Eötvös Loránd University (ELTE)
Faculty of Informatics (IK)
Pázmány Péter sétány 1/c
1117 Budapest, Hungary

DATA ANALYSIS SOLUTIONS: SPARK

Open-source Technologies for Real-Time Data Analytics

Imre Lendák, PhD, Associate Professor

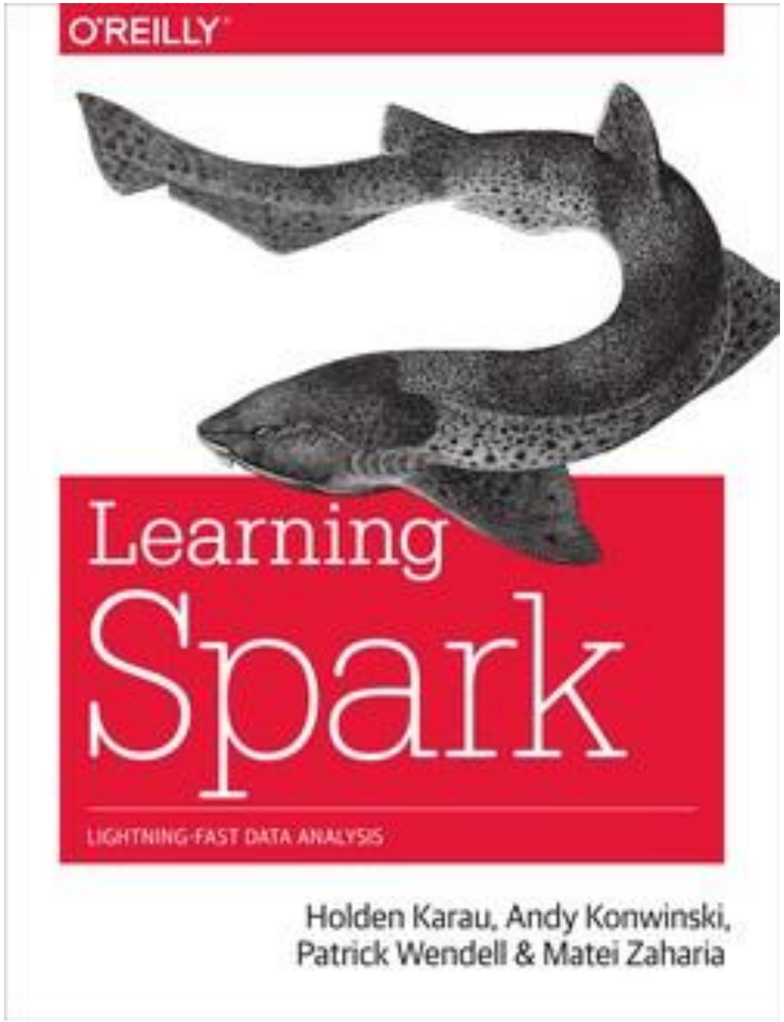
2020
Budapest, Hungary

Chosen data analytics topics

- **MapReduce** implements the map & reduce paradigm known from functional programming
  - Discussed in last time

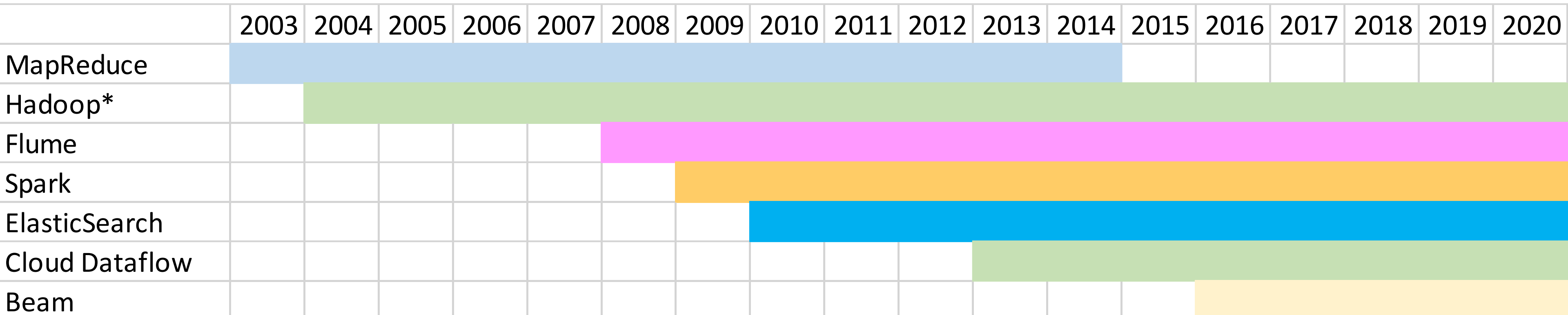
- Apache Spark is an open-source, distributed, general-purpose cluster-computing framework
  - Discussed in this lecture!

- **ElasticSearch** search & analytics engine
  - Discussed later



Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia, "Learning Spark: Lightning-Fast Big Data Analysis", O'Reilly, 2015.

