Databases and Big Data

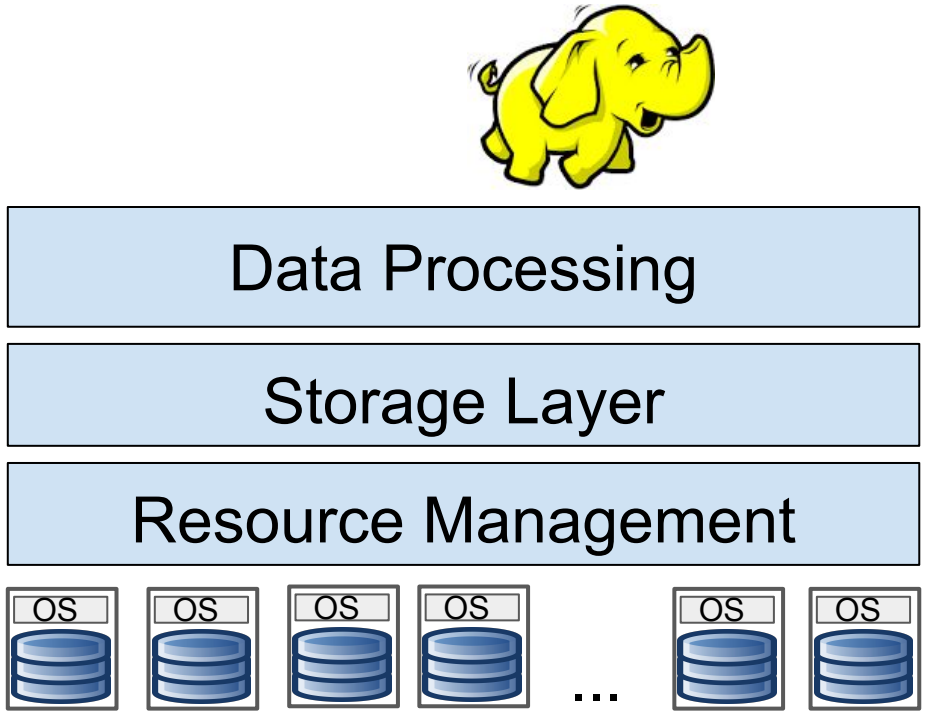
Spark

Agenda

- Why Spark?
- Spark APIs
- Examples

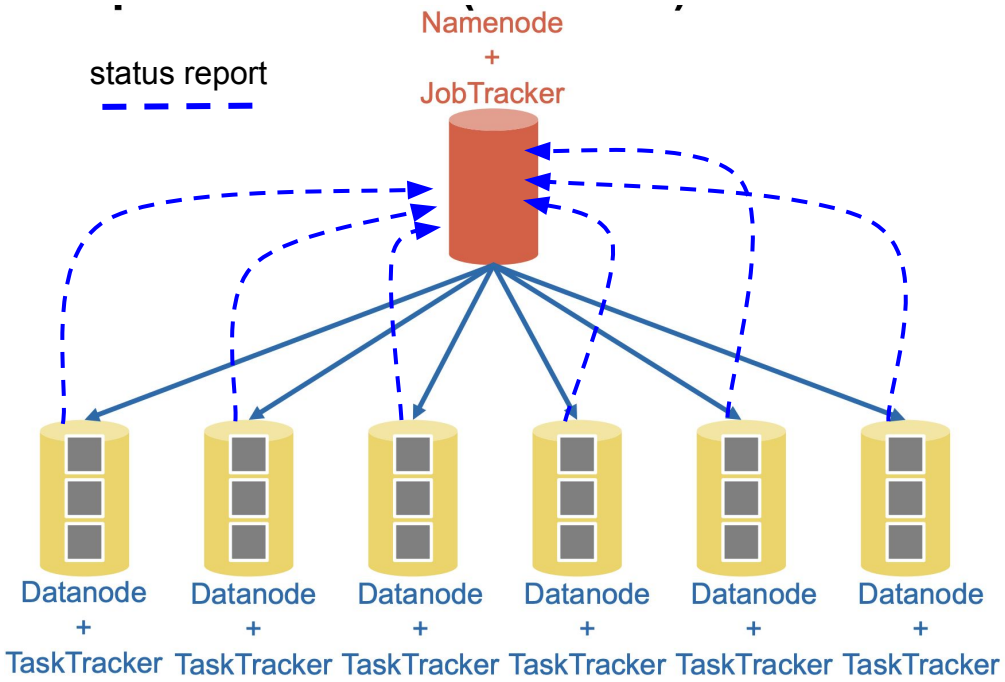
Recap

- Hadoop stack
  - HDFS: distributed file system
  - MapReduce: data processing
- We **haven't seen**
  - How MapReduce handles failures



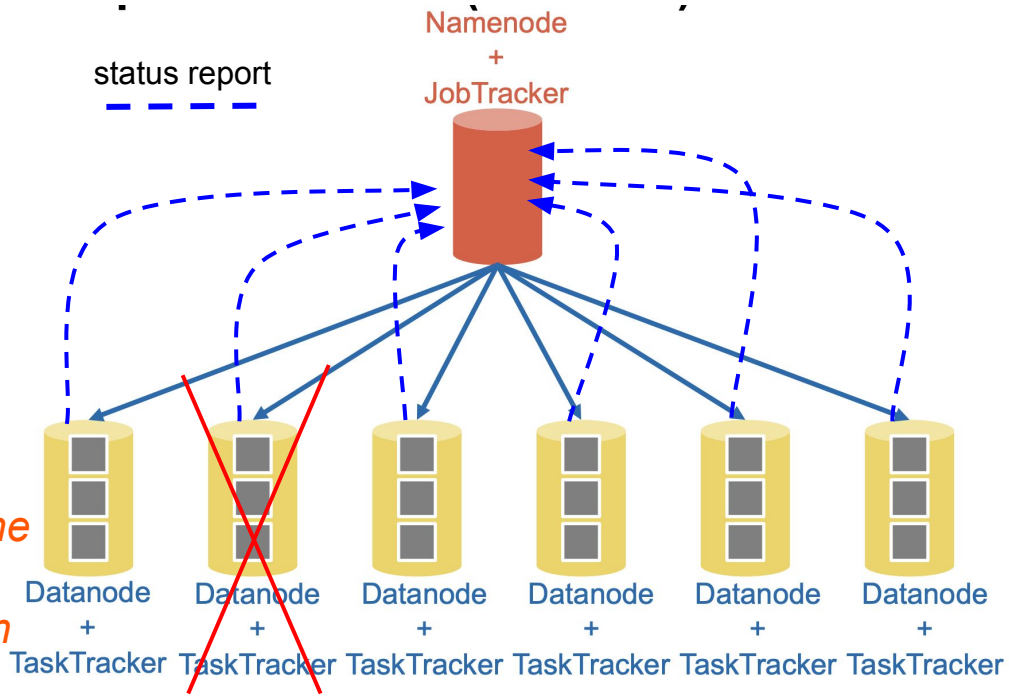
Recap

- Architecture
  - JobTracker tracks progress of map/reduce tasks
  - TaskTracker reports progress



MapReduce Fault Tolerance

- When a node fails:
  - Both DataNode and TaskTracker fail
- How did HDFS handle failures?
- JobTracker starts another tasks
  - Why possible?



When reboot, Jobtracker needs to recover all the hdfs blocks. Namenode keep tracks of all the metadata on the node and is the master, so can coordinate the recovering.
If some node fail, just start a new machine and run

Problems with MapReduce

Not this!

- Probably just sour grapes
  - Remember how DB people reacted to MongoDB?

The Database Column
A multi-author blog on database technology and innovation.

MapReduce: A major step backwards

By David DeWitt on January 17, 2008 4:20 PM | Permalink | Comments (44) | TrackBacks (1)
[Note: Although the system attributes this post to a single author, it was written by David J. DeWitt and Michael Stonebraker]

On January 8, a Database Column reader asked for our views on new distributed database research efforts, and we'll begin here with our views on MapReduce. revolution of so-called "cloud computing." This paradigm entails harnessing large numbers of (low-end) processors working in parallel to solve a computing pr beans" rather than utilizing a much smaller number of high-end servers.

MapReduce

*Completely **independent** of each other.*
(suitable for map task, cause you can just throw more machines in and the task can run faster in parallel)

- To fulfill a real need at **end of 1990s**:
  - Search is the killer app
  - Need to create index on huge data (Web)
    - What index?
  - Data process is embarrassingly parallel
    - ETL

But then, needs changed

What Changed?

- Application trends in late 00s
  - **Iterative computation:**
    - Machine learning: SGD has multiple passes
  - **Interactive computation:**
    - Data science: exploratory analysis
      - Don’t always know what you want in advance

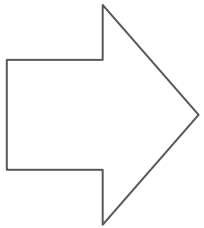
MapReduce designed for **batch computation**

Both -> not supported by MapReduce framework

What Changed?

Batch computation

- Do it once, or infrequently
- Throughput
- Examples:
  - Build index
  - Sort
  - ETL



TP: *your system can handle that huge volume of data, main requirement*

Iterative computation

- Do it many times
- Throughput
- Examples:
  - PageRank
  - kMeans
  - Graphs
  - SGD

Interactive computation

- Find out what “it” is
- Ad-hoc queries
- Human in the loop
- Latency
- Example:
  - Ad-hoc queries
  - Exploratory data science

Latency:
Fast queries (1-10s)

What Changed?

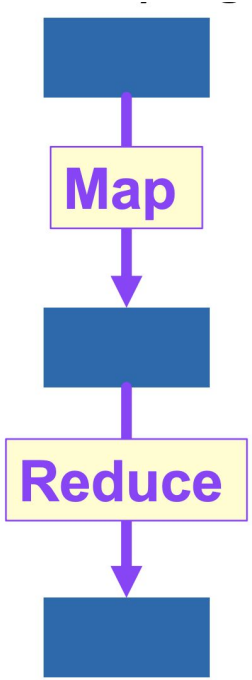
- Most applications **fit in memory**

Memory (GB)	Facebook (% jobs)	Microsoft (% jobs)	Yahoo! (% jobs)
8	69	38	66
16	74	51	81
32	96	82	97.5
64	97	98	99.5
128	98.8	99.4	99.8
192	99.5	100	100
256	99.6	100	100

*G Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica, "Disk-Locality in Datacenter Computing Considered Irrelevant", HotO 2011

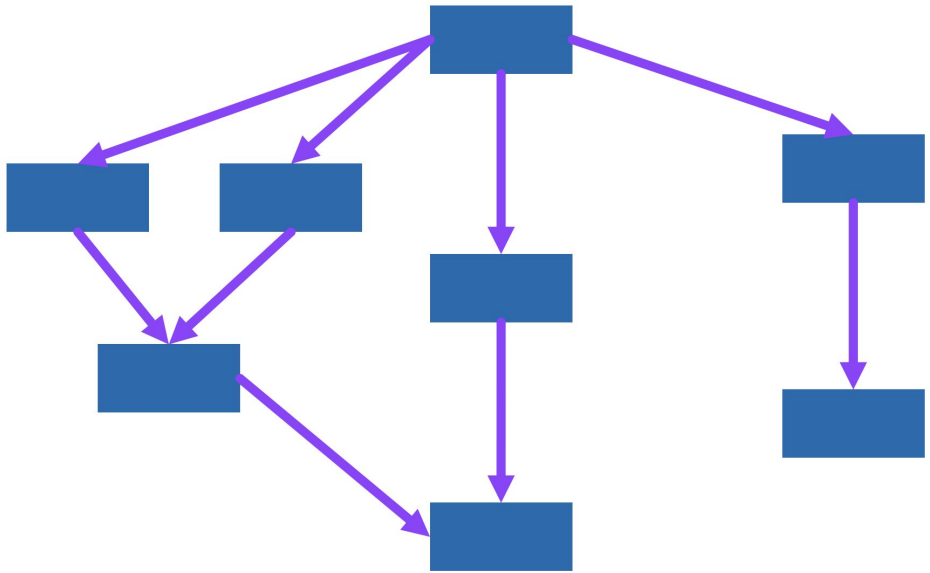
MapReduce

- Applications changed
- Also, MapReduce model too rigid
  - Just 2 steps per MapReduce job
  - You can chain multiple jobs together
    - **What's the catch?**



MapReduce

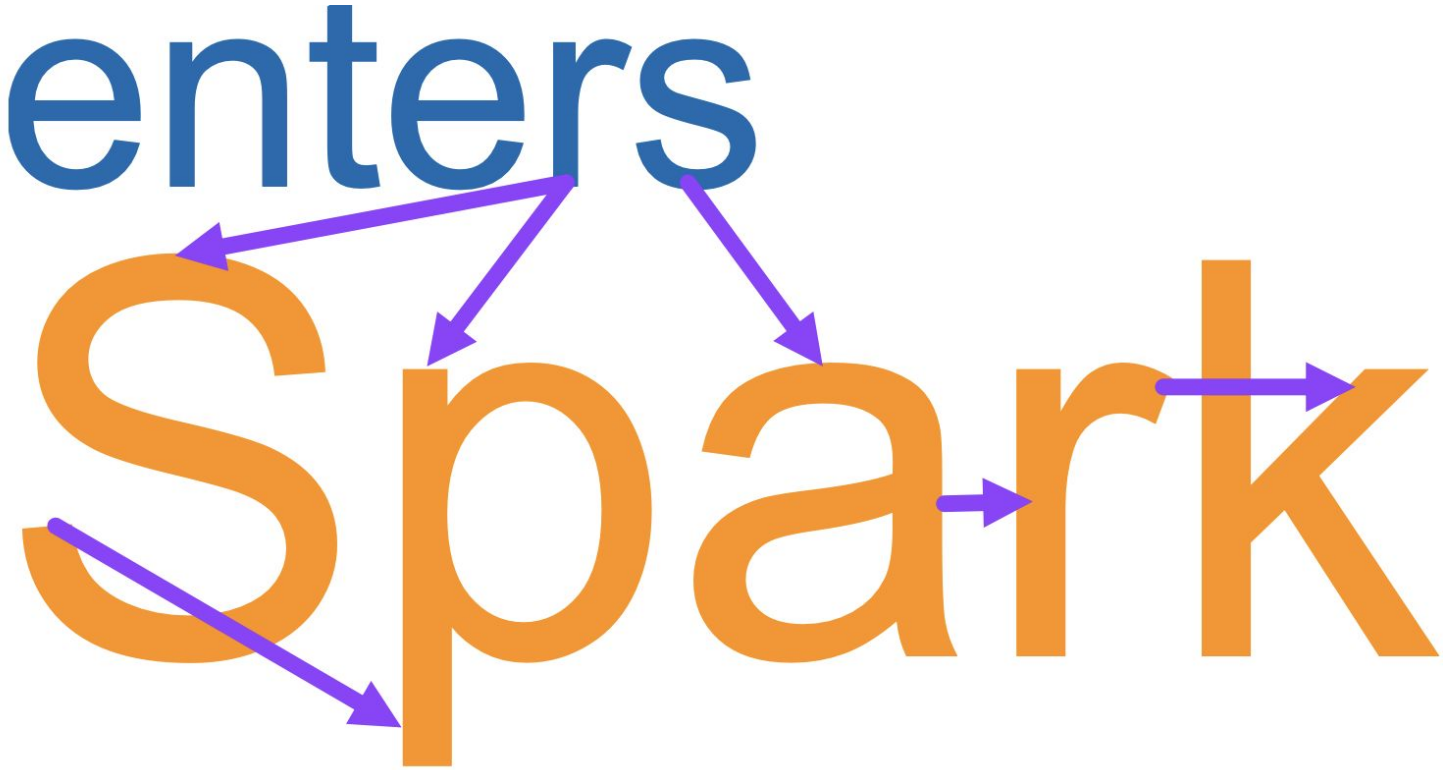
- What if I want this type of computation?
  - Direct Acyclic Graph
  - More general



