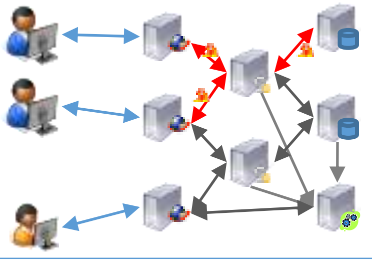


Verteilte Systeme/ Distributed Systems

Artur Andrzejak

5

Apache Spark:
DataFrames and Datasets

DataFrames in Spark

- **DataFrames** (DF) are tables with named and typed data columns
  - Similar to a dataframe in R, or Pandas (Python), or tables in DBMS/SQL
  - Impose a structure and schema on data
- Example

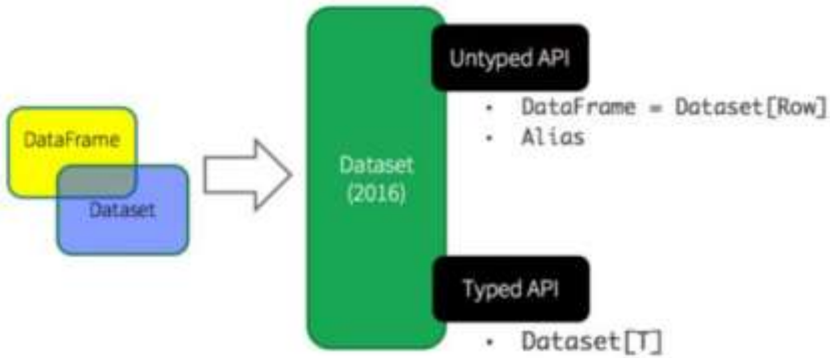
	Partition 1			Partition 2			Partition 3			Partition 4		
	Time (Str)	Site (Str)	Req (Int)	Time (Str)	Site (Str)	Req (Int)	Time (Str)	Site (Str)	Req (Int)	Time (Str)	Site (Str)	Req (Int)
Row 1	ts	m	1304	ts	d	3901	ts	m	1172	ts	m	2538
Row 2	ts	d	2237	ts	d	2491	ts	m	2137	ts	d	2837
Row 3	ts	m	1600	ts	d	2288	ts	d	3176	ts	d	3400

From: Databricks, 7 Steps for a Developer to Learn Apache Spark, p. 14, 2017

From: Databricks, 7 Steps for a Developer to Learn Apache Spark, 2017

DataFrames and Datasets in 2016

- Spark 2.0 unified data structures: DF became a specialized **Dataset**
- High-level DSL-like APIs for DFs and DS introduced



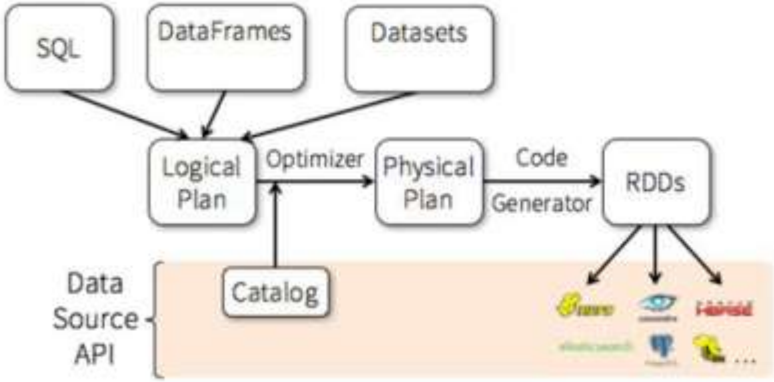
- Dataset is a strongly typed collection of *anything* T
  - APIs only for Scala and Java, not for Python
- DataFrame a is a collection of Row-objects
  - DataFrame = Dataset[Row]

	SQL	DataFrame	Dataset
Syntax errors	Runtime	Compile Time	Compile Time
Analysis errors	Runtime	Runtime	Compile Time

From: Databricks, 7 Steps for a Developer to Learn Apache Spark, 2017

API ~ Domain Specific Language

- APIs for DFs and DSs provide high-level operators like sum, count, avg, min, max etc.
- Highly-efficient code generated (faster than for RDDs!)
- Example (in Scala):




```
// a dataset with field names fname, lname, age, weight
// access using object notation
val seniorDS = peopleDS.filter( p => p.age > 55 )
// a dataframe with named columns fname, lname, age, weight
// access using col name notation
val seniorDF = peopleDF.where( peopleDF(“age”) > 55)
// equivalent Spark SQL code
val seniorDF = spark.sql(“SELECT age from person where age > 35”)
```

Creating DataFrames in PySpark

- A DF can be created from multiple sources …
  - By converting „normal" RDDs
  - Loading from text, csv, json, xml, parquet files
  - Importing from DBMS (Hive, Cassandra, ..)
- Each DF has a **schema**: definition of names and types of columns
  - Schema can be set programmatically, or inferred from data

DataFrames from RDDs

```
// Read file with rows: <name, age>
filePath = „/home/immd-user/spark-2 …/examples/src/main/resources/people.txt“
parts = sc.textFile(filePath).map( lambda line: line.split(“,“) )
// Each row should become a tuple (name, age)
peopleRDD = parts.map( lambda p: (p[0], p[1].strip() ) )

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName(„Exmpl").getOrCreate()
schema = StructType( [
                    StructField (“name“,  StringType(), True),
                    StructField ( „age“,    StringType(), True) ] )

dfPeople = spark.createDataFrame (peopleRDD, schema)
print ( dfPeople.take(5) )
```

Read DFs From Other Sources

filePath = „/home/immd-user/spark-2 …/examples/src/main/resources/people.txt“

```
// Read from CSV (comma separated values) files
df_csv = spark.read.csv(filePath, schema = schema)
print (df_csv.take(5) )

// Read from json – schema is inferred!
df_json =
spark.read.json("examples/src/main/resources/people.json")

df_json.show()
    # +----+-------+
    # | age|   name|
    # +----+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +----+-------+
```

DFs Operations

```
# Print schema
dfPeople.printSchema()
    # root
    # |-- age: string (nullable = true)
    # |-- name: string (nullable = true)
# Select only the "name" column
dfPeople.select("name").show()
    # |    name|
    # |Michael| …
# Group by age and count per group
dfPeople. groupBy("age").count().show()
    # | age|count|
    # |  19|    1|
    # |null|    1|
    # |  30|    1|
```

SQL: More User-Friendly

Standard DF API:

```
# Select people older than 25
dfPeople.filter(
        dfPeople['age'] > 25)
        .show()
  # |age|name|
  # | 30|Andy|
```

Same result with **SQL**:

```
# Register the DataFrame as a SQL temporary view
dfPeople
  .createOrReplaceTempView
   ("people")

sqlDF = spark.sql("SELECT * FROM people where age > 25")
sqlDF.show()
```

User Defined Functions (UDFs)

```
from pyspark.sql.functions import *
from pyspark.sql.types import *
df = sqlContext.read.parquet('hdfs:///.../stations.parquet')

lat2dir =  udf(lambda x: 'N' if x > 0 else 'S',        StringType())
lon2dir = udf(lambda x: 'E' if x > 0 else 'W',        StringType())

df.select(df.lat, lat2dir(df.lat).alias('latdir'),
        df.long, lon2dir(df.long).alias('longdir')).show()
```

```
|     lat          |latdir|     long     |longdir|
|37.329732         |     N|-121.901782|       W|
|37.330698         |     N|-121.888979|       W|
|   ...            |  ... |    ...         |  ...    |
```

From: Jeffrey Aven: Sams teach yourself Apache Spark in 24 hours, 2017

Standard DFs Functions

There are many ready-to-use functions, similar to those in SQL

Type	Sample Available Functions
String functions	startswith, substr, concat, lower, upper, regexp_extract, regexp_replace
Math functions	abs, ceil, floor, log, round, sqrt
Statistical functions	avg, max, min, mean, stddev
Date functions	date_add, datediff, from_utc_timestamp
Hashing functions	md5, sha1, sha2
Algorithmic functions	soundex, levenshtein
Windowing functions	over, rank, dense_rank, lead, lag, ntile

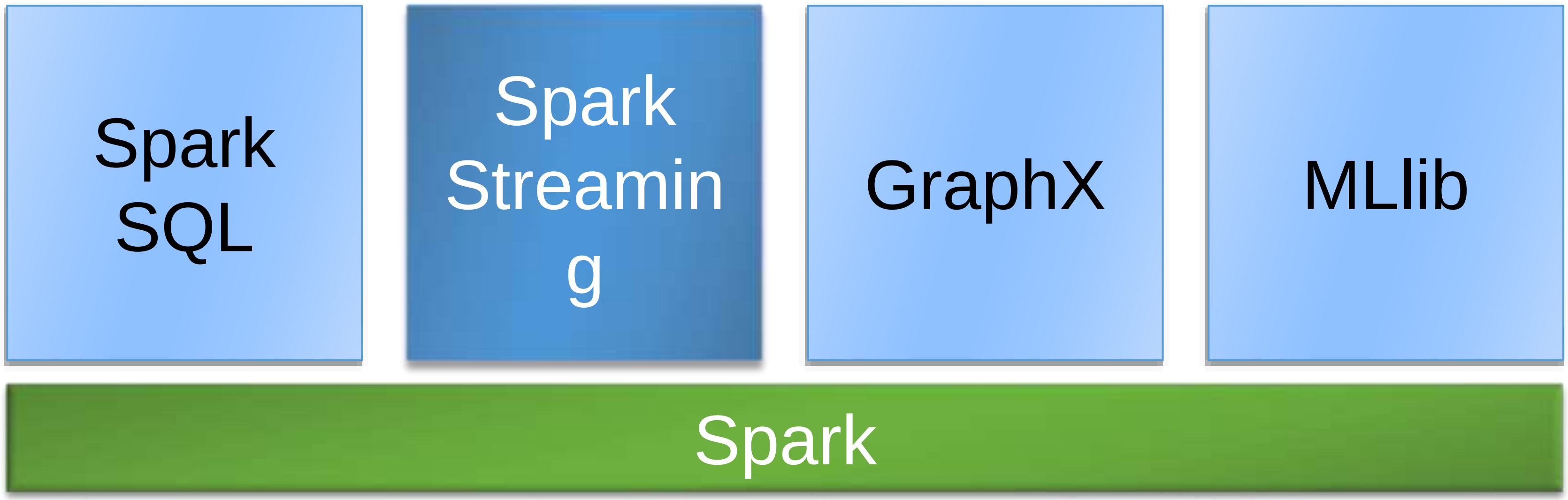
From: Jeffrey Aven: Sams teach yourself Apache Spark in 24 hours, 2017