Data analysis timeline



* Analysis elements of the Hadoop ecosystem

Survey: Spark (Streaming) XP

Attempts: 40 out of 40

+0.59

Discrimination Index ?

Please rate you past experience in using Spark Streaming:

No prior experience	32 respondents	**80 %**
Heard/learned about it in an online or university course	4 respondents	10 %
My course project team used it	2 respondents	5 %
Used it myself in a course project	2 respondents	5 %
Used it professionally, i.e. in a for money project		0 %

80% answered correctly

Spark intro

Introduction & history

Definitions

- **DEF:** Apache Spark is an open-source, distributed, general-purpose **cluster-computing framework**
- The authors aimed to perform **in-memory** calculations in computing clusters without ‘touching the disk’ before reaching the final data processing stage (i.e. output)
- Written in Scala
- Additionally optimized for interactive queries and iterative computing jobs

History

- Originally developed by the AMPLab at UC Berkeley around 2009
- As soon as 2009 it was outperforming MapReduce 10-20x in certain types of problems
- Open sourced in 2010 (BSD license)
- Donated to the Apache Software Foundation in 2013
- Top-level Apache project since 2014

https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

Open-source permissions and limitations

- **Linking** - linking of the licensed code with code licensed under a different licence (e.g. when the code is provided as a library)
- **Distribution** - distribution of the code to third parties
- **Modification** - modification of the code by a licensee
- **Patent grant** - protection of licensees from patent claims made by code contributors regarding their contribution, and protection of contributors from patent claims made by licensees
- **Private use** - whether modification to the code must be shared with the community or may be used privately (e.g. internal use by a corporation)
- **Sublicensing** - whether modified code may be licensed under a different licence (for example a copyright) or must retain the same licence under which it was provided
- **TM grant** - use of trademarks associated with the licensed code or its contributors by a licensee

https://en.wikipedia.org/wiki/Comparison_of_free_and_open-source_software_licences

Open-source licenses compared

Licence	Author	Latest version	Publication date	Linking	Distribution	Modification	Patent grant	Private use	Sublicensing	TM grant
Academic Free License[11]	Lawrence E. Rosen	3.0	2002	Permissive	Permissive	Permissive	Yes	Yes	Permissive	No
Affero General Public License	Affero Inc	2.0	2007	Copylefted[12]	Copyleft except for the GNU AGPL[12]	Copyleft[12]	?	Yes[12]	?	?
Apache License	Apache Software Foundation	2.0	2004	Permissive[13]	Permissive[13]	Permissive[13]	Yes[13]	Yes[13]	Permissive[13]	No[13]
Apple Public Source License	Apple Computer	2.0	August 6, 2003	Permissive	?	Limited	?	?	?	?
Artistic License	Larry Wall	2.0	2000	With restrictions	With restrictions	With restrictions	No	Permissive	With restrictions	No
Beerware	Poul-Henning Kamp	42	1987	Permissive	Permissive	Permissive	No	Permissive	Permissive	No
BSD License	Regents of the University of California	3.0	?	Permissive[14]	Permissive[14]	Permissive[14]	Manually[14]	Yes[14]	Permissive[14]	Manually[14]

https://en.wikipedia.org/wiki/Comparison_of_free_and_open-source_software_licences

Whois AMPLab?

AMPLab

- AMP = Algorithms, Machines and People Lab
- Doing research and publishing scientific publications since 2008
- AMPLab officially launched in 2011
- Worked on different ‘big data’ projects under the Berkeley Data Analytics Stack (BDAS)
- UC Berkeley launched RISELab as the successor to AMPLab in 2017

Best known projects

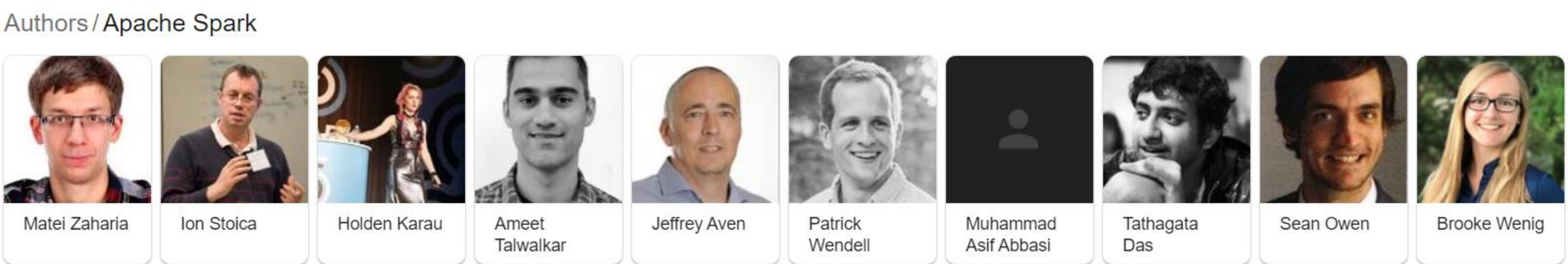
- **Apache Spark** – distributed, general-purpose computing platform
- **Apache Mesos** – cluster management platform
- **Alluxio** – virtual distributed file system (VFDS) – Alluxio ‘sits’ between computation & storage in large-scale data processing environments. Used by Cray, IBM, Lenovo, Intel, etc.