Summary

- MapReduce couldn’t keep up:
  - Applications that need more than 1 MapReduce jobs
  - Iterative computations that reuse a lot of data
  - Interactive computations that need fast response
- MapReduce didn’t take advantage of memory

Even if whole workload can fit into memory


enters
Spark

Spark History

- Started in 2009
  - As a faster alternative for MapReduce

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

Spark History

- Now a multi-billions company

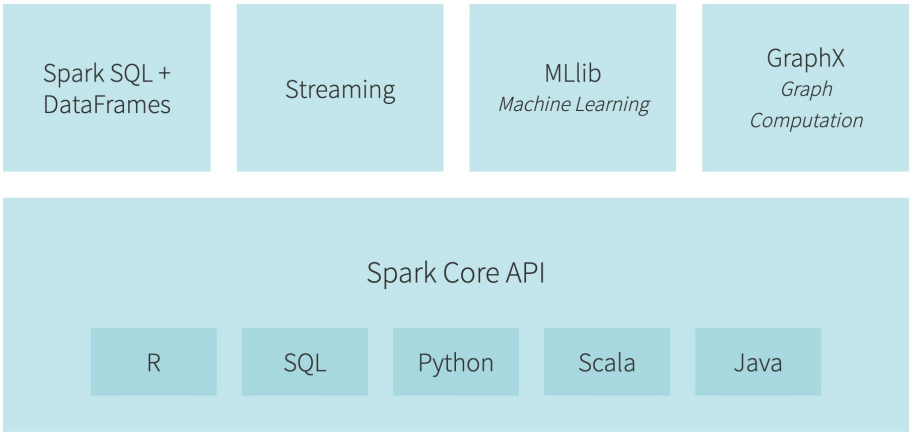
Databricks announces $400M round on $6.2B valuation as analytics platform continues to grow

Ron Miller @ron_miller / 2:14 am +08 • October 23, 2019

Comment

Spark History

- With a vibrant ecosystem

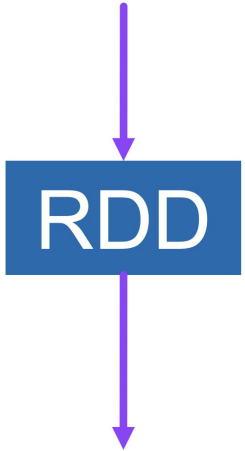


Spark

- RDD Abstraction
  - RDDs in Spark are like objects in Java
  - Meaning:
    - Your Spark programs manipulate RDDs

Spark's first-class citizen

Resilient
Distributed
Dataset



Spark RDD

it's just a

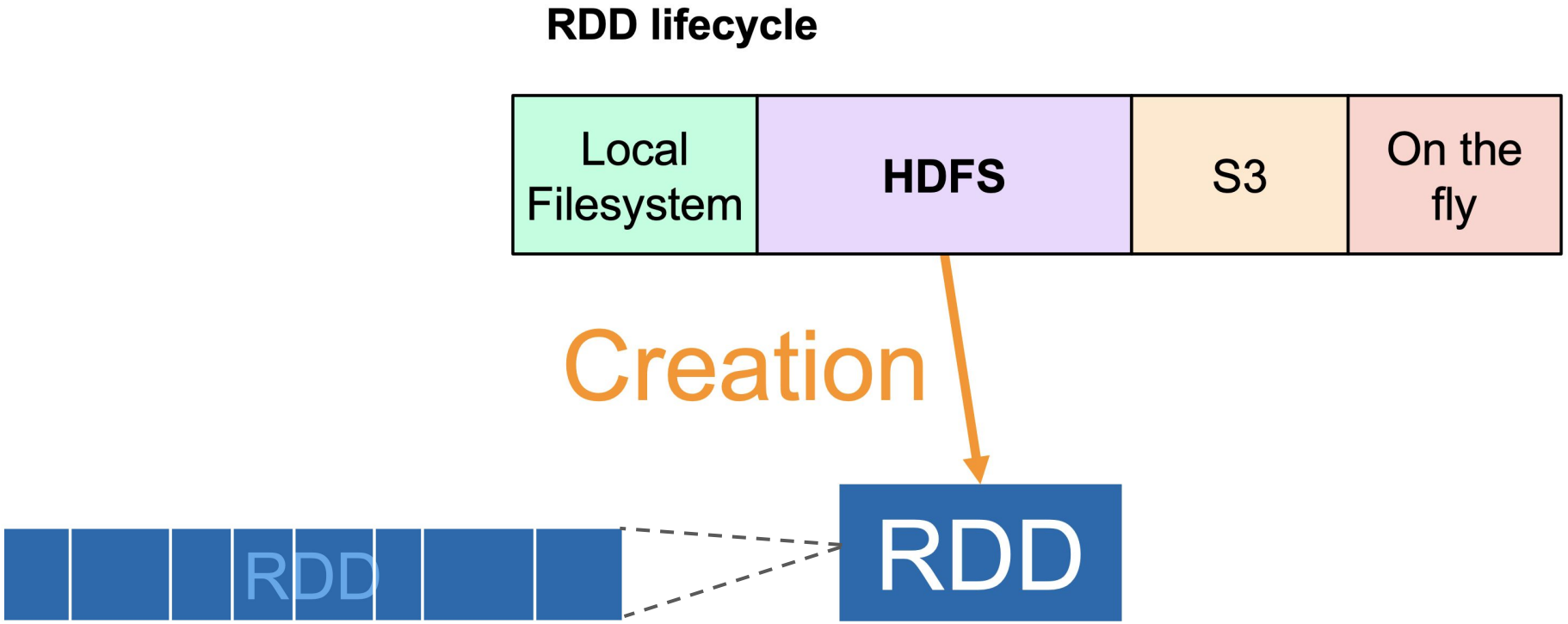
Big

collection

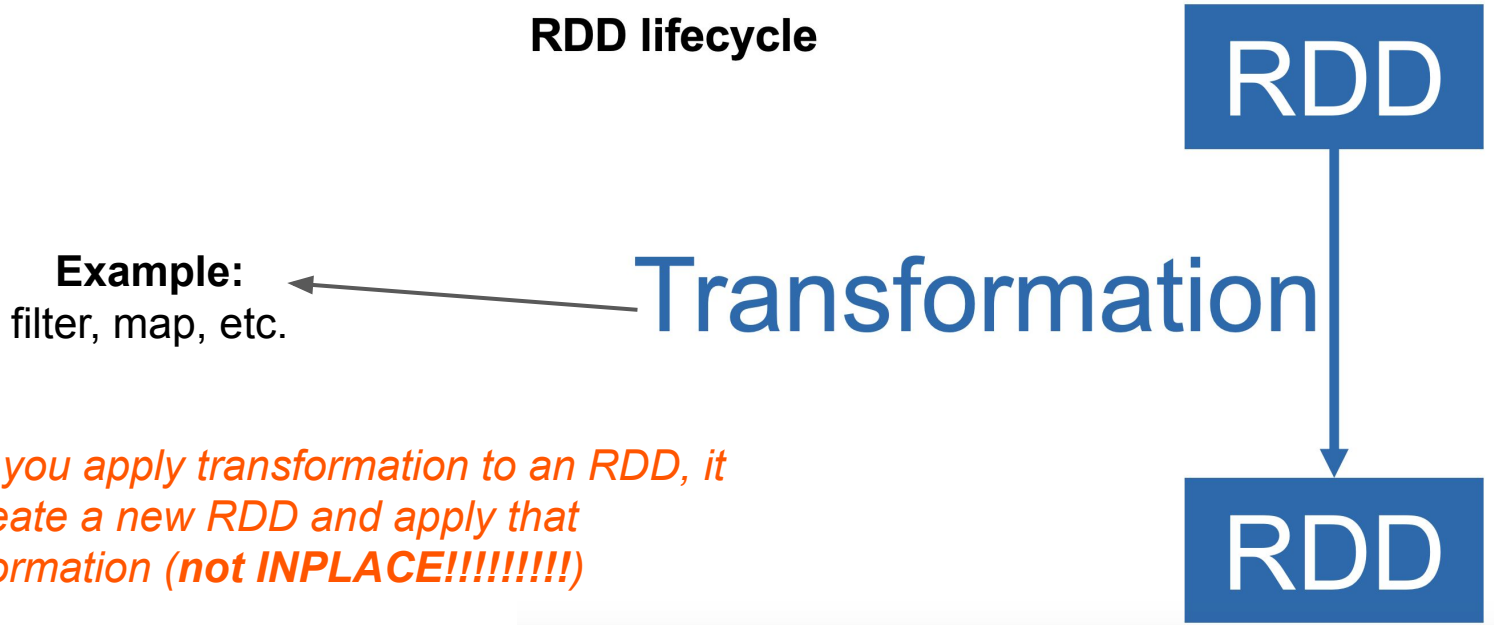
Spark will try to fit partitions in memory

... and it is par ti tion ed

Life of a RDD


RDD lifecycle
Local Filesystem
HDFS
S3
On the fly
Creation
RDD
RDD

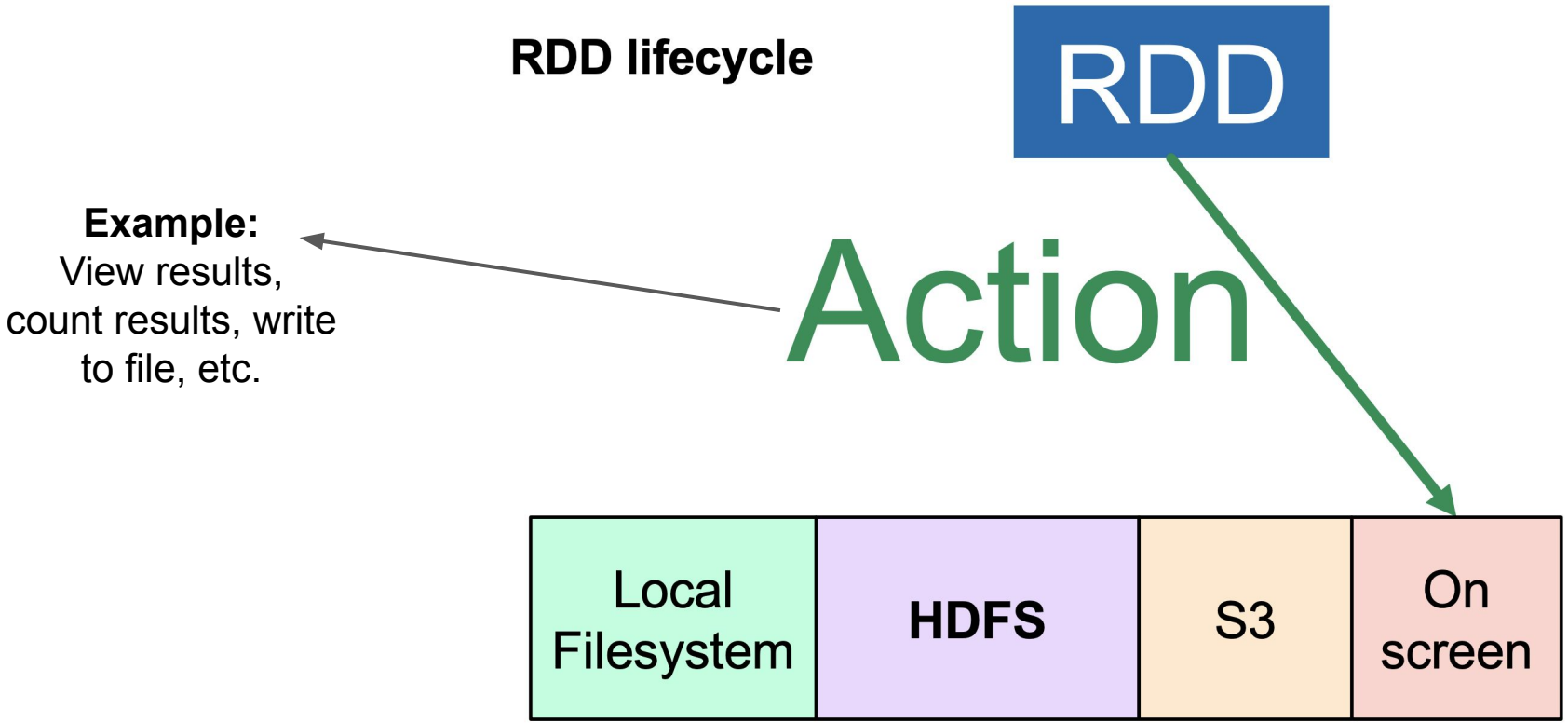
Life of a RDD



*When you apply transformation to an RDD, it will create a new RDD and apply that transformation (**not INPLACE!!!!!!!!!**)*

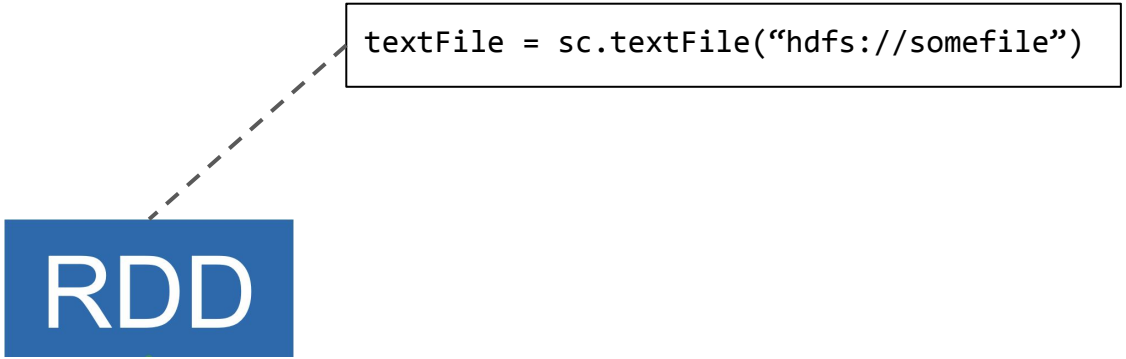
IMMUTABLE!!!!!!!!

