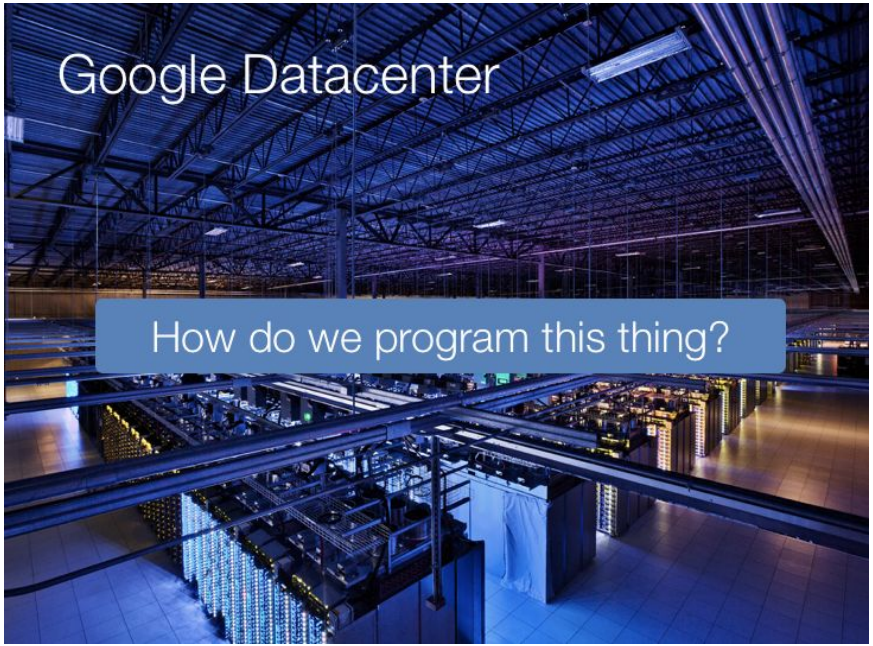
Databases and Big Data

Spark 2

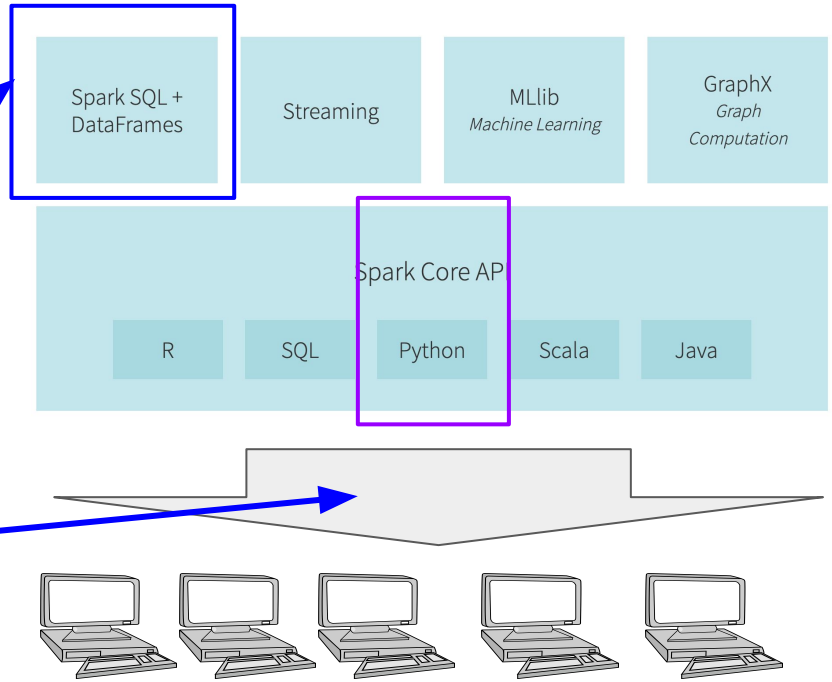
Recap

- Spark much faster than MapReduce
- Designed for
  - Iterative computation
  - Interactive computation
- Exploit abundant memory
- Lazy evaluation



Recap

- What we have seen:
  - APIs
  - High-level execution flow
  - Examples
- Today:
  - Deeper dive into execution
  - Other libraries



Spark

- Core APIs

```
map
filter
groupBy
sort
union
join
leftOuterJoin
rightOuterJoin
```

```
reduce
count
fold
reduceByKey
groupByKey
cogroup
cross
zip
```

```
sample
take
first
partitionBy
mapWith
pipe
save    ...
```

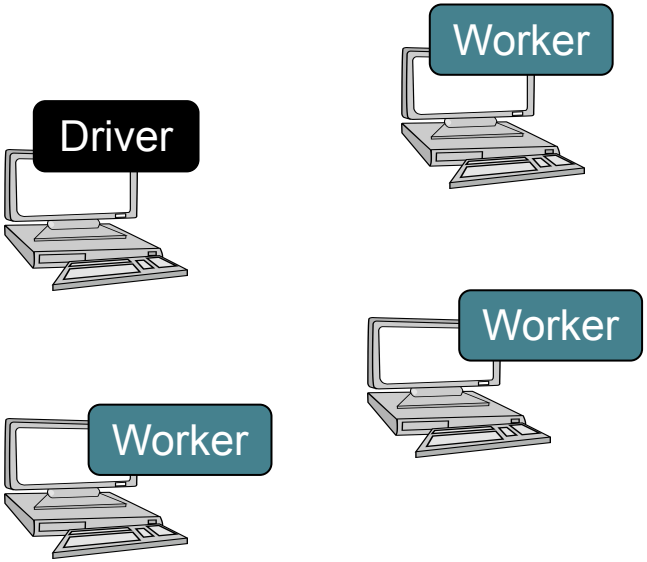
Physical Execution

Physical Execution

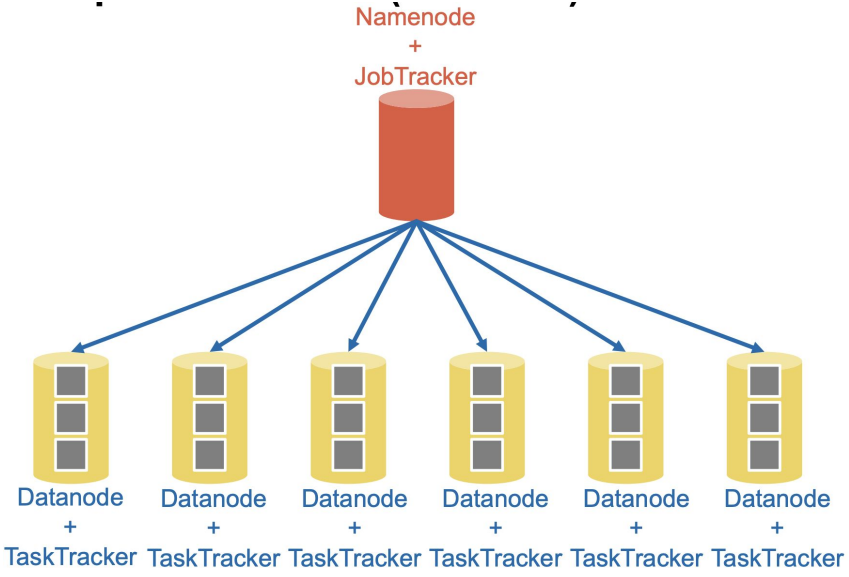
Spark Architecture

- Recall these

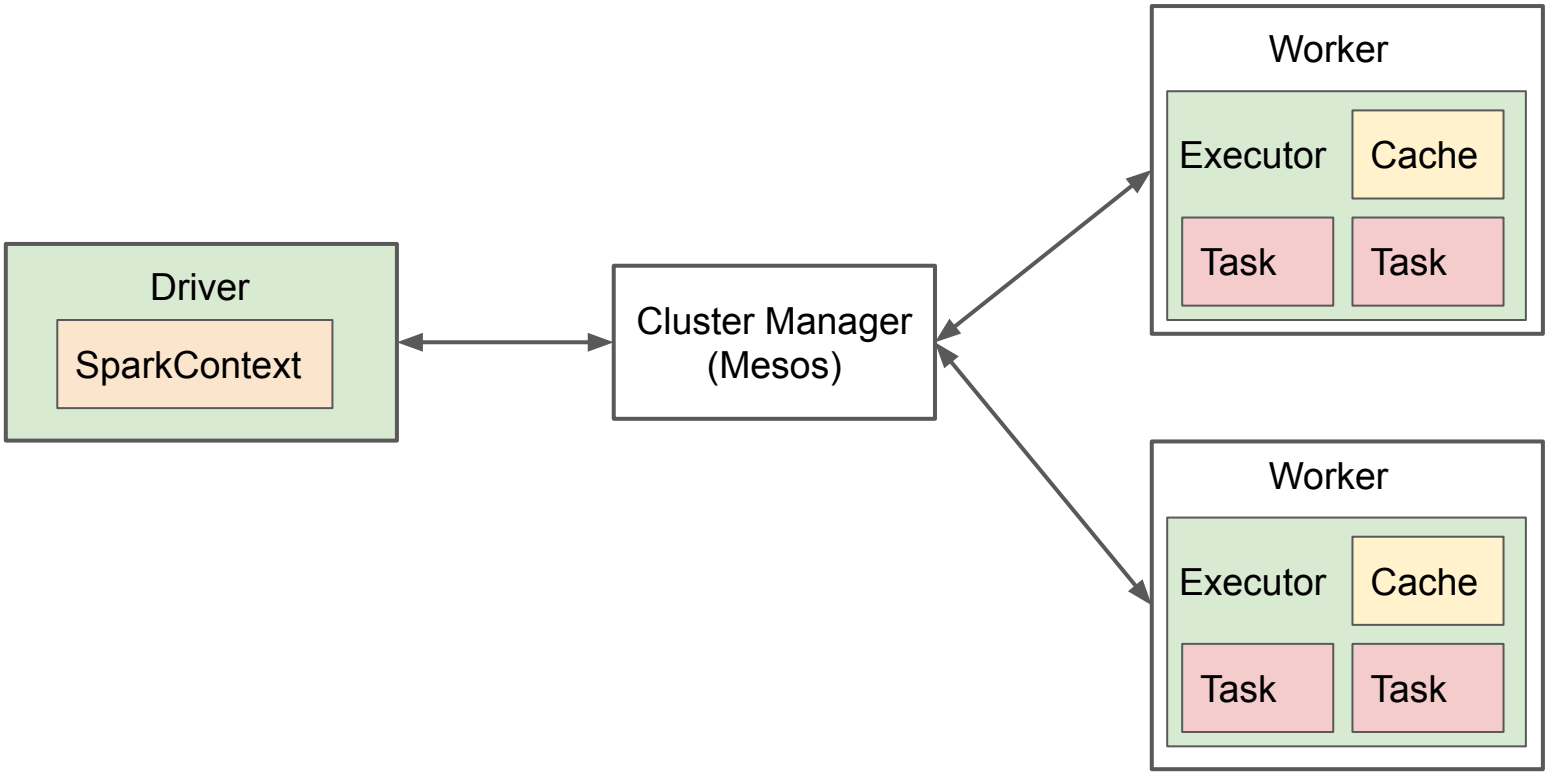
Different terms for Spark vs Hadoop



vs.



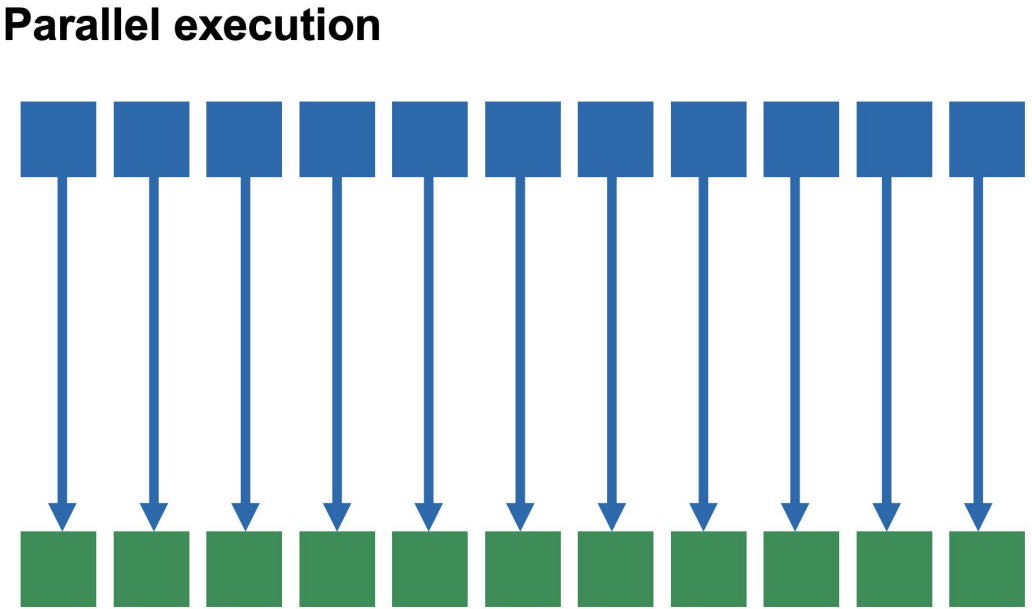
Spark Architecture



Execution

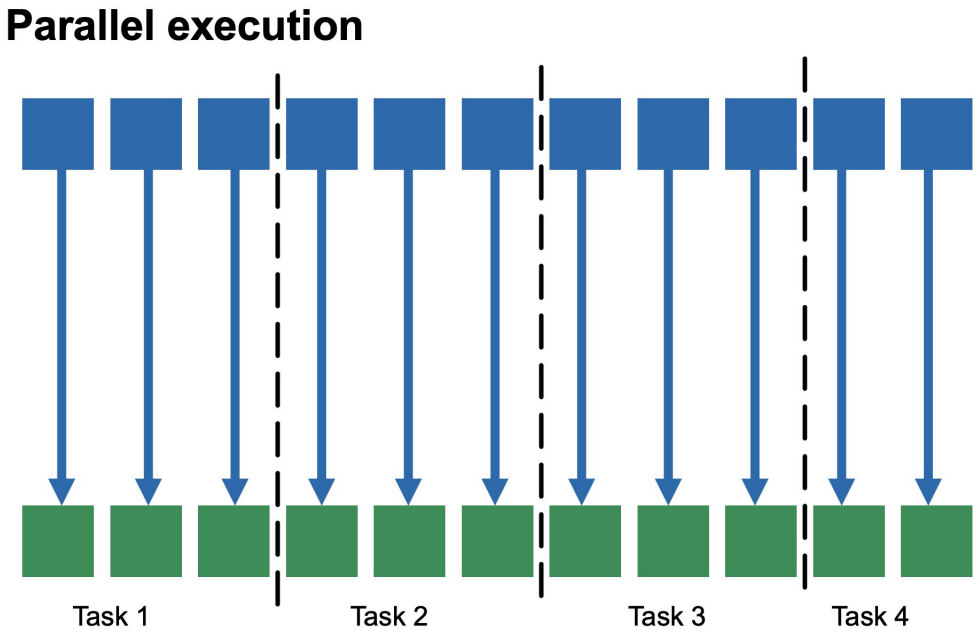
like .map
*can be executed in **parallel***
(no independence so can)

- How to execute this?
  - Embarrassingly parallel



Execution

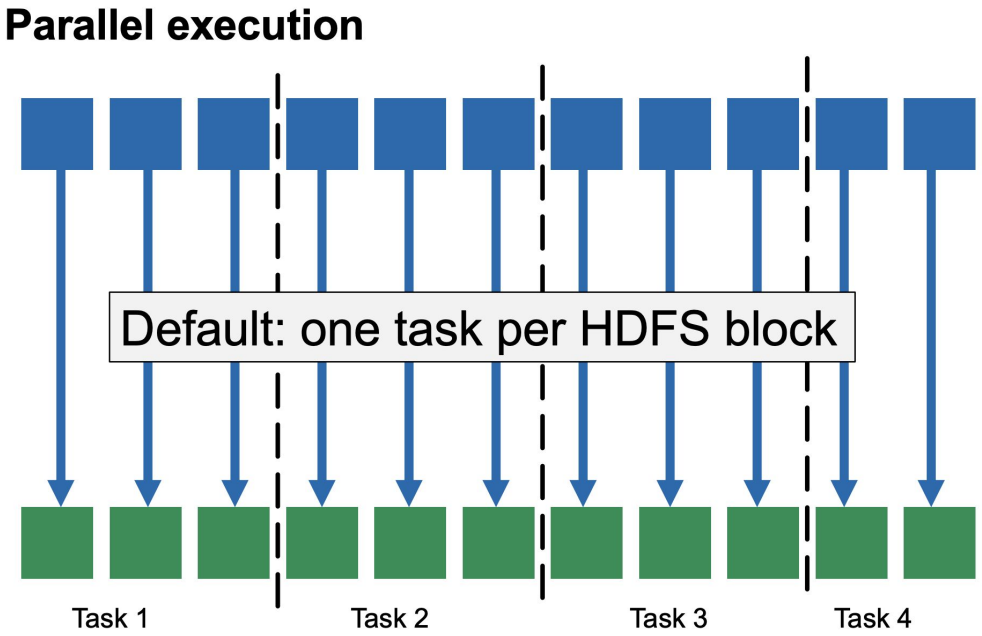
- How to execute this?
  - Embarrassingly parallel



