Partitioning

Sample Problem

Customer ID	Product ID	Item Cost
129482221	A	10.99
129482221	B	4.99
129482221	A	9.99
583910109	C	0.99
583910109	D	13.99
873803751	D	13.99
873803751	B	5.99
873803751	A	11.99

Product ID	Customer Count
A	2
B	2
C	1
D	2

Preparing our RDD

```
data = ((129482221,'A',10.99),
        (129482221,'B',4.99),
        (129482221,'A',9.99),
        (583910109,'C',0.99),
        (583910109,'D',13.99),
        (873803751,'D',13.99),
        (873803751,'B',5.99),
        (873803751,'A',11.99))
rdd = sc.parallelize(data)
```

How `parallelize()` stores our data?

- It distributes data across all available worker nodes
  - By default, Spark uses all machine (logical) cores as workers
  - e.g. on a 6-core/12-thread laptop, there are 12 workers
- By default, # of partitions = # of workers:
  - 12 partitions for 8 records, there should be a few empty ones

```
rdd.getNumPartitions()
```

```
12
```

```
rdd.glom().collect()
```

```
[[],
 [(129482221, 'A', 10.99)],
 [(129482221, 'B', 4.99)],
 [],
 [(129482221, 'A', 9.99)],
 [(583910109, 'C', 0.99)],
 [],
 [(583910109, 'D', 13.99)],
 [(873803751, 'D', 13.99)],
 [],
 [(873803751, 'B', 5.99)],
 [(873803751, 'A', 11.99)]]
```

Prepare our RDD using only 3 workers

```
data = ((129482221,'A',10.99),
        (129482221,'B',4.99),
        (129482221,'A',9.99),
        (583910109,'C',0.99),
        (583910109,'D',13.99),
        (873803751,'D',13.99),
        (873803751,'B',5.99),
        (873803751,'A',11.99))
rdd = sc.parallelize(data, numSlices=3)
```

```
rdd.getNumPartitions()
```

```
3
```

```
rdd.glom().collect()
```

```
[[(129482221, 'A', 10.99), (129482221, 'B', 4.99)],
 [(129482221, 'A', 9.99), (583910109, 'C', 0.99)],
 [(583910109, 'D', 13.99),
  (873803751, 'D', 13.99),
  (873803751, 'B', 5.99),
  (873803751, 'A', 11.99)]]
```

Approach 1: Grouping then Count

```
rdd.map(lambda x: (x[1],x[0])) \
    .groupByKey() \
    .mapValues(lambda x: len(set(x))) \
    .collect()
```

```
[('B', 2), ('C', 1), ('D', 2), ('A', 2)]
```

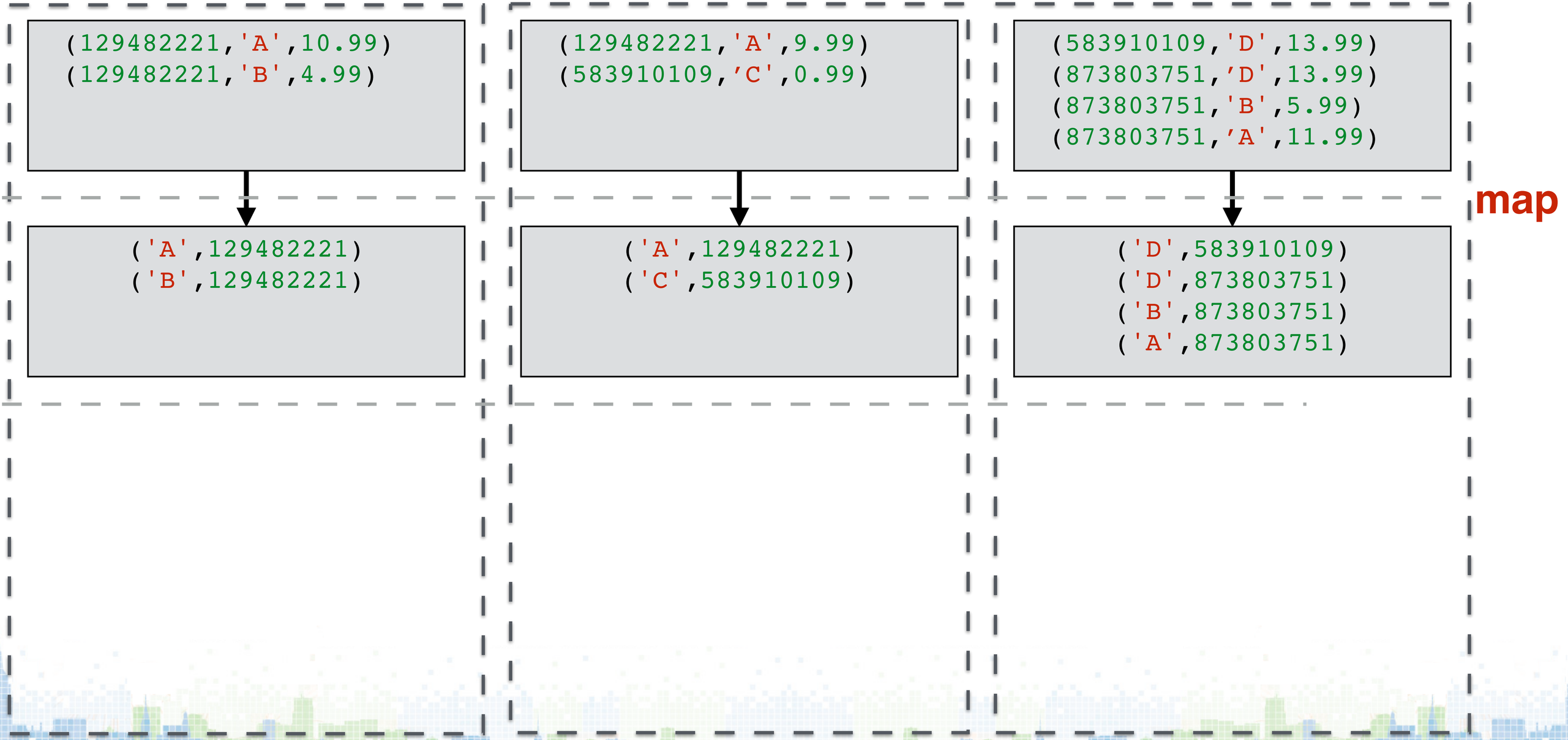
What might have happened

```
(129482221,'A',10.99)
(129482221,'B',4.99)
```