More Resources on DataFrames

- Jeffrey Aven: Sams teach yourself Apache Spark in 24 hours, 2017, http://katalog.ub.uni-heidelberg.de/cgi-bin/titel.cgi?katkey=68102164 (free from univ. domain!)
- Databricks, 7 Steps for a Developer to Learn Apache Spark, 2017, https://goo.gl/chn8GE
- Spark Documentation: Spark SQL, DataFrames and Datasets Guide, https://goo.gl/HuHNEq
- Ankit Gupta: Complete Guide on DataFrame Operations in PySpark, 2016, https://goo.gl/mrNL4i
- Michael Armbrust, Wenchen Fan, Reynold Xin and Matei Zaharia: Introducing Apache Spark Datasets, 2016, https://goo.gl/VLu9dn

Apache Spark:
Spark Streaming

What is Spark Streaming?



- Spark library / module: extends Spark for (large-scale) distributed data stream processing
- Started in 2012, included in 2014 in Spark 0.9
- Bindings in Spark version 1.2
  - Scala, Java, Python (partial)

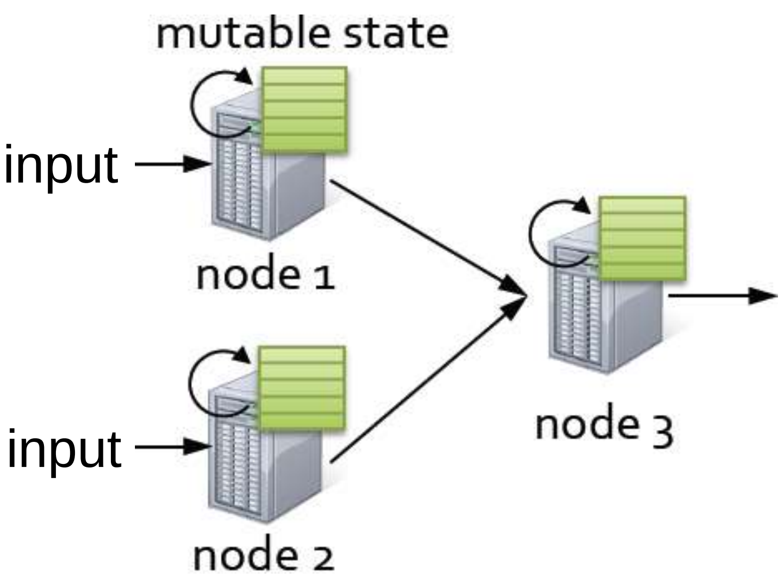
Problem 1: Stream vs. Batch Processing

- Many apps require processing <u>the same data</u> in live streaming as well as in batches
  - E.g. finance: trading robots / high freq. trading
    - Batch: testing and evaluating trading systems (backtests)
    - Stream: live trading using prepared systems
  - Detecting DoS attacks
    - Batch: understand patterns of DoS, tune algorithms
    - Stream: apply prepared algorithms to live data
- => Need for two separate programming models
  - Doubled effort, inconsistency, hard to debug

Problem 2: Fault-tolerance

- Traditional processing model:
  - Pipeline of nodes
  - Each node maintain a **mutable state**
  - Each input record updates the state and new records are sent out

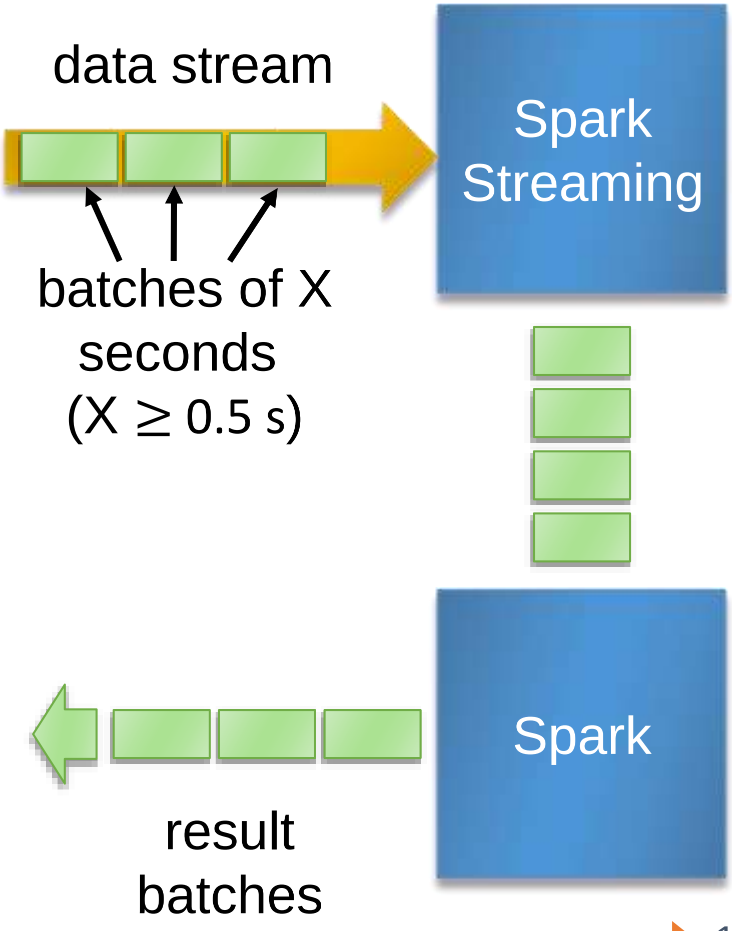
- => Mutable state is lost if node fails
- => Making stateful stream processing **fault-tolerant** is challenging



From: Tathagata Das, Spark Streaming, Spark Summit 2014

Spark Streaming: Concept

- Process stream as a **series of small batch jobs**
  - Chop up the live stream into batches of X sec
- Spark treats each batch of data as an RDD and processes them using (normal) RDD op's
- The results of the RDD operations are returned in batches



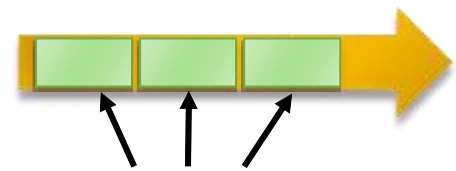
Data Sources and Sinks

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

- Input data streams can come from many sources, e.g.:
  - HDFS/S3 (files), TCP sockets, Kafka, Flume, Twitter, …
- Output data can be pushed out to …
  - File systems, databases, live dashboards

Programming Model - **DStream**

- **DStream** = Discretized Stream
  - "Container" for a stream
  - Implemented as a **sequence of RDDs**

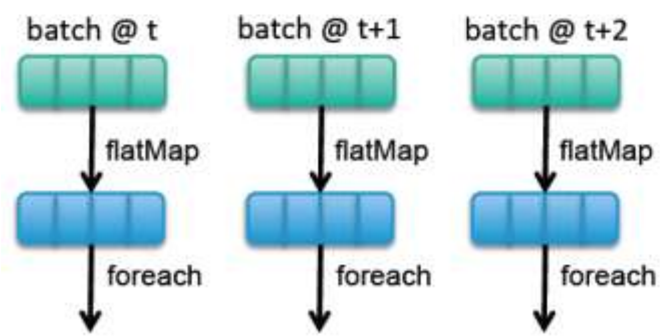


Each chunk = Resilient Distributed Dataset (RDD)

- DStreams can be …
  - Created from "raw" input streams
  - Obtained by transforming existing DStreams

Dstream A

Dstream A'



…

From: Tathagata Das, Spark Streaming, Spark Summit 2014

Example: (Stream) Word Count

- Goal: We want to count the occurrences of each word <u>in each batch</u> a text stream
  - Data received from a TCP socket 9999, each "event" (= record) is a <u>line of text</u>
  - Stream is split into RDDs, each 1 second "length"
    - Each RDD can have 0 or more records!
  - Output: first ten elements of each RDD

- **Program structure**
  - 1. Set up the processing "pipeline"
  - 2. Start the computation and specify termination

Word Count: Pipeline Setup /1

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
```

Use <u>two threads</u>: 1 for source feed, 1 for processing

```
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
```

Set <u>batch interval</u> to 1 second

```
lines = ssc.socketTextStream("localhost", 9999)
```

Create a DStream that will connect to hostname:port, like localhost:9999

Word Count: Pipeline Setup /2

- Since each "event" in a DStream is a "normal" RDD-record, we can process it with Spark operations
  - Here: each record is a line of text

New DStream (and new RDD for each batch)

Split each line into words

```
words = lines.flatMap( lambda line: line.split(" ") )
pairs = words.map( lambda word: (word, 1) )
wordCounts = pairs.reduceByKey( lambda x, y: x + y )
```

Count each word in each batch