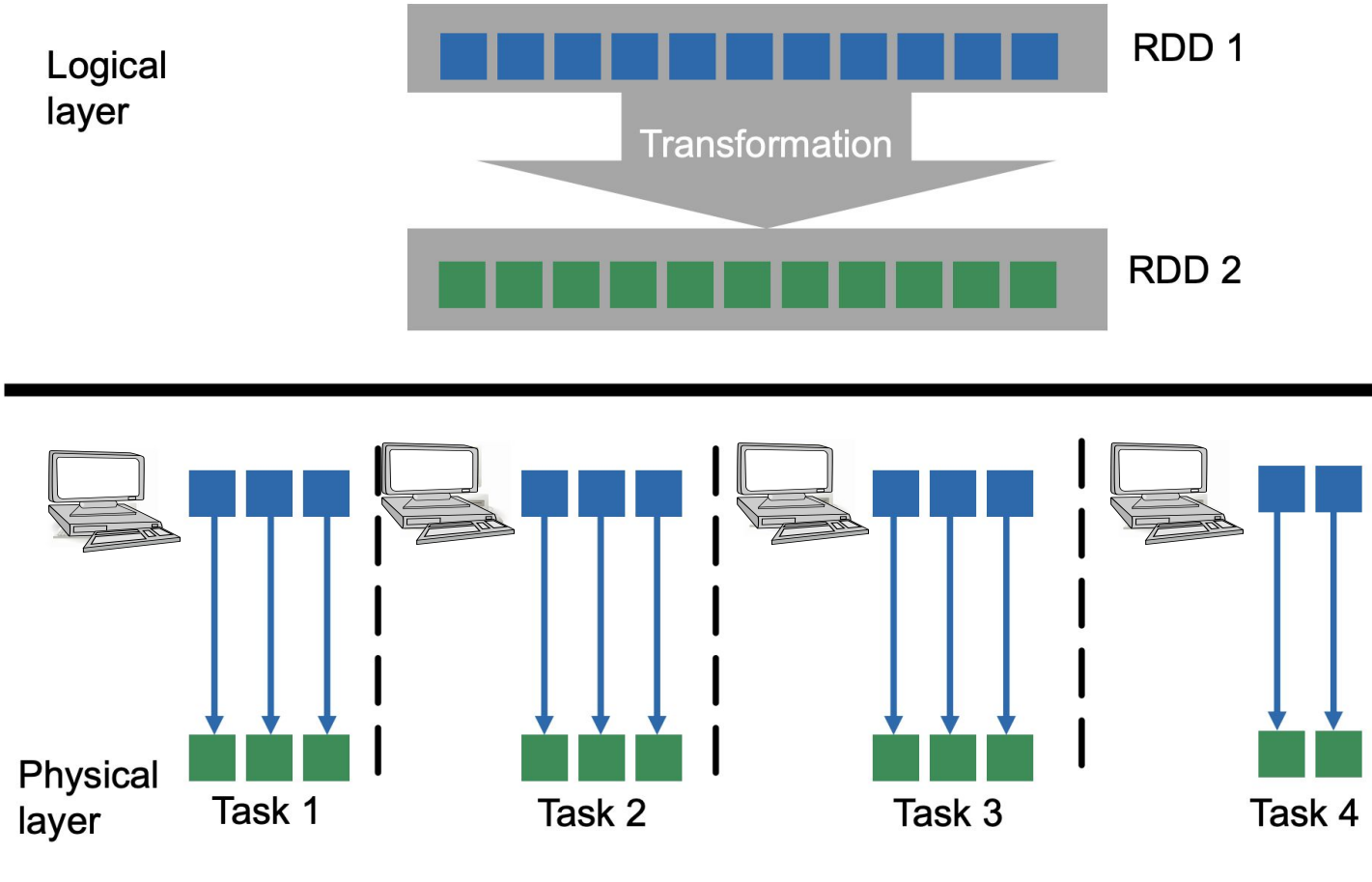
Execution

- Just like MapReduce!

- each partition correspond to 1 hdfs block
- send a task to whatever machine that store this block and execute there.



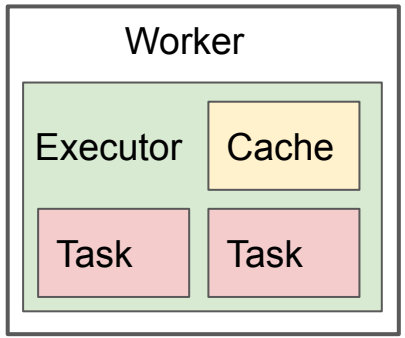
Parallel execution

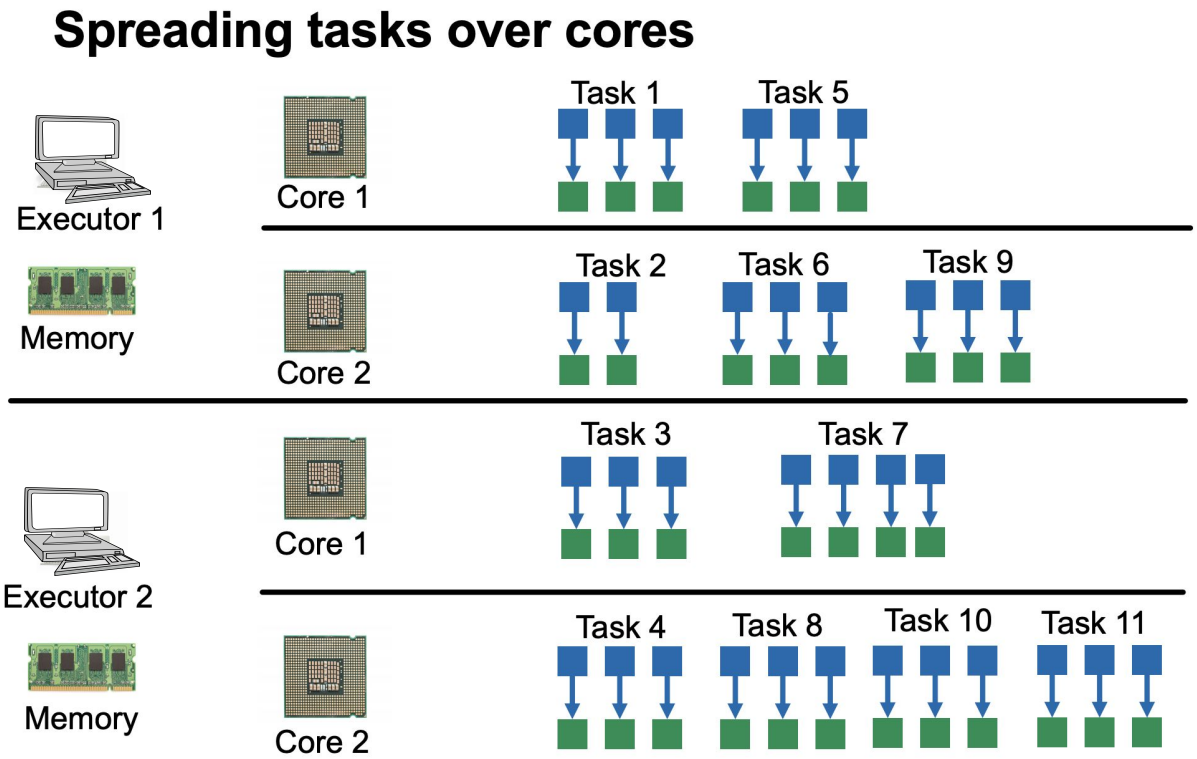


Executor

- memory used for caching
- Worker takes the task and spreads it over all the cores they have

- Execute tasks



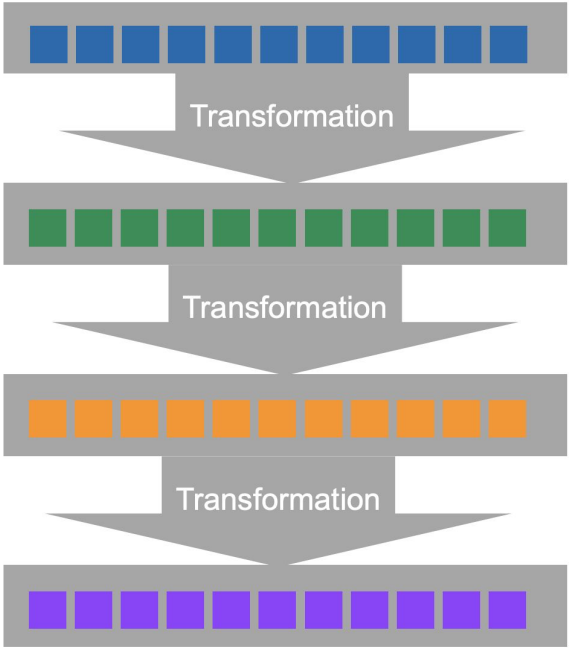


More Complex Execution

- Sequence of
  - Transformations
  - Actions

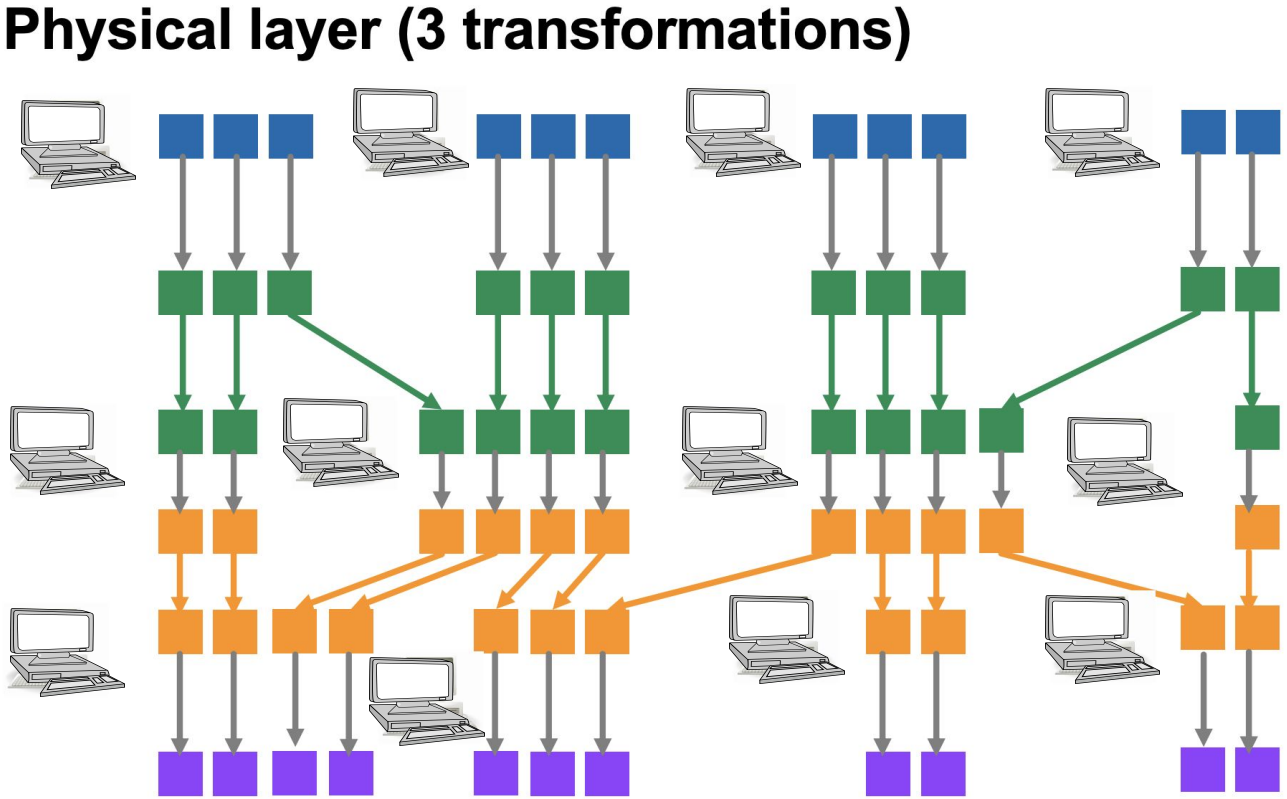
*In each task (different colours), they try to do as **parallel** as possible*
*- In second transformation (green), **cross machine communication** cause there is some input required from previous layer.*
*- Spark delay all these transformation so it will know all the transformation needed for the output and **optimise** and organise based on that so that there is no communication b/w machines (or minimised)*
<<< NEXT SLIDE >>>>

Sequence of (parallelizable) transformations



Complex Execution

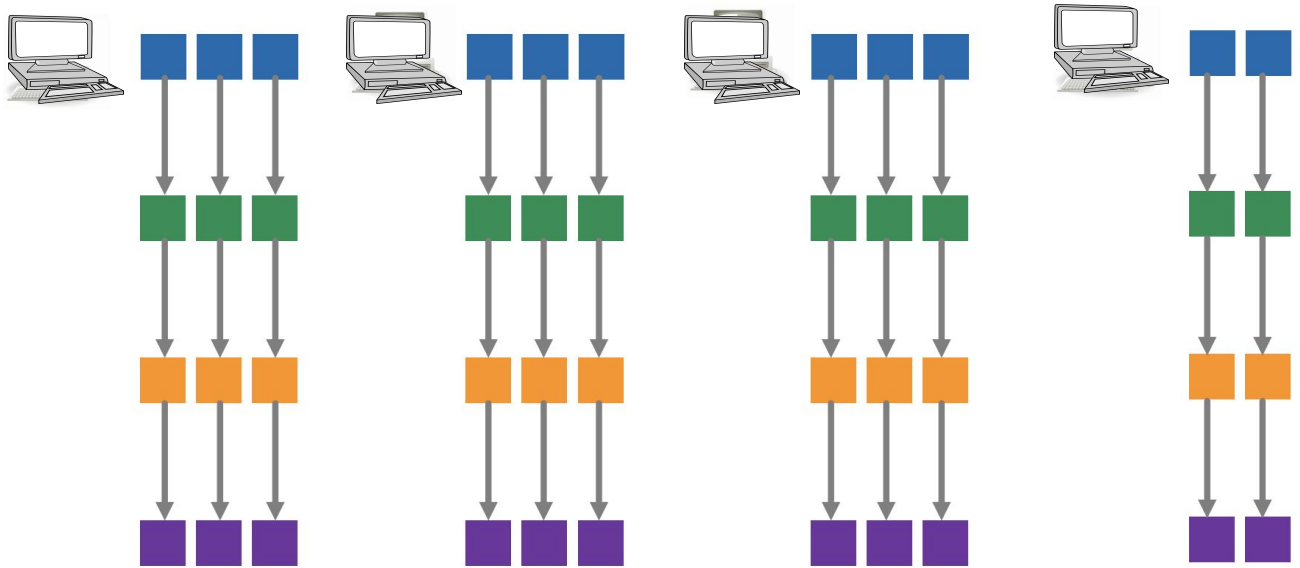
How could this happen?



Complex Execution

Optimization

Better way to organize it!



