SHIJIE ZHANG

SPARK CORE

OUTLINE

- Intro
- Standalone-view RDD concepts
  - Basic RDD
  - Key-Pair RDD
- Distributed-view RDD concepts
  - Lazy evaluation
  - Cache
  - Shuffle/Partition
  - Broadcast
  - Accumulator
- Conclusion

Latency numbers "every programmer should know:"[1]

```
L1 cache reference ..........................  0.5 ns
Branch mispredict ............................    5 ns
L2 cache reference ...........................    7 ns
Mutex lock/unlock ...........................   25 ns
Main memory reference ......................  100 ns
Compress 1K bytes with Zippy ............. 3,000 ns  =    3 µs
Send 2K bytes over 1 Gbps network ....... 20,000 ns  =   20 µs
SSD random read ........................ 150,000 ns  =  150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  =  250 µs
Round trip within same datacenter ......  500,000 ns  =  0.5 ms
Read 1 MB sequentially from SSD* .....  1,000,000 ns  =    1 ms
Disk seek ........................... 10,000,000 ns  =   10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns  =   20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns  =  150 ms
```

(Assuming ~1GB/sec SSD.)

- To get a better intuition about differences of these numbers, humanize these durations

Method: multiply all these durations by a billion.

- Then, we can map each latency number to a human activity

Humanized durations grouped by magnitude:

Minute:

L1 cache reference	0.5 s	One heart beat (0.5 s)
Branch mispredict	5 s	Yawn
L2 cache reference	7 s	Long yawn
Mutex lock/unlock	25 s	Making a coffee

Hour:

Main memory reference	100 s	Brushing your teeth
Compress 1K bytes with Zippy	50 min	One episode of a TV show

Day:

Send 2K bytes over 1 Gbps network	5.5 hr	From lunch to end of work day

Week:

SSD random read	1.7 days	A normal weekend
Read 1 MB sequentially from memory	2.9 days	A long weekend
Round trip within same datacenter	5.8 days	A medium vacation
Read 1 MB sequentially from SSD	11.6 days	Waiting for almost 2

Year:

Disk seek	16.5 weeks	A semester in university
Read 1 MB sequentially from disk	7.8 months	Almost producing a new human being
The above 2 together	1 year	

Decade:

Send packet CA->Netherlands->CA	4.8 years	Average time it takes to complete a bachelor's degree

Humanized numbers

Shared Memory

Seconds

L1 cache reference.....0.5s

L2 cache reference.....7s

Mutex lock/unlock.....25s

Minutes

Main memory reference.....1m 40s

Distributed

Days

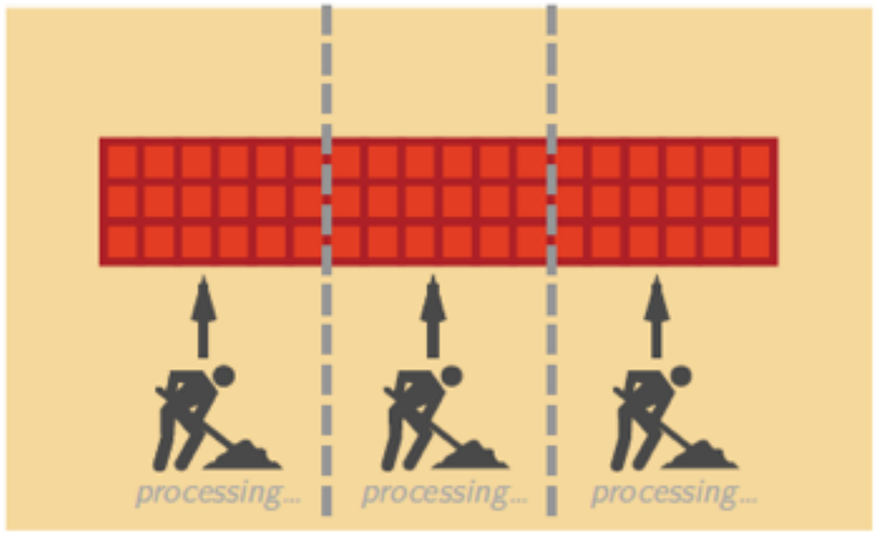
Roundtrip within same datacenter.....5.8 days

Years

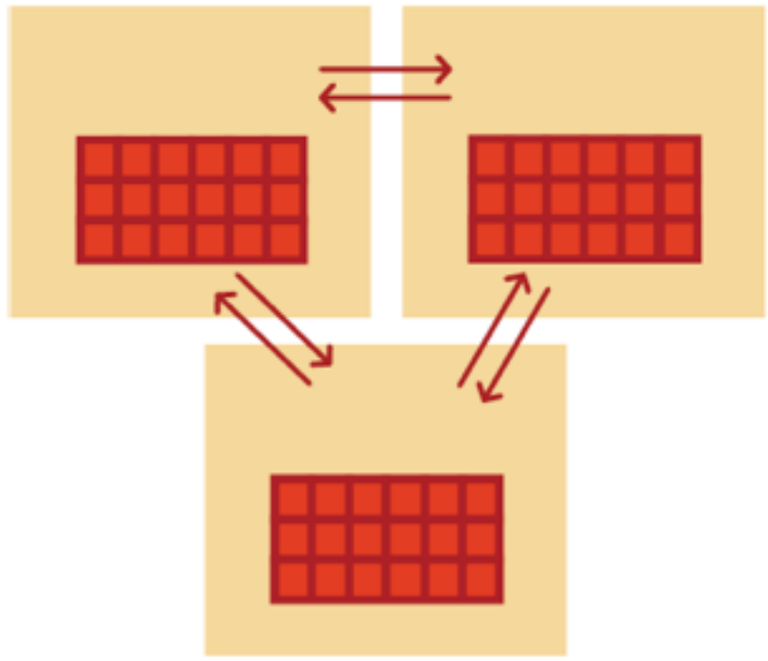
Send packet CA->Netherlands->CA....4.8 years

What does **distributed** data-parallel look like?

Shared memory:



Distributed:



Shared memory case: Data-parallel programming model. Data partitioned in memory and operated upon in parallel.

Distributed case: Data-parallel programming model. Data partitioned between machines, network in between, operated upon in parallel.

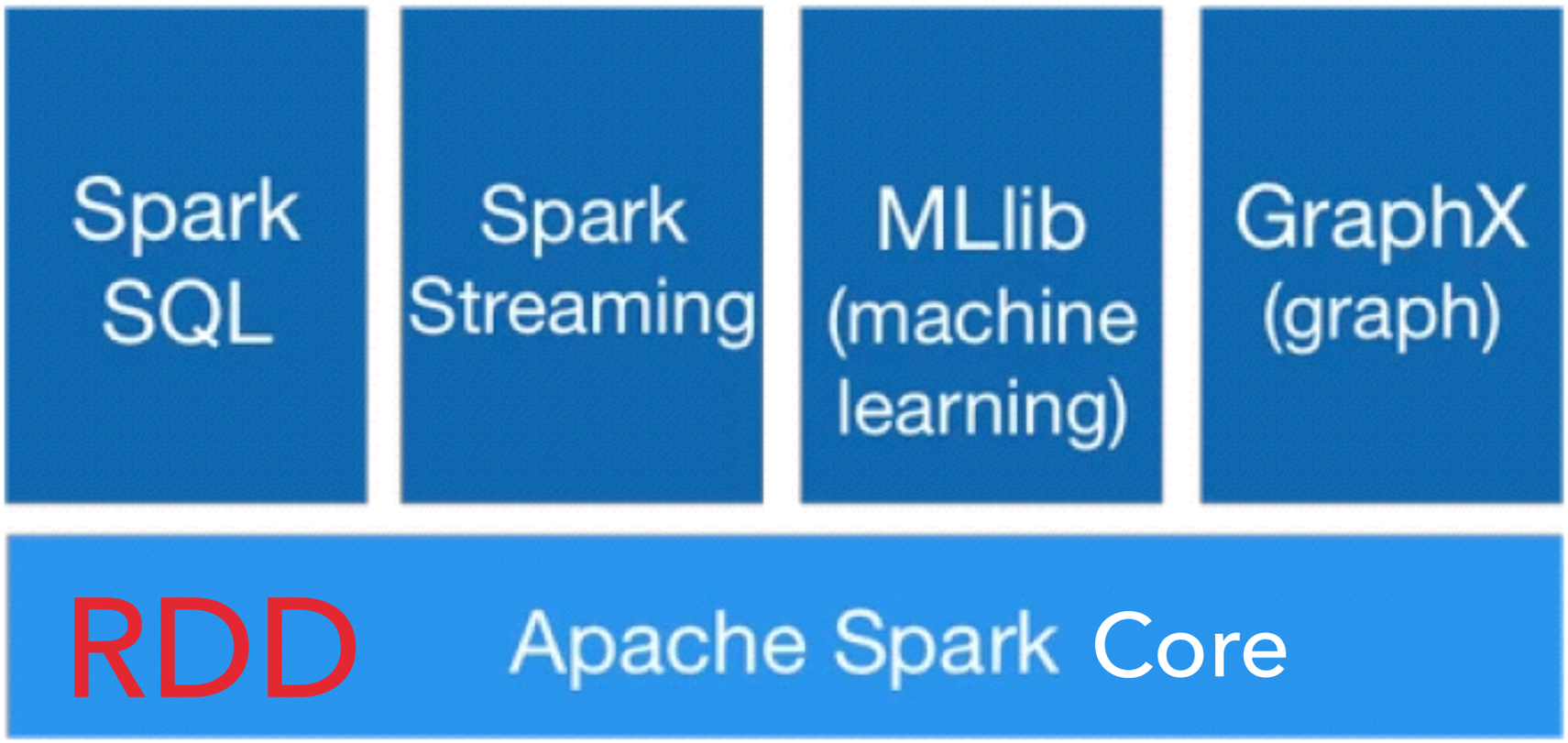
Distributed data parallel programming, two more worries