Life of a RDD


RDD lifecycle
RDD
Example:
View results,
count results, write
to file, etc.
Action
Local
Filesystem
HDFS
S3
On
screen

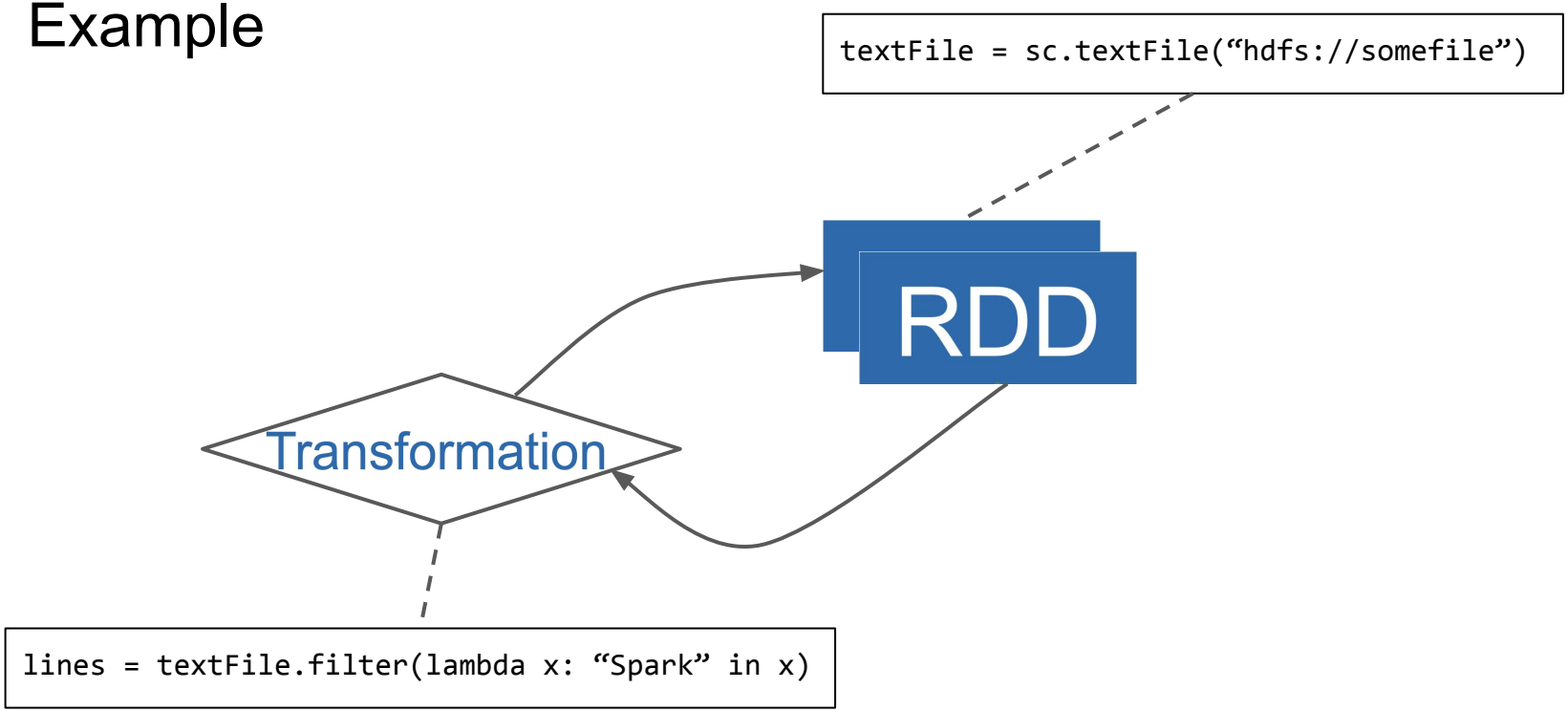
Life of a RDD

- Example



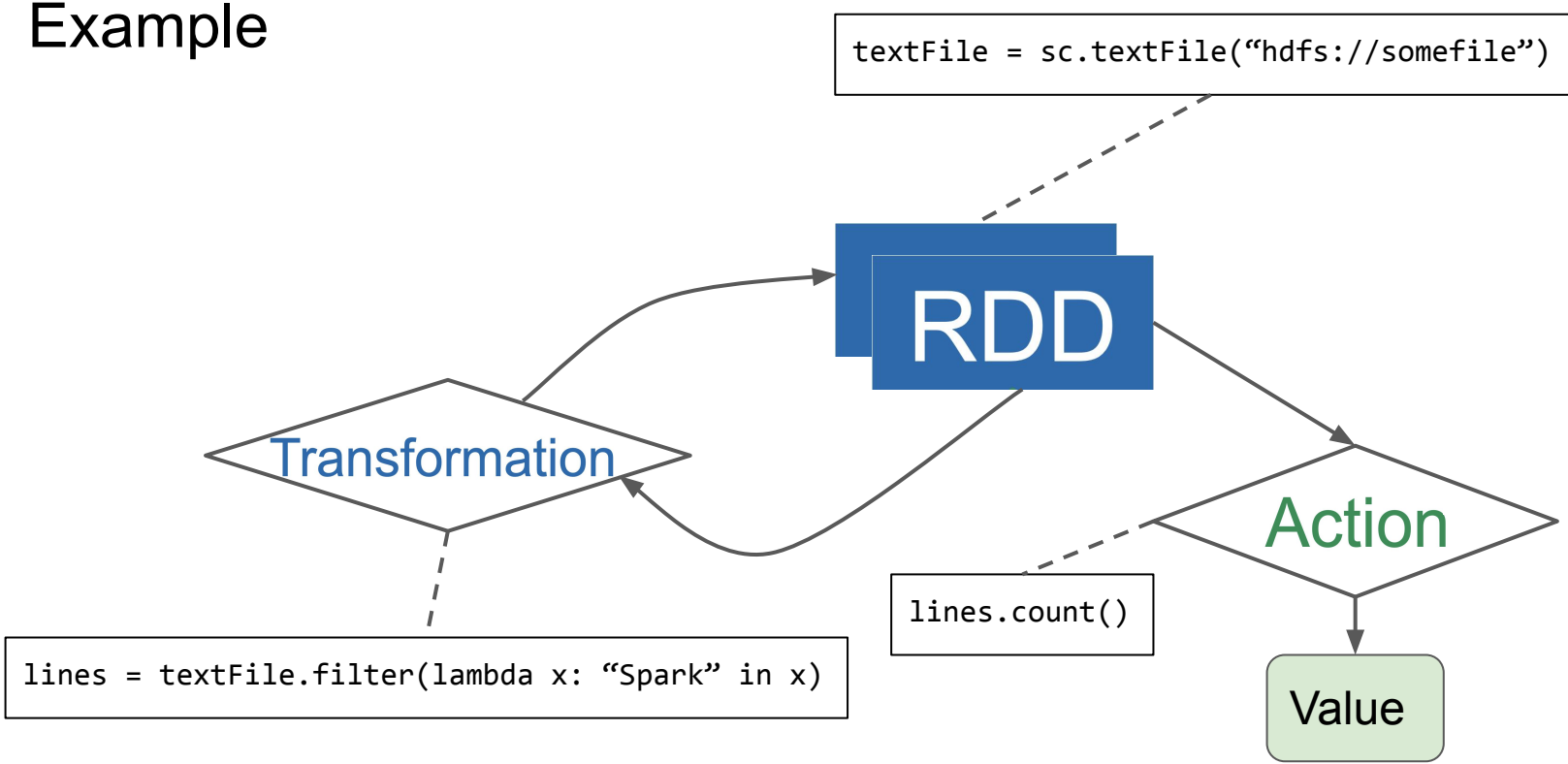
Life of a RDD

- Example



Life of a RDD

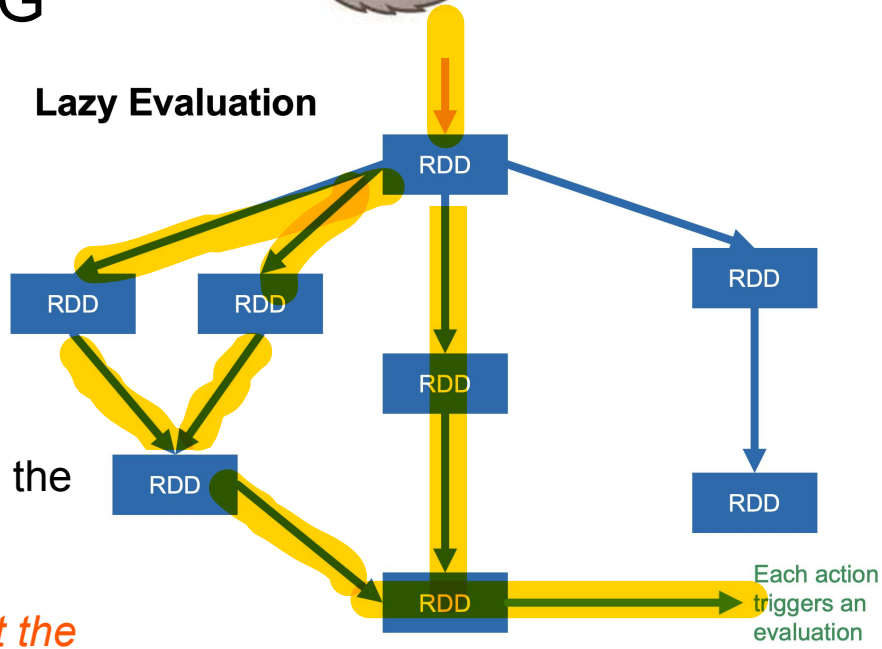
- Example



Spark Execution

- Transformed RDDs form a DAG
- Lazy evaluation:
  - Does not execute transformation immediately
  - Wait until Action!
    - Because it's **when** you **want** to see the results

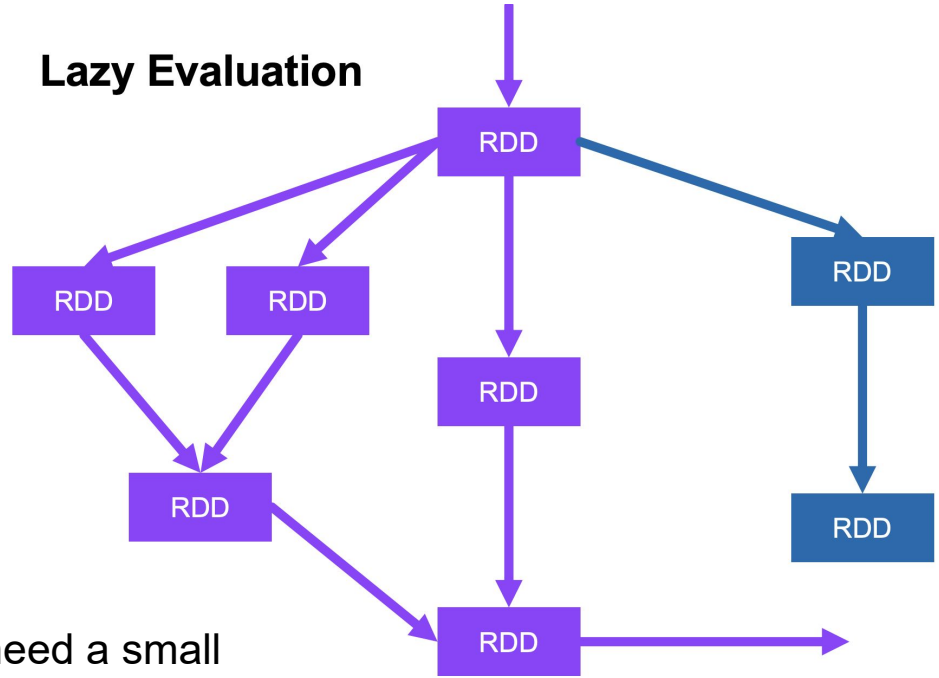
Doesn't apply all the transformation at once (if not the memory consumption will be huge)



Spark Execution

- Lazy evaluation

Almost all functional programming languages do this!



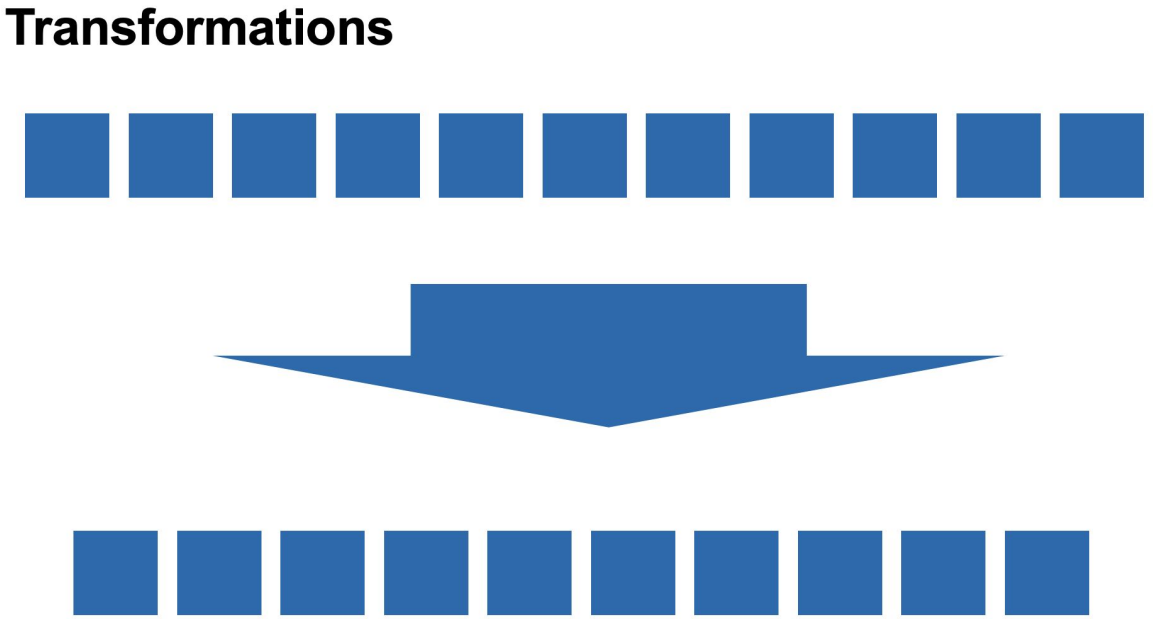
Why delaying execution?