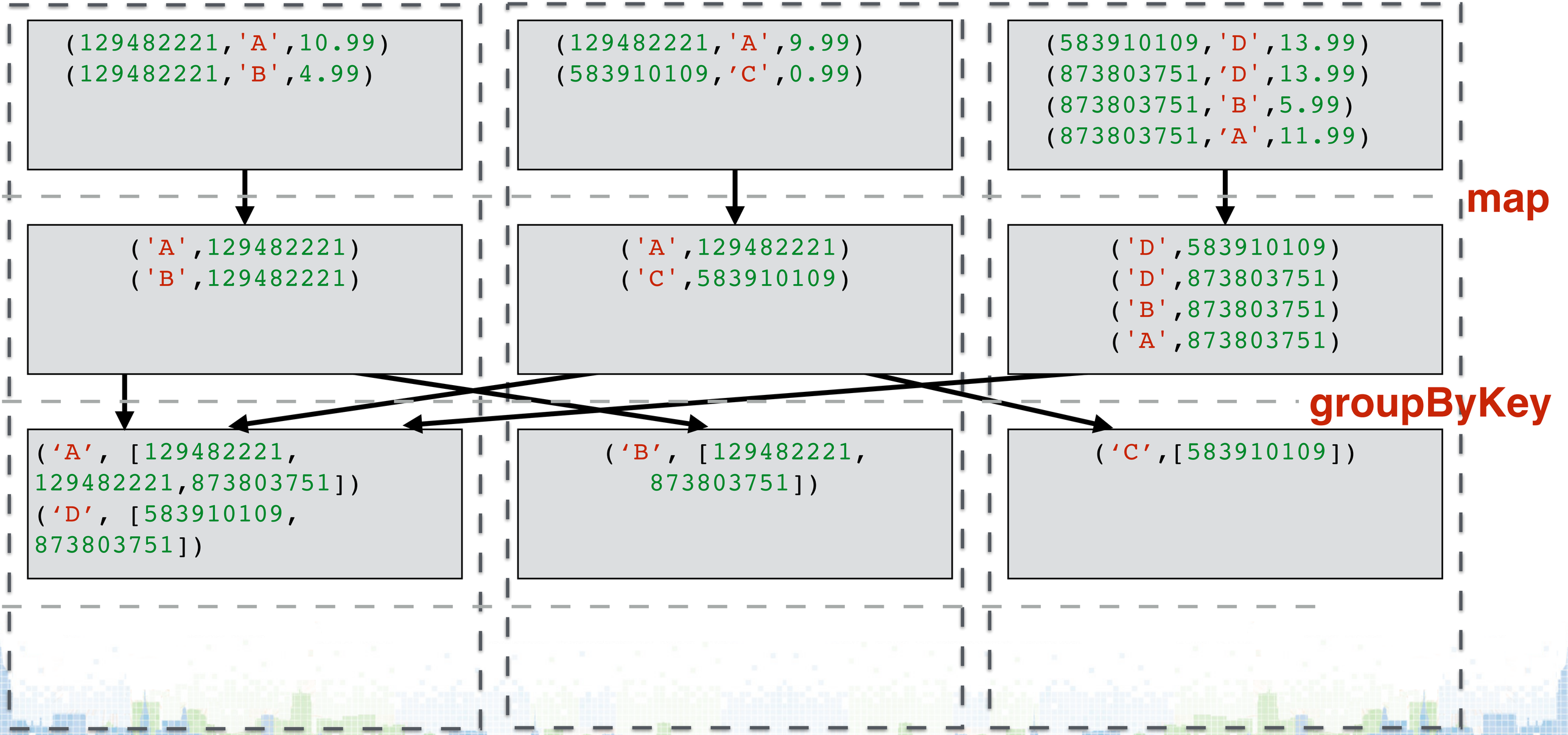
```
(129482221,'A',9.99)
(583910109,’C',0.99)
```

```
(583910109,'D',13.99)
(873803751,’D',13.99)
(873803751,'B',5.99)
(873803751,’A',11.99)
```