***Best case** (1 inp -> 1 o/p)*
- 1 op can have many inp but 1 inp only goes to 1 op

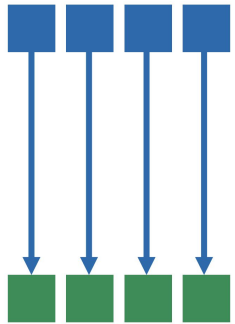
Dependencies

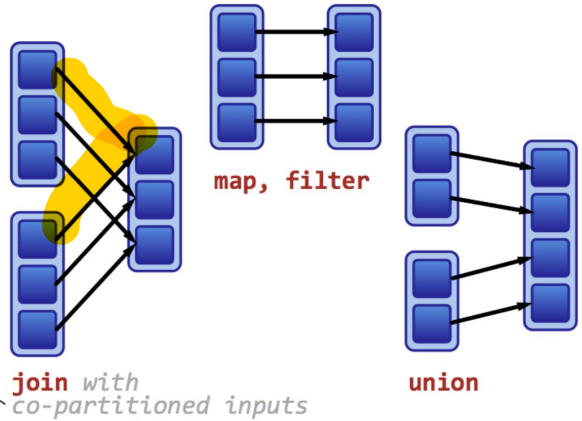
- Between input and output RDDs
- Narrow dependency:

relationship between input and output

  - Input partition used by **at most** 1 output partition
  - Example: map, filter

Narrow Dependency





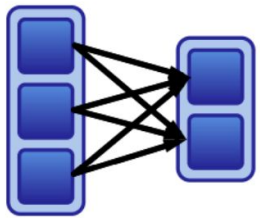
Co-partitioned inputs

RDDs are partitioned using same hash function

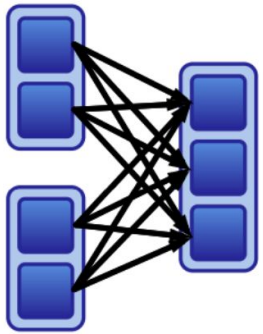
Dependencies

- Between input and output RDDs
- Wide dependency:
  - Shuffle!

Bad but you can't avoid it
E.g. GroupByKey and join (with no co-partition)

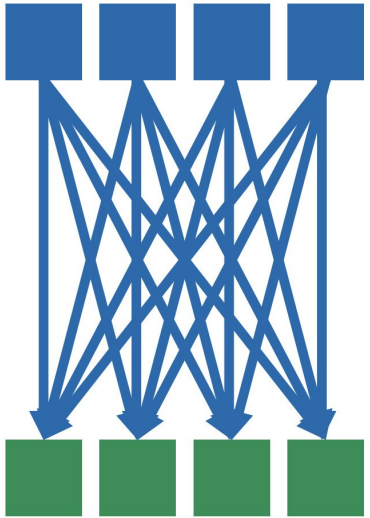


groupByKey



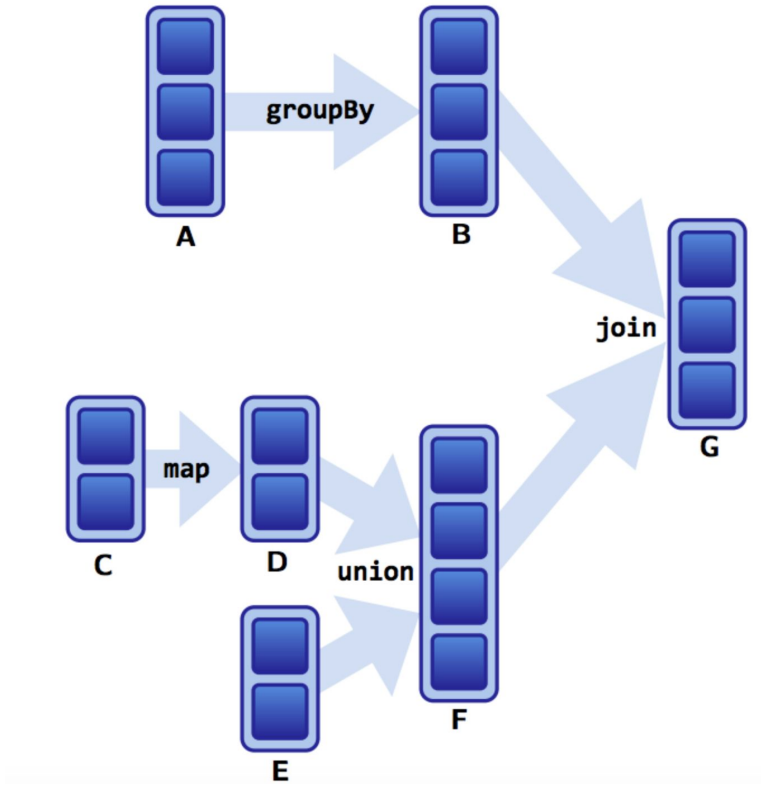
join

Wide Dependency



Scheduler

- Given a program
  - Spark infers dependencies

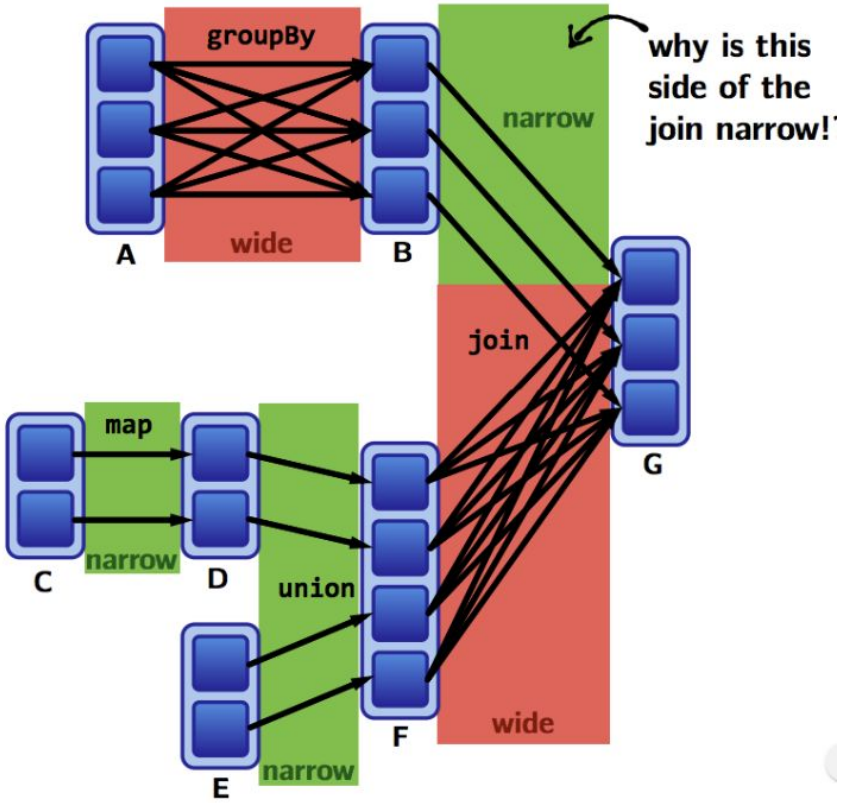


Scheduler

- Given a program
  - Spark infers dependencies

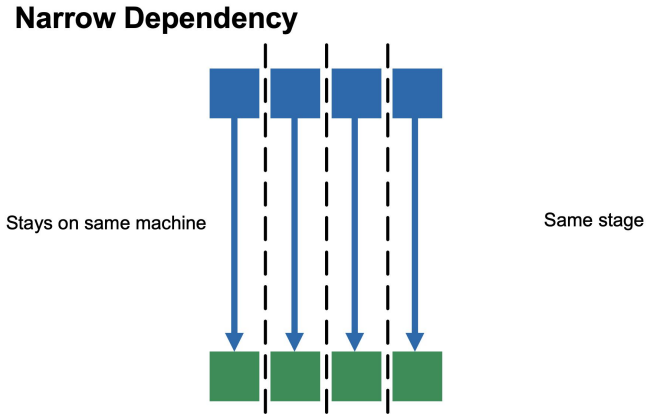
Spark infer dependencies at runtime

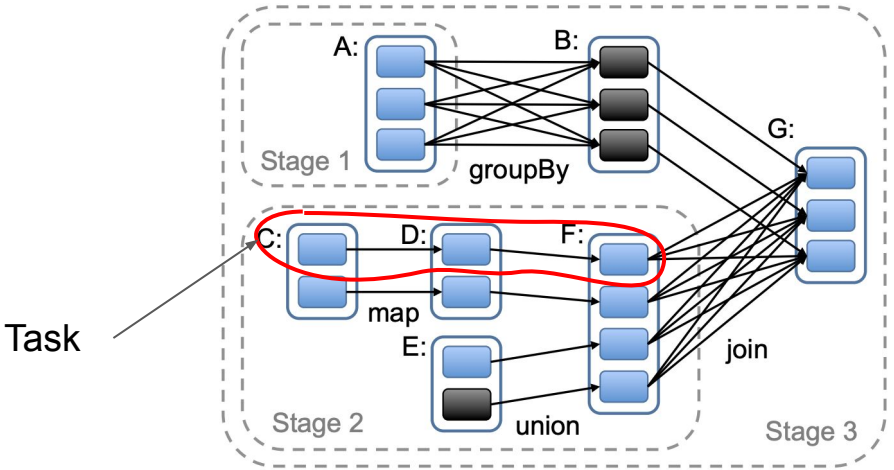
*theres partitions in B cause it is a side effect of the groupBy --> so your join will be **narrow dependency***
(minimise shuffling between B and G since they are partition by the same function)



Scheduler

- Narrow dependencies
  - Grouped into stages
- Stage execution
  - Pipeline
  - Parallel over many machines



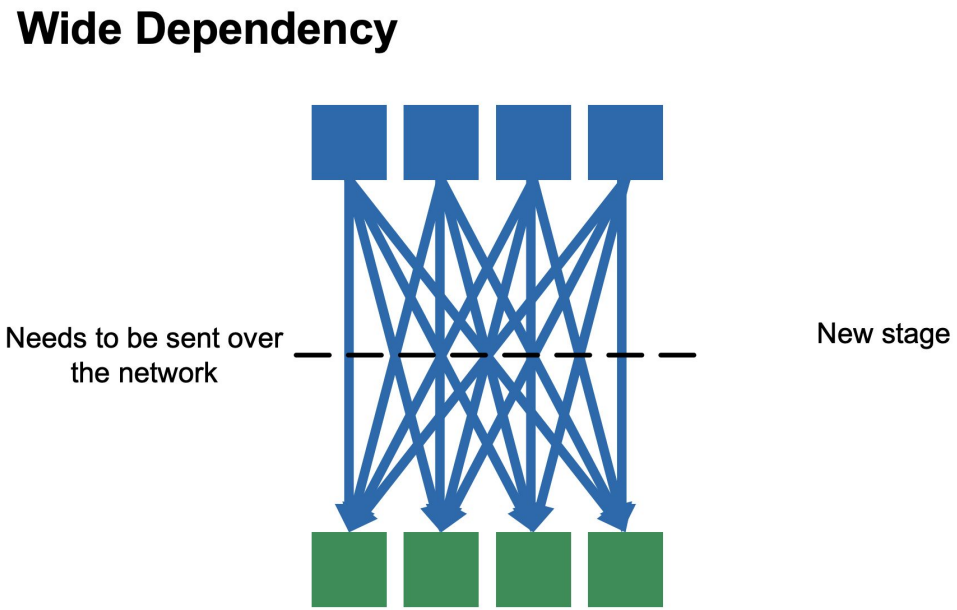


*- A needs to finish before you start B so need to group in **stages**.*
*- Everything in a stage can be done in **parallel** (no dependency)*

Scheduler

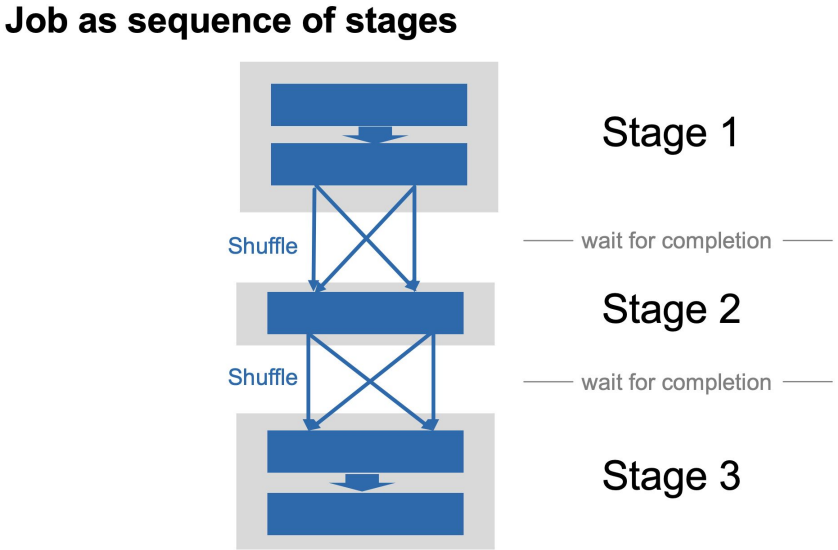
- What about wide dependencies
  - Create new stage

Stage 2 can only start after stage 1 has finished



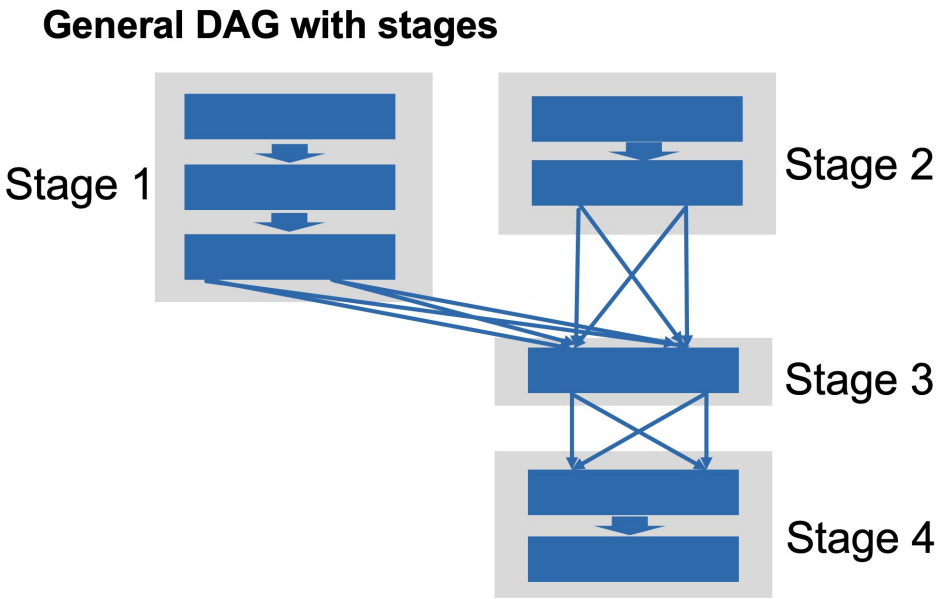
Scheduler

- Job: made up of stages
  - A sequence of stages



Scheduler

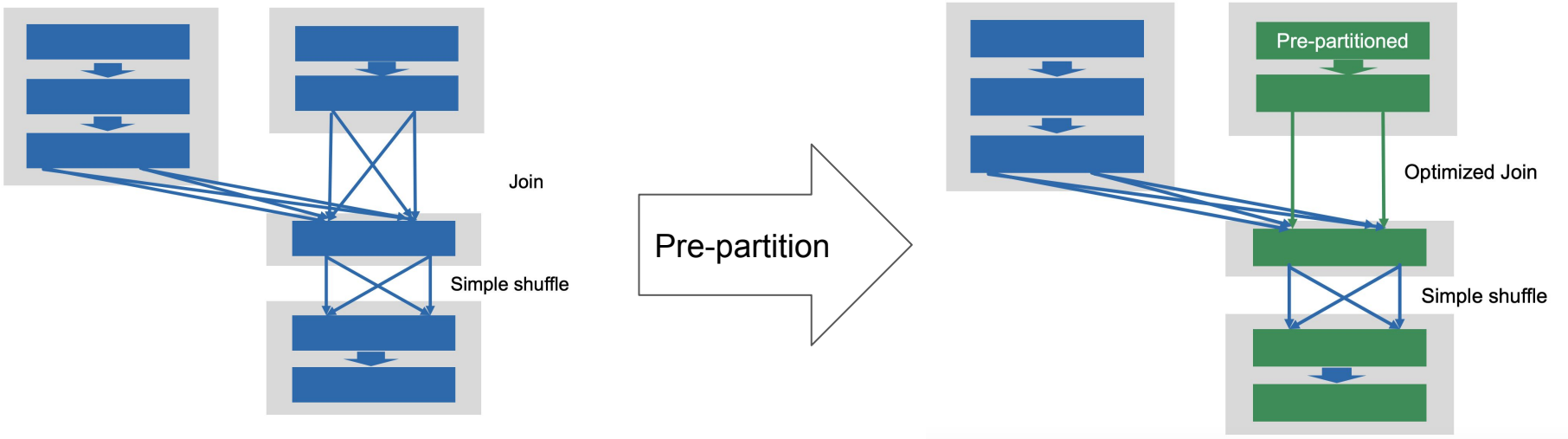
- Job: made up of stages
  - Or a general DAG



