Aalto University
School of Science

CS-E4610
Modern Database Systems
05.01.2018-05.04.2018

Lecture 09
Spark for batch and streaming processing

FREDERICK AYALA-GÓMEZ
PHD STUDENT IN COMPUTER SCIENCE, ELTE UNIVERSITY
VISITING RESEARCHER, AALTO UNIVERSITY

Agenda

Why not MapReduce?

- Iterative Algorithms
- Interactive Analytics

Spark at a glance

- Scala at a glance
- Distributed Data Parallelism
- Fault Tolerance
- Programming Model
- Spark Runtime

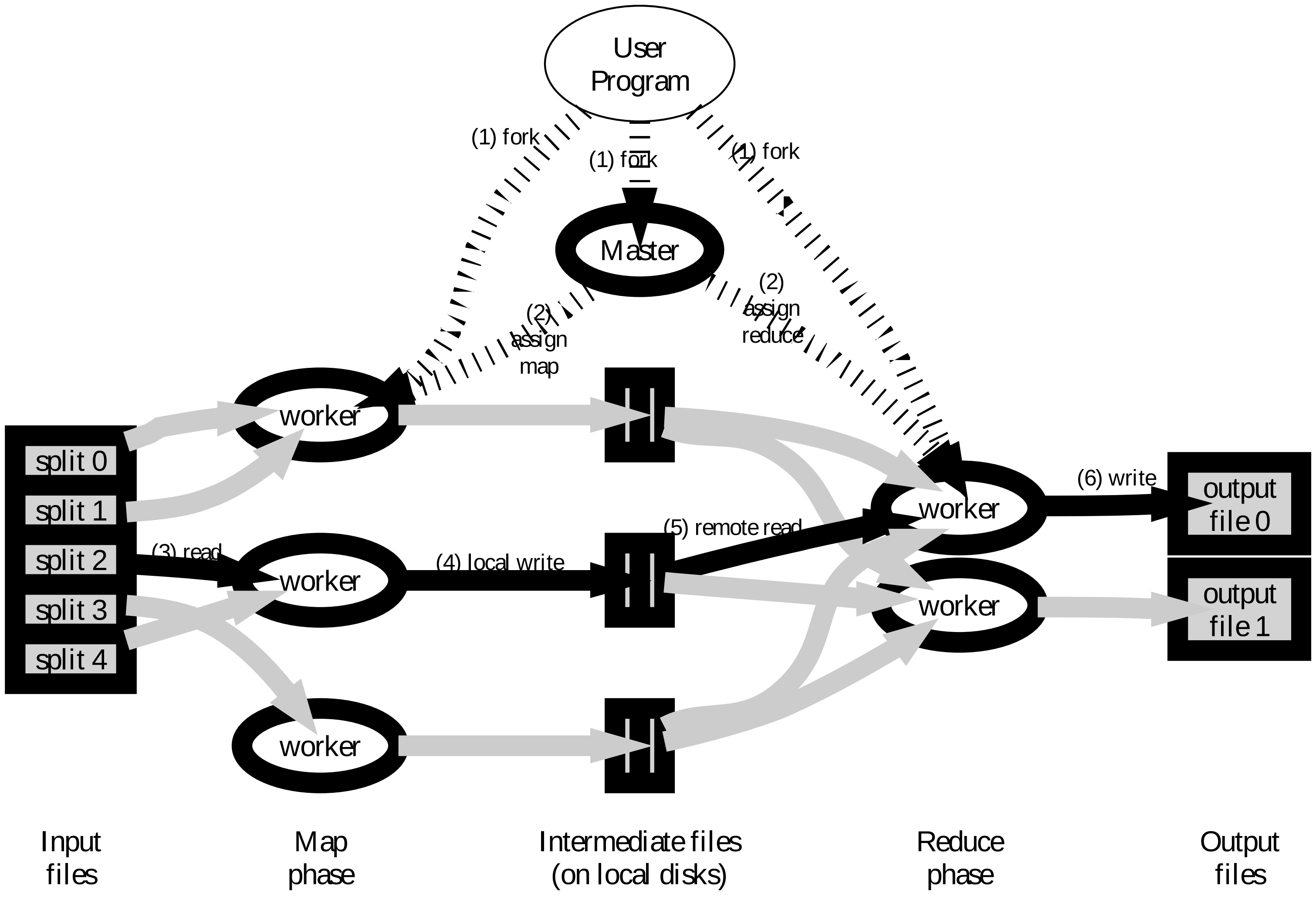
How to use Spark?

- RDD
- Transformations (Lazy)
- Actions (Eager)
- Reduction Operations
- Pair RDDs
- Join
- Shuffling and Partitioning

Why not map reduce?

MapReduce flows are ***acyclic***

Not efficient for ***some*** applications



Why not map reduce?

Zaharia, Matei, et al. "Spark: Cluster computing with working sets." HotCloud 10.10-10 (2010): 95.

Iterative algorithms

Many common machine learning algorithms **repeatedly apply the same function on the same dataset** (e.g., gradient descent)

MapReduce repeatedly reloads (reads & writes) data which is costly

Why not map reduce?

Zaharia, Matei, et al. "Spark: Cluster computing with working sets." HotCloud 10.10-10 (2010): 95.

Interactive analytics

Load data in memory and query repeatedly

MapReduce would re-read data

Lightning-fast cluster computing

Spark at a Glance.