RISELab in November 2020

https://rise.cs.berkeley.edu

Spark authors

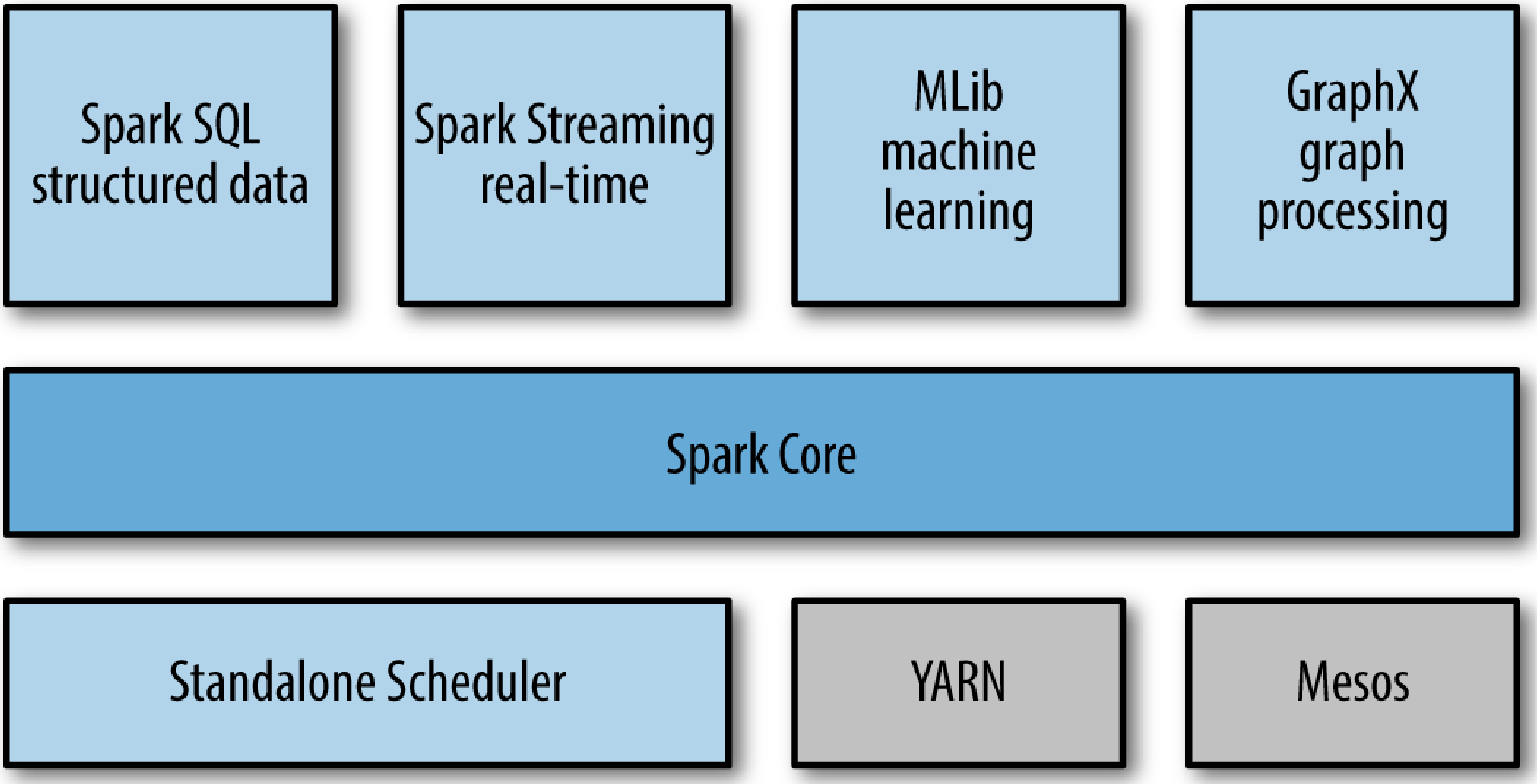
- PhD students at UC Berkeley
  - Matei Zaharia – Spark founder, Databricks CTO and professor at Stanford
  - Benjamin Hindman (Mesos)
  - Andy Konwinski (Mesos, Spark)
  - Haoyuan Li (Alluxio)
- Uni-based research supervised by professor Ion Stoica
- Developers: Holden Karau, Patrick Wendell, Andy Konwinski, …



* Image source: www.google.com search

SPARCHITECTURE

The Spark stack



Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia, “Learning Spark: Lightning-Fast Big Data Analysis”, O’Reilly, 2015.

