accenture
Latvia ATC

DATA ENGINEERING BOOTCAMP

Spark - Performance

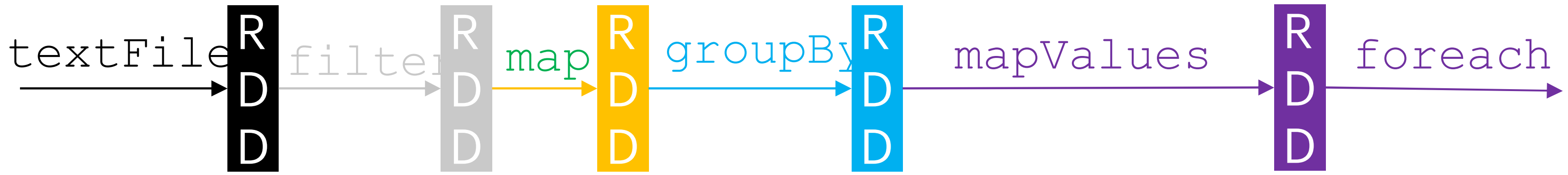
PERSISTENCE (CACHING)

```
rdd = sc.textFile(???).filter(???).map(???).groupBy(???)

rdd.mapValues(???).foreach(???)

println(rdd.filter(???).count())
```

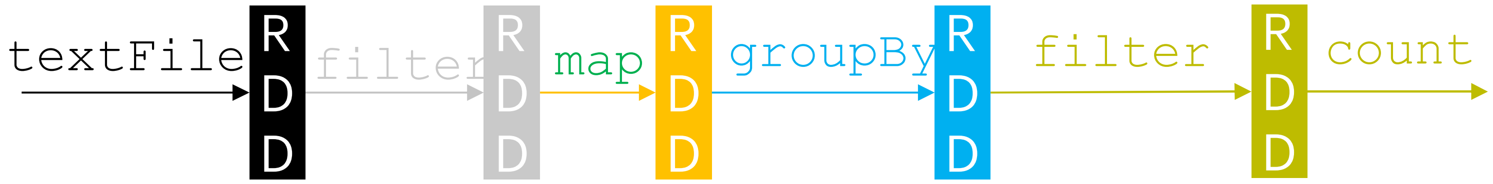
PERSISTENCE (CACHING)

```
rdd = sc.textFile(???).filter(???).map(???).groupBy(???)
```

```
rdd.mapValues(???).foreach(???)
```




```
println(rdd.filter(???).count())
```



PERSISTENCE (CACHING)

Caching or **persistence** are optimization techniques for (iterative and interactive) Spark computations.

Main point to remember:

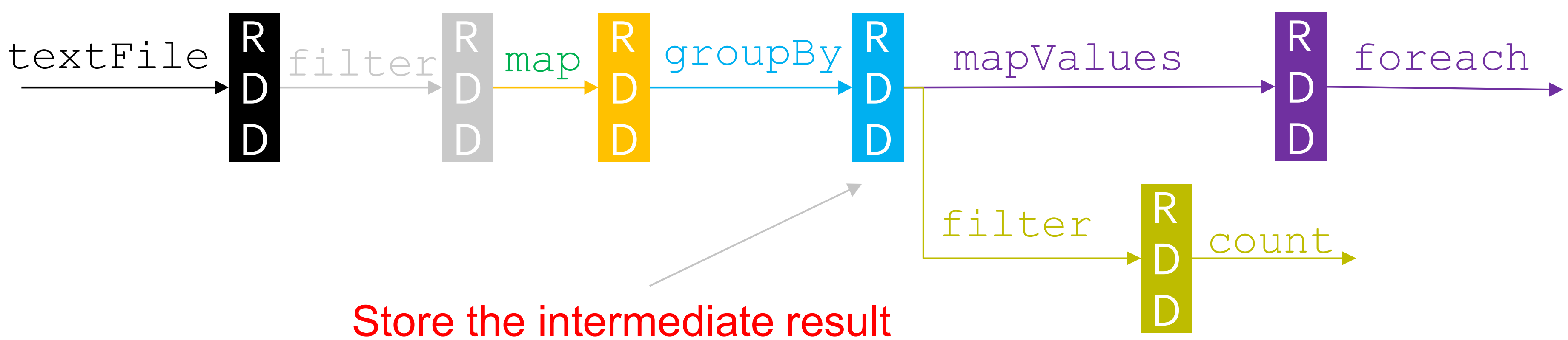
Transformations does not trigger calculations (only actions do).

That means:

Transformation can be executed multiple times!

PERSISTENCE (CACHING)

```
rdd = sc.textFile(???).filter(???).map(???)
            .groupBy(???).cache()  // .persist(<type>)
rdd.mapValues(???).foreach(???)
println(rdd.filter(???).count())
rdd.unpersist()                 // release persistence
```

