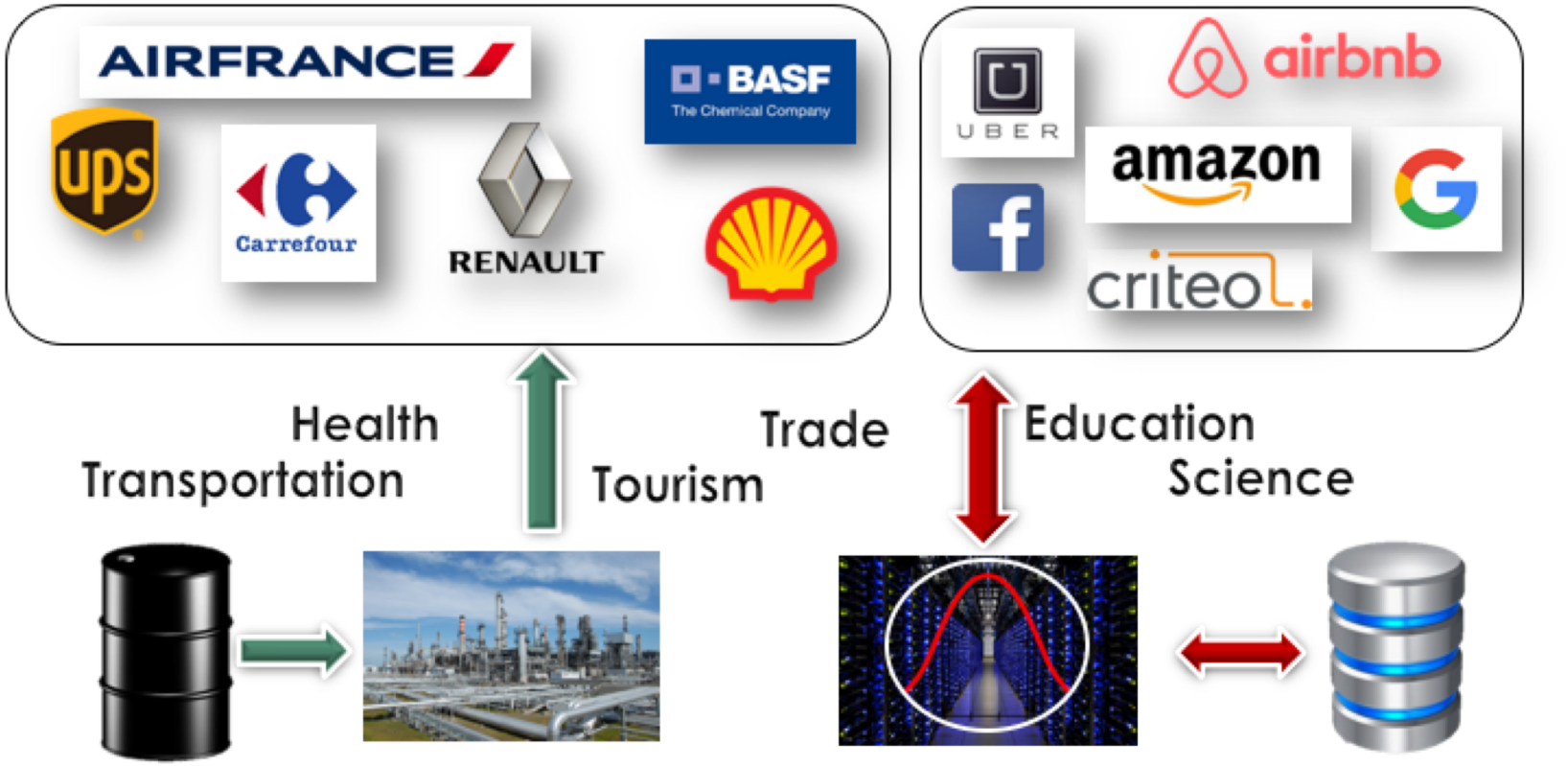
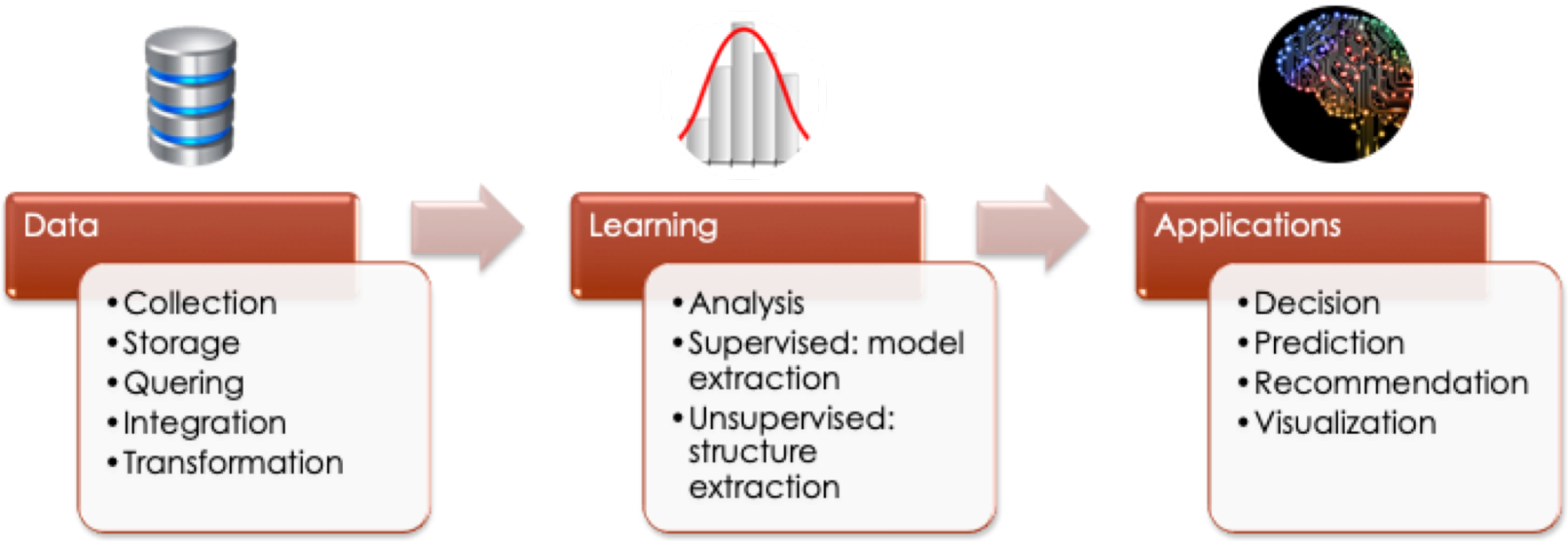
Apache Spark DataFrames

Source: Cloud Computing et Big Data -- FC Machine Learning et IA (B. Amann)