- Partial failure: a subset of machine crash
- Latency: certain operations have higher latency

Spark

- A distributed data parallel programming framework
- Implements a data parallel model called RDD

Spark components



WHAT'S RDD

- Just looks like Scala collections (but distributed across machines)

Combinators on Scala parallel/sequential collections:

```
map
flatMap
filter
reduce
fold
aggregate
```

Combinators on RDDs:

```
map
flatMap
filter
reduce
fold
aggregate
```

WHAT'S RDD

- While signatures differ a bit, their semantics are the same

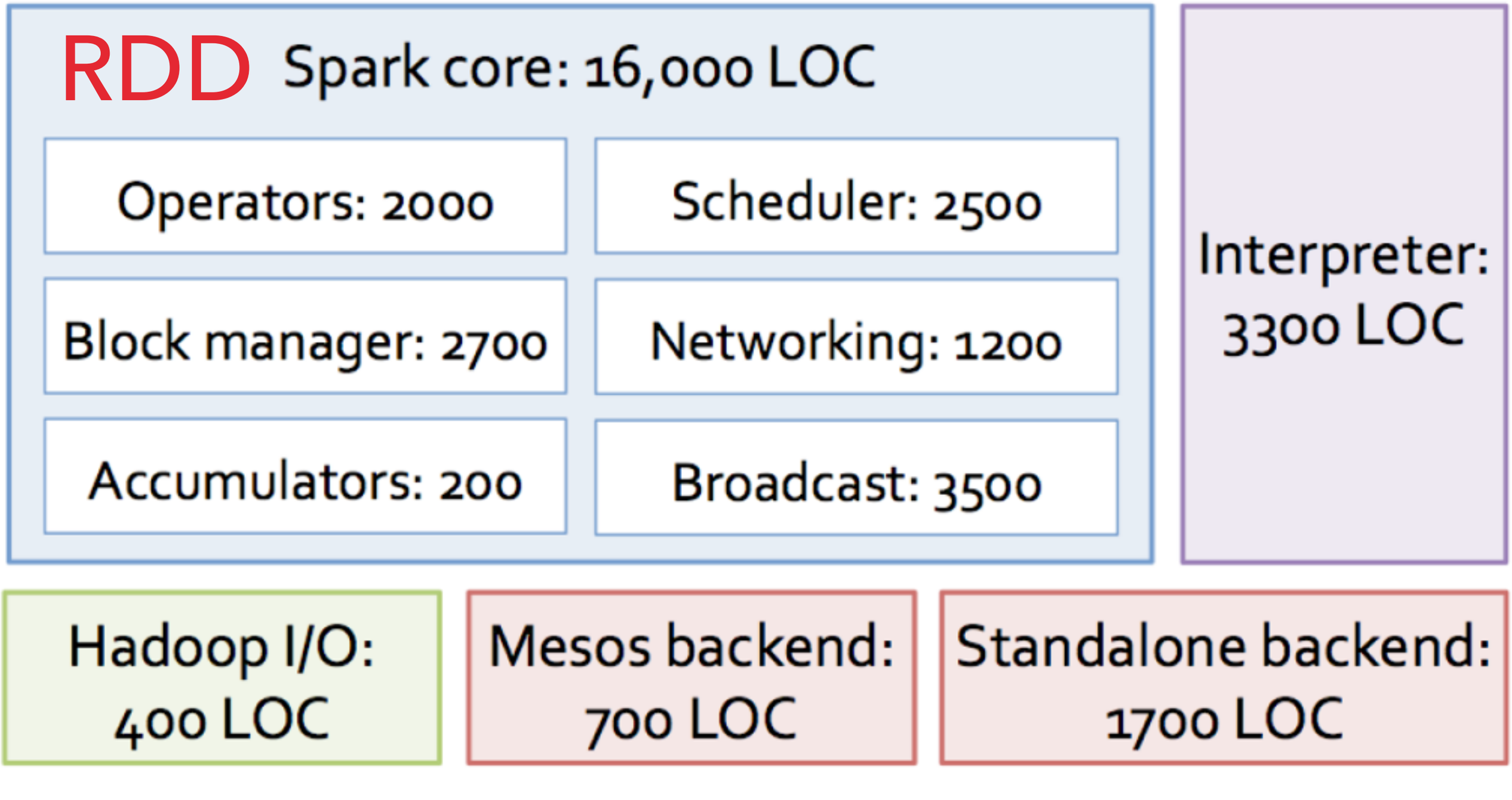
```
map[B](f: A => B): List[B] // Scala List
map[B](f: A => B): RDD[B] // Spark RDD
```

```
flatMap[B](f: A => TraversableOnce[B]): List[B] // Scala List
flatMap[B](f: A => TraversableOnce[B]): RDD[B] // Spark RDD

filter(pred: A => Boolean): List[A] // Scala List
filter(pred: A => Boolean): RDD[A] // Spark RDD

reduce(op: (A, A) => A): A // Scala List
reduce(op: (A, A) => A): A // Spark RDD

fold(z: A)(op: (A, A) => A): A // Scala List
fold(z: A)(op: (A, A) => A): A // Spark RDD

aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B // Scala
aggregate[B](z: B)(seqop: (B, A) => B, combop: (B, B) => B): B // Spark RDD
```

WHAT'S RDD

- Scientific Def: An interface containing five properties
  - Set of partitions
  - List of dependencies
  - Function to compute a partition given its parents
  - (Optional) Partition method
  - (Optional) Preferred locations

Spark Core components - code base (2012)



OUTLINE

- Intro
- Standalone-view RDD concepts
  - Basic RDD
  - Key-Pair RDD
- Distributed-view RDD concepts
  - Lazy evaluation
  - Cache
  - Partition/Shuffle
  - Broadcast
  - Accumulator
- Conclusion

} Operators

} Scheduler

} Block manager

} Networking

} Broadcast

} Accumulator

OUTLINE

- Intro
- Standalone-view RDD concepts
  - Basic RDD
  - Key-Pair RDD
- Distributed-view RDD concepts
  - Lazy evaluation
  - Cache
  - Partition/Shuffle
  - Broadcast
  - Accumulator
- Conclusion

} Operators

} Scheduler

} Block manager

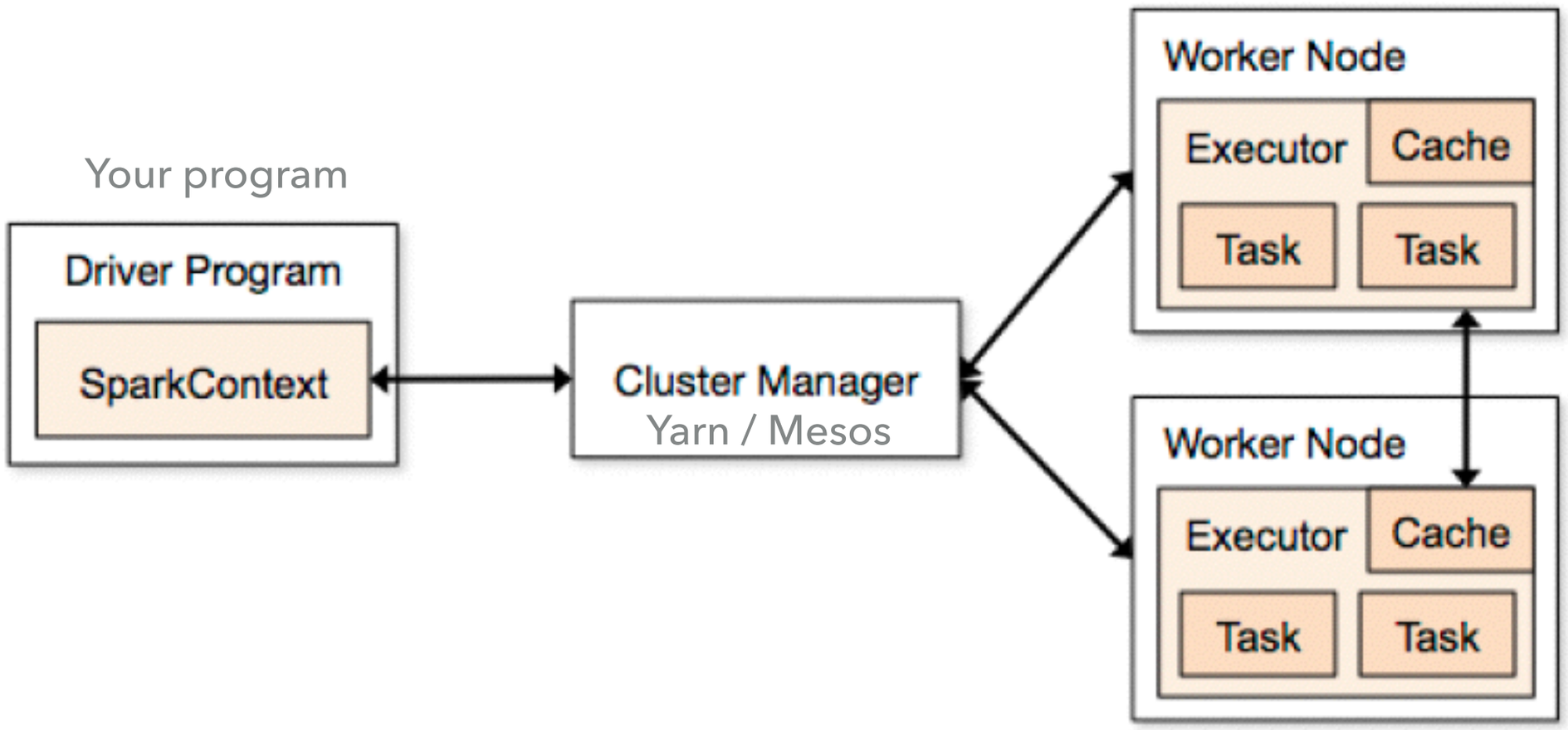
} Networking

} Broadcast

} Accumulator

Execution of Spark program

- Execution mode: Standalone / Cluster



OUTLINE

- Intro
- Standalone-view RDD concepts
  - Basic RDD
  - Key-Pair RDD
- Distributed-view RDD concepts
  - Lazy evaluation
  - Cache
  - Partition/Shuffle
  - Broadcast
  - Accumulator
- Conclusion