Components

- **Spark Core** tasks:
  - Memory management
  - Fault recovery
  - Implements the RDD (v1.0) and Dataset (v2+) Application Programming Interfaces (RDD API vs Dataset API)
- **Spark SQL** is Spark's package for working with structured data.
- **Spark Streaming** is a consistent micro-batch processing environment for live streams of data.
- **MLlib** provides multiple types of machine learning algorithms, e.g. classification, regression, clustering
- **GraphX** is a library for manipulating graphs
- Cluster management via YARN, Mesos or Spark's own Standalon Scheduler

Sparchitecture: Resilient Distributed Dataset (RDD)

RDD intro

- **DEF:** Resilient Distributed Datasets (RDDs) are data items distributed over a cluster of machines and maintained in a fault-tolerant way
  - RDDs are essentially a restricted form of distributed shared memory
  - RDDs can contain different data types, not just (key, value) pairs as in MapReduce
  - Java objects are kept in memory deserialized
  - Python objects are ‘pickled’

Resilient Distributed Datasets (RDD)

Abstraction

- Partitioned collection of records
  - Data is spread across the cluster
  - RDDs are read-only, i.e. no in place updates
- Caching dataset in memory (if able)
  - different storage levels available
  - fallback to disk possible

Operations

- *Transformations* create new RDDs from existing RDDS
  - *map*, *filter*, *join*
  - Lazy operation
- *Actions* return a value to the Spark application or export data
  - Actions include *count*, *collect*, *etc.*
  - Triggers execution

Spark inputs & outputs

- Input/output **file formats**: text, JSON, CSV, sequence files & object files
  - File compression, e.g. gzip
  - Filesystems: local, HDFS, Amazon S3
- **Protocol buffers** are a fast, space-efficient multilanguage format
  - Originally developed at Google, open source, structured data, fields and types well-defined
- **RDBMS** accessed via JDBC
- **Distributed data stores**: Cassandra, HBase, Elasticsearch
- Note: **Broadcast variables** allow Spark applications to send shared, mid-size, static data to all worker nodes
  - The data for the broadcast variable is loaded from storage by the driver and sent out to all workers (e.g. lookup data)

Partitions

- In some Spark applications an RDD is scanned multiple times
  - In those cases it is useful to control the dataset's partitioning across the nodes
  - Partitioning is usually controlled with RDDs of key-value pairs → a set of keys are stored together on a node
- Operations benefiting from partitioning: various joins (e.g. with smaller lookup table), group by, reduce by, etc.
- Not all transformations set a partitioner for RDDs → e.g. general map calls might modify the keys → reset partitioner
- **Note:** after an explicit re-partitioning of an RDD it should be persisted as otherwise the RDD would be re-evaluated and repartitioned on each action

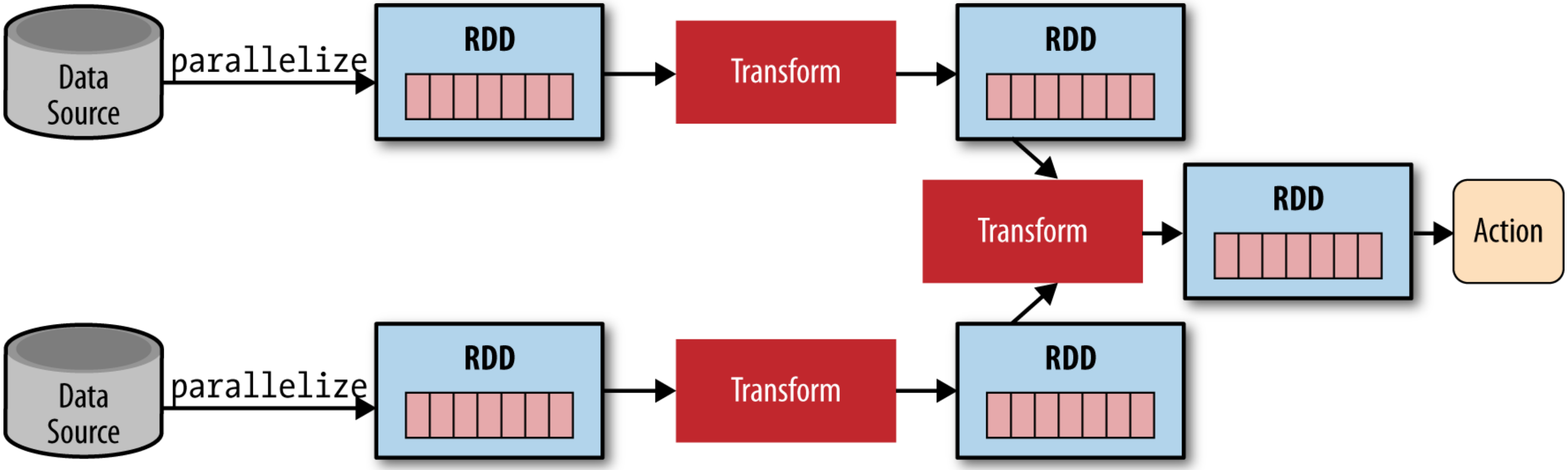
Storage levels and caching

- By default, Spark recomputes the RDD and its dependencies each time an action is called
- Spark can be told to persist an RDD, i.e. nodes which computed an RDD are asked to persist their partitions
- Off-heap caching can be implemented in the Alluxio data orchestrator, https://www.alluxio.io