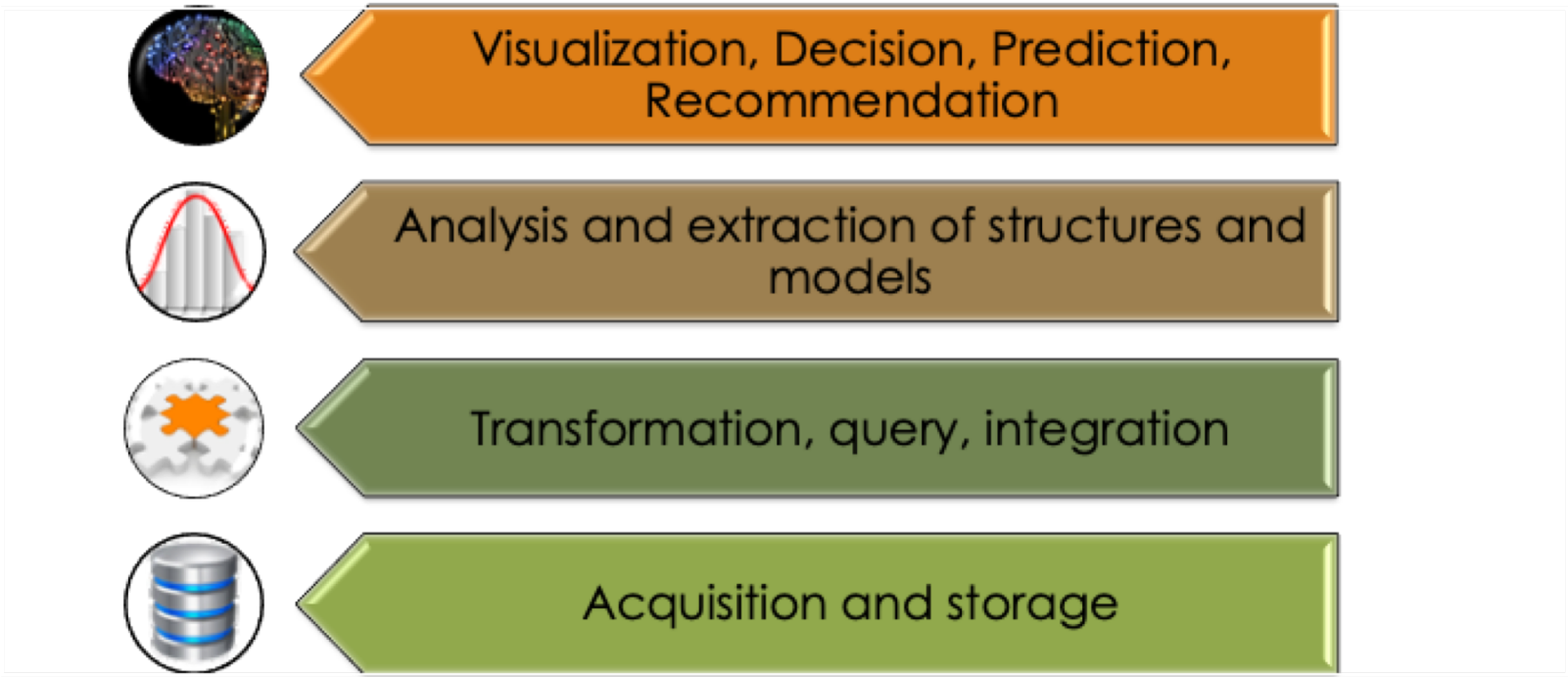
Data gathering and processing



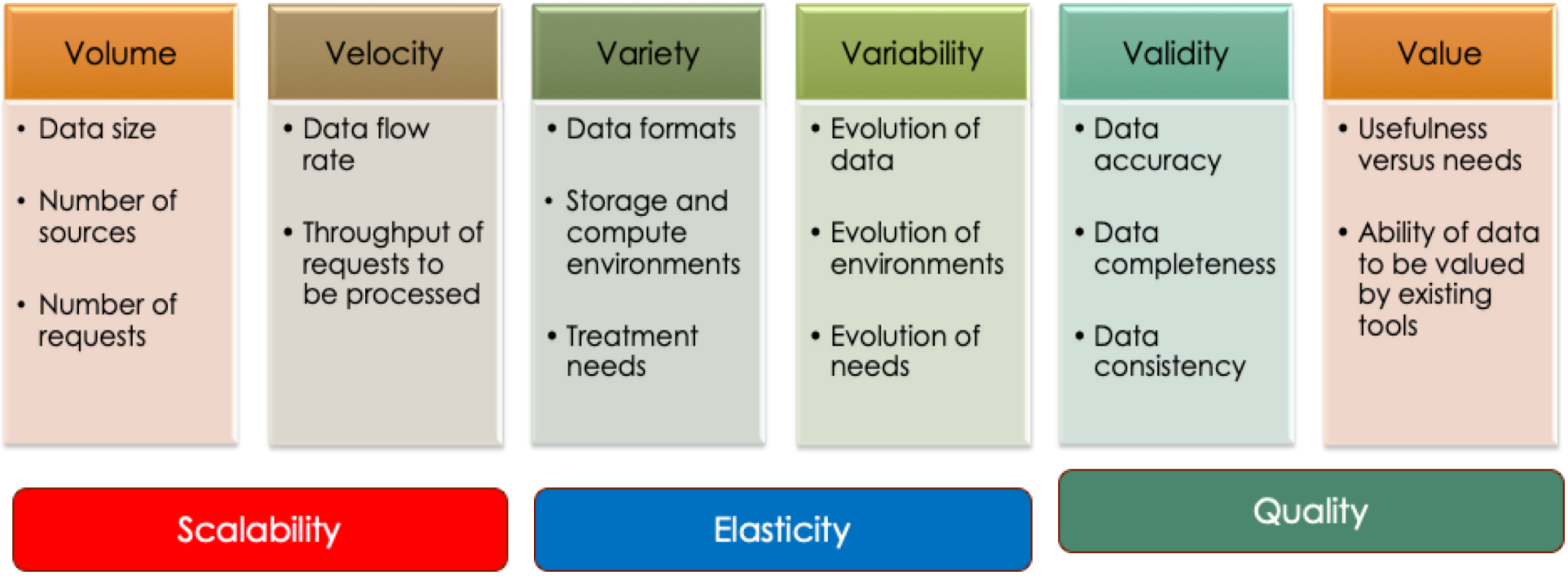
Big Data, Data Sciences and AI



Big Data Stack



The Challenges of Big Data: The 6 Vs



We need to provide solutions (languages, models, systems, hardware) that meet these new needs.

The Challenges of Big Data Systems

Server Systems

- Vertical architecture
- Central servers with large storage and computing capacities
- Separation between the compute layer and the storage layer

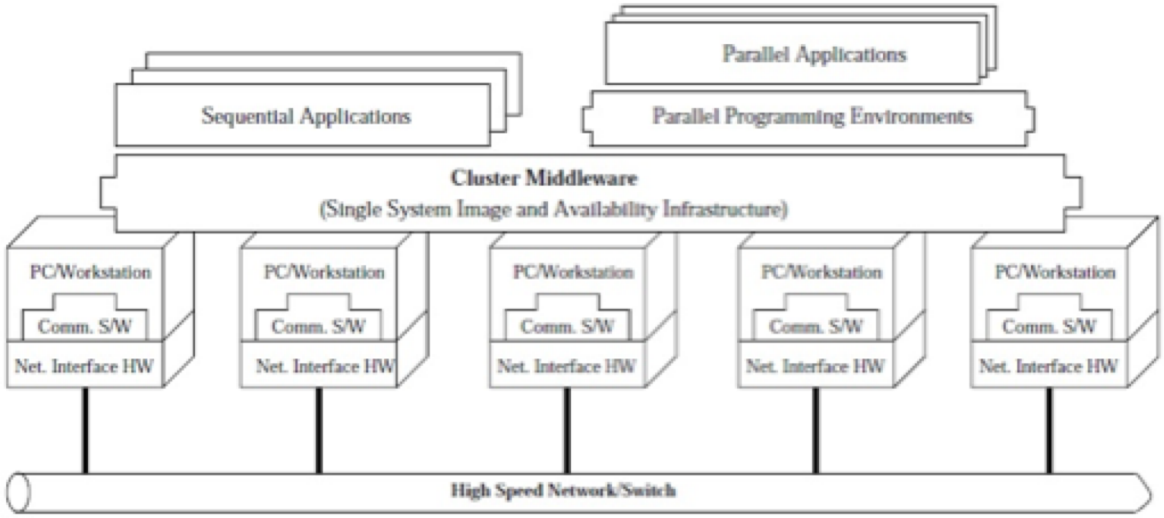
- Difficult to adapt to changing data and needs
- Low cost/benefit ratio

Big Data Systems

- Parallel architecture
- Set of independent computing and storage nodes
- Virtualization of resources and services

- Easy to adapt
- Cost proportional to performance benefit

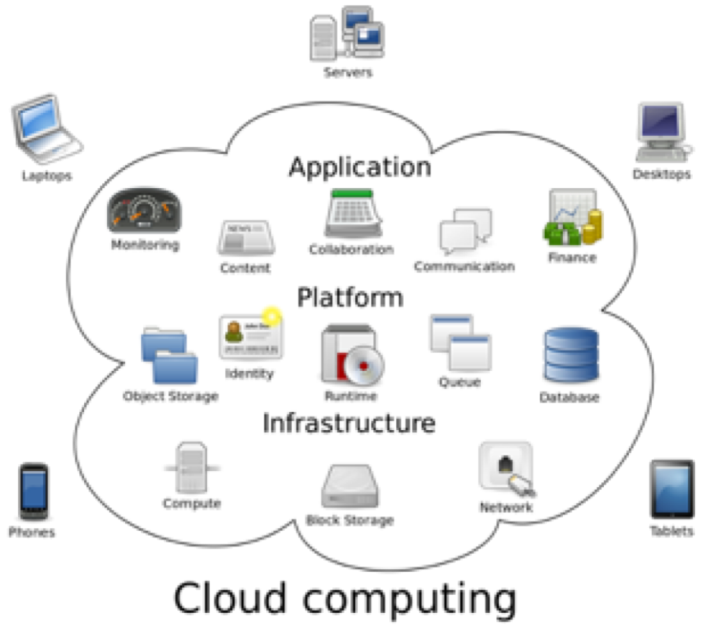
Parallel architecture (cluster)