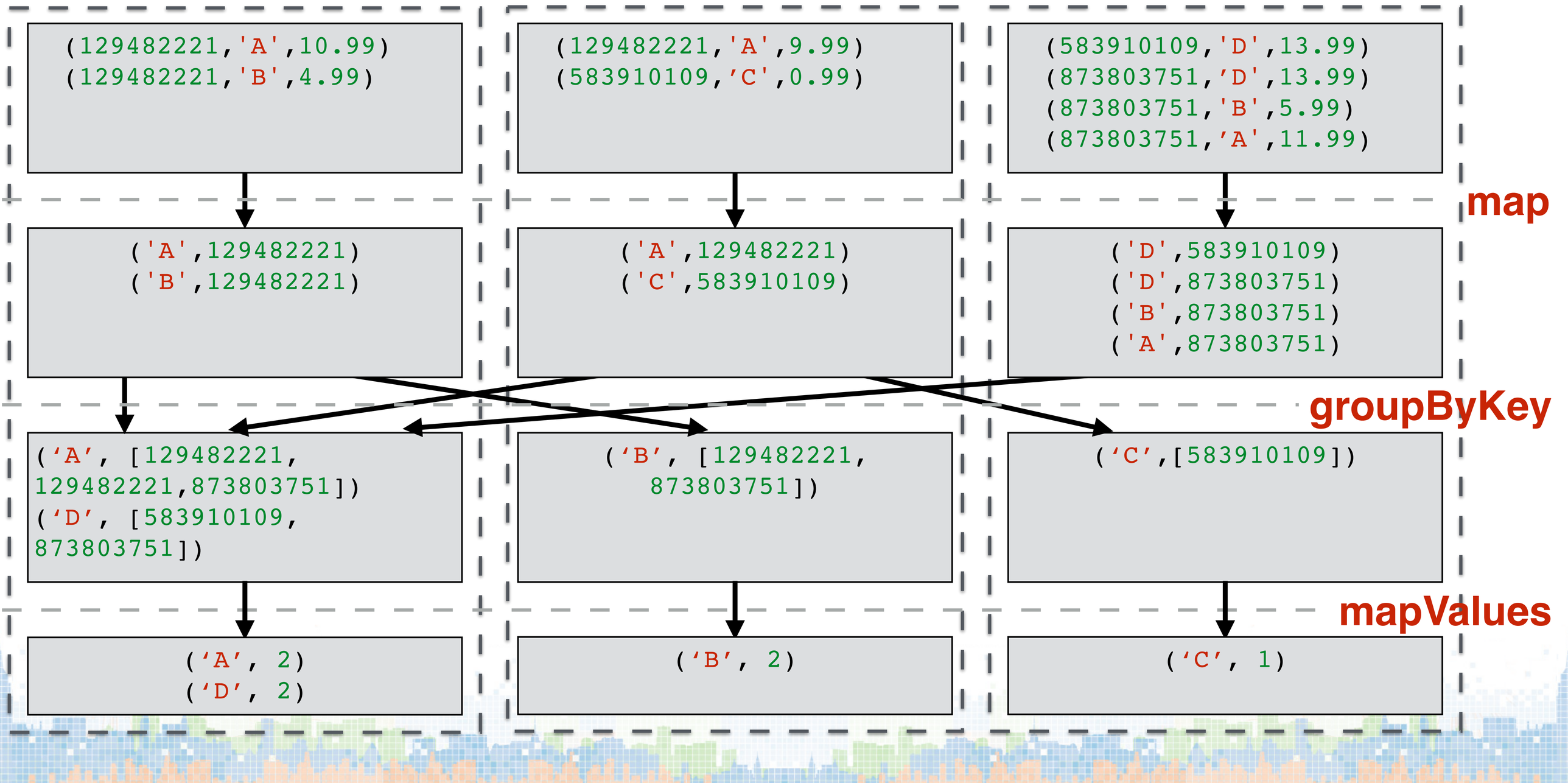
What might have happened



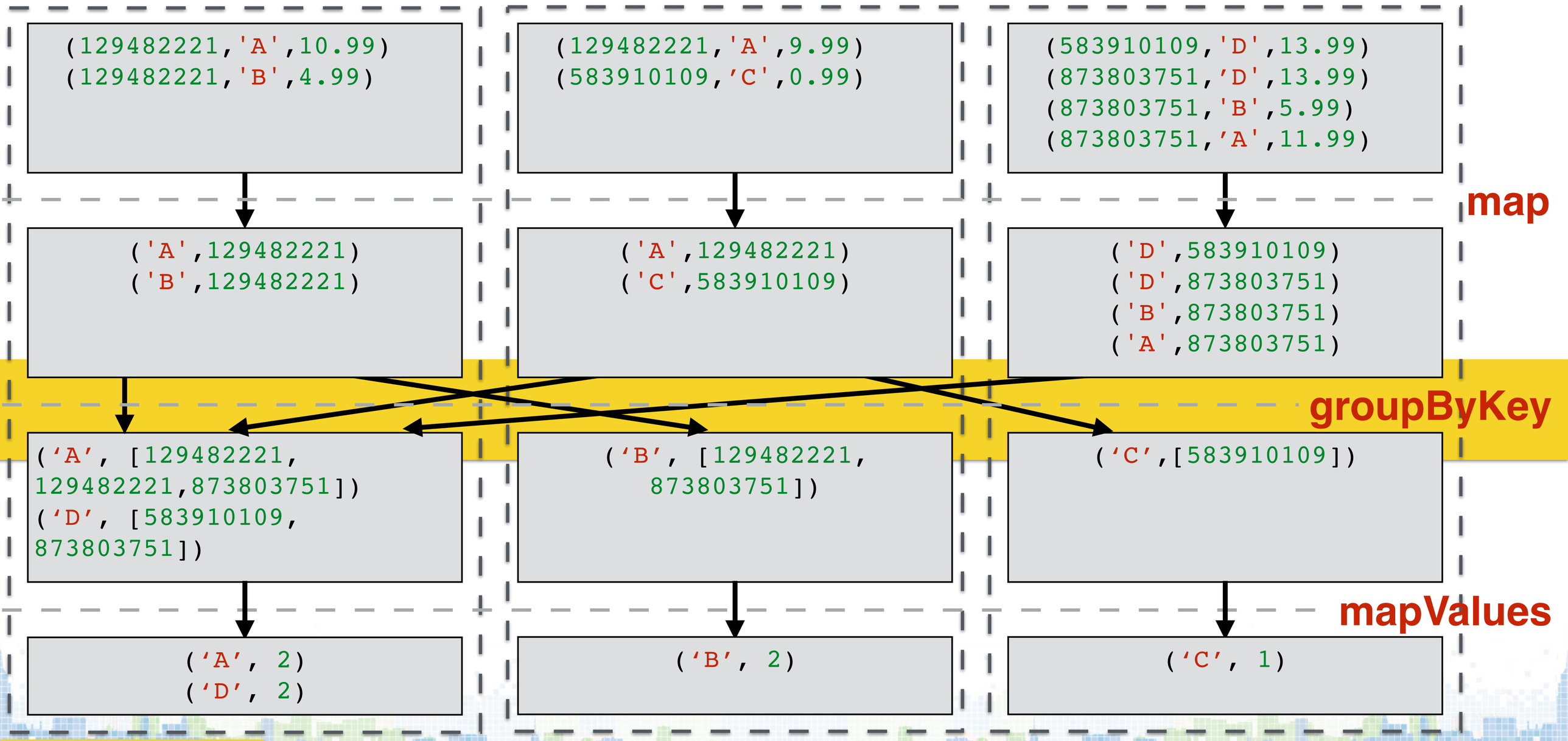
What might have happened



What might have happened



What might have happened



Issues with `groupByKey()`

- Requires collecting of all values for each key
  - A lot of data are sent over the network
- Each key group is a single record (key, [list of values]), and its processing cannot be shared across workers
  - Limit parallelism in processing the key

