ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Shuffling: What it is and why it's important

Big Data Analysis with Scala and Spark

Heather Miller

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

Think again what happens when you have to do a `groupBy` or a `groupByKey`. Remember our data is distributed! **Did you notice anything odd?**

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

Think again what happens when you have to do a groupBy or a groupByKey. Remember our data is distributed! **Did you notice anything odd?**

```
val pairs = sc.parallelize(List((1, "one"), (2, "two"), (3, "three")))
pairs.groupByKey()
// res2: org.apache.spark.rdd.RDD[(Int, Iterable[String])]
//    = ShuffledRDD[16] at groupByKey at <console>:37
```

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

Think again what happens when you have to do a groupBy or a groupByKey. Remember our data is distributed! **Did you notice anything odd?**

```
val pairs = sc.parallelize(List((1, "one"), (2, "two"), (3, "three")))
pairs.groupByKey()
// res2: org.apache.spark.rdd.RDD[(Int, Iterable[String])]
//    = ShuffledRDD[16] at groupByKey at <console>:37
```

We typically have to move data from one node to another to be "grouped with" its key. Doing this is called "shuffling".

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

Think again what happens when you have to do a groupBy or a groupByKey. Remember our data is distributed! **Did you notice anything odd?**

```
val pairs = sc.parallelize(List((1, "one"), (2, "two"), (3, "three")))
pairs.groupByKey()
// res2: org.apache.spark.rdd.RDD[(Int, Iterable[String])]
//    = ShuffledRDD[16] at groupByKey at <console>:37
```

We typically have to move data from one node to another to be "grouped with" its key. Doing this is called "shuffling".

Shuffles Happen

Shuffles can be an enormous hit to because it means that Spark must send data from one node to another. Why? **Latency!**

Grouping and Reducing, Example

Let's start with an example. Given:

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

Assume we have an RDD of the purchases that users of the Swiss train company's, the CFF's, mobile app have made in the past month.

```scala
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.

Grouping and Reducing, Example

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

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



val purchasesPerMonth = ...
```

Grouping and Reducing, Example

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
```

Grouping and Reducing, Example

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]]
```

Grouping and Reducing, Example

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

```scala
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()
```

Grouping and Reducing, Example – What's Happening?

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?

Grouping and Reducing, Example – What's Happening?

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

Starting with `purchasesRdd`:

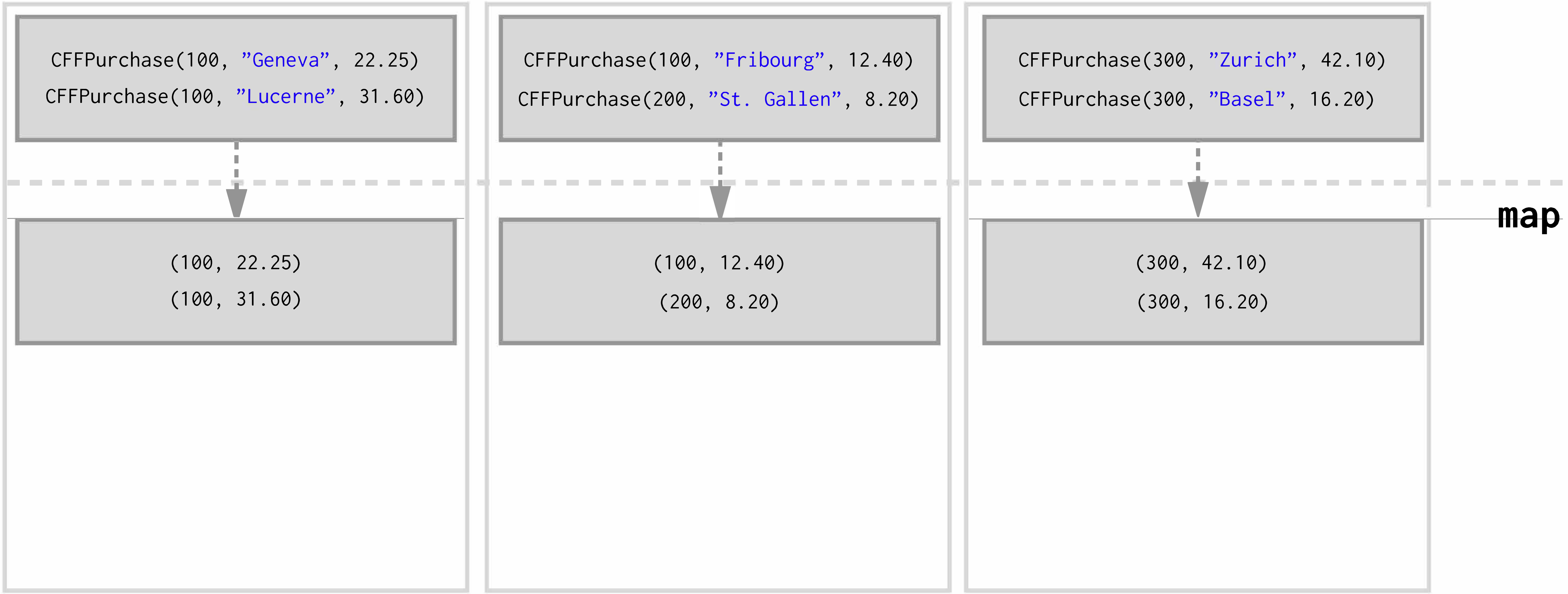
```
CFFPurchase(100, "Geneva", 22.25)
CFFPurchase(100, "Lucerne", 31.60)
```

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

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

Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?



Grouping and Reducing, Example

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]]
```

Grouping and Reducing, Example

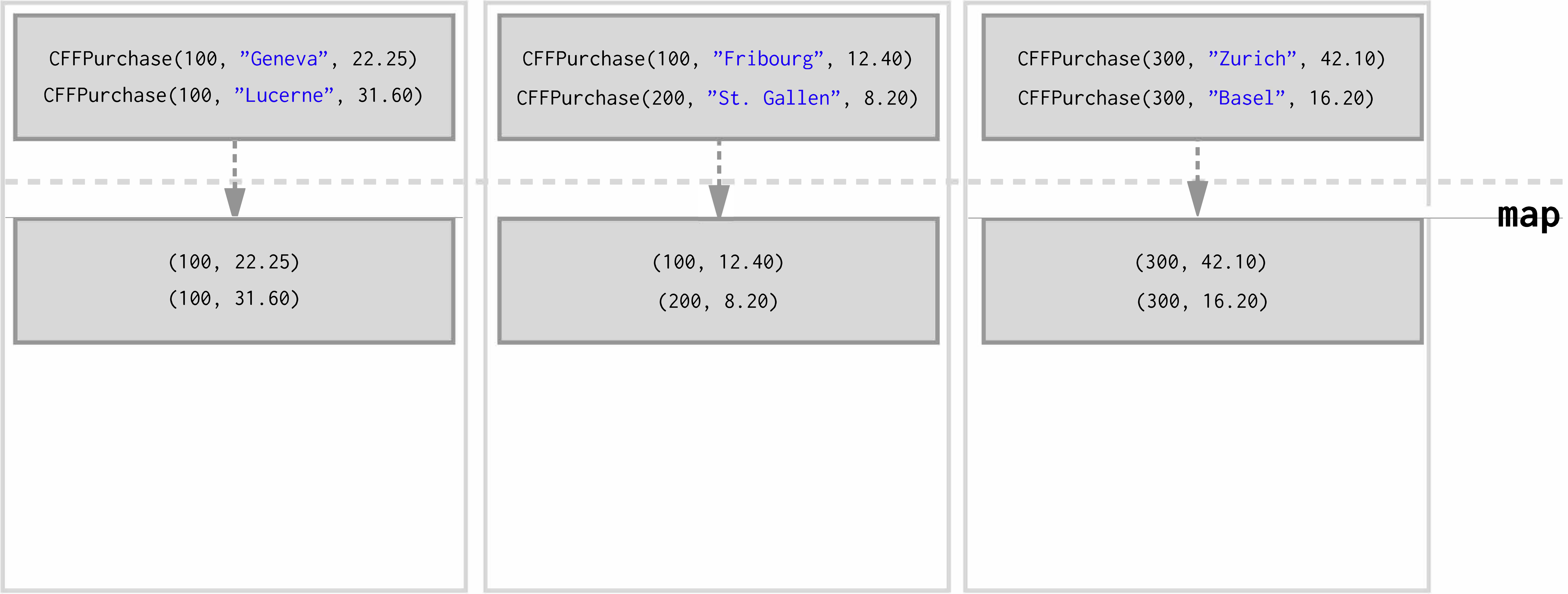
Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

```
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.

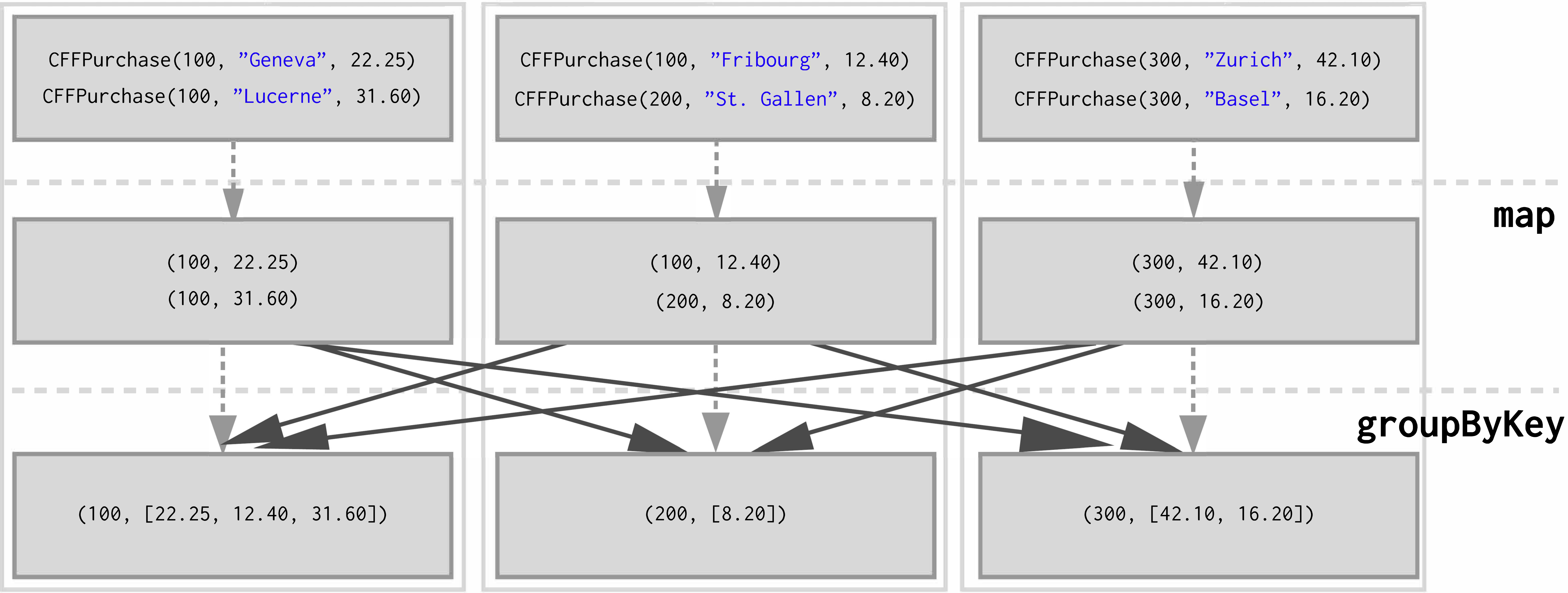
Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?



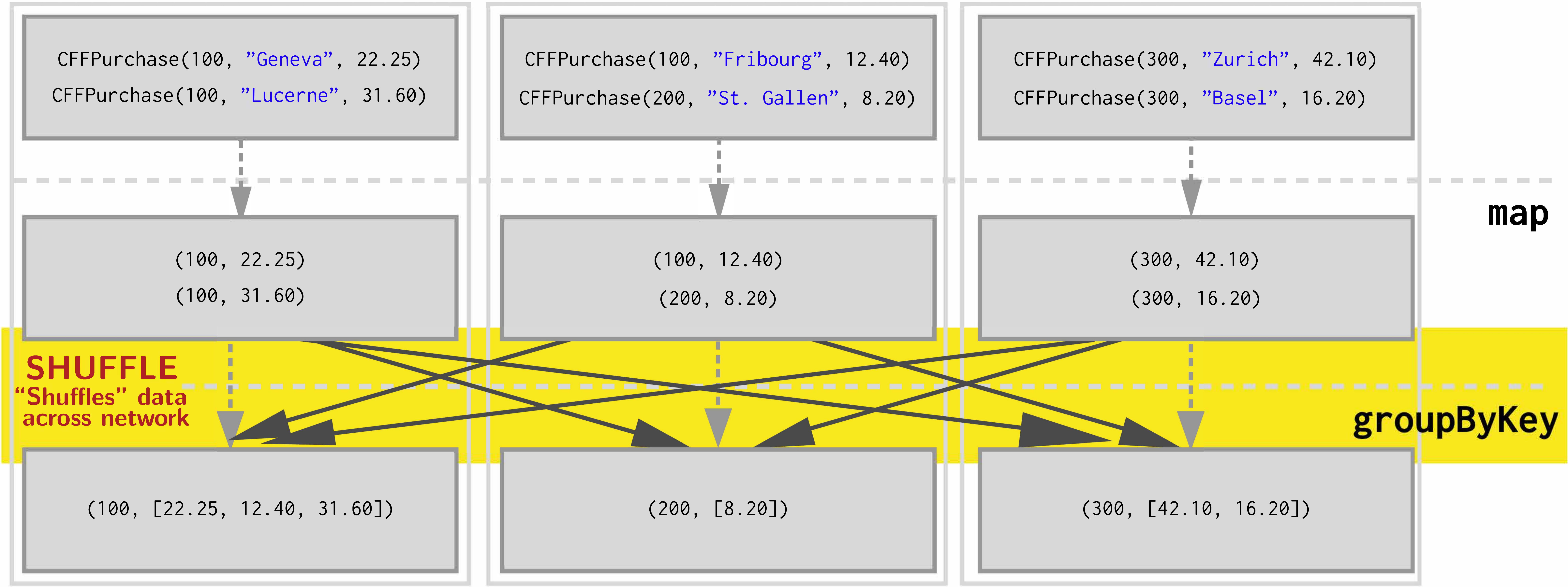
Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?



Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?



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
```

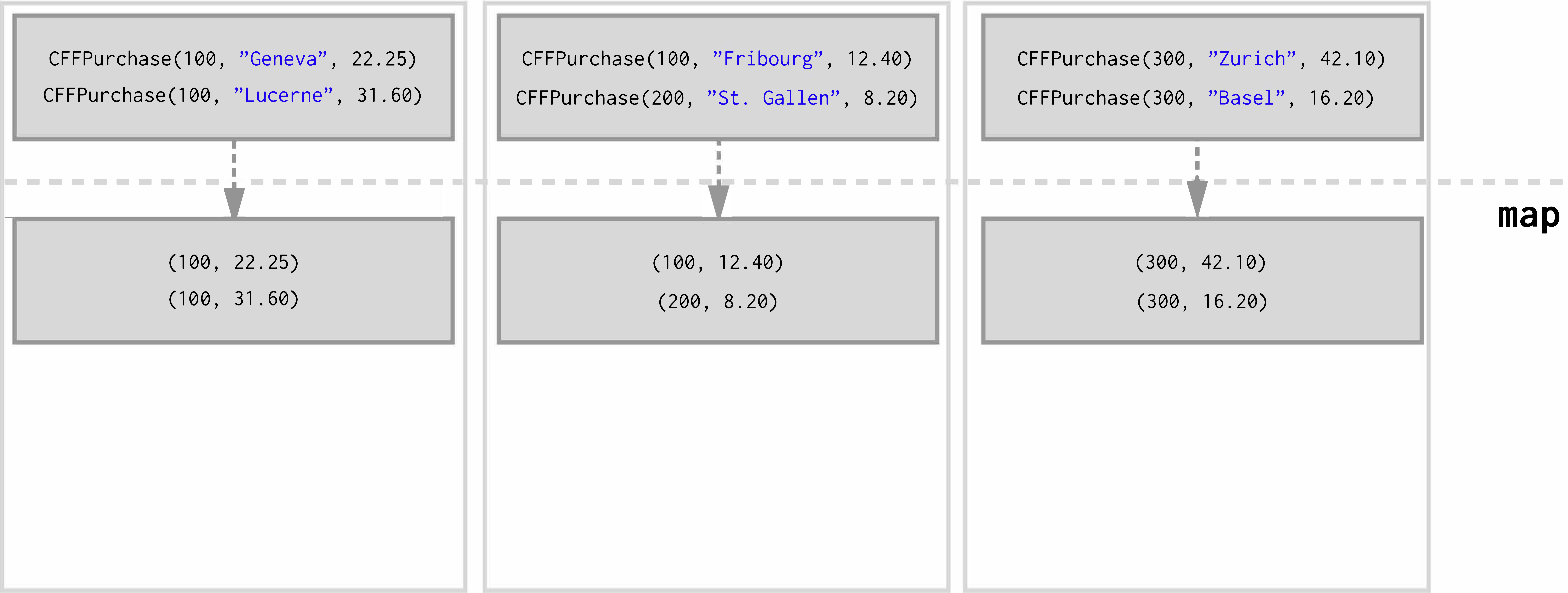
We don't want to be sending all of our data over the network if it's not absolutely required. Too much network communication kills performance.

Can we do a better job?

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

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.

Grouping and Reducing, Example – Optimized

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)]
```

Grouping and Reducing, Example – Optimized

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, (1, p.price))) // Pair RDD
              .reduceByKey(...) // ?
```

Grouping and Reducing, Example – Optimized

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, (1, p.price))) // Pair RDD
              .reduceByKey(...) // ?
```

Notice that the function passed to `map` *has changed. It's now* `p => (p.customerId, (1, p.price))`*.*

What function do we pass to `reduceByKey` in order to get a result that looks like: `(customerId, (numTrips, totalSpent))` returned?