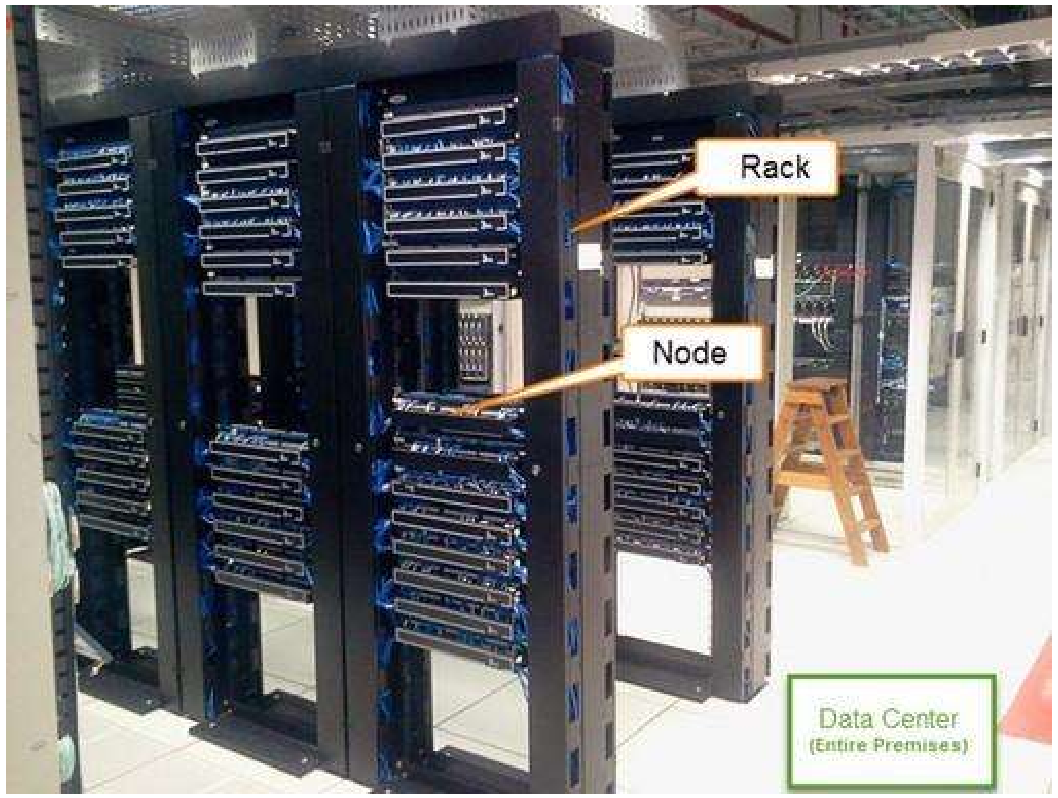


PERSISTENCE TYPES

- MEMORY_ONLY
- MEMORY_AND_DISK
- DISK_ONLY
- …

- `.cache()` same as `.persist(MEMORY_ONLY)`

CLUSTER TOPOLOGY



PARTITIONING

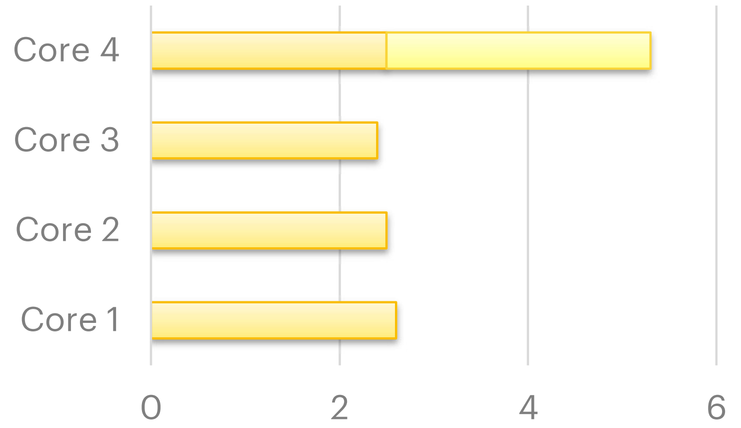
- Partition **cannot** span multiple nodes
- Single concurrent task for every partition
  - More partitions – more parallelism
- By default number of partitions is number of cores
- Number of partitions is configurable
- By default data is read from nodes that are close

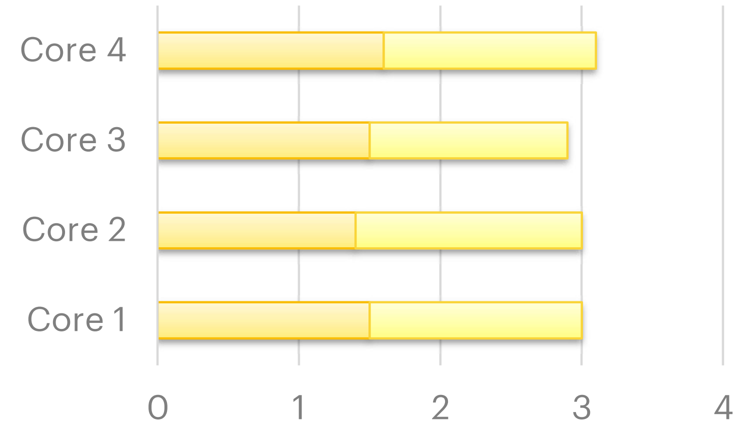
WHY PARTITIONING?

- Increase parallelism
- More equally spread the data
  - e.g. after `filter` operations
- To reduce network traffic
  - More about that later

PARTITIONING: OPTIMAL NUMBER OF PARTITIONS

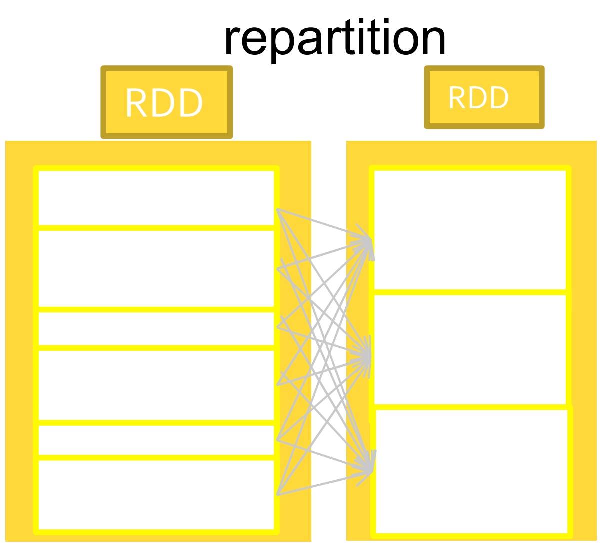
- N * c
  - Where c is number of cores in whole cluster
  - N – natural number (No fanaticism!!!)
- This way data will be equally spread for processing

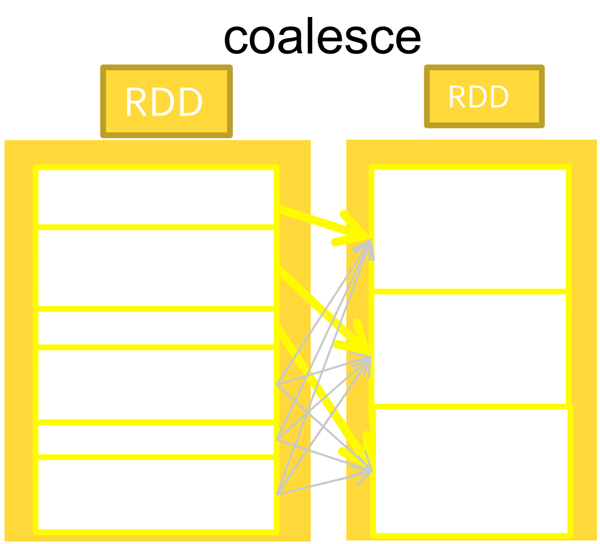




HOW TO SET NUMBER OF PARTITIONS

- `repartition(numPartitions: Int)`
- `coalesce(numPartitions: Int, shuffle: Boolean = false, …)`





HOW TO SET NUMBER OF PARTITIONS

- `repartition(`**`numPartitions`**`: int)`
- `coalesce(`**`numPartitions`**`: int, shuffle: Boolean = false, …)`
- Optional parameter to transformations:
  - `textFile(path: String, `**`minPartitions`**`: int)`
  - `distinct(`**`numPartitions`**`: int)`
  - `groupBy(f: Callable[[T], K], `**`numPartitions`**`: int)`
  - `join(other: RDD[Tuple[K, U]], `**`numPartitions`**`: int)`

TYPES OF CUSTOM PARTITIONING

- Hash Partitioning
  - `key.hashCode() % numPartitions`
  - Does not ensure equal spread of data

- Range Partitioning
  - For keys with particular ordering
  - Split keys by ranges depending on number of partitions

HOW TO SET PARTITIONING

- `partitionBy` method on an RDD
- Specific transformations. Examples:
  - `(reduce|fold|combine|group)ByKey` – HashPartitioner
  - `sortByKey` – RangePartitioner
  - `filter, mapValues, flatMapValues` – keep parent RDD’s partitioner
  - `cogroup, join` – HashPartitioner
- NOTE: some transformations (like `map`) resets the partitioner!