```
wordCounts.pprint()
```

Print the first ten elements of each RDD generated in this DStream to the console

Word Count: Start & End

ssc.start() — Start the computation

ssc.awaitTermination() — Wait to terminate

Terminal 1

- Netcat (link) utility can redirect std input to a TCP port (here: 9999)
- **nc -lk 9999**
- <type anything…>
- Hello IMMD

Terminal 2

- ./bin/spark-submit network_wordcount.py localhost 9999
- ---
- Time: 2015-01-08 13:22:51
- ---
- (hello,1)
- (IMMD,1)
- ...

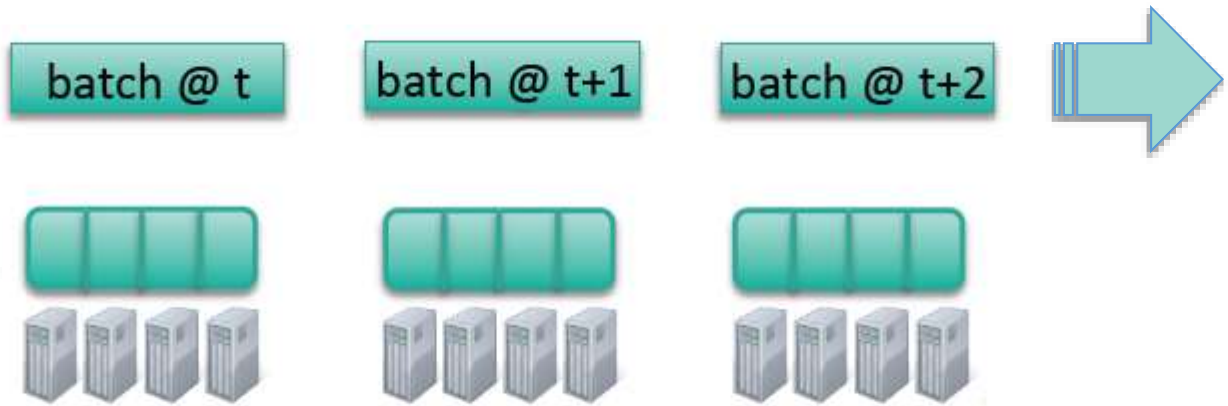
Example – Get hashtags from Twitter

Example in **Scala**

```
val ssc = new StreamingContext (sparkContext, Seconds(1))
val tweets = TwitterUtils.createStream (ssc, auth)
```

Twitter Streaming API

DStream tweets



RDDs, stored in memory

Get hashtags from Twitter /2

```
val ssc = new StreamingContext (sparkContext, Seconds(1))
val hashTags = tweets.flatMap(status => getTags(status))
```

Transformed DStream

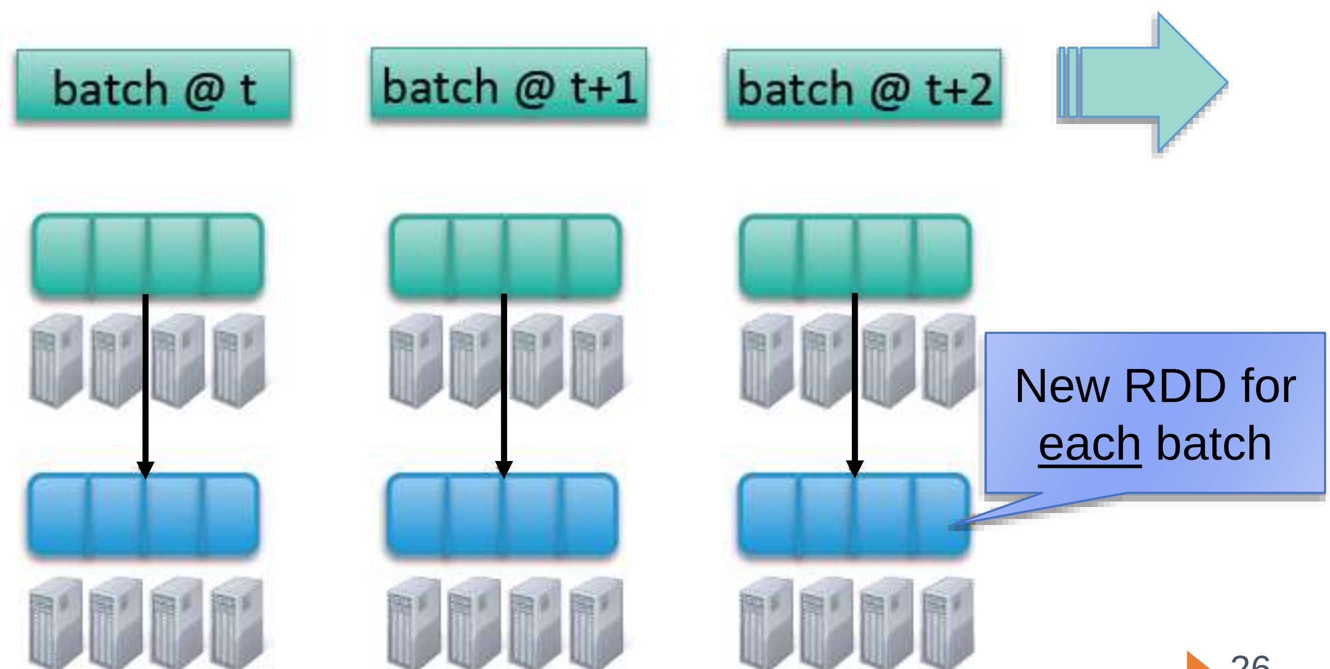
Spark flatMap

In Python:
lambda status: getTags(status)

Twitter Streaming API

DStream tweets

DStream hashTags
[#cat, #dog,…]

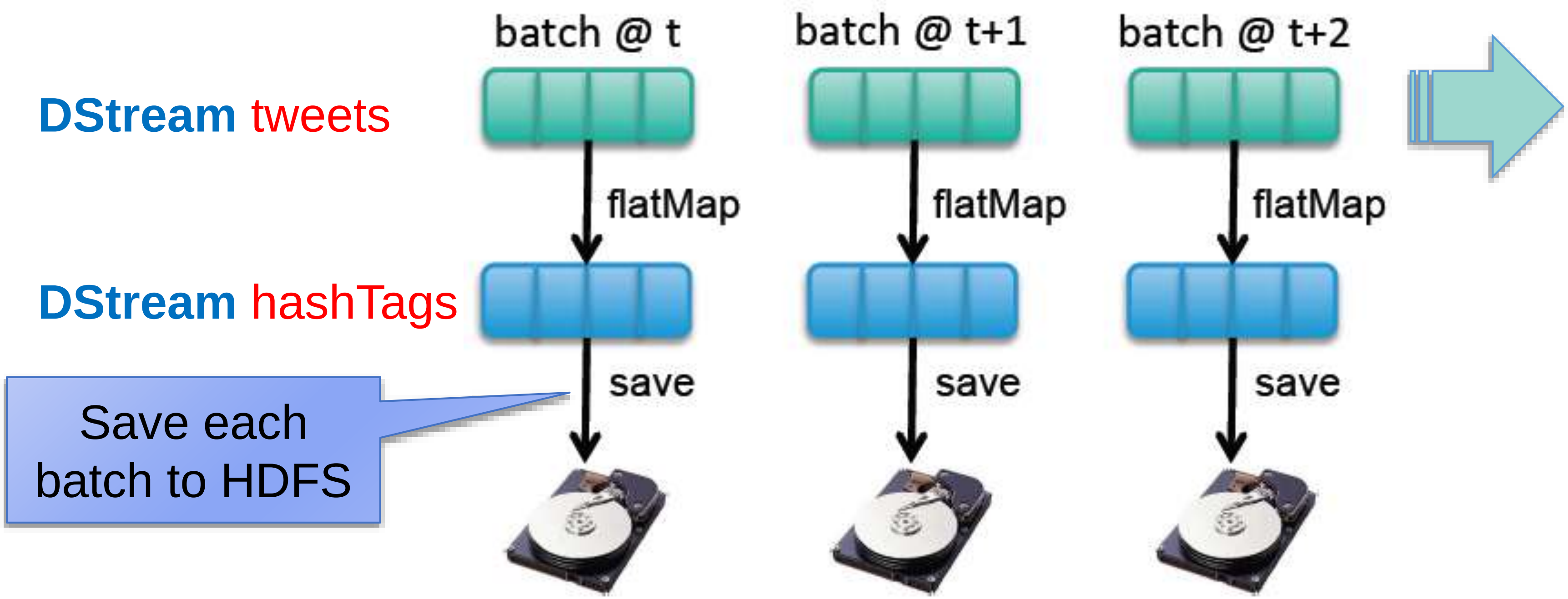


From: Tathagata Das, Spark Streaming, Spark Summit 2014

Get hashtags from Twitter /3

```
val ssc = new StreamingContext (sparkContext, Seconds(1))
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