Grouping and Reducing, Example – Optimized

```
val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, (1, p.price))) // Pair RDD
              .reduceByKey(...) // ?
```

(Int, Double)

Grouping and Reducing, Example – Optimized

```
val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, (1, p.price))) // Pair RDD
              .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
              .collect()
```

1+1

price+price

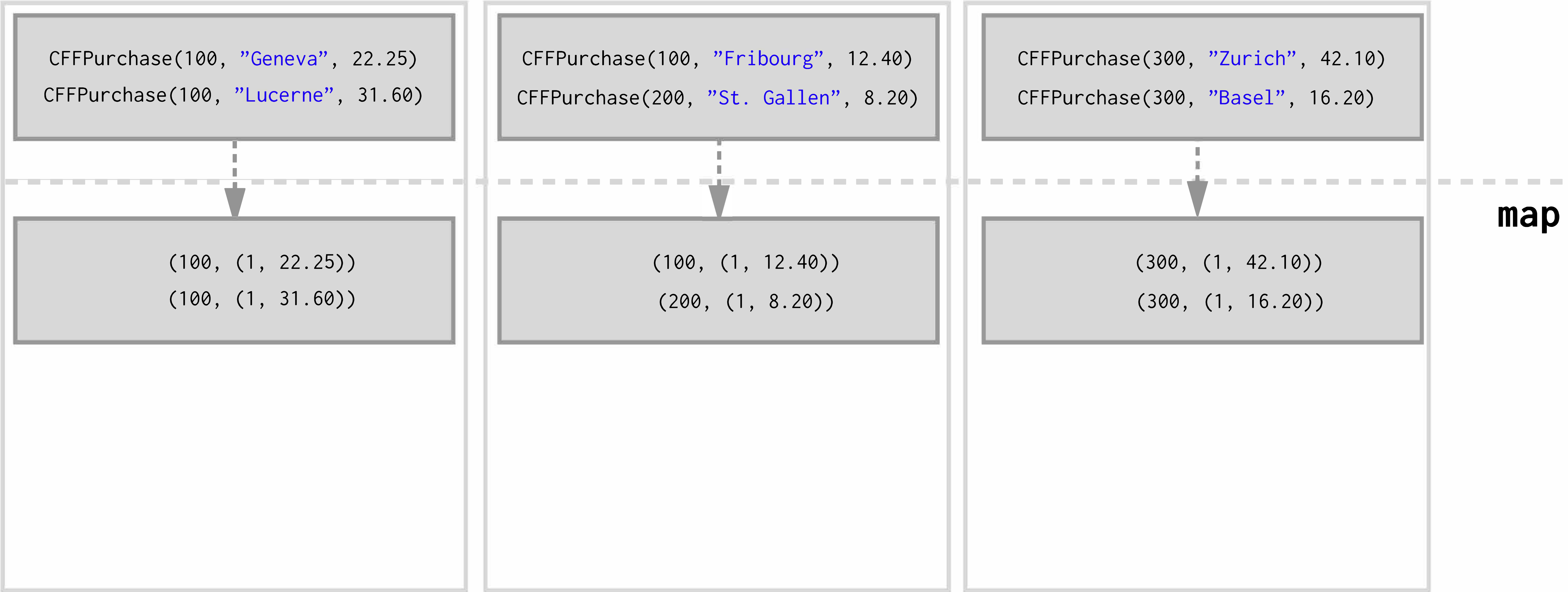
Grouping and Reducing, Example – Optimized

```
val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, (1, p.price))) // Pair RDD
              .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
              .collect()
```

What might this look like on the cluster?

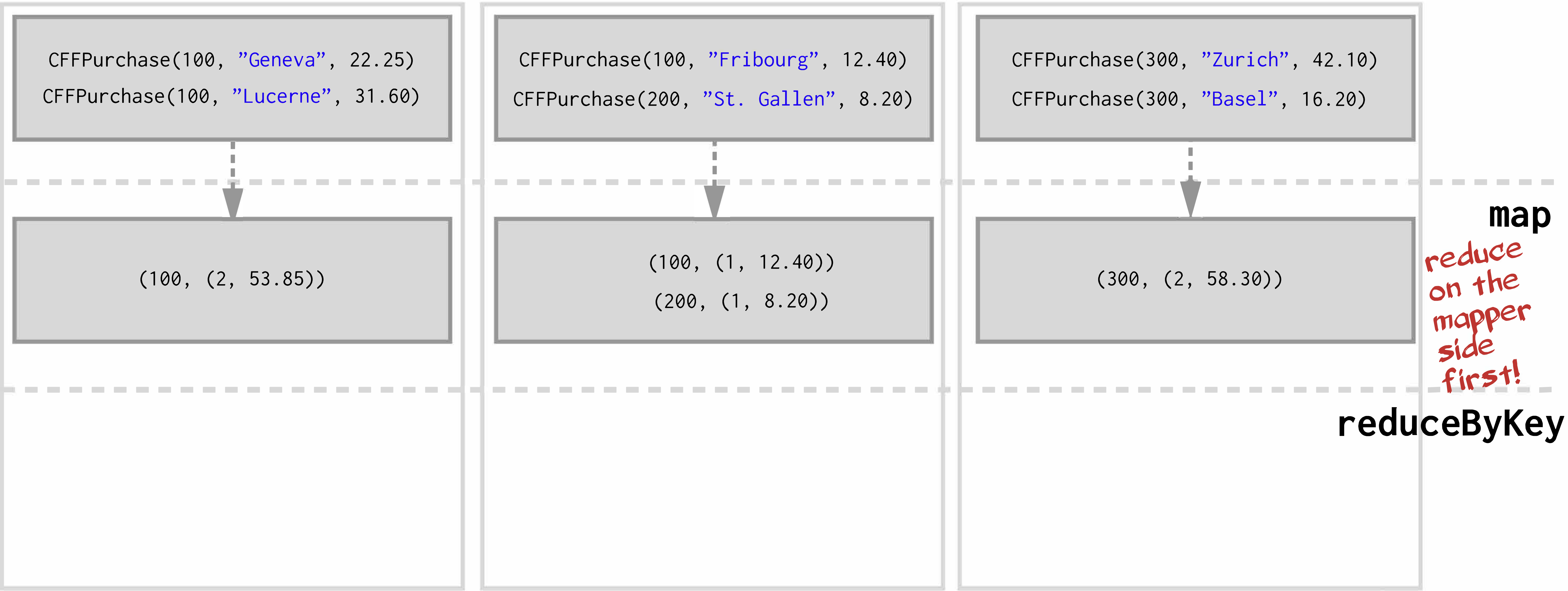
Grouping and Reducing, Example – Optimized

What might this look like on the cluster?



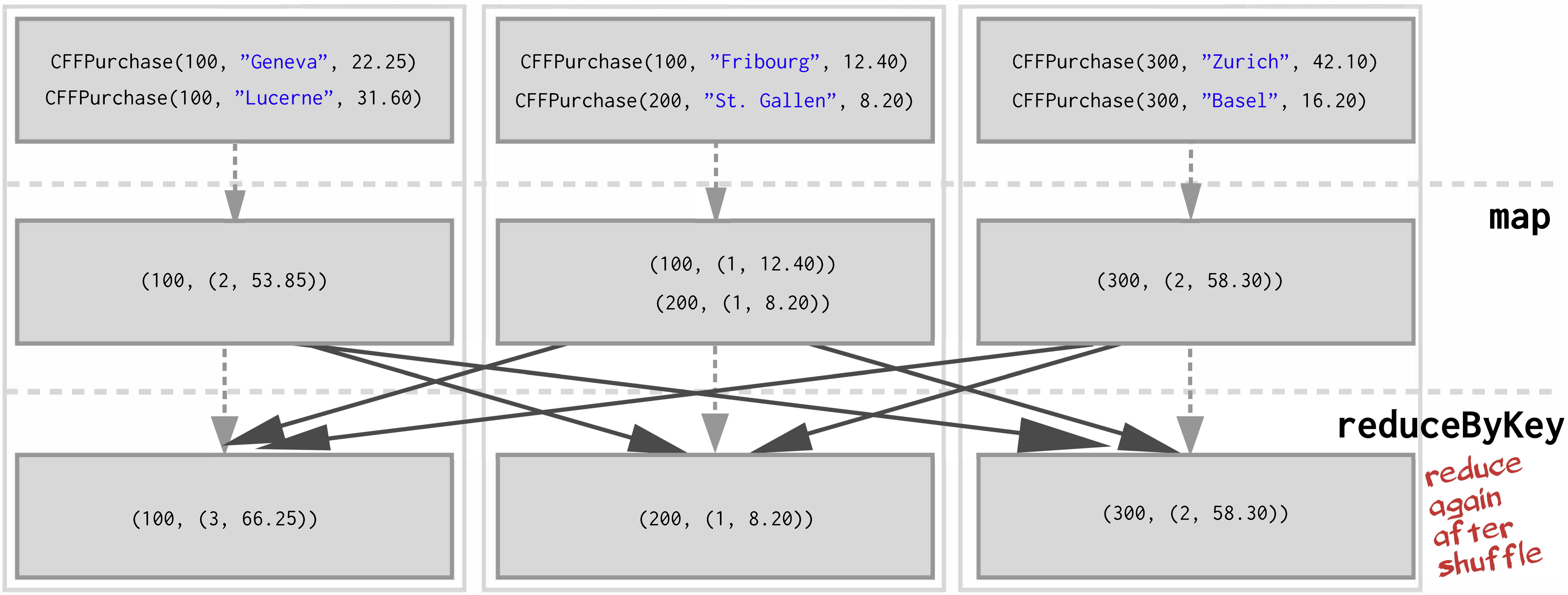
Grouping and Reducing, Example – Optimized

What might this look like on the cluster?



Grouping and Reducing, Example – Optimized

What might this look like on the cluster?



Grouping and Reducing, Example – Optimized

What are the benefits of this approach?

Grouping and Reducing, Example – Optimized

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!

Grouping and Reducing, Example – Optimized

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.

groupByKey and reduceByKey Running Times

```
> 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
```

Shuffling

Recall our example using `groupByKey`:

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

Shuffling

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?

Shuffling

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.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Partitioning

Big Data Analysis with Scala and Spark

Heather Miller

“Partitioning”?

In the last session, we were looking at an example involving `groupByKey`, before we discovered that this operation causes data to be *shuffled* over the network.

> *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.*

We concluded the last session asking ourselves,

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

Before we try to optimize that example any further, let’s first take a quick detour into what partitioning is...

Partitions

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.

Hash partitioning

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

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

Hash partitioning

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 partitioning

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.

Hash Partitioning: Example

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

Hash Partitioning: Example

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`).

$$p = K.\text{hashcode}() \ \% \ \text{numPartitions}$$
$$= K \% 4$$

Hash Partitioning: Example

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: `[]`

$p = k \% 4$

The result is a very unbalanced distribution which hurts performance.

Range Partitioning: Example

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]`

Range Partitioning: Example

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.

Partitioning Data

How do we set a partitioning for our data?

Partitioning Data

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: `partitionBy`

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

Partitioning Data: `partitionBy`

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

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
```

Partitioning Data: `partitionBy`

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()
```

Partitioning Data: `partitionBy`

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*.

Partitioning Data: `partitionBy`

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.

Partitioning Data Using Transformations

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.

Partitioning Data Using Transformations

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.

Partitioning Data Using Transformations

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

Why?

Partitioning Data Using Transformations

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

Why?

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

Partitioning Data Using Transformations

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

Why?

Consider the `map` transformation. Given that we 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.*,:

Partitioning Data Using Transformations

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

Why?

Consider the `map` transformation. Given that we 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.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Optimizing with Partitioners

Big Data Analysis with Scala and Spark

Heather Miller

Optimizing with Partitioners

We saw in the last session that Spark makes a few kinds of partitioners available out-of-the-box to users:

- **hash partitioners** and
- **range partitioners**.

We also learned what kinds of operations may introduce new partitioners, or which may discard custom partitioners.

However, we haven't covered *why* someone would want to repartition their data.

Optimizing with Partitioners

We saw in the last session that Spark makes a few kinds of partitioners available out-of-the-box to users:

- **hash partitioners** and
- **range partitioners**.

We also learned what kinds of operations may introduce new partitioners, or which may discard custom partitioners.

However, we haven't covered *why* someone would want to repartition their data.

Partitioning can bring substantial performance gains, especially in the face of shuffles.

Optimization using range partitioning

Using range partitioners we can optimize our earlier use of `reduceByKey` so that it does not involve any shuffling over the network at all!

Optimization using range partitioning

Using range partitioners we can optimize our earlier use of reduceByKey so that it does not involve any shuffling over the network at all!

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

val partitioned = pairs.partitionBy(tunedPartitioner)
                       .persist()

val purchasesPerCust =
  partitioned.map(p => (p._1, (1, p._2)))

val purchasesPerMonth = purchasesPerCust
      .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
      .collect()
```

Optimization 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
```

Optimization 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!

Partitioning Data: `partitionBy`, Another Example

From pages 61-64 of the Learning Spark book

Consider an application that keeps a large table of user information in memory:

- `userData` - **BIG**, containing `(UserID, UserInfo)` pairs, where `UserInfo` contains a list of topics the user is subscribed to.

The application periodically combines this **big** table with a smaller file representing events that happened in the past five minutes.

- `events` – *small*, containing `(UserID, LinkInfo)` pairs for users who have clicked a link on a website in those five minutes:

For example, we may wish to count how many users visited a link that was not to one of their subscribed topics. We can perform this combination with Spark's `join` operation, which can be used to group the `UserInfo` and `LinkInfo` pairs for each `UserID` by key.

Partitioning Data: partitionBy, Another Example

From pages 61-64 of the Learning Spark book

```
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

def processNewLogs(logFileName: String) {
  val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
  val joined = userData.join(events) //RDD of (UserID, (UserInfo, LinkInfo))
  val offTopicVisits = joined.filter {
    case (userId, (userInfo, linkInfo)) => // Expand the tuple
      !userInfo.topics.contains(linkInfo.topic)
  }.count()
  println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

Is this OK?

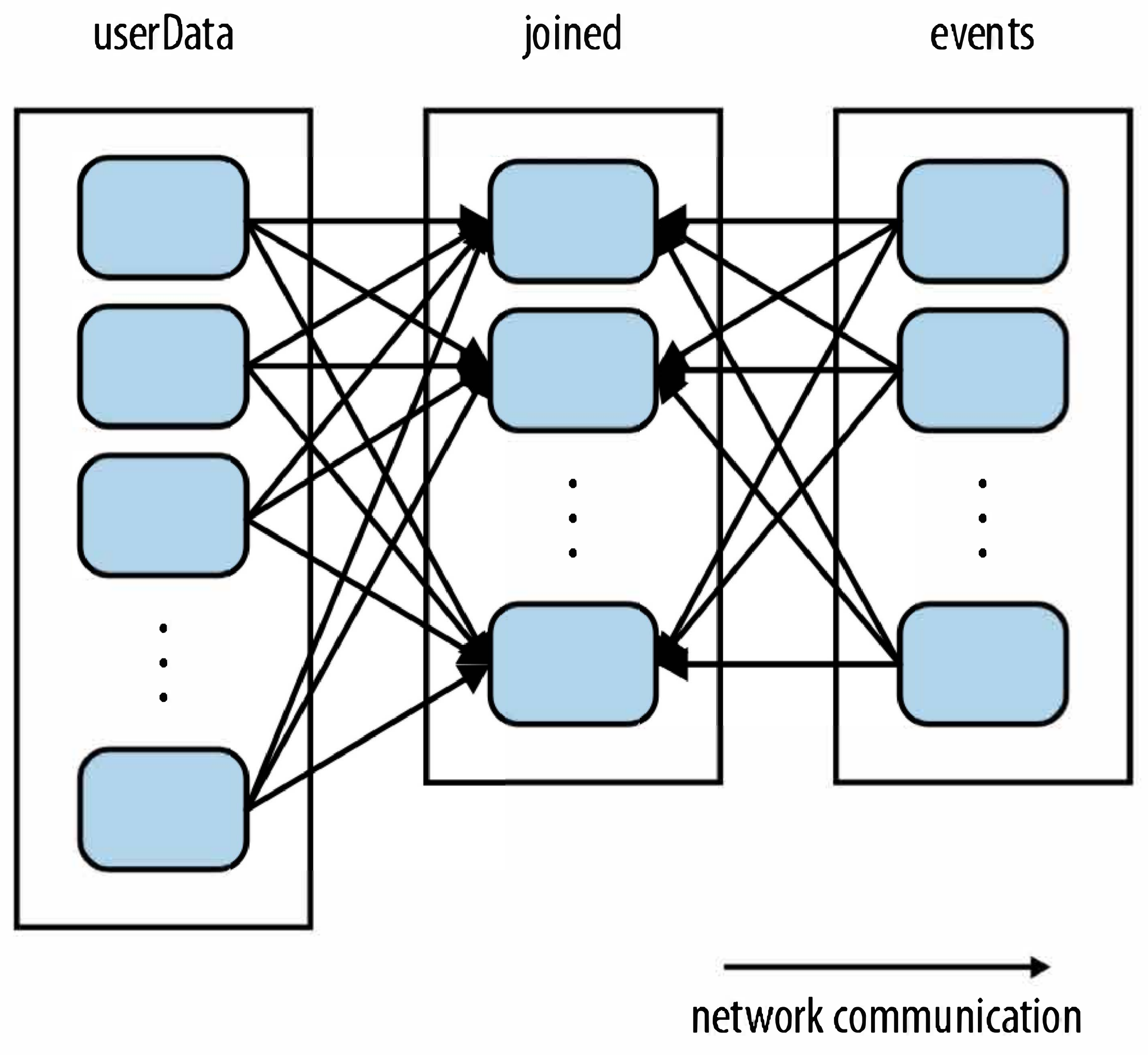
Partitioning Data: `partitionBy`, Another Example

From pages 61-64 of the Learning Spark book

It will be very inefficient!

Why? The `join` operation, called each time `processNewLogs` is invoked, does not know anything about how the keys are partitioned in the datasets.

By default, this operation will hash all the keys of both datasets, sending elements with the same key hash across the network to the same machine, and then join together the elements with the same key on that machine. **Even though userData doesn't change!**



Partitioning Data: `partitionBy`, Another Example

Fixing this is easy. Just use `partitionBy` on the **big** `userData` RDD at the start of the program!

Partitioning Data: `partitionBy`, Another Example

Fixing this is easy. Just use `partitionBy` on the **big** `userData` RDD at the start of the program!

Therefore, `userData` becomes:

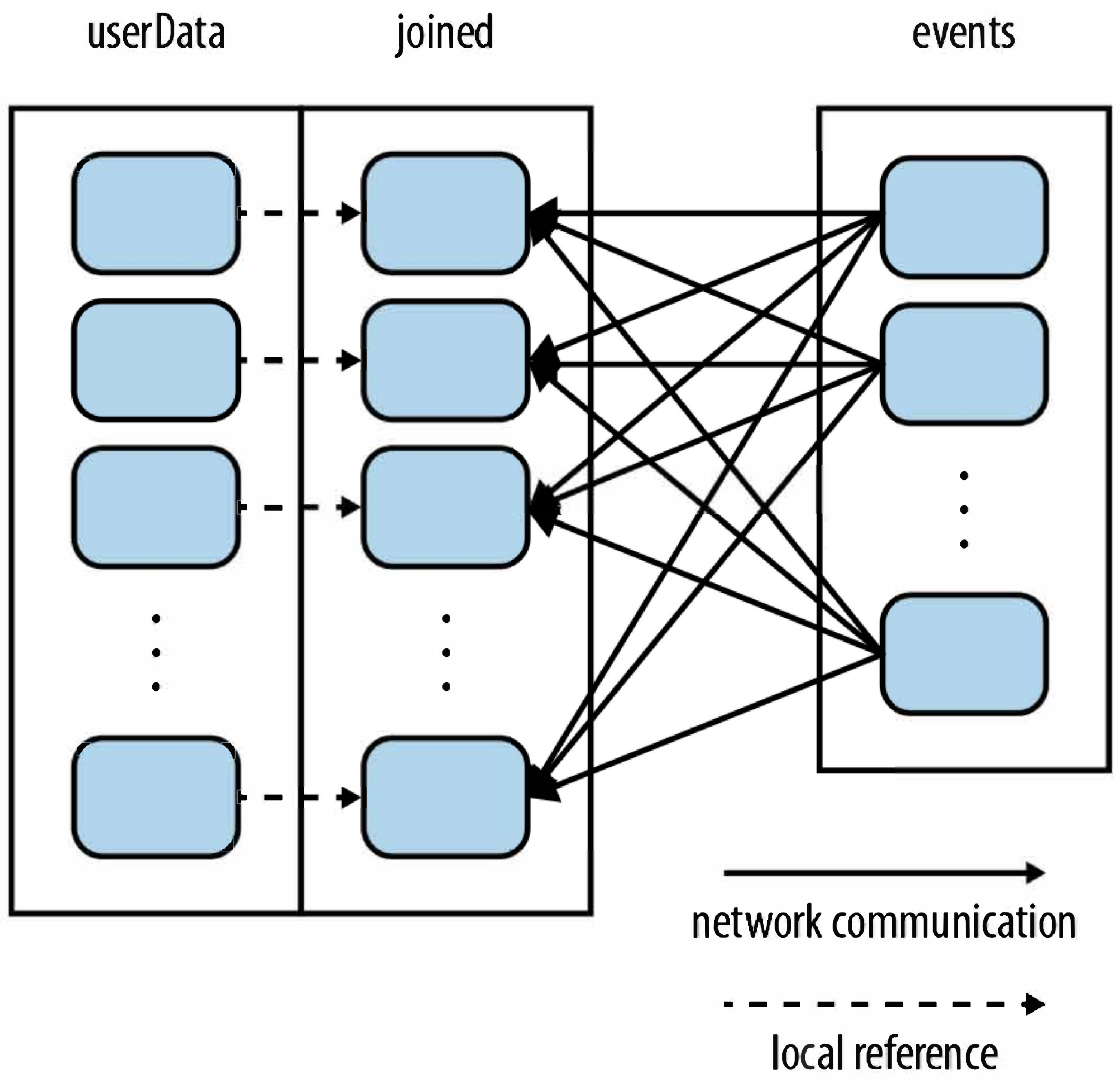
```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
                 .partitionBy(new HashPartitioner(100)) // Create 100 partitions
                 .persist()
```

Since we called `partitionBy` when building `userData`, Spark will now know that it is hash-partitioned, and calls to `join` on it will take advantage of this information.

In particular, when we call `userData.join(events)`, Spark will shuffle only the `events` RDD, sending events with each particular `UserID` to the machine that contains the corresponding hash partition of `userData`.

Partitioning Data: `partitionBy`, Another Example

Or, shown visually:



Now that `userData` is pre-partitioned, Spark will shuffle only the `events` RDD, sending events with each particular `UserID` to the machine that contains the corresponding hash partition of `userData`.

Back to shuffling

Recall our example using `groupByKey`:

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

Back to shuffling

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.

Back to shuffling

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.

Grouping is done using a hash partitioner with default parameters.

Back to shuffling

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.

Grouping is done using a hash partitioner with default parameters.

The result RDD, `purchasesPerCust`, is configured to use the hash partitioner that was used to construct it.

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.

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*

How do I know a shuffle will occur?

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
```

Operations that *might* cause a shuffle

- `cogroup`
- `groupWith`
- `join`
- `leftOuterJoin`
- `rightOuterJoin`
- `groupByKey`
- `reduceByKey`
- `combineByKey`
- `distinct`
- `intersection`
- `repartition`
- `coalesce`

Avoiding a Network Shuffle By Partitioning

There are a few ways to use operations that *might* cause a shuffle and to still avoid much or all network shuffling.

