbors,rank))
  ranks=contribs\
    .reduceByKey(lambda v1,v2: v1+v2)\
    .map(lambda (page,contrib): \
         (page,contrib * 0.85 + 0.15))
```

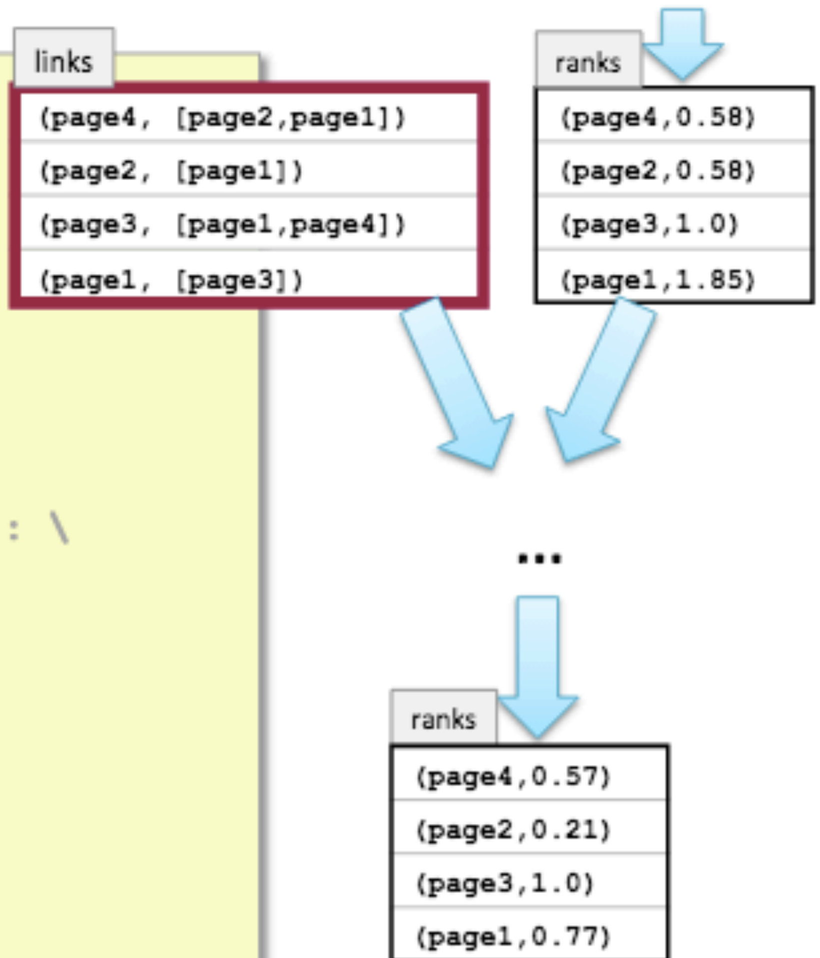
| contribs |
|---|
| (page2,0.5) |
| (page1,0.5) |
| (page1,1.0) |
| (page1,0.5) |
| (page4,0.5) |
| (page3,1.0) |

| |
|---|
| (page4,0.5) |
| (page2,0.5) |
| (page3,1.0) |
| (page1,2.0) |

| ranks |
|---|
| (page4,.58) |
| (page2,.58) |
| (page3,1.0) |
| (page1,1.85) |

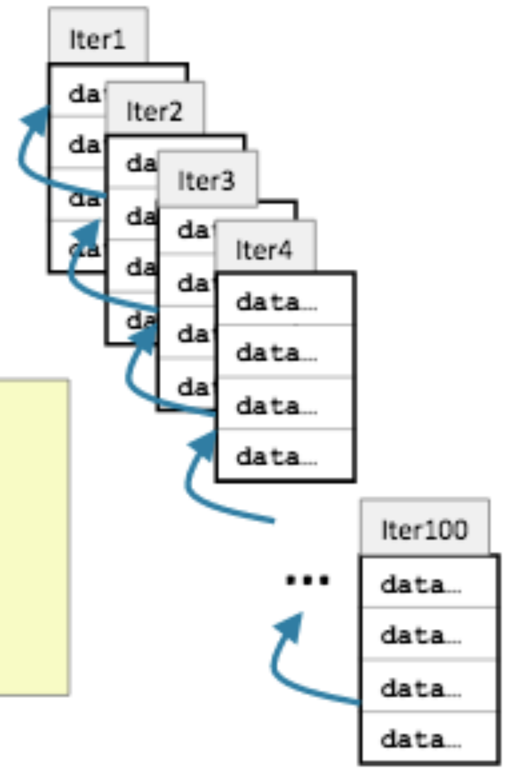
# Second iteration

```
def computeContribs(neighbors, rank):…

links = …

ranks = …

for x in xrange(10):
  contribs=links\
    .join(ranks)\
    .flatMap(lambda (page,(neighbors,rank)): \
       computeContribs(neighbors,rank))
  ranks=contribs\
    .reduceByKey(lambda v1,v2: v1+v2)\
    .map(lambda (page,contrib): \
         (page,contrib * 0.85 + 0.15))