PARTITIONER NOTES

```
rdd.partitionBy(new RangePartitioner(8,
rdd)).persist()
```

NB! Always persist after partitioning

```
pairRDD.mapValues // preserved partitioner
```

```
pairRDD.map // lost partitioner
```

PARTITIONS: SPECIAL OPERATIONS

```
RDD.mapPartitions(f: Callable[[Iterable[T]], Iterable[U]], preservesPartitioning: bool =
False)
```

```
RDD.mapPartitionsWithIndex(f: Callable[[int, Iterable[T]], Iterable[U]],
preservesPartitioning: bool = False)
```

```
RDD.foreachPartition(f: Callable[[Iterable[T]], None])
```

```
def foreachPartitionDemo(iterator):
// Open connection to storage system (e.g. a database connection)
    for x in iterator:
         foreach (x)
        // Use connection to push item to system
// Close connection
def foreachDemo(x): print(x)
sc.parallelize([1, 2, 3, 4, 5]).foreachPartition(foreachPartitionDemo)
```

USEFUL FUNCTIONS

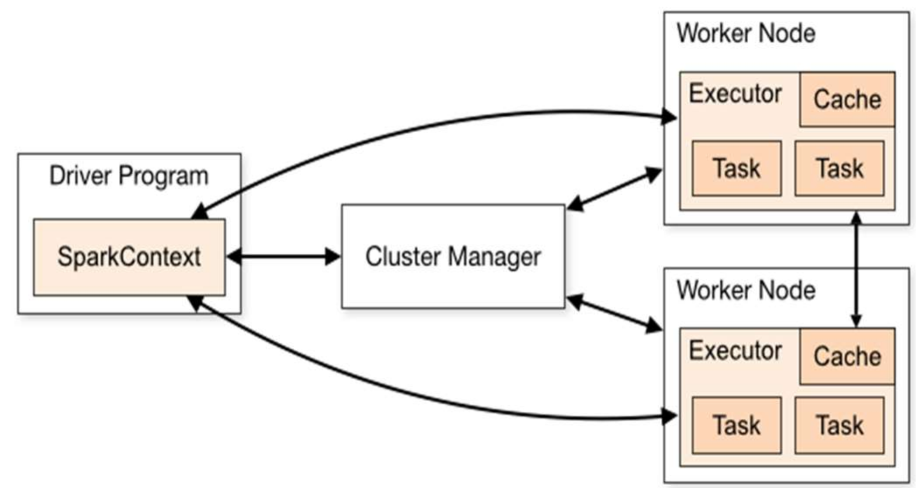
```
// Check number of partitions

rdd.getNumPartitions

// Get array of partitions

rdd.partitions

// Get current partitioner

rdd.partitioner
```

TAKEAWAYS (PARTITIONING AND PERSISTENCE)

- Number of partitions defines parallelism
- Number of partitions can be set programmatically
- RDD can be persisted (cached) for re-use without re-calculation
  - Especially useful after “heavy” transformations
- By default Spark decides how to split data, but
- Some transformation apply default partitioners and
- Custom partitioners can be created and applied to RDDs
- Transformation which may change keys (like `map`) will reset partitioner

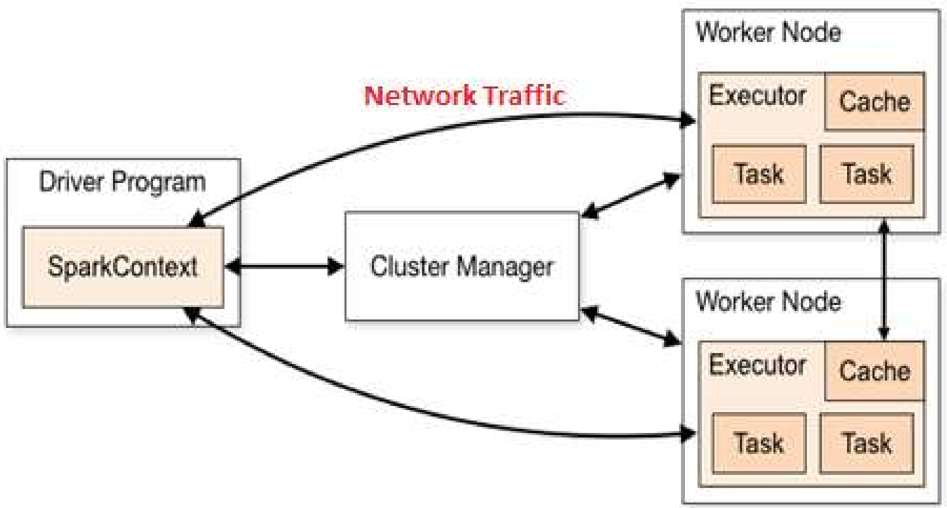
SPARK APPLICATION EXECUTION STEPS

1. `spark-submit` launches the **driver program** and invokes the `main()` method
2. The **driver program** contacts the **cluster manager** to launch **executors.**
3. The **cluster manager** launches **executors** on behalf of the driver program.
4. Based on the RDD **actions** and **transformations**, the **driver** sends **work to executors** in the **form of tasks.**
5. **Tasks** are run on executor processes to compute and save results.
6. If the driver's `main()` method exits or it calls `SparkContext.stop()`, it will terminate the executors.



SPARK APPLICATION ARCHITECTURE

```
conf = ...

sc = ...


textFile = sc.textFile("hdfs://...")

counts = textFile

.flatMap(lambda x: x.split(" "))

.map(lambda word: (word, 1))

.reduceByKey(lambda x, y: x + y)

.collect()
```



SHUFFLING

Data:

(28,Kate)

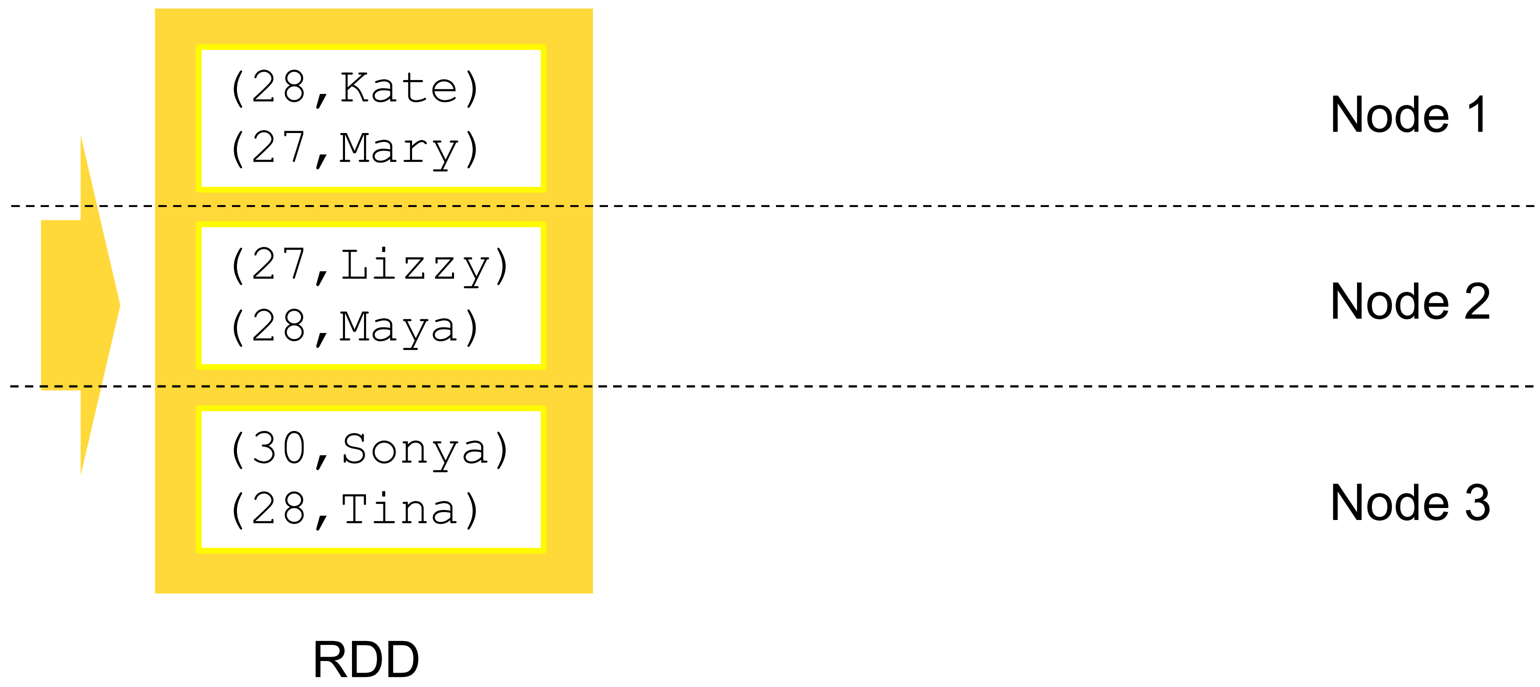
(27,Mary)