Can you think of an example?

Avoiding a Network Shuffle By Partitioning

There are a few ways to use operations that *might* cause a shuffle and to still avoid much or all network shuffling.

Can you think of an example?

2 Examples:

1. `reduceByKey` running on a pre-partitioned RDD will cause the values to be computed *locally*, requiring only the final reduced value has to be sent from the worker to the driver.
2. `join` called on two RDDs that are pre-partitioned with the same partitioner and cached on the same machine will cause the join to be computed *locally*, with no shuffling across the network.

Shuffles Happen: Key Takeaways

How your data is organized on the cluster, and what operations you're doing with it matters!

We've seen speedups of 10x on small examples just by trying to ensure that data is not transmitted over the network to other machines.

This can hugely affect your day job if you're trying to run a job that should run in 4 hours, but due to a missed opportunity to partition data or optimize away a shuffle, it could take **40 hours** instead.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Wide vs Narrow Dependencies

Big Data Analysis with Scala and Spark

Heather Miller

Not All Transformations are Created Equal

Some transformations significantly more expensive (latency) than others

E.g., requiring lots of data to be transferred over the network, sometimes unnecessarily.

Not All Transformations are Created Equal

Some transformations significantly more expensive (latency) than others

E.g., requiring lots of data to be transferred over the network, sometimes unnecessarily.

In the past sessions:

- we learned that shuffling sometimes happens on some transformations.

In this session:

- we'll look at how RDDs are represented.
- we'll dive into how and when Spark decides it must shuffle data.
- we'll see how these dependencies make fault tolerance possible.

Lineages

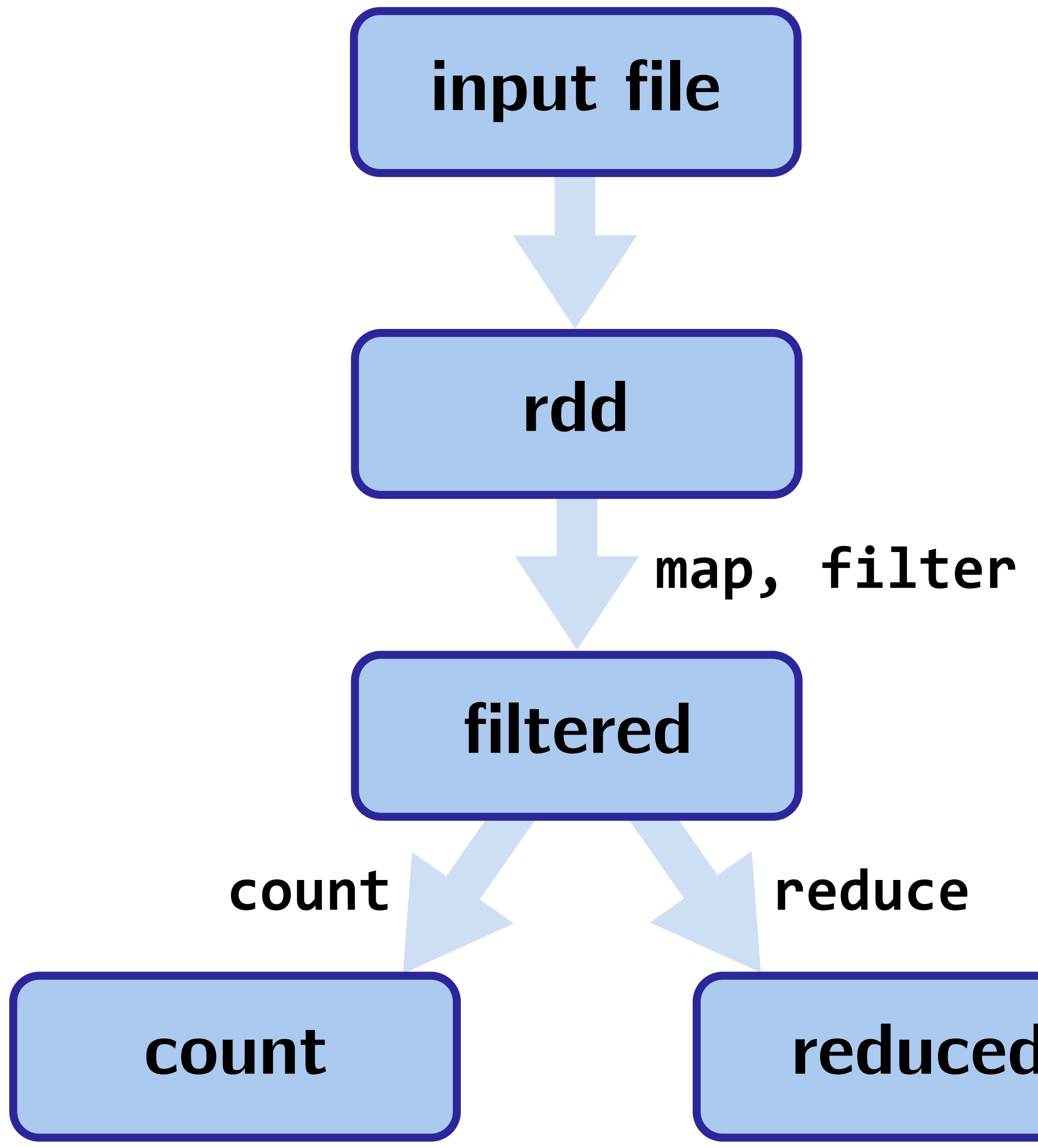
Computations on RDDs are represented as a **lineage graph**; a Directed Acyclic Graph (DAG) representing the computations done on the RDD.

Lineages

Computations on RDDs are represented as a **lineage graph**; a Directed Acyclic Graph (DAG) representing the computations done on the RDD.

Example:

```
val rdd = sc.textFile(...)
val filtered = rdd.map(...)
                  .filter(...)
                  .persist()
val count = filtered.count()
val reduced = fitered.reduce(...)
```

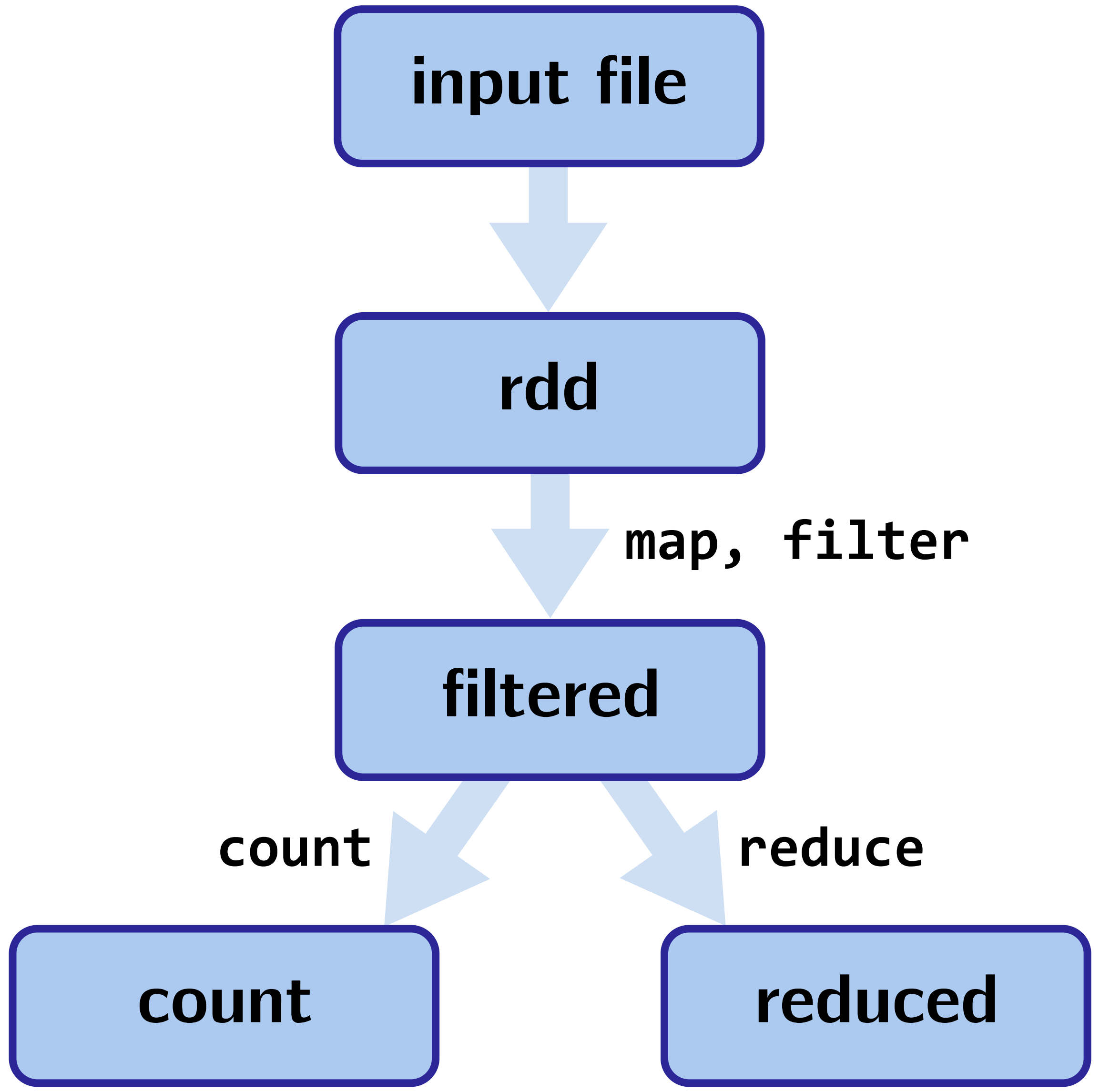


Lineages

Computations on RDDs are represented as a **lineage graph**; a Directed Acyclic Graph (DAG) representing the computations done on the RDD.

Example:

```
val rdd = sc.textFile(...)
val filtered = rdd.map(...)
                  .filter(...)
                  .persist()
val count = filtered.count()
val reduced = fitered.reduce(...)
```



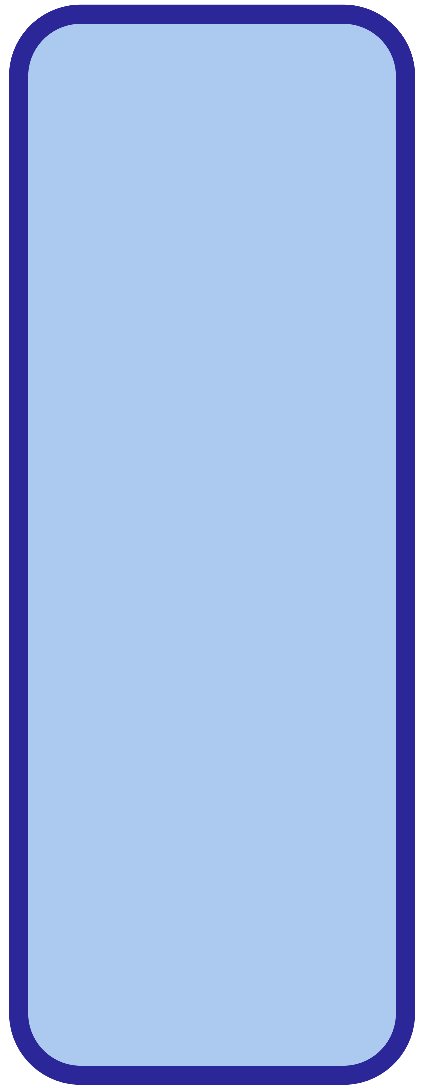
Spark represents RDDs in terms of these lineage graphs/DAGs

In fact, this is the representation/DAG is what Spark analyzes to do optimizations.

How are RDDs represented?

RDDs are made up of 2 important parts.

(but are made up of 4 parts in total)



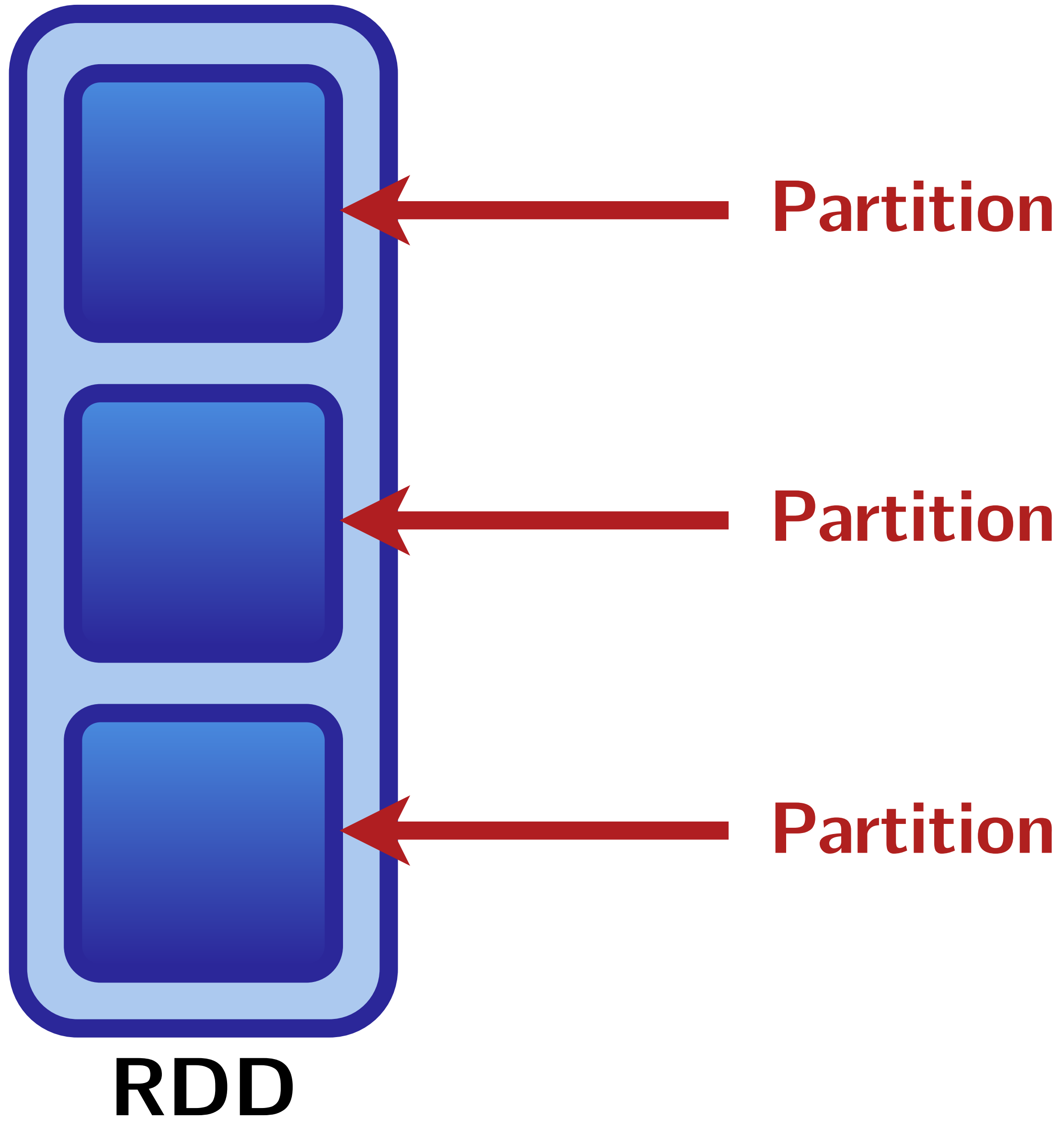
RDD

RDDs are represented as:

How are RDDs represented?

RDDs are made up of 2 important parts.