Faster, sometimes you only need a small fraction of original RDD

Spark APIs - Transformation

Transformation

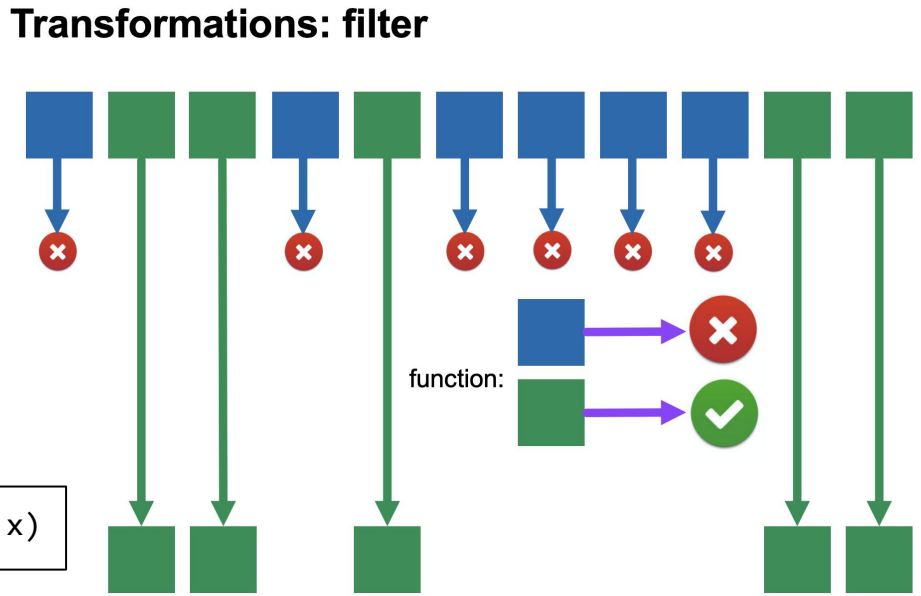
- Create one RDD from another



Transformation

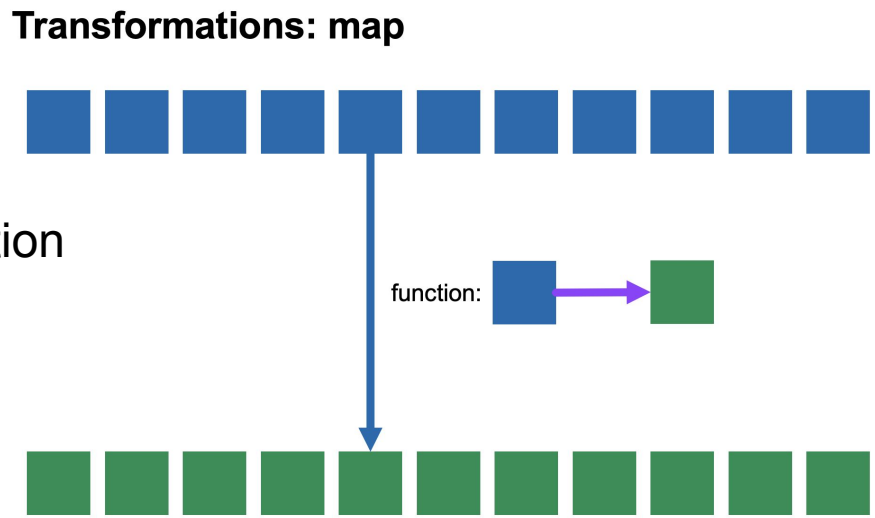
- Filter:
  - Input: RDD, a filter function
  - Output: RDD with element satisfying the function
  - Example:

```
lines = textFile.filter(lambda x: “Spark” in x)
```



Transformation

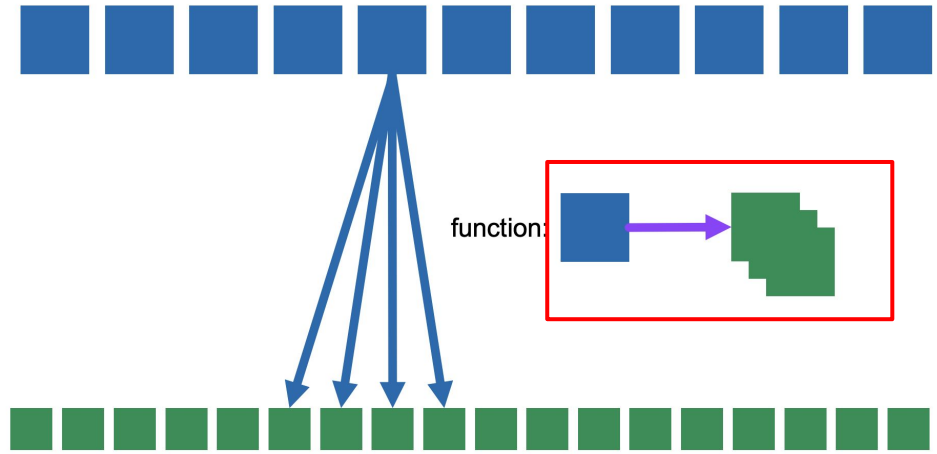
- Map:
  - Input: RDD, map function
  - Output: RDD, where the map function applied to every element



Transformation

- flatMap
  - Like map, but
  - Input function returns a list
  - Output RDD flattens the list

Transformations: flatMap



map:
f is one to one

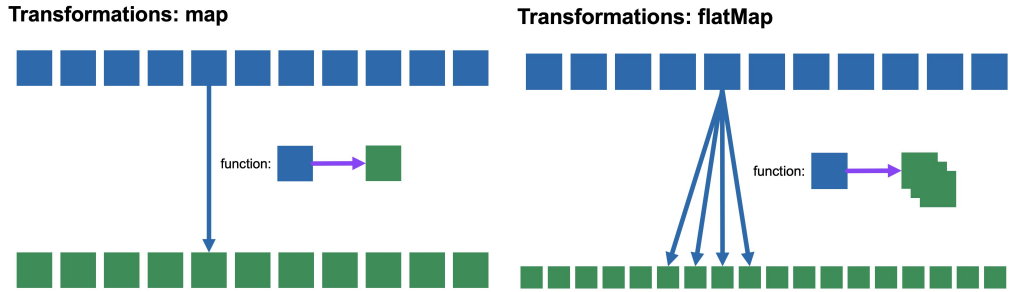
f(x) → y

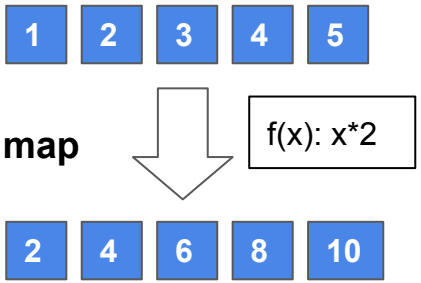
flatMap:
f is one to many

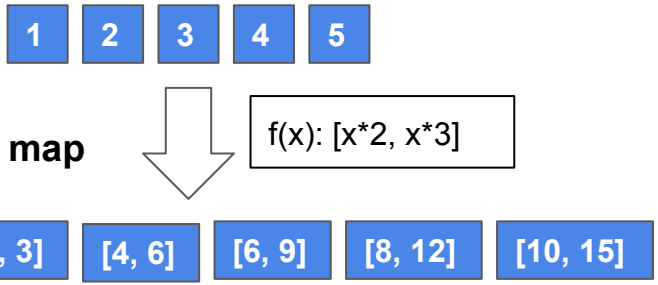
f(x) → [y1,y2,..]