for rank in ranks.collect(): print rank
```



# Checking point

- **Maintaining RDD lineage provides resilience but can also cause problems when the lineage gets very long**
  - e.g., iterative algorithms, streaming
- **Recovery can be very expensive**
- **Potential stack overflow**

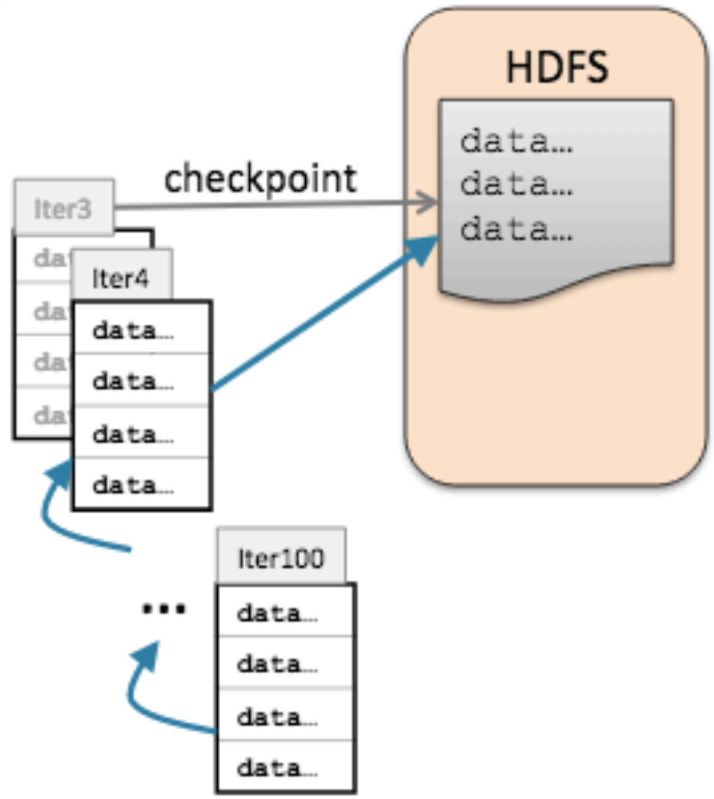



```
myrdd = ...initial-value...
while x in xrange(100):
    myrdd = myrdd.transform(...)
myrdd.saveAsTextFile(dir)
```

# Checking point

- **Checkpointing saves the data to HDFS**
  - Provides fault-tolerant storage across nodes
- **Lineage is not saved**
- **Must be checkpointed before any actions on the RDD**

```
sc.setCheckpointDir(directory)
myrdd = ...initial-value...
while x in xrange(100):
  myrdd = myrdd.transform(...)
  if x % 3 == 0:
    myrdd.checkpoint()
    myrdd.count()
myrdd.saveAsTextFile(dir)
```



# GraphX in Spark

- **Spark is very well suited to graph parallel algorithms**
- **GraphX**
  - UC Berkeley AMPLab project on top of Spark
  - Unifies optimized graph computation with Spark's fast data parallelism and interactive abilities
  - Supersedes predecessor Bagel (Pregel on Spark)

# Examples in GraphX

```
import org.apache.spark.graphx.GraphLoader

// Load the edges as a graph
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Run PageRank
val ranks = graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}
// Print the result
println(ranksByUsername.collect().mkString("\n"))
```

```
import org.apache.spark.graphx.GraphLoader

// Load the graph as in the PageRank example
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Find the connected components
val cc = graph.connectedComponents().vertices
// Join the connected components with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ccByUsername = users.join(cc).map {
  case (id, (username, cc)) => (username, cc)
}
// Print the result
println(ccByUsername.collect().mkString("\n"))
```

# MLlib in Spark

https://spark.apache.org/docs/2.0.2/ml-guide.html

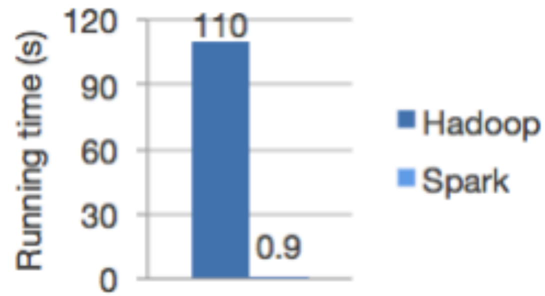
# What is MLlib?

## Algorithms:

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

# Why MLlib?

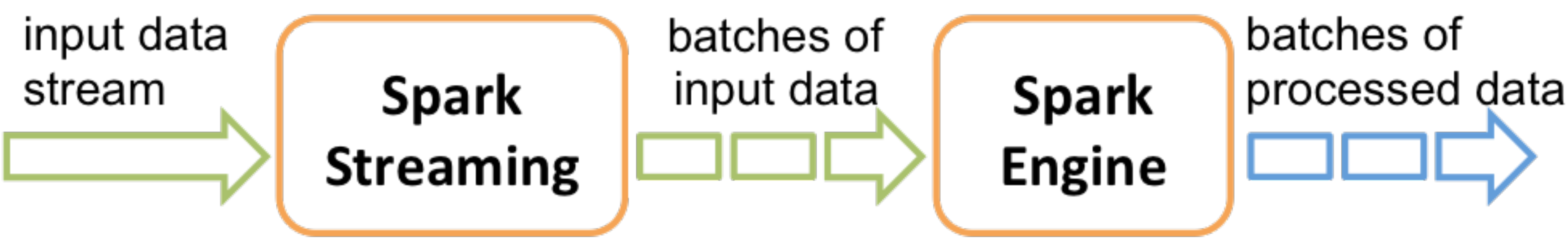
- It is built on Apache Spark, a fast and general engine for large-scale data processing.
  - Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.




  - Write applications quickly in Java, Scala, or Python.

https://docs.databricks.com/spark/latest/mllib/decision-trees.html

# Spark streaming


http://spark.apache.org/docs/latest/streaming-programming-guide.html