Approach 2: reduceByKey()

```
rdd.map(lambda x: (x[1],set([x[0]]))) \
    .reduceByKey(lambda x,y: x | y) \
    .map(lambda x: (x[0], len(x[1]))) \
    .collect()
```

```
[('B', 2), ('C', 1), ('D', 2), ('A', 2)]
```

What might have happened

```
(129482221,'A',10.99)
(129482221,'B',4.99)
```

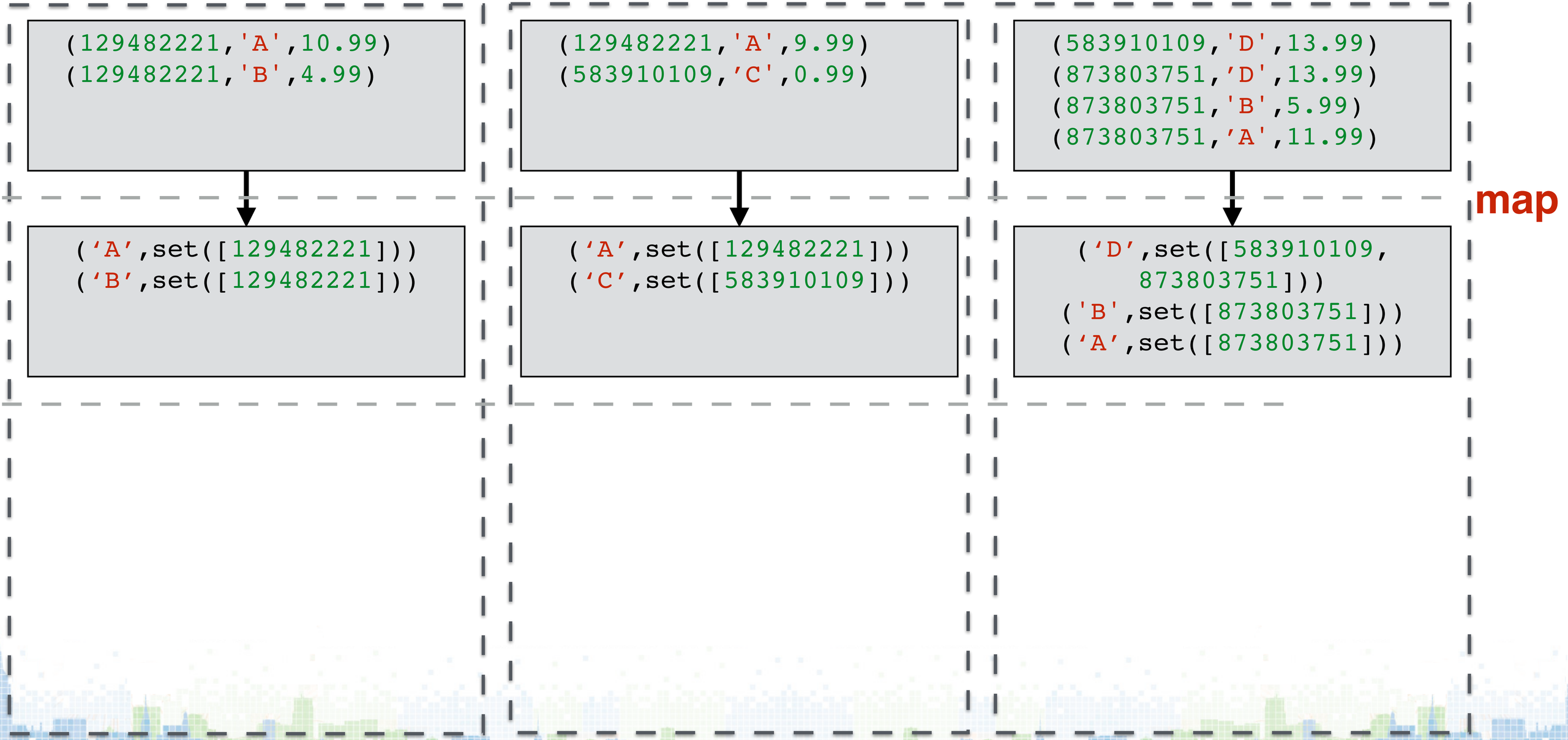
```
(129482221,'A',9.99)
(583910109,’C',0.99)
```

```
(583910109,'D',13.99)
(873803751,’D',13.99)
(873803751,'B',5.99)
(873803751,’A',11.99)
```