(27,Lizzy)

(28,Maya)

(30,Sonya)

(28,Tina)



Create RDD

SHUFFLING

Data:

(28,Kate)

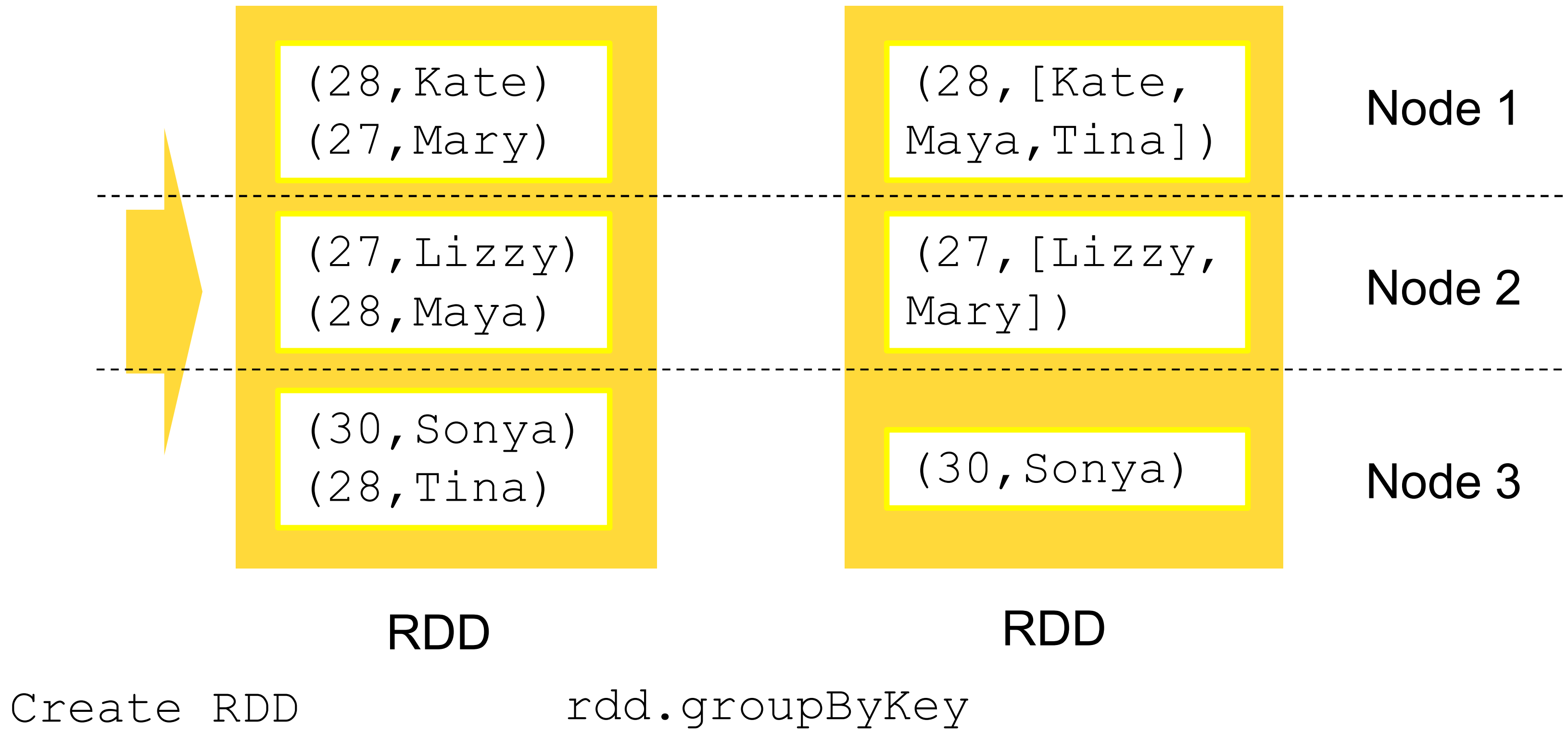
(27,Mary)

(27,Lizzy)

(28,Maya)

(30,Sonya)

(28,Tina)



SHUFFLING

Data:

(28,Kate)

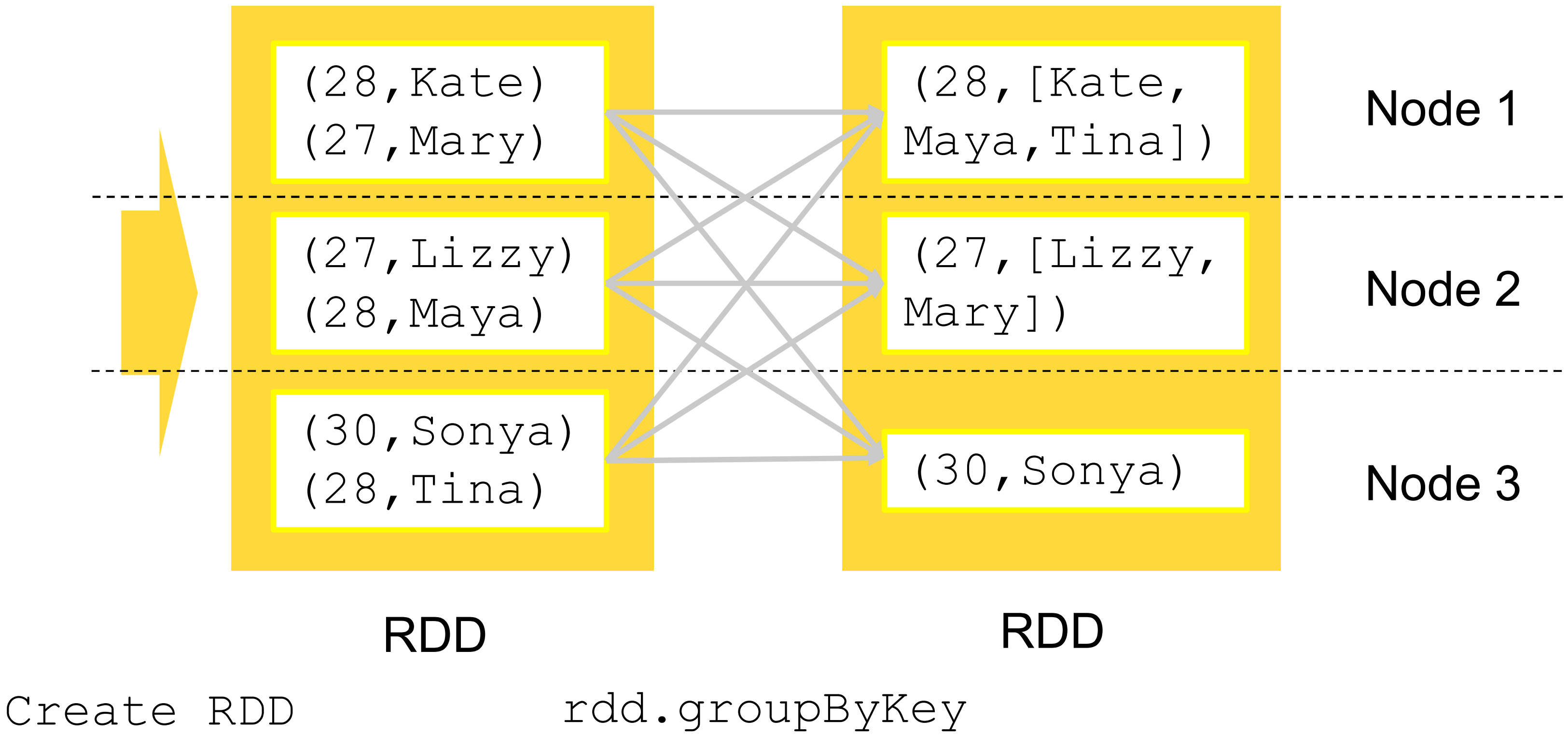
(27,Mary)

(27,Lizzy)

(28,Maya)

(30,Sonya)

(28,Tina)



WHAT CAUSES SHUFFLE?