Before we talk about Spark... Let's talk about **Scala**

Aalto University
School of Science

Scala

Prof. Martin Odersky

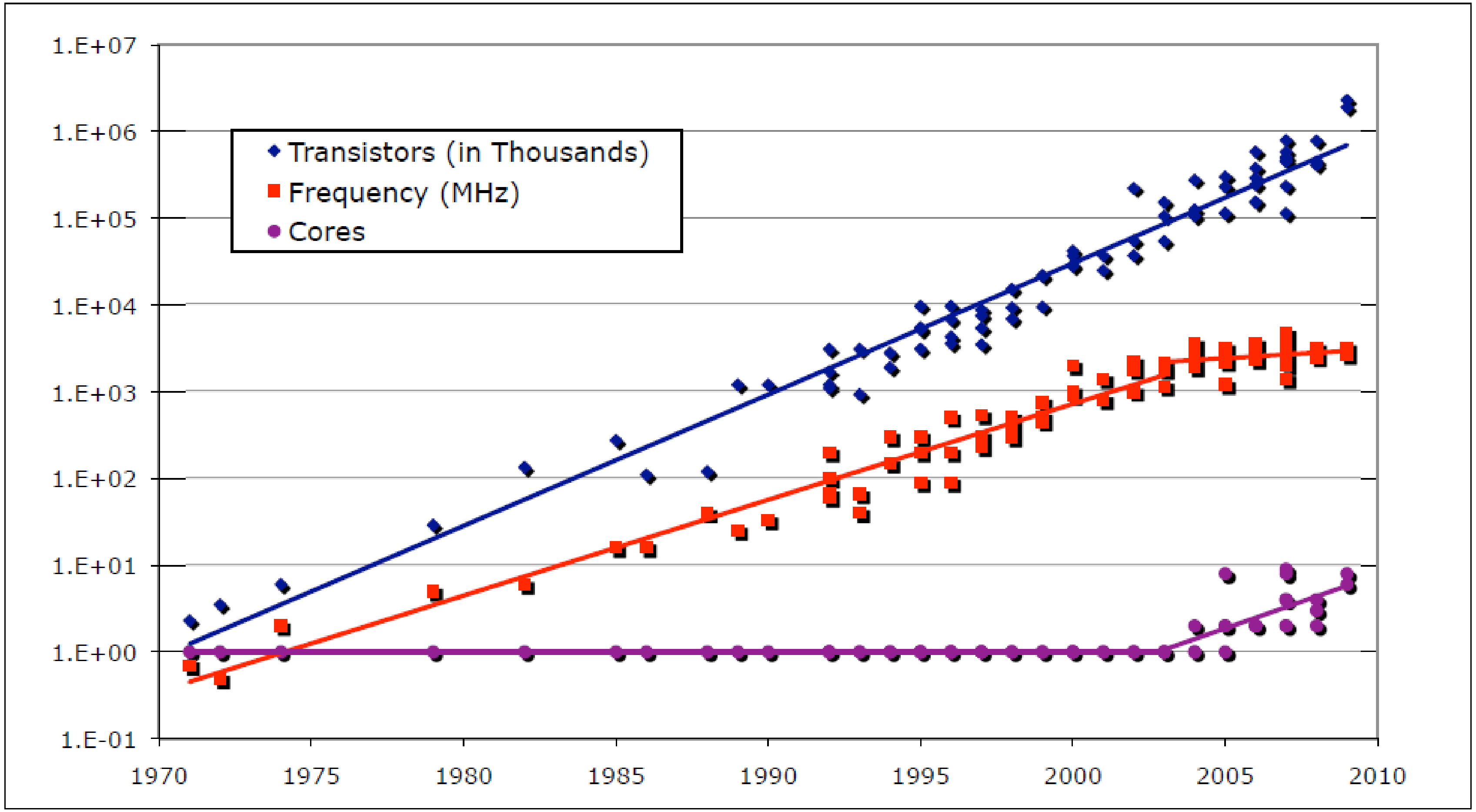
Java Generics

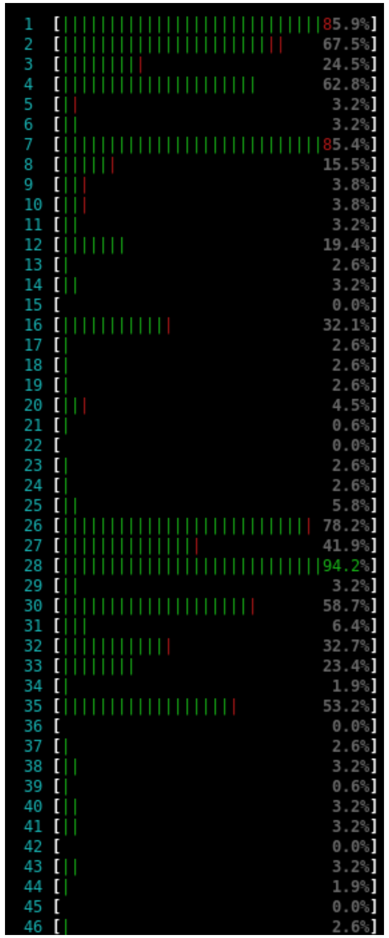
Scala

Lightbend (Typesafe)

Coursera: Functional Programming in Scala,
École **P**olytechnique **F**édérale de **L**ausanne

From Fast Single Cores to Multicores





Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten and Krste Asanovic

Martin Odersky, "Working Hard to Keep It Simple". OSCON Java 2011

Typical Bare Metal Servers

Intel Xeon E5-2690 v3

Dual Intel Xeon E5-2690 v3 (24 Cores, 2.60 GHz)

64GB RAM (64GB maximum)

Up to 4 Internal Hard Drives

Intel Xeon E7-4820 v2

Quad Intel Xeon E7-4820 v2 (32 Cores, 2.00 GHz)

128GB RAM (3072GB maximum)

Up to 24 Internal Hard Drives

Intel Xeon E5-4650

Quad Intel Xeon E5-4650 (32 Cores, 2.70 GHz)

64GB RAM (1024GB maximum)

Up to 24 Internal Hard Drives

Intel Xeon E7-4850 v2

Quad Intel Xeon E7-4850 v2 (48 Cores, 2.30 GHz)

128GB RAM (3072GB maximum)

Up to 24 Internal Hard Drives

Intel Xeon E5-2690 v3

Dual Intel Xeon E5-2690 v3 (24 Cores, 2.60 GHz)

256GB RAM (256GB maximum)

Up to 4 Internal Hard Drives

Intel Xeon E5-2650

Dual Intel Xeon E5-2650 (16 Cores, 2.00 GHz)

128GB RAM (128GB maximum)

Up to 4 Internal Hard Drives

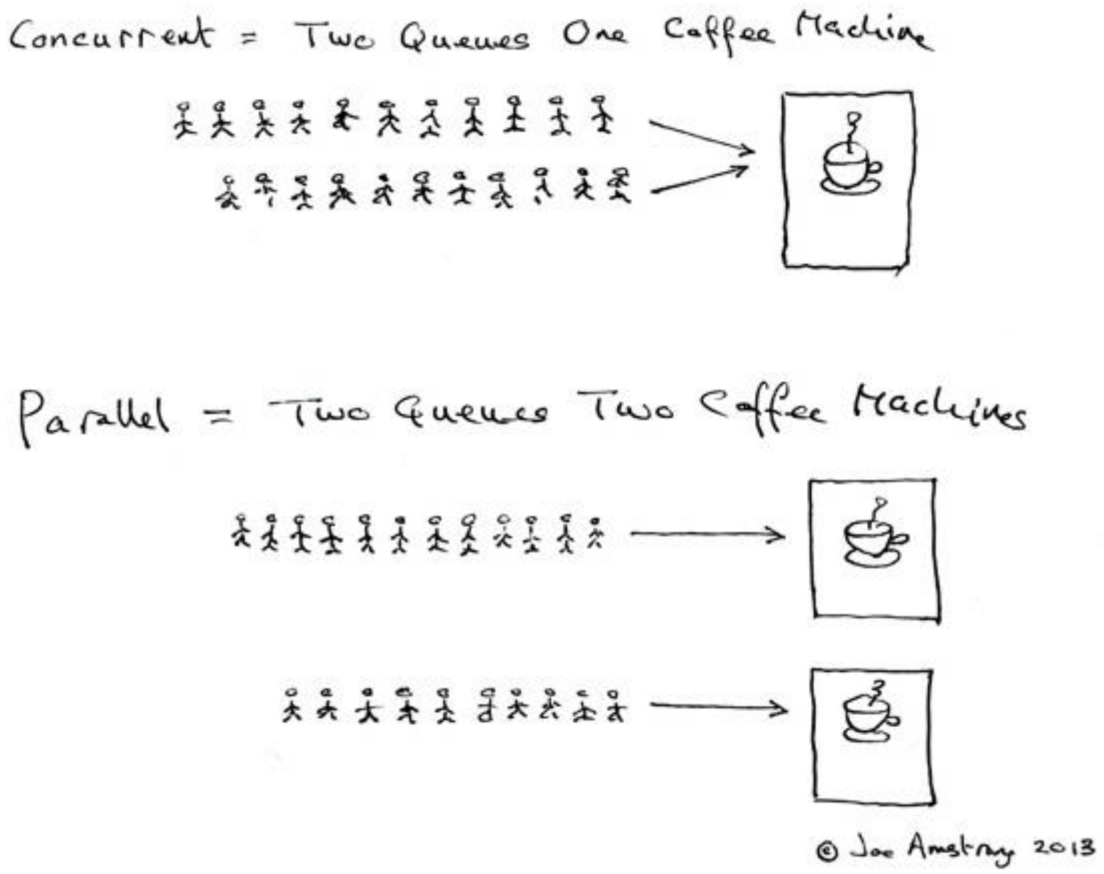
https://softlayer.com

High Performance Computing (15/11)

RANK	SITE	CORES
1	National Super Computer Center in Guangzhou China	3,120,000
2	DOE/SC/Oak Ridge National Laboratory United States	560,640
3	DOE/NNSA/LLNL United States	1,572,864
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	705,024
5	DOE/SC/Argonne National Laboratory United States	786,432
6	DOE/NNSA/LANL/SNL United States	301,056
7	Swiss National Supercomputing Centre (CSCS) Switzerland	115,984
8	HLRS - Höchstleistungsrechenzentrum Stuttgart Germany	185,088
9	King Abdullah University of Science and Technology Saudi Arabia	196,608
10	Texas Advanced Computing Center/Univ. of Texas United States	462,462