} Operators

} Scheduler

} Block manager

} Networking

} Broadcast

} Accumulator

STANDALONE-VIEW

Note: actions/transformations will not modify input RDD, but will always create a new one

RDD CREATION

- Loading an external dataset

Example 3-5. parallelize() method in Python

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

- Parallelizing a collection

Example 3-8. textFile() method in Python

```
lines = sc.textFile("/path/to/README.md")
```

TRANSFORMATIONS

Table 3-2. Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

TRANSFORMATIONS

Table 3-3. Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
union()	Produce an RDD containing elements from both RDDs.	rdd.union(other)	{1, 2, 3, 3, 4, 5}
intersection()	RDD containing only elements found in both RDDs.	rdd.intersection(other)	{3}
subtract()	Remove the contents of one RDD (e.g., remove training data).	rdd.subtract(other)	{1, 2}
cartesian()	Cartesian product with the other RDD.	rdd.cartesian(other)	{(1, 3), (1, 4), … (3,5)}

ACTIONS

Table 3-4. Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`

`take(num)`	Return num elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top num elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return num elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplacement, num, [seed])`	Return num elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., sum).	`rdd.reduce((x, y) => x + y)`	9
`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	9
`aggregate(zeroValue)(seqOp, combOp)`	Similar to `reduce()` but used to return a different type.	`rdd.aggregate((0, 0))((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	`(9, 4)`
`foreach(func)`	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing

PYTHON EXAMPLE - COUNT RATING NUM FOR EACH STAR

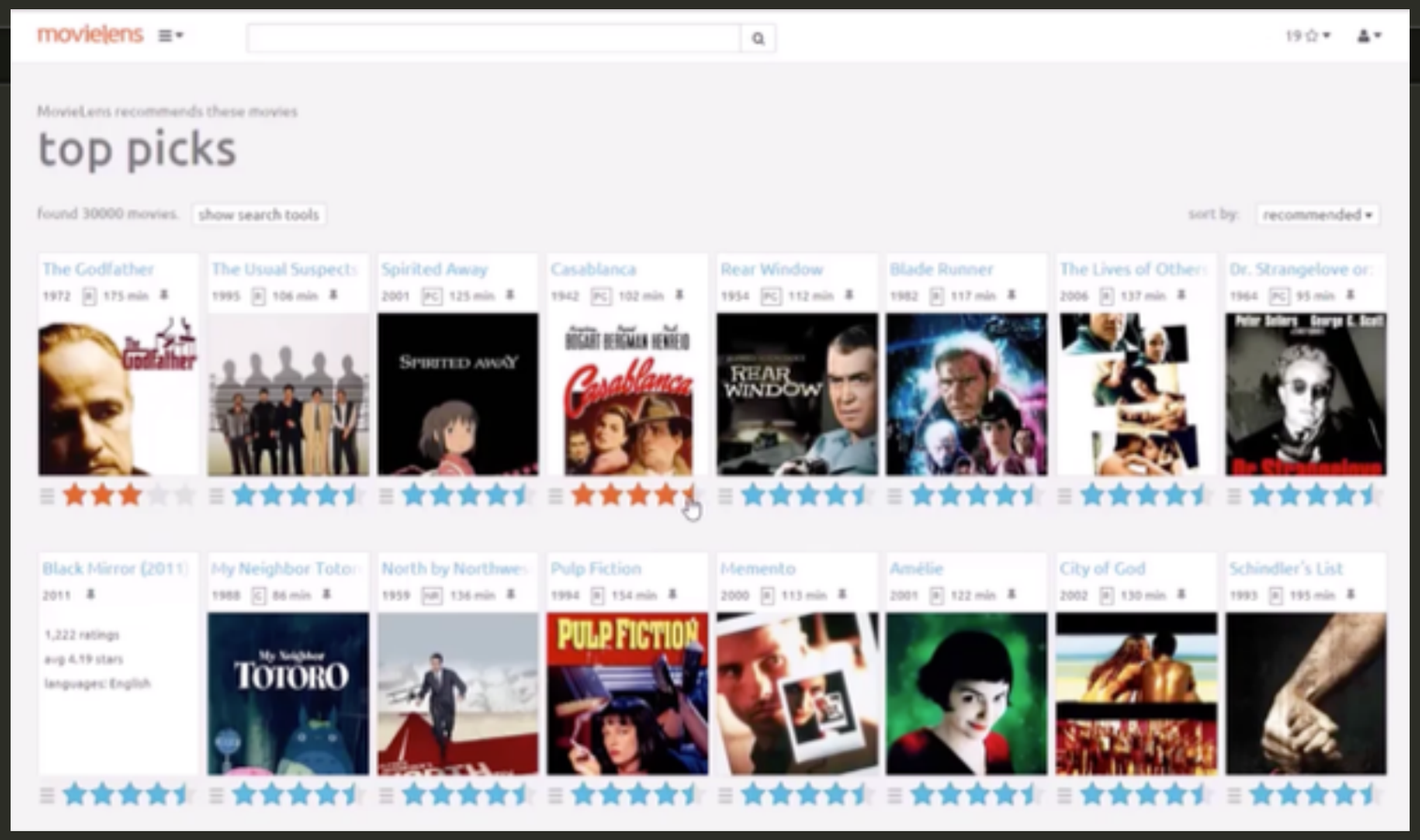
Ratings-Counter.py

```python
from pyspark import SparkConf, SparkContext
import collections

conf = SparkConf().setMaster("local").setAppName("RatingsHistogram")
sc = SparkContext(conf = conf)

lines = sc.textFile("file:///SparkCourse/ml-100k/u.data")
ratings = lines.map(lambda x: x.split()[2])
result = ratings.countByValue()

sortedResults = collections.OrderedDict(sorted(result.items()))
for key, value in sortedResults.iteritems():
    print "%s %i" % (key, value)
```

u.data

README

```
# movie rating data from grouplens.org
# user id | item id | rating | timestamp.

196 242 3   881250949
186 302 3   891717742
22  377 1   878887116
244 51  2   880606923
166 346 1   886397596
298 474 4   884182806
115 265 2   881171488
253 465 5   891628467
305 451 3   886324817
6   86  3   883603013
62  257 2   879372434
286 1014    5   879781125
200 222 5   876042340
210 40  3   891035994
224 29  3   888104457
303 785 3   879485318
122 387 5   879270459
194 274 2   879539794
291 1042    4   874834944
234 1184    2   892079237
119 392 4   886176814
167 486 4   892738452
299 144 4   877881320
```



For more detailed step, see:

https://www.udemy.com/taming-big-data-with-apache-spark-hands-on/learn/v4/overview

PYTHON EXAMPLE - COUNT RATING NUM FOR EACH STAR

```
1
2  from pyspark import SparkConf, SparkContext
3  import collections
4
5  conf = SparkConf().setMaster("local").setAppName("RatingsHistogram")
6  sc = SparkContext(conf = conf)
7
8  lines = sc.textFile("file:///SparkCourse/ml-100k/u.data")
9  ratings = lines.map(lambda x: x.split()[2])
10 result = ratings.countByValue()
11
12 sortedResults = collections.OrderedDict(sorted(result.items()))
13 for key, value in sortedResults.iteritems():
14     print "%s %i" % (key, value)
15
```

PYTHON EXAMPLE - COUNT RATING NUM FOR EACH STAR

- **The Spark Framework can adopt several cluster managers**
  - *Local Mode*
  - *Standalone mode*
  - *Apache Mesos*
  - *Hadoop YARN*

```
1
2  from pyspark import SparkConf, SparkContext
3  import collections
4
5  conf = SparkConf().setMaster("local").setAppName("RatingsHistogram")
6  sc = SparkContext(conf = conf)
7
8  lines = sc.textFile("file:///SparkCourse/ml-100k/u.data")
9  ratings = lines.map(lambda x: x.split()[2])
10 result = ratings.countByValue()
11
12 sortedResults = collections.OrderedDict(sorted(result.items()))
13 for key, value in sortedResults.iteritems():
14     print "%s %i" % (key, value)
15
```