(but are made up of 4 parts in total)



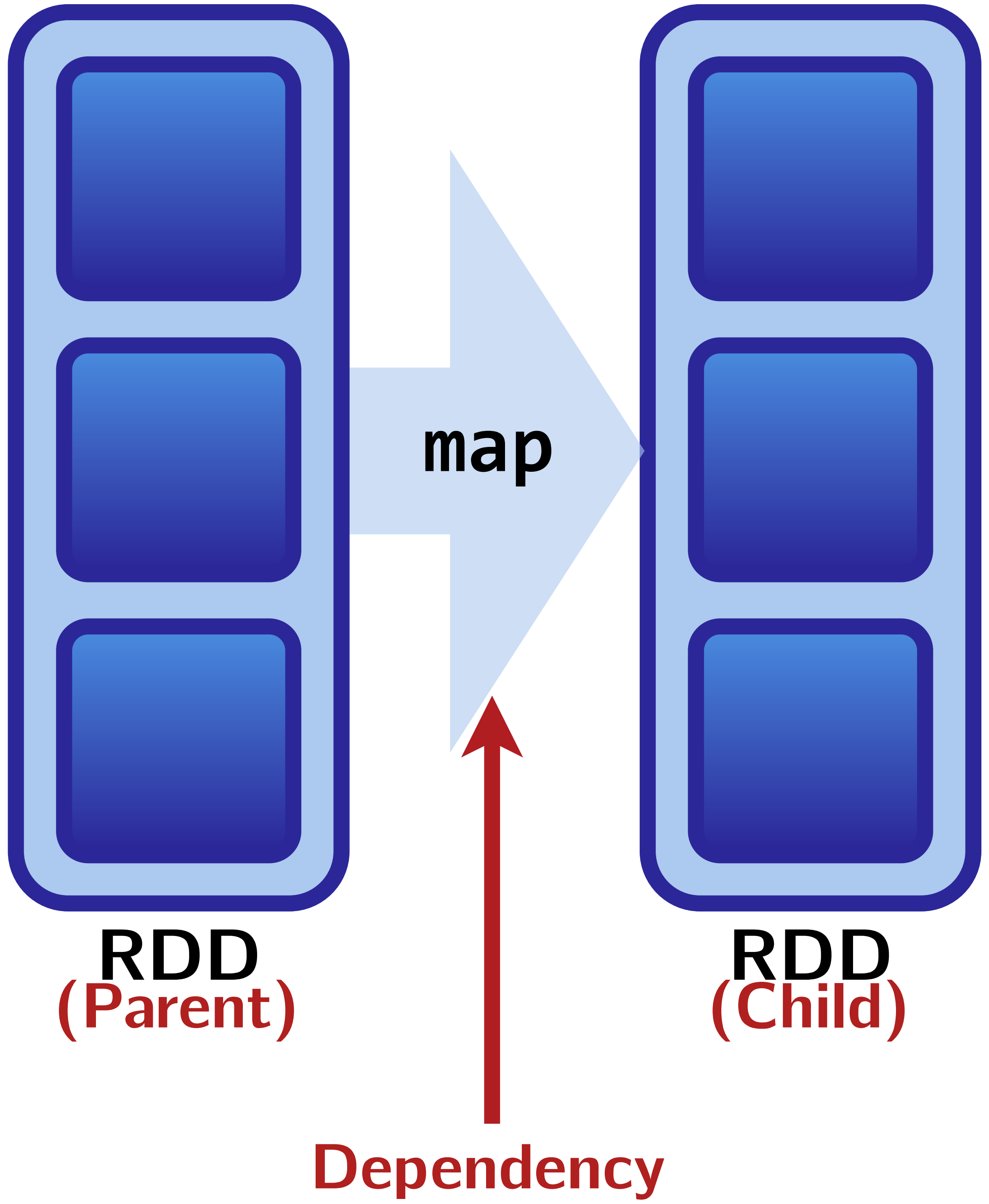
RDDs are represented as:

- **Partitions**. Atomic pieces of the dataset. One or many per compute node.

How are RDDs represented?

RDDs are made up of 2 important parts.

(but are made up of 4 parts in total)



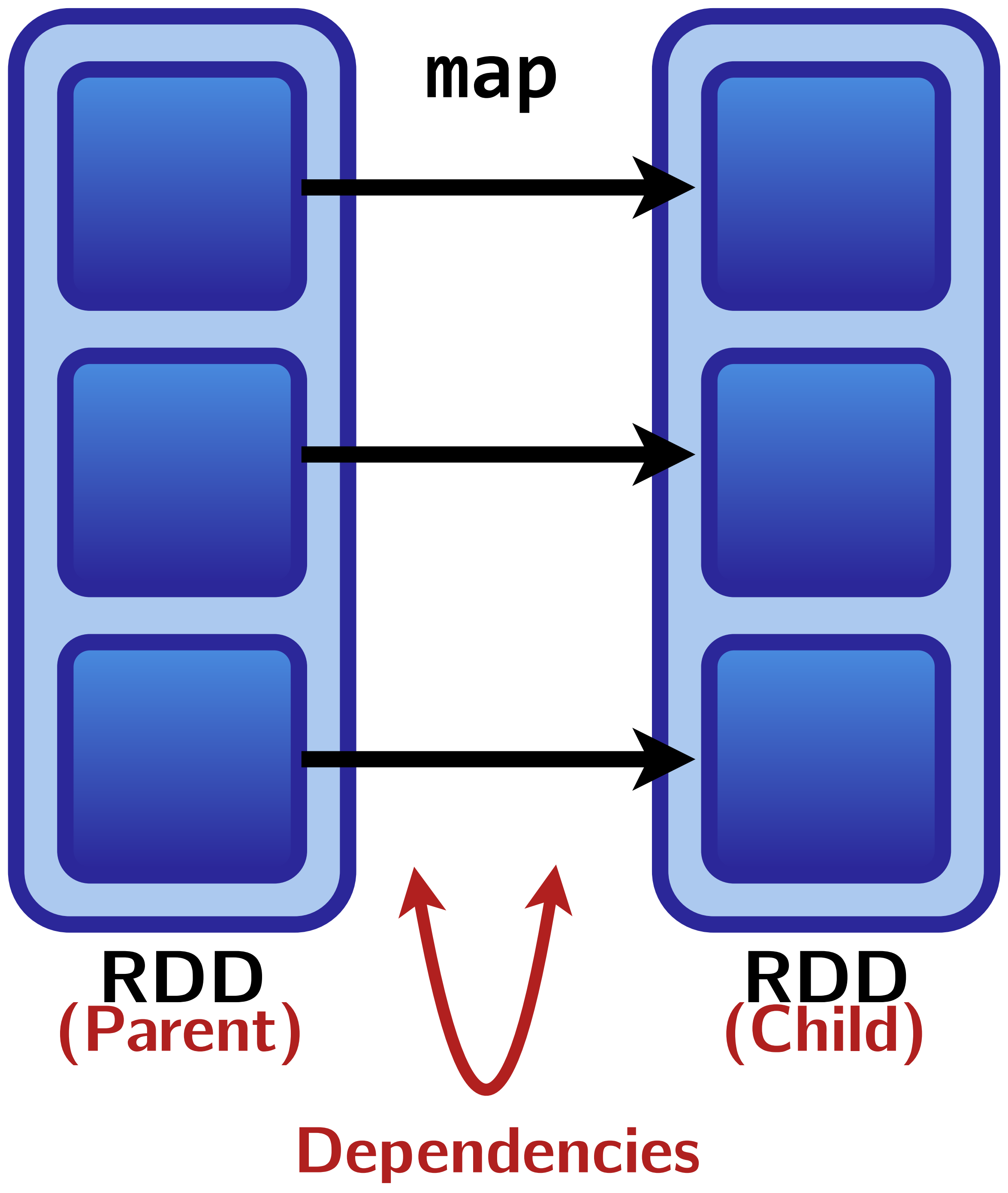
RDDs are represented as:

- **Partitions.** Atomic pieces of the dataset. One or many per compute node.
- **Dependencies.** Models relationship between this RDD and its partitions with the RDD(s) it was derived from.

How are RDDs represented?

RDDs are made up of 2 important parts.

(but are made up of 4 parts in total)



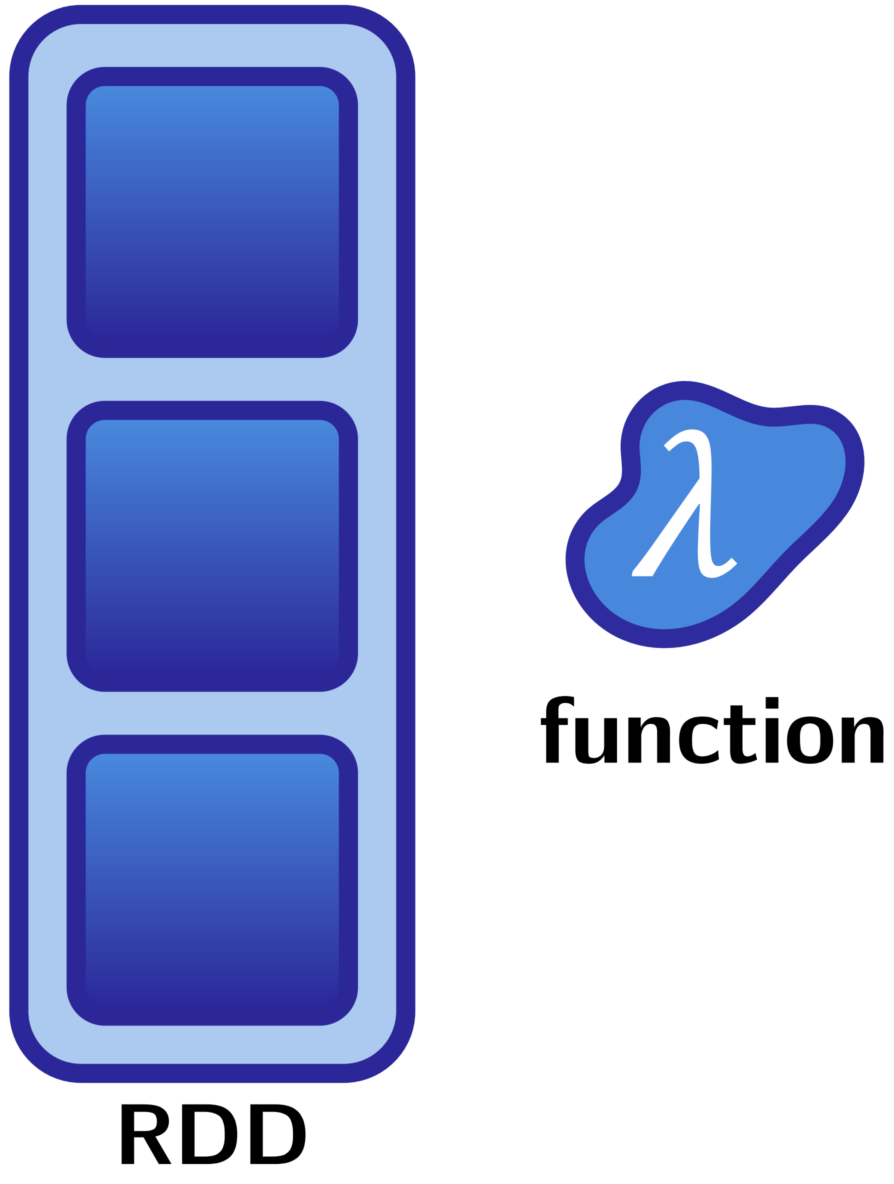
RDDs are represented as:

- **Partitions.** Atomic pieces of the dataset. One or many per compute node.
- **Dependencies.** Models relationship between this RDD **and its partitions** with the RDD(s) it was derived from.

How are RDDs represented?

RDDs are made up of 2 important parts.

(but are made up of 4 parts in total)



RDDs are represented as:

- **Partitions.** Atomic pieces of the dataset. One or many per compute node.
- **Dependencies.** Models relationship between this RDD and its partitions with the RDD(s) it was derived from.
- **A function** for computing the dataset based on its parent RDDs.
- **Metadata** about its partitioning scheme and data placement.

RDD Dependencies and Shuffles

Previously, we arrived at the following rule of thumb for trying to determine when a shuffle might occur:

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

RDD Dependencies and Shuffles

Previously, we arrived at the following rule of thumb for trying to determine when a shuffle might occur:

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

In fact, RDD dependencies encode when data must move across the network.

RDD Dependencies and Shuffles

Previously, we arrived at the following rule of thumb for trying to determine when a shuffle might occur:

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

In fact, RDD dependencies encode when data must move across the network.

Transformations cause shuffles. Transformations can have two kinds of dependencies:

1. **Narrow Dependencies**
2. **Wide Dependencies**

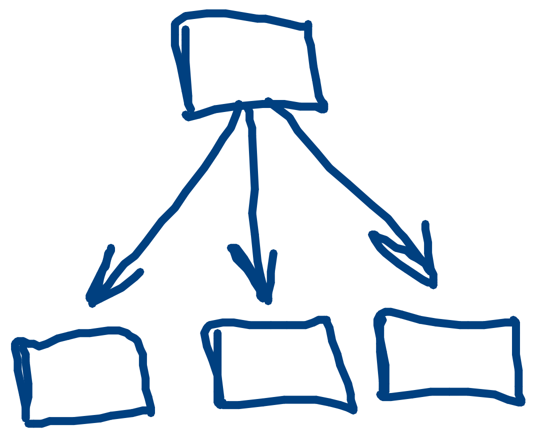
Narrow Dependencies vs Wide Dependences

Narrow Dependencies

Each partition of the parent RDD is used by at most one partition of the child RDD.

Wide Dependencies

Each partition of the parent RDD may be depended on by **multiple** child partitions.



Narrow Dependencies vs Wide Dependences

Narrow Dependencies

Each partition of the parent RDD is used by at most one partition of the child RDD.

Fast! No shuffle necessary. Optimizations like pipelining possible.

Wide Dependencies

Each partition of the parent RDD may be depended on by **multiple** child partitions.

Slow! Requires all or some data to be shuffled over the network.

Narrow Dependencies vs Wide Dependences, Visually

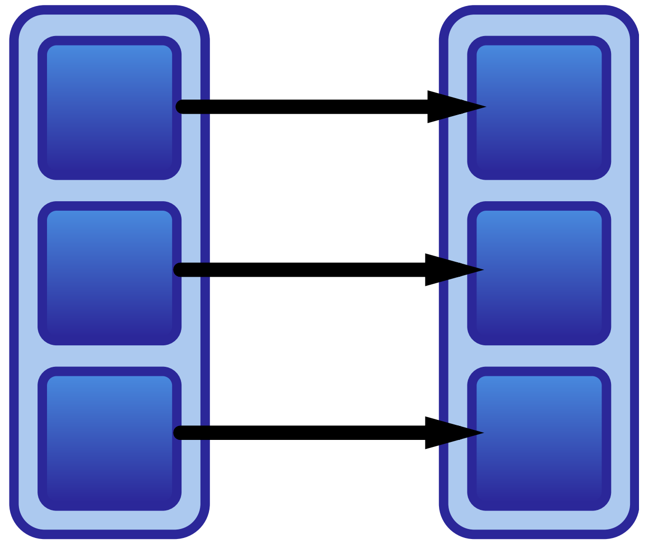
Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.

Narrow Dependencies vs Wide Dependences, Visually

Narrow dependencies:

Each partition of the parent RDD is used
by at most one partition of the child RDD.

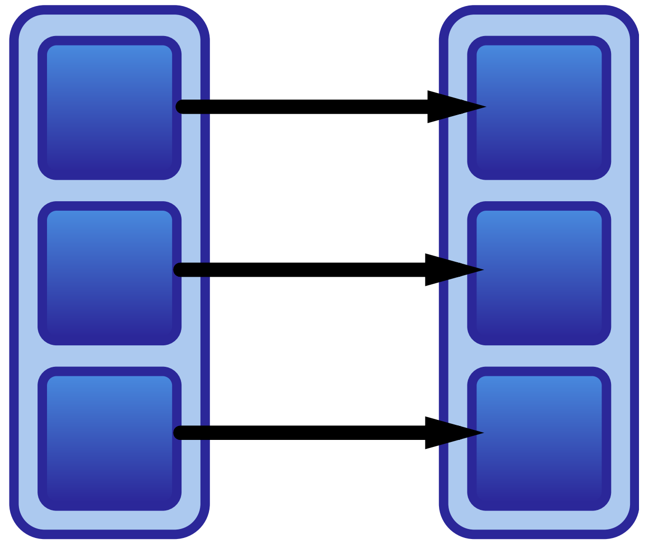


`map, filter`

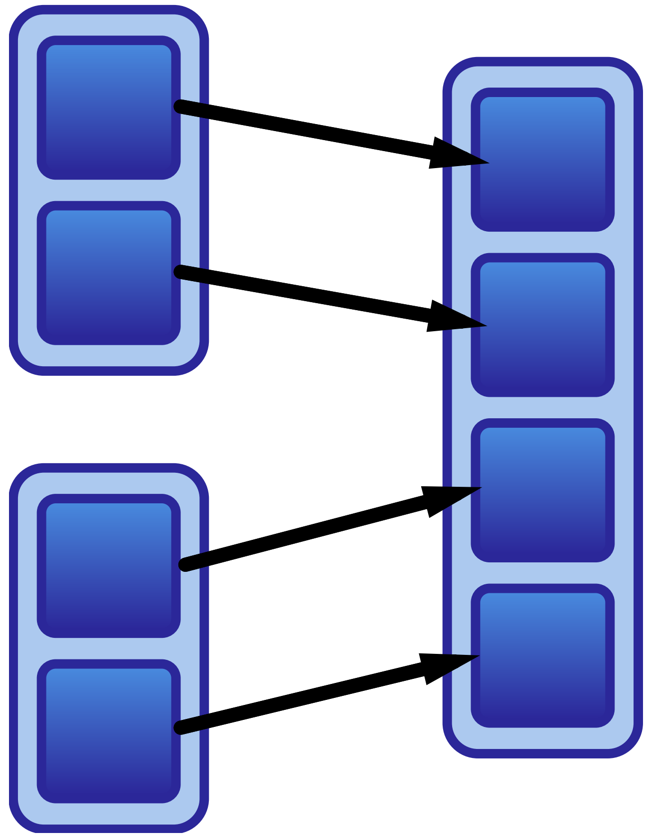
Narrow Dependencies vs Wide Dependences, Visually

Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.