- Independent computing and storage nodes
- Very high speed network
- Coordination by middleware

Scalability

Virtualization (cloud)

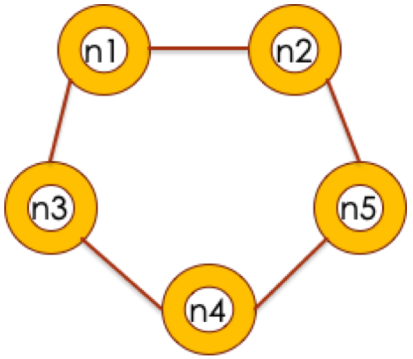


- separation between a function and its execution environment
- Infrastructure (IaaS)
- Platform (PaaS)
- Services (SaaS)

Elasticity

Challenges and Choices: The CAP theorem

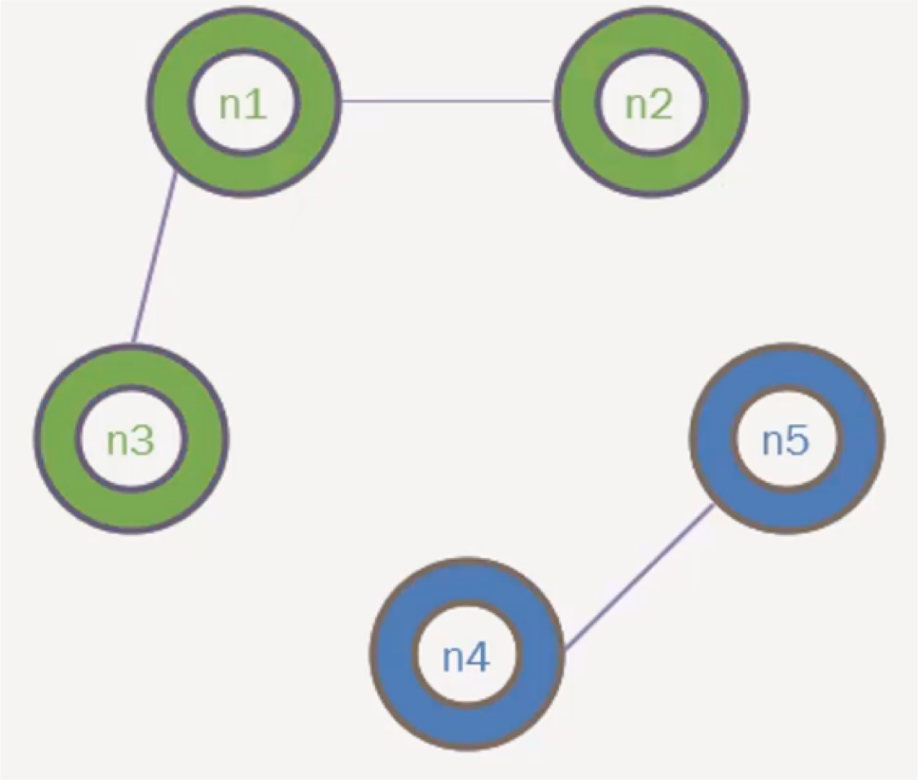
Eric Brewer, PODC 2000



A distributed system can only guarantee two of the following three constraints:

- **Consistency** : each query receives the most recent data or an error;
- **Availability** : all requests are answered, with no guarantee that it is the most recent data;
- **Partition Tolerance** : No failure less than a complete network outage prevents the system from responding properly.

The CAP theorem



- Fault tolerance is not guaranteed

➔ you have to choose between **availability** or **consistency**

- ACID:
  - **A** tomicity, **C** onsistency, **I** solation , **D** urability
  - Promotes **consistency**
  - Transactional systems (DBMS)
- BASE :
  - **B** asically **A** vailable, **Soft** state, **E** ventual consistency
  - Promotes **availability**
  - noSQL " systems

CAP and noSQL : ~~One~~ No Size Fits All

Transaction (SQL)	Interaction	Analysis

• Atomicity
• Consistency
• Isolation
• Durability
• Operations
  • Reading
  • Writing
	• Availability
	• Real time
	• Operations
	  • Reading
	  • *Insertion*
		• Batch
		• Exploration
		• Operations
		  • Reading
		  • Aggregation

Computations: Task and data parallelism

Task parallelism

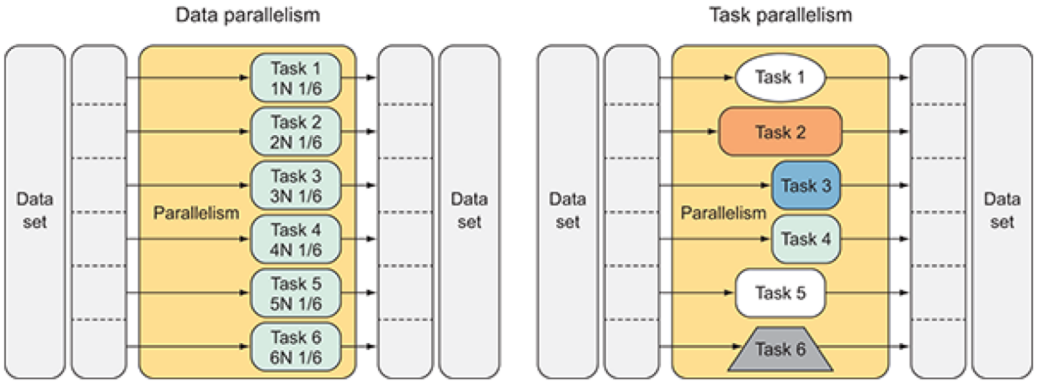
(Parallel programming)

- Distribution and execution of a set of tasks across multiple nodes (CPU /GPU + disks)
- Shared (concurrency management) and non-shared data
- Heterogeneous tasks
- Asynchronous computing
- Degree of parallelism (scalability) depends on the number of tasks
- Scheduling algorithms (static, dynamic)

Data parallelism

(Big Data Programming)

- Data fragmentation and distribution across a set of nodes (memory, disk)
- Isolated data (no competition)
- Homogeneous tasks
- Synchronous computation
- Degree of parallelism (scalability) depends on data size
- Data fragmentation and placement strategies (data locality)



Data parallelism is the simultaneous execution of the same function across the elements of a data set.

Task parallelism is the simultaneous execution of multiple and different functions across the same or different data sets.

Apache Spark

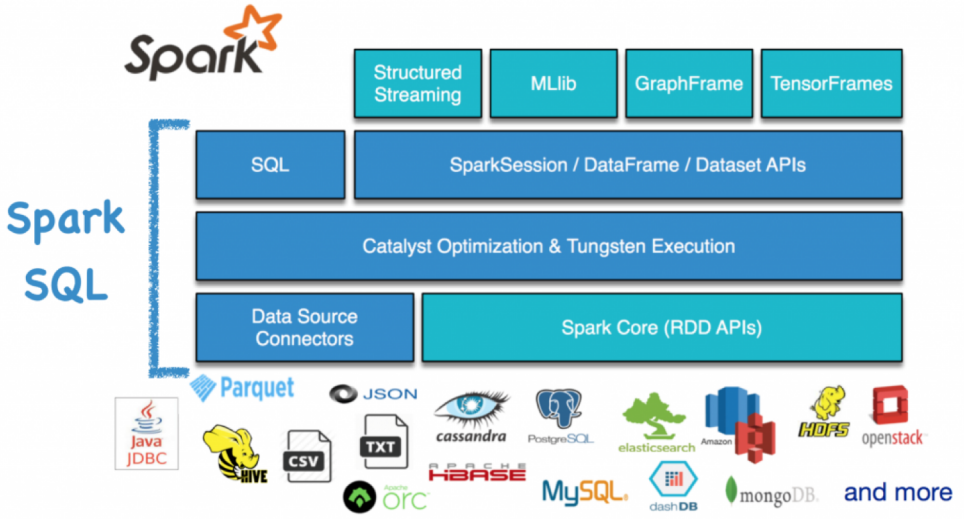
Apache Spark and the PySpark API

- *Apache Spark*
  - open-source project, unified analytics engine for large-scale data processing (combines data and AI)
  - written in Scala, provides high-level APIs in Java, Scala, Python, R, SQL
  - uses the Hadoop MapReduce distributed computing framework
  - stores and works with large volumes of data in memory (memory persistence)
  - batch processing