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	Hi	Lo	Yes	No	
MEMORY_ONLY_SER	Lo	Hi	Yes	No	
MEMORY_AND_DISK	Hi	Med	Some	Some	Spills to disk if insufficient memory
MEMORY_AND_DISK_SER	Lo	Hi	Some	Some	Serialized objects in memory, spills to disk
DISK_ONLY	Lo	Hi	No	Yes	

Processes: Data analytics

Data processing steps



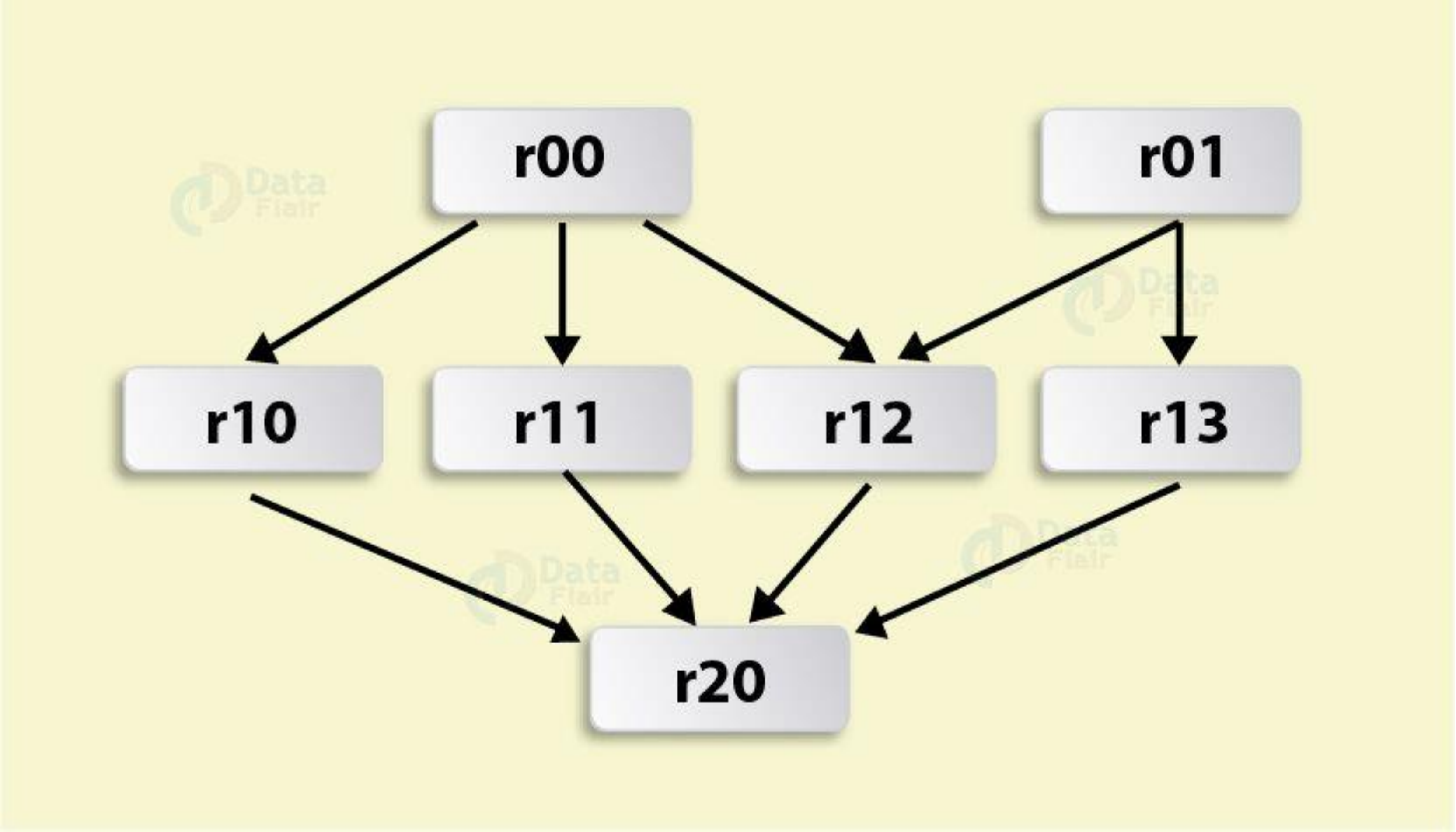
- Operation types on RDDs:
  - transformations and
  - actions

https://www.oreilly.com/library/view/data-analytics-with/9781491913734/ch04.html

Transformations

- **DEF: Spark transformations** are operations which read RDDs as inputs and produce RDDs as outputs
  - Transformations do not mutate input RDDs → they just produce new output RDDs and return a pointer to it
  - Many, but not all transformations are element-wise, i.e. they operate on the elements of the input RDDs in sequence
  - Transformations can operate on one (e.g. filter()), two (e.g. union()) or more input RDDs
- **DEF:** The **Spark lineage graph** is the set of dependencies between RDDs
  - Lineage graphs are maintained for each Spark application separately
  - The lineage graph is used to re-computer RDDs on demand and to recover lost data if parts of a persisted RDD are lost
  - **Note:** be careful and do not confuse the lineage graph with the directed acyclic graph (DAG) of task execution