`map, filter`

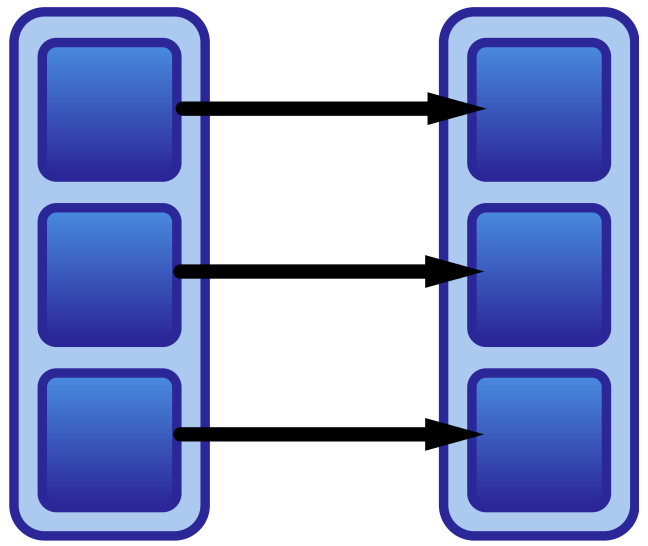


`union`

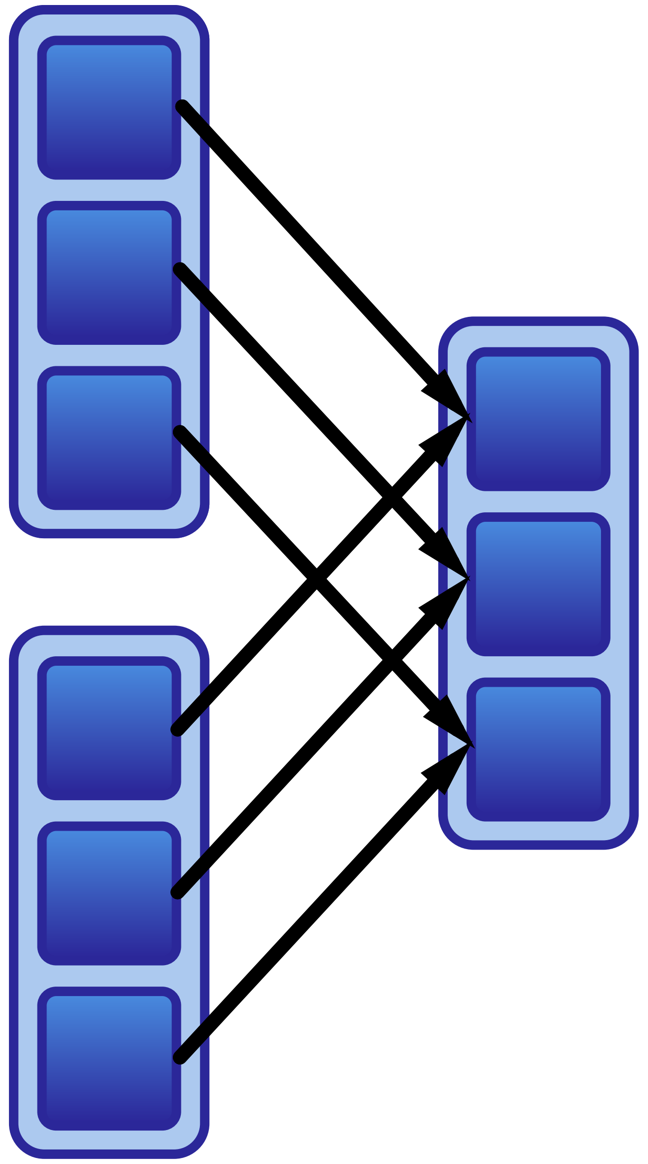
Narrow Dependencies vs Wide Dependences, Visually

Narrow dependencies:

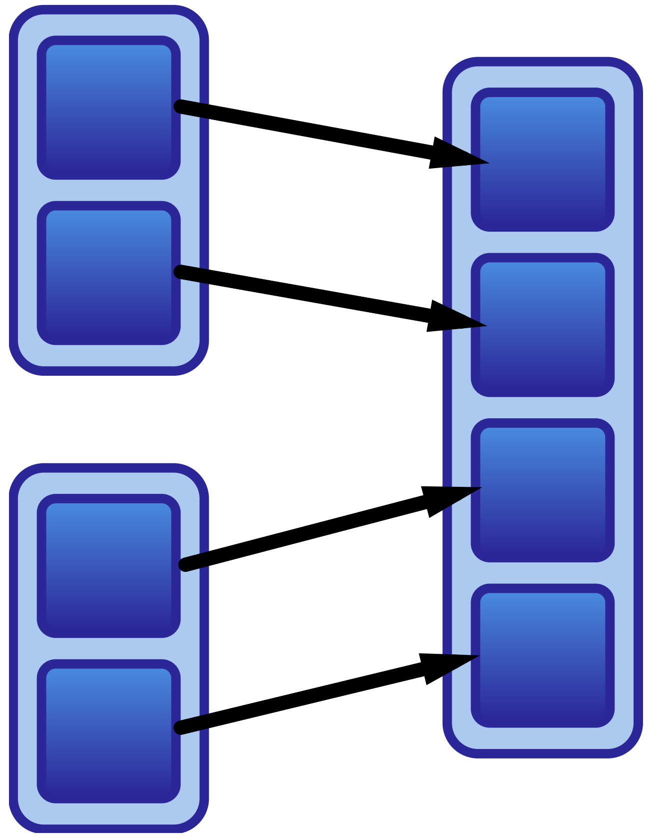
Each partition of the parent RDD is used by at most one partition of the child RDD.



`map, filter`



`join` *with co-partitioned inputs*



`union`

Narrow Dependencies vs Wide Dependences, Visually

Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.

Wide dependencies:

Each partition of the parent RDD may be depended on by multiple child partitions.

Narrow Dependencies vs Wide Dependences, Visually

Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.

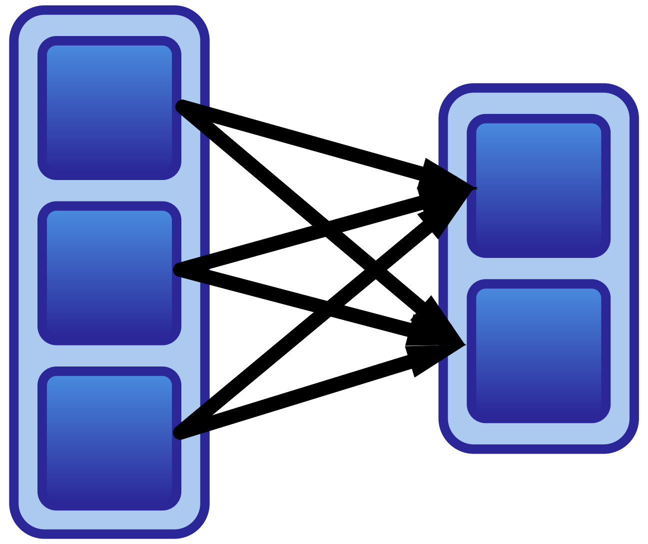
`join` *with co-partitioned inputs*

`map, filter`

`union`

Wide dependencies:

Each partition of the parent RDD may be depended on by multiple child partitions.



`groupByKey`

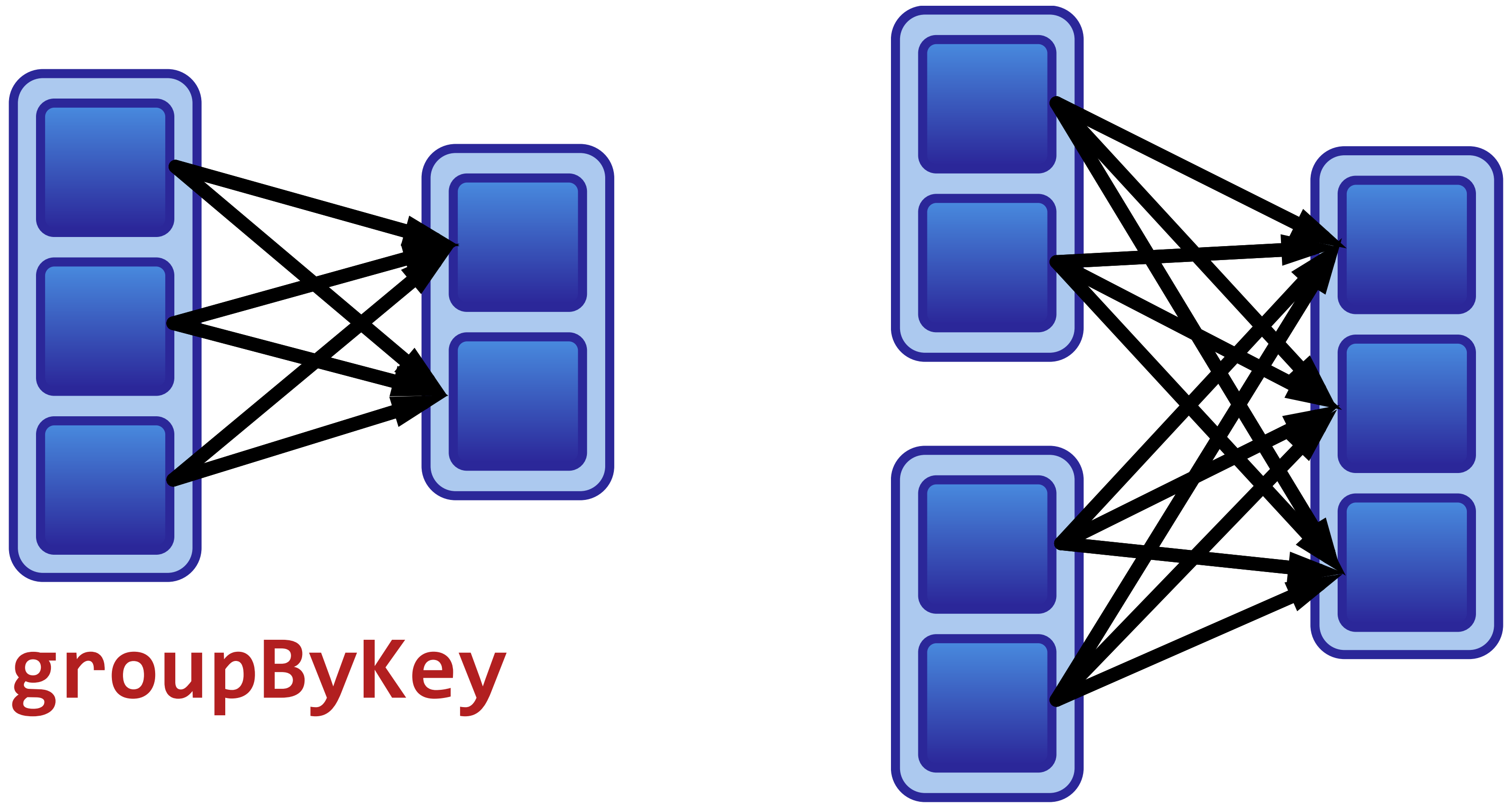
Narrow Dependencies vs Wide Dependences, Visually

Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.

Wide dependencies:

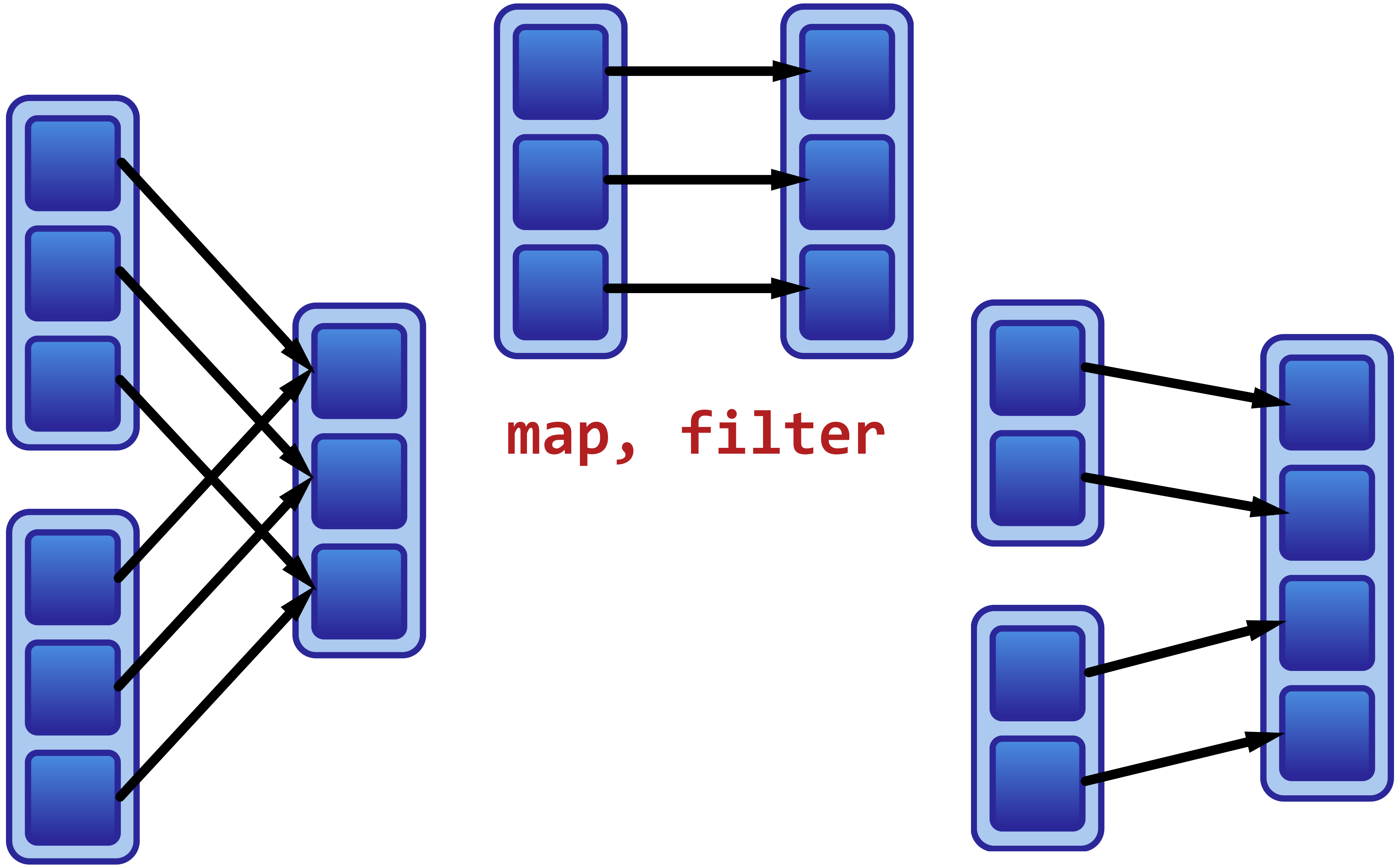
Each partition of the parent RDD may be depended on by multiple child partitions.



Narrow Dependencies vs Wide Dependences, Visually

Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.

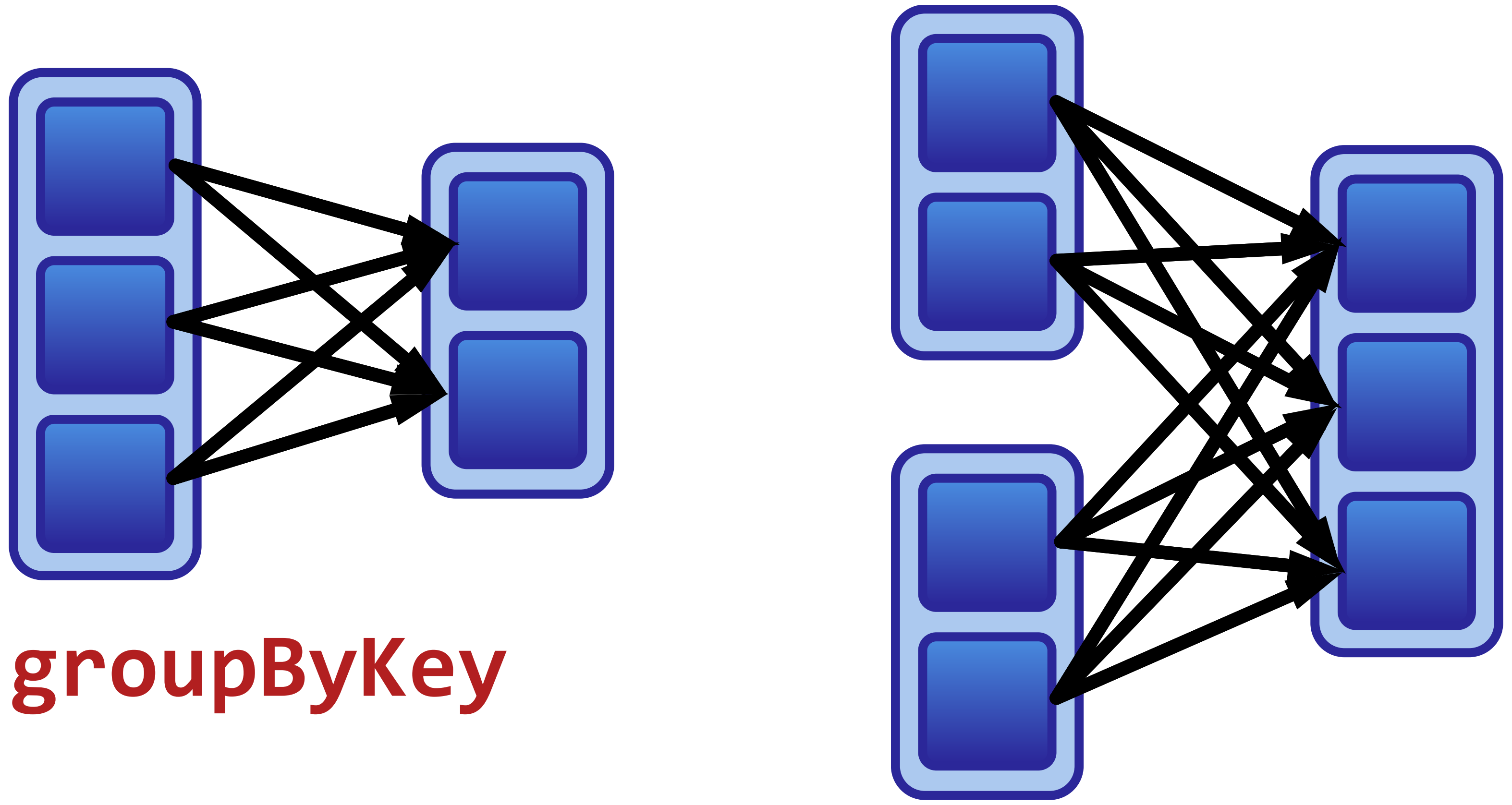


join *with co-partitioned inputs*

union

Wide dependencies:

Each partition of the parent RDD may be depended on by multiple child partitions.



groupByKey

join *with inputs not co-partitioned*

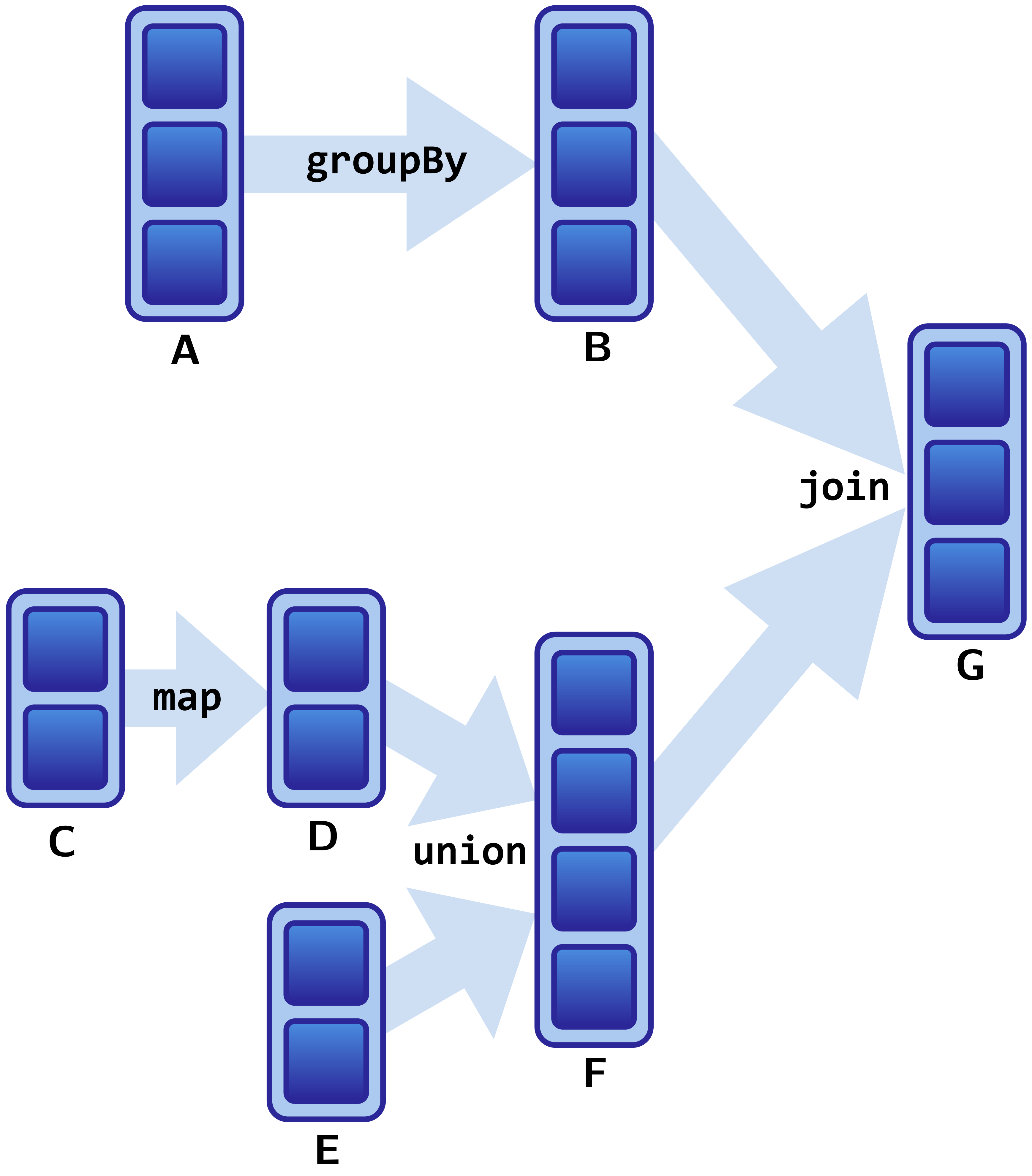
Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