- *Resilient Distributed Dataset (RDD)*
  - logical collection of data distributed across multiple nodes (data that can be processed in parallel)
  - large granule treatment (no partial modification)
  - reference data on disk (HDFS, S3, etc.), data in central memory or other RDD on which it depends
  - Operations as DAGs:
    - order and relationship between operations
    - fault tolerance by re-execution of a processing chain

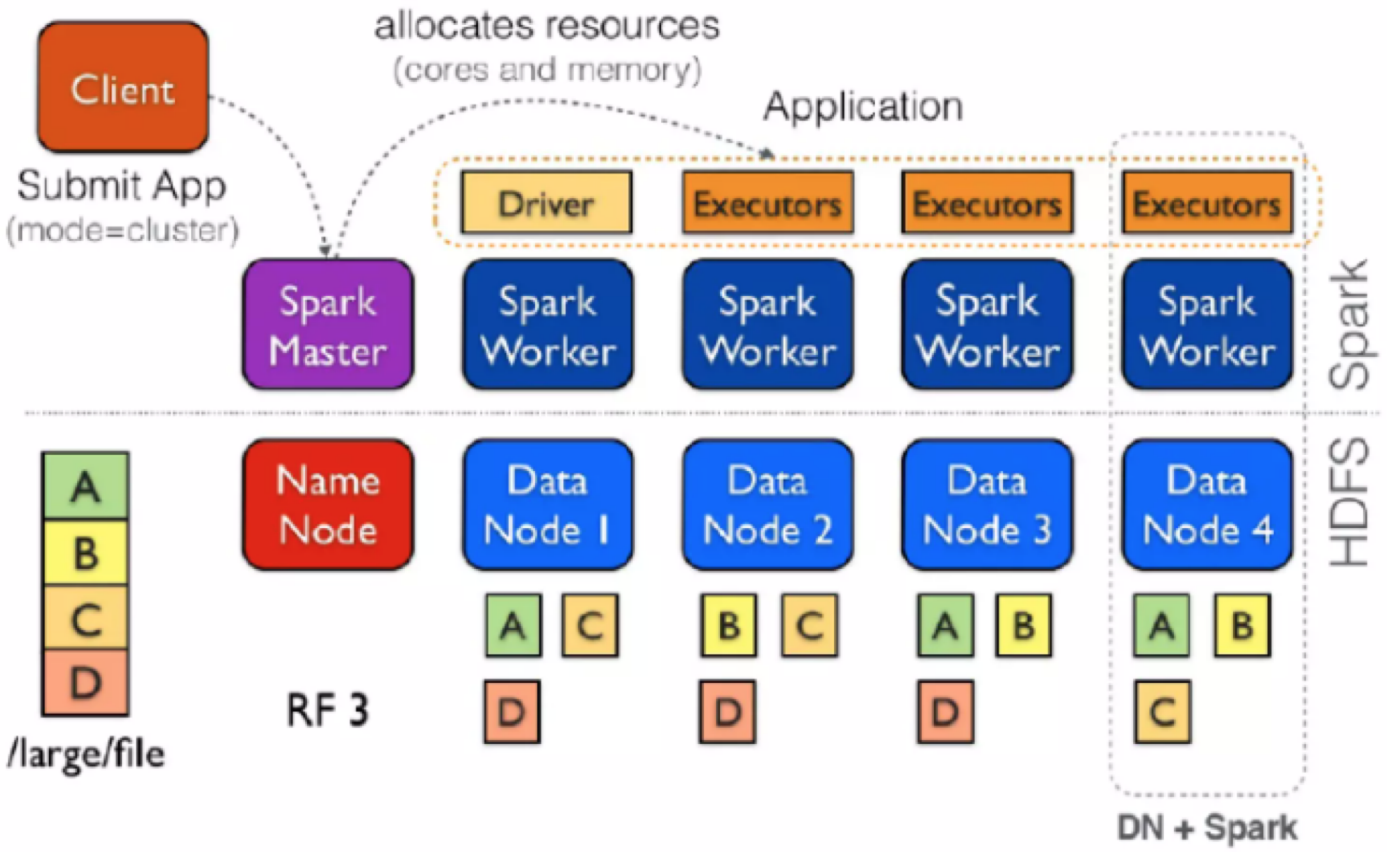
Apache Spark and the PySpark API

- Spark ecosystem
  - Spark Core
  - Spark SQL
  - Spark Streaming and Structured Streaming
  - Machine Learning Library (MLlib)
  - GraphX/GraphFrames (general graph processing)
- PySpark
  - a set of **Spark APIs** in **Python** language
  - supports all of Spark's features: Spark SQL, DataFrames, Structured Streaming, Machine Learning (MLlib) and Spark Core



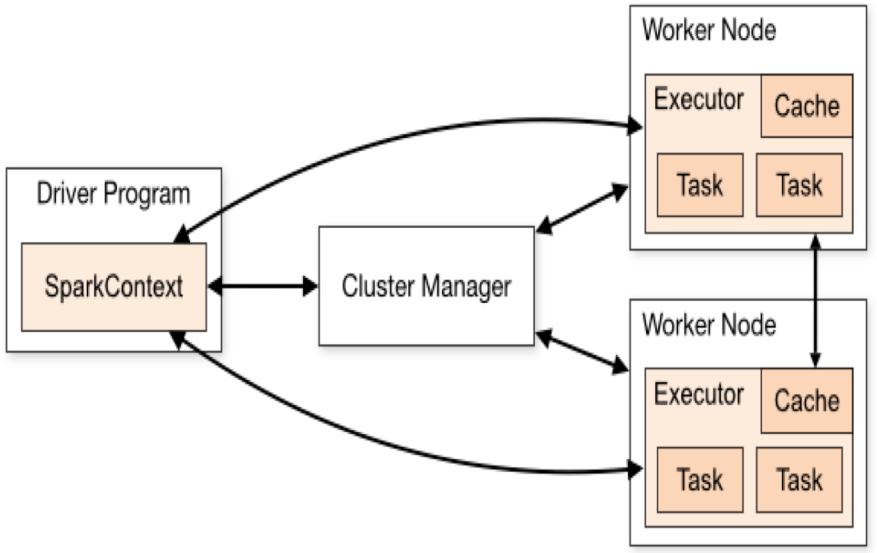
Spark: a unified pipeline

Spark Architecture



Cluster overview (Architecture of a Spark Application)

https://spark.apache.org/docs/latest/cluster-overview.html



1. Master connects to a Cluster manager to allocate resources across applications
2. Acquires Executors on cluster nodes (run compute tasks, cache data)
3. Sends app code to Executors
4. Sends tasks to Executors

Spark Context

- Tells Spark how to access the cluster
- Used subsequently to create other variables
- Master parameter for SparkContext:
  - Determines which cluster to use

master	*description*
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to # cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
`mesos://HOST:PORT`	connect to a Mesos cluster; PORT depends on config (5050 by default)

RDD (Resilient Distributed Dataset)

- Primary abstraction of Spark
- A distributed collection of data elements without any schema
- Can contain any data type (Python, Java, Scala objects, including user defined classes)
- Can handle structured and unstructured data easily and effectively

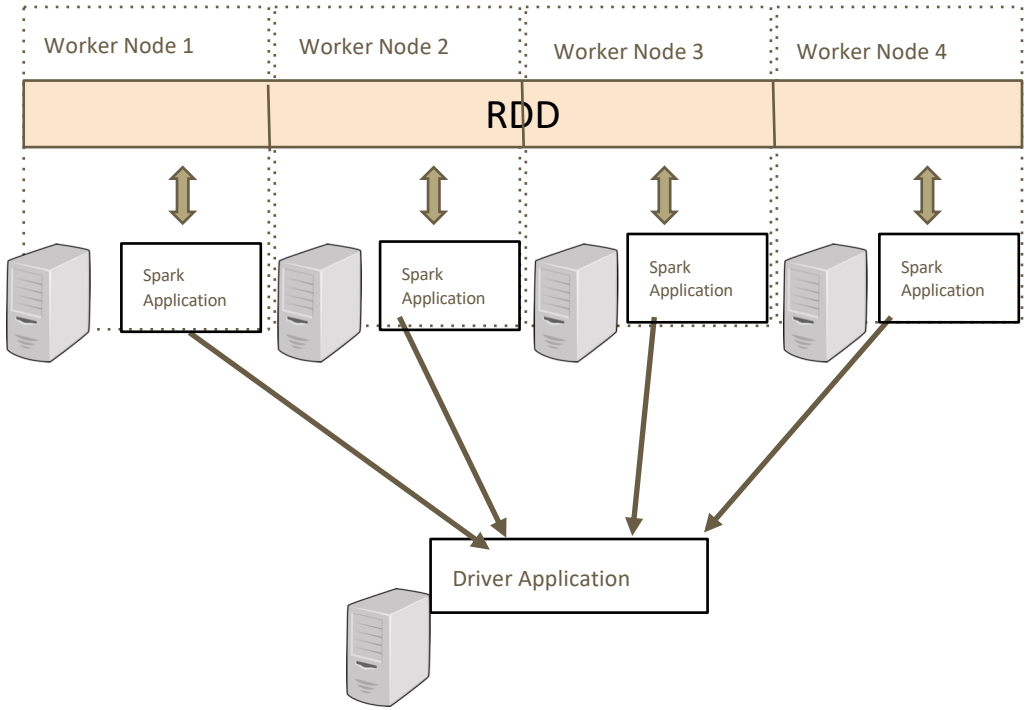
Creation of RDDs from:

- local memory
- local/distributed file/set of files
- existing database
- another RDD: transformations

Two types of operations on RDDs:

- Transformations
- Actions

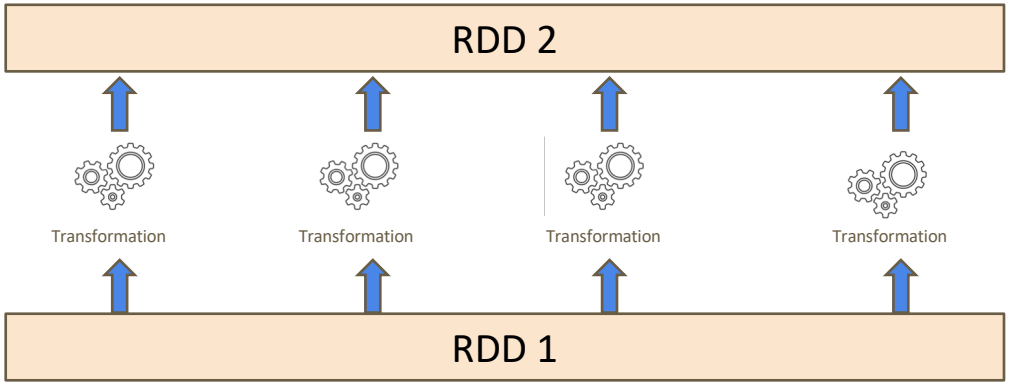
RDD: Distributed Data Abstraction



RDD: partitioned across the cluster

Computations on the RDD are executed in parallel on each partition

RDD Operations: transformations



- create a **new** RDD from an existing one
- **lazily evaluated** (not executed immediately): create a lineage of transformations
- only computed when an action requires a result to be returned to the driver program.
- Advantages:
  - optimize the required calculations
  - Recover from lost data partitions

Common transformations on RDD

- **pyspark.RDD.map:** Return a new RDD by applying a function to each element of this RDD.
  RDD.map(***f: Callable[[T], U]***, ***preservesPartitioning: bool = False***) → **pyspark.rdd.RDD[U]**

  …

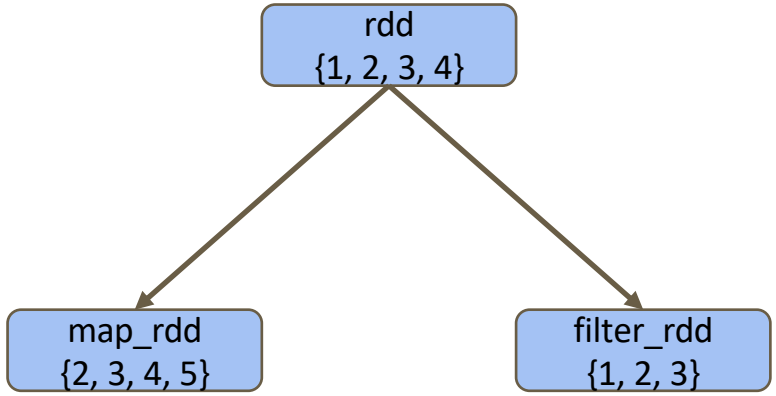
- **pyspark.RDD.flatMap:** Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.
  RDD.flatMap(***f: Callable[[T], Iterable[U]]***, ***preservesPartitioning: bool = False***) → **pyspark.rdd.RDD[U]**

  …

- **pyspark.RDD.filter:** Return a new RDD containing only the elements that satisfy a predicate.
  RDD.filter(***f: Callable[[T], bool]***) → **pyspark.rdd.RDD[T]**

Examples of transformations: map, filter

```
# import of the SparkContext library
from pyspark import SparkContext

# environment initialization (local with 4 cores)
sc = SparkContext("local[4]", "first app")

# definition of an RDD from a python array (2 partitions)
rdd =sc.parallelize([1,2,3,4],2)

# definition of the Python inc function
def inc(x): return x+1

# map transformation with the inc function
rdd_map = rdd.map(inc)

# definition of the Python test function
def test(x): return x!=4

# filter transformation with the test function
rdd_filter = rdd.filter(test)
```



Transformations on Pair RDD

- RDD[(K, V)]: can be both iterated and indexed
- **pyspark.RDD.aggregateByKey:** Aggregate the values of each key, using given combine functions and a neutral "zero value".

  aggregateByKey(*zeroValue: U, seqFunc: Callable[[U, V], U], combFunc: Callable[[U, U], U], numPartitions: Optional[int] = None, partitionFunc: Callable[[K], int] = <function portable_hash>*) → pyspark.rdd.RDD[Tuple[K, U]]

- **RDD.reduceByKey:** Merge the values for each key using an **associative and commutative** reduce function.

  reduceByKey(*func: Callable[[V, V], V], numPartitions: Optional[int] = None, partitionFunc: Callable[[K], int] = <function portable_hash>*) → pyspark.rdd.RDD[Tuple[K, V]]

- **RDD.join:** Return an RDD containing all pairs of elements with matching keys in self and other. Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in self and (k, v2) is in other.

  RDD.join(*other: pyspark.rdd.RDD[Tuple[K, U]], numPartitions: Optional[int] = None*) → pyspark.rdd.RDD[Tuple[K, Tuple[V, U]]]

Examples of transformations: reduceByKey, join

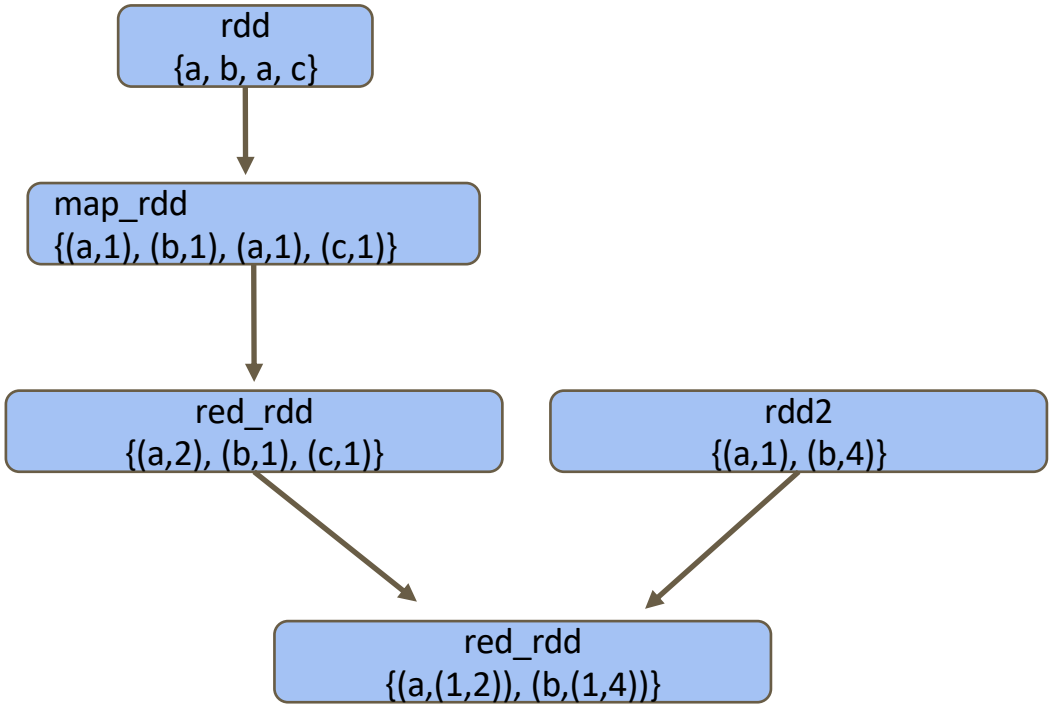
Python Functions

– def inc(x) : return x+1

Notation :