Performance Tuning 1

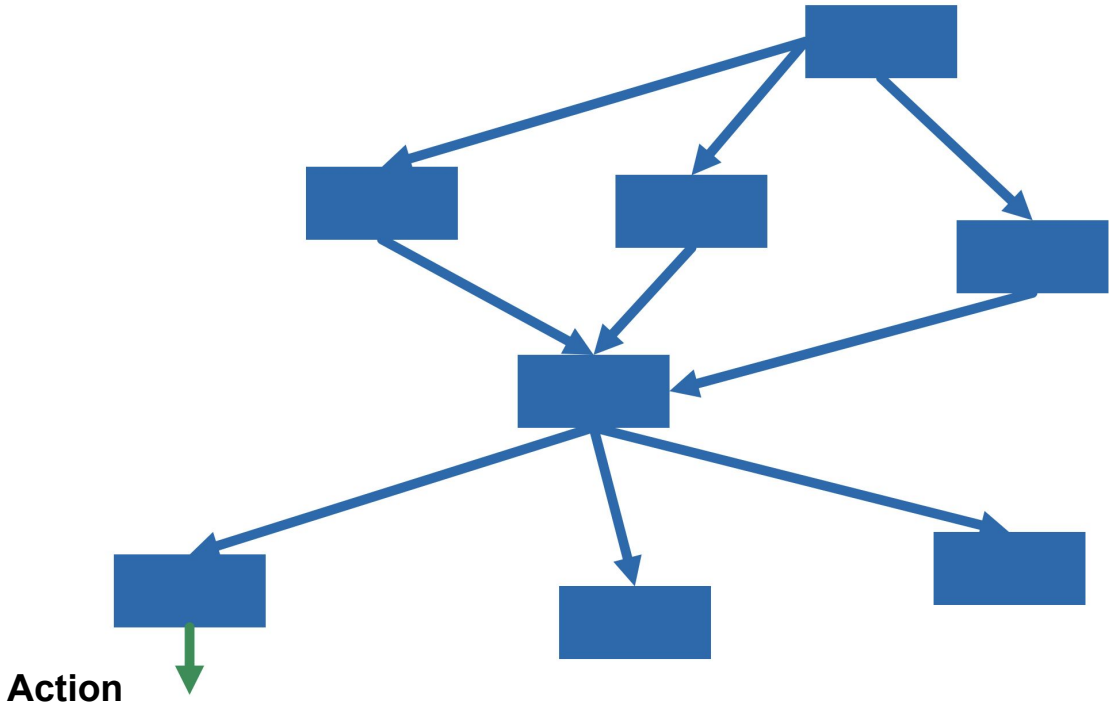
- Wide dependencies are expensive
- Minimize it!

prepartition -> *can use* ***join with co-paritition*** *where there will be a* ***narrow dependency***



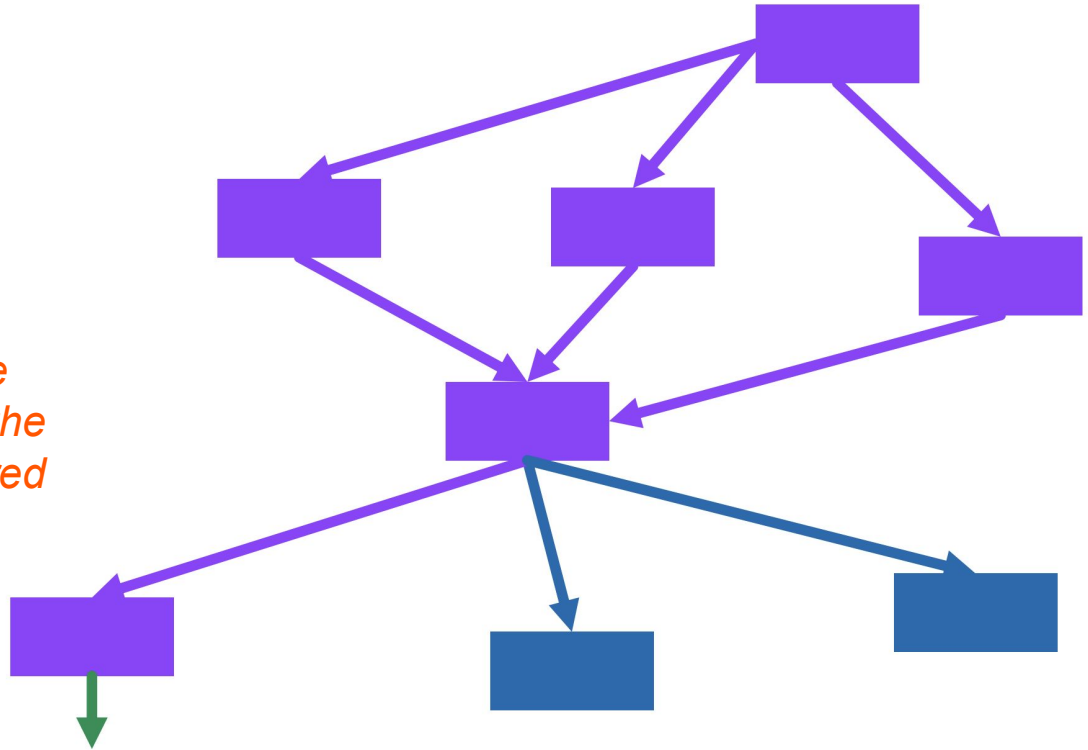
Performance Tuning 2

- Image this DAG



Performance Tuning 2

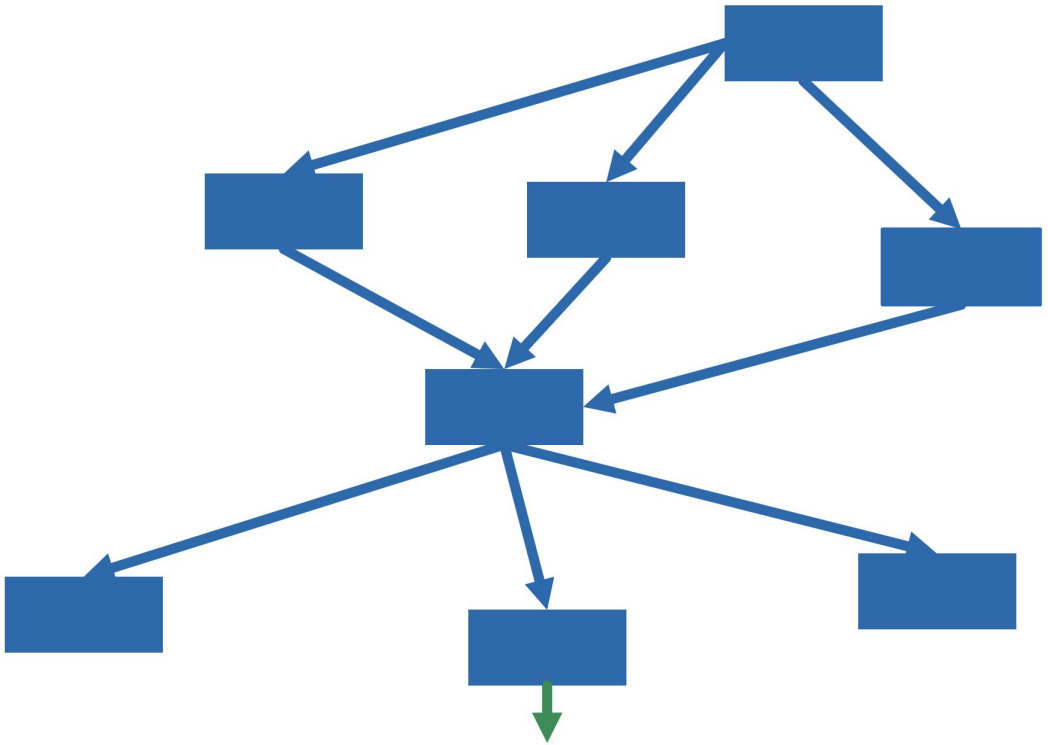
- Image this DAG



cause of lazy evaluation
(steps kept in memory, will not be called until theres an action and the action will trigger the steps required to get your output)