http://top500.org/lists/2015/11/

Concurrency and Parallelism



Manage concurrent execution threads

Execute programs faster using the multi-cores

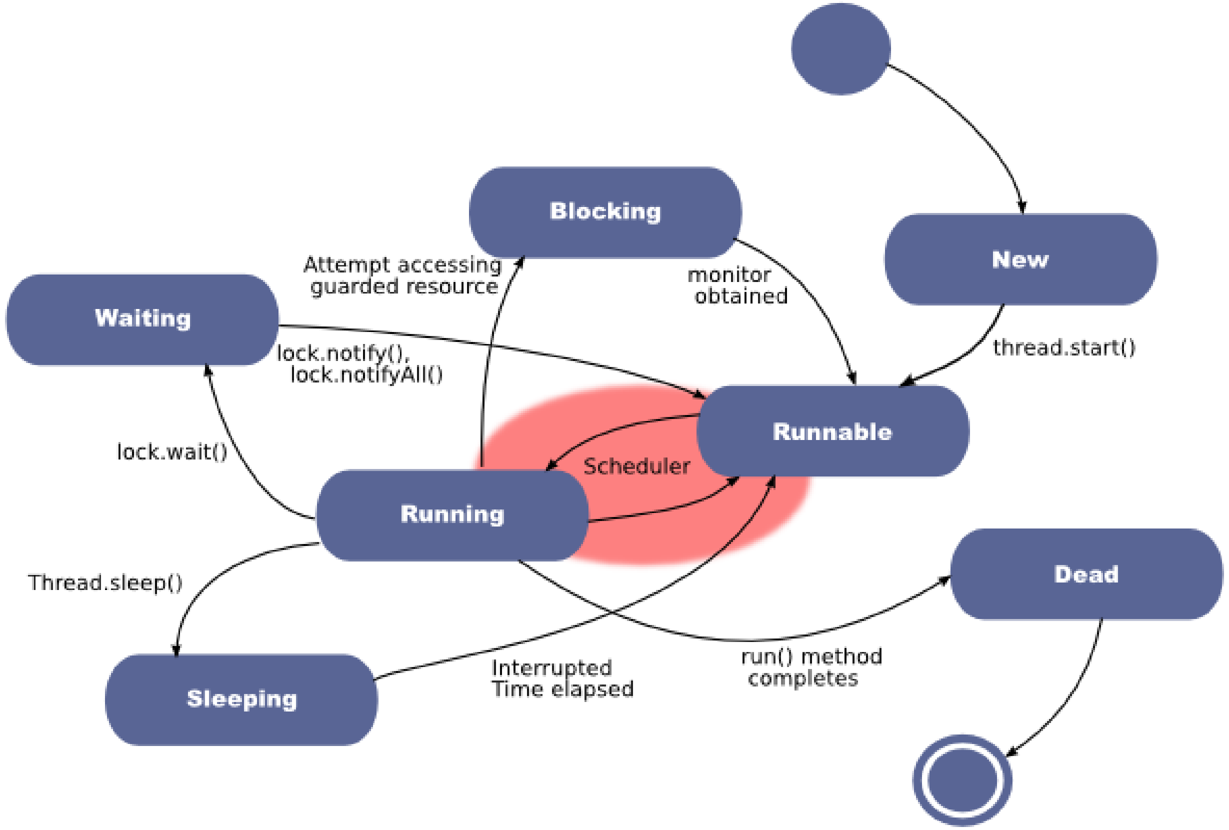
What can go wrong...?

Concurrent Threads

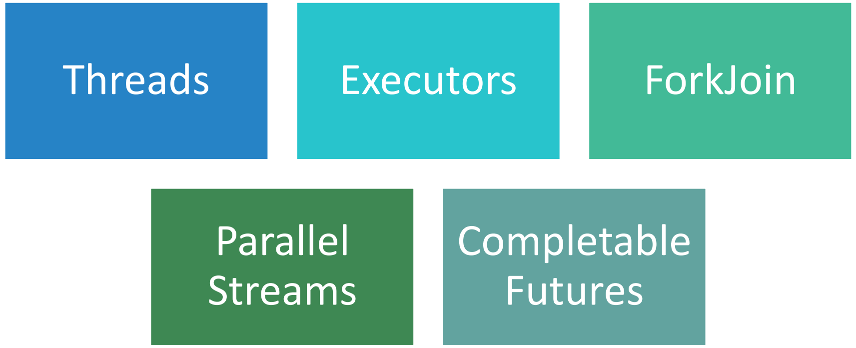
Shared Mutable State

```
var x = 0

async { x = x + 1}

async { x = x * 2}

// x could be 0, 1, 2
```

Threads in Java



Manage concurrent & parellel executions



http://booxs.biz/EN/java/Threads%20in%20Java.html

Scala at a Glance

Static Typing

Functional

Scala

Lightweight Syntax

Object Oriented

- High Order Functions
- Immutable over mutable
- Avoid Shared Mutable States
- Efficient Immutable Data Structures

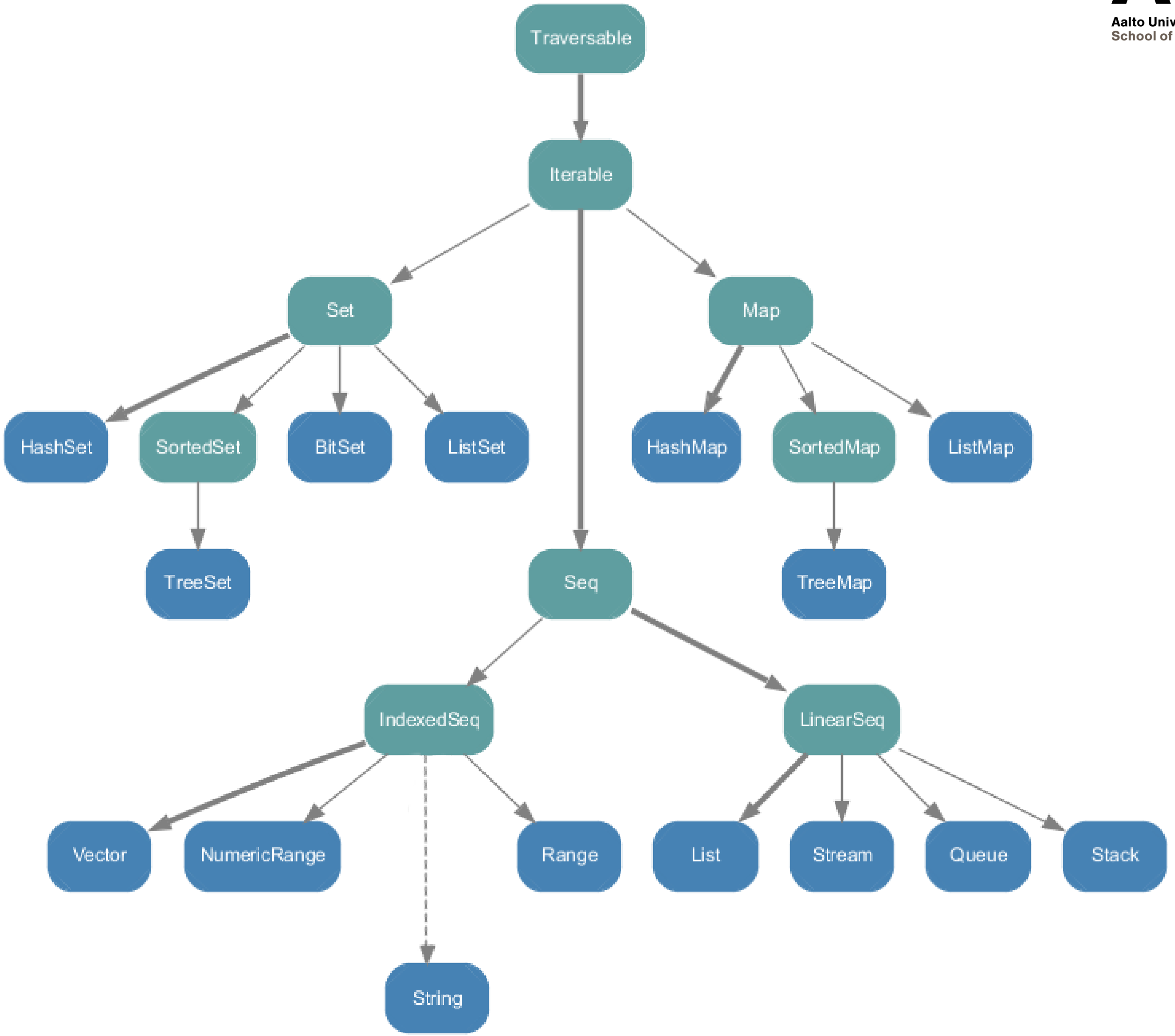
Scala Collections

Help to Organize Data

***Immutable* collections never change.**

Collections are Sequential or **Parallel**

https://docs.scala-lang.org/overviews/collections/overview.html



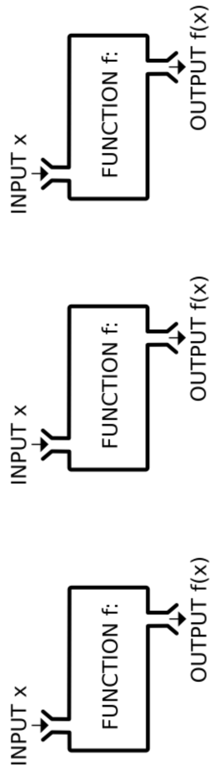
Shared Memory Data Parallelism (Scala Parallel collections)

user_id	movie_id	rating	timestamp
5513	2020	4	959710399
2372	110	4	974675977
3973	2001	4	966357968
48	360	2	978066474
1021	3873	5	975008402
3601	2919	5	966640755
5152	2262	4	962834153
5115	3186	4	962307039
1104	2395	3	974921595