– lambda x: (x+1)

```
rdd=sc.parallelize(['a','b','a','c'], 2)

def ind(x): return (x,1)
map_rdd = rdd.map(ind)
# or
map_rdd = rdd.map(lambda x: (x,1))

def add(x,y): return x+y
red_rdd = map_rdd.reduceByKey(add)
# or
red_rdd = map_rdd.reduceByKey(lambda x,y: x+y)

rdd2 = sc.parallelize([("a", 1), ("b", 4)])
join_rdd = red_rdd.join(rdd2)
```



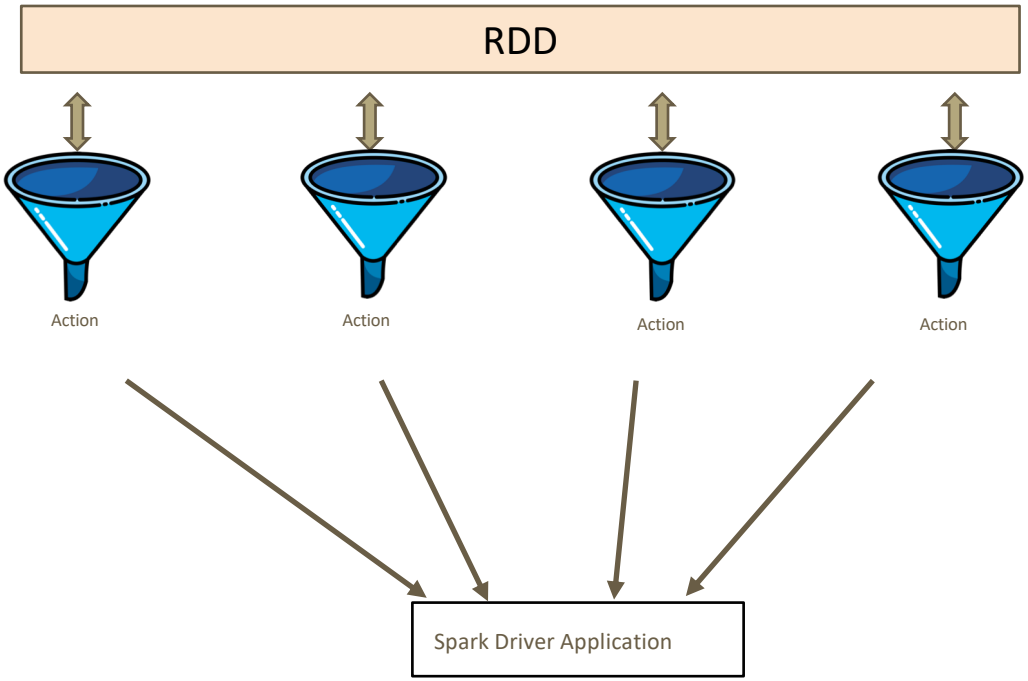
List of Transformations (1)

transformation	*description*
map(*func***)**	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func***)**	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func***)**	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement, fraction, seed***)**	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset***)**	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks***]))**	return a new dataset that contains the distinct elements of the source dataset

List of Transformations(2)

transformation	*description*
groupByKey([*numTasks*]**)**	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Seq[V])` pairs
reduceByKey(*func*, [*numTasks*]**)**	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, V)` pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending*], [*numTasks*]**)**	when called on a dataset of `(K, V)` pairs where `K` implements `Ordered`, returns a dataset of `(K, V)` pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join(*otherDataset*, [*numTasks*]**)**	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, (V, W))` pairs with all pairs of elements for each key
cogroup(*otherDataset*, [*numTasks*]**)**	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, Seq[V], Seq[W])` tuples – also called `groupWith`
cartesian(*otherDataset***)**	when called on datasets of types `T` and `U`, returns a dataset of `(T, U)` pairs (all pairs of elements)

RDD Operations: Actions



- Causes full execution of transformations, return a value to the driver program or write the data to an external storage
- Each transformed RDD may be recomputed **each time an action is run** on it (unless persisted in memory/disk/replicated across cluster)

Common actions on RDD

- **pyspark.RDD.collect:** Return a list that contains all the elements in a RDD (should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory).
  RDD.collect() → List[T]

  ...

- **pyspark.RDD.take:** Take the first num elements of a RDD.
  RDD.take(*num: int*) → List[T]

  ...

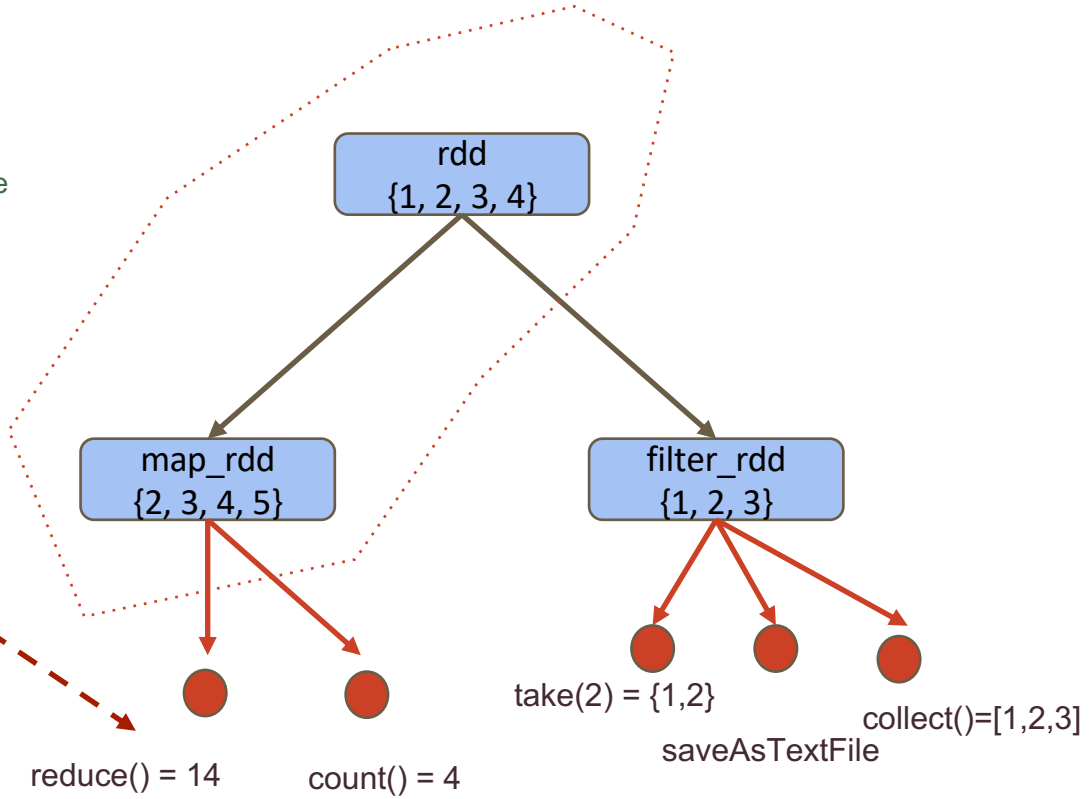
- **pyspark.RDD.reduce:** Combine all elements of a RDD to a single result of the same type, using an associative and commutative reduce function.
  RDD.reduce(*f: Callable[[T, T], T]*) → T

  ...

- **pyspark.RDD.aggregate:** Aggregate the elements of each partition, and then the results for all the partitions, using a given combine functions and a neutral "zero value."
  RDD.aggregate(*zeroValue: U*, *seqOp: Callable[[U, T], U]*, *combOp: Callable[[U, U], U]*) → U

Example of Actions

```
# definition of the Python add function
def add(x,y): return x+y
# application of the reduce action with the commutative and associative
add function
# (the result is a Python value)
red = map_rdd.reduce(add)

map_rdd.count()
filter_rdd.take(2)
# writes into mydirectory on the HDFS home directory
filter_rdd.saveAsTextFile("mydirectory")

# collect(): retrieve all the elements of the dataset (from all nodes) to
the driver node (data must fit into memory)
# result: Python Array
local_array= filter_rdd.collect()
```