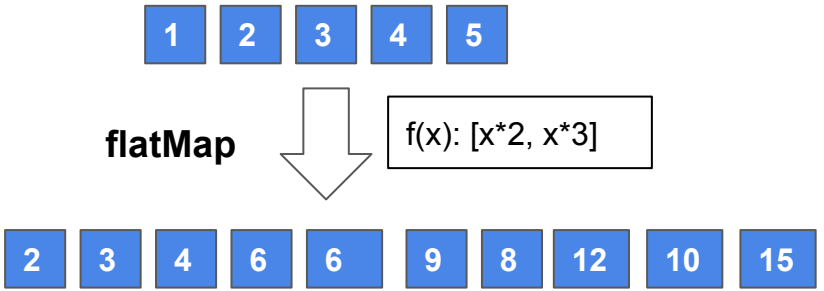
Transformation

- map vs. flatMap



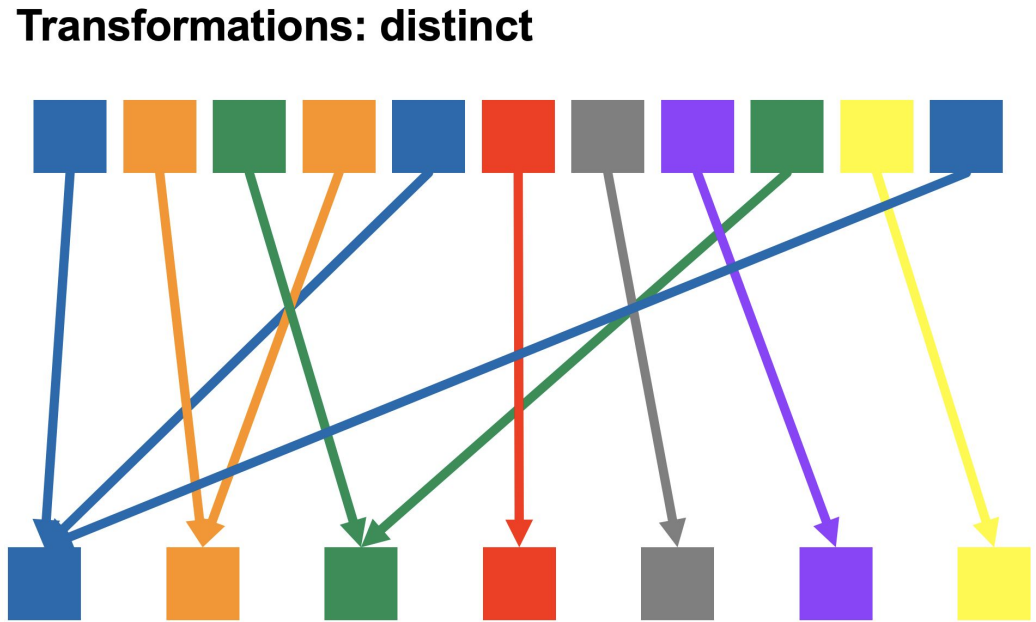






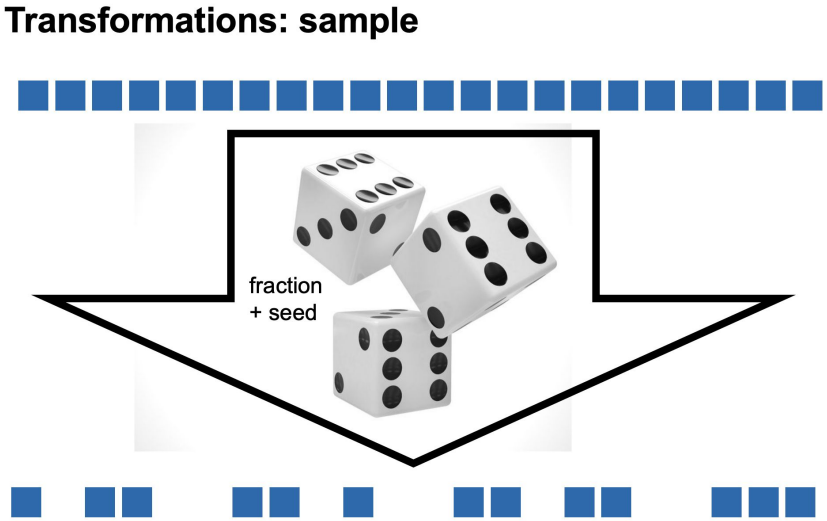
Transformation

- Distinct:
  - Input: RDD
  - Output: RDD without duplicates



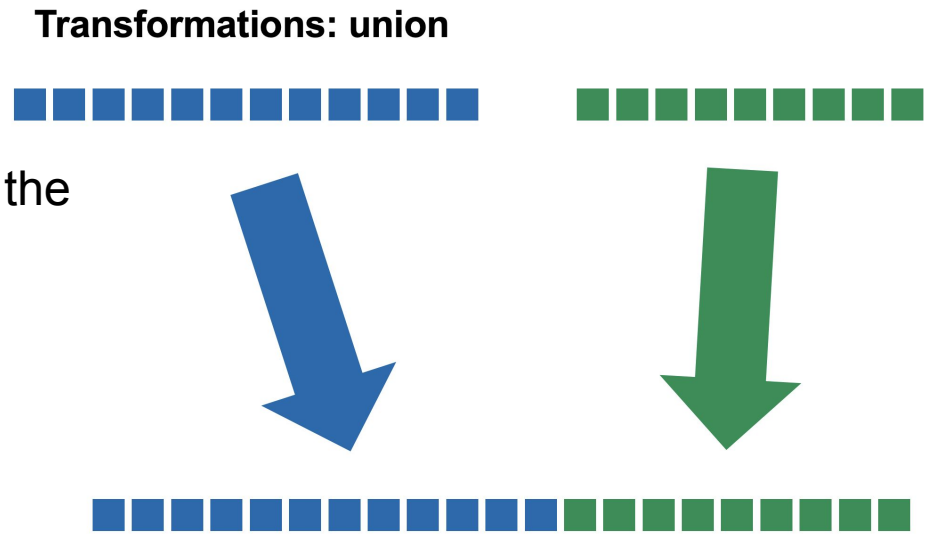
Transformation

- Sample
  - Input: RDD, fraction, seed
  - Output: RDD with sampled element



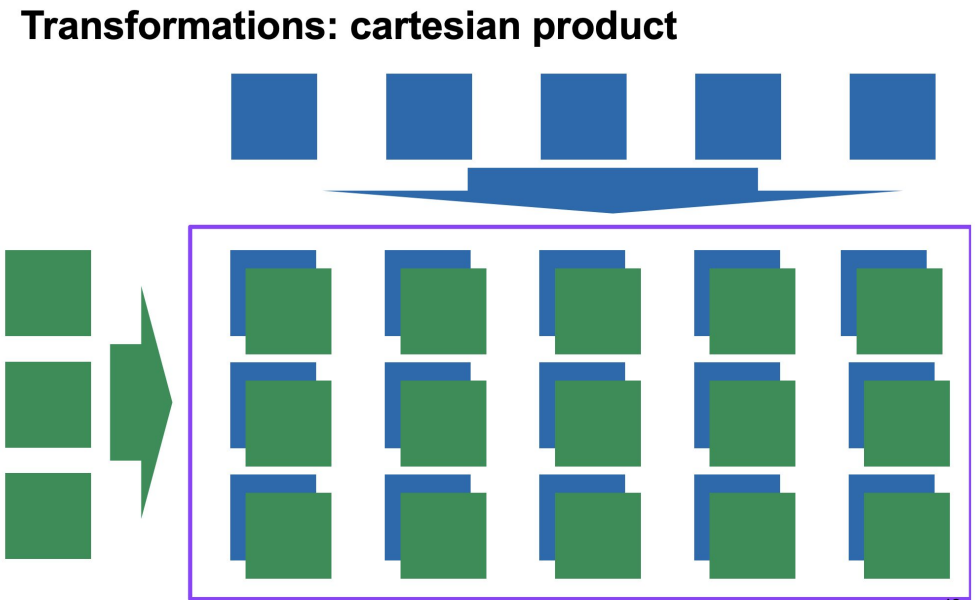
Transformation

- Union:
  - Input: 2 RDDs
  - Output: RDD with elements from the input



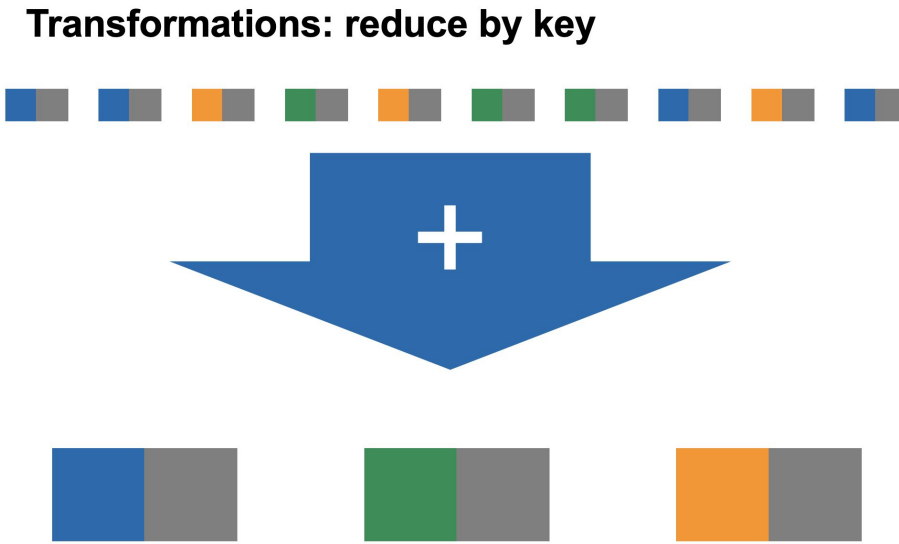
Transformation

- Cartesian:
  - Input: RDD T and U
  - Output: RDD of (T, U) pairs



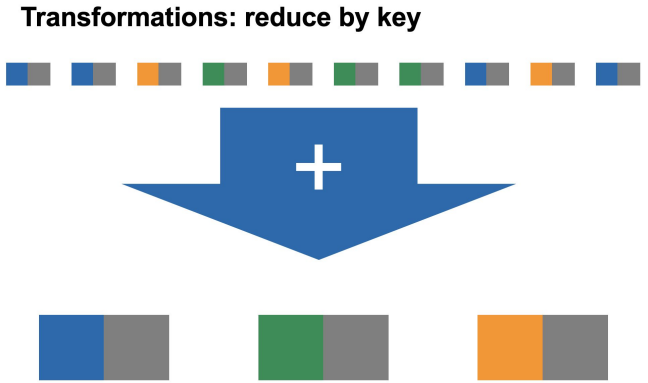
Transformation

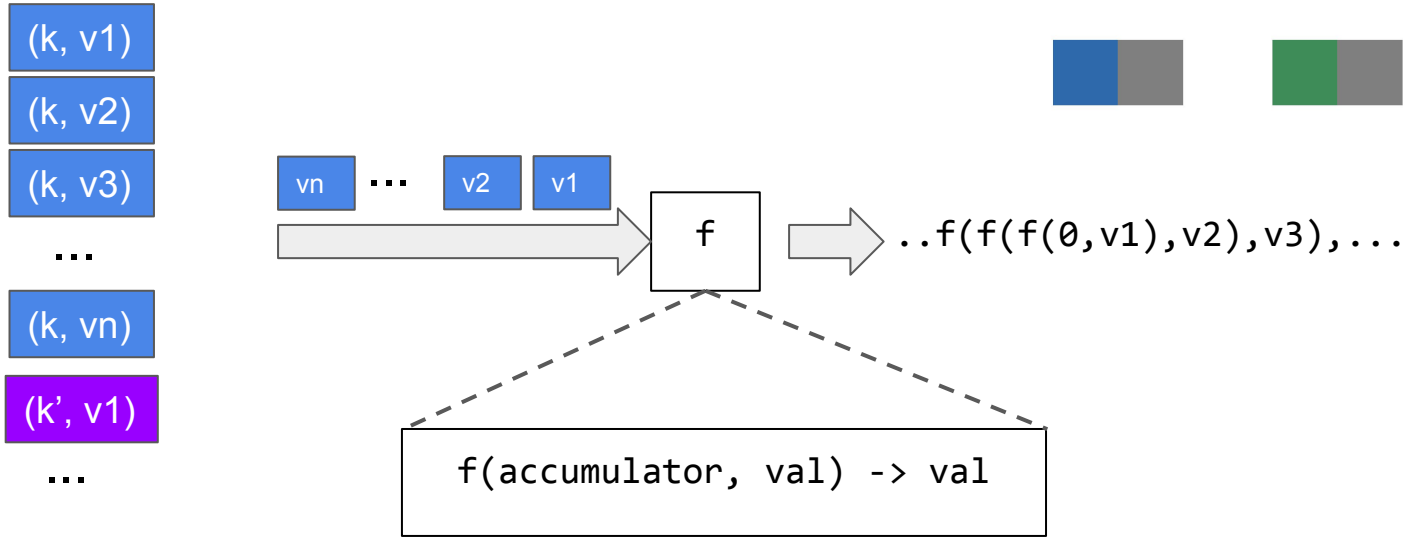
- reduceByKey:
  - ***Similar*** to Reduce function in MapReduce
  - Input: RDD of key-value pairs, reduce function f
  - Output: RDD of key-value pairs where values for each key is aggregated using f



Transformation

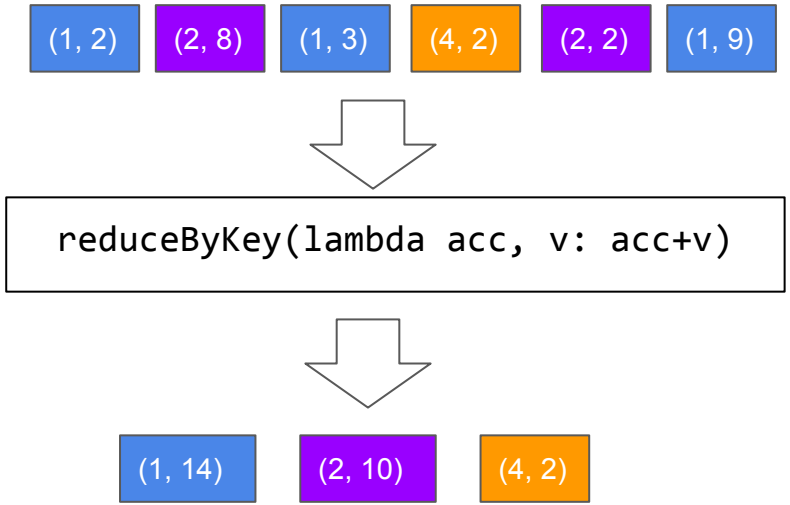
- reduceByKey
  - f is an accumulator function

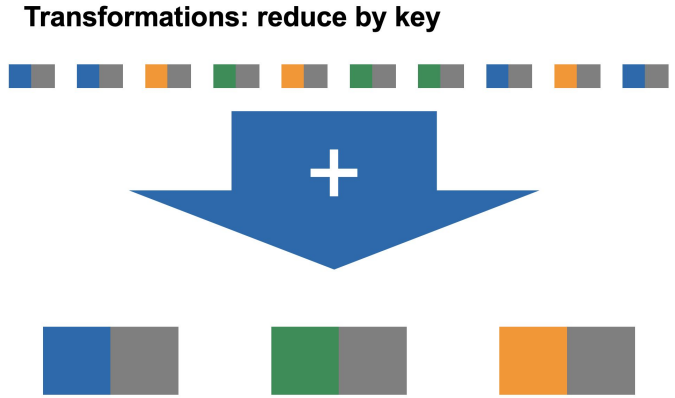




Transformation

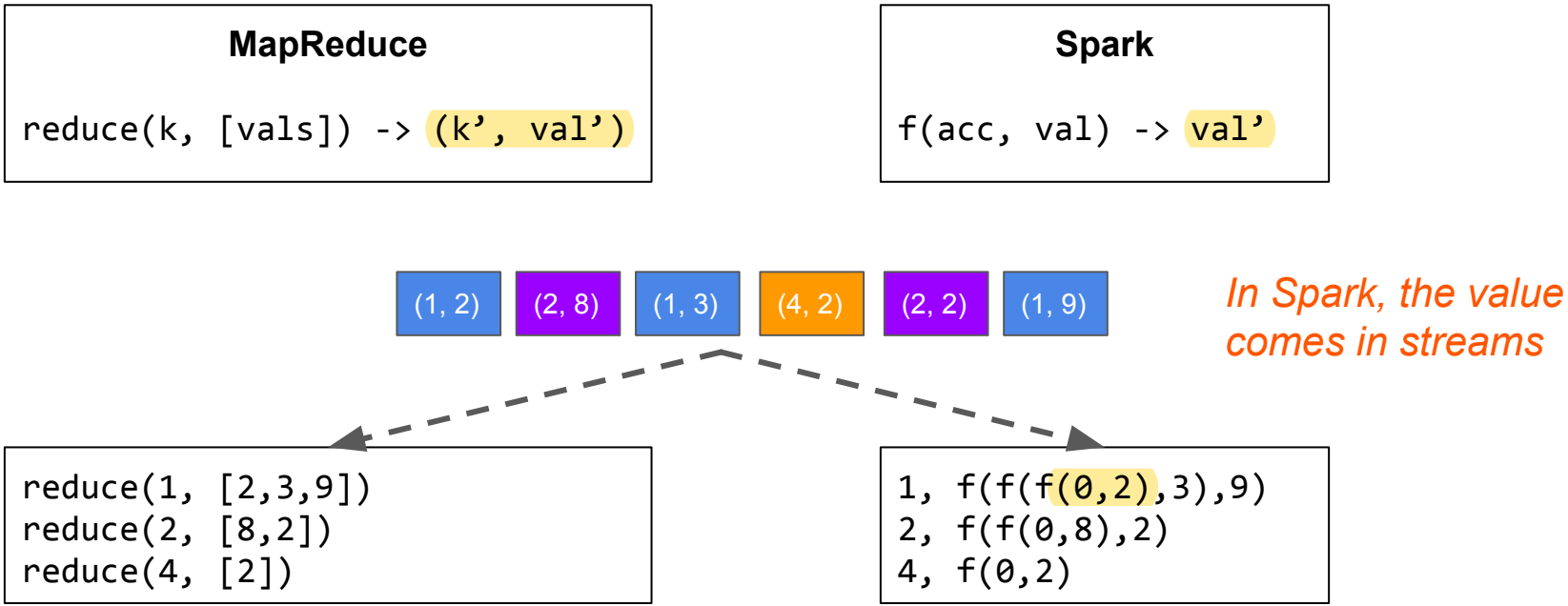
- reduceByKey
  - Example





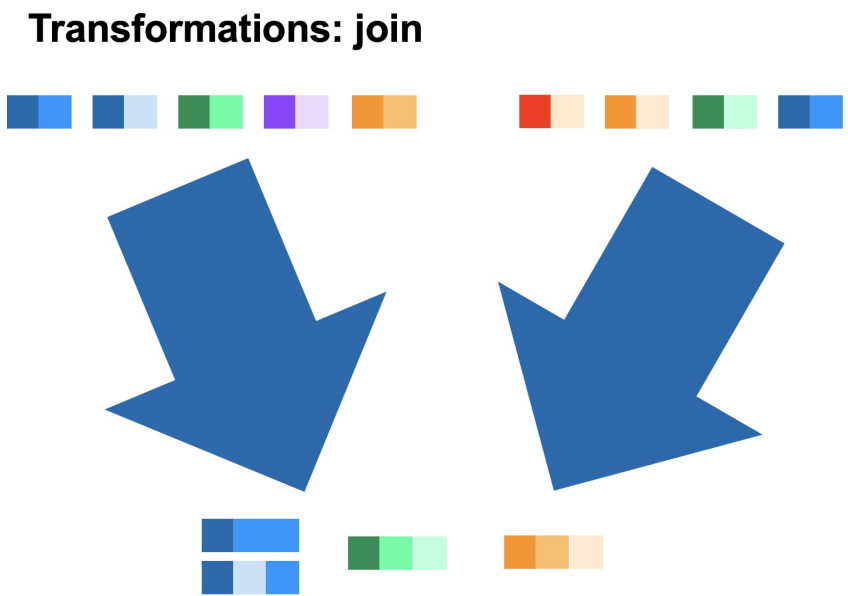
Transformation

- MapReduce's reduce vs. Spark's reduceByKey



Transformation

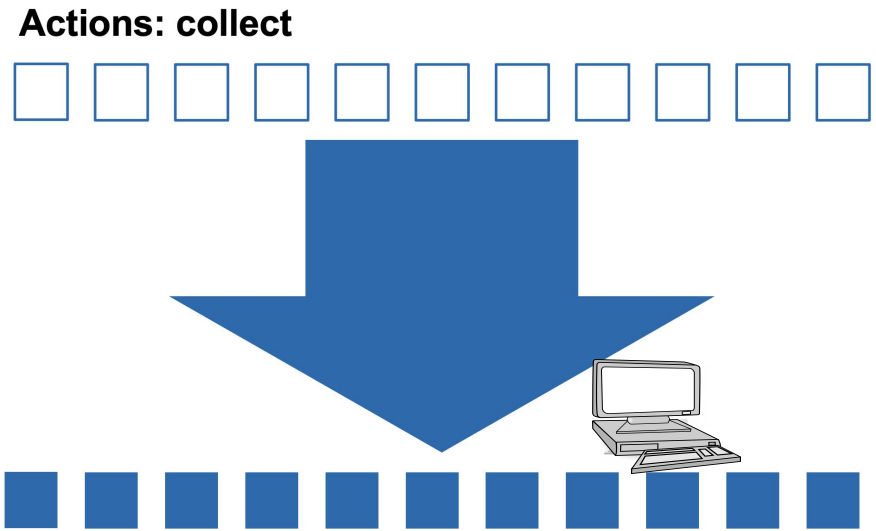
- Join:
  - Input: two RDDs of key-value pairs
  - Output: joined by key
  - Just like SQL join on 2 tables
    - Each table has 2 columns