- `groupBy`
- …`byKey`
  - Except `countByKey`
- `join`
- …
- RULE: When elements in RDD depends on elements in other RDD or other elements in same RDD

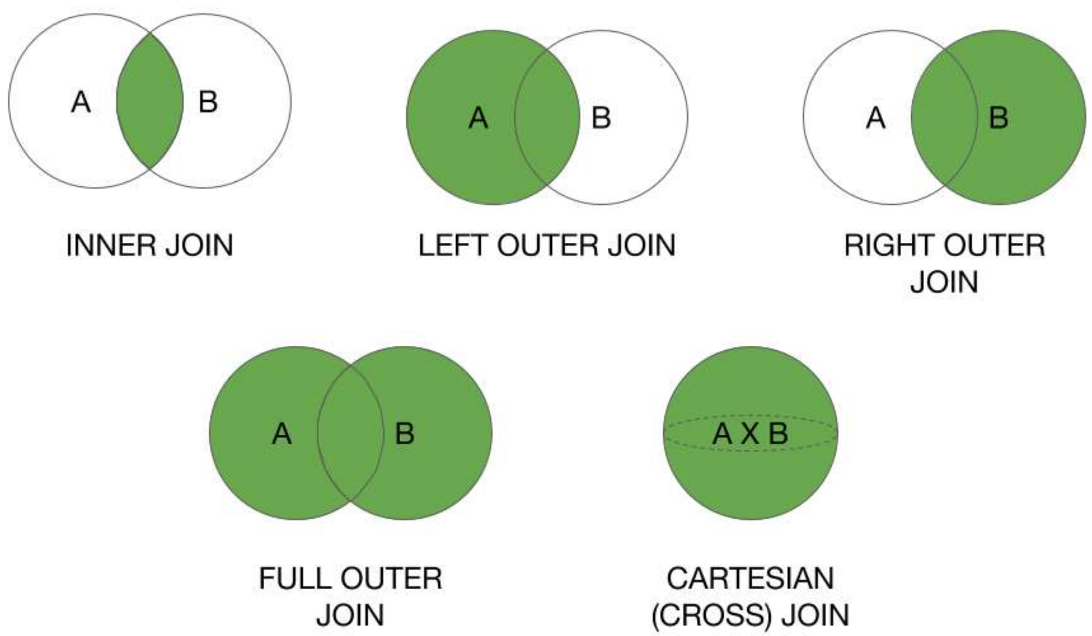
HOW TO MINIMIZE SHUFFLE IMPACT

Minimize data to be shuffled

- Filter unneeded data as early as possible
    - `personRDD.filter(lambda p: p.age > 21).groupBy(???)`
- Use map to discard redundant data
    - `personRDD.map(lambda p: (p.age, p.salary)).groupByKey`

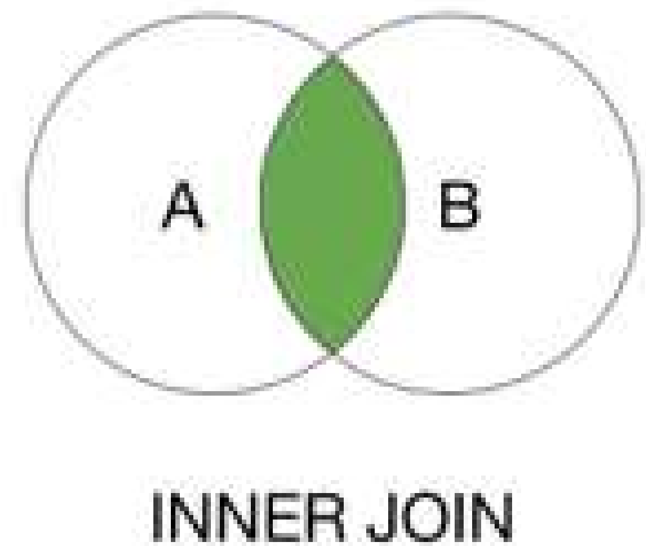
HOW TO MINIMIZE SHUFFLE IMPACT: JOINS

What does joins do?