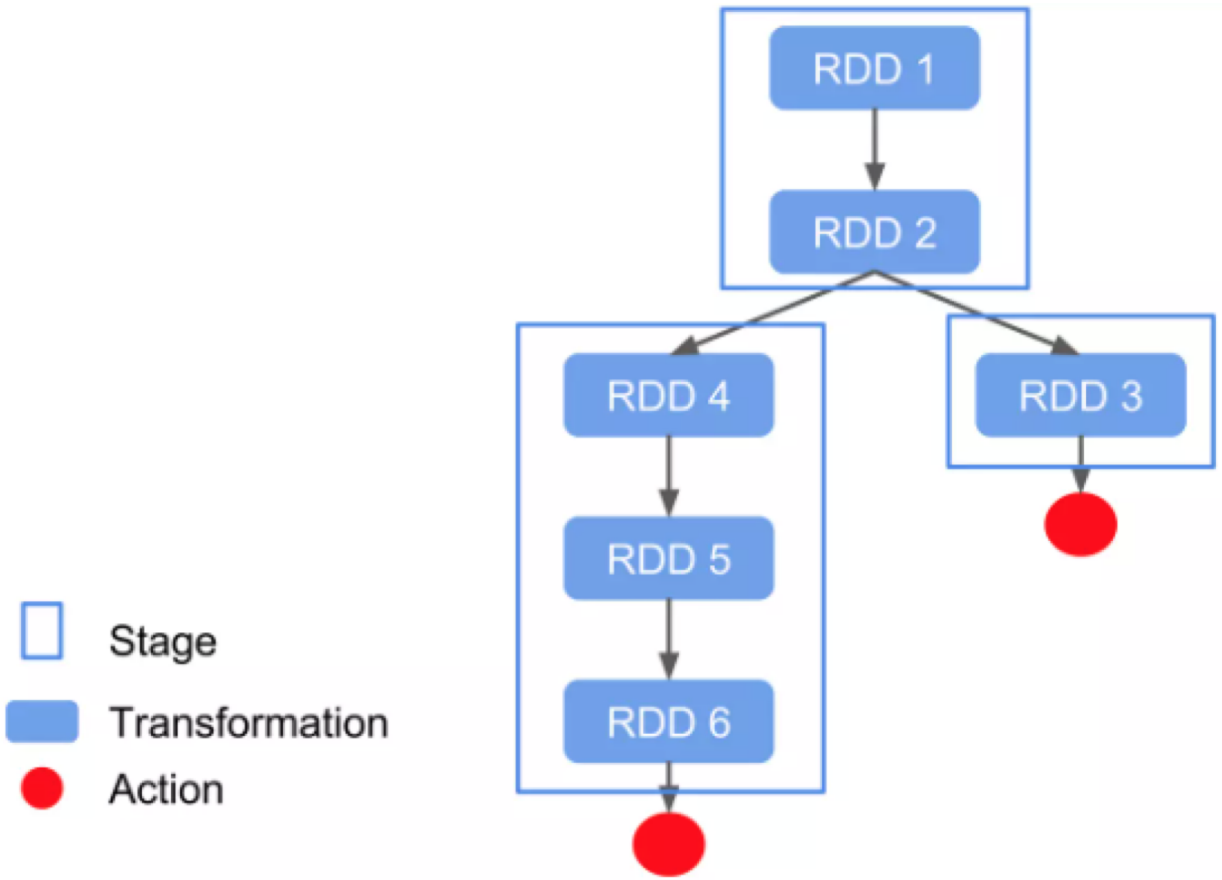
List of Actions (1)

action	description
reduce(*func***)**	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count()	return the number of elements in the dataset
first()	return the first element of the dataset – similar to *take(1)*
take(*n***)**	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement, fraction, seed***)**	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

List of Actions (2)

action	*description*
`saveAsTextFile(`*`path`***`)`**	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
`saveAsSequenceFile(`*`path`***`)`**	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
`countByKey()`	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
`foreach(`*`func`***`)`**	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

RDD Lineage Graph



Execution workflow:

- managed as a directed acyclic graph (DAG), executed when actions are executed

When an **action is requested**:

- Spark walks the RDD dependency graph *backwards*
- Creates a **Job** (sequence of transformations on data)

RDD: conclusion

Immutable:

- each transformation creates a new RDD (advantage: optimization)

Resilient (fault-tolerant):

- can be recovered (recreated) in any point of the execution time from the “lineage” of each RDD (the sequence of operations that produced it)

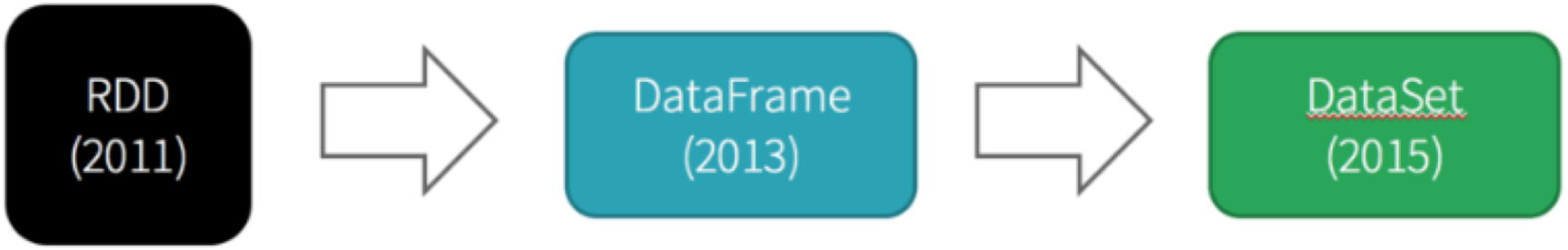
Lazy evaluated:

- RDDs are materialized only when an action is performed (an entire chain of the graph is executed)

Two types of operations:

	Transformation	Action
Examples	map()	take()
Returns	Another RDD	Local value
Executes	Lazily	Immediately. Executes transformations

History of Spark APIs



Distribute collection of JVM objects

Functional Operators (map, filter, etc.)

Distribute collection of Row objects

Expression-based operations and UDFs

Logical plans and optimizer

Fast/efficient internal representations

Internally rows, externally JVM objects

Almost the "Best of both worlds": **type safe + fast**

But slower than DF
Not as good for interactive analysis, especially Python

databricks

Spark Datasets and Dataframes

Dataframe (Schema RDD):

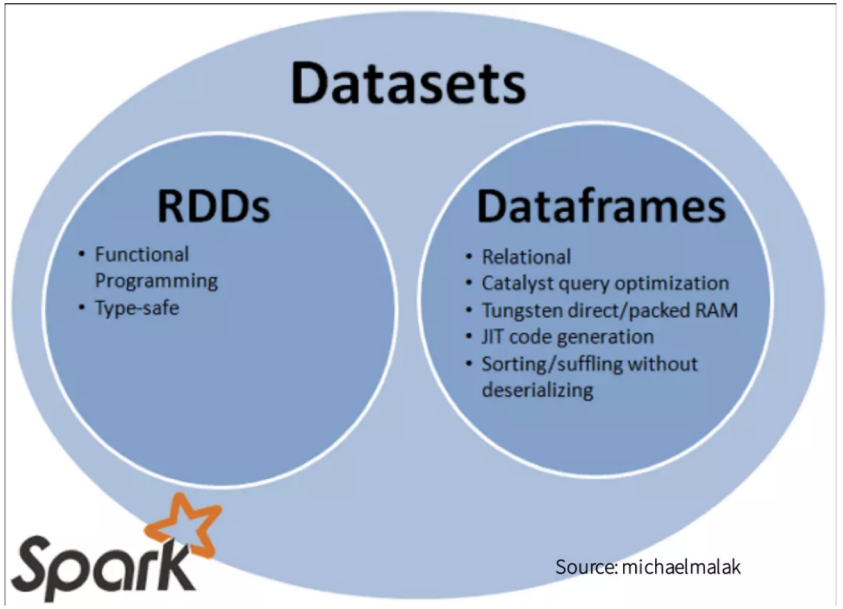
- *Dataframe = Dataset[Row]*
- *Row = tuple with attributes*
- *Implemented in Scala, Java, Python, R*

Dataset:

- *RDD + SQL*
- *Implemented in Scala and Java*

Spark SQL

- *Package pyspark.sql*
- *Dataframe and Datasets*
- *« SQL » operators*
- *Optimization : Catalyst optimizer*



Spark DataFrames

Creating a session

```
import pyspark # only run after findspark.init()
from pyspark.sql import SparkSession
from pyspark import Row
spark = SparkSession.builder.master(“local[4]”).appName(“SQL_sess").getOrCreate()
```

DataFrame: creation/file reading

```
row1 = Row(name='Alice', age=11)
row2 = Row(name='Bob', age=12)
df=spark.createDataFrame([row1,row2])
df.show()
```

```
+---+-----+
|age| name|
+---+-----+
| 11|Alice|
| 12|  Bob|
+---+-----+
```

```
films =
spark.read.json("movielens/films.json")
films.show(3)
```

```
+--------------------+------+--------------------+
|                   g|    nF|               titre|
+--------------------+------+--------------------+
|             [Drama]|  8754|Prime of Miss Jea...|
|          [Thriller]|111486|Lesson of the Evi...|
|[Animation, Child...|  1033|Fox and the Hound...|
+--------------------+------+--------------------+
```