Spark APIs - Action

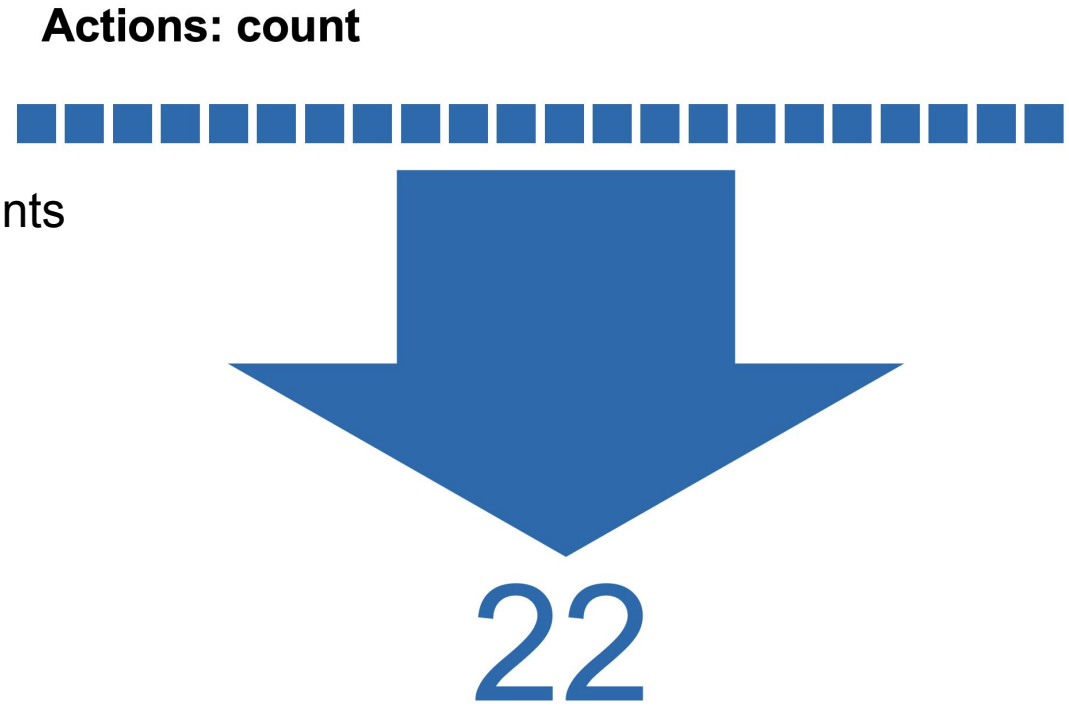
Action

- Collect
  - See all values



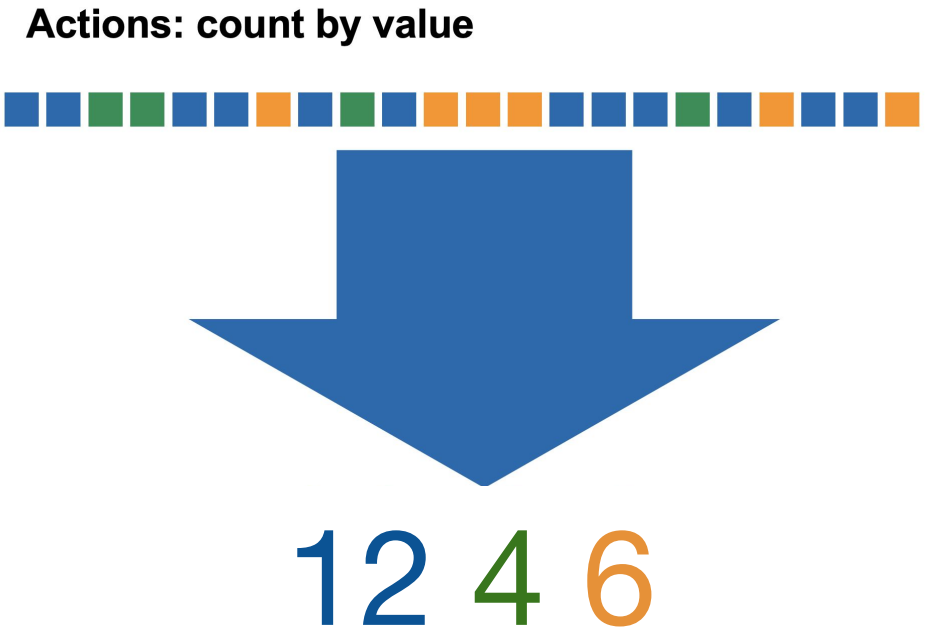
Action

- Count
  - Return number of elements



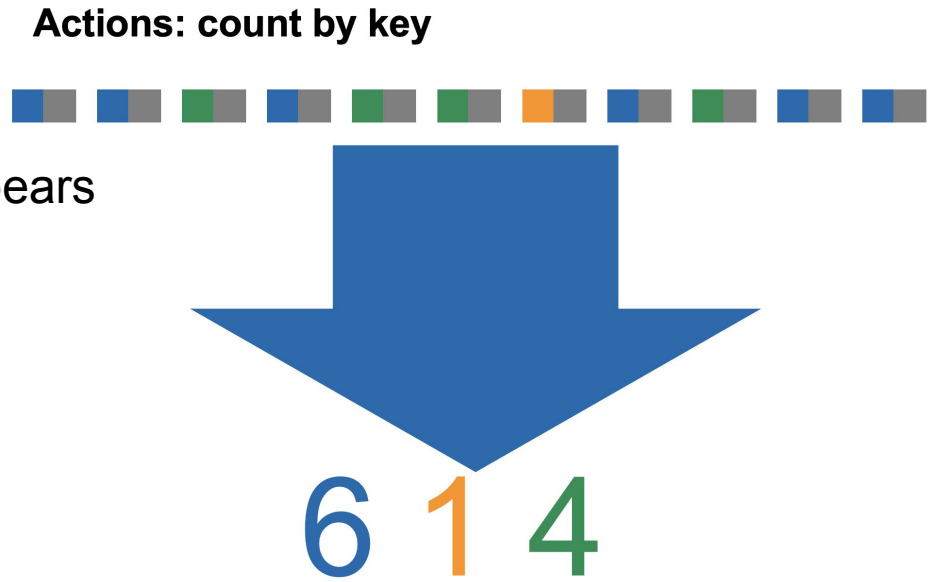
Action

- Count by value
  - How many times a value appears



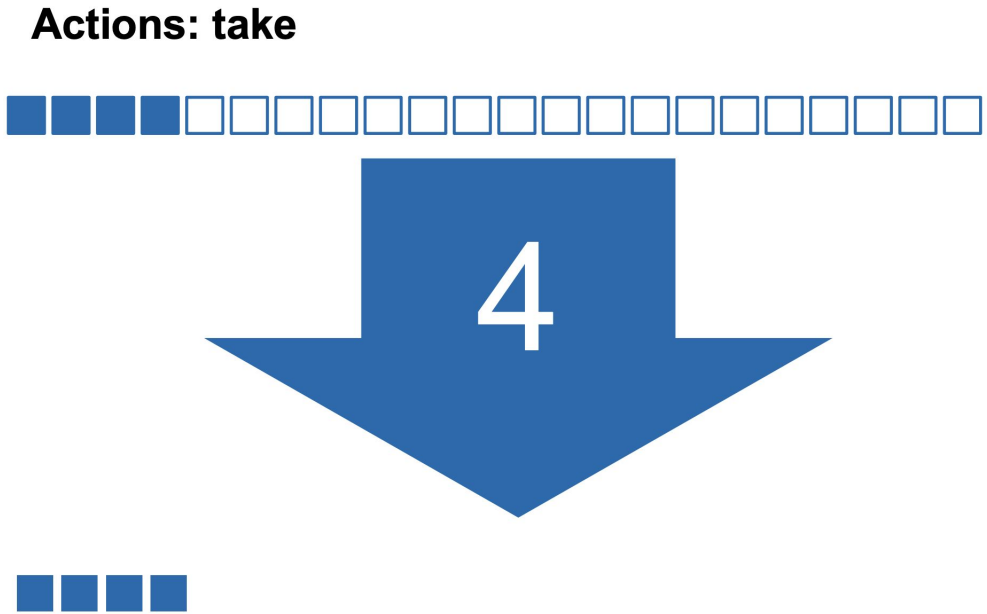
Action

- Count by key
  - How many times a key appears



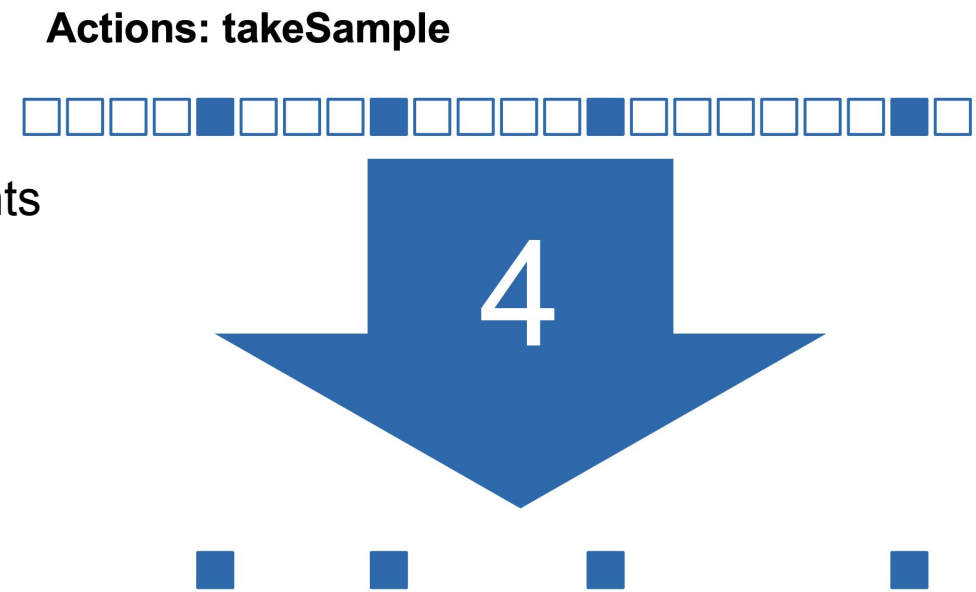
Action

- Take
  - Return a given number of elements



Action

- takeSample
  - Sample a number of elements



APIs Summary

- Many more APIs
- See https://spark.apache.org/docs/latest/rdd-programming-guide.html

Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R) and pair RDD functions doc (Scala, Java) for details.

Transformation	Meaning
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).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
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.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*])	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number

Actions

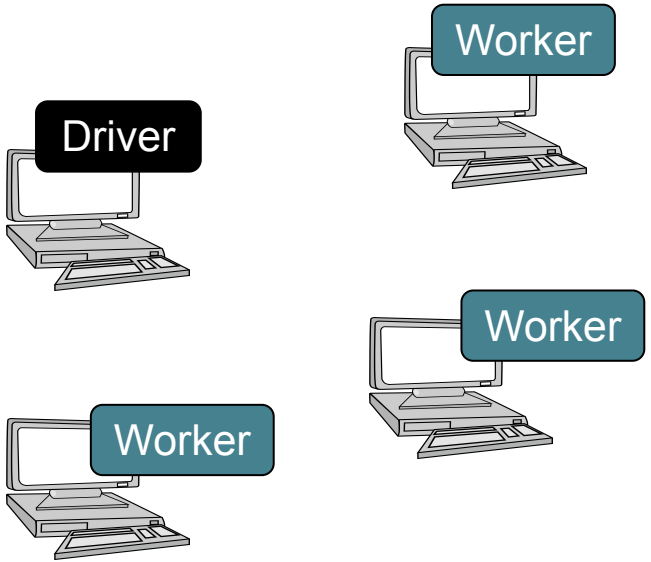
The following table lists some of the common actions supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R)

and pair RDD functions doc (Scala, Java) for details.

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should 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. This is 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.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
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*) (Java and Scala)	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. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.

Example

- Task: log mining
  - Given a large number of log records
  - Filter out ERROR message
  - Interactively look for various pattern



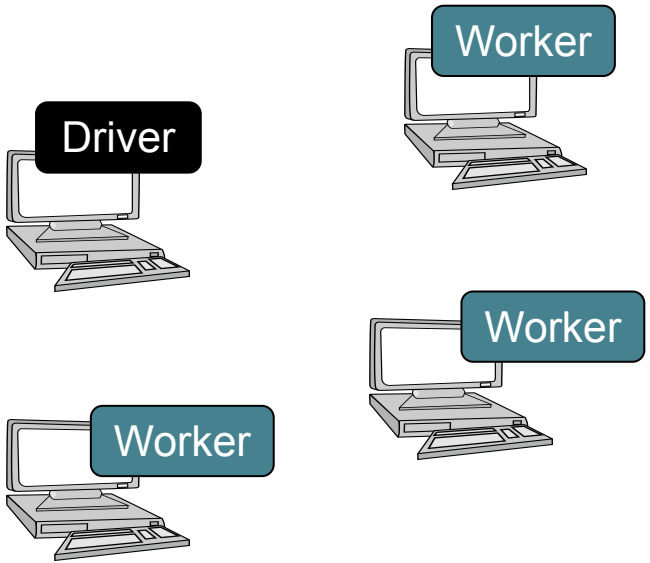
Example

Spark program / shell

Base RDD

```
lines = spark.textFile(“hdfs://…”)
```

Nothing happens yet!