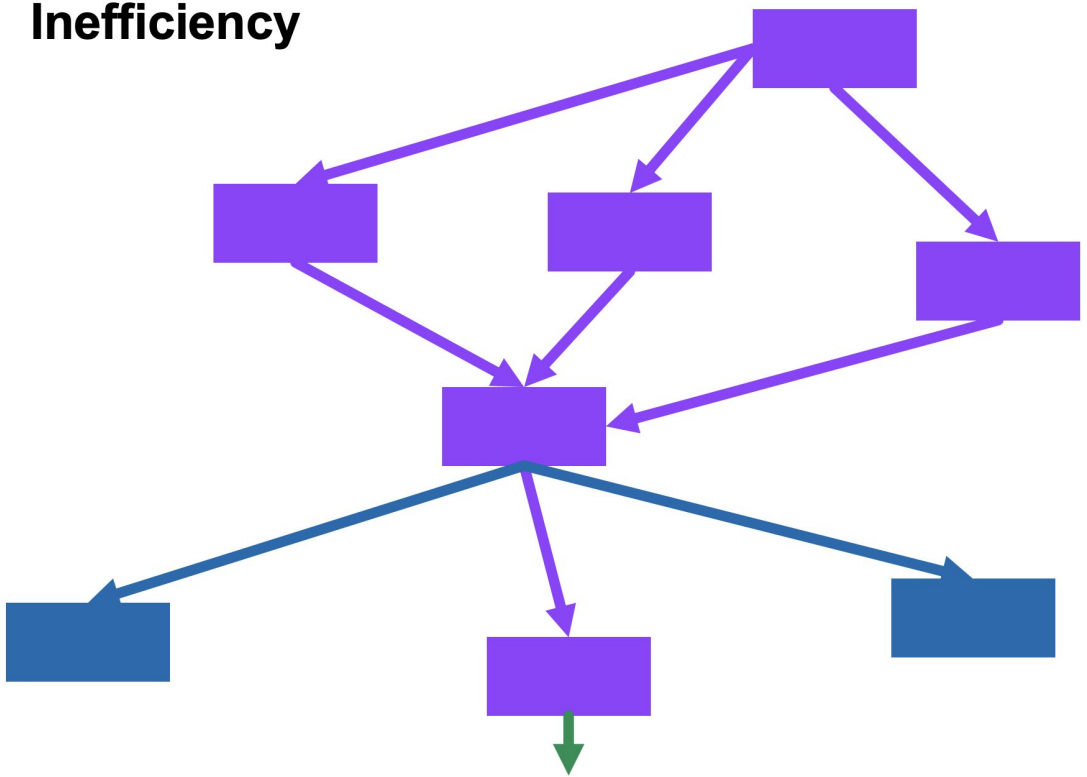
Performance Tuning 2

- Now this



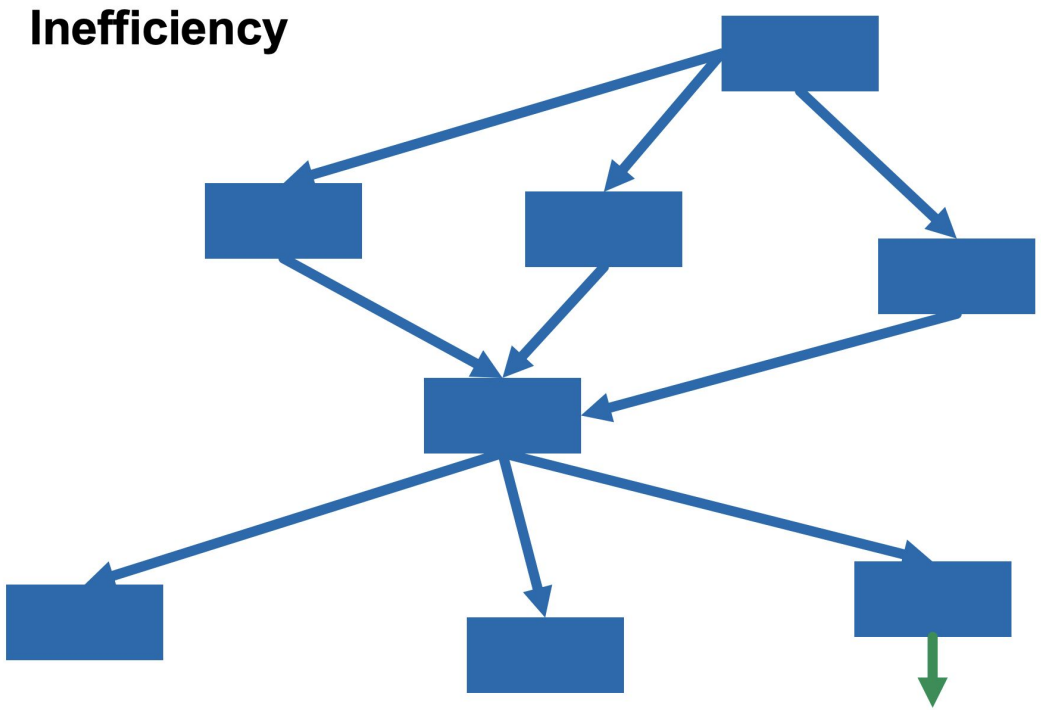
Performance Tuning 2

- Now this



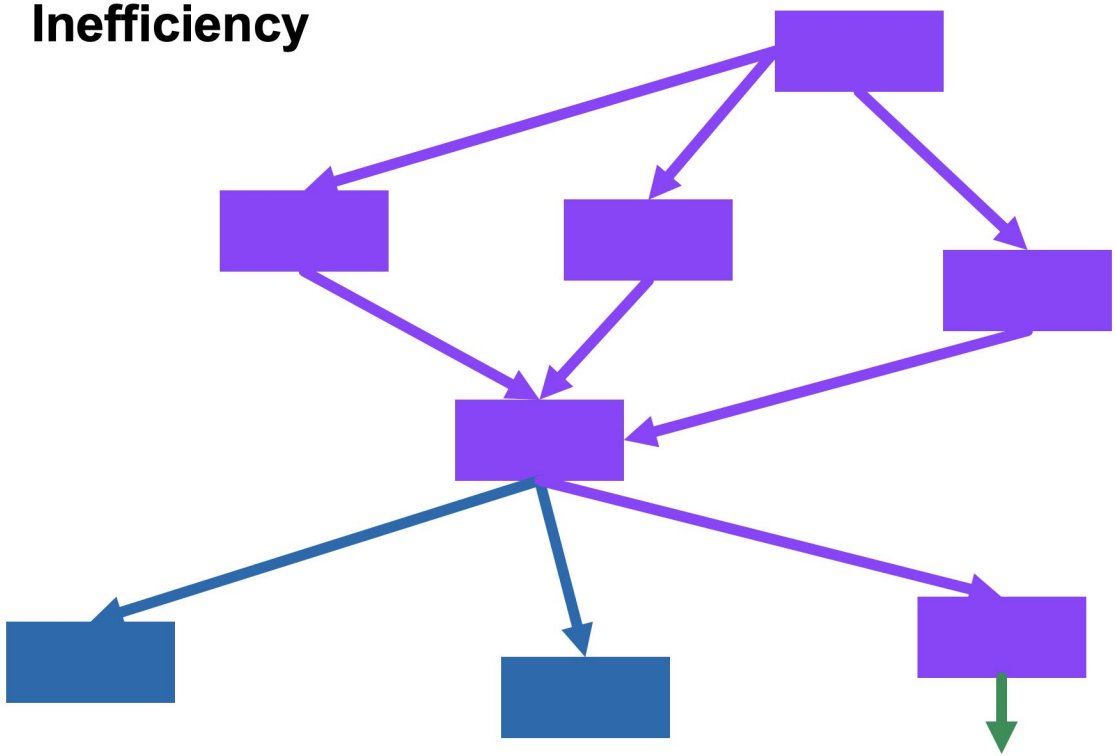
Performance Tuning 2

- And this



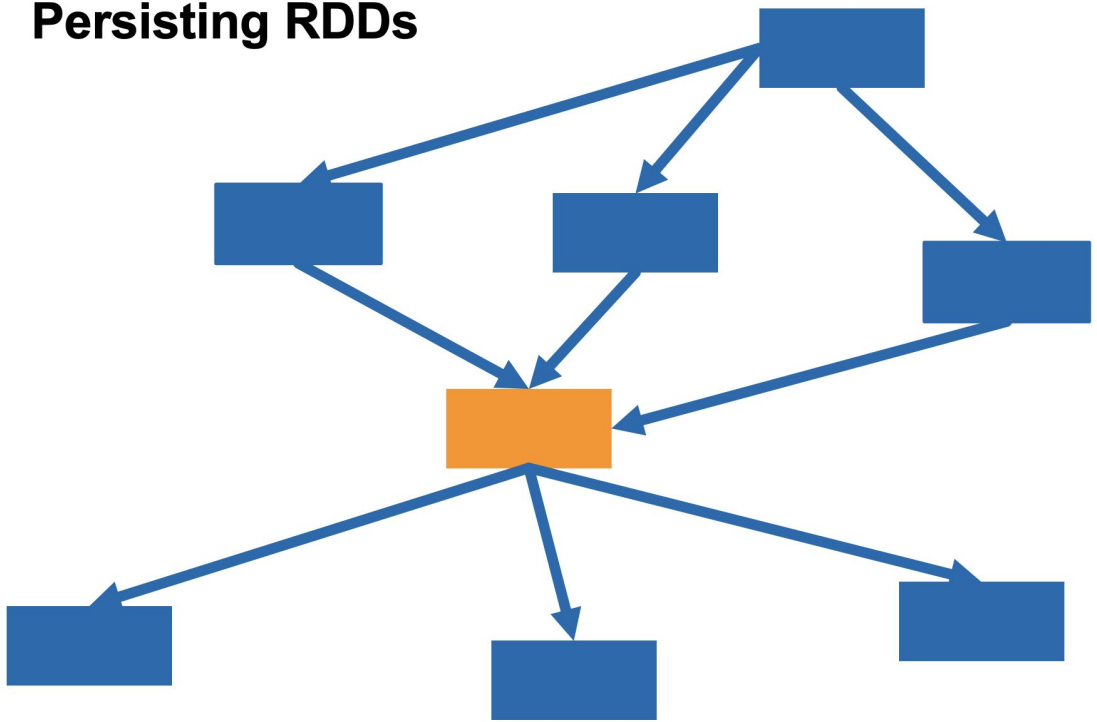
Performance Tuning 2

- And this



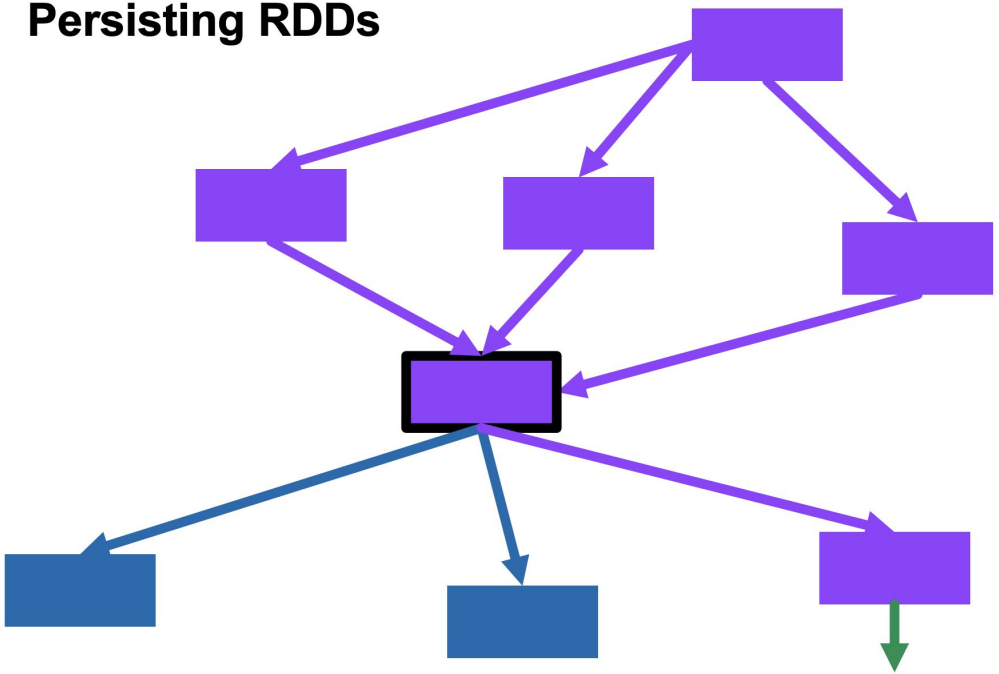
Performance Tuning 2

- Cache RDD
- rdd.persist()
  - Cache using both memory and disk
- rdd.cache()
  - A variant that uses only memory



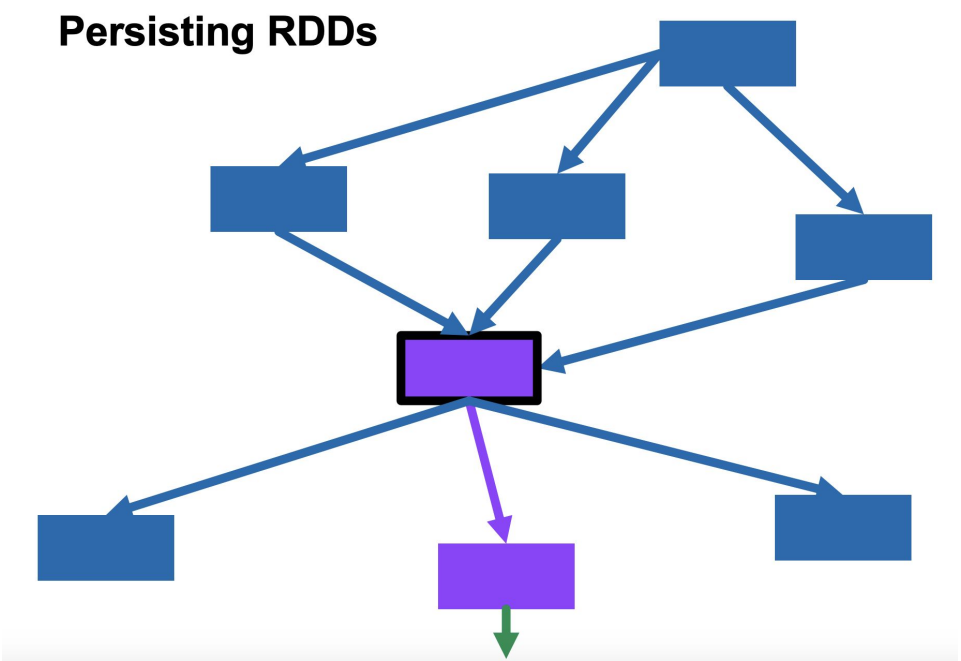
Performance Tuning 2

- Cache RDD



Performance Tuning 2

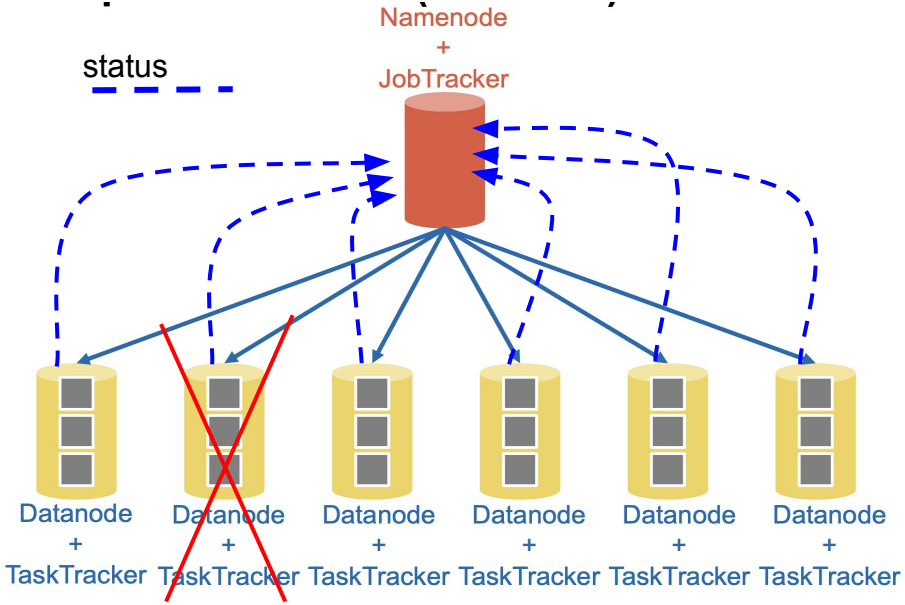
- Cache RDD
- This is why:
  - Spark so much faster than MapReduce



Fault Tolerance

- Nodes fail during execution
- Recall how Hadoop does it:
  - JobTracker monitors
  - Restarts any failed job
- Spark is similar

*The task are **independent** so can just restart the task*
(if you don't have the master, it is very difficult to do)

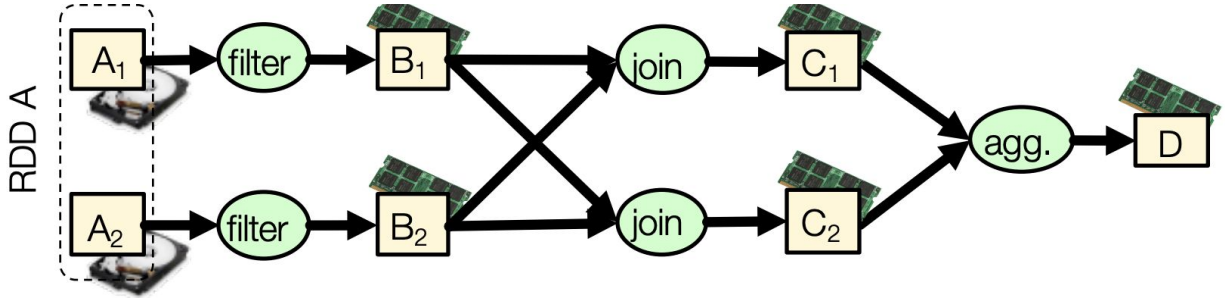


Fault Tolerance

- Spark tracks **lineage**
  - History of every partition
  - **Metadata**, not actual content

Lineage:
- How to get to D from C1/C2
- Keep tracks of relationship between rdds one layer before it

Metadata:
- Description of rdd



Lineage

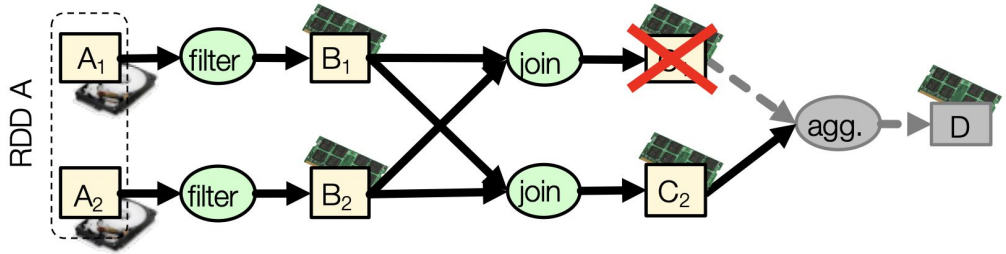
D: C1, C2, agg
C1: B1, B2, join
C2: B1, B2, join
B1: A1, filter
B2: A2, filter

Fault Tolerance

*Can't just recover from **cache** cause cache is in C1 machine but C1 machine is goneee*
*- assume that hdfs is good enough to have enough **replicas** so the input is always there (A1 and A2) if not there is no way of recovering*

- C1 lost before agg finishes
  - Due to node failure
- Lineage has information for rebuilding C1
  - By joining B1 and B2
- What if B1 also lost?

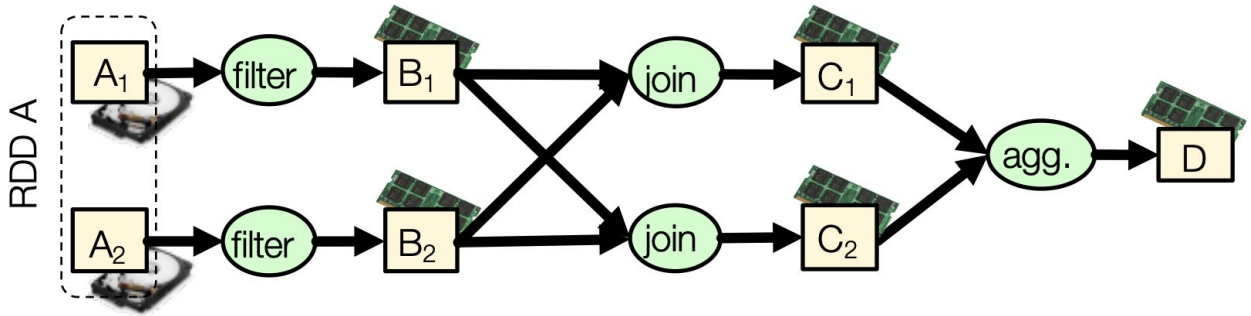
Lineage	
D:	C1, C2, agg
C1:	B1, B2, join
C2:	B1, B2, join
B1:	A1, filter
B2:	A2, filter



In step of recovering C1, Spark discovers B1 is also lost, just trace back.
- Read from A1 filter and recover B1 and then join with B2 to get C1.
- Worst case is you have to trace back all the way to the beginning

Fault Tolerance

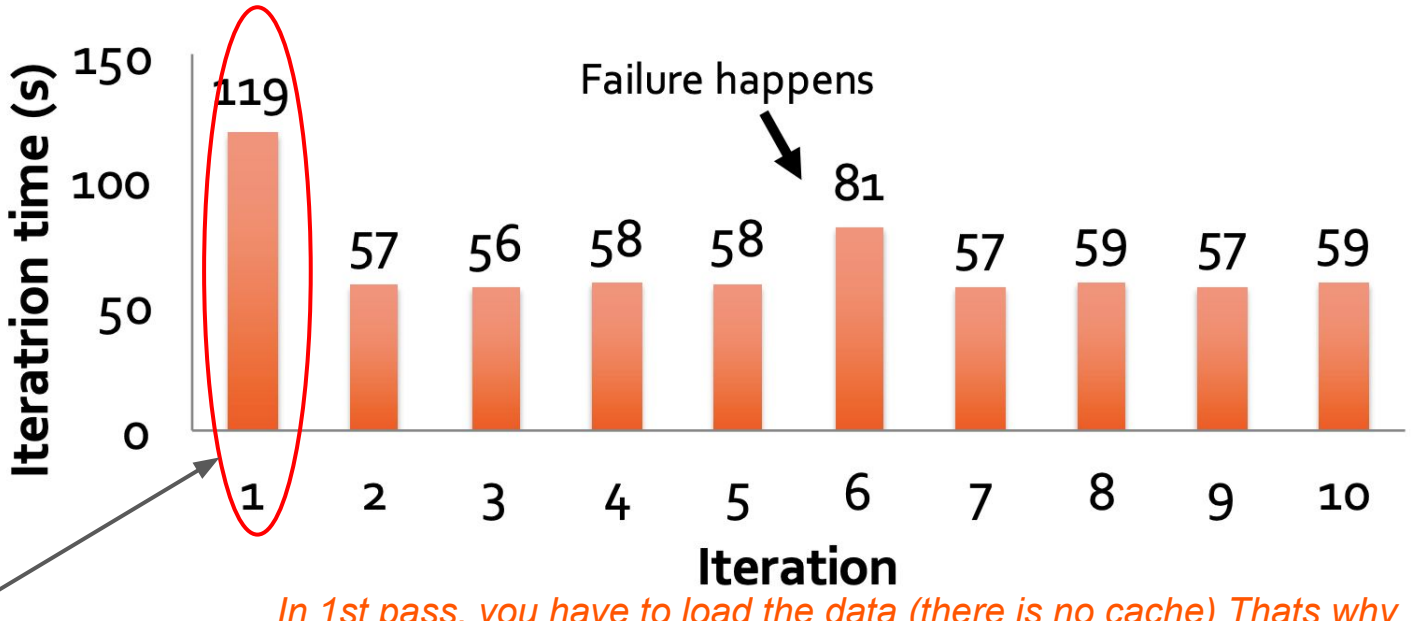
- Rebuild C1 on another node
- And continue



Fault Tolerance

- How long does it take?