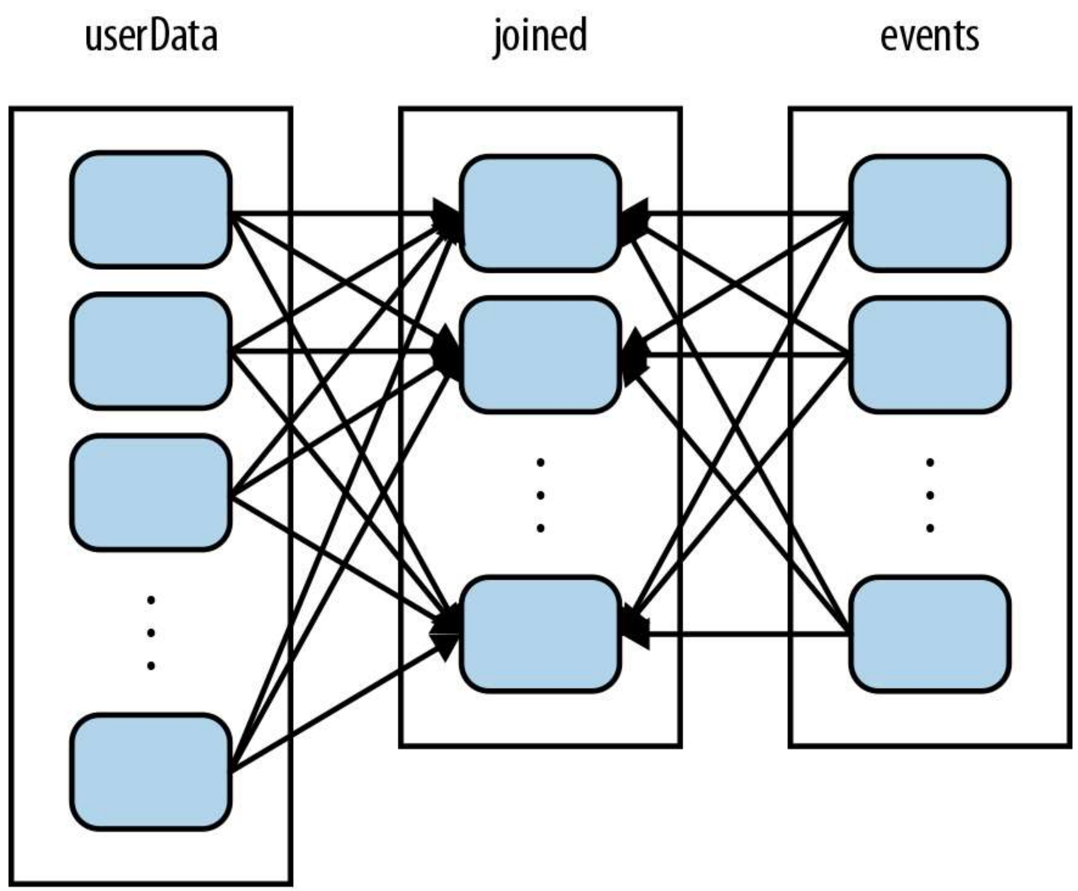


JOIN EXAMPLE USE CASE

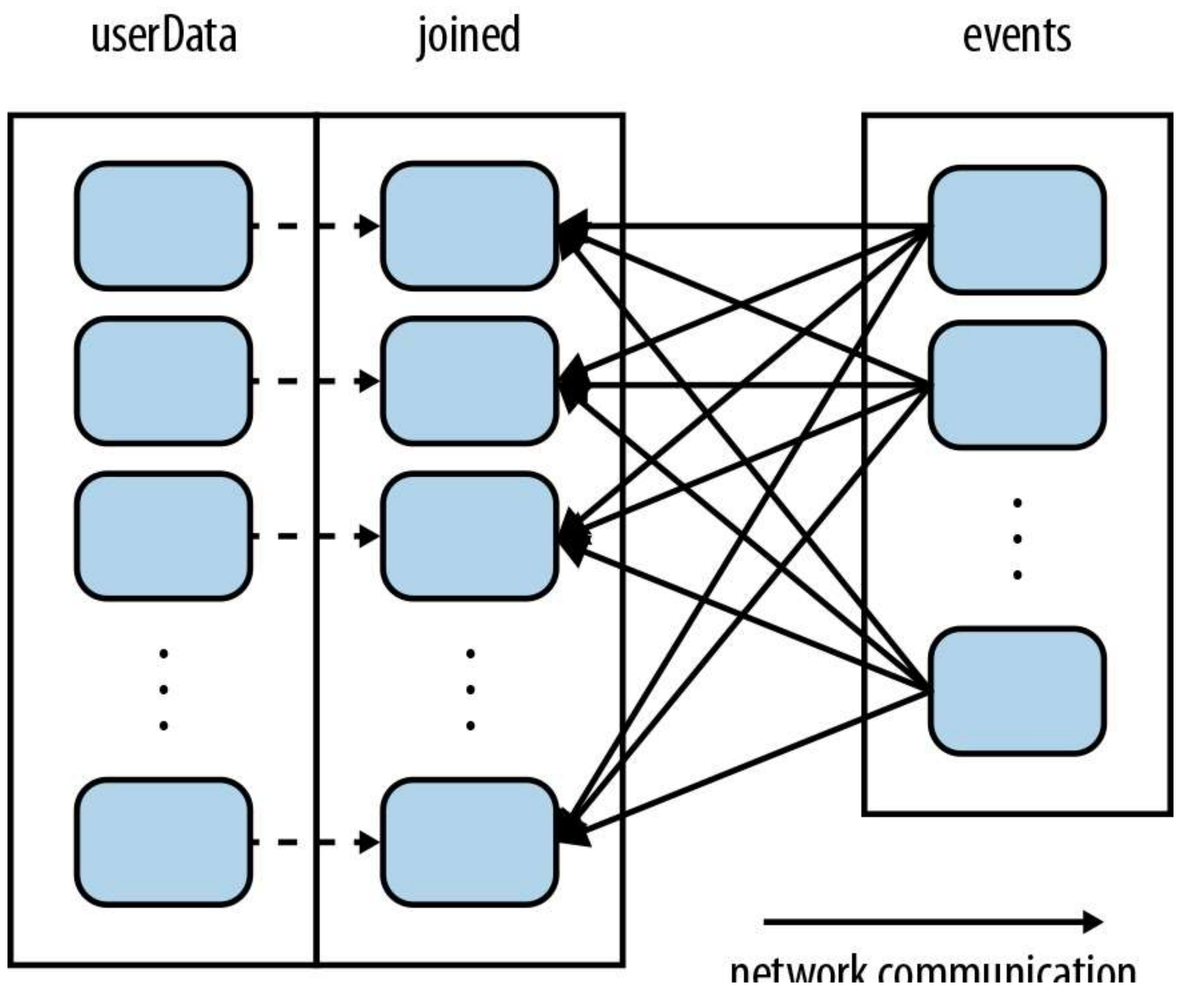
- One **large** dataset – userData (A)
  - changes once per day
- **Small** datasets – events (B)
  - Comes every minute
- Question:
  - For each event get user information for analysis



JOIN EXAMPLE (BAD CASE)



JOIN EXAMPLE (GOOD CASE)



HOW TO MINIMIZE SHUFFLE IMPACT: JOINS

Careful with joins

- Already partitioned data will not be shuffled again
  - `rdd.groupBy(…).join(rdd2)// only rdd2 is shuffled`

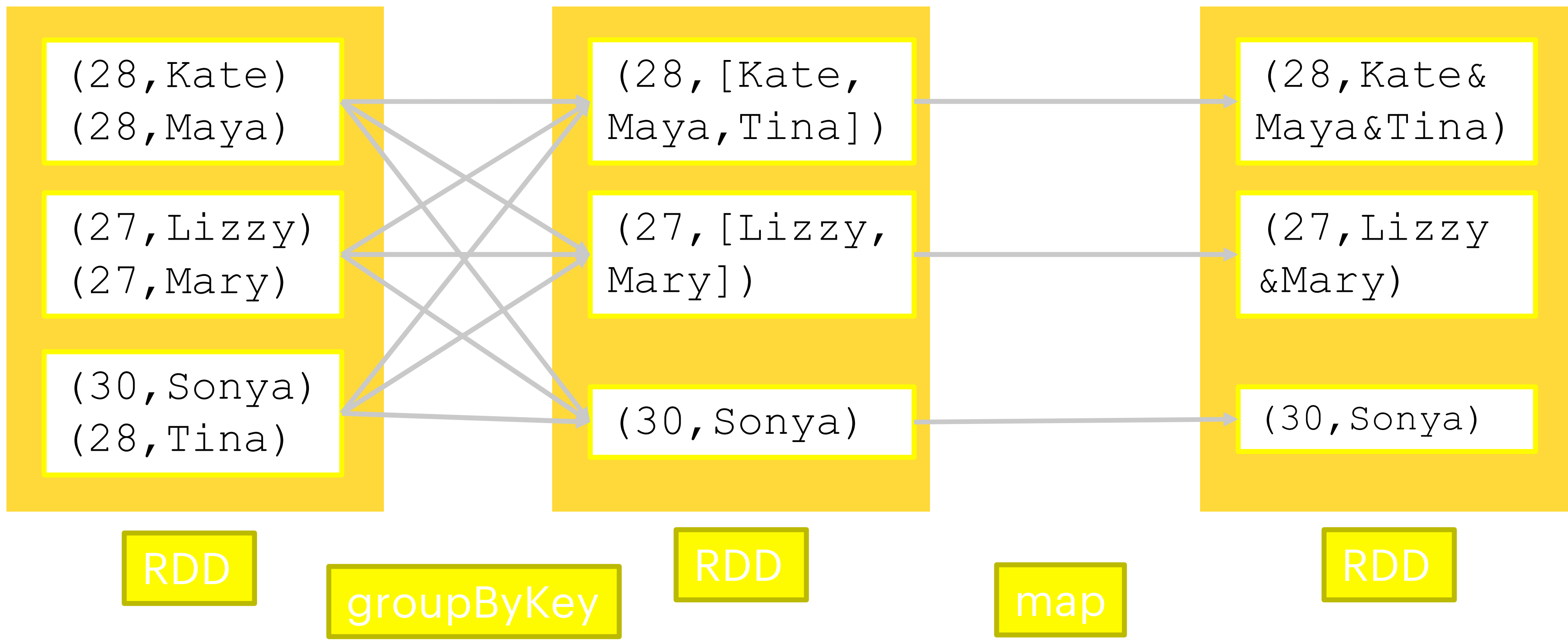
- Use same **partitioner** (co-grouped joins does not cause shuffle)