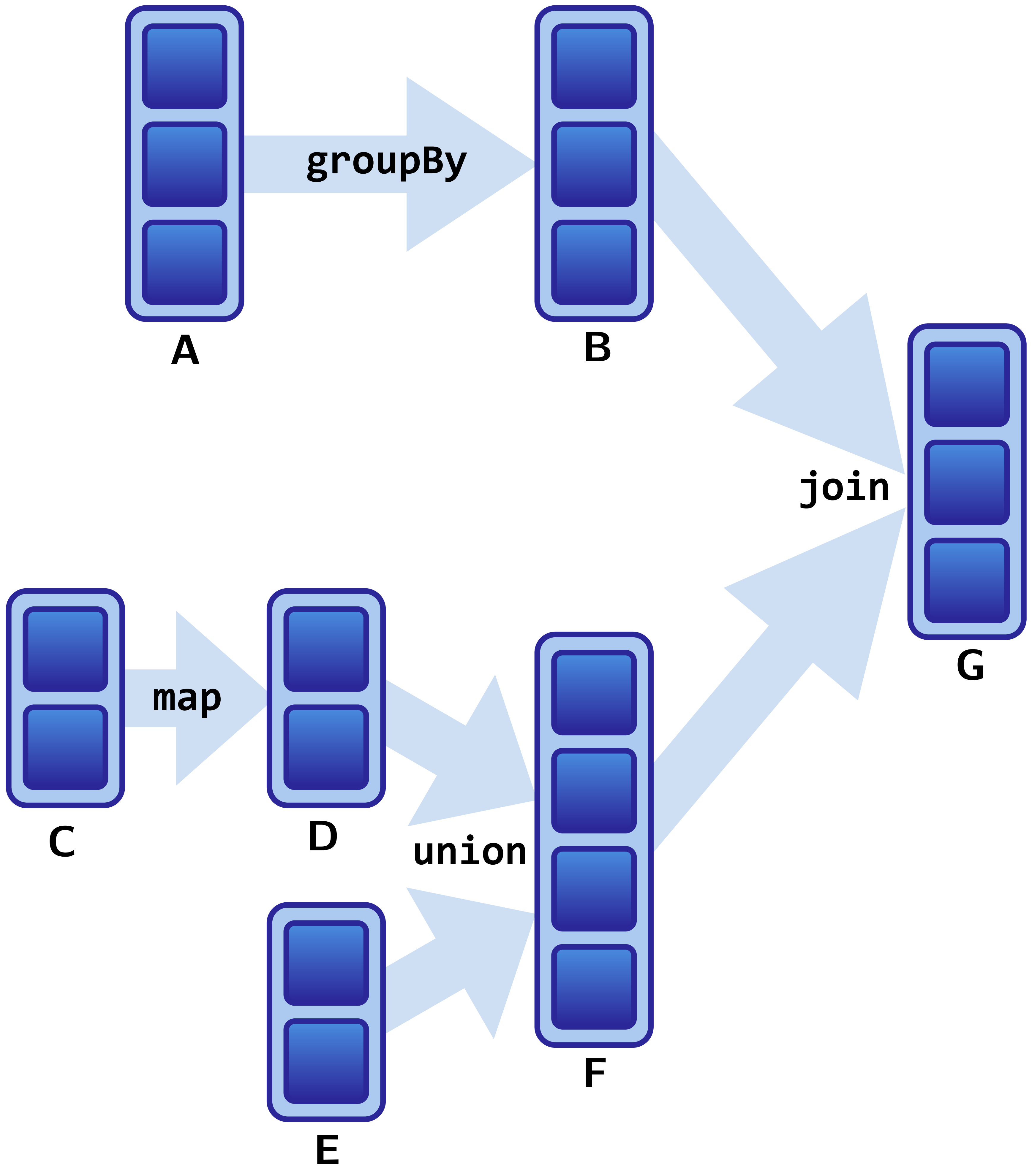
Conceptually assuming the DAG:



Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

Conceptually assuming the DAG:

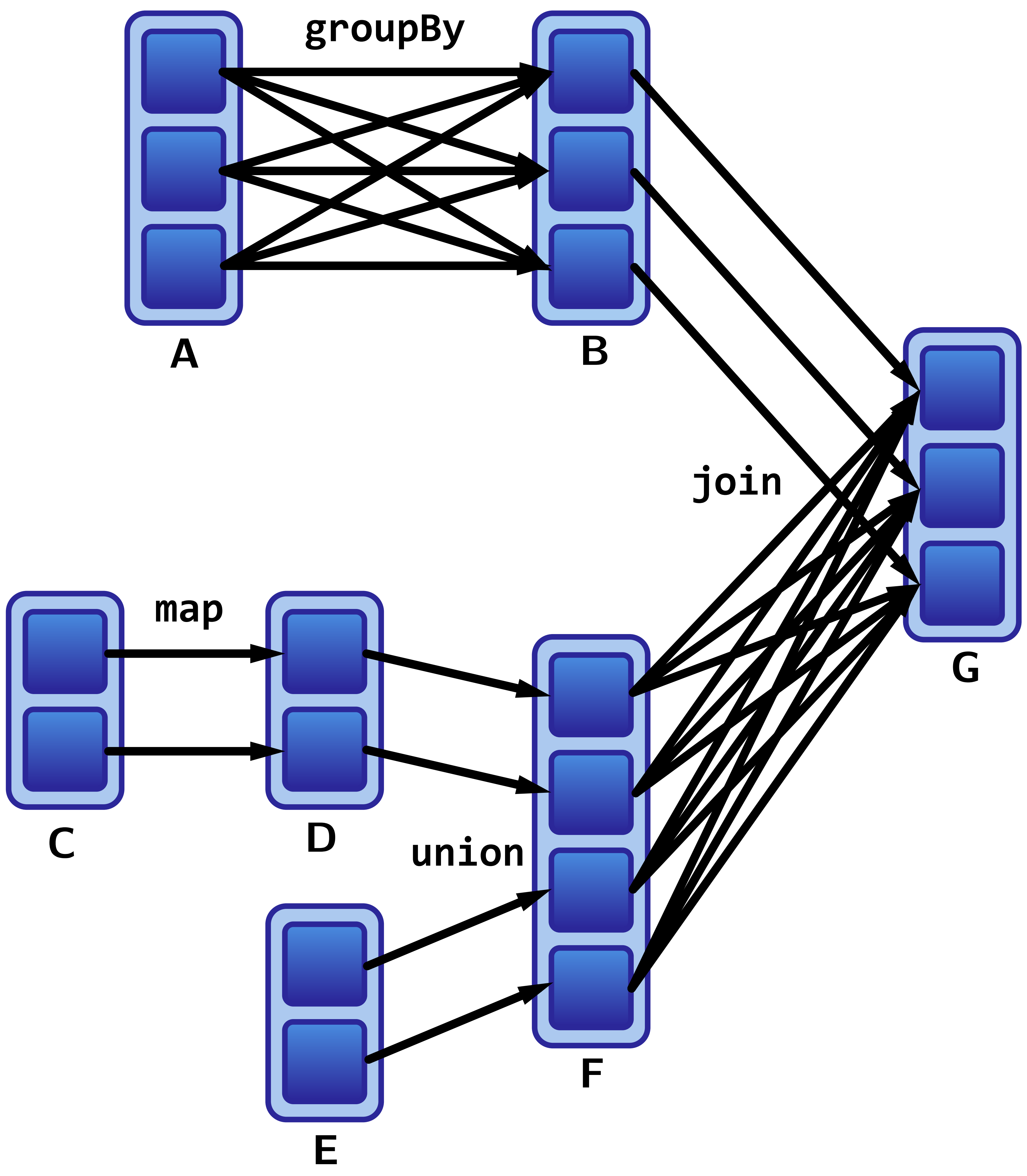


What do the dependencies look like?

Which dependencies are wide, and which are narrow?

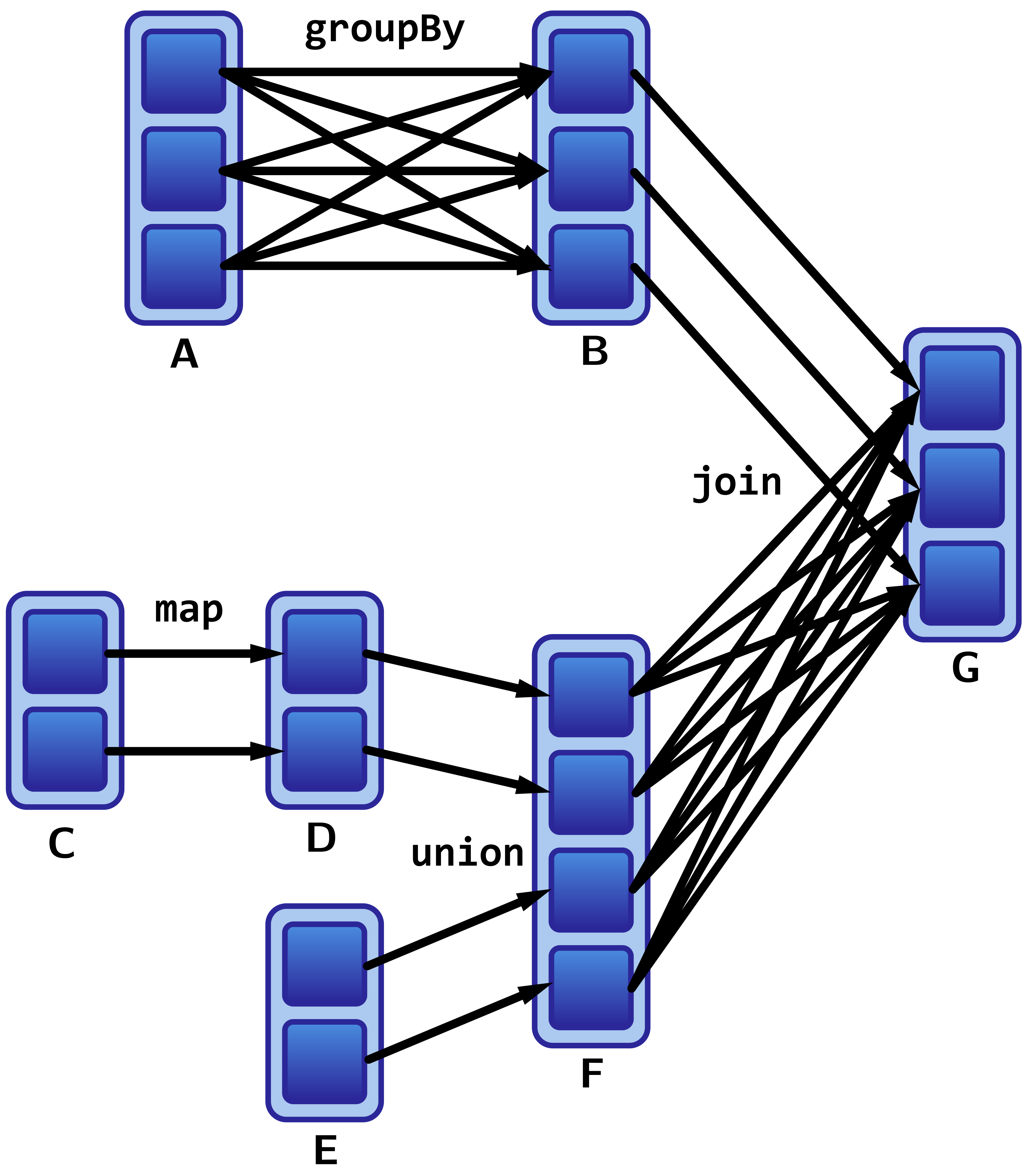
Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.



Narrow Dependencies vs Wide Dependences, Visually

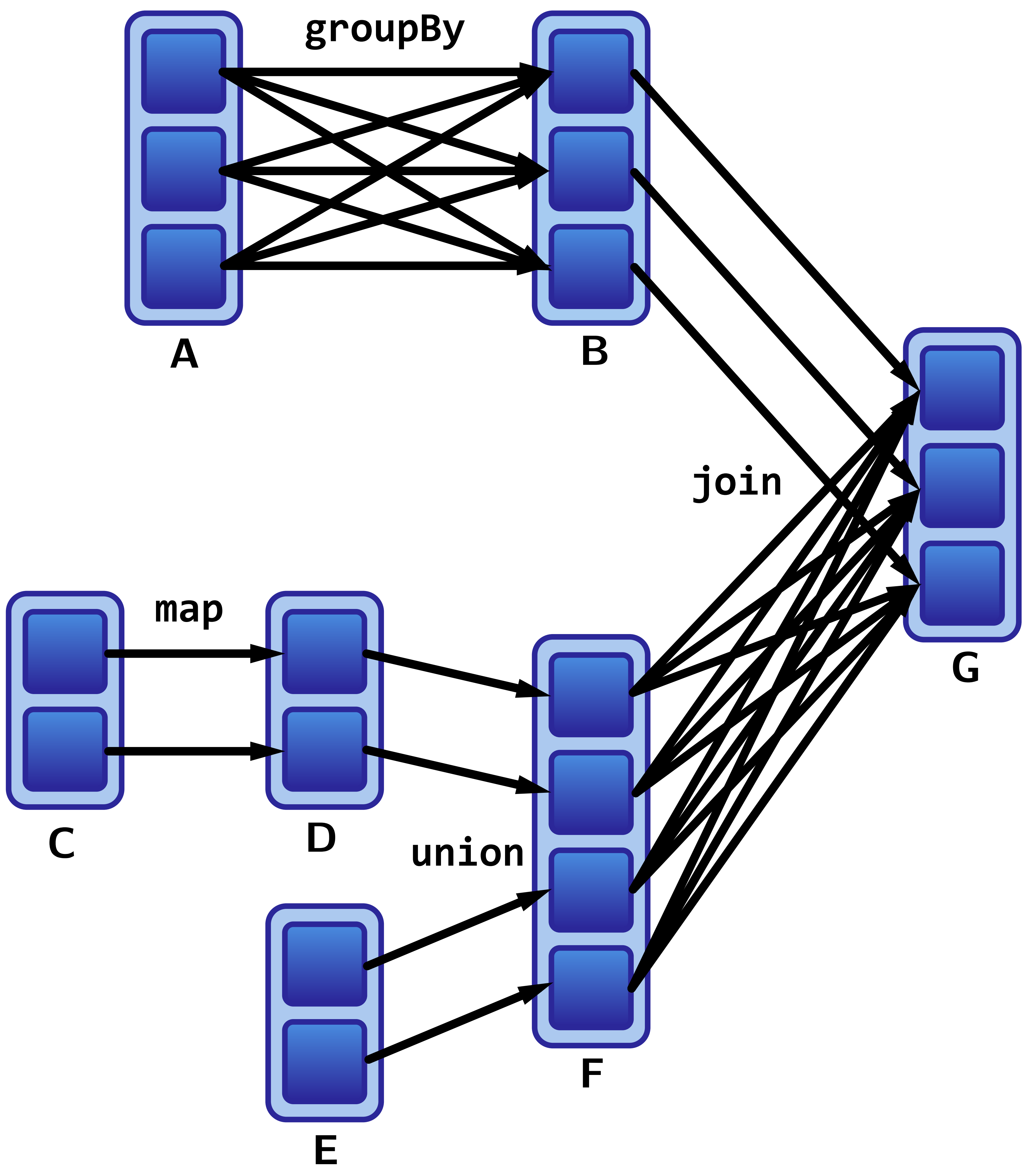
Let's visualize an example program and its dependencies.



Which dependencies are wide, and which are narrow?

Narrow Dependencies vs Wide Dependences, Visually

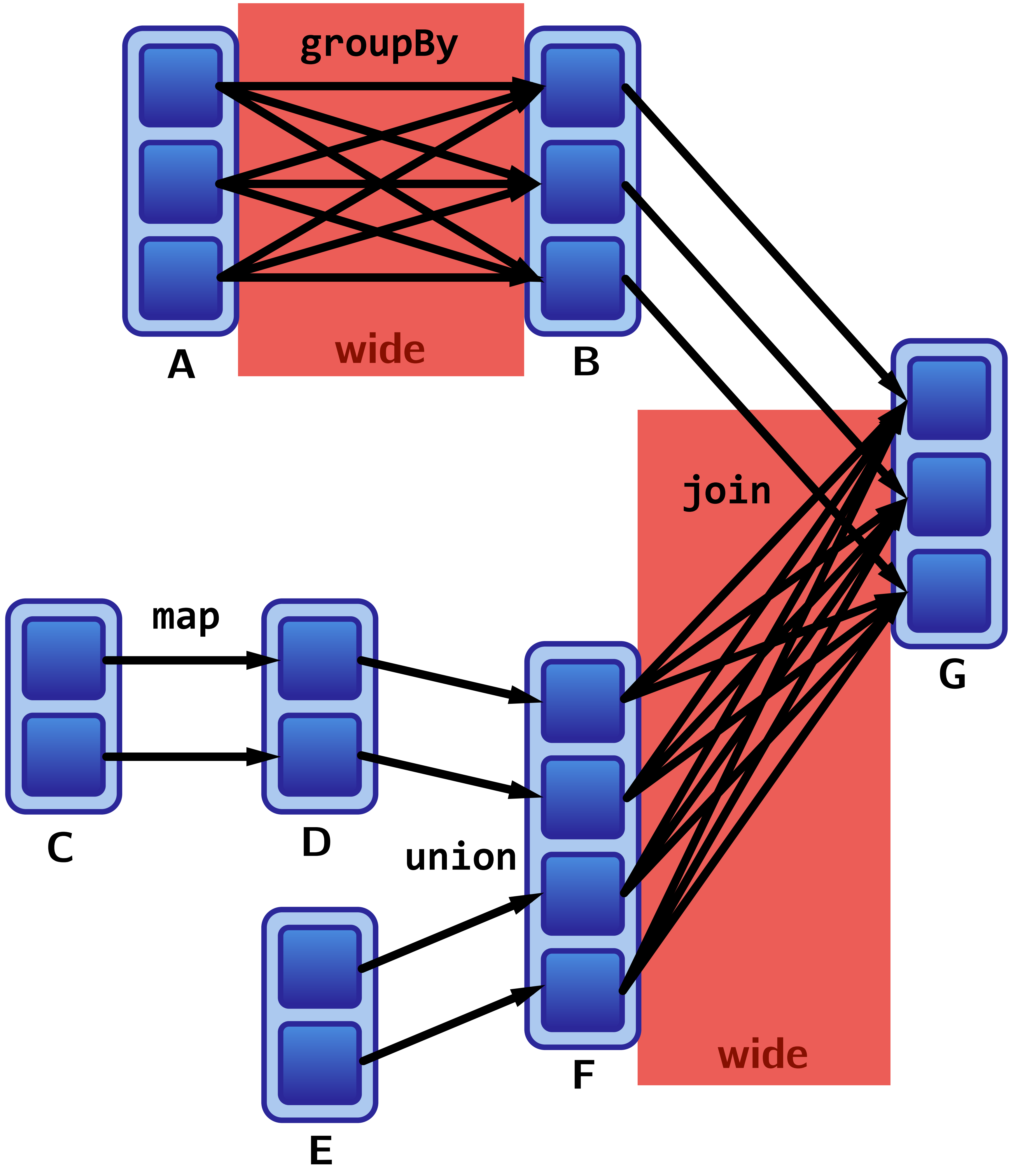
Let’s visualize an example program and its dependencies.