CPU Thread 0

CPU Thread 1

CPU Thread 2




```
result = ratings.map(...)
```

Split data

Threads run independently

Combine Results

Distributed Data Parallelism

Node 1

user_id	movie_id	rating	timestamp
5686	2644	1	958690915
263	552	3	976651725
4227	3710	1	965323696
3475	2338	1	997331179
1004	2568	2	975042992
3823	2348	4	965942528
33	3359	3	978982566
2069	1747	4	974659419
3469	2115	3	967155526

CPU Thread 0

CPU Thread 1

CPU Thread 2

Node 2

user_id	movie_id	rating	timestamp
1128	3504	4	974907449
1125	784	3	1018464072
3311	2505	2	968651313
3054	1028	4	970157625
5181	2476	5	1037810320
4906	21	5	962735529
2153	919	3	976935927
2794	1276	4	972920942
2624	3926	3	973651923

CPU Thread 0

CPU Thread 1

CPU Thread 2

Node 3

user_id	movie_id	rating	timestamp
1044	1097	4	974966184
214	2390	4	976901020
4227	380	5	965322628
1474	1206	5	975100333
1050	2692	4	974962477
5621	288	5	959098092
2235	1911	3	974614456
3361	1917	4	967687406
3021	2858	5	970506920

CPU Thread 0

CPU Thread 1

CPU Thread 2

Node 4

user_id	movie_id	rating	timestamp
5513	2020	4	959710399
2372	110	4	974675977
3973	2001	4	966357968
48	360	2	978066474
1021	3873	5	975008402
3601	2919	5	966640755
5152	2262	4	962834153
5115	3186	4	962307039
1104	2395	3	974921595

CPU Thread 0

CPU Thread 1

CPU Thread 2

`result = ratings.map(...)`

Split data among nodes

Nodes run independently

Combine Results

Network latency is a problem

Distribution: Failure and Latency

Task	Latency	Humanized (Latency * 1 Billion)	
L1 cache reference	0.5 ns	0.5 s	One heart beat (0.5 s)
Branch mispredict	5 ns	5 s	Yawn
L2 cache reference	7 ns	7 s	Long yawn
Mutex lock/unlock	25 ns	25 s	Making a coffee
Main memory reference	100 ns	100 s	Brushing your teeth
Compress 1K bytes with Zippy	3,000 ns = 3 µs	50 min	One episode of a TV show (including ad breaks)
Send 2K bytes over 1 Gbps network	20,000 ns = 20 µs	5.5 hr	From lunch to end of work day
SSD random read	150,000 ns = 150 µs	1.7 days	A normal weekend
Read 1 MB sequentially from memory	250,000 ns = 250 µs	2.9 days	A long weekend
Round trip within same datacenter	500,000 ns = 0.5 ms	5.8 days	A medium vacation
Read 1 MB sequentially from SSD*	1,000,000 ns = 1 ms	11.6 days	Waiting for almost 2 weeks for a delivery
Disk seek	10,000,000 ns = 10 ms	16.5 weeks	A semester in university
Read 1 MB sequentially from disk	20,000,000 ns = 20 ms	7.8 months	Almost producing a new human being
Send packet CA->Netherlands->CA	150,000,000 ns = 150 ms	4.8 years	Average time it takes to get a bachelor's degree

Memory Disk Network

Fast--------------->Slow

https://gist.github.com/jboner/2841832
http://norvig.com/21-days.html

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

tion, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.,* looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.,* to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a

Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

Spark Resilient Distributed Datasets (RDD)

Immutable collection of objects (Read-only)

Partitioned across machines

Once defined, programmer treats it as available (System re-builds it if lost / leaves memory)

Users can **explicitly cache** RDDs in memory

Re-use across MapReduce-like parallel operations

Main Challenge: Efficient fault-tolerance

Should be easy to re-build if part of data (e.g., a partition) is lost.

Achieved through **coarse-grained transformations** and **lineage**

Fault-tolerance

Coarse transformations

- e.g., *map* applies the same function to the data items.

Lineage:

- **Series of transformations** that led to a dataset.

If a partition is **lost**, there is enough information to re-apply the transformations and **re-compute** it

Programming Model

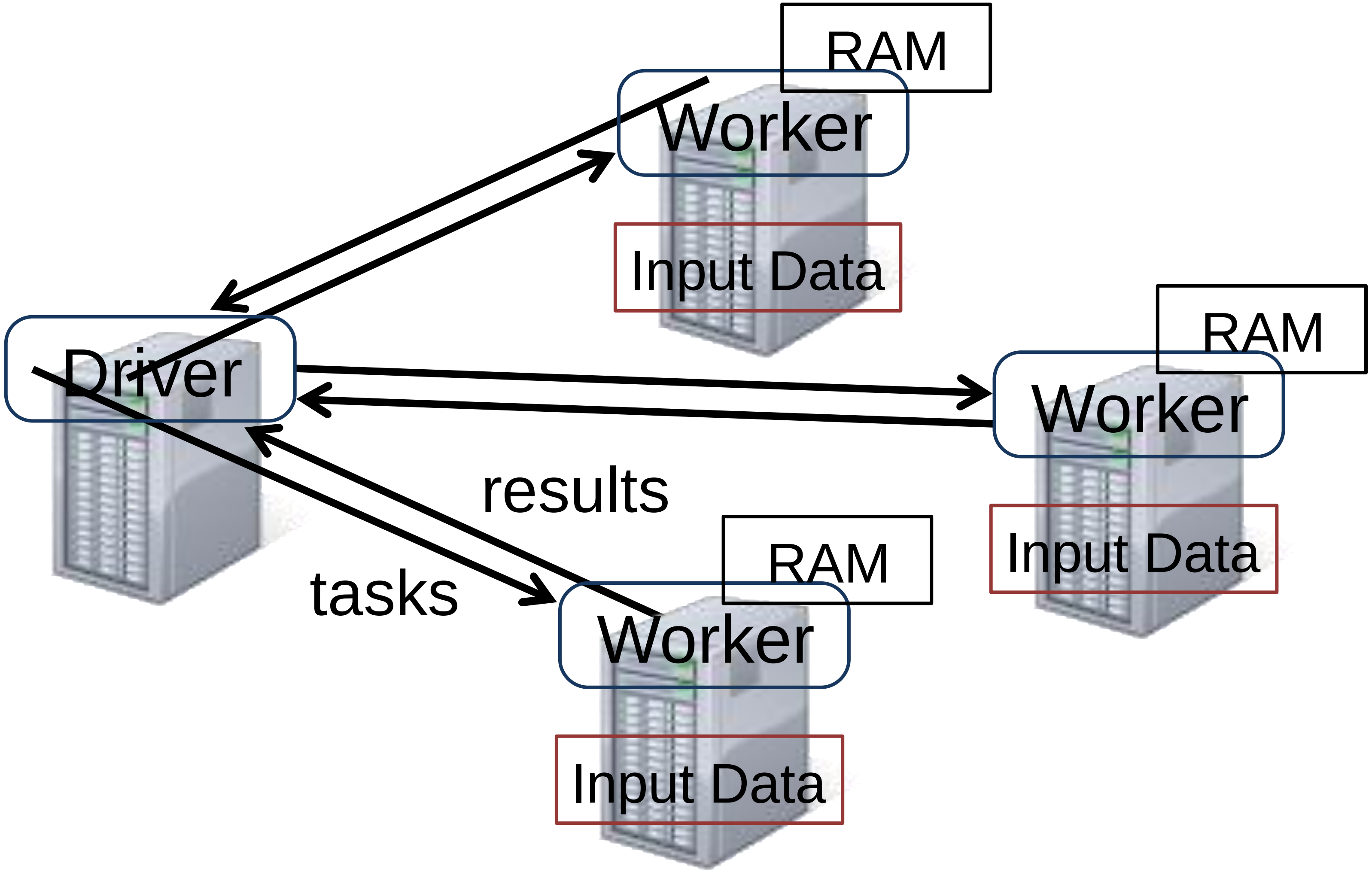
Developers write a **drive** program

- high-level control flow

Think of ***RDDs*** as objects that represent datasets that you distribute among several workers, and **transform** and apply **actions** in parallel.