(30s more when failure happens)

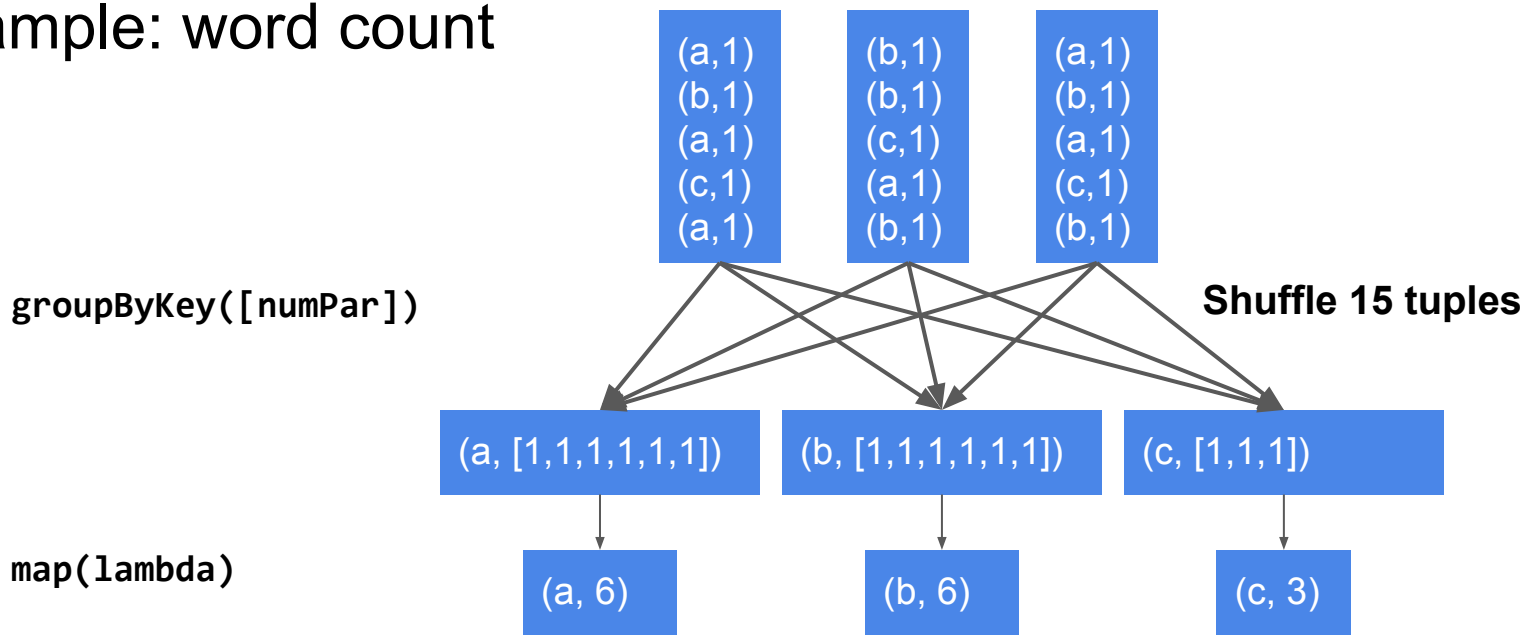


Remember why it is like this?

In 1st pass, you have to load the data (there is no cache) Thats why if your task only have 1 iteration, not faster than MapReduce

groupByKey

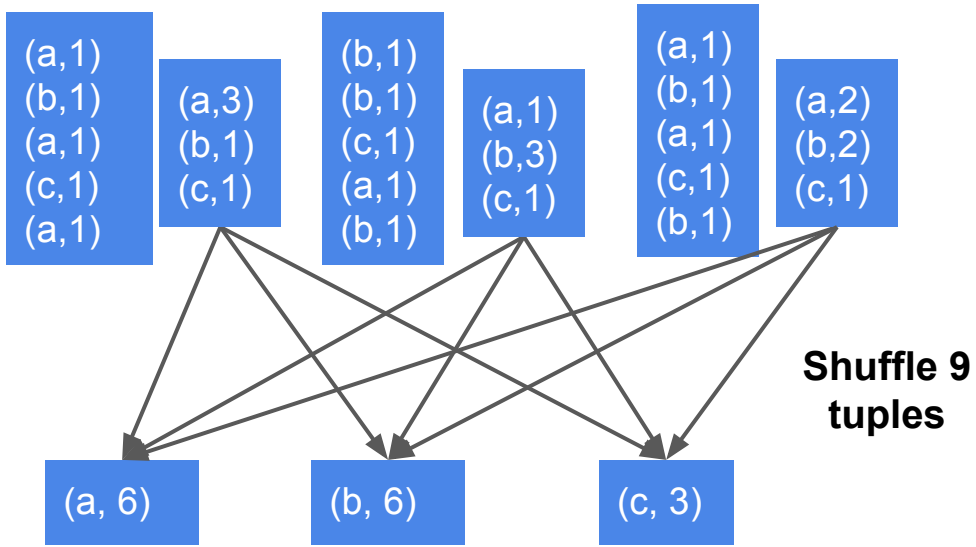
- Similar to Reduce in Hadoop
- Example: word count



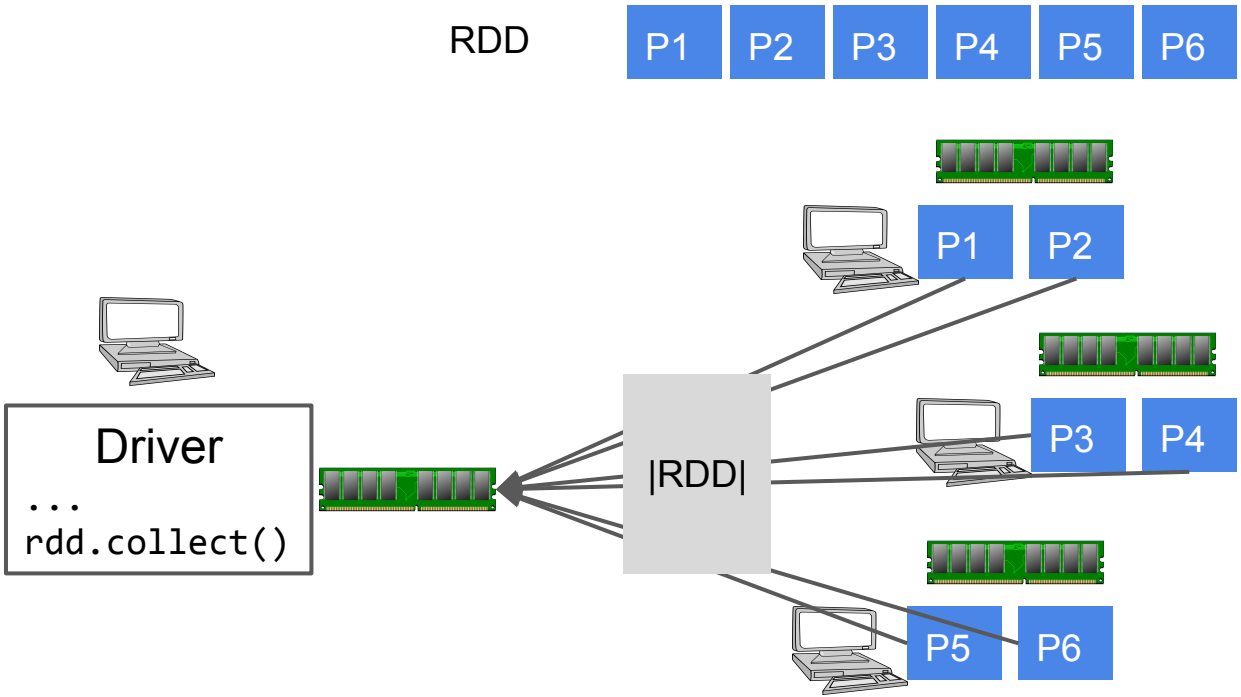
vs. reduceByKey

- Combining before shuffling
  - Fewer messages shuffled

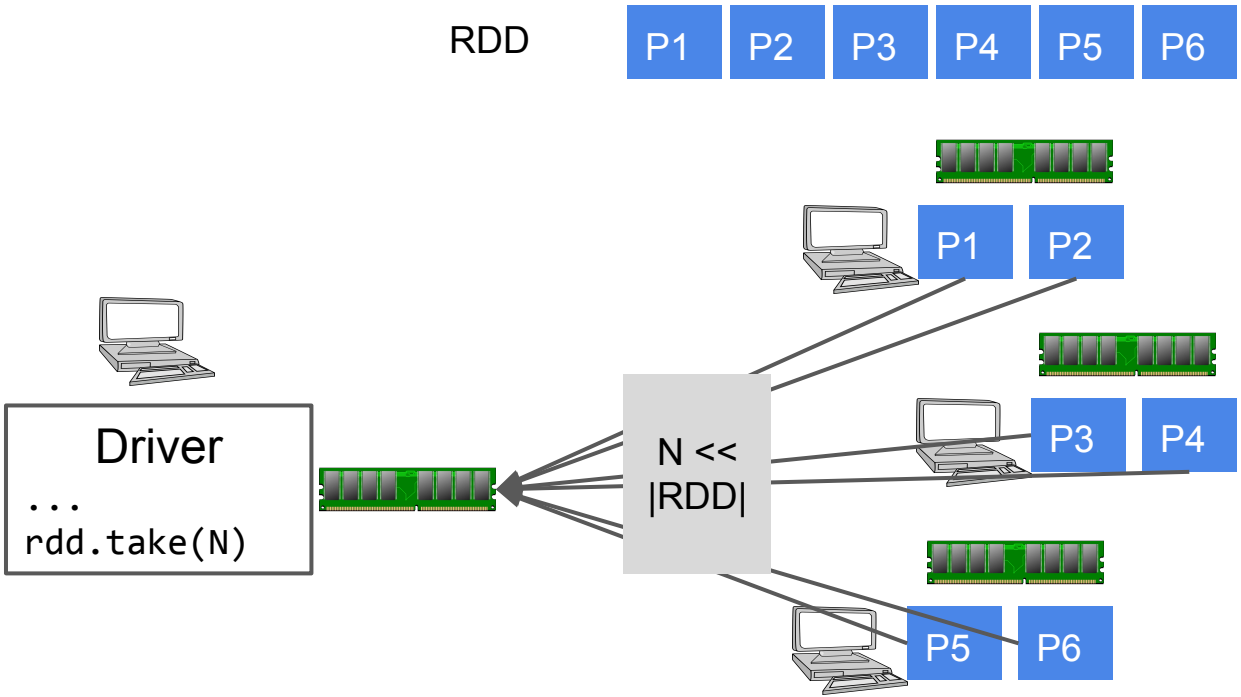
```
reduceByKey(lambda, [numPar])
```



collect()



take(N)



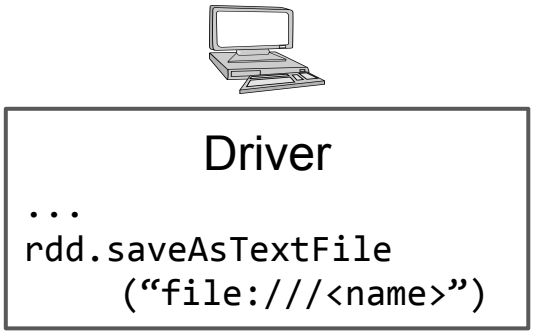
File location

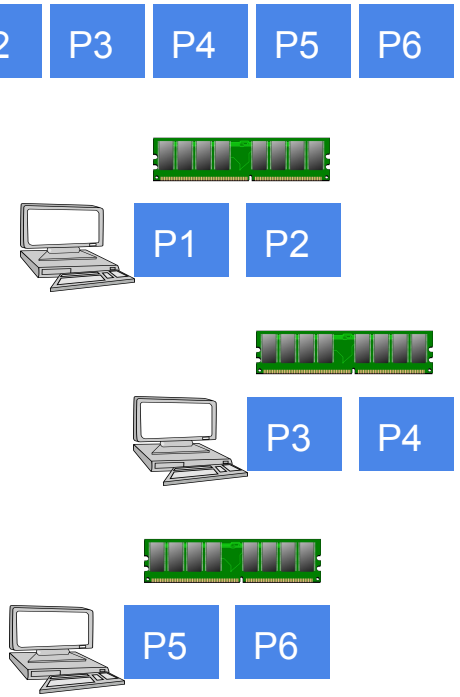
- saveAsTextFile(<path>)
  - file:///<name>
  - hdfs:///<name>
- You know how it works with HDFS
- What happen when using “file:///”

File location

- What would happen?

- The file will be created and distributed at all the machines.
- File is cut off and stored in each machine.
- But no file in the driver machine.





Know Your APIs

API	Description	Note
`groupByKey(.)`	Return (K, Iterator<V>) Like reduce in Hadoop	Expensive shuffle
`reduceByKey(.)`	Merge values for each key using a function	Combine values in a node before shuffle
`collect(.)`	Action, return all values	Download data from all nodes to the driver
`take(.)`	Action, return a number of values	Download only the required number of data to the driver
`count(.)`	Action, count number of values	Sum total number of elements from all nodes

Don’t use `collect(.)` just because you want to see some values

Spark Example

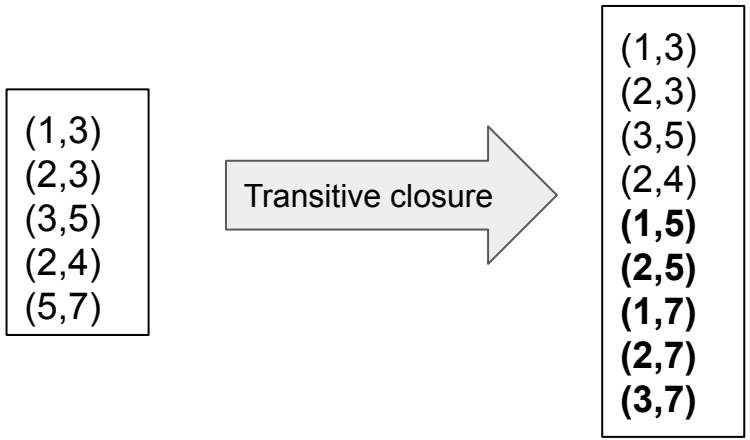
- Wordcount
  - You've seen with reduceByKey(.)
  - Now see the effect of groupByKey(.)

```
words = data.flatMap(lambda x: re.split('\W+', x))
          .map(lambda x: (x,1))
          .reduceByKey(lambda x,y: x+y)
```

```
words = data.flatMap(lambda x: re.split('\W+', x))
          .map(lambda x: (x,1)).groupByKey()
          .map(lambda x: (x[0], sum(x[1])))
```

Spark Example

- Transitive Closure:
  - Graph analytics
  - All pair of nodes reachable from one to another



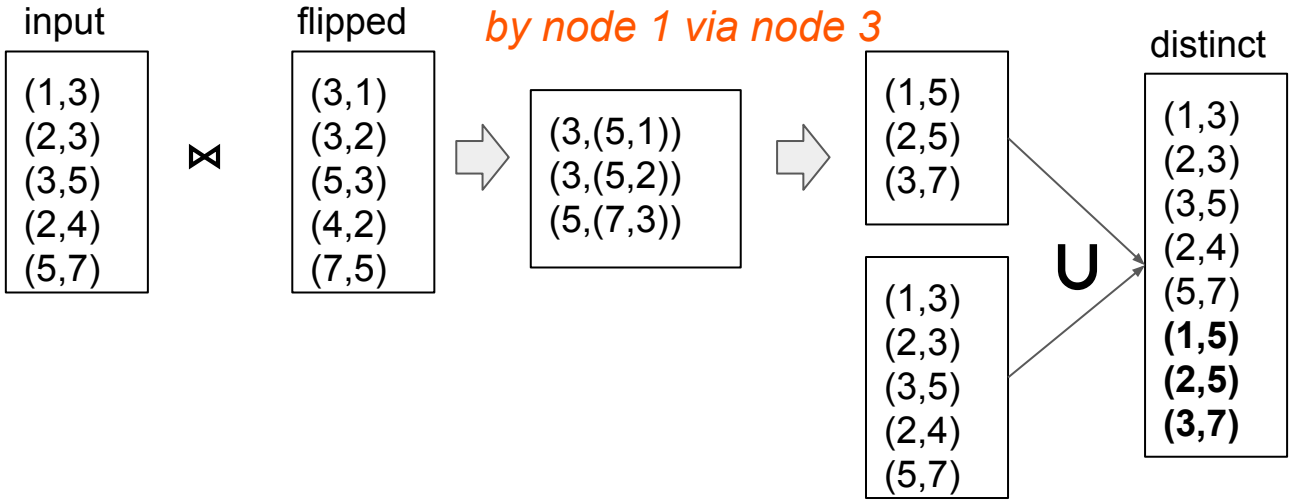
Spark Example

- Transitive Closure:
  - How to compute it?
  - Iteratively: Union and Join!