Narrow Dependencies vs Wide Dependences, Visually

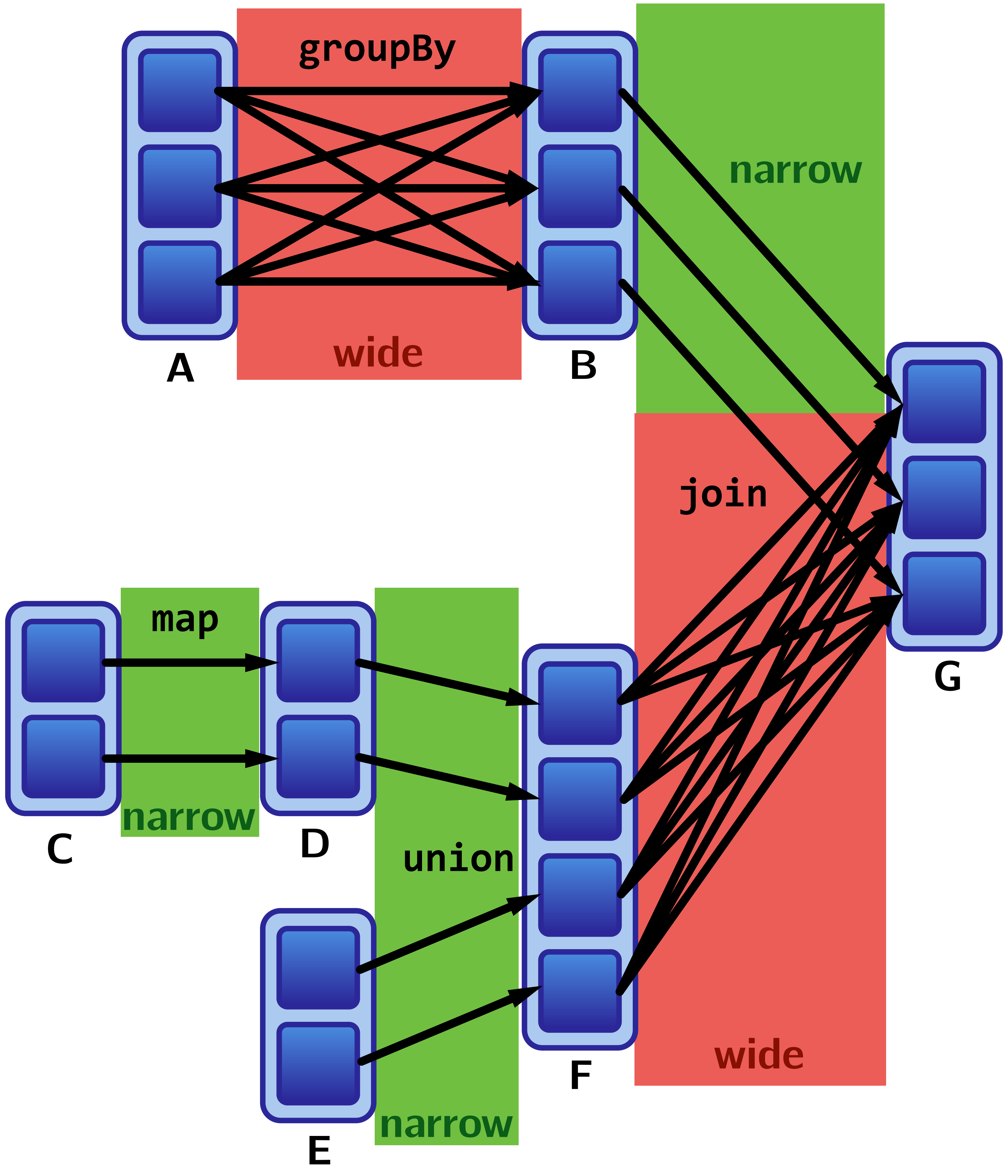
Let's visualize an example program and its dependencies.

Wide transformations:
`groupBy, join`



Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

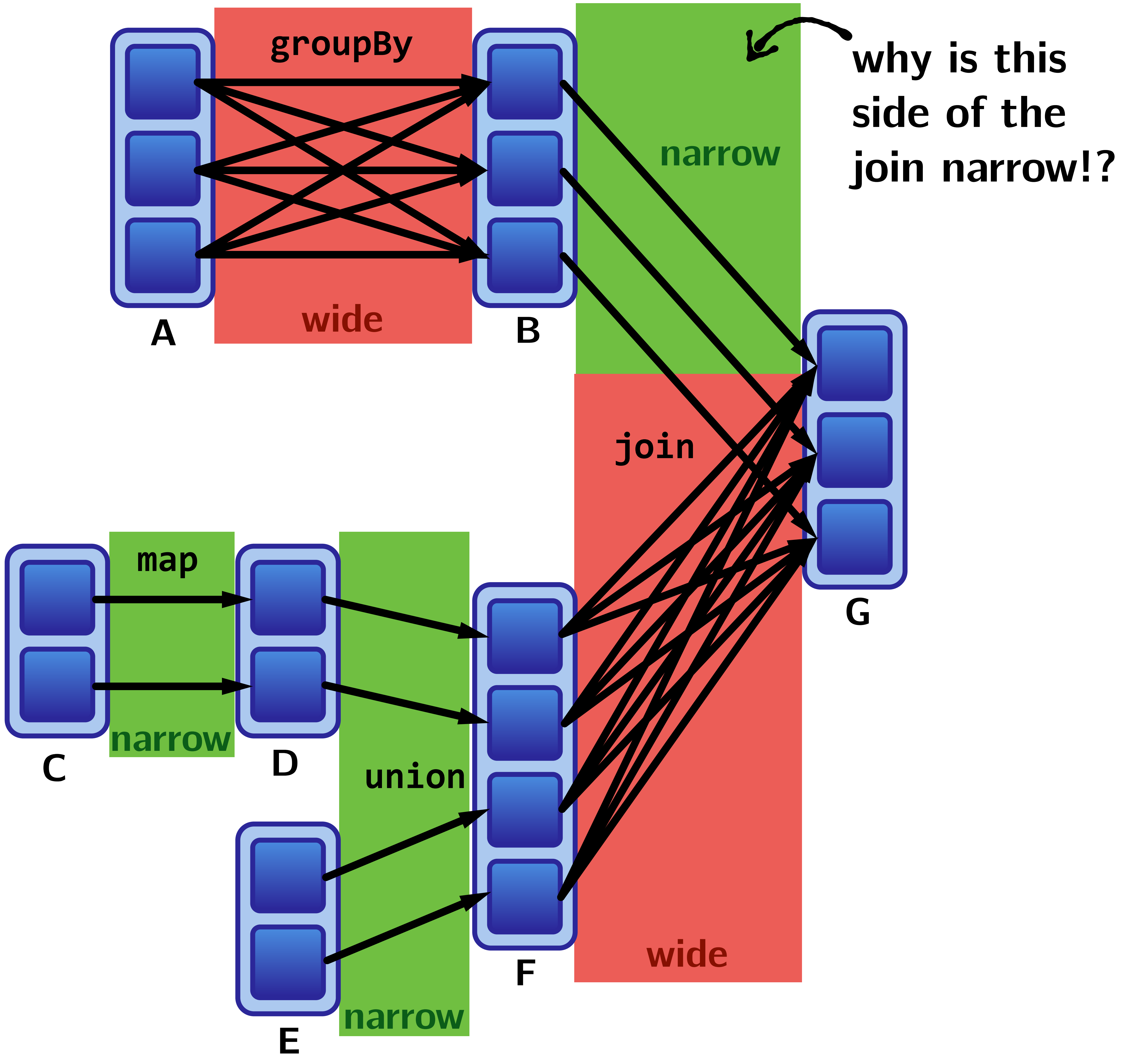


Wide transformations:
`groupBy, join`

Narrow transformations:
`map, union, join`

Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.



Wide transformations:
`groupBy, join`

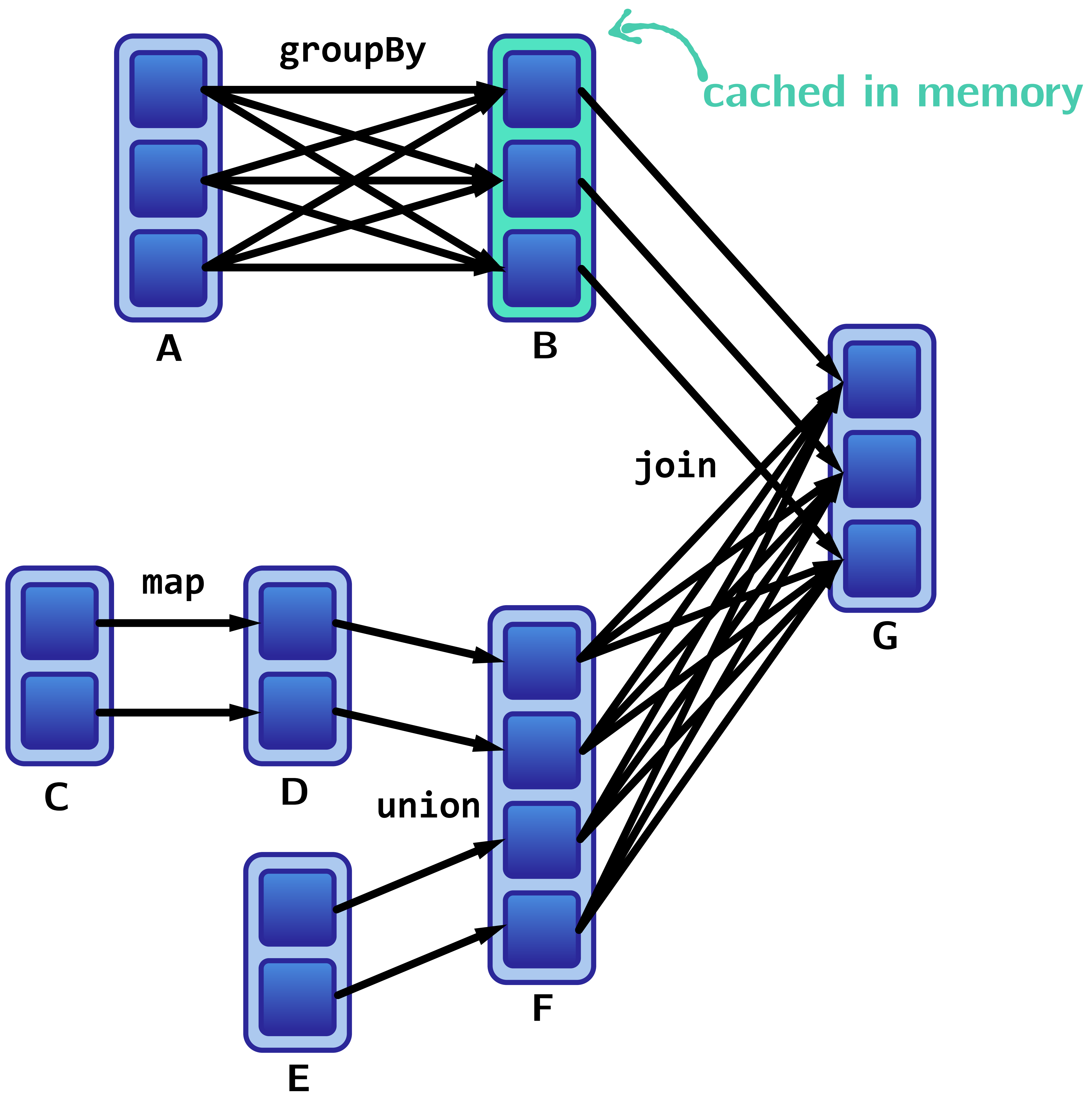
Narrow transformations:
`map, union, join` ⚠

Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

Since **G** would be derived from **B**, which itself is derived from a **groupBy** and a shuffle on **A**, you could imagine that we will have already co-partitioned and cached **B** in memory following the call to **groupBy**.

Part of this join is thus a narrow transformation.



Which transformations have which kind of dependency?

Transformations with narrow dependencies:

`map`
`mapValues`
`flatMap`
`filter`
`mapPartitions`
`mapPartitionsWithIndex`

Transformations with wide dependencies:
(might cause a shuffle)

`cogroup`
`groupWith`
`join`
`leftOuterJoin`
`rightOuterJoin`
`groupByKey`
`reduceByKey`
`combineByKey`
`distinct`
`intersection`
`repartition`
`coalesce`

How can I find out?

`dependencies` method on RDDs.

`dependencies` returns a sequence of `Dependency` objects, which are actually the dependencies used by Spark's scheduler to know how this RDD depends on other RDDs.

The sorts of dependency objects the `dependencies` method may return include:

Narrow dependency objects:

- `OneToOneDependency`
- `PruneDependency`
- `RangeDependency`

Wide dependency objects:

- `ShuffleDependency`

How can I find out?

`dependencies` method on RDDs.

`dependencies` returns a sequence of `Dependency` objects, which are actually the dependencies used by Spark's scheduler to know how this RDD depends on other RDDs.

```
val wordsRdd = sc.parallelize(largeList)
val pairs = wordsRdd.map(c => (c, 1))
                    .groupByKey()
                    .dependencies
// pairs: Seq[org.apache.spark.Dependency[_]] =
// List(org.apache.spark.ShuffleDependency@4294a23d)
```

How can I find out?

`toDebugString` method on RDDs.

`toDebugString` prints out a visualization of the RDD's lineage, and other information pertinent to scheduling. For example, indentations in the output separate groups of narrow transformations that may be pipelined together with wide transformations that require shuffles. These groupings are called *stages*.

```
val wordsRdd = sc.parallelize(largeList)
val pairs = wordsRdd.map(c => (c, 1))
                    .groupByKey()
                    .toDebugString
//pairs: String =
//(8) ShuffledRDD[219] at groupByKey at <console>:38 []
// +-(8) MapPartitionsRDD[218] at map at <console>:37 []
//    |  ParallelCollectionRDD[217] at parallelize at <console>:36 []
```

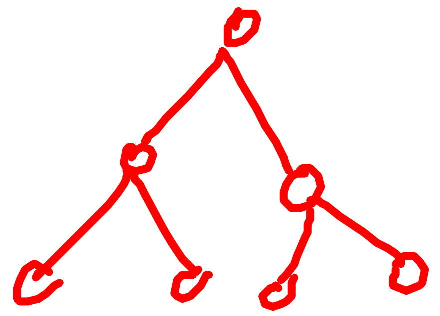
Lineages and Fault Tolerance

Lineages graphs are the key to fault tolerance in Spark.

Lineages and Fault Tolerance

Lineages graphs are the key to fault tolerance in Spark.

Ideas from **functional programming** enable fault tolerance in Spark:



- RDDs are immutable.
- We use higher-order functions like `map`, `flatMap`, `filter` to do *functional* transformations on this immutable data.
- A function for computing the dataset based on its parent RDDs also is part of an RDD's representation.

Lineages and Fault Tolerance

Fault tolerance w/out having to checkpoint & write data to disk!

Lineages graphs are the key to fault tolerance in Spark.

Ideas from **functional programming** enable fault tolerance in Spark:

- RDDs are immutable.
- We use higher-order functions like `map`, `flatMap`, `filter` to do *functional* transformations on this immutable data.
- A function for computing the dataset based on its parent RDDs also is part of an RDD's representation.

Along with keeping track of dependency information between partitions as well, this allows us to:

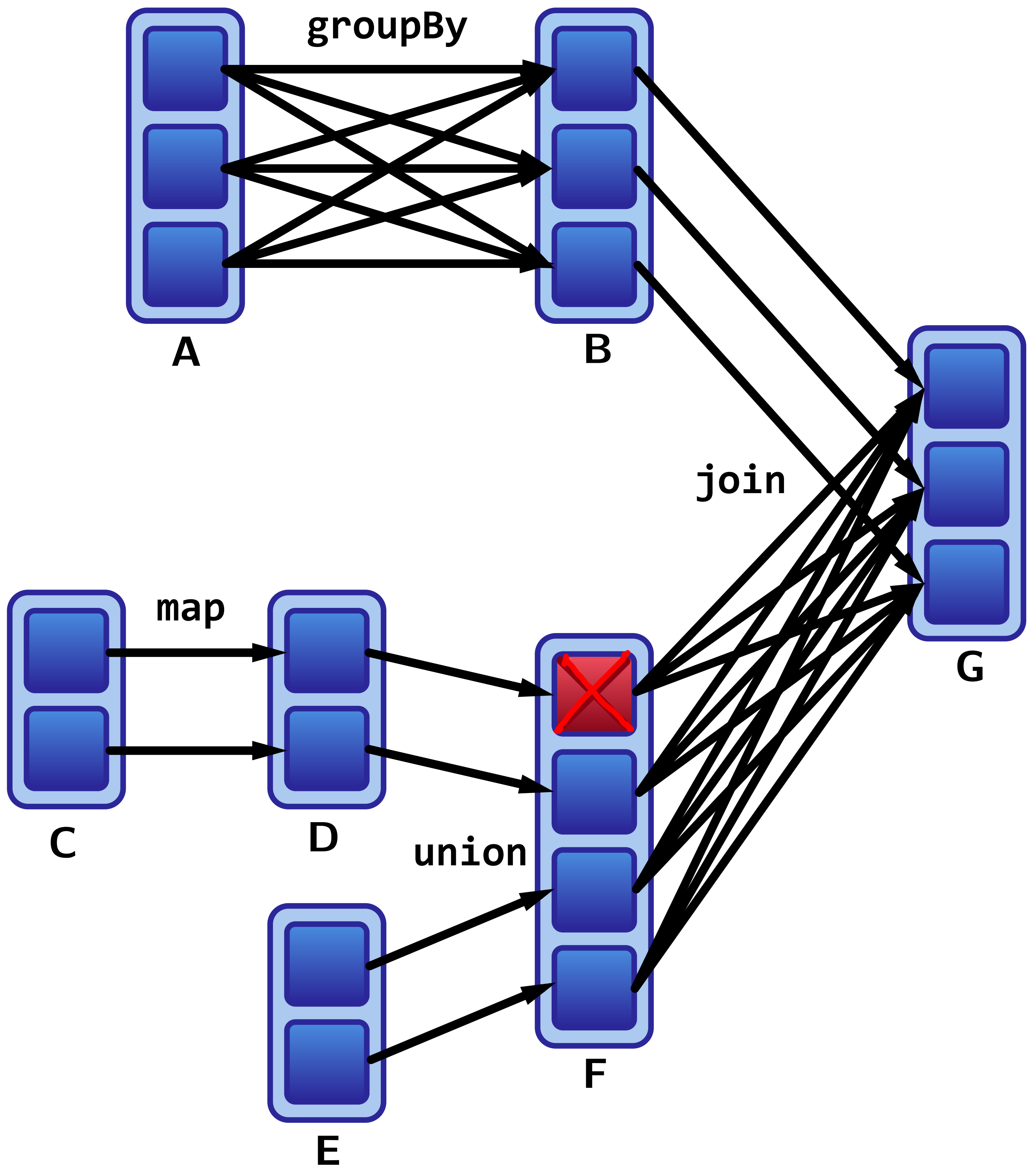
in-memory + fault tolerant!