Output: write to external storage

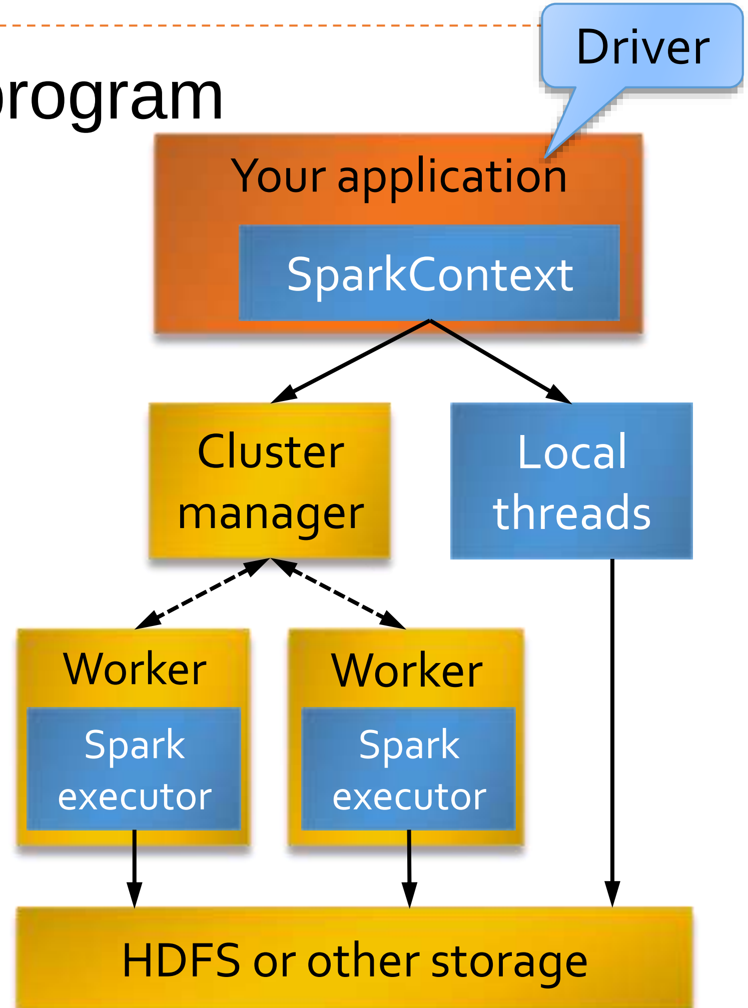


From: Tathagata Das, Spark Streaming, Spark Summit 2014

Spark: Execution Details

Software Components

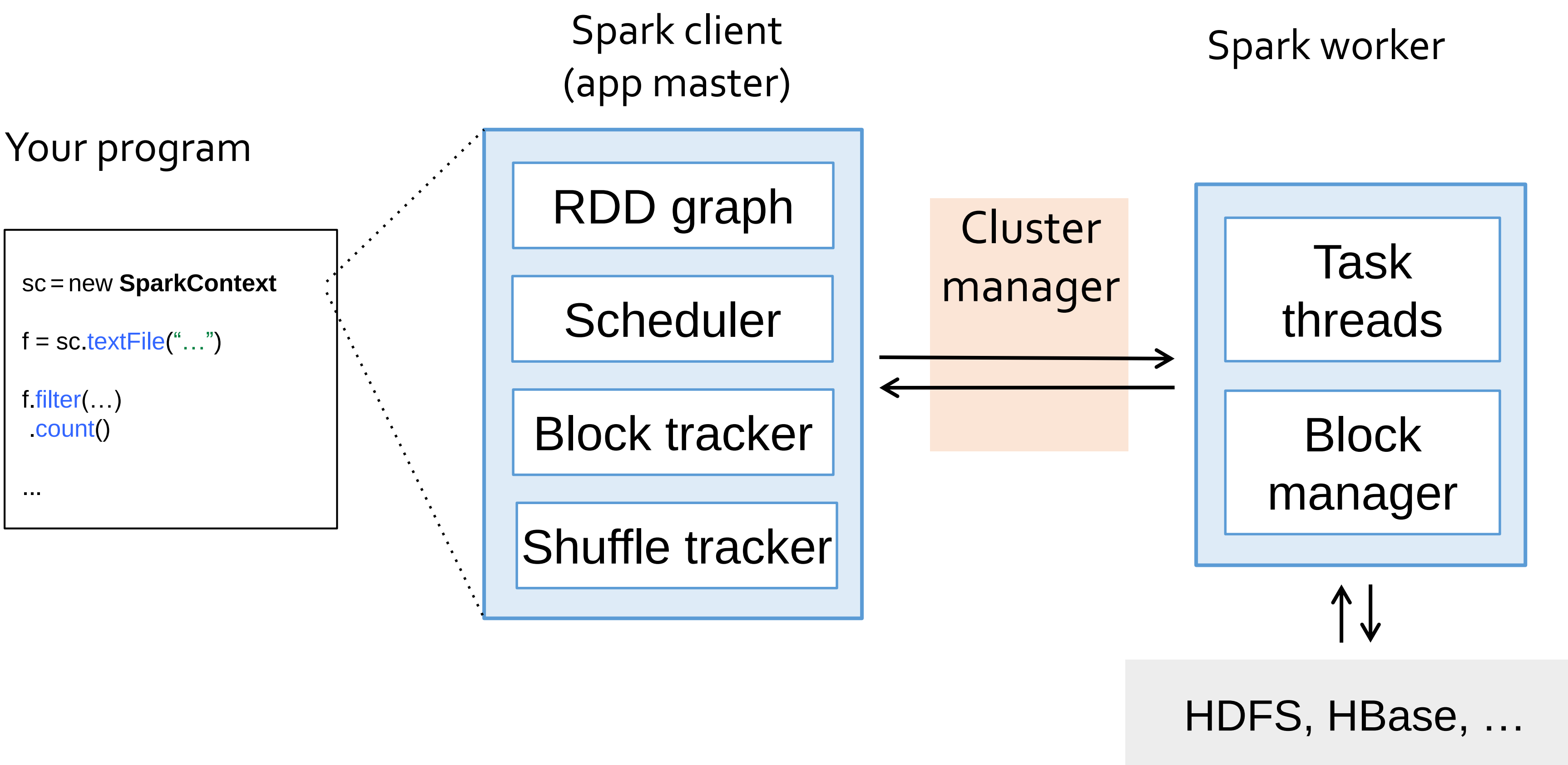
- Spark runs as a library in your program
  - Runs tasks locally / on cluster*
    - Standalone, YARN, Mesos
    - See Cluster Mode Overview*
- Accesses storage systems via Hadoop API
    - Can use HBase, HDFS, S3, …



*=http://spark.apache.org/docs/latest/cluster-overview.html

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Components



For more info see video "Introduction to AmpLab Spark Internals" (https://www.youtube.com/watch?v=49Hr5xZyTEA) and read slides http://files.meetup.com/3138542/dev-meetup-dec-2012.pptx

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Example Job

```
sc = SparkContext (appName="PythonExample")

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

errors = file.filter(lambda line:"ERROR" in line))

errors.cache()

errors.count()
```