Example lineage graph



https://data-flair.training/blogs/rdd-lineage/

Element-wise transformations

- Most common element-wise transformations for an RDD containing [1,2,3,3]

Function name	Purpose	Example	Result
map()	Apply function to each element	rdd.map(x => x+1)	[2,3,4,4]
flatMap()	Apply function and return flat data	rdd.flatMap(x=>x.to(3))	[1,2,3,2,3,3,3]
filter()	Get RDD with elements filtered	rdd.filter(x=>x!=1)	[2,3,3]
distinct()	Remove duplicates	rdd.distinct()	[1,2,3]
sample(withRepalcement, fraction, seed)	Select sample from an RDD w or w/o replacement	rdd.sample(false, 0.5)	? (non deterministic)

Pseudo-set transformations

- Most common element-wise transformations for RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
union()	Elements from both input RDDs	rdd.union(other)	{1, 2, 3, 3, 4, 5}
intersection()	Elements found in both RDDs	rdd.intersection(other)	{3}
subtract()	Remove contents of one RDD	rdd.subtract(other)	{1, 2}
cartesian()	Cartesian product	rdd.cartesian(other)	{(1, 3), (1, 4), … (3,5)}

- The Cartesian product for sets A and B is denoted with $A \times B$. It is the set of all ordered pairs (a,b) where a is in A and b is B

Actions

- **DEF:** **Spark actions** are operations which make some calculation and return the result to the driver or persist it in external storage
- Actions force the evaluation of all (upstream) transformations in the lineage graph of the RDD they are called on
- Each different action forces the evaluation of upstream transformations unless the intermediate RDDs are persisted (which is not default behavior)
- The simplest actions are count(), take() and collect()
- **Note:** be careful when calling collect() as its return value should be limited in size and able to fit in the driver's memory → usually called during testing and/or when the transformations result in RDDs of limited size(s)

Basic Spark actions

- Basic actions on an RDD containing {1, 2, 3, 3}

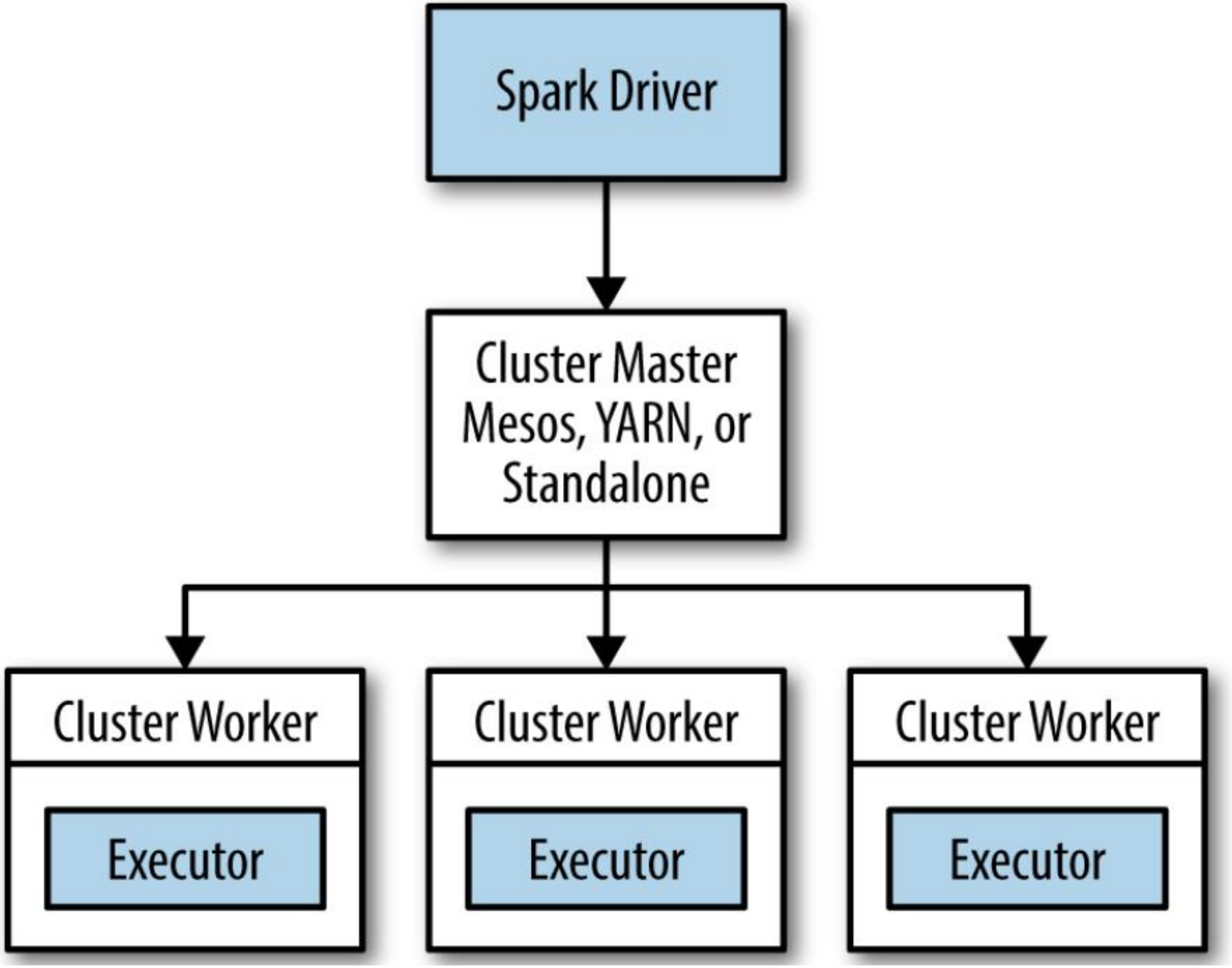
Function name	Purpose	Example	Result
collect()	Retrieve all elements	rdd.collect()	{1, 2, 3, 3}
count()	Number of elements	rdd.count()	4
countByValue()	Number of each unique element	rdd.countByValue()	{(1, 1), (2, 1), (3, 2)}
take(num)	Return ‘num’ elements from the RDD	rdd.take(2)	{1, 2}
top(num)	Return ‘num’ top elements from RDD	rdd.top(2)	{3, 3}
reduce(func)	Combine RDD elements with function ‘func’	rdd.reduce((x, y) => x + y)	9
fold(zero)(func)	Same as reduce, but with zero value	rdd.fold(0)((x, y) => x + y)	9