HOW TO MINIMIZE SHUFFLE IMPACT: API

Use reduction API when applicable

HOW TO MINIMIZE SHUFFLE IMPACT: API

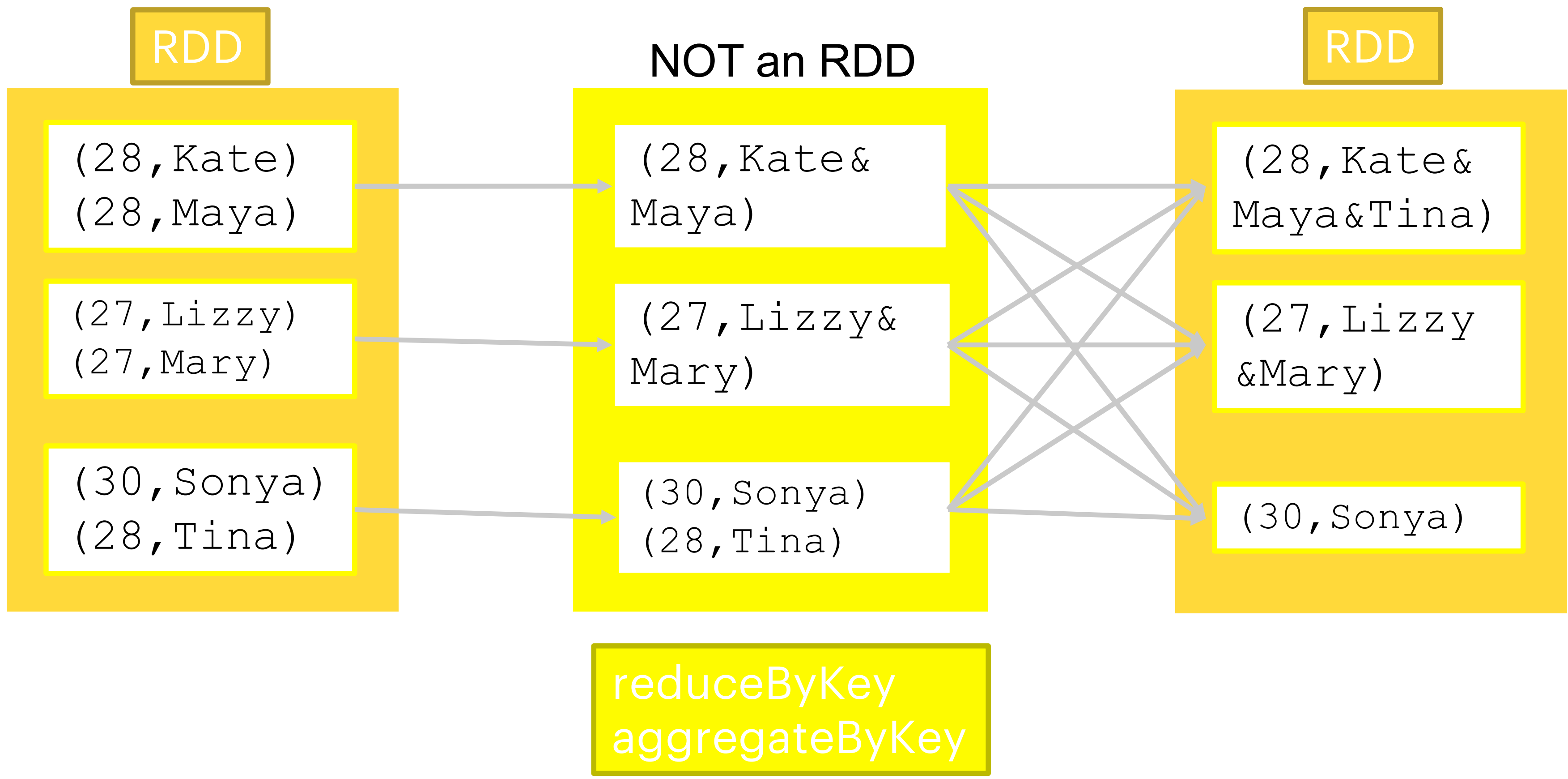
Use reduction API when applicable



HOW TO MINIMIZE SHUFFLE IMPACT: API

Use reduction API when applicable

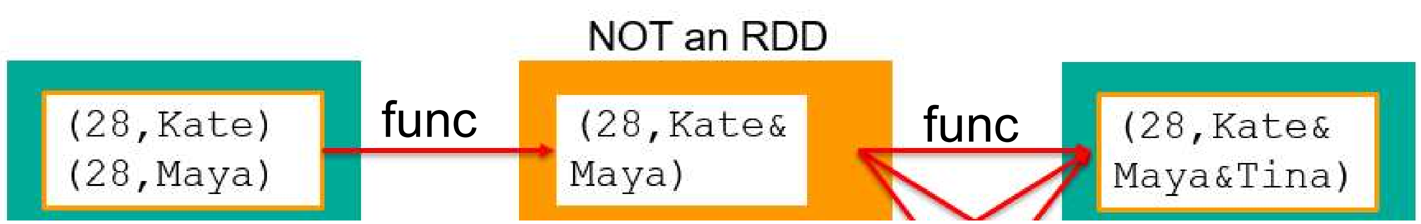
- `reduceByKey` or `aggregateByKey` is a better solution



REDUCBYKEY VS AGGREGATEBYKEY

- `reduceByKey(func: (V, V) => V) => RDD[(K, V)]`
  - **Value type change is impossible**

```
rdd.reduceByKey(lambda v1, v2: v1 + "&" + v2)
```

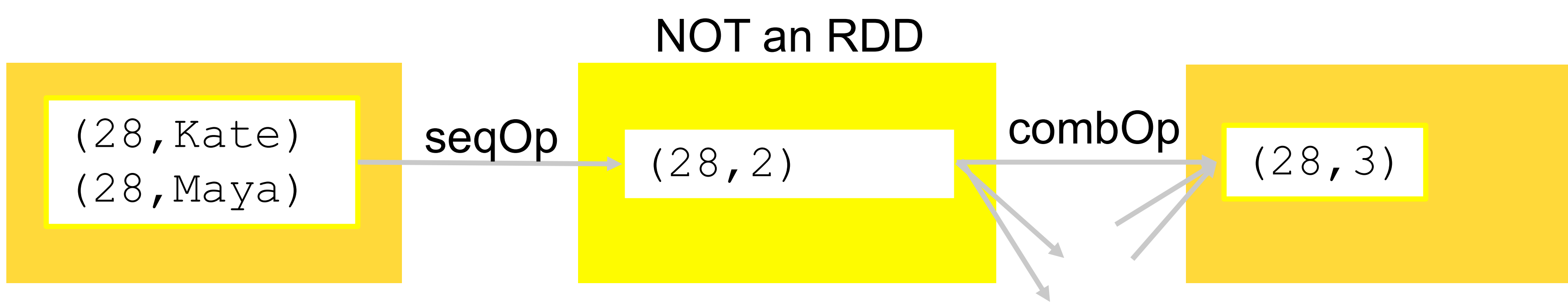



```
(Kate, Maya)        (Kate&Maya, Tina)
     =>                     =>
 Kate&Maya           Kate&Maya&Tina
```