Can also use restricted types of ***shared variables***

Spark runtime



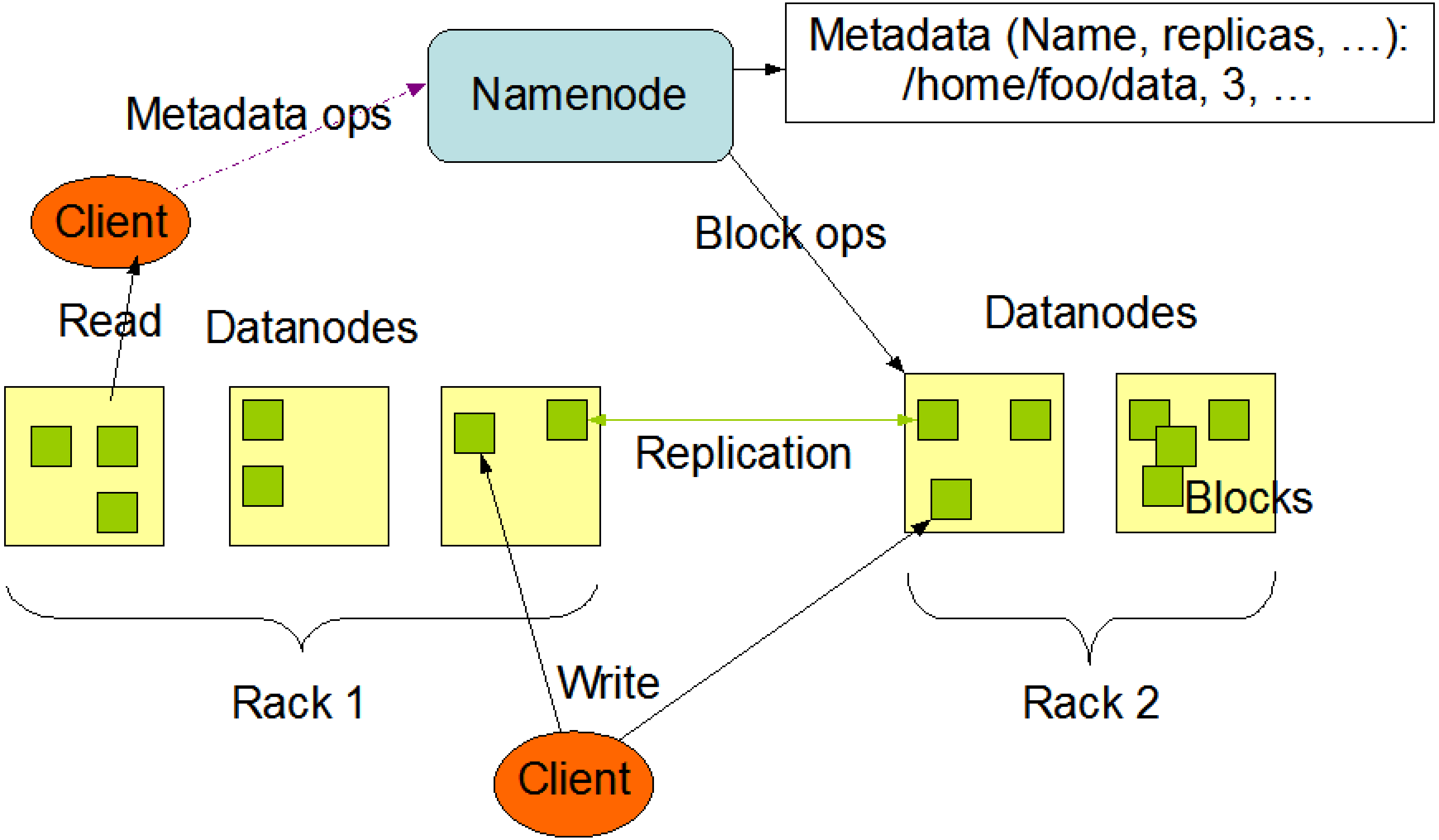
RDD

Immutable *(read-only)* collection of objects partitioned across a set of machines, that can be **re-built** if a partition is lost.

Constructed in the following ways:

- **From a file** in a shared file system (e.g., HDFS)
- **Parallelizing a collection** (e.g., an array) divide into partitions and send to multiple nodes
- **Transforming** an **existing RDD** (applying a map operation)

Hadoop Distributed FS (HDFS) architecture – *Different than Spark*



RDD

It does not exist at all time. Instead, there is enough information to compute the RDD when needed.

RDDs are ***lazily-created*** and ***ephemeral***

Lazy: Materialized only when information is extracted from them (through *actions*!)

Ephemeral: Might be discarded after use

Transformations **(Lazy)**

Lazy operations. The results are not immediately computed

Create a new RDD

Actions **(Eager)**

Aalto University
School of Science

RDDs are computed every time you run an action.

Return a value to the program or output the results (e.g., HDFS)

Why Spark?

Zaharia, Matei, et al. "Spark: Cluster computing with working sets." HotCloud 10.10-10 (2010): 95.

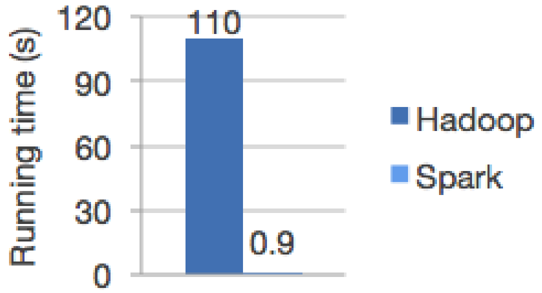
MapReduce:

Simple API (map, reduce)

Fault-tolerant

Spark:

- Simple and rich API.
- Fault-tolerant
- Reduces latency using ideas from functional programming (immutability, in-memory).
- l00x more performant than MapReduce (Hadoop), and more productive!



Logistic regression in Hadoop and Spark

How does a Spark program looks like? (PySpark)

Driver

Transformation

```
spark = pyspark.SparkContext(master="local[*]", appName="tour")
lines = spark.textFile("hdfs://namenodehost/dbpedia.csv")
zombie_movies = lines.filter(lambda x: "zombie" in x.lower())
count = zombie_movies.count()
print(f"There are {count} movies about zombies... scary.")
```

Action

```
Out[]: There are 20 movies about zombies... scary.
```

RDD

How does a Spark program looks like? (PySpark)

Driver

RDD

Action

Transformation

```python
spark = pyspark.SparkContext(master="local[*]", appName="tour")
lines = spark.textFile("hdfs://namenodehost/dbpedia.csv")
zombie_movies = lines.filter(lambda x: "zombie" in x.lower())
count = zombie_movies.count()
print(f"There are {count} movies about zombies... scary!")

romantic_movies = lines.filter(lambda x: "romantic" in x.lower())
count = romantic_movies.count()
print(f"There are {count} movies about romance <3")
```

```
Out[]: There are 20 movies about zombies... scary!
Out[]: There are 524 movies about romance <3
```

Let’s think about what happened…

Caching and Persistence

To prevent re-computing the RDDs, we can persist the data.

cache:

- **Memory** only storage

persist:

- Persistence can be **customized at different levels** (e.g., memory, disk)
- The default persistence is at memory level

How does a Spark program looks like? (PySpark)

Driver

RDD

Action

Persist

Transformation

```python
spark = pyspark.SparkContext(master="local[*]", appName="tour")
lines = spark.textFile("hdfs://namenodehost/dbpedia.csv").persist()
zombie_movies = lines.filter(lambda x: "zombie" in x.lower())
count = zombie_movies.count()
print(f"There are {count} movies about zombies... scary!")

romantic_movies = lines.filter(lambda x: "romantic" in x.lower())
count = romantic_movies.count()
print(f"There are {count} movies about romance <3")
```

```
Out[]: There are 20 movies about zombies... scary!
Out[]: There are 524 movies about romance <3
```

In the second time, the *lines* RDD was loaded from memory

Cluster Topology

Contains the **main**
Creates RDDs
Coordinates the execution
(transformations and actions)

Driver
Program

Cluster
Manager
(YARN/Mesos)