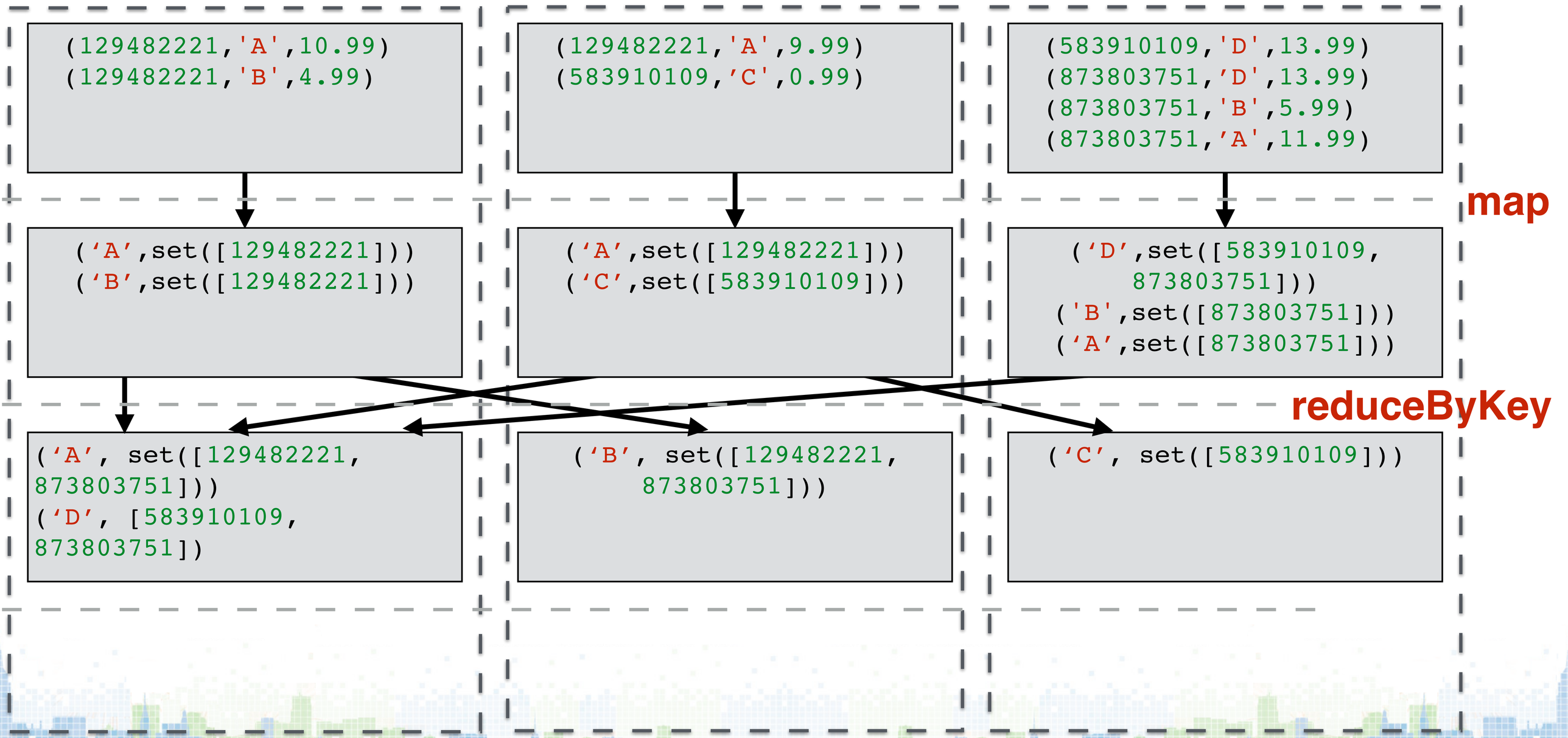
What might have happened

What might have happened

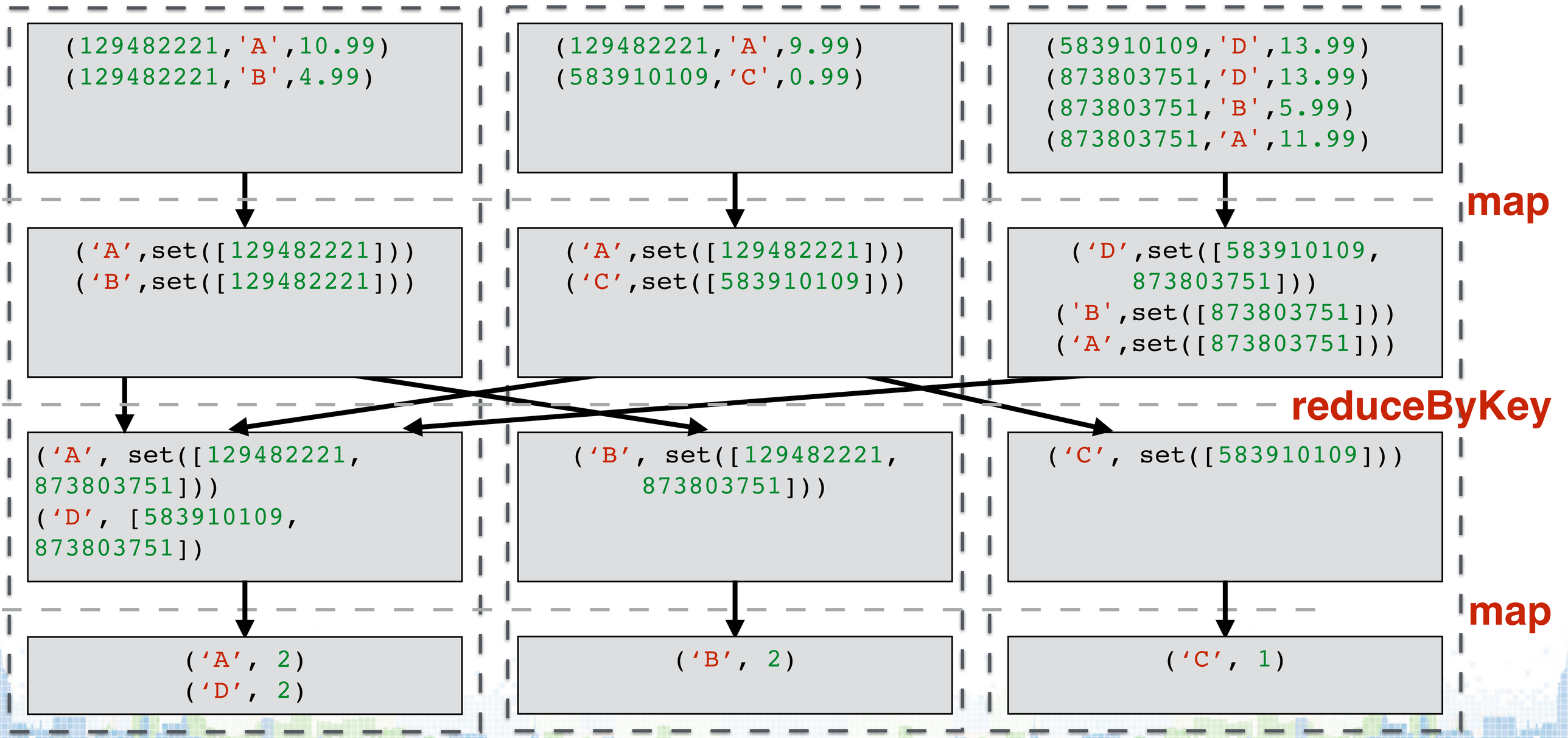
What might have happened



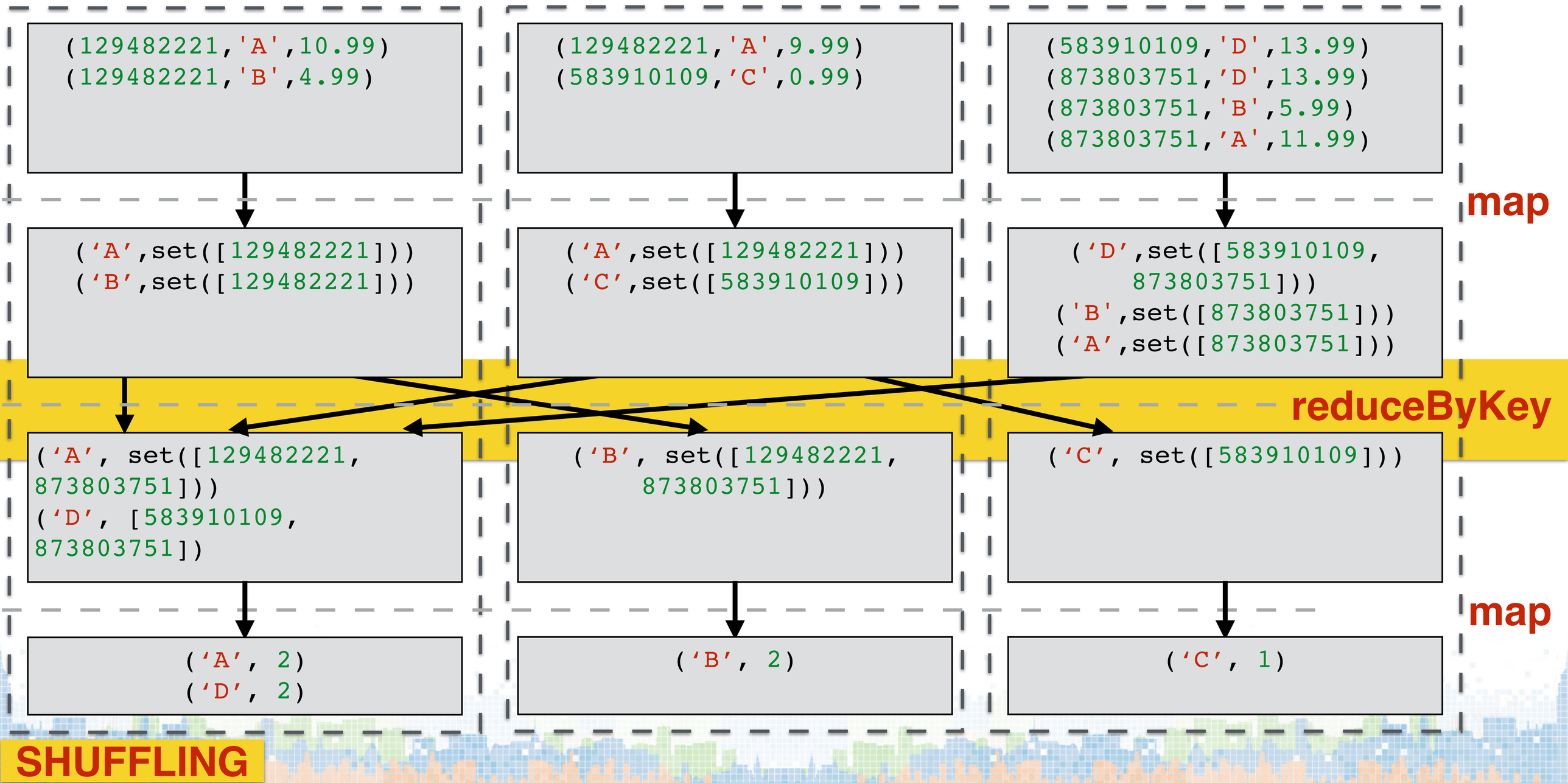
What might have happened



What might have happened



What might have happened



`reduceByKey()` vs. `groupByKey()`

- Reduce the amount of data sent over the network
- reduceByKey() only needs to store aggregated results instead of all elements
- Allow computations to be done in parallel (on the map side)
  - since it's no longer a single key/value pair

Partitioning

- Shuffling is based on partitioning
  - Each partition is responsible for a set of keys
  - All tuples/records with the same key will go to the same partition
- Each partition will be stored wholly on a single node, or each worker contains one or more partition
  - Partitions can be processed as a whole (since it will fit into memory), e.g. using mapPartitions()
- The number of partitions (and how to partition the data) is configurable