For 2 hops, take this as input and do the union & join again

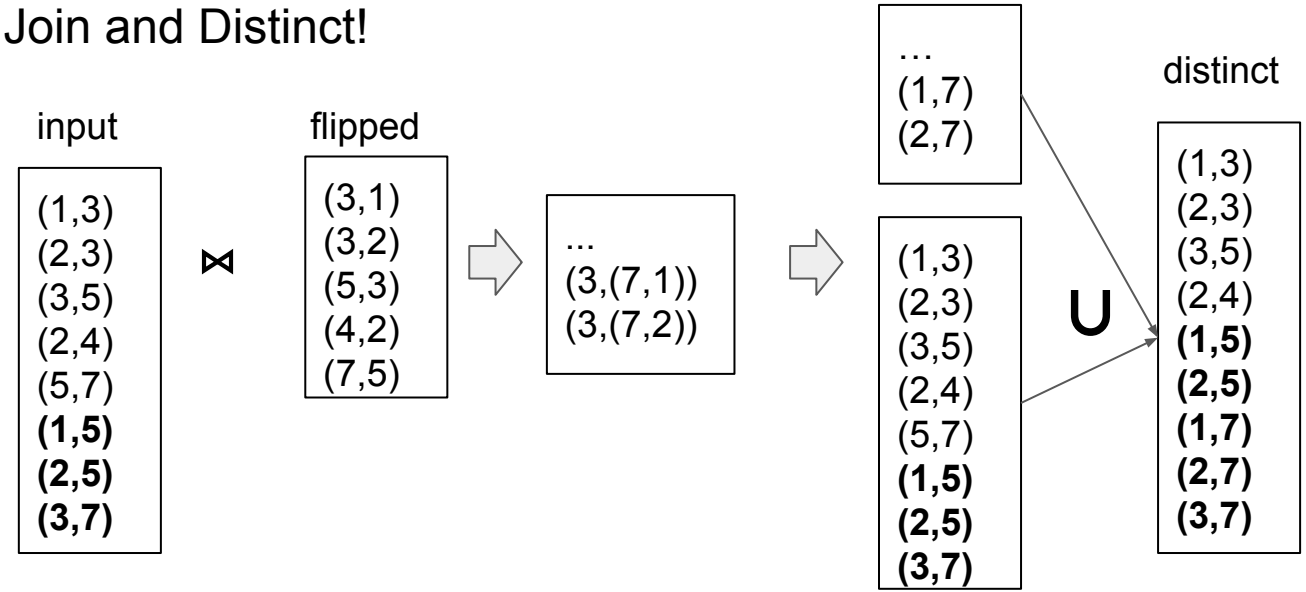
v

edge visitable by 1 hop
node 5 can be reached by node 1 via node 3



Spark Example

- Transitive Closure:
  - How to compute it?
  - Union, Join and Distinct!



Spark Example

flipped

join

union + distinct

```
sc = pyspark.SparkContext("spark://{}:7077".format(sys.argv[1]), "tc")
partitions = int(sys.argv[2]) if len(sys.argv) > 2 else 2
tc = sc.parallelize(generateGraph(), partitions).cache()
# Linear transitive closure: each round grows paths by one edge,
# by joining the graph's edges with the already-discovered paths.
# e.g. join the path (y, z) from the TC with the edge (x, y) from
# the graph to obtain the path (x, z).

# Because join() joins on keys, the edges are stored in reversed order.
edges = tc.map(lambda x_y: (x_y[1], x_y[0]))

oldCount = 0
nextCount = tc.count()
while True:
    oldCount = nextCount
    # Perform the join, obtaining an RDD of (y, (z, x)) pairs,
    # then project the result to obtain the new (x, z) paths.
    new_edges = tc.join(edges).map(lambda __a_b: (__a_b[1][1], __a_b[1][0]))
    tc = tc.union(new_edges).distinct().cache()
    nextCount = tc.count()
    if nextCount == oldCount:
        break
```

SparkSQL

- How to do SQL on Spark?
  - See how we go full circle again
    - SQL → NoSQL → SQL

Spark SQL: Relational Data Processing in Spark

Michael Armbrust[†], Reynold S. Xin[†], Cheng Lian[†], Yin Huai[†], Davies Liu[†], Joseph K. Bradley[†], Xiangrui Meng[†], Tomer Kaftan[‡], Michael J. Franklin[†‡], Ali Ghodsi[†], Matei Zaharia[†*]

[†]Databricks Inc. [*]MIT CSAIL [‡]AMPLab, UC Berkeley

SparkSQL

- What does this piece of code do?

rdd2: k:course, v: (age, 1)
rdd3: k:course, v:(total age, total count)
res: k:course, v: total_age/total_count

input.csv

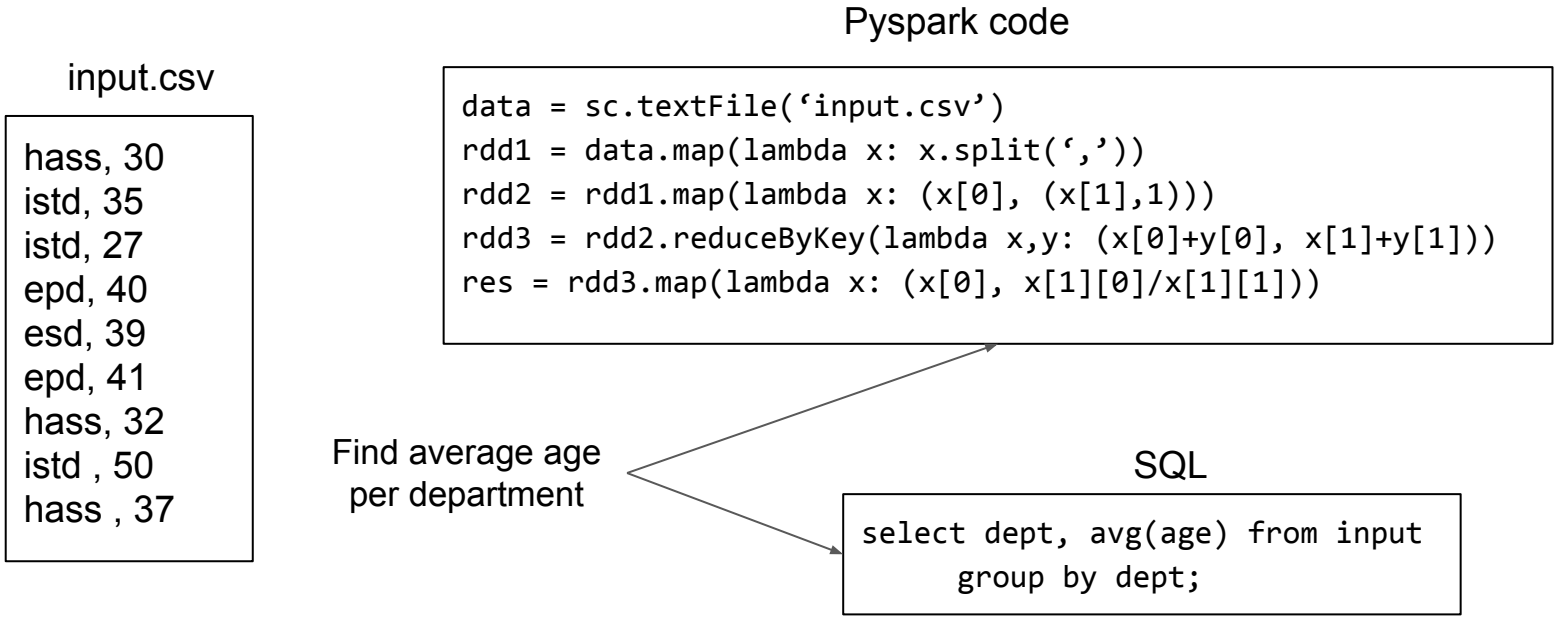
```
hass, 30
istd, 35
istd, 27
epd, 40
esd, 39
rpd, 41
hass, 32
istd , 50
hass , 37
```

Pyspark code

```
data = sc.textFile('input.csv')
rdd1 = data.map(lambda x: x.split(','))
rdd2 = rdd1.map(lambda x: (x[0], [x[1],1]))
rdd3 = rdd2.reduceByKey(lambda x,y: (x[0]+y[0], [x[1]+y[1]))
res = rdd3.map(lambda x: [x[0], x[1][0]/x[1][1]])
```

SparkSQL

- What does this piece of code do?



SparkSQL

- Pyspark code vs. SQL
  - Which one is easier to understand/code/use?
- Some “limitations” of RDD APIs:
  - Too low level
  - Most data is (semi-)structured, e.g. csv or JSON
    - Processing them inevitably ends up with a lot of tuples
  - SQL is easier to understand than functional transformation (map, reduce, etc.)
    - Maybe because people know SQL way before RDDs

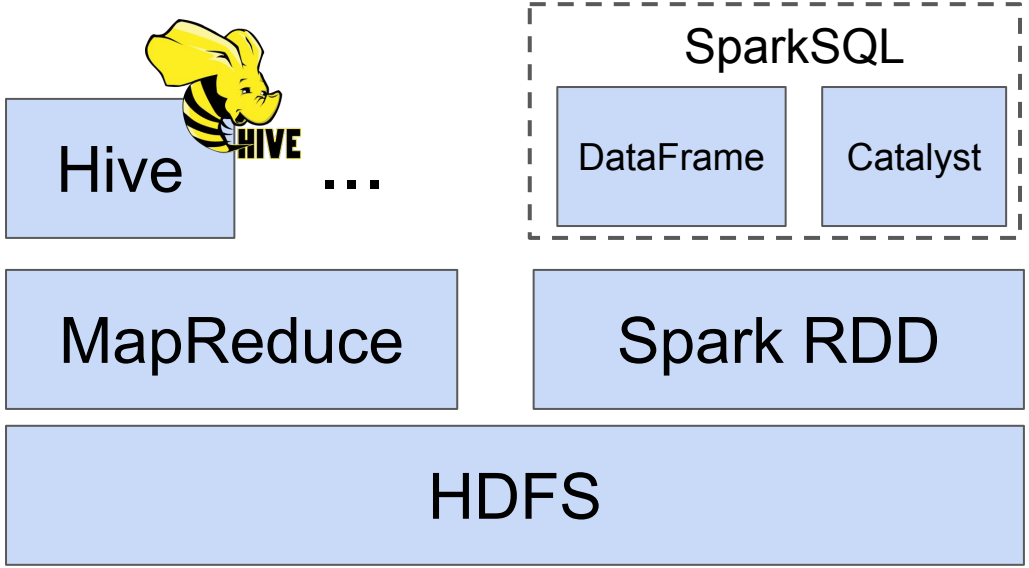
SparkSQL

- Where in the stack?
  - Hadoop SQL support: Hive
  - Spark Core (RDDs)?

Hive - A Petabyte Scale Data Warehouse using Hadoop

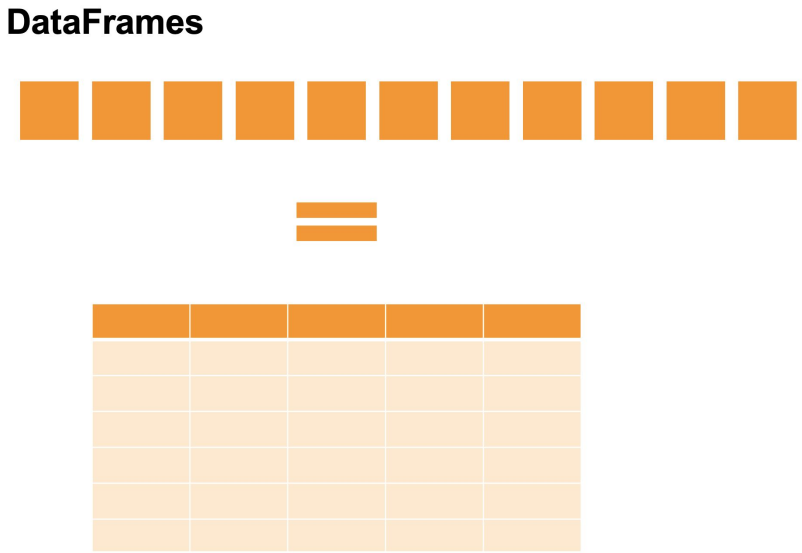
10 June 2009 at 23:49

A number of engineers from Facebook are speaking at the Yahoo! Hadoop Summit today about the ways we are using Hadoop and Hive for analytics. Hive is an open source, peta-byte scale date warehousing framework based on Hadoop that was developed by the Data Infrastructure Team at Facebook. In this blogpost we'll talk more about Hive, how it



DataFrame

- DataFrame = RDD + Schema
- Columnar storage:
  - Recall row vs. column store



Row Format

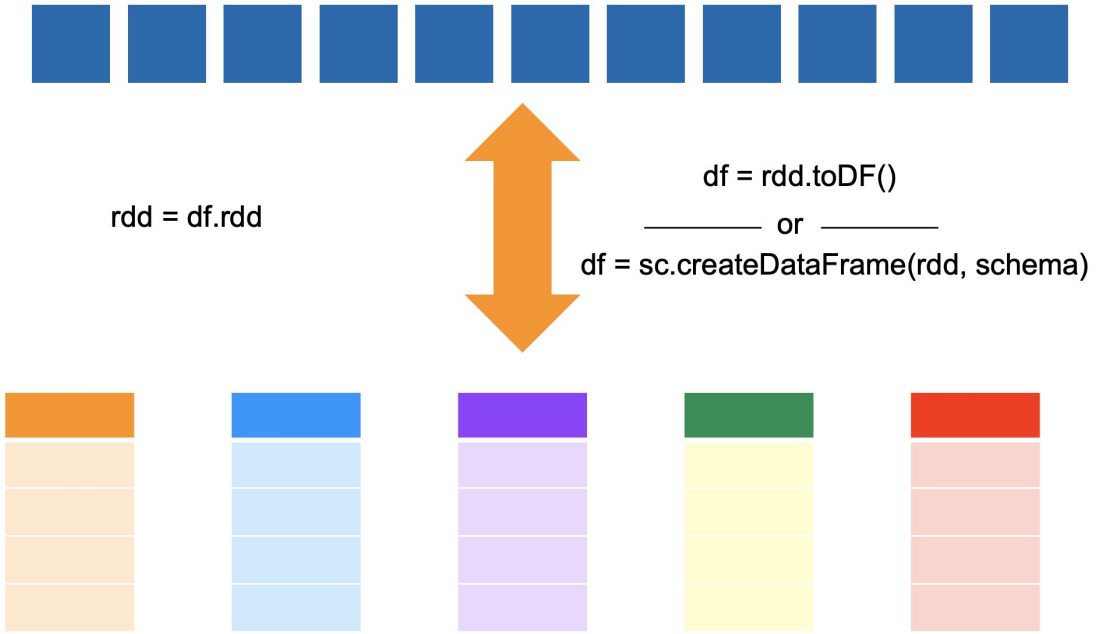
1	john	4.1
2	mike	3.5
3	sally	6.4

Column Format

1	2	3
john	mike	sally
4.1	3.5	6.4

DataFrame

- Back and forth conversion



DataFrame vs. RDD

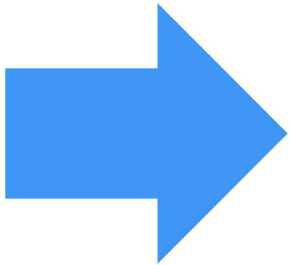
Dataframe	RDD
Lazy execution	Lazy execution
Spark is aware of the data model	The data model is hidden from Spark
Spark is aware of the query logic	The transformations and actions are black boxes
Can optimize the query	Cannot optimize the query

DataFrame

- Schema inference
  - Infer structure from raw data
  - Example: csv

```
foo,bar
1,true
2,true
3,false
4,true
5,true
6,false
7,true
```

dataset.csv



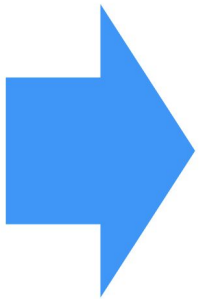
foo	bar
integer	boolean
1	true
2	true
3	false
4	true
5	true
6	false
7	true

DataFrame

DataFrame

- Schema inference
  - Infer structure from raw data
  - Example: json

```
{ "foo" : 1, "bar" : true}
{ "foo" : 2, "bar" : true}
{ "foo" : 3, "bar" : false}
{ "foo" : 4, "bar" : true}
{ "foo" : 5, "bar" : true}
{ "foo" : 6, "bar" : false}
{ "foo" : 7, "bar" : true}
```



foo	bar
integer	boolean
1	true
2	true
3	false
4	true
5	true
6	false
7	true

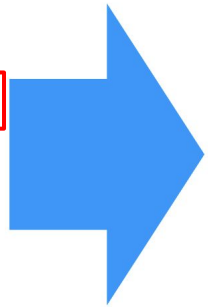
dataset.json

DataFrame

DataFrame

- Schema inference
  - Only best effort!

```
{ "foo" : 1, "bar" : true}
{ "foo" : 2, "bar" : true}
{ "foo" : [3, 4], "bar" : false}
{ "foo" : 4, "bar" : true}
{ "foo" : 5, "bar" : true}
{ "foo" : 6, "bar" : false}
{ "foo" : 7, "bar" : true}
```