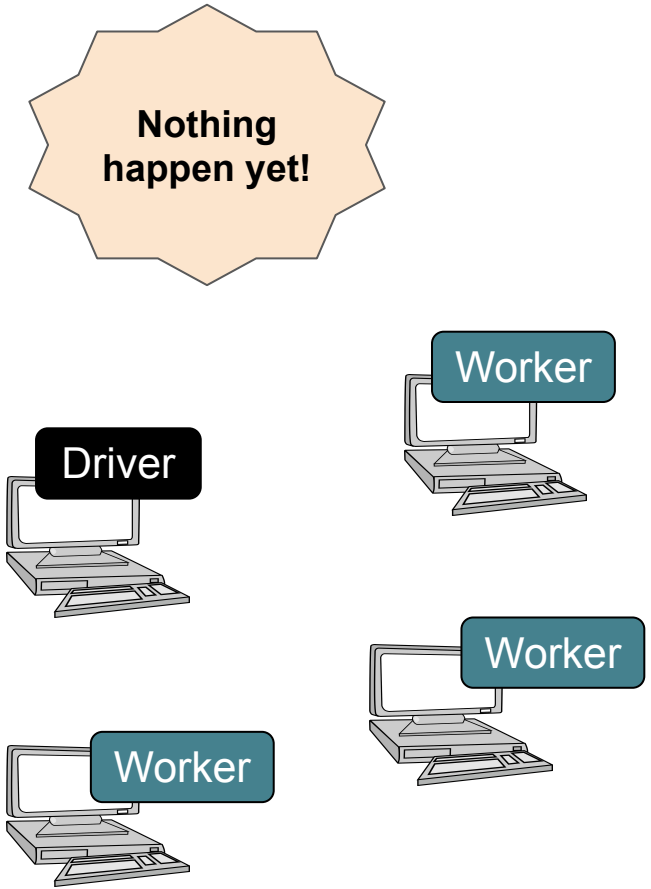


Example

Spark program / shell

Transformed RDD

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))
```

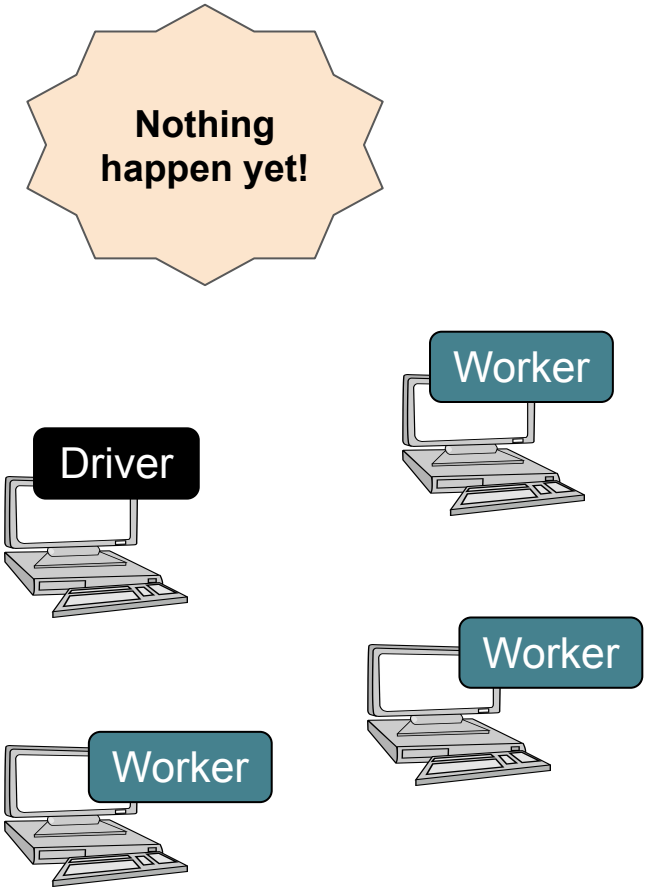


Example

Spark program / shell

Transformed RDD

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])
```

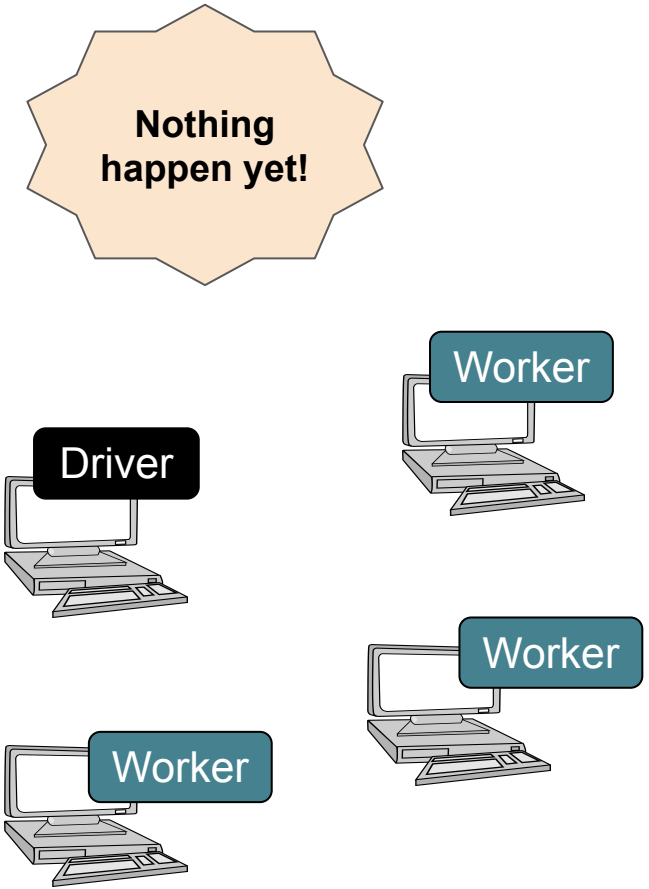


Example

Spark program / shell

Next lecture

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])

messages.persit()
```

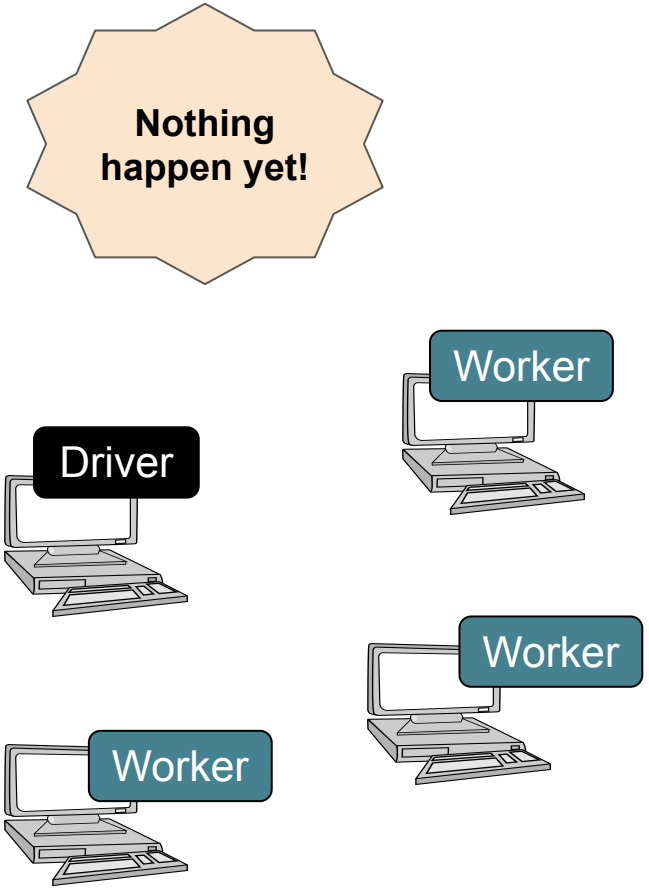


Example

Spark program / shell

Transformed RDD

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])

messages.persit()

mysqlMsg = messages.filter(lambda x: “mysql” in x)
```



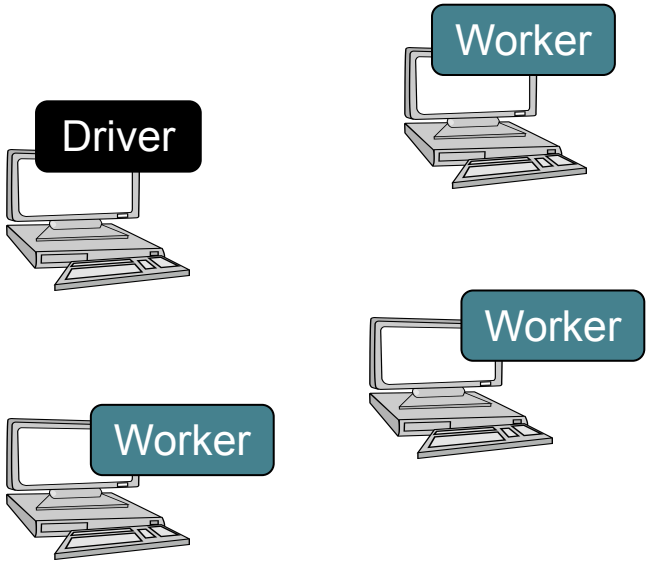
Example

Spark program / shell

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])

messages.persit()

mysqlMsg = messages.filter(lambda x: “mysql” in x)

mysqlMsg.count()
```

Action!

Something ABOUT to happen!

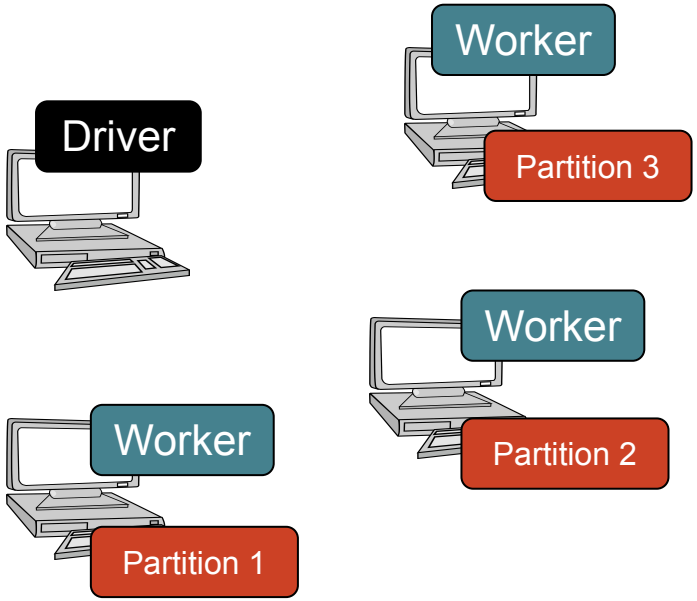


Example

Spark program / shell

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])

messages.persit()

mysqlMsg = messages.filter(lambda x: “mysql” in x)

mysqlMsg.count()
```

Action!

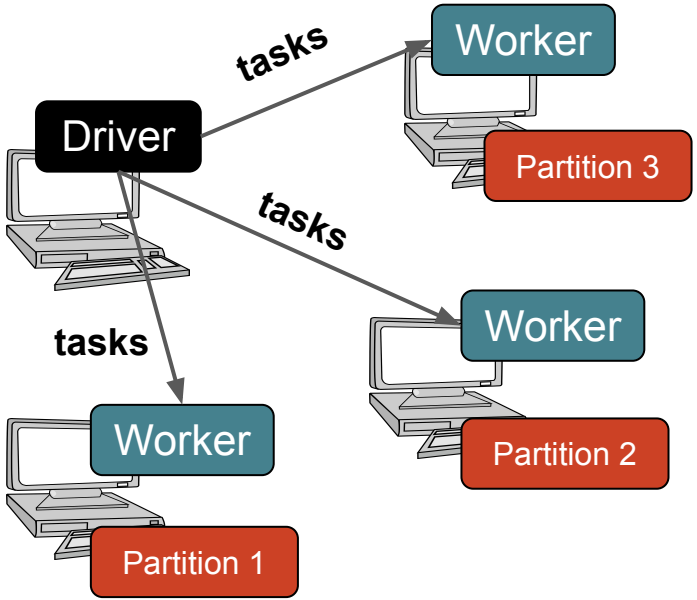


Example

Spark program / shell

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])

messages.persit()

mysqlMsg = messages.filter(lambda x: “mysql” in x)

mysqlMsg.count()
```

Action!

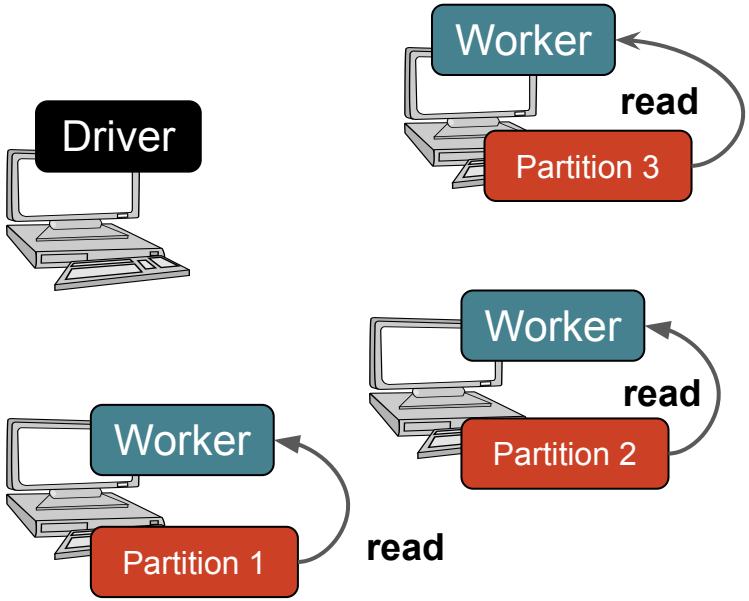


Example

Spark program / shell

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])

messages.persit()

mysqlMsg = messages.filter(lambda x: “mysql” in x)

mysqlMsg.count()
```

Action!