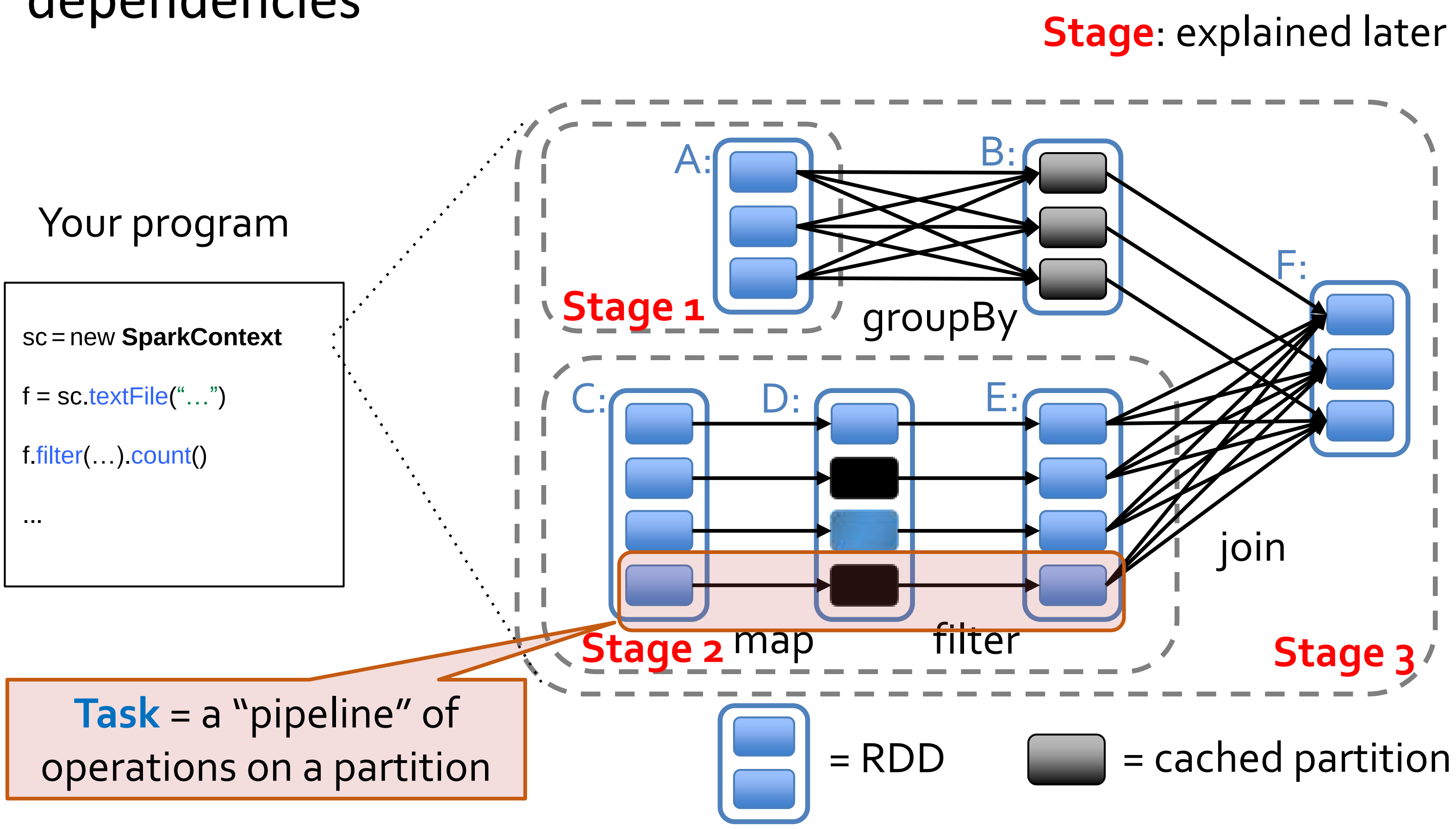
RDDs

Action

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Operator DAG

- The **operator DAG** (Directed Acyclic Graph) captures RDD dependencies

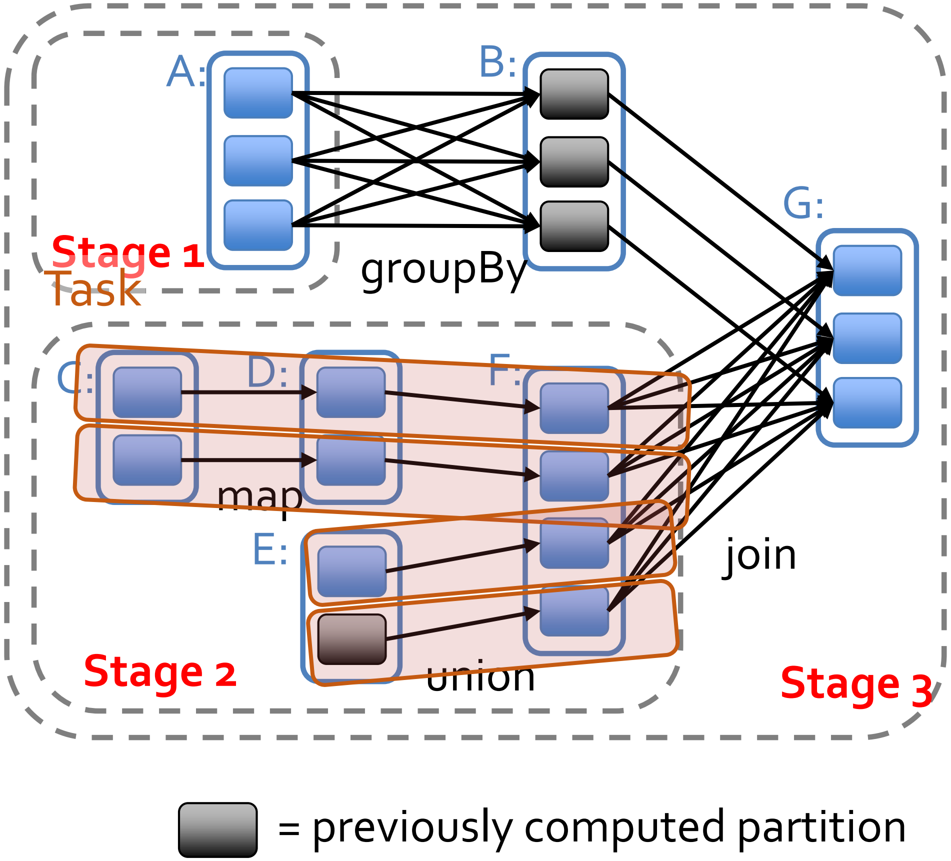


From: *Parallel Programming with Spark*, **Matei Zaharia**, AmpCamp 2013

Stages

- A set of independent tasks, as large as possible

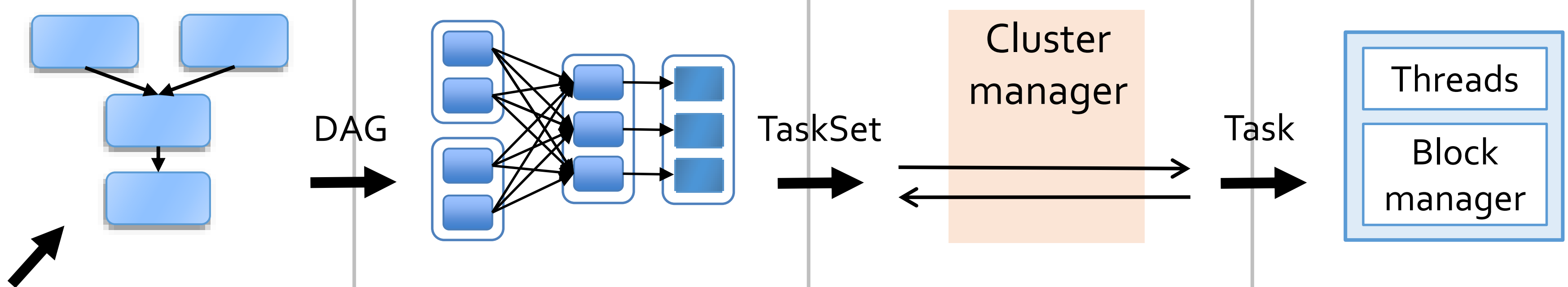
- Stage boundaries are at:
  - Input RDDs
  - “Shuffle”-like operations (e.g. groupBy*, join, ..)



From: *Introduction to Spark Internals,* **Matei Zaharia**, 2012

Scheduling Process

RDD Objects | **DAGScheduler** | **TaskScheduler** | Worker




```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
```

build **operator DAG** (DAG = Directed Acyclic Graph)

split graph into **stages** of tasks (stages ="maximal" sets of independent tasks)

stage failed

launch tasks via cluster manager

retry failed or straggling tasks

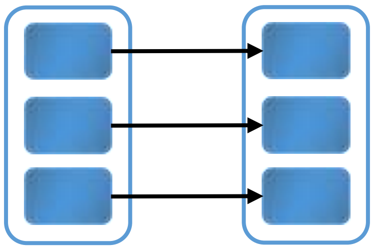
execute tasks

store and serve **blocks**

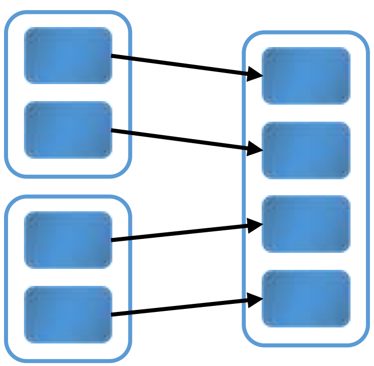
From: *Introduction to Spark Internals*, **Matei Zaharia**, Meetup Dec 2012

Dependency Types in DAG

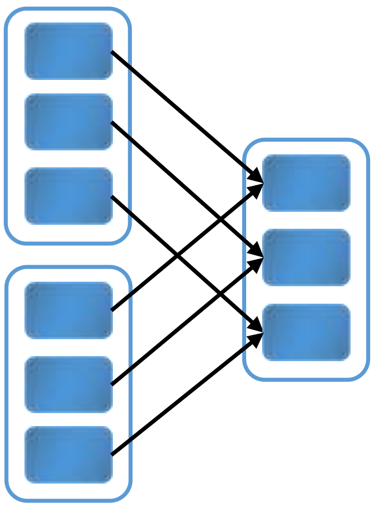
“Narrow” dependencies:



map, filter

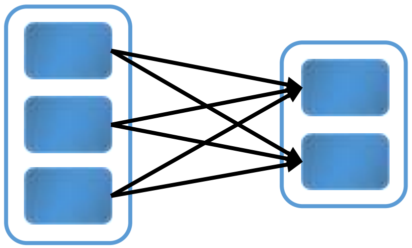


union

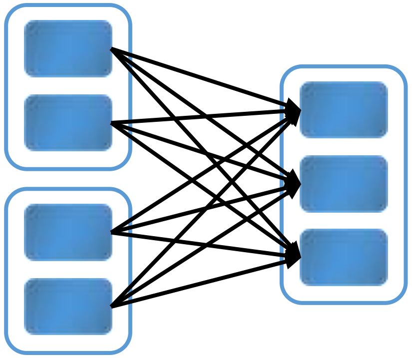


join with inputs co-partitioned

“Wide” (shuffle) deps:



groupByKey

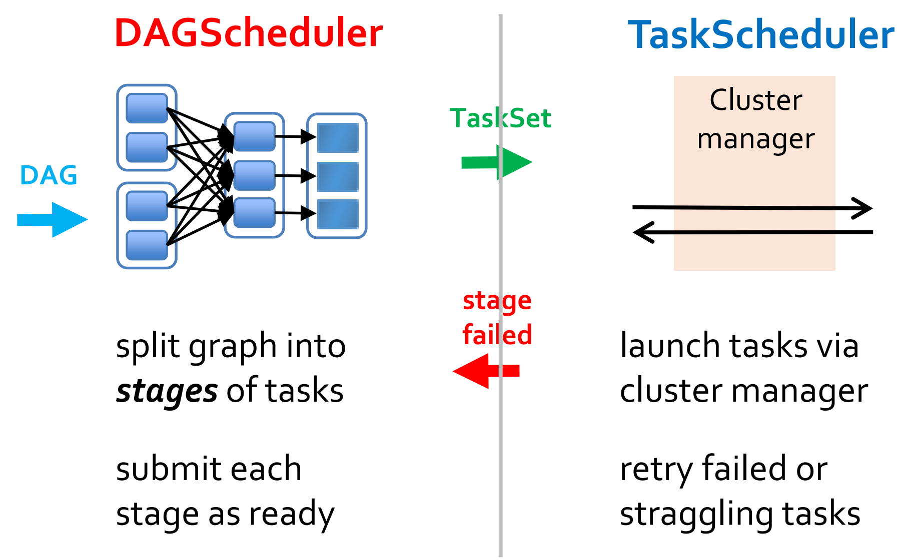


join with inputs not co-partitioned

From: *Introduction to Spark Internals*, **Matei Zaharia**, Meetup Dec 2012

DAG Scheduler vs. Task Scheduler

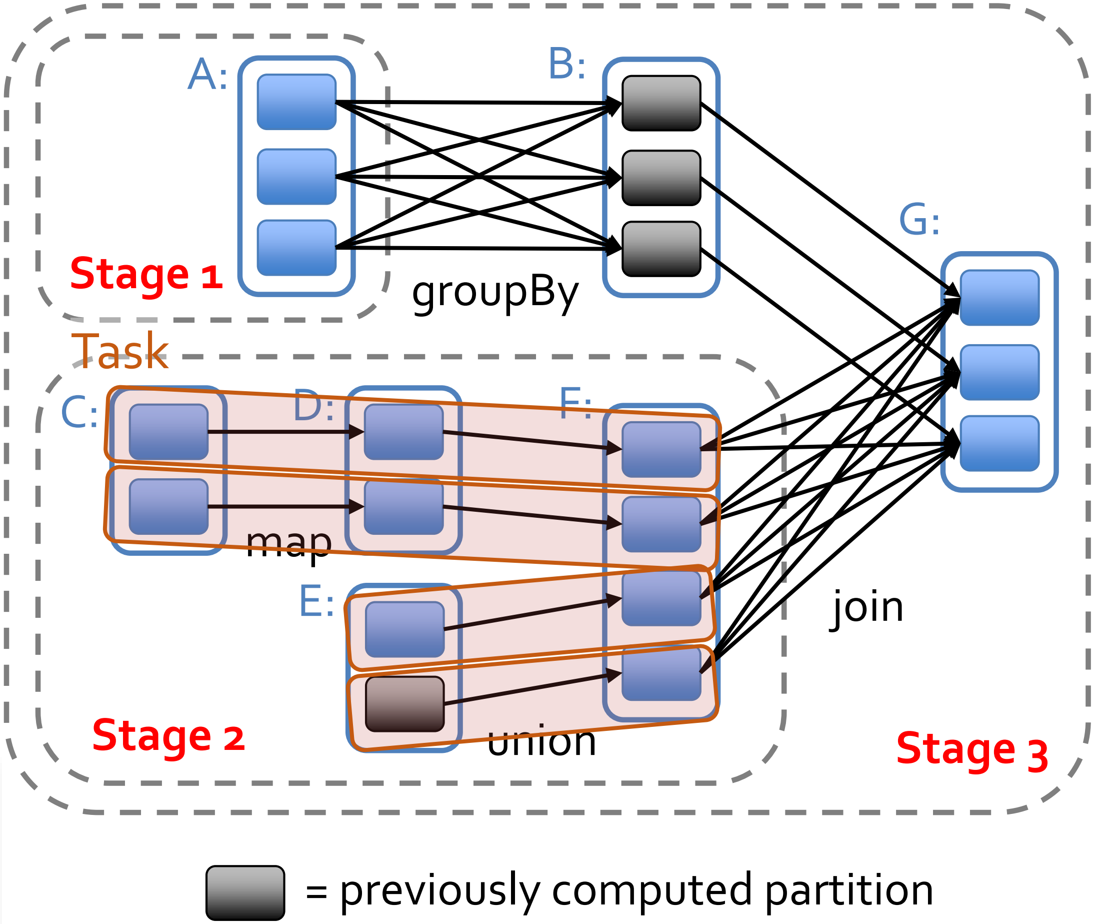
- DAG Scheduler – “higher level”
  - Builds **stages** of task objects (by code + preferred location)
  - Submits them to TaskScheduler as ready
  - Resubmits failed stages if outputs are lost
- TaskScheduler
  - “Lower level” – similar to Hadoop master
  - Given a set of tasks, runs it and reports results
  - Exploits data locality
  - Local / cluster implementation



From: *Introduction to Spark Internals*, **Matei Zaharia**, Meetup Dec 2012

Scheduler Optimizations

- Pipelines narrow ops. within a stage
- Picks join algorithms based on partitioning (minimize shuffles)
- Reuses previously cached data

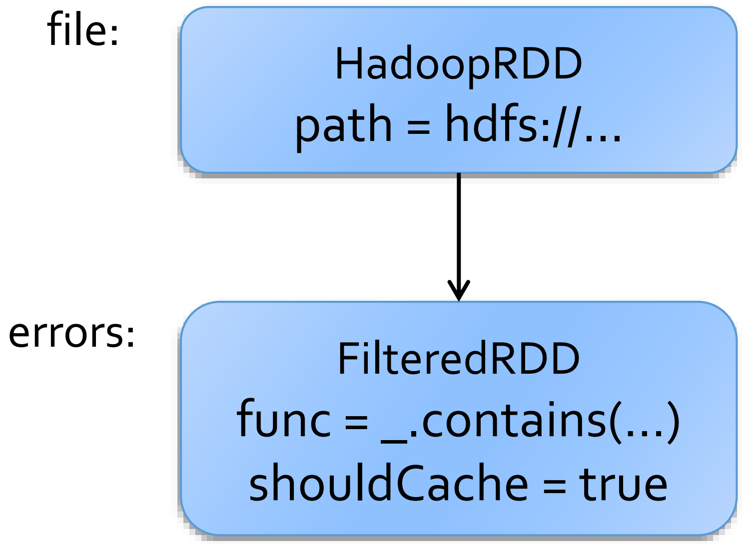


In MapReduce, each M-R phase is "individual" => Less optimization!

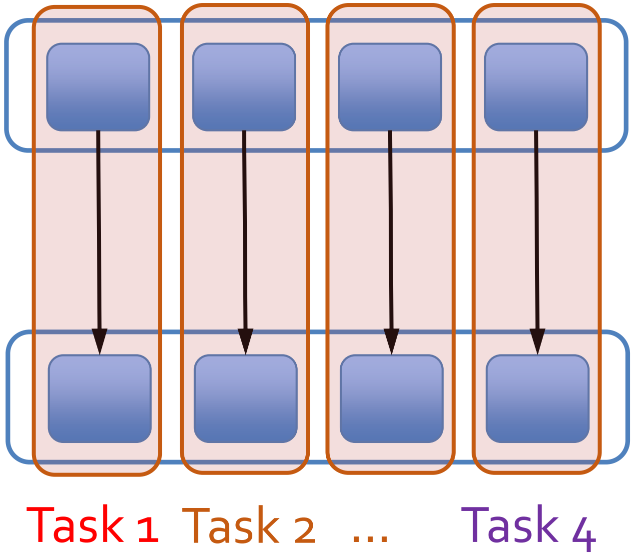
From: *Introduction to Spark Internals*, **Matei Zaharia**, 2012

RDD Graph

Dataset-level view:



Partition-level view:



- **Partition**: a subset of RDD, usually corresponding to a block of HDFS (or other file system)
- **Task**: a “pipeline” of operations on a single partition

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

RDD Interface

- Set of *partitions* ("splits")
- List of *dependencies* on parent RDDs
- Function to *compute* a partition given parents
- Optional *preferred locations*
- Optional *partitioning info* (Partitioner)

Captures all current Spark operations!

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Example: HadoopRDD

- partitions = one per HDFS block
- dependencies = none
- compute*(partition)* = read corresponding block

- preferredLocations*(part)* = HDFS block location
- partitioner = none

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Example: JoinedRDD

- partitions = one per reduce task
- dependencies = “shuffle” on each parent
- compute*(partition)* = read and join shuffled data

- preferredLocations*(part)* = none
- partitioner = HashPartitioner(numTasks)

Spark will now know this data is hashed!

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Thank you.

Additional Slides:
K-Means Clustering

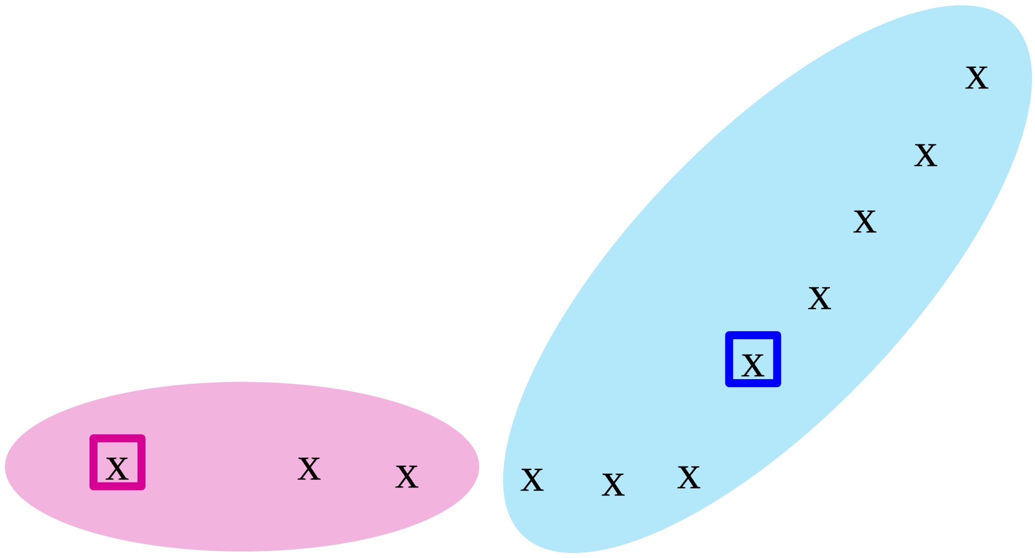
k–means Algorithm(s)

- Assumes Euclidean space/distance
- Start by picking ***k***, the number of clusters
- Initialize clusters by picking one point per cluster
  - **Example:** Pick one point at random, then ***k*-1** other points, each as far away as possible from the previous points