DataFrame: print schema

```
films.printSchema()

root
 |-- g: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- nF: long (nullable = true)
 |-- titre: string (nullable = true)
```

DataFrame: projection, selection

```
# projection
films.select(films.titre).show(1)
films.select(films['titre']).show(1)
films.select('titre').show(1)

+--------------------+
|              titreF|
+--------------------+
|Prime of Miss Jea...|
+--------------------+
```

```
# projection / selection
films.select(films['titre']) \
     .where(films['nF']==1).show()

+----------------+
|           titre|
+----------------+
|Toy Story (1995)|
+----------------+
```

DataFrame: orderBy

```
films.select(films.nF, films.titre, films.g).orderBy('titre').show(3,0)

+-----+-----------------------------------+-----------+
|nF   |titre                              |g          |
+-----+-----------------------------------+-----------+
|51372|"""Great Performances"" Cats (1998)"|[Musical]  |
|7789 |"11'09""01 - September 11 (2002)"  |[Drama]    |
|74486|$9.99 (2008)                       |[Animation]|
+-----+-----------------------------------+-----------+
```

DataFrame: column rename

```
films.select(films.titre.alias('t')).show(3,0)
+----------------------------------------+
|t                                       |
+----------------------------------------+
|Prime of Miss Jean Brodie, The (1969)   |
|Lesson of the Evil (Aku no kyôten) (2012)|
|Fox and the Hound, The (1981)           |
+----------------------------------------+

films.withColumnRenamed('titre','t').show(3)
+--------------------+------+--------------------+
|                   g|    nF|                   t|
+--------------------+------+--------------------+
|             [Drama]|  8754|Prime of Miss Jea...|
|          [Thriller]|111486|Lesson of the Evi...|
+--------------------+------+--------------------+
```

DataFrame: explode

```
films.select(films.titre,films.g) \
     .show(4)

+--------------------+--------------------+
|               titre|                   g|
+--------------------+--------------------+
|Prime of Miss Jea...|             [Drama]|
|Lesson of the Evi...|          [Thriller]|
|Fox and the Hound...|[Animation, Child...|
|Sinbad: Legend of...|[Adventure, Anima...|
+--------------------+--------------------+
```

```
fgenres=films.select(films.nF,
                     explode(films.g) \
                     .alias('genre')) \
             .orderBy('nf') \
             .show(4)

+---+---------+
| nF|    genre|
+---+---------+
|  1|Animation|
|  1|Adventure|
|  1| Children|
|  1|   Comedy|
+---+---------+
```

DataFrames: join, cross join (cartesian product)

```
films2 = films_sg.join(genres,films.nF==genres.nF)
films2.show(2)
+------+------------------+------+--------+
|    nF|             titre|    nF|   genre|
+------+------------------+------+--------+
|  8754|Prime of Miss Jea...|  8754|   Drama|
|111486|Lesson of the Evi...|111486|Thriller|
+------+------------------+------+--------+
films2 = films_sg.crossJoin(fgenres)
films2.show(2)
+----+------------------+---+---------+
|  nF|             titre| nF|    genre|
+----+------------------+---+---------+
|8754|Prime of Miss Jea...|  1|Adventure|
|8754|Prime of Miss Jea...|  1|Animation|
+----+------------------+---+---------+
```

```
fgenres.show(2)
+---+---------+
| nF|    genre|
+---+---------+
|  1|Adventure|
|  1| Children|
+---+---------+

films_sg.show(2)
+------+--------------------+
|    nF|               titre|
+------+--------------------+
|  8754|Prime of Miss Jea...|
|111486|Lesson of the Evi...|
+------+--------------------+
```

DataFrames: group by/aggregate functions

```
films_max=fgenres.groupBy('genre')\
                 .max('nF').show(2)
+-------+-------+
|  genre|max(nF)|
+-------+-------+
|  Crime| 161582|
|Romance| 162672|
+-------+-------+
films_cpt=fgenres.groupBy('genre') \
         .agg(count('nF'))
films_cpt.show(2)
+-------+---------+
|  genre|count(nF)|
+-------+---------+
|  Crime|     1100|
|Romance|     1545|
+-------+---------+
```

```
genres_cpt=fgenres.groupBy('nF') \
           .agg(countDistinct('genre'))
genres_cpt.show(2)

+---+---------------------+
| nF|count(DISTINCT genre)|
+---+---------------------+
|  1|                    5|
|  2|                    3|
+---+---------------------+
```

Aggregate function: collect_list

```
films_notes.printSchema()

root
 |-- g: array (nullable = true)
 |    |-- element: string (containsNull =
true)
 |-- nF: long (nullable = true)
 |-- titre: string (nullable = true)
 |-- nF2: long (nullable = true)
 |-- note: float (nullable = true)
```

```
films_notes.groupBy('nF')\
    .agg(collect_list('note')
    .alias('notes'))\
    .show(2)

+---+--------------------+
| nF|               notes|
+---+--------------------+
| 26|[4.0, 5.0, 5.0, 3...|
| 29|[5.0, 5.0, 4.0, 5...|
+---+--------------------+
```

SQL functions: lower, max

```
from pyspark.sql.functions import *

films.select(lower(films.titre)).show(3)
+--------------------+
|        lower(titre)|
+--------------------+
|prime of miss jea...|
|lesson of the evi...|
|fox and the hound...|
+--------------------+

films.select(max(films.nF)).show(1)
+-------+
|max(nF)|
+-------+
| 164979|
+-------+
```

SQL functions: split

```
films.select(films.titre).show(1,0)
+-----------------------------------+
|titre                              |
+-----------------------------------+
|Prime of Miss Jean Brodie, The (1969)|
+-----------------------------------+

films.select(split(films.titre,'[()]+')[0].alias('titre'),
             split(films.titre,'[()]+')[1].alias('annee')).show(1,0)
+-----------------------------+-----+
|titre                        |annee|
+-----------------------------+-----+
|Prime of Miss Jean Brodie, The |1969 |
+-----------------------------+-----+
```

DataFrames: filtering, new columns

```
f2=films.withColumn('nbgenres',size(films.g)).show(2)

+----------+------+--------------------+--------+
|         g|    nF|               titre|nbgenres|
+----------+------+--------------------+--------+
|   [Drama]|  8754|Prime of Miss Jea...|       1|
|[Thriller]|111486|Lesson of the Evi...|       1|
+----------+------+--------------------+--------+
f2.filter(f2.nbgenres>2).show(2)

+--------------------+----+--------------------+--------+
|                   g|  nF|               titre|nbgenres|
+--------------------+----+--------------------+--------+
|[Animation, Child...|1033|Fox and the Hound...|       3|
|[Adventure, Anima...|6536|Sinbad: Legend of...|       4|
+--------------------+----+--------------------+--------+
```

DataFrames: user functions

```
@udf('float')  # float: type du résultat
def mamoy(l):
    sum=0
    len=0
    for x in l:
        sum=sum+x
        len=len+1
    return sum/len
```

```
fn.show(2)
+---+--------------------+
| nF|               notes|
+---+--------------------+
| 26|[4.0, 5.0, 5.0, 3...|
| 29|[5.0, 5.0, 4.0, 5...|
+---+--------------------+
fn2=fn.withColumn('avg',mamoy(fn.notes))
fn2.show(2)
+---+--------------------+-----+
| nF|               notes|  avg|
+---+--------------------+-----+
| 26|[4.0, 5.0, 5.0, 3...|  4.1|
| 29|[5.0, 5.0, 4.0, 5...|4.025|
+---+--------------------+-----+
```

Spark: conclusion

- Advantages:
  - cluster computing platform, fast and fault-tolerant large-scale computations
  - Rich API for fast advanced analytics: ‘MAP’ and ‘reduce’ operations, ML algorithms, graphs, SQL queries, streaming data, etc.
  - Processing of high volumes of structured and unstructured data
  - Speed due to low latency RAM(in-memory data processing ability)
  - Dynamic: parallel execution of applications
  - Multi-Language engine: Java, Scala, Python, R, SQL
  - Very flexible: data from various data sources can be used (usage with HDFS)

- Disadvantages:
  - Dependence on external storage systems
  - Large number of Small files
  - Less number of algorithms in MLlib.
  - High memory consumption and increased hardware costs:
  - Unsuitable for for multi-user environments
  - Non automatic optimization process