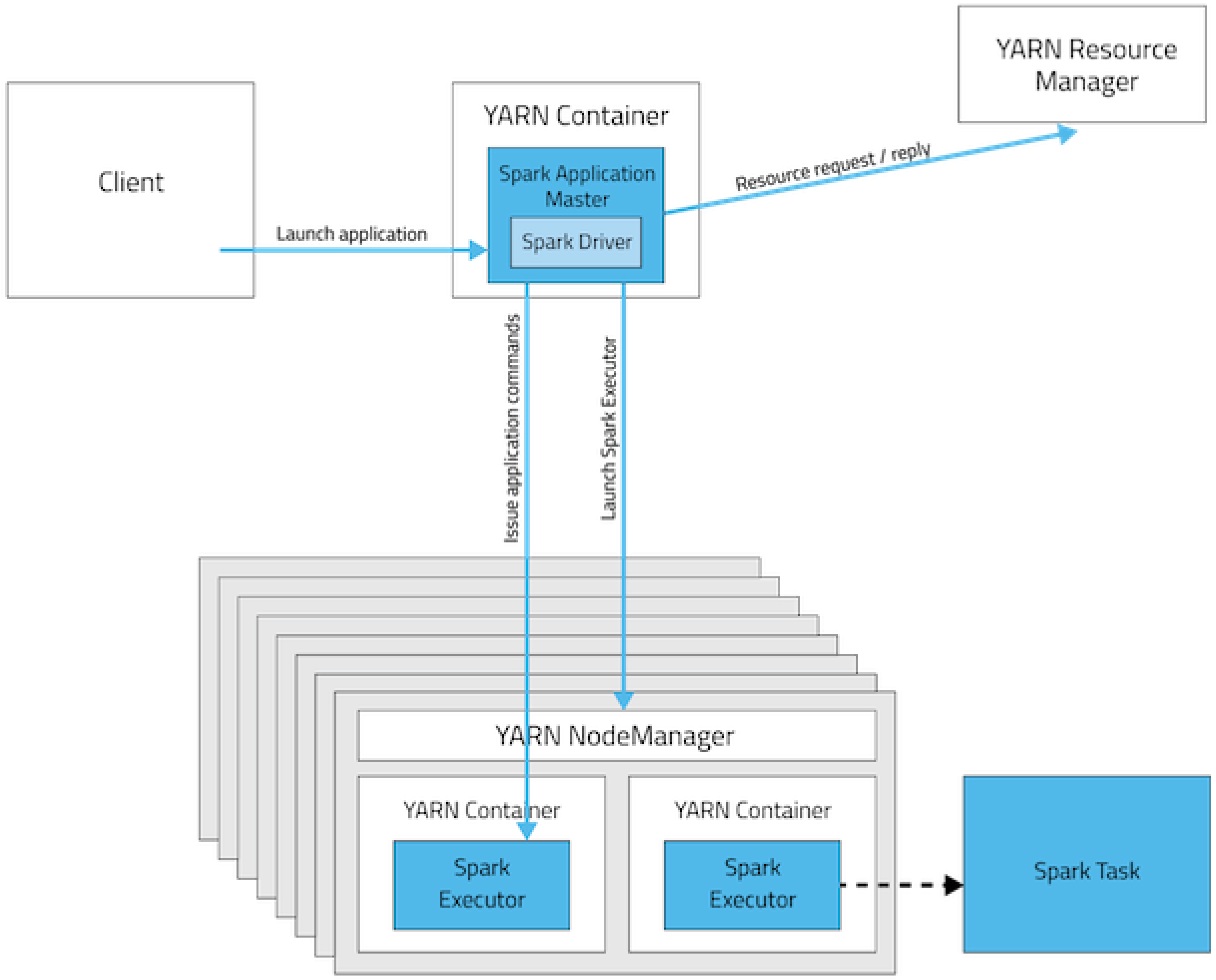
Run tasks
Return Results
Persist RDDs

Worker Node
(Executor)

Worker Node
(Executor)

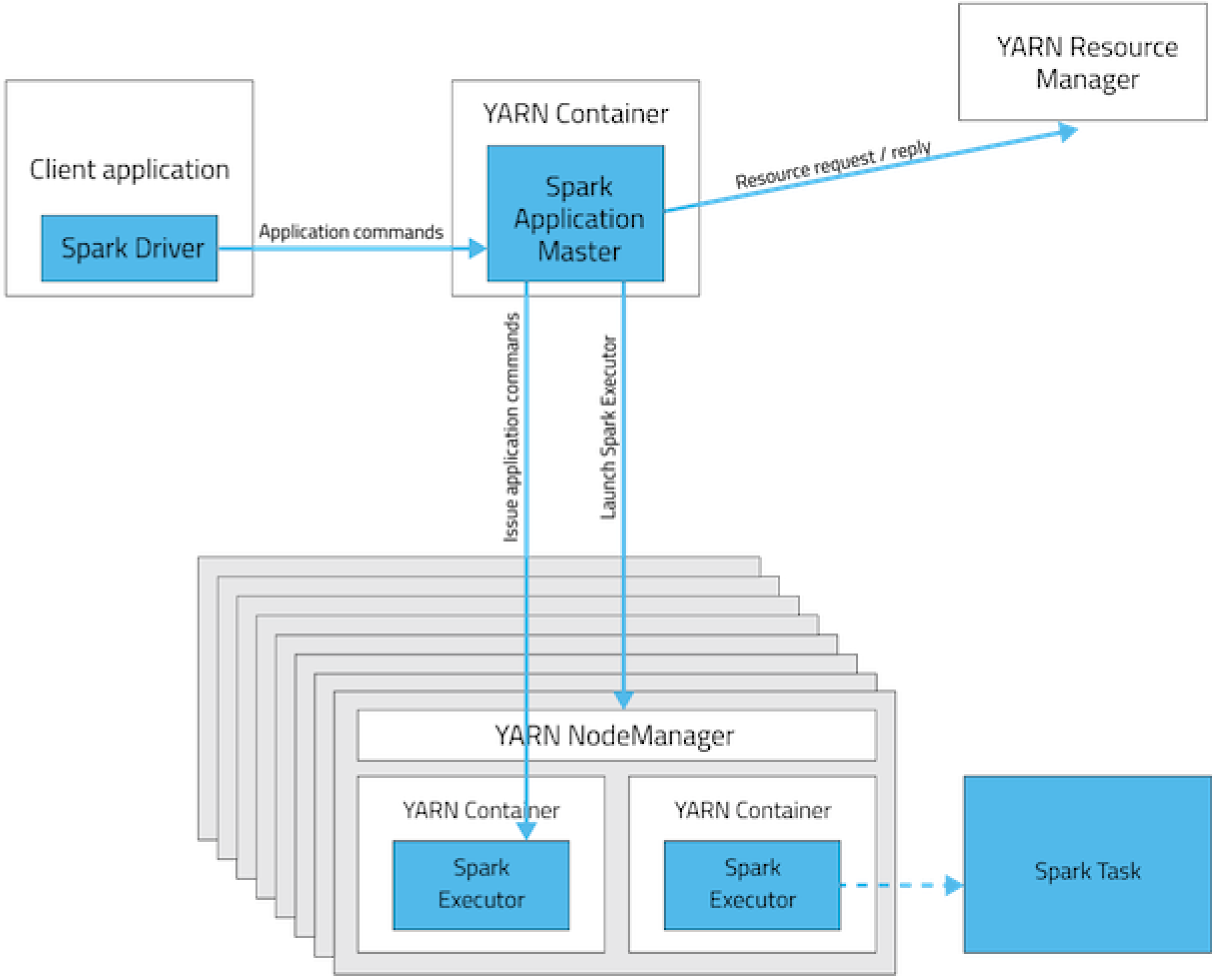
Worker Node
(Executor)

YARN-cluster mode



https://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/

YARN-client mode



https://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/

Difference between running modes

	YARN Cluster	YARN Client	Spark Standalone
Driver runs in:	Application Master	Client	Client
Who requests resources?	Application Master	Application Master	Client
Who starts executor processes?	YARN NodeManager	YARN NodeManager	Spark Slave
Persistent services	YARN ResourceManager and NodeManagers	YARN ResourceManager and NodeManagers	Spark Master and Workers
Supports Spark Shell?	No	Yes	Yes

https://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/

Cluster Topology - Evaluation

Aalto University
School of Science

```
spark = pyspark.SparkContext(master="local[*]", appName="tour")
lines = spark.textFile("hdfs://namenodehost/dbpedia.csv").persist()
zombie_movies = lines.filter(lambda x: "zombie" in x.lower())
lines.filter(lambda x: "zombie" in x.lower()).foreach(lambda x: print(x))
count = zombie_movies.count()
print(f"There are {count} movies about zombies... scary!")
```

```
Out[]: There are 20 movies about zombies... scary.
```

Action

Where are the zombie movies printed?