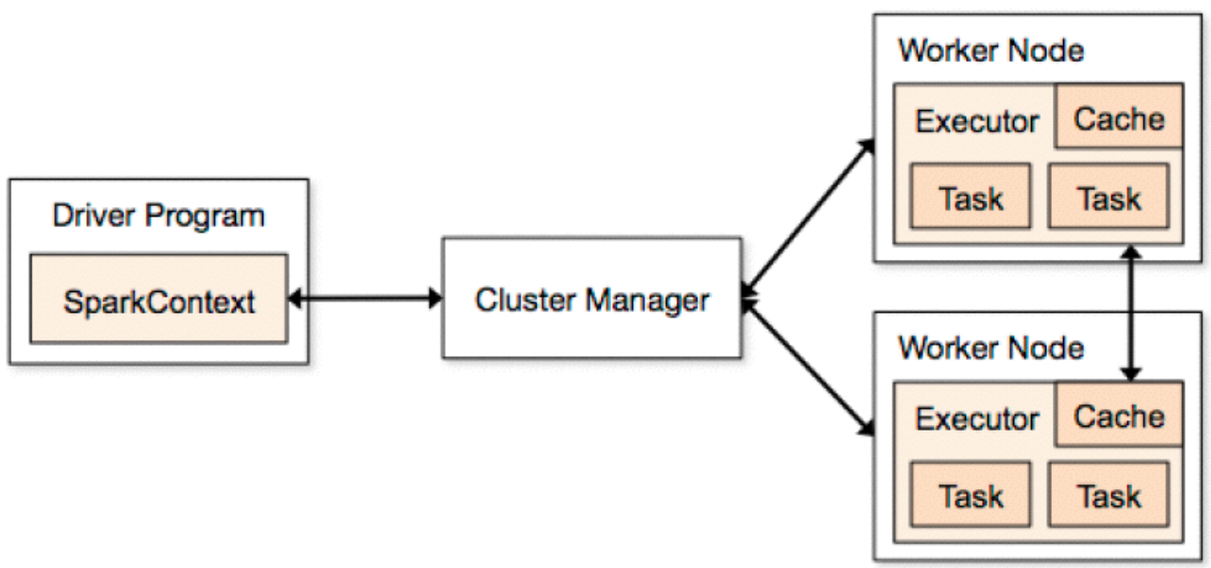
running mode

app name

PYTHON EXAMPLE - COUNT RATING NUM FOR EACH STAR

```
1
2  from pyspark import SparkConf, SparkContext
3  import collections
4
5  conf = SparkConf().setMaster("local").setAppName("RatingsHistogram")
6  sc = SparkContext(conf = conf)
7
8  lines = sc.textFile("file:///SparkCourse/ml-100k/u.data")
9  ratings = lines.map(lambda x: x.split()[2])
10 result = ratings.countByValue()
11
12 sortedResults = collections.OrderedDict(sorted(result.items()))
13 for key, value in sortedResults.iteritems():
14     print "%s %i" % (key, value)
15
```

get a Spark Context, initialize the driver program



PYTHON EXAMPLE - COUNT RATING NUM FOR EACH STAR

```
1
2  from pyspark import SparkConf, SparkContext
3  import collections
4
5  conf = SparkConf().setMaster("local").setAppName("RatingsHistogram")
6  sc = SparkContext(conf = conf)
7
8  lines = sc.textFile("file:///SparkCourse/ml-100k/u.data")
9  ratings = lines.map(lambda x: x.split()[2])
10 result = ratings.countByValue()
11
12 sortedResults = collections.OrderedDict(sorted(result.items()))
13 for key, value in sortedResults.iteritems():
14     print "%s %i" % (key, value)
15
```

read a text file as a RDD

PYTHON EXAMPLE - COUNT RATING NUM FOR EACH STAR

```
1
2  from pyspark import SparkConf, SparkContext
3  import collections
4
5  conf = SparkConf().setMaster("local").setAppName("RatingsHistogram")
6  sc = SparkContext(conf = conf)
7
8  lines = sc.textFile("file:///SparkCourse/ml-100k/u.data")
9  ratings = lines.map(lambda x: x.split()[2])
10 result = ratings.countByValue()
11
12 sortedResults = collections.OrderedDict(sorted(result.items()))
13 for key, value in sortedResults.iteritems():
14     print "%s %i" % (key, value)
15
```

transformation

user id	item id	rating	timestamp
192	242	3	881250949
186	302	3	891717742

3
3

PYTHON EXAMPLE - COUNT RATING NUM FOR EACH STAR

```
1
2  from pyspark import SparkConf, SparkContext
3  import collections
4
5  conf = SparkConf().setMaster("local").setAppName("RatingsHistogram")
6  sc = SparkContext(conf = conf)
7
8  lines = sc.textFile("file:///SparkCourse/ml-100k/u.data")
9  ratings = lines.map(lambda x: x.split()[2])
10 result = ratings.countByValue()
11
12 sortedResults = collections.OrderedDict(sorted(result.items()))
13 for key, value in sortedResults.iteritems():
14     print "%s %i" % (key, value)
15
```

Action which returns (value, count) pairs

3
3
3
4
4
5

(3,3)
(4,2)
(5,1)

PYTHON EXAMPLE - COUNT RATING NUM FOR EACH STAR

```
1
2  from pyspark import SparkConf, SparkContext
3  import collections
4
5  conf = SparkConf().setMaster("local").setAppName("RatingsHistogram")
6  sc = SparkContext(conf = conf)
7
8  lines = sc.textFile("file:///SparkCourse/ml-100k/u.data")
9  ratings = lines.map(lambda x: x.split()[2])
10 result = ratings.countByValue()
11
12 sortedResults = collections.OrderedDict(sorted(result.items()))
13 for key, value in sortedResults.iteritems():
14     print "%s %i" % (key, value)
15
```

Loop through, printout python collection

```
(Canopy 64bit) C:\SparkCourse>spark-submit ratings-counter.py
15/09/29 11:39:35 WARN NativeCodeLoader: Unable to load native-hadoop library fo
r your platform... using builtin-java classes where applicable
15/09/29 11:39:37 WARN : Your hostname, HomeOfficePower resolves to a loopback/n
on-reachable address: fe80:0:0:0:0:5efe:c0a8:180%net2, but we couldn't find any
external IP address!
15/09/29 11:39:38 WARN MetricsSystem: Using default name DAGScheduler for source
 because spark.app.id is not set.
1 6110
2 11370
3 27145
4 34174
5 21201
```

KEY-PAIR RDD

- Def: RDDs containing key/value pairs
- Motivation: allows to act on each key

- Create from RDDs by transforming to tuples

Example 4-1. Creating a pair RDD using the first word as the key in Python

```
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```

- Load external data

KEY-PAIR RDD TRANSFORMATIONS

- Key-Pair RDDs have all transformations available to standard RDDs

Table 4-1. Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)})

Function name	Purpose	Example	Result
`reduceByKey(func)`	Combine values with the same key.	`rdd.reduceByKey( (x, y) => x + y)`	`{(1, 2), (3, 10)}`
`groupByKey()`	Group values with the same key.	`rdd.groupByKey()`	`{(1, [2]), (3, [4, 6])}`
`combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)`	Combine values with the same key using a different result type.	See Examples 4-12 through 4-14.	
`mapValues(func)`	Apply a function to each value of a pair RDD without changing the key.	`rdd.mapValues(x => x+1)`	`{(1, 3), (3, 5), (3, 7)}`
`flatMapValues(func)`	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	`rdd.flatMapValues(x => (x to 5)`	`{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}`
`keys()`	Return an RDD of just the keys.	`rdd.keys()`	`{1, 3, 3}`

- Key-Pair RDDs have all transformations available to standard RDDs

Function name	Purpose	Example	Result
`values()`	Return an RDD of just the values.	`rdd.values()`	`{2, 4, 6}`
`sortByKey()`	Return an RDD sorted by the key.	`rdd.sortByKey()`	`{(1, 2), (3, 4), (3, 6)}`

Table 4-2. Transformations on two pair RDDs (rdd = {(1, 2), (3, 4), (3, 6)} other = {(3, 9)})

Function name	Purpose	Example	Result
`subtractByKey`	Remove elements with a key present in the other RDD.	`rdd.subtractByKey(other)`	`{(1, 2)}`
`join`	Perform an inner join between two RDDs.	`rdd.join(other)`	`{(3, (4, 9)), (3, (6, 9))}`
`rightOuterJoin`	Perform a join between two RDDs where the key must be present in the first RDD.	`rdd.rightOuterJoin(other)`	`{(3,(Some(4),9)), (3,(Some(6),9))}`
`leftOuterJoin`	Perform a join between two RDDs where the key must be present in the other RDD.	`rdd.leftOuterJoin(other)`	`{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}`
`cogroup`	Group data from both RDDs sharing the same key.	`rdd.cogroup(other)`	`{(1,([2],[])), (3, ([4, 6],[9]))}`

Table 4-3. Actions on pair RDDs (example ({(1, 2), (3, 4), (3, 6)}))

Function	Description	Example	Result
`countByKey()`	Count the number of elements for each key.	`rdd.countByKey()`	`{(1, 1), (3, 2)}`
`collectAsMap()`	Collect the result as a map to provide easy lookup.	`rdd.collectAsMap()`	`Map{(1, 2), (3, 4), (3, 6)}`
`lookup(key)`	Return all values associated with the provided key.	`rdd.lookup(3)`	`[4, 6]`

OUTLINE

- Intro
- Standalone-view RDD concepts
  - Basic RDD
  - Key-Pair RDD
- Distributed-view RDD concepts
  - Lazy evaluation
  - Cache
  - Partition/Shuffle
  - Broadcast
  - Accumulator
- Conclusion

} Operators

} Scheduler

} Block manager

} Networking

} Broadcast

} Accumulator

OUTLINE

- Intro
- Standalone-view RDD concepts
  - Basic RDD
  - Key-Pair RDD
- Distributed-view RDD concepts
  - Lazy evaluation
  - Cache
  - Partition/Shuffle
  - Broadcast
  - Accumulator
- Conclusion

} Operators

} Scheduler

} Block manager

} Networking

} Broadcast

} Accumulator

LAZY-EVALUATION

- Def:
  - Transformations are lazy. Their result RDD is not immediately computed.
  - Actions are eager. Their result is immediately computed.
- Example:

```
lines = sc.textFile("file:///SparkCourse/ml-100k/u.data")
ratings = lines.map(lambda x: x.split()[2])
result = ratings.countByValue()
```

loading data operation, not executed at this point

LAZY-EVALUATION

- Def:
  - Transformations are lazy. Their result RDD is not immediately computed.
  - Actions are eager. Their result is immediately computed.
- Example:

```
lines = sc.textFile("file:///SparkCourse/ml-100k/u.data")
ratings = lines.map(lambda x: x.split()[2])
result = ratings.countByValue()
```

map transformation operation, not executed at this point

LAZY-EVALUATION

- Def:
  - Transformations are lazy. Their result RDD is not immediately computed.
  - Actions are eager. Their result is immediately computed.
- Example:

```
lines = sc.textFile("file:///SparkCourse/ml-100k/u.data")
ratings = lines.map(lambda x: x.split()[2])
result = ratings.countByValue()
```

countByValue action operation, executed immediately

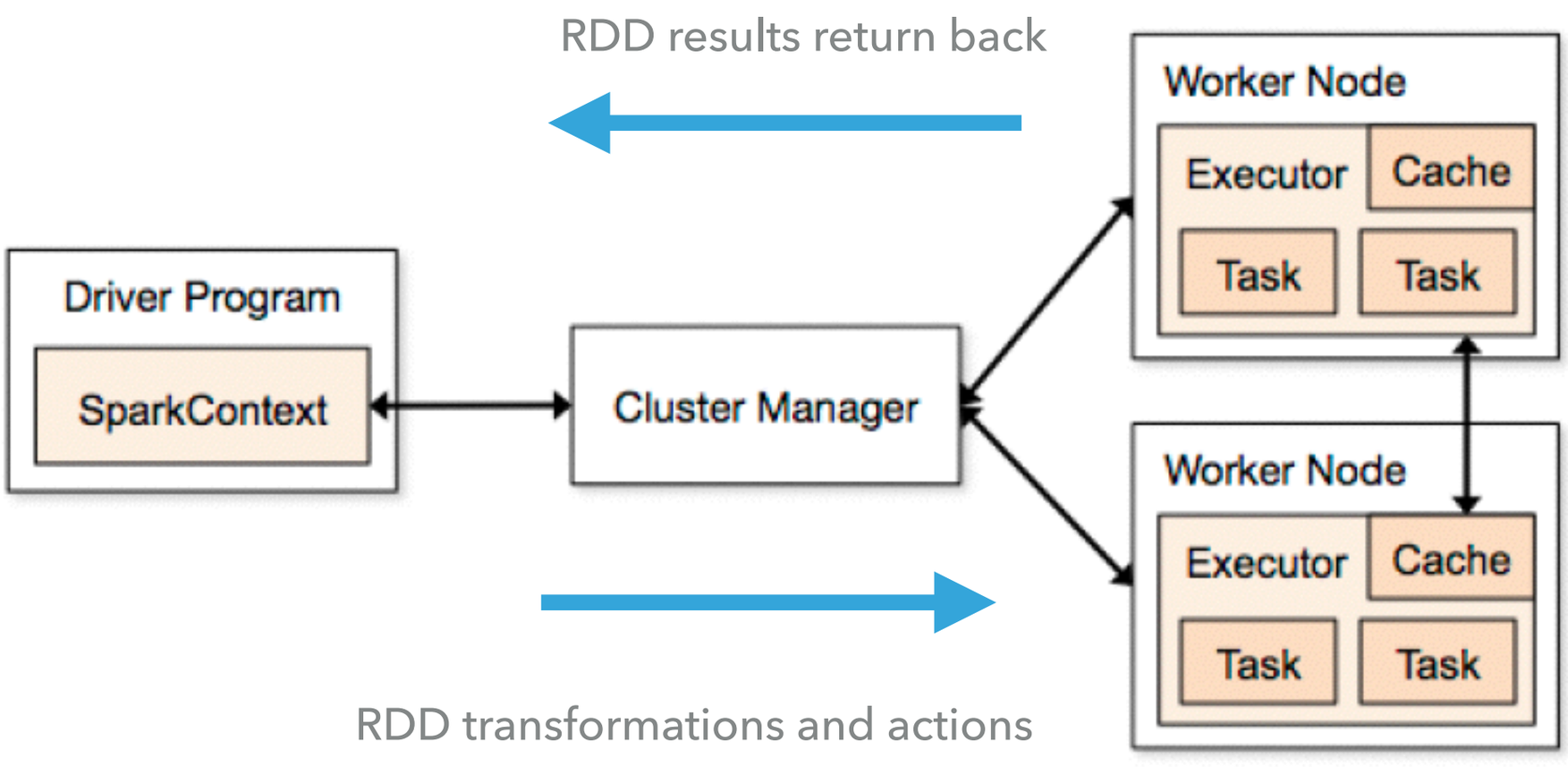
LAZY-EVALUATION

- Why:
  - Enable computation optimization

```
val lastYearsLogs: RDD[String] = ...
val firstLogsWithErrors = lastYearsLogs.filter(_.contains("ERROR")).take(10)
```

LAZY-EVALUATION

- Why:
  - Limit network communication
    - If you perform an action, its result returns back to the driver program

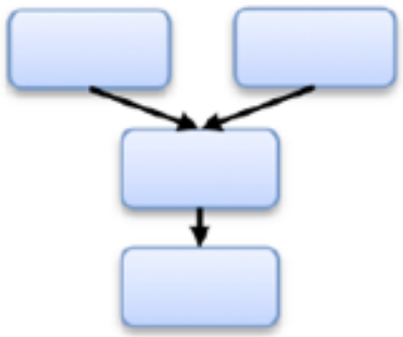


LAZY-EVALUATION

- What does it do when loading/transformation are performed:
  - Record RDD dependencies with a directed acyclic graph (DAG)
    - For failure recovery
    - Pipeline operations execution order
    - Optimize by utilizing network traffic based on partition and shuffling

Life time of a job in Spark

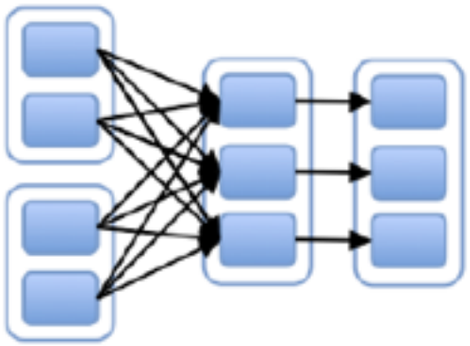
RDD Objects



```
rdd1.join(rdd2)
.groupBy(...)
.filter(...)
```

Build the operator DAG

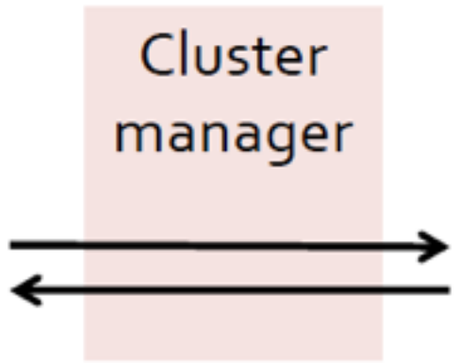
DAG Scheduler



Split the DAG into *stages* of *tasks*

Submit each stage and its tasks as ready

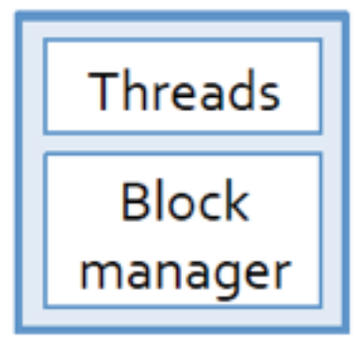
Task Scheduler



Launch tasks via Master

Retry failed and straggler tasks

Worker



Execute tasks

Store and serve blocks

OUTLINE

- Intro
- Standalone-view RDD concepts
  - Basic RDD
  - Key-Pair RDD
- Distributed-view RDD concepts
  - Lazy evaluation
  - Cache
  - Partition/Shuffle
  - Broadcast
  - Accumulator
- Conclusion

} Operators

} Scheduler

} Block manager

} Networking

} Broadcast

} Accumulator

- Def
  - By default, RDDs are recomputed each time you run an action on them

Example 3-39. Double execution in Scala

```
val result = input.map(x => x*x)
println(result.count())
println(result.collect().mkString(","))
```

Example 3-40. persist() in Scala

```
val result = input.map(x => x * x)
result.persist(StorageLevel.DISK_ONLY)
println(result.count())
println(result.collect().mkString(","))
```

- Cache options

Table 3-6. Persistence levels from org.apache.spark.storage.StorageLevel and pyspark.StorageLevel; if desired we can replicate the data on two machines by adding _2 to the end of the storage level

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

CACHE

- Why
  - Avoid network IO and recomputation
- How
  - Implemented with Least Recently Used (LRU) cache policy
  - Users do not have to worry about caching too much data

OUTLINE

- Intro
- Standalone-view RDD concepts
  - Basic RDD
  - Key-Pair RDD
- Distributed-view RDD concepts
  - Lazy evaluation
  - Cache
  - Shuffle/Partition
  - Broadcast
  - Accumulator
- Conclusion

} Operators

} Scheduler

} Block manager

} Networking

} Broadcast

} Accumulator

OUTLINE

- Intro
- Standalone-view RDD concepts
  - Basic RDD
  - Key-Pair RDD
- Distributed-view RDD concepts
  - Lazy evaluation
  - Cache
  - Shuffle/Partition
  - Broadcast
  - Accumulator
- Conclusion

} Operators

} Scheduler

} Block manager

} Networking

} Broadcast

} Accumulator

Let's start with an example. Given:

```
case class CFFPurchase(customerId: Int, destination: String, price: Double)
```

Assume we have an RDD of the purchases that users of the CFF mobile app have made in the past month.