Recover from failures by recomputing lost partitions from lineage graphs.

Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

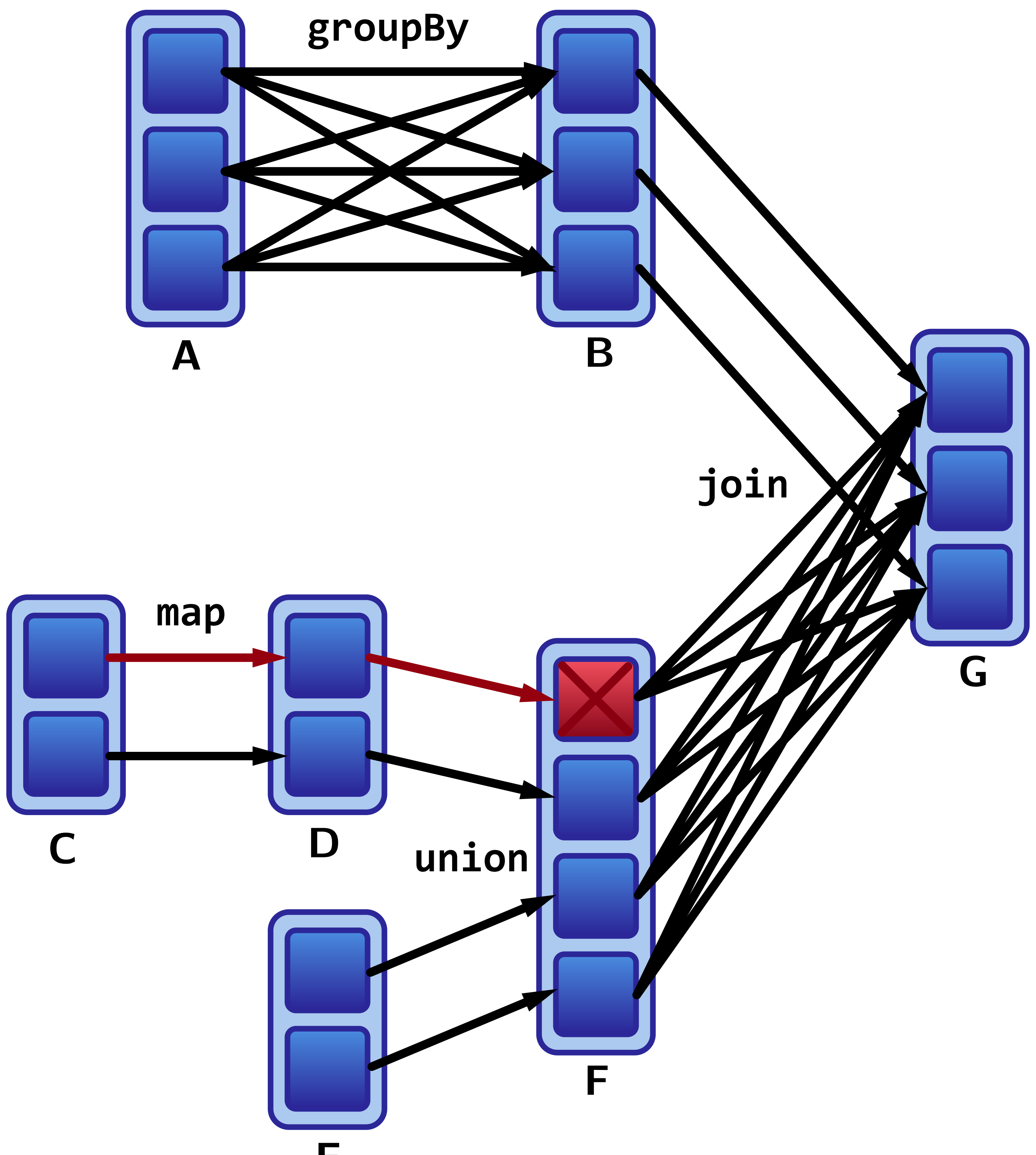
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

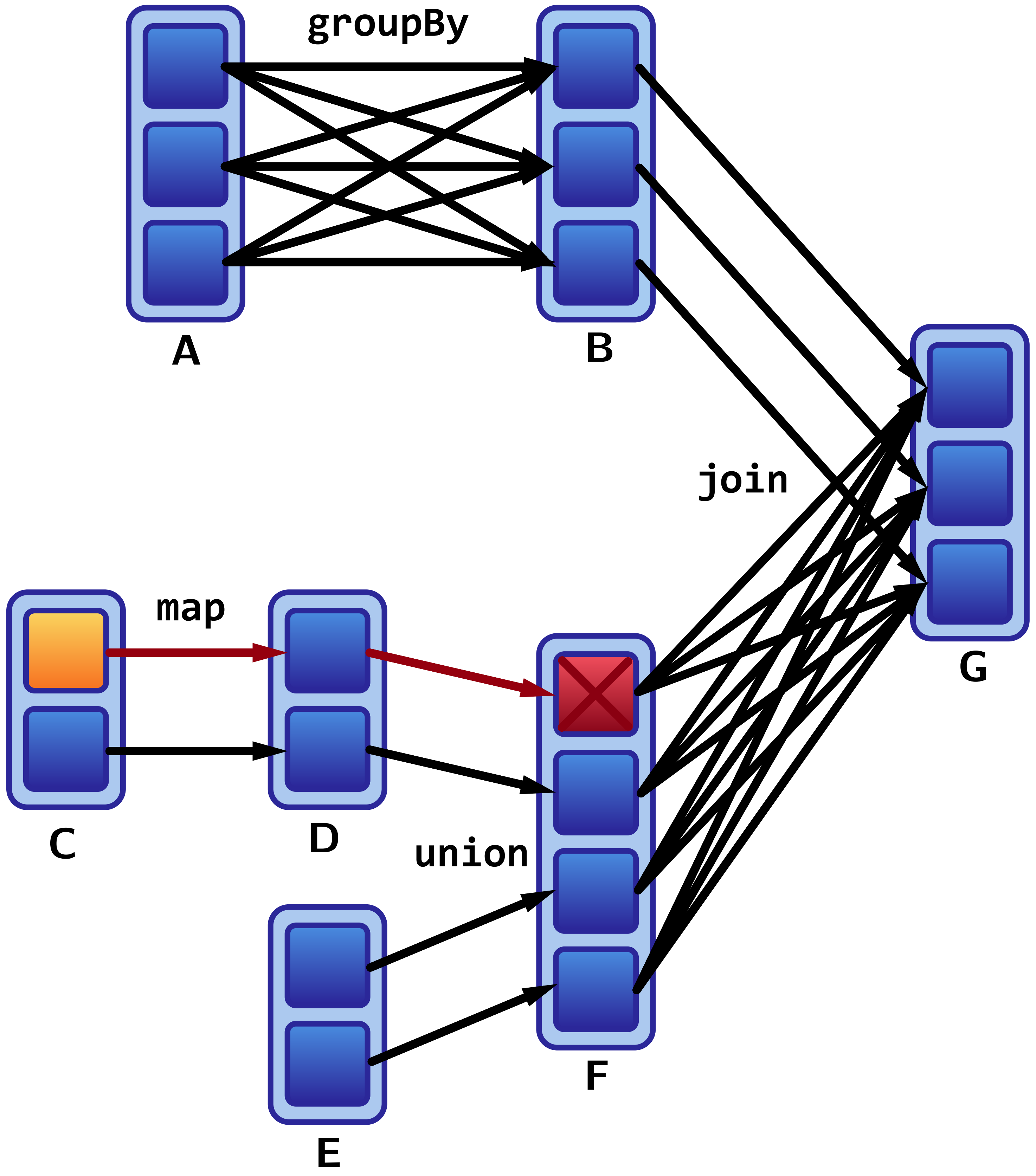
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

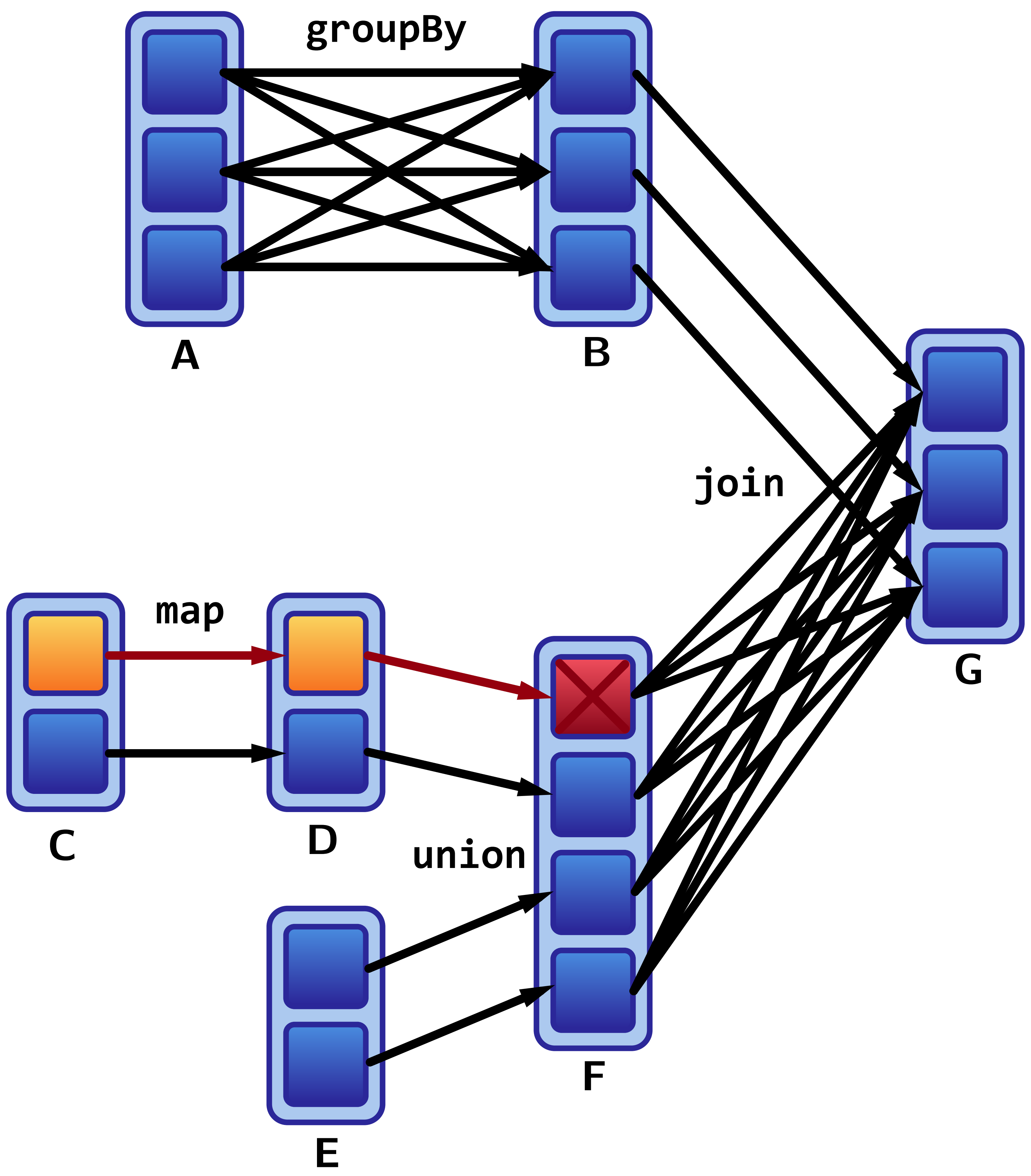
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

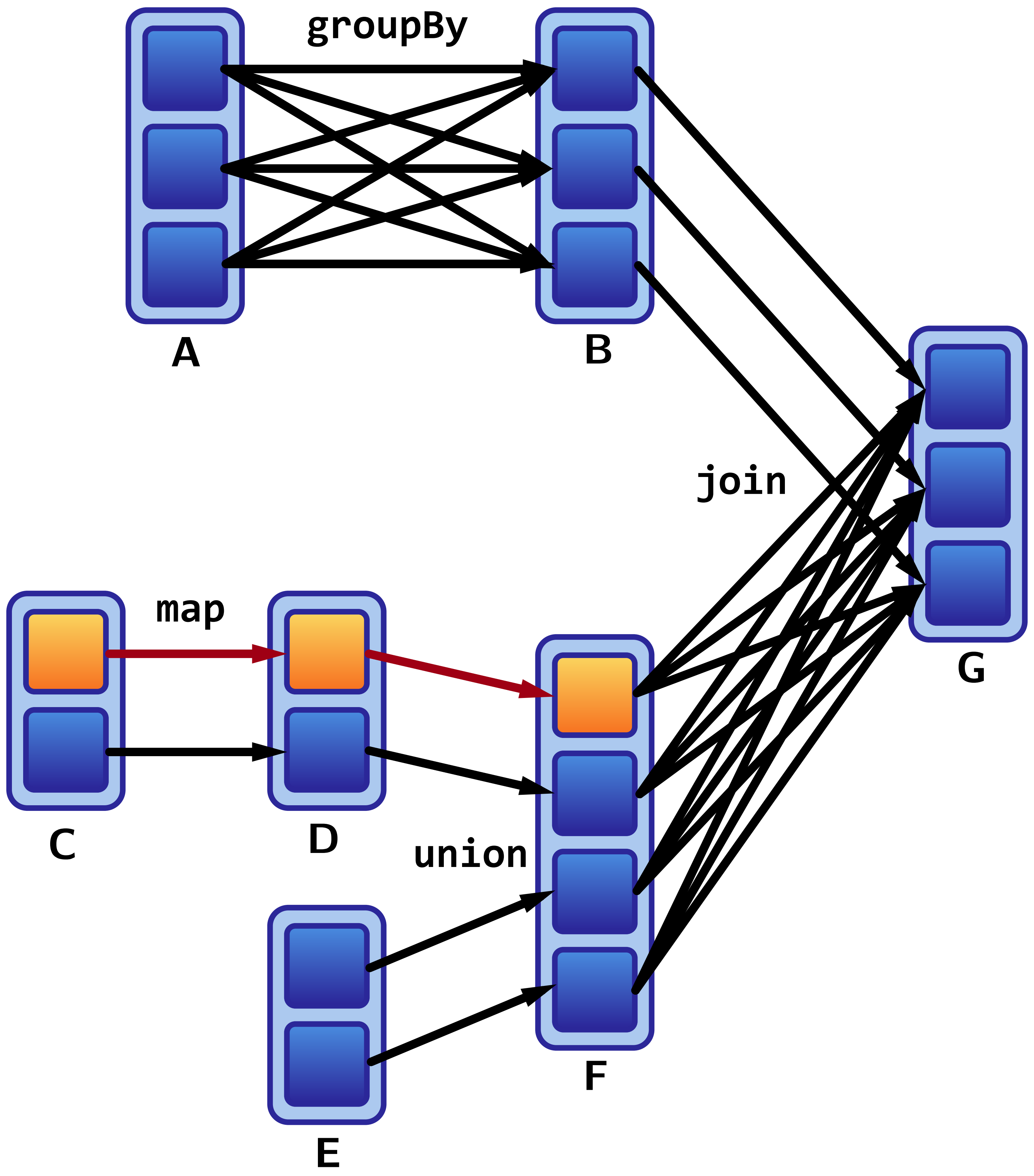
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

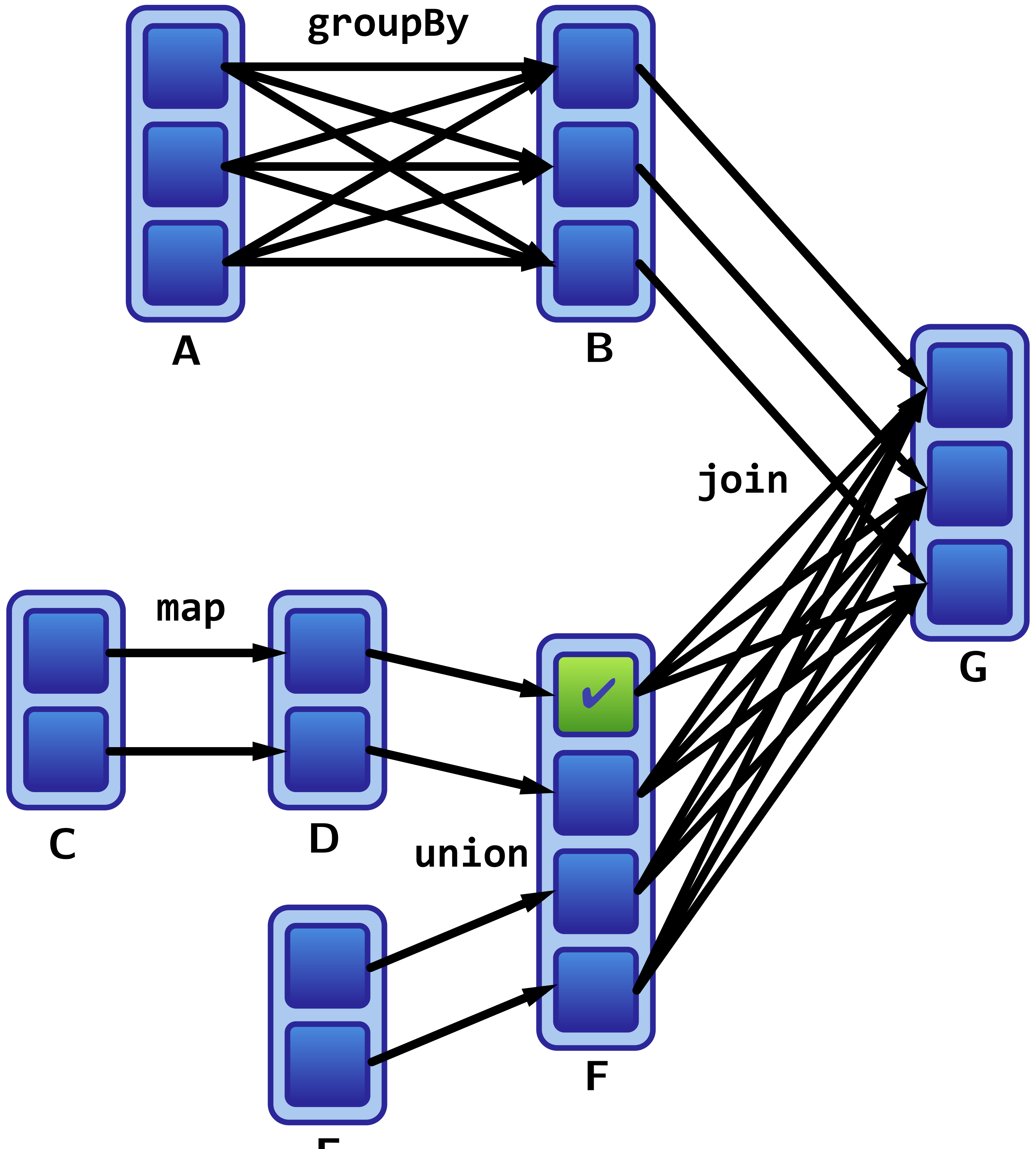
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

Recomputing missing partitions fast for narrow dependencies. But slow for wide dependencies!

Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

Recomputing missing partitions fast for narrow dependencies. But slow for wide dependencies!