Cluster Topology - Evaluation

Actions *usually* communicate between workers' nodes and the driver's node.

It is important to think about where the tasks are going to be executed.

Large RDDs may cause out of memory errors in the driver node for some actions (e.g., collect). In that case, it's a good idea to output directly from the worker.

Reduction Operations

Traverse a collection and combine elements to produce a single combined result.

reduce(op)

- Reduces the elements of this RDD using the specified associative and cumulative operator.

fold(zeroValue, op)

- Aggregate the elements of each partition, and then the results for all the partitions, using a given associative function and a neutral "zero value."
- Requires the same type of data in the return.

aggregate(zeroValue, seqOp, combOp)

- Aggregate the elements of each partition, and then the results for all the partitions, using a given combine functions and a neutral "zero value."
- Possible to change the return type.

Pair RDD

Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

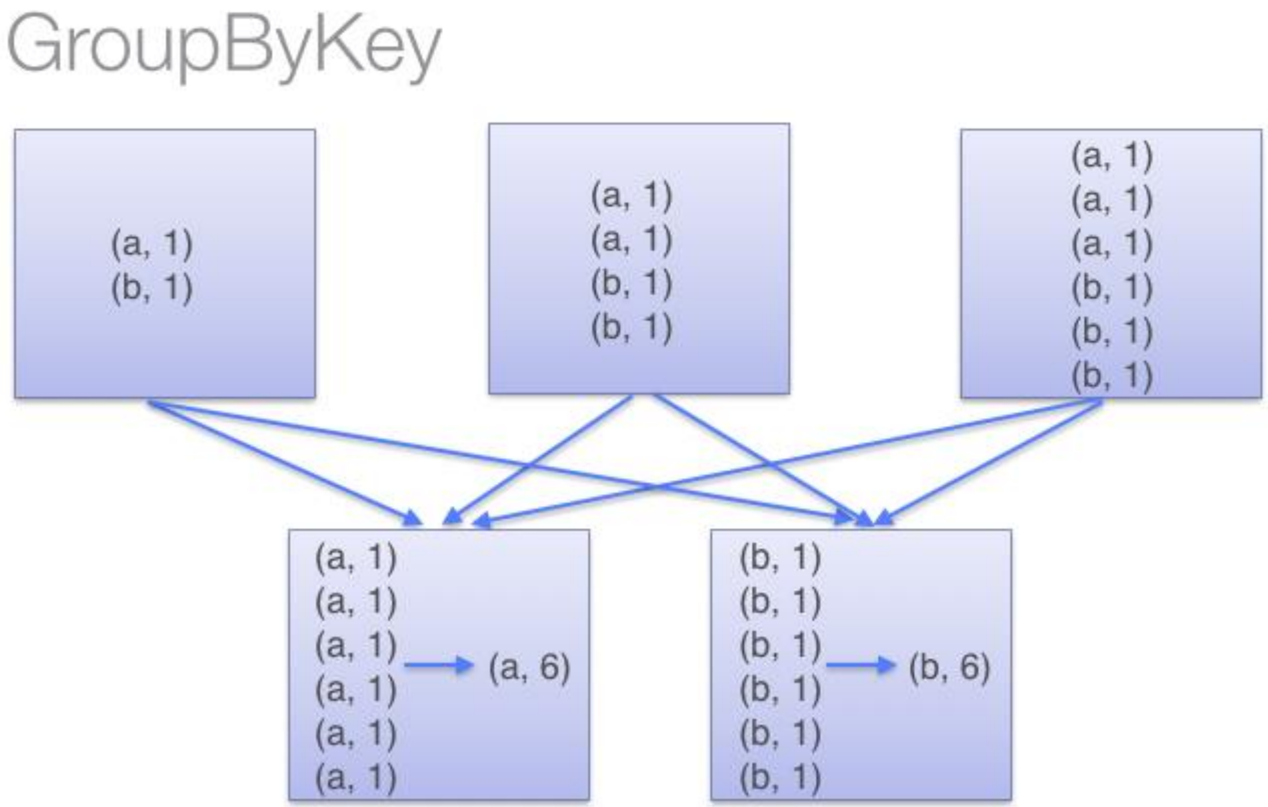
We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately.

Usually, we have large datasets that we can organize by a key (e.g., movie_id, user_id)

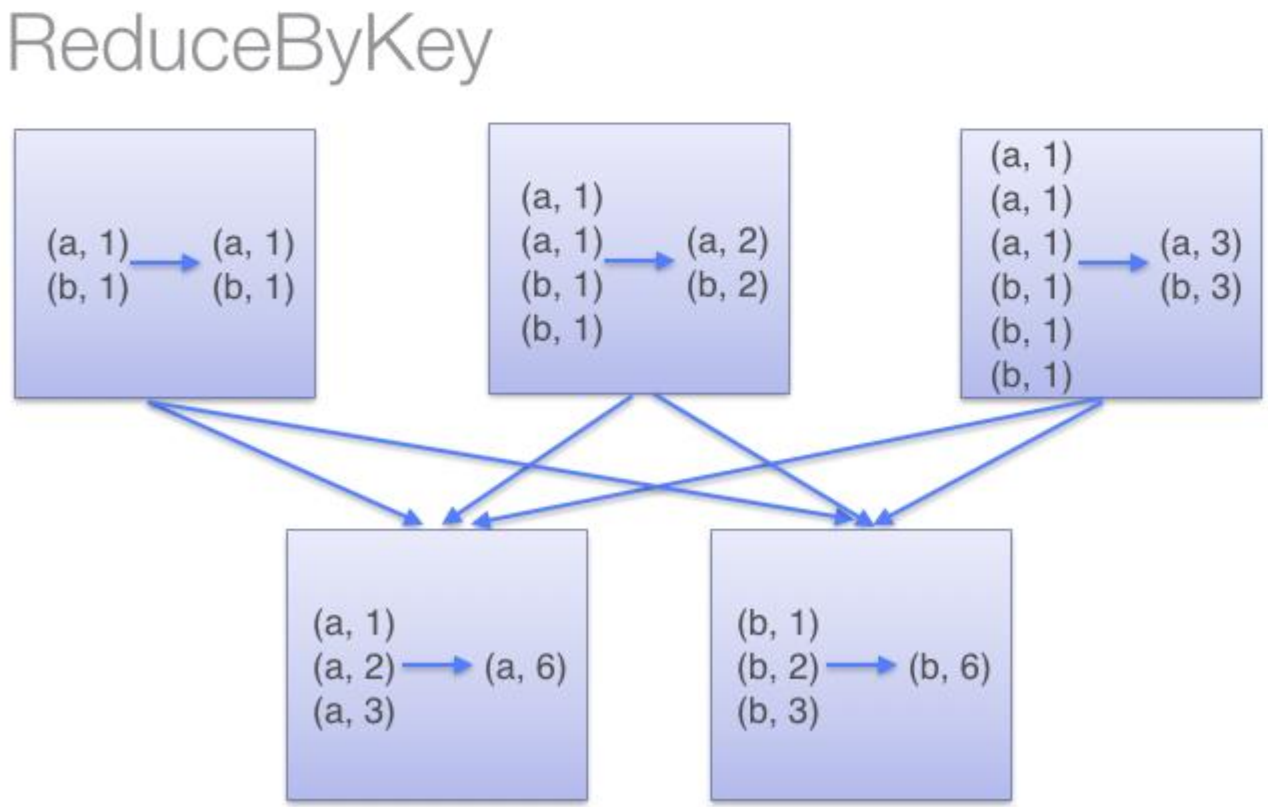
Useful because it improves how we handle the RDD

Pair RDDs have special methods for working with the data associated to the keys.

GroupByKey example



ReduceByKey example



Word Count example

```
spark = pyspark.SparkContext(master="spark://my_cluster:7070", appName="word_count")
lines = spark.textFile("hdfs://namenodehost/dbpedia.csv")
word_count = lines.flatMap(lambda x: [(w,1) for w in x.split(" ")]).reduceByKey(add).sortBy(lambda x: x[1], ascending=False)
for wc in word_count.take(10):
    print(wc)
```

```
Out[]:
('the', 20711)
('and', 15343)
('a', 10785)
('of', 9892)
('film', 9206)
('by', 8377)
('in', 7646)
('The', 7362)
('is', 6290)
('was', 5728)
```

Join

You can combine Pair RDDs using a **join**.