Populating Clusters

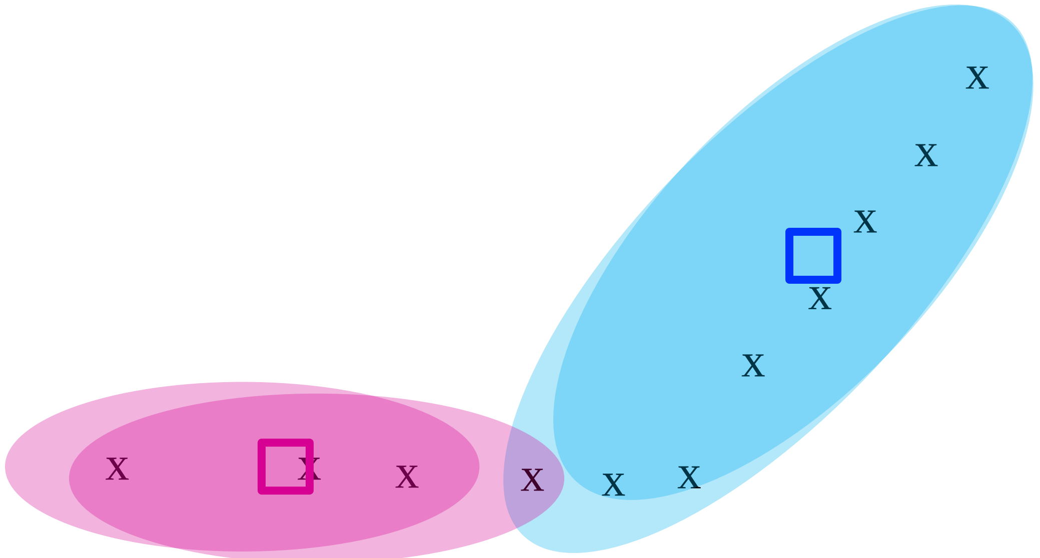
- **1)** For each point, place it in the cluster whose current centroid it is nearest
- **2)** After all points are assigned, update the locations of centroids of the ***k*** clusters
- **Repeat 1 and 2 until convergence**
  - **Convergence:** Points don't move between clusters and/or centroids stabilize
    - "stabilize":
      - e.g. sum of (squared) centroid changes < threshold

Example: Assigning Clusters



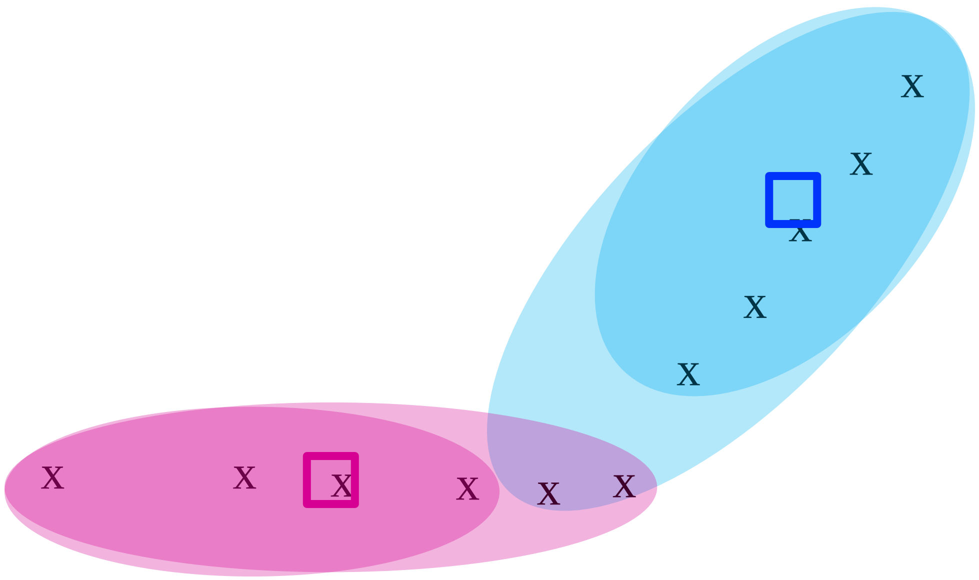
x … data point
… centroid

Example: Assigning Clusters



x … data point
… centroid

Example: Assigning Clusters



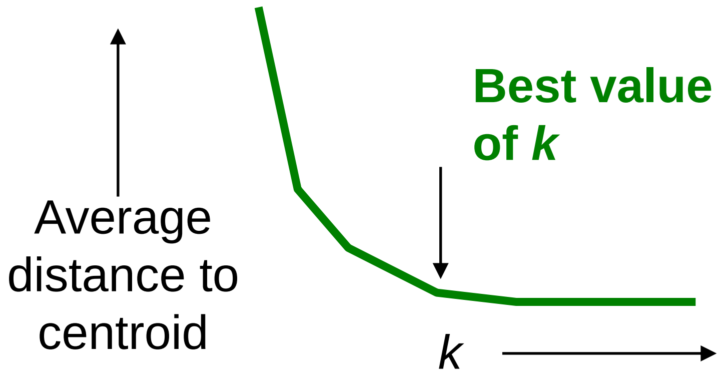
x … data point
□ … centroid

Clusters at the end

Getting the *k* right

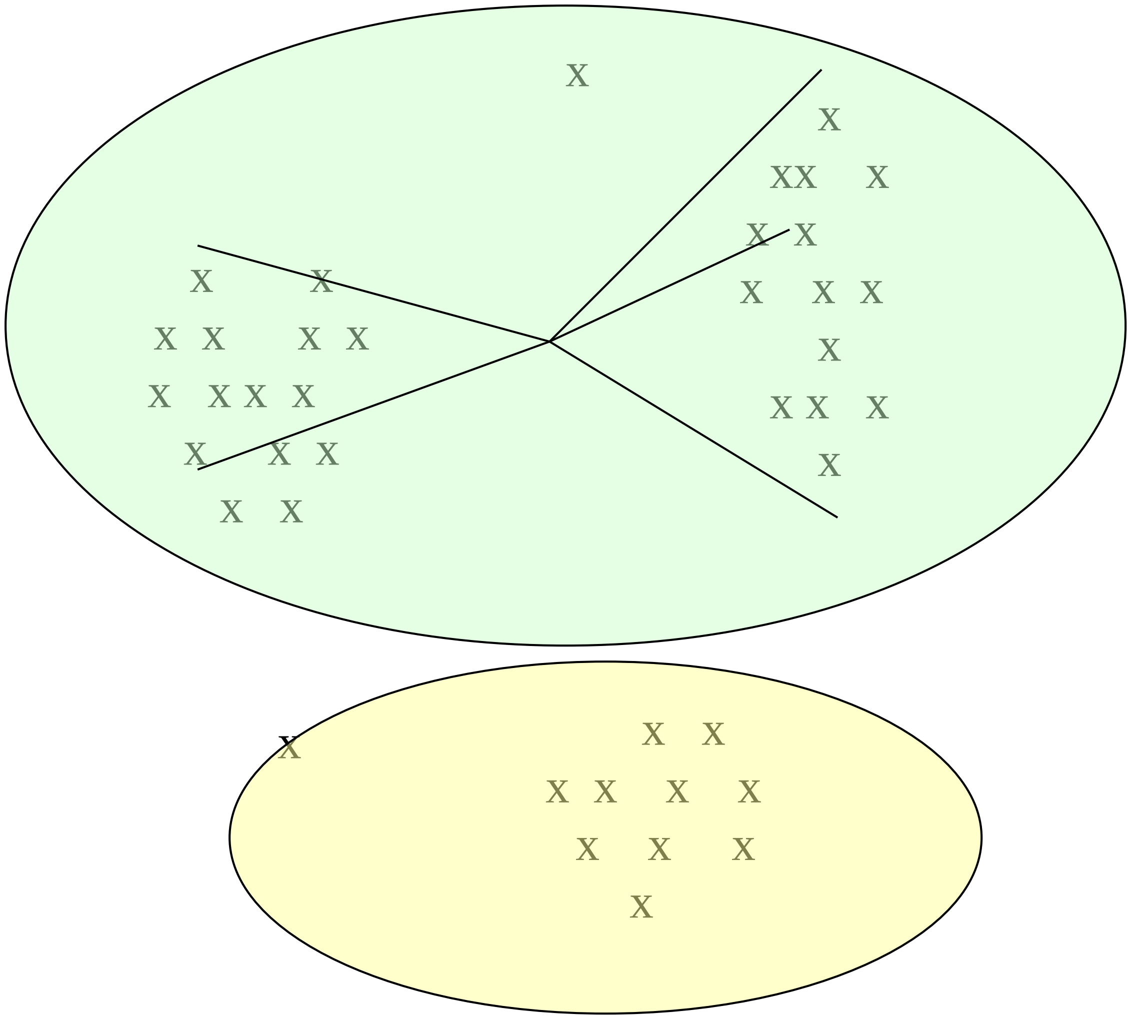
How to select *k*?

- Try different **k**, looking at the change in the average distance to centroid as **k** increases
- Average falls rapidly until right **k**, then changes little