```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)
```

Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)

// Returns: Array[(Int, (Int, Double))]
val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
              .groupByKey() // groupByKey returns RDD[K, Iterable[V]]
              .map(p => (p._1, (p._2.size, p._2.sum)))
              .collect()
```

Let's start with an example dataset:

```
val purchases = List(CFFPurchase(100, "Geneva", 22.25),
                     CFFPurchase(300, "Zurich", 42.10),
                     CFFPurchase(100, "Fribourg", 12.40),
                     CFFPurchase(200, "St. Gallen", 8.20),
                     CFFPurchase(100, "Lucerne", 31.60),
                     CFFPurchase(300, "Basel", 16.20))
```

What might the cluster look like with this data distributed over it?

What might the cluster look like with this data distributed over it?

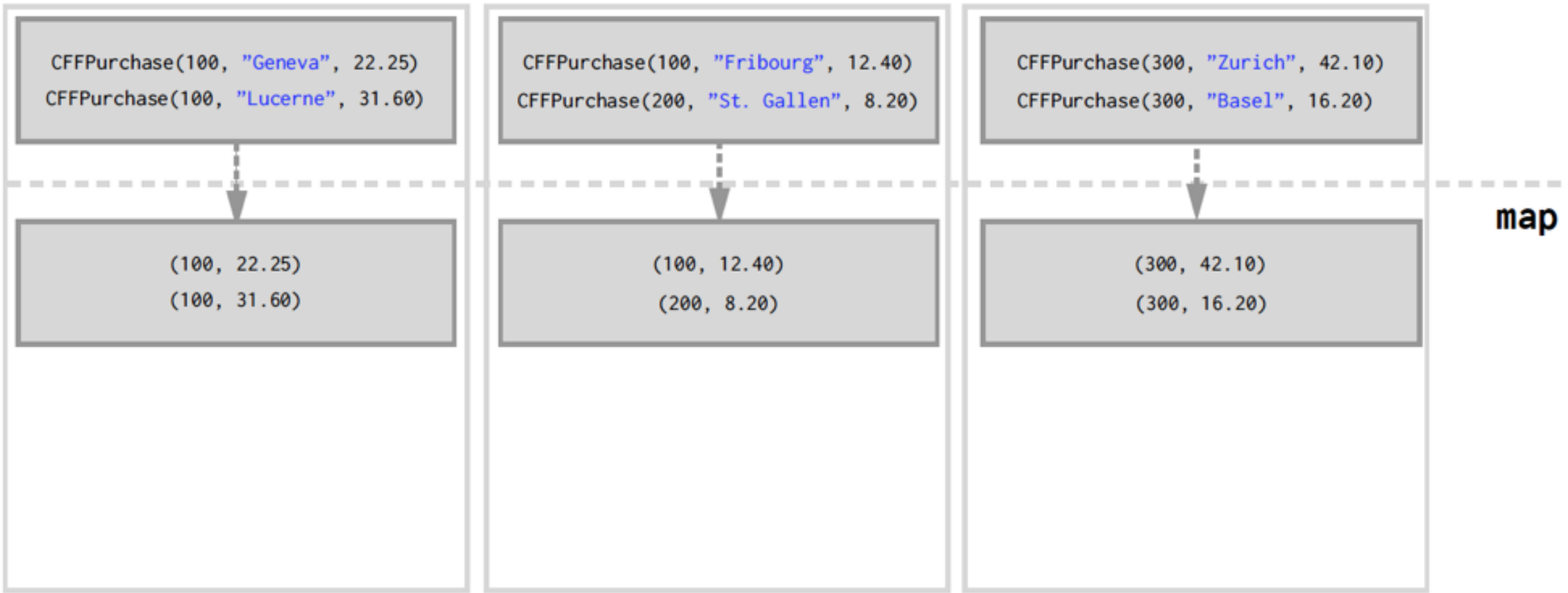
Starting with purchasesRdd:

```
CFFPurchase(100, "Geneva", 22.25)
CFFPurchase(100, "Lucerne", 31.60)
```

```
CFFPurchase(100, "Fribourg", 12.40)
CFFPurchase(200, "St. Gallen", 8.20)
```

```
CFFPurchase(300, "Zurich", 42.10)
CFFPurchase(300, "Basel", 16.20)
```

What might this look like on the cluster?



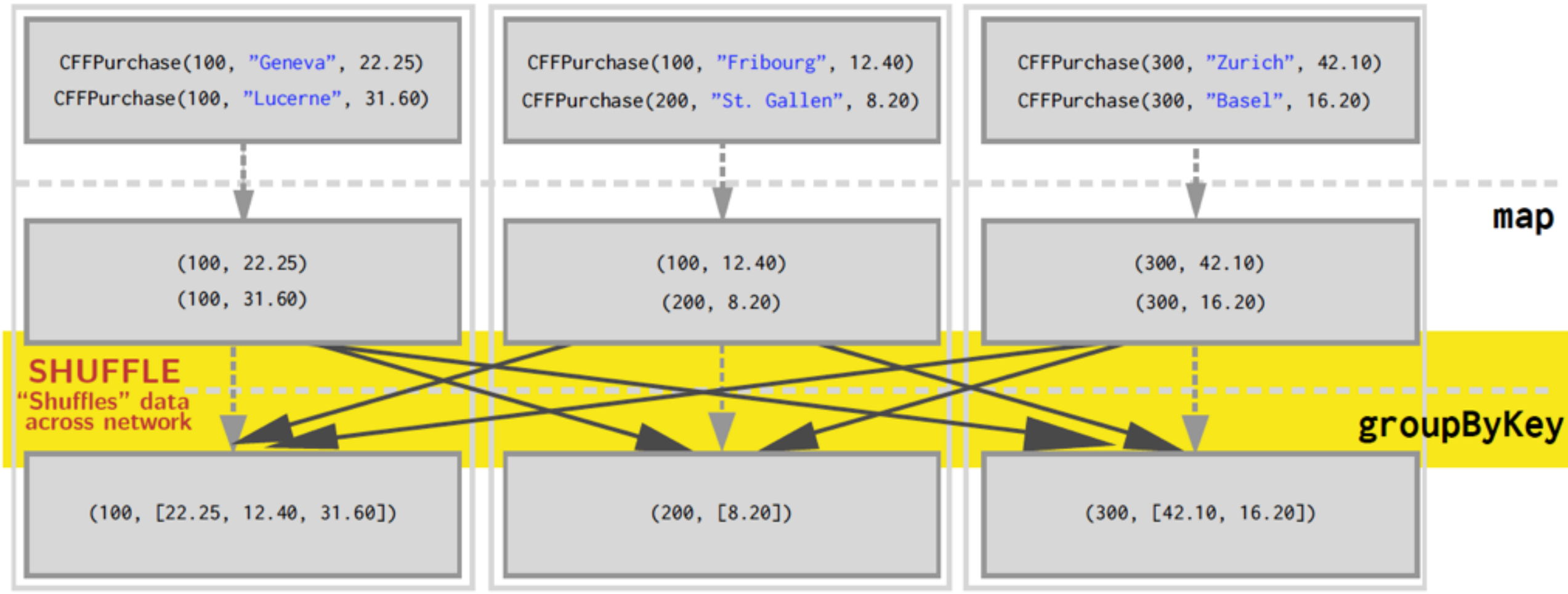
Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)
```