Lazy evaluation

- Transformations on RDDs are lazily evaluated → Spark will not begin to execute transformations until it sees an action
  - Many transformations can be chained together and none will execute until an action generating an output is ‘seen’
- Instead of immediate execution, Spark does the following:
  - Internally **record metadata** about transformation requests → this in essence means that in-memory RDDs can be regarded as instructions for computing data instead of data itself (which is not materialized immediately)
  - **Lazy data load**, i.e. actual data read and parallelize will be executed when needed to perform an action downstream
- Lazy evaluation allows Spark to optimize data processing pipelines inline, transparently to the user, thereby reducing the number of passes over the data

Processes: Spark applications

Anatomy of a Spark application



The Driver

- The driver is the center of a Spark application
- The main() method is in the driver
- It runs the user code
- Creates the Spark context
- Loads the input RDDs
- Performs transformations and actions

Driver duty #1: Create tasks

- **DEF:** Spark tasks are the smallest units of physical execution (inside the computing cluster)
- The driver converts the user's program into tasks
- A Spark program implicitly creates a logical directed acyclic graph (DAG) of operations
- Drivers convert the DAG into physical execution plans, pipeline transformations and merge them where able
- The DAG is converted into a set of 'stages'
- Each stage consist of multiple tasks
- Tasks are sent to the cluster for execution

Driver duty #2: Task scheduling

- Tasks are scheduled on individual ‘executors’
- Executors register with the driver when started
- Drivers analyze their current sets of executors and schedule tasks based on data placement
  - This is known in MapReduce as data locality
- When tasks execute, they produce intermediate data which can be cached, e.g. persisted RDDS
  - The driver tracks the location of cached data and schedules additional tasks which use the cached data
- The driver exposes information about the Spark application’s status via a web interface (usually HTTP on port 4040)

Spark executors

- Executors are worker processes running tasks
- Key executor roles:
  - Run tasks and return intermediate results to the driver
  - Provide in-memory storage for RDDs (this is done by the Block Manager process)
- Executors are launched when a Spark application is started
- Their lifetime is usually equal to the Spark app's

Launching a Spark application

- Spark applications are launched via the **spark-submit** script
  - It connects to the various supported cluster managers and controls resource usage
  - It launches the driver and invokes main()
- The **driver** contacts the cluster manager to acquire resources (CPU, memory) and run tasks
- The **cluster manager** launches executors on behalf of the driver
- Task are run in **executor** processes to compute and save results
- The **Spark app ends** when main() ends or when the Spark context is explicitly stopped from (user) code