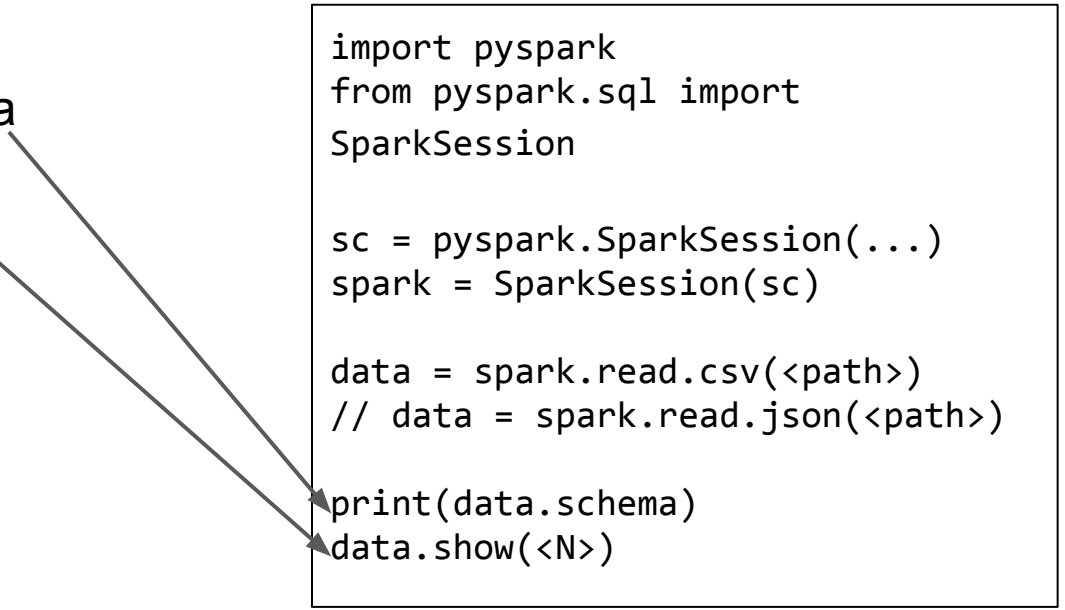
foo	bar
string	boolean
"1"	true
"2"	true
"[3, 4]"	false
"4"	true
"5"	true
"6"	false
"7"	true

dataset.json

DataFrame

DataFrame

- Data loading example
  - spark.read.csv(..)
  - Infer CSV schema
  - See some values



```
import pyspark
from pyspark.sql import
SparkSession

sc = pyspark.SparkSession(...)
spark = SparkSession(sc)

data = spark.read.csv(<path>)
// data = spark.read.json(<path>)

print(data.schema)
data.show(<N>)
```

DataFrame APIs

- Column
  - `df.<colName>`
  - Can be renamed: `df.<colName>.alias(<newName>)`
- Projection
  - `df.select(<cols>)`
  - `<cols>` can be a column string, or a Column object
  - Return another DataFrame
- Filter
  - `df.filter(<condition>)`
  - `<condition>` can be a string, like SQL statement: `df.filter(“age > 3”)`

DataFrame APIs

- Join:
  - `join(other, <on>, <how>)`
  - `<on>` can be a list of column names: `df.join(df1, ['col1', 'col2'])`
    - Joined column must exist at both sides
  - `<how>`: inner, left_outer join, etc.
- Much more at:

https://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html

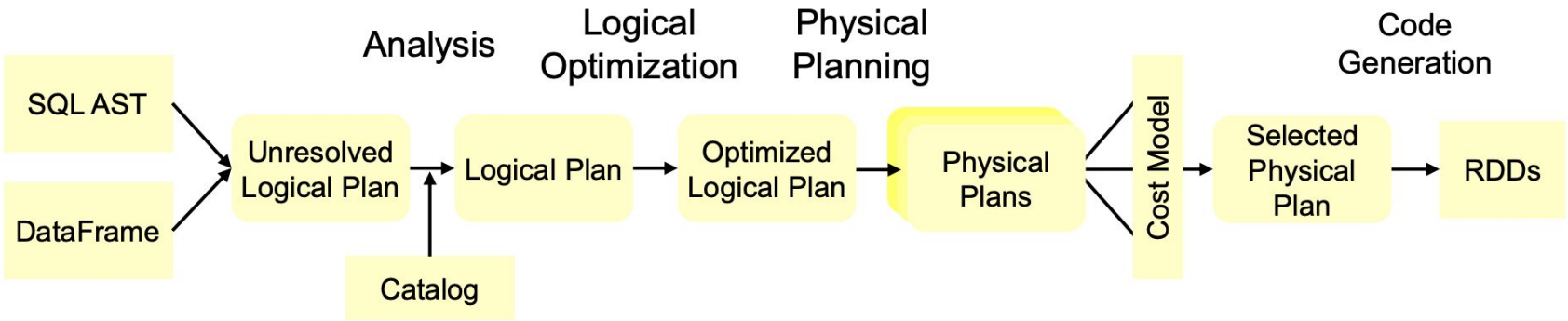
DataFrame APIs

- And other related functions
  - Max, Min
  - Avg
  - Length
  - Etc.
- Applied on columns

```
from pyspark.sql.functions import *
```

SQL

- SparkSQL = DataFrame + Database Engine
  - Catalyst is its Query Optimizer



Streaming

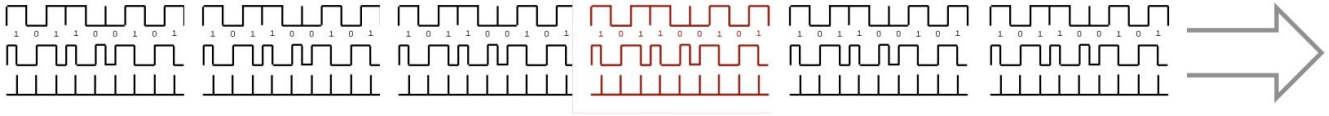
- Data comes in continuously
- Near real-time latency requirements
  - Example: trend analysis, online ML, etc.
  - **NOT high-frequency trading**

Spark Streaming

Streaming

Fraud detection in bank transactions

Anomalies in sensor data



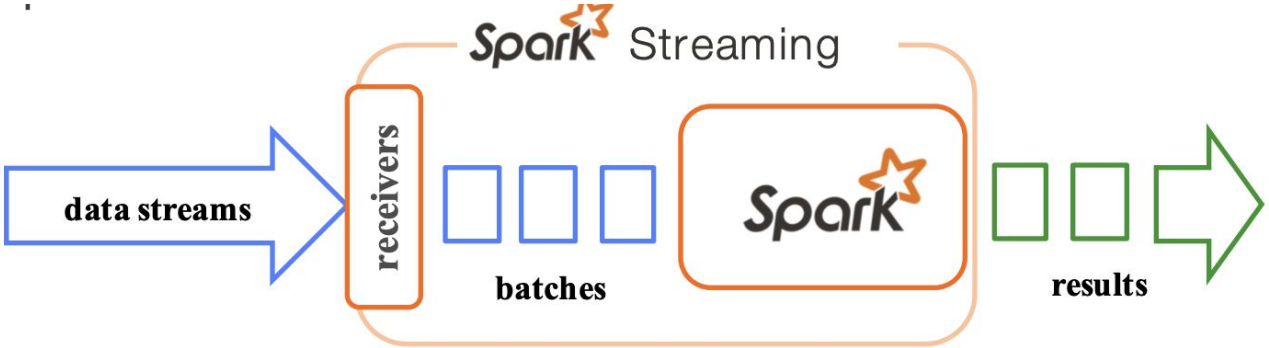
Cat videos in tweets

Streaming

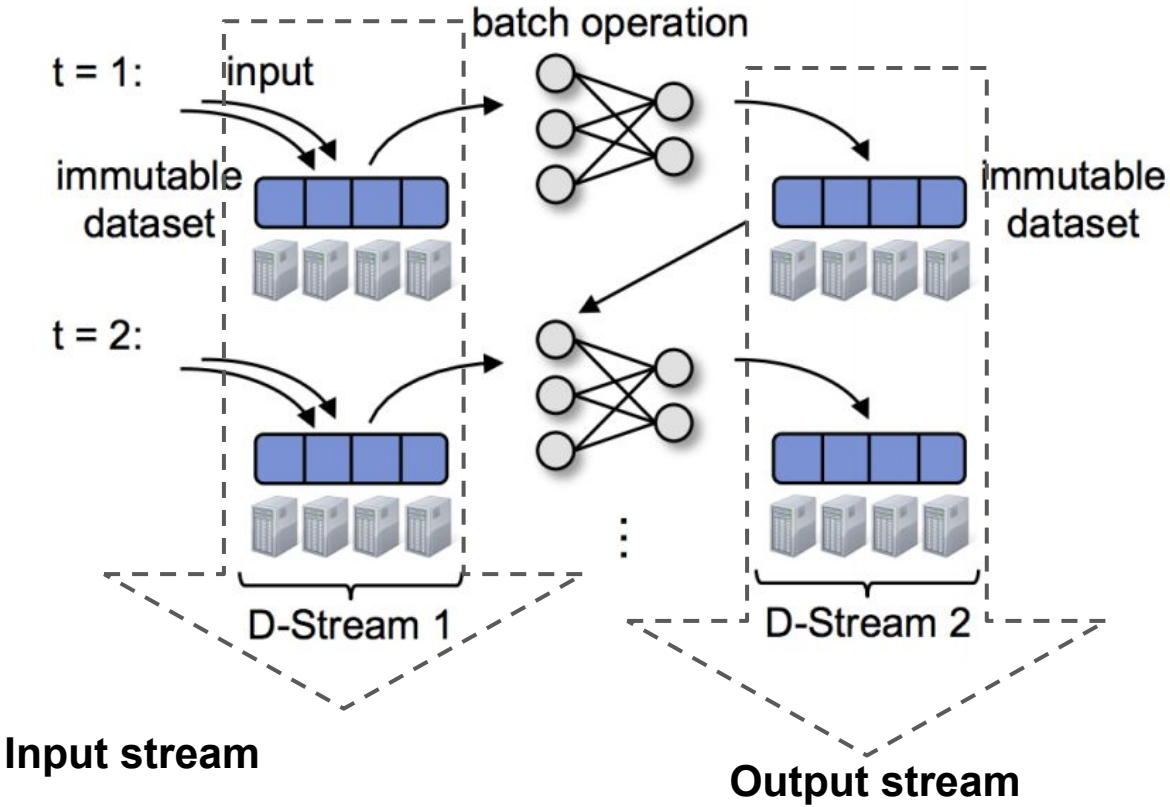
- Requirements
  - Scale
  - Low latency
  - Fault tolerance
  - Integration with other systems

Streaming

- Spark Streaming
  - Data stream chopped up to small batches
  - Feed batches to Spark
- Called Discretized Stream (DStream))



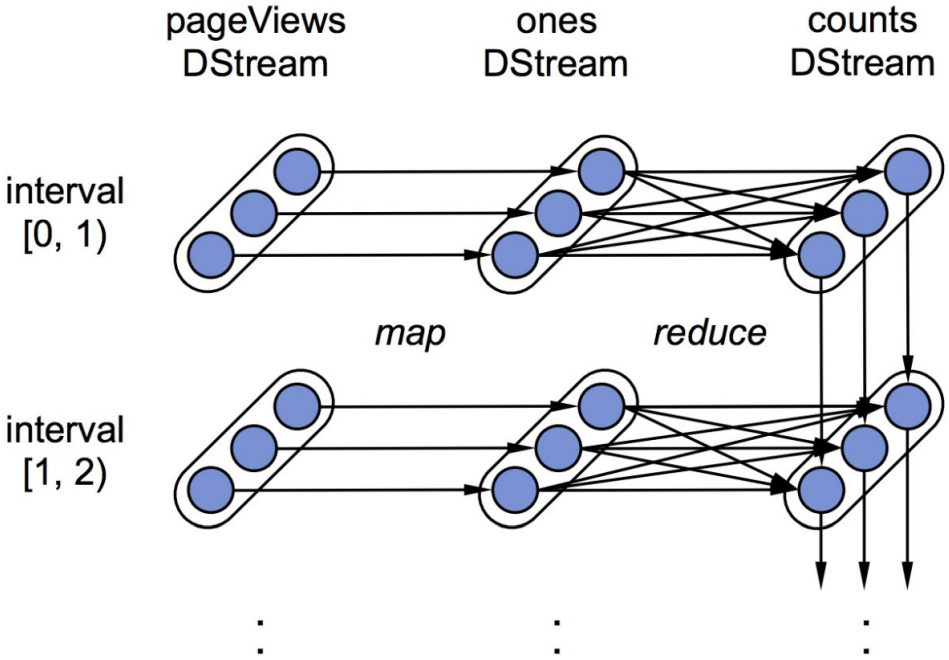
Spark Streaming



Spark Streaming

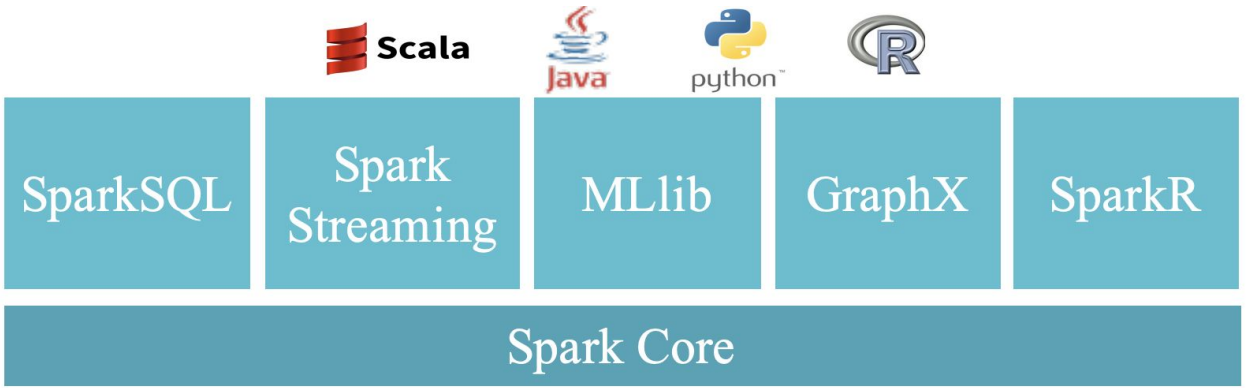
- Example:
  - Continuous view count

```
pageViews = readStream("http://…", 1s)
ones = pageViews.map(
               lambda x: (x.url, 1))
counts = ones.runningReduce(
               lambda x,y: x+y)
```



Spark Summary

- A complete stack
- Unifying
  - Batch
  - Interactive
  - Iterative
  - ML
  - SQL
  - Streaming



Question 1

Given this Spark job. It uses 100 HDFS chunks/partitions spread over 100 servers.

Suppose all servers have finished their work for line 3. At this point one machine crashes. The system has to recover part of `kvwords`.

How much work does the system have to do to recover?

```
1. lines = spark.textFile(...)
2. words = lines.flatMap(lambda x: x.split(' '))
3. kvwords = words.map(lambda x: (x, 1))
4. counts = kvwords.reduceByKey(lambda x, y: x+y)
5. counts.lookup("consistency")
```

kvwords (narrow dependency from flatMap and map)
- To recover, boot up new machine download that partition and execute line 1,2,3 on that partition.
- recover only 1 machine and process only 1 partition out of the original 100 partitions. Spark keeps track of which machine is running which partitions.

Question 2

Following up from Question 1.

After recovery, suppose all 100 machines finished their works for line 4. Then, one server holding the count of “consistency” crashes. The system has to recover that part of `counts`.

How much work does the system have to do to recover?

```
1. lines = spark.textFile(input_file)
2. words = lines.flatMap(lambda x: x.split(’ ’))
3. kvwords = words.map(labmda x: (x, 1))
4. counts = kvwords.reduceByKey(lambda x, y: x+y)
5. counts.lookup("consistency")
```

reduceByKey *=>* ***wide*** *dependency with kvwords.*
- Boot up new machine to recover this failure, need to recompute everything from line 1,2,3,4 on all machines.

Recovery is ***very expensive*** *as compared to q1 where there is narrow dependency.*
To reduce cost, can try to ***cache*** *kvwords (dont have to redo 1,2,3 anymore if it crashes later. but still need to do the shuffling at line 4.)*