```
val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
              .groupByKey() // groupByKey returns RDD[K, Iterable[V]]
```

Note: groupByKey results in one key-value pair per key. And this single key-value pair cannot span across multiple worker nodes.

SHUFFLE


CFFPurchase(100, "Geneva", 22.25)
CFFPurchase(100, "Lucerne", 31.60)
CFFPurchase(100, "Fribourg", 12.40)
CFFPurchase(200, "St. Gallen", 8.20)
CFFPurchase(300, "Zurich", 42.10)
CFFPurchase(300, "Basel", 16.20)
map
(100, 22.25)
(100, 31.60)
(100, 12.40)
(200, 8.20)
(300, 42.10)
(300, 16.20)
SHUFFLE
"Shuffles" data across network
groupByKey
(100, [22.25, 12.40, 31.60])
(200, [8.20])
(300, [42.10, 16.20])

REMINDER: LATENCY MATTERS (HUMANIZED)

Shared Memory

Seconds

L1 cache reference.....0.5s

L2 cache reference.....7s

Mutex lock/unlock.....25s

Minutes

Main memory reference.....1m 40s

Distributed

Days

Roundtrip within
same datacenter.....5.8 days

Years

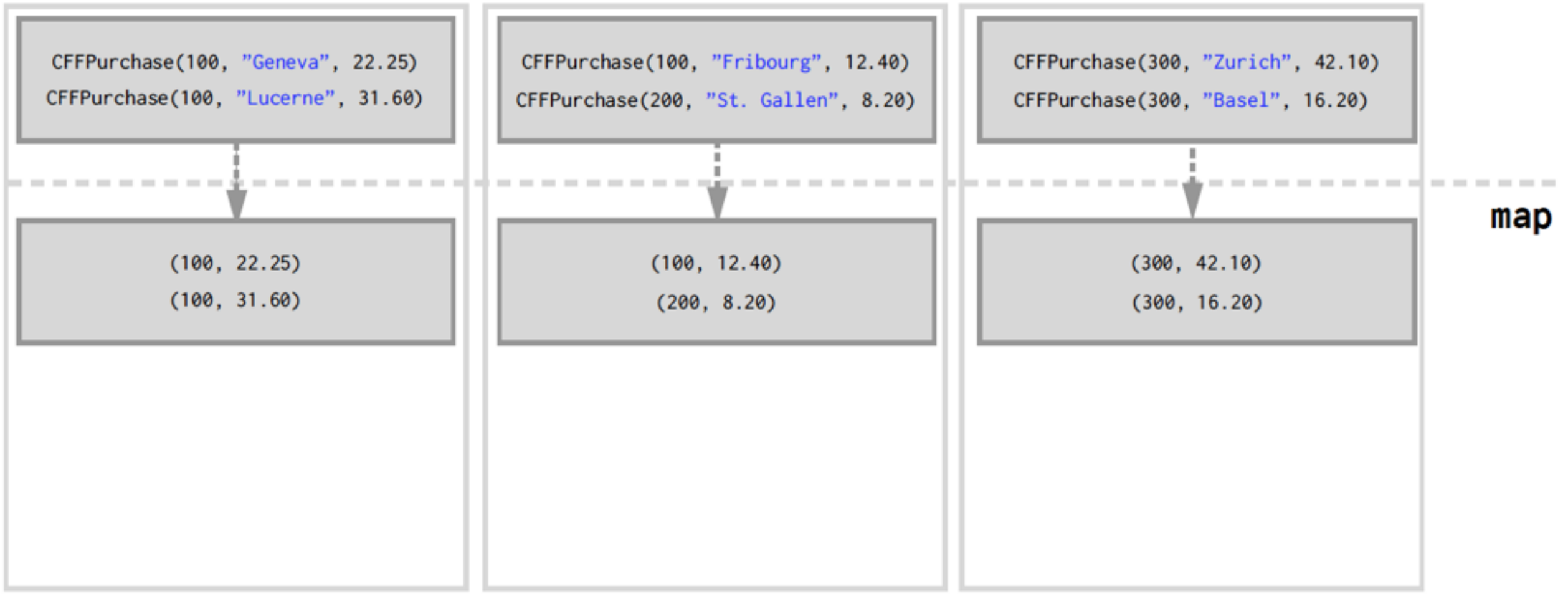
Send packet
CA->Netherlands->CA....4.8 years

We don't want to be sending all of our data over the network if it's not absolutely required. Too much network communication kills performance.

SHUFFLE

- Can we do a better job

Perhaps we don't need to send all pairs over the network.



Perhaps we can reduce before we shuffle. This could greatly reduce the amount of data we have to send over the network.

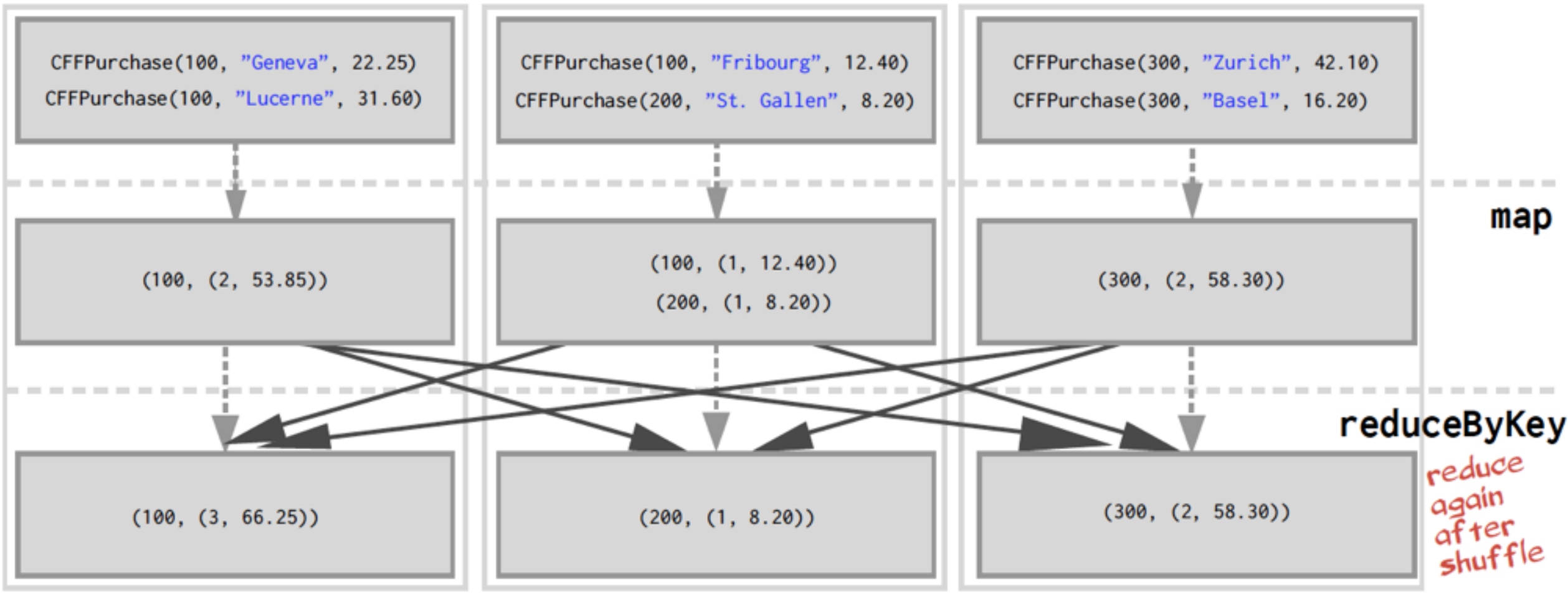
We can use `reduceByKey`.

Conceptually, `reduceByKey` can be thought of as a combination of first doing `groupByKey` and then `reduce`-ing on all the values grouped per key. It's more efficient though, than using each separately. We'll see how in the following example.

Signature:

```
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
```

What might this look like on the cluster?



What are the benefits of this approach?

By reducing the dataset first, the amount of data sent over the network during the shuffle is greatly reduced.

This can result in non-trival gains in performance!

Let's benchmark on a real cluster.

SHUFFLE

```
> val purchasesPerMonthSlowLarge = purchasesRddLarge.map(p => (p.customerId, p.price))
                                                    .groupByKey()
                                                    .map(p => (p._1, (p._2.size, p._2.sum)))
                                                    .count()

  purchasesPerMonthSlowLarge: Long = 100000
  Command took 15.48s
```

```
> val purchasesPerMonthFastLarge = purchasesRddLarge.map(p => (p.customerId, (1, p.price)))
                                                    .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
                                                    .count()

  purchasesPerMonthFastLarge: Long = 100000
  Command took 4.65s
```

Full example with 20 million element RDD can be found in the lecture2-apr2 notebook on our Databricks Cloud installation.

Recall our example using `groupByKey`:

```
val purchasesPerCust =
  purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
              .groupByKey()
```

Grouping all values of key-value pairs with the same key requires collecting all key-value pairs with the same key on the same machine.

But how does Spark know which key to put on which machine?

- By default, Spark uses *hash partitioning* to determine which key-value pair should be sent to which machine.

OUTLINE

- Intro
- Standalone-view RDD concepts
  - Basic RDD
  - Key-Pair RDD
- Distributed-view RDD concepts
  - Lazy evaluation
  - Cache
  - Shuffle/Partition
  - Broadcast
  - Accumulator
- Conclusion

} Operators

} Scheduler

} Block manager

} Networking

} Broadcast

} Accumulator

The data within an RDD is split into several *partitions*.

Properties of partitions:

- Partitions never span multiple machines, i.e., tuples in the same partition are guaranteed to be on the same machine.
- Each machine in the cluster contains one or more partitions.
- The number of partitions to use is configurable. By default, it equals the *total number of cores on all executor nodes*.

Two kinds of partitioning available in Spark:

- Hash partitioning
- Range partitioning

Customizing a partitioning is only possible on Pair RDDs.

Back to our example. Given a Pair RDD that should be grouped:

```
val purchasesPerCust =
  purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
              .groupByKey()
```

Back to our example. Given a Pair RDD that should be grouped:

```
val purchasesPerCust =
  purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
              .groupByKey()
```

`groupByKey` first computes per tuple `(k, v)` its partition `p`:

```
p = k.hashCode() % numPartitions
```

Then, all tuples in the same partition `p` are sent to the machine hosting `p`.

Intuition: hash partitioning attempts to spread data evenly across partitions *based on the key*.

RANGE PARTITION

Consider a Pair RDD, with keys `[8, 96, 240, 400, 401, 800]`, and a desired number of partitions of `4`.

RANGE PARTITION

Consider a Pair RDD, with keys `[8, 96, 240, 400, 401, 800]`, and a desired number of partitions of `4`.

Furthermore, suppose that `hashCode()` is the identity (`n.hashCode() == n`).

Consider a Pair RDD, with keys `[8, 96, 240, 400, 401, 800]`, and a desired number of partitions of `4`.

Furthermore, suppose that `hashCode()` is the identity (`n.hashCode() == n`).

In this case, hash partitioning distributes the keys as follows among the partitions:

- partition 0: `[8, 96, 240, 400, 800]`
- partition 1: `[401]`
- partition 2: `[]`
- partition 3: `[]`

The result is a very unbalanced distribution which hurts performance.

RANGE PARTITION

Pair RDDs may contain keys that have an *ordering* defined.

- Examples: `Int`, `Char`, `String`, ...

For such RDDs, *range partitioning* may be more efficient.

Using a range partitioner, keys are partitioned according to:

1. an *ordering* for keys
2. a set of *sorted ranges* of keys

Property: tuples with keys in the same range appear on the same machine.

RANGE PARTITION

Using *range partitioning* the distribution can be improved significantly:

- Assumptions: (a) keys non-negative, (b) `800` is biggest key in the RDD.
- Set of ranges: `[1, 200]`, `[201, 400]`, `[401, 600]`, `[601, 800]`

Using *range partitioning* the distribution can be improved significantly:

- Assumptions: (a) keys non-negative, (b) `800` is biggest key in the RDD.
- Set of ranges: `[1, 200]`, `[201, 400]`, `[401, 600]`, `[601, 800]`

In this case, range partitioning distributes the keys as follows among the partitions:

- partition 0: `[8, 96]`
- partition 1: `[240, 400]`
- partition 2: `[401]`
- partition 3: `[800]`

The resulting partitioning is much more balanced.

How do we set a partitioning for our data?

There are two ways to create RDDs with specific partitionings:

1. Call `partitionBy` on an RDD, providing an explicit `Partitioner`.
2. Using transformations that return RDDs with specific partitioners.

PARTITIONING DATA:

Invoking `partitionBy` creates an RDD with a specified partitioner.

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
val tunedPartitioner = new RangePartitioner(8, pairs)
val partitioned = pairs.partitionBy(tunedPartitioner).persist()
```

Creating a `RangePartitioner` requires:

1. Specifying the desired number of partitions.
2. Providing a Pair RDD with *ordered keys*. This RDD is *sampled* to create a suitable set of *sorted ranges*.

Important: the result of `partitionBy` should be persisted. Otherwise, the partitioning is repeatedly applied (involving shuffling!) each time the partitioned RDD is used.

Partitioner from parent RDD:

Pair RDDs that are the result of a transformation on a *partitioned* Pair RDD typically is configured to use the hash partitioner that was used to construct it.

Automatically-set partitioners:

Some operations on RDDs automatically result in an RDD with a known partitioner – for when it makes sense.

For example, by default, when using `sortByKey`, a `RangePartitioner` is used. Further, the default partitioner when using `groupByKey`, is a `HashPartitioner`, as we saw earlier.

Operations on Pair RDDs that hold to (and propagate) a partitioner:

- `cogroup`
- `groupWith`
- `join`
- `leftOuterJoin`
- `rightOuterJoin`
- `groupByKey`
- `reduceByKey`
- `foldByKey`
- `combineByKey`
- `partitionBy`
- `sort`
- `mapValues` (if parent has a partitioner)
- `flatMapValues` (if parent has a partitioner)
- `filter` (if parent has a partitioner)

All other operations will produce a result without a partitioner.

...All other operations will produce a result without a partitioner.

Why?

Consider the `map` transformation. Given have a hash partitioned Pair RDD, why would it make sense for `map` to lose the partitioner in its result RDD?

Because it's possible for `map` to change the key . *E.g.,*:

```
rdd.map((k: String, v: Int) => ("doh!", v))
```

In this case, if the `map` transformation preserved the partitioner in the result RDD, it no longer make sense, as now the keys are all different.

Hence `mapValues`. It enables us to still do `map` transformations without changing the keys, thereby preserving the partitioner.

OPTIMIZING PREVIOUS EXAMPLE USING RANGE PARTITIONING

```
> val purchasesPerMonthSlowLarge = purchasesRddLarge.map(p => (p.customerId, p.price))
                                                    .groupByKey()
                                                    .map(p => (p._1, (p._2.size, p._2.sum)))
                                                    .count()

  purchasesPerMonthSlowLarge: Long = 100000
  Command took 15.48s
```

```
> val purchasesPerMonthFastLarge = purchasesRddLarge.map(p => (p.customerId, (1, p.price)))
                                                    .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
                                                    .count()

  purchasesPerMonthFastLarge: Long = 100000
  Command took 4.65s
```

On the range partitioned data:

```
> val purchasesPerMonthFasterLarge = partitioned.map(x => x)
                                                .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
                                                .count()

  purchasesPerMonthFasterLarge: Long = 100000
  Command took 1.79s
```

almost a 9x speedup over purchasePerMonthSlowLarge!

HOW DO I KNOW A SHUFFLE WILL OCCUR?

Rule of thumb: a shuffle *can* occur when the resulting RDD depends on other elements from the same RDD or another RDD.

Note: sometimes one can be clever and avoid much or all network communication while still using an operation like `join` *via smart partitioning*

You can also figure out whether a shuffle has been planned/executed via:

1. The return type of certain transformations, *e.g.*,

```
org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[366]
```

2. Using function `toDebugString` to see its execution plan:

```
partitioned.reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
           .toDebugString
res9: String =
(8) MapPartitionsRDD[622] at reduceByKey at <console>:49 []
 |  ShuffledRDD[615] at partitionBy at <console>:48 []
 |      CachedPartitions: 8; MemorySize: 1754.8 MB; DiskSize: 0.0 B
```

HOW DO I KNOW A SHUFFLE WILL OCCUR?

- Operations that might cause a shuffle
  - cogroup
  - groupWith
  - join
  - leftOuterJoin
  - rightOuterJoin
  - groupByKey
  - reduceByKey
  - combineByKey
  - distinct
  - intersection
  - repartition
  - coalesce

OUTLINE

- Intro
- Standalone-view RDD concepts
  - Basic RDD
  - Key-Pair RDD
- Distributed-view RDD concepts
  - Lazy evaluation
  - Cache
  - Shuffle/Partition
  - Broadcast
  - Accumulator
- Conclusion

} Operators

} Scheduler

} Block manager

} Networking

} Broadcast

} Accumulator

SHARED VARIABLES

- Start with an example

```
val points: RDD[Point] = // ...
var clusterCenters = new Array[Point](k)

val closestCenter = points.map {
  p => findClosest(clusterCenters, p)
}
...
```

How should this variable be shipped?

BROADCAST VARIABLES

- Motivation:
  - Normally, Spark closures, including variables they use, are sent separately within each task.
    - But in same cases, the same variable will be used in multiple parallel operations
  - By default, Spark task launching mechanism is optimized for small task sizes.
    - But in some cases, a large read-only variable needs to be shared across tasks, or across operations
- Examples: large lookup tables

Example 6-6. Country lookup in Python

```
# Look up the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup.
signPrefixes = loadCallSignTable()

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

Example 6-7. Country lookup with Broadcast values in Python

```
# Look up the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup.
signPrefixes = sc.broadcast(loadCallSignTable())

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

OUTLINE

- Intro
- Standalone-view RDD concepts
  - Basic RDD
  - Key-Pair RDD
- Distributed-view RDD concepts
  - Lazy evaluation
  - Cache
  - Shuffle/Partition
  - Broadcast
  - Accumulator
- Conclusion

} Operators

} Scheduler

} Block manager

} Networking

} Broadcast

} Accumulator

BROADCAST VARIABLES

```
# Look up the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup.
signPrefixes = sc.broadcast(loadCallSignTable())

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))

countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

Create with
sparkContext.broadcast(initVal)

Access with .value inside tasks
First task on each node to fetch val

!!! Cannot modify after creation
if you try, change only on one node

ACCUMULATORS

- Motivation:
  - Normally, Spark closures, including variables they use, are sent separately within each task.
    - But values inside worker nodes are not propagated back to the driver
  - Normally, we could get the entire RDD back with reduce(),
    - But sometimes, we just need a simpler way
- Examples: counters

Example 6-2. Accumulator empty line count in Python

```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def extractCallSigns(line):
    global blankLines   # Make the global variable accessible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
```

Created with sparkContext.accumulator(initValue)

Add to the value with += inside tasks

!!! Access with .value, but only on master
Exception if you try it on workers

OUTLINE

- Intro
- Standalone-view RDD concepts
  - Basic RDD
  - Key-Pair RDD
- Distributed-view RDD concepts
  - Lazy evaluation
  - Cache
  - Shuffle/Partition
  - Broadcast
  - Accumulator
- Conclusion

} Operators

} Scheduler

} Block manager

} Networking

} Broadcast

} Accumulator

CONCLUSION

- Spark core
  - basic Spark RDDs
  - Things to take care when programming in clusters
- What's next?
  - Spark SQL/Hive

REFERENCES

- Udemy hands-one course
  - https://www.udemy.com/taming-big-data-with-apache-spark-hands-on/learn/v4/overview
- Several slides:
  - http://heather.miller.am/teaching/cs212/slides
  - http://ampcamp.berkeley.edu/wp-content/uploads/2012/06/matei-zaharia-amp-camp-2012-advanced-spark.pdf
  - https://databricks-training.s3.amazonaws.com/slides/advanced-spark-training.pdf
- <<Learning Spark>>