Synchronization and scheduling

Scheduling and cluster mgmt

- Scheduling and cluster management is performed with
  - Spark's built-in Standalone Cluster Manager
  - Apache YARN
  - Apache Mesos

https://www.cartoonstock.com/directory/o/oars.asp

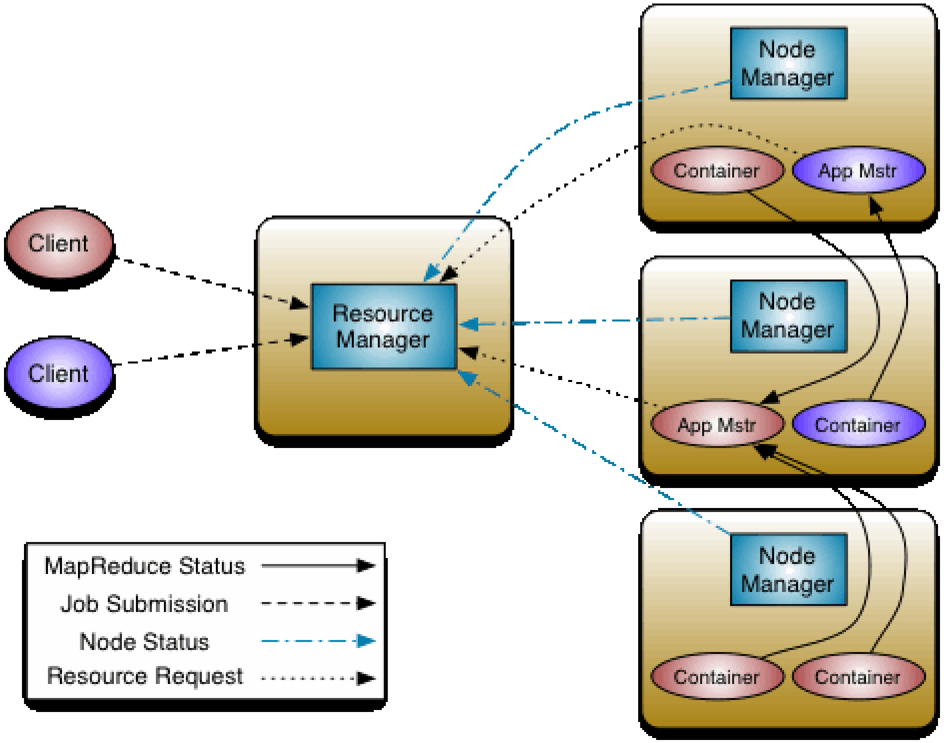
Standalone Cluster Manager

- The Standalone Cluster Manager (SCM) consists of a master and multiple workers
- Workers are assigned configured amounts of memory and CPU cores
- The SCM is by default available on the following URI: spark://masternode:7077
- The SCM's web UI is usually accessible via http://masternode:8080
- The SCM supports 2 deploy modes:
  - Client: the driver runs on the machine where spark-submit is run
  - Cluster mode: the driver is launched on one of the worker nodes

- **Note:** The SCM is a good fit when the computing cluster is not shared with other users and or computing platforms

Apache YARN

- The YARN cluster manager was introduced with Hadoop v2
- It runs on the HDFS nodes → YARN is good scheduling choice if the data consumed by the Spark application is stored in HDFS



Apache Mesos

- **DEF:** Apache Mesos is a general-purpose cluster manager
- Mesos can run both analytics workloads and long-running services
- Mesos clusters can also use ZooKeeper to elect a master when running in multi-master node
- Mesos modes: fine-grained & coarse-grained

Cluster manager comparison

- The **Standalone Cluster Manager** is easiest to set up and is a good choice with dedicated clusters, i.e. when running only Spark on a set of compute nodes
- **YARN** is a good choice when Spark is run on a (shared) cluster where we already have Hadoop installed, e.g. when the data is stored in HDFS
- **Mesos** is attractive (compared to the SCM & YARN) when running multiple interactive user sessions, as it can scale up & down resource use (CPU & memory) between commands issued in a user session
- **Note:** In all cases, it is a good idea to design the Spark cluster with **data locality** in mind, i.e. to deploy the executors as close to the data as possible

Fault tolerance

Faults in the scheduler

- **Executor node failure:** If a node (e.g. a single server computer) fails, its tasks are re-run on a different node and the affected RDD partitions are re-computed based on the lineage graph
- **Driver node failure:** If the node running the driver, or the driver code fails, the Spark context is lost → re-launch the Spark application, re-start the driver and all executors
  - With file-based inputs this does not result in data loss → everything is re-computed
  - With input streams, buffered data would be lost in the executors → Spark 1.2+ have write-ahead logs
- **Cluster manager failure:** The Standalone Cluster Manager, YARN or Mesos can be run in hot-standby mode via the Apache Zookeeper distributed coordination system → this ensures that the Spark cluster will not fail when a single cluster manager fails

https://stackoverflow.com/questions/24762672/how-does-apache-spark-handles-system-failure-when-deployed-in-yarn
https://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html

Spark summary

- Spark intro
- Sparkitechture
- Resilient Distributed Dataset (RDD)
- Processes
  - Data analytics
  - The Spark cluster
- Synchronization and scheduling
- Fault tolerance

Thank you for your attention!