REDUCBYKEY VS AGGREGATEBYKEY

- `aggregateByKey(zero:U)(seqOp:(U,V) => U, combOp:(U,U)=> U) => RDD[(K, U)]`
    - **Type of value can be changed**

```
rdd.aggregateByKey(0)((acc,v) => acc + 1, (v1,v2) => v1 + v2)
```




```
(0, Kate) => 0 + 1 = 1          (2, 1) => 2 + 1 = 3
(1, Maya) => 1 + 1 = 2
```

USEFUL FUNCTIONS

```
/** A description of this RDD and its recursive
dependencies for debugging. */

rdd.toDebugString
```

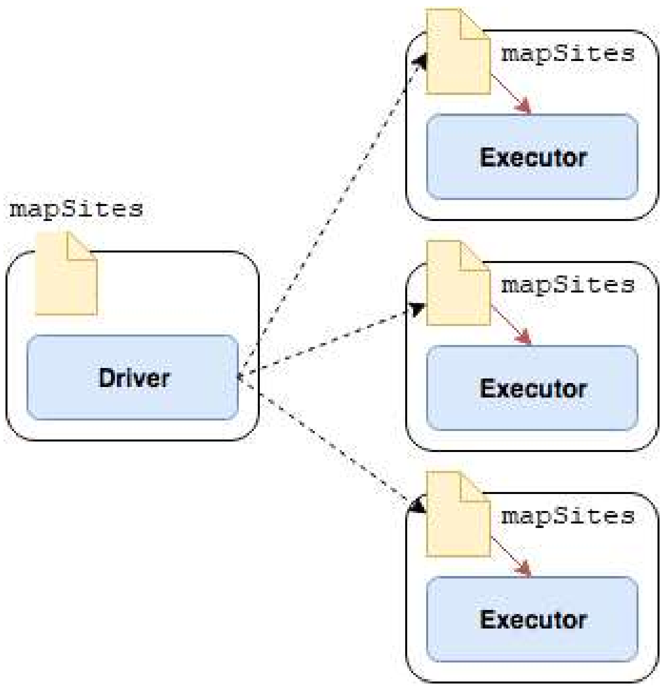
```
(8) MapPartitionsRDD[11] at join at <console>:23 [] |
MapPartitionsRDD[10] at join at <console>:23 [] |
CoGroupedRDD[9] at join at <console>:23 []

+-(8) ParallelCollectionRDD[8] at parallelize at
<console>:21 []

+-(8) ParallelCollectionRDD[8] at parallelize at
<console>:21 []
```

TAKEAWAYS (SHUFFLING)

- Shuffle can occur, when elements in RDD depends on:
  - Elements in another RDD
  - Other elements in same RDD
- Shuffle can dramatically impact performance
- Shuffle can be controlled (optimized) by:
  - Caching and persistence
  - Partitioning
  - Reduction API
  - Reduction of shuffled data size

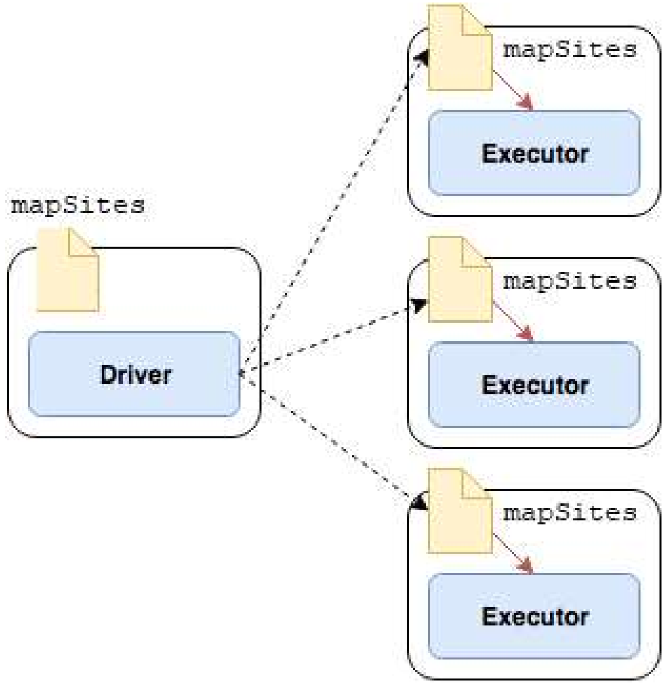
BROADCAST VARIABLES

```
# Define the mapSites dictionary
mapSites = ???

# Use the map transformation
mapped_rdd = rdd.map(lambda v: mapSites[v])
```



BROADCAST VARIABLES

```
# Define the mapSites dictionary
mapSites = ???

# Use the map transformation
mapped_rdd = rdd.map(lambda v: mapSites[v])

# Use the filter transformation with contains
filtered_rdd1 = rdd1.filter(lambda v: v in
mapSites)
```



x2 times

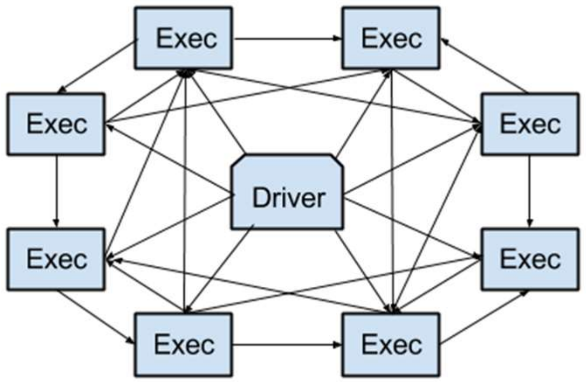
BROADCAST VARIABLES

- Read-only
  - Once broadcasted values can’t be changed

- Efficient distribution for large variables
  - BitTorrent-like data distribution

- When same data is used multiple times

SHARED VARIABLES




```
# Define the mapSites dictionary
mapSites = ???

# Broadcast the mapSites dictionary
broadcastMapSites = sc.broadcast(mapSites)

# Use the map transformation with broadcasted value
mapped_rdd = rdd.map(lambda v: broadcastMapSites.value[v])

# Use the filter transformation with broadcasted value
filtered_rdd1 = rdd1.filter(lambda v: v in broadcastMapSites.value)

broadcastMapSites.unpersist()
```

ACCUMULATOR VARIABLES

```
// How to count filtered lines
sc.textFile(???).filter(???).map(???)
```

How to know how much lines were filtered out?

How to know which lines were filtered out?

ACCUMULATOR VARIABLES

- Can be only “added” to
- Nodes can’t read the value
  - Only driver can
- Can be either *named* or *unnamed*
- Can be custom
  - Subclass AccumulatorParam (Spark 1.6)
  - Subclass AccumulatorV2 (Spark 2.0)
- NB! In transformations - might add multiple times

ACCUMULATOR VARIABLES

```
// In Spark 1.6
acc = sc.accumulator(0)

sc.textFile(???).filter { v =>
    if (???) {
      acc += 1
      false
    }
    true
}.map(???)

acc.value
```

```
// In Spark 2.0
acc = sc.longAccumulator
```

TAKEAWAYS

- Optimize data processing with:
  - Persistence
  - Partitioning
  - Avoiding or lessen impact from shuffles
  - Broadcast reusable or large data
  - Use accumulators to avoid re-calculations