Spark’s Partition Schemes

- Hash partitioning
- Range partitioning

Hash Partitioner

- Spread data across partition based on key's hash codes:

```
partitionId = hash(key) % numPartitions
```

- Records with a same key are grouped into the same partition

Hash Partitioner Example

- Consider an RDD with odd keys `[21,15,19,93,51,27,65,33]`, and we would like to split them into 4 partitions
- Given the hash function is identity (the key is also the hash value)
- The resulting partitions using a *Hash Partitioner* would be:

```
partition_0 = []
partition_1 = [21,93,65,33]
partition_2 = []
partition_3 = [15,19,51,27]
```

Range Partitioner

- Spread data across partition based on key's **ordering**, roughly:

```
[(key,record)] = enumerate(sorted(records))

ranges = [a_1,…,a_n]

partitionId = bisect.bisect_left(ranges, key)
```

  - n is the number of partitions

- Records with the same key range are grouped into the same partition

Range Partitioner

- Assumption: the key value range is `[1,100]`
- The 4 ranges we could use are:

```
[1,25],[26,50],[51,75],[76,100]

ranges = [25,50,75,100]
```

- The resulting partitions using a *Range Partitioner* would be:

```
partition_0 = [15,19,21]
partition_1 = [27,33]
partition_2 = [51,65]
partition_3 = [93]
```

Range Partitioner spread data more evenly

- But need to know the data distribution
  - Could be costly to compute for large data set

Specify Partitioners in Spark

- Use RDD `partitionBy(numPartitions, partitionFunc)`
  - `partitionFunc` takes a key and return a partition number (or a hash)
- Use RDD Transformations that operates on key/value pairs and takes `partitionFunc` or `numPartitions` as a parameter. Examples:

```
cogroup, groupWith, join(s)
(aggregate/combine/group/fold/reduce/substract)ByKey
coalesc, partitionBy, repartition
```

When to pay attention to Partitioning

- When a shuffle might occur
  - A good partitioner can help speed up the shuffling process
- A shuffle might occur when the resulting data records depends on other data records (regardless of RDD)