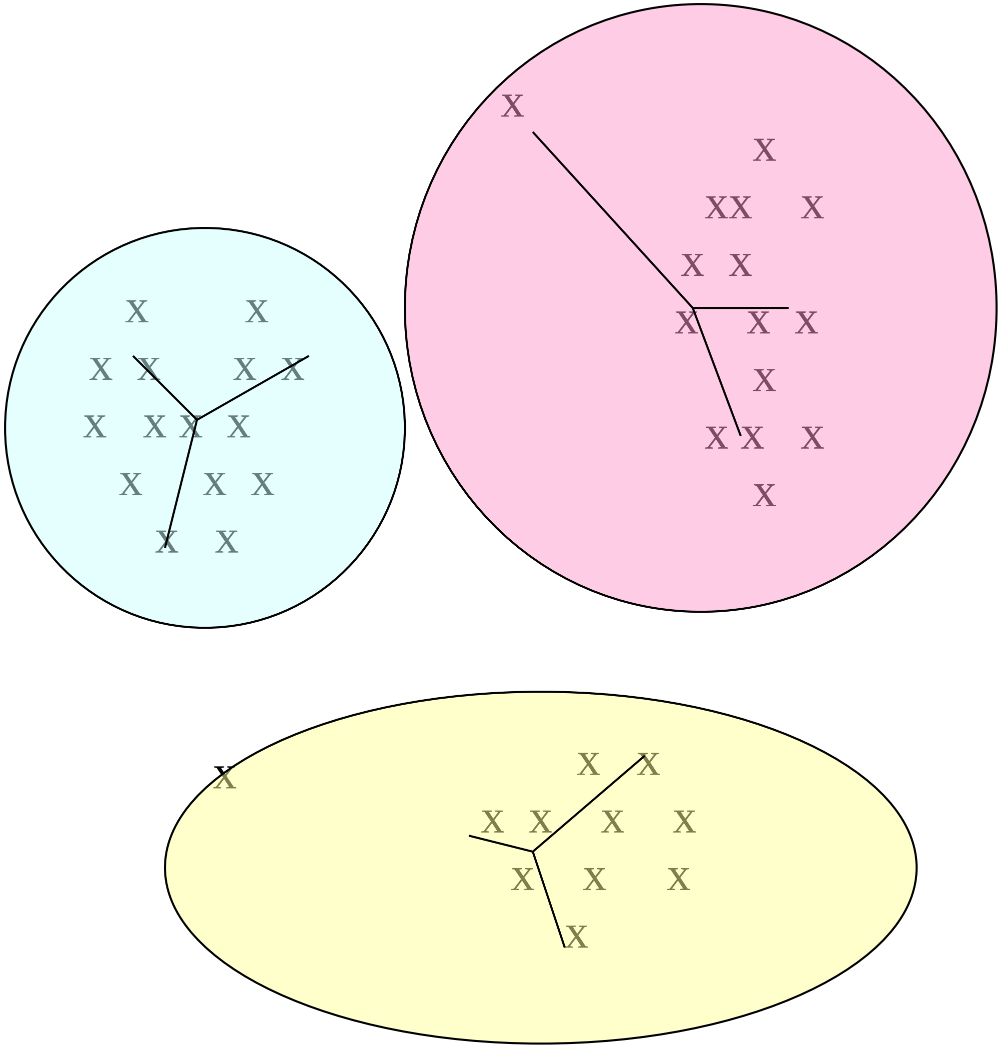
Example: Picking *k*

Too few;
many long distances to centroid



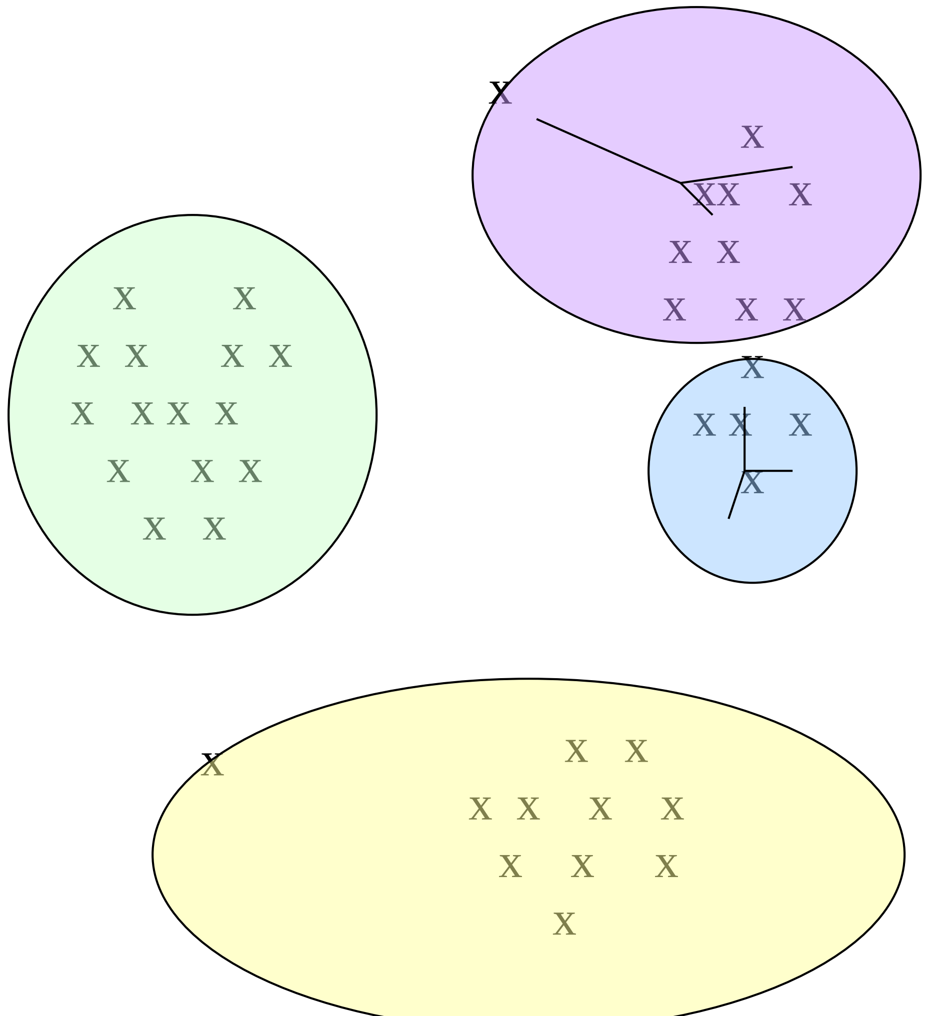
Example: Picking *k*

Just right;
distances
rather short



Example: Picking *k*

Too many;
little improvement
in average
distance



Implementing k-Means in Spark

Reading Points

Read-in points into RDD
Pick k points at random
Repeat until convergence
    1) Place each point it in the cluster with nearest centroid
    2) Update the locations of centroids of the ***k*** clusters

NumPy: optimized library for arrays and linear algebra

```
import numpy as np
from pyspark import SparkContext

def parseVector(line):
        return np.array([float(x) for x in line.split(' ')])

sc = SparkContext(appName="PythonKMeans")
lines = sc.textFile(sys.argv[1])
data = lines.map(parseVector).cache()
```

Function parseVector turns a text line with numbers into a numpy-vector

Created RDD has numpy-vectors as records; cached in memory

Picking k Initial Centroids

Read-in points into RDD
Pick k points at random
Repeat until convergence
  1) Place each point it in the cluster with nearest centroid
  2) Update the locations of centroids of the ***k*** clusters

2nd argument given to python process is parsed as K

```
K = int(sys.argv[2])
```

Collection with K numpy-vectors

Spark action: samples K records and returns to the driver

```
centroids = data.takeSample(False, K, 1)
newCentroids = centroids[:] # copy array
```

Testing Convergence

Read-in points into RDD
Pick k points at random
Repeat until convergence
- **1)** Place each point it in the cluster with nearest centroid
- **2)** Update the locations of centroids of the ***k*** clusters

Threshold for convergence

```
convergeDist = float(sys.argv[3])
```

Computes sum of squared Euclidean distances between old and new centroids

```
def distanceCentroidsMoved(oldCentroids, newCentroids):
    sum = 0.0
    for index in range(len(oldCentroids)):
        sum += np.sum( (oldCentroids[index] - newCentroids[index]) ** 2 )
    return sum
```

```
tempDist = 2* convergeDist                # constant larger than convergeDist
while  tempDist > convergeDist:
        <k-Means iteration, use centroids, compute newCentroids>
        tempDist = distanceCentroidsMoved(centroids, newCentroids)
        centroids = newCentroids[:]       # copy
```

Finding Closest Centroids /1

Read-in points into RDD
Pick k points at random
Repeat until convergence

1) Place each point it in the cluster with nearest centroid
2) Update the locations of centroids of the ***k*** clusters

Input is a point p and list of K centroids

```
def closestPoint(p, centroids):
    bestIndex = 0
    closest = float("+inf")
    for index in range(len(centroids)):
        tempDist = np.sum((p - centroids[index]) ** 2)
        if tempDist < closest:
            closest = tempDist
            bestIndex = index
    return bestIndex
```

For a point p (=numpy vector) computes index of the closest centroid in centroids

Finding Closest Centroids /2

Read-in points into RDD
Pick k points at random
Repeat until convergence
- **1)** Place each point it in the cluster with nearest centroid
- **2)** Update the locations of centroids of the ***k*** clusters

```
while tempDist > convergeDist:
    closest = data.map(
        lambda p: (closestPoint(p, centroids), (p, 1)) )
```

What is the "1" for?

Point p assigned to cluster with index j becomes a record (j, (p,1)) in a new RDD (i.e. record is a nested tuple)

Update the Location of Centroids /1

Read-in points into RDD
Pick k points at random
Repeat until convergence

1) Place each point it in the cluster with nearest centroid

2) Update the locations of centroids of the ***k*** clusters

```
while tempDist > convergeDist:
    closest =  …
    for cIndex in range(K):
        closestOneCluster=closest.filter(lambda d: d[0] == cIndex)
                    .map(lambda d: d[1])
```

Each d is a record (j, (p,1)), so d[0] is cluster index j of p

This RDD contains tuples $(p_0,1)$, $(p_1,1)$,… for all points in cluster with index = cIndex

Update the Location of Centroids/2

Read-in points into RDD
Pick k points at random
Repeat until convergence
- **1)** Place each point it in the cluster with nearest centroid
- **2)** Update the locations of centroids of the ***k*** clusters

```
while tempDist > convergeDist:
    closest =  …
    for cIndex in range(K):
        closestOneCluster=closest.filter(lambda d: d[0] == cIndex)
                .map(lambda d: d[1])
        sumAndCountOneCluster=closestOneCluster.reduce(
                lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1]) )
        vectorSum = sumAndCountOneCluster[0]
        count = sumAndCountOneCluster[1]
        newCentroids[cIndex] = vectorSum / count
```

Each d is a record (j, (p,1)), so d[0] is cluster index j of p

The Complete Loop

```
tempDist = 2* convergeDist
while  tempDist > convergeDist:
        closest = data.map(lambda p: (closestPoint(p, centroids), (p, 1)) )
        for cIndex in range(K):
                closestOneCluster=closest.filter(lambda d: d[0] == cIndex)
                        .map(lambda d: d[1])
                sumAndCountOneCluster=closestOneCluster.reduce(
                        lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1]))
                vectorSum = sumAndCountOneCluster[0]
                count = sumAndCountOneCluster[1]
                newCentroids[cIndex] = vectorSum / count

        tempDist = distanceCentroidsMoved(centroids, newCentroids)
        centroids = newCentroids[:]
```