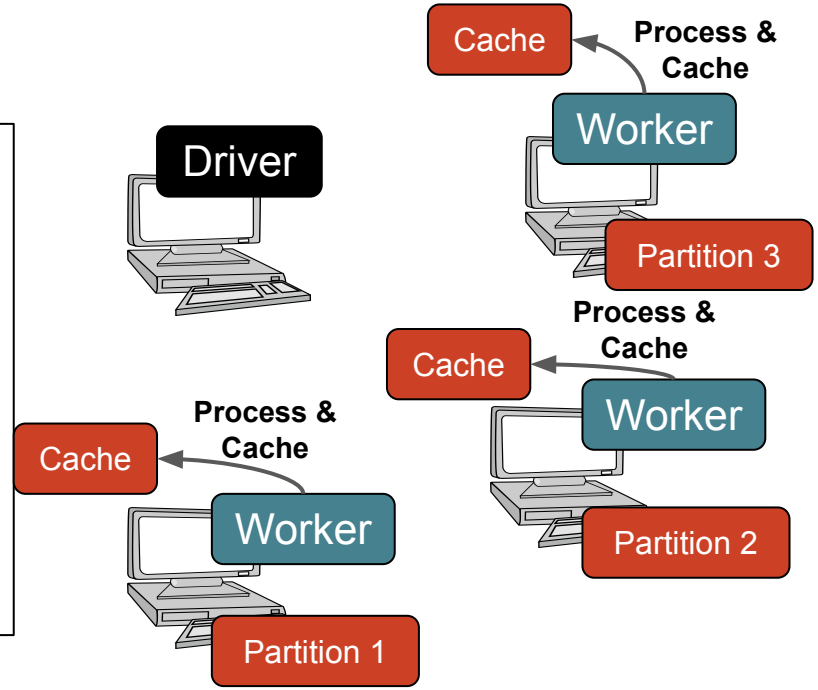


Example

Spark program / shell

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])

messages.persit()

mysqlMsg = messages.filter(lambda x: “mysql” in x)

mysqlMsg.count()
```

Action!

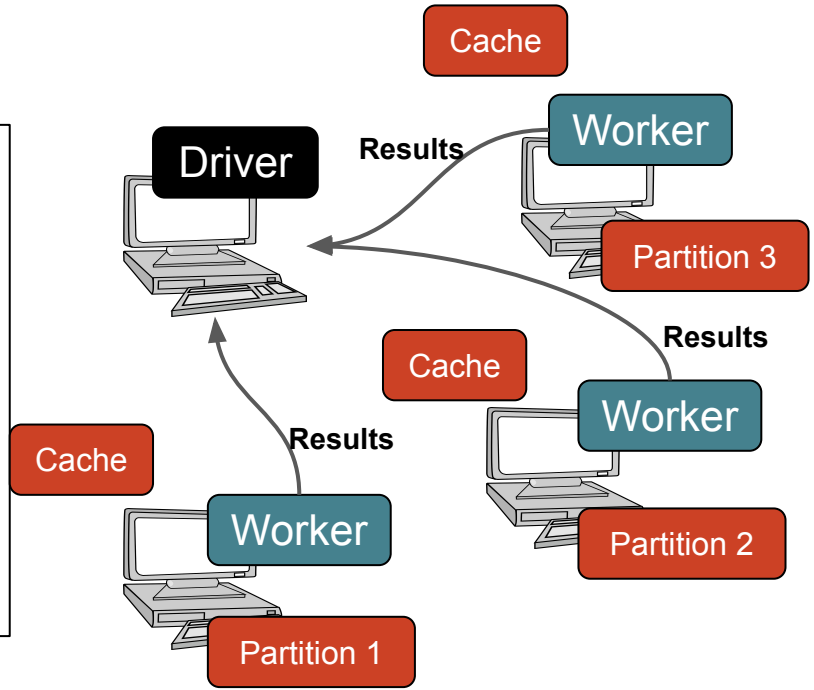


Example

Spark program / shell

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])

messages.persit()

mysqlMsg = messages.filter(lambda x: “mysql” in x)

mysqlMsg.count()
```

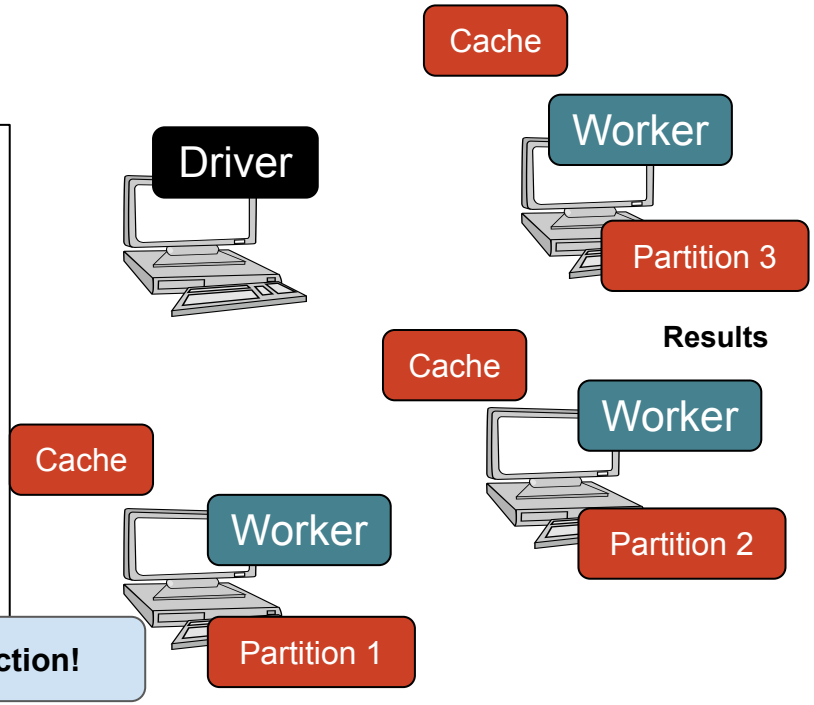
Action!



Example

Spark program / shell

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])

messages.persit()

mysqlMsg = messages.filter(lambda x: “mysql” in x)

mysqlMsg.count()

messages.filter(lambda x: “mongodb” in x).count()
```

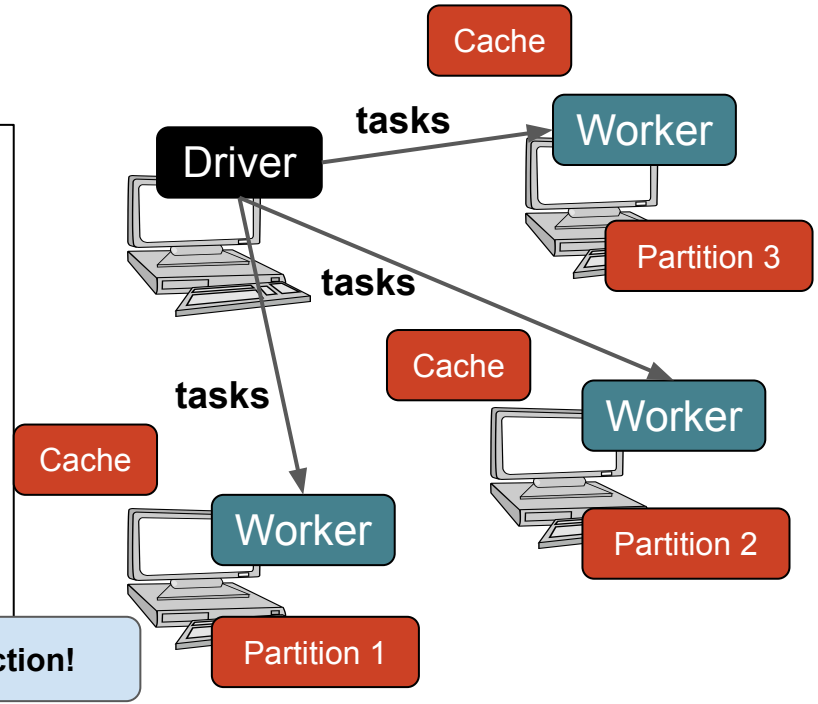


Example

Spark program / shell

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])

messages.persit()

mysqlMsg = messages.filter(lambda x: “mysql” in x)

mysqlMsg.count()

messages.filter(lambda x: “mongodb” in x).count()
```

Action!

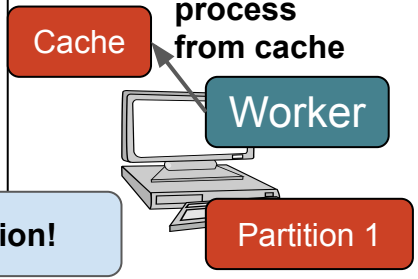


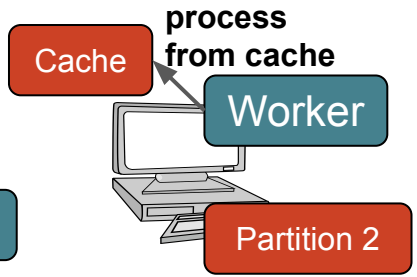
Example

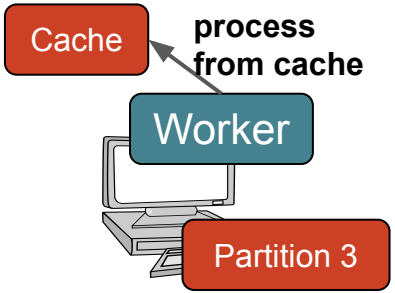
Spark program / shell

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])

messages.persit()

mysqlMsg = messages.filter(lambda x: “mysql” in x)

mysqlMsg.count()

messages.filter(lambda x: “mongodb” in x).count()
```

Action!





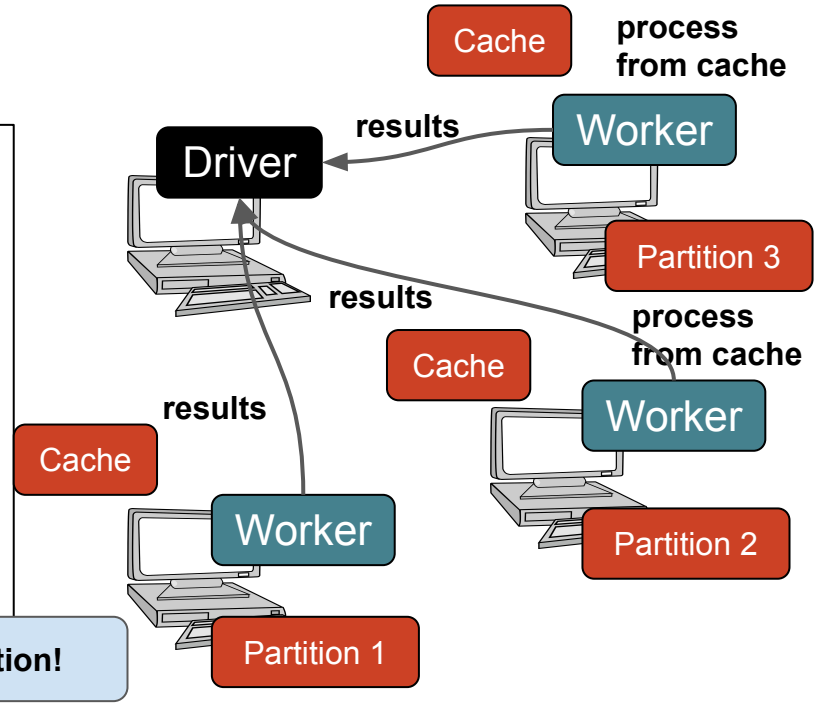


Example

Spark program / shell

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])

messages.persit()

mysqlMsg = messages.filter(lambda x: “mysql” in x)

mysqlMsg.count()

messages.filter(lambda x: “mongodb” in x).count()
```

Action!



Example

Cache your data ➔ Faster Results

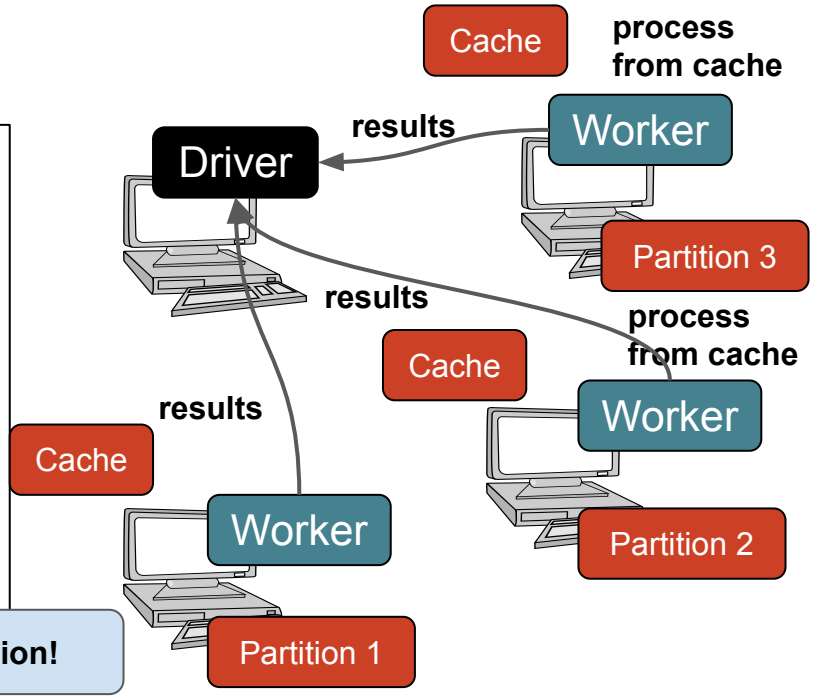
Full-text search of Wikipedia

- 60GB on 20 EC2 machines
- 0.5 sec from mem vs. 20s for on-disk

Spark program / shell

```
lines = spark.textFile(“hdfs://…”)

errors = lines.filter(lambda x: x.startswith(“ERROR”))

messages = erros.map(lambda x: x.split(“\t”)[2])

messages.persit()

mysqlMsg = messages.filter(lambda x: “mysql” in x)

mysqlMsg.count()

messages.filter(lambda x: “mongodb” in x).count()
```

Action!



Demo

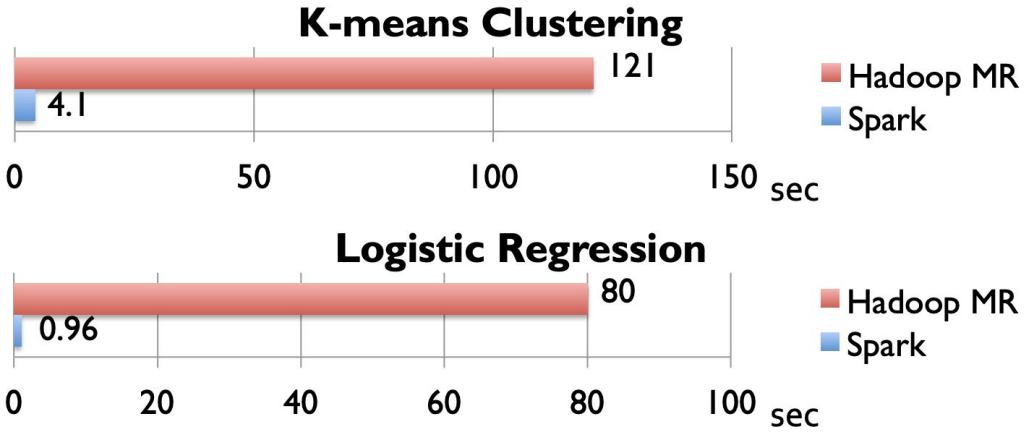
- 3 nodes Spark cluster
  - 0.9GB text file
- Wordcount
- Text search
  - Search for lines with word “excellent”
  - Search for other words within those lines

find lines with both excellent and good
.filter(...).filter(...)

(lambda x: 'excellent' in x)

Summary

- Spark much faster than MapReduce
- Especially for
  - Iterative computation
  - Interactive computation
- Key techniques:
  - Caching of RDDs
  - Lazy evaluation



Question 1

- Why are RDDs called *immutable* if there are transformation operations on RDDs?

Question 2

- Is Spark strictly better than MapReduce in all use cases?
    - In other words, why do people still use MapReduce?

- RAM
if you don't have enough RAM, Spark is not as good because it takes advantage of RAM to speed up computation.
- One Pass Jobs (More fundamental)
If your job only takes one pass over data, still need to load all the data from disk. Spark has no fundamental advantage over MapReduce. MapReduce is designed specifically for one pass. (e.g. indexing)
(If more than one pass, Spark is strictly better because they cache the results, faster)