The combination can be by:

Inner joins (**join**)

- Key that appear in both Pair RDDs

Outer joins (**leftOuterJoin/rightOuterJoin**)

- Guarantees that on the RDD all they keys (left or right) will be present
- Keys that do not appear on the other RDD have a *None* value.

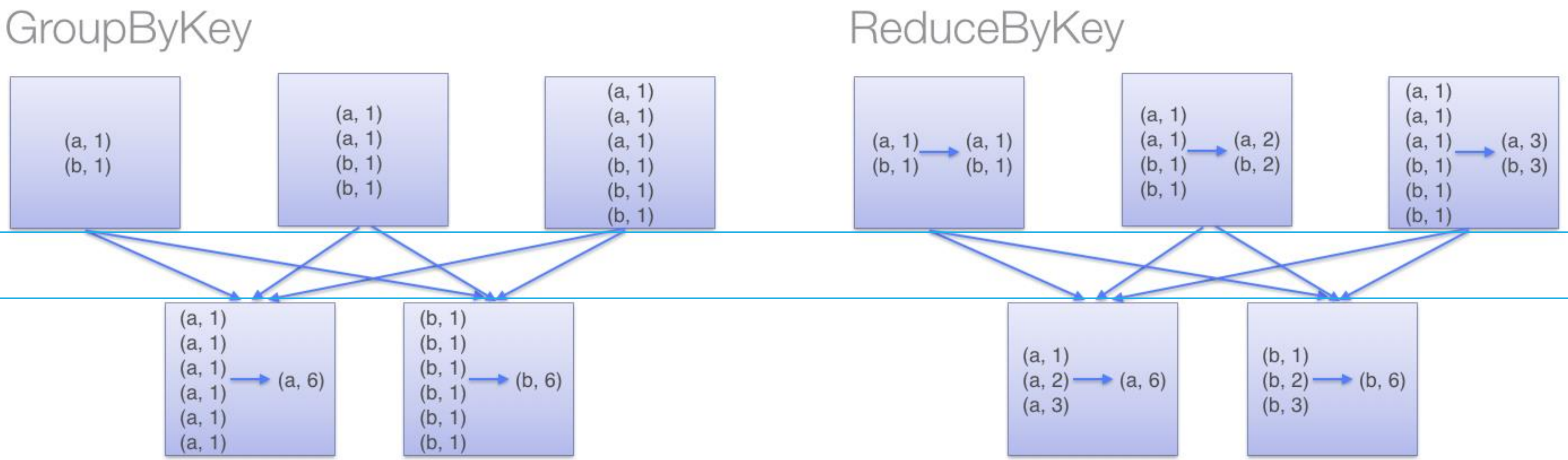
Shuffling and Partitioning

```
Help on method parallelize in module pyspark.context:

parallelize(c, numSlices=None) method of pyspark.context.SparkContext instance
    Distribute a local Python collection to form an RDD. Using xrange
    is recommended if the input represents a range for performance.
```

Partitioning

Partitioning

Shuffle



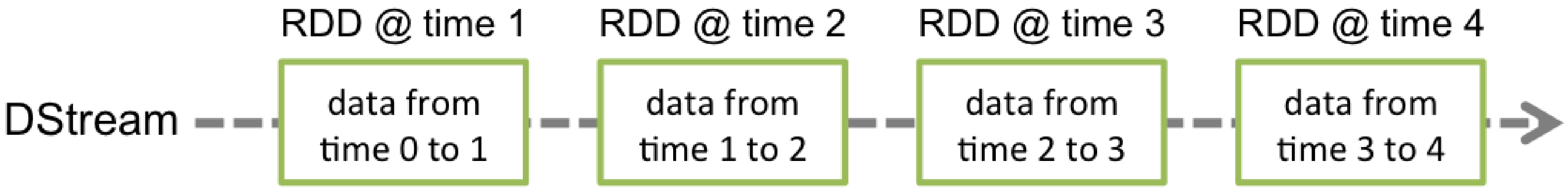
Shuffle and partitioning is expensive! As it they have to send data in the network

Spark Streaming

Motivation

- Big Data never stops
  - Data is being produced all the time

Spark provides a DStream to handle sources that send data constantly



DStream

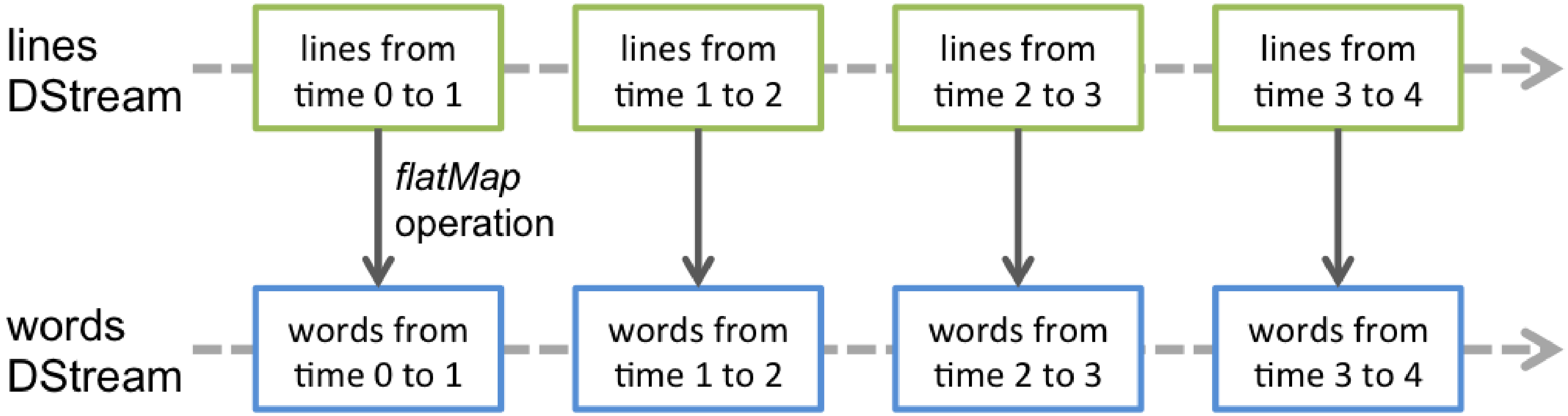
Discretized Stream represents a continuous stream of data

Input data stream is received from source, or the processed data stream generated by transforming the input stream. Internally

DStream is a continuous series of RDDs

Each RDD in a DStream contains data from a certain interval

Similarly, we can apply transformations and actions to Dstreams.



As a summary

Spark is a distributed big data processing framework.

- Distribution brings new concerns: Node failure and latency

Uses Resilient Distributed Datasets (RDD) to distribute and parallelize the data.

- RDDs are ***lazily-created*** and ***ephemeral***
- Caching and persistence is used to preserve a RDD in memory, disk, or both

RDDs are fault tolerant

- Able to recover the state of an RDD using **coarse-grained transformations** and **lineage.**

Transformations are **lazy** (e.g., map, filter, groupBy, sortBy, reduceByKey)

Actions are **eager** (e.g., take, collect, reduce, first, foreach)

The topology of the cluster matters

Working with RDDs implies **shuffling** and **partitioning**

- Impact on performance due to latency

Spark provides Big Data Streaming processing via **DStreams**

That’s all for now!

Tutorial: Batch and streaming processing with PySpark
Monday, 19 March, 14:15 » 16:00,
T2 / C105 (T2), Tietotekniikka, Konemiehentie 2

Thanks!

Questions?

Frederick Ayala-Gómez
frederick.ayala@aalto.fi

Credits and References

- Slides from Michael Mathioudakis from previous Aalto's Modern Database Systems Course.
- https://spark.apache.org/docs/2.2.0/streaming-programming-guide.html
- Big Data Analysis with Scala and Spark, Dr. Heather Miller, École Polytechnique Fédérale de Lausanne.
- Zaharia, Matei, et al. "Learning Spark: Lightning-Fast Big Data Analysis". O'Reilly Media 2015.
- Zaharia, Matei, et al. "Spark: Cluster Computing with Working Sets." *HotCloud* 10 (2010): 10-10.
- 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.*
- Learning Spark: Lightning-Fast Big Data Analysis, by Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia