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

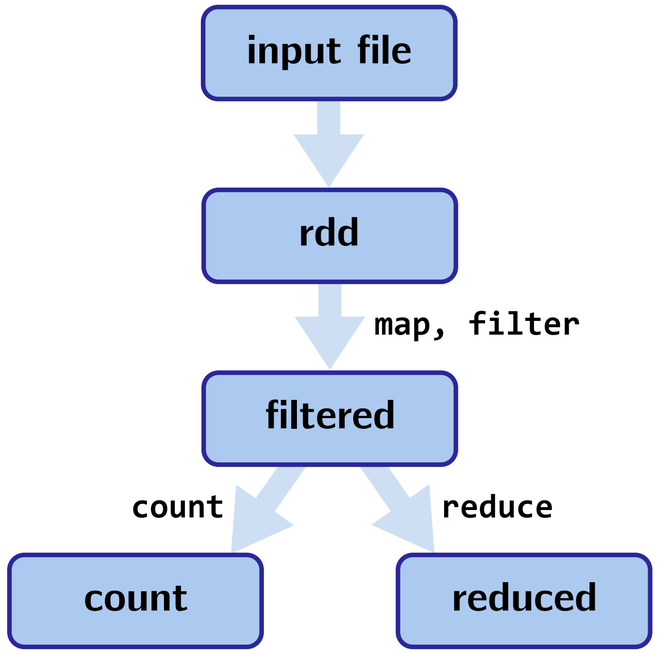
Computations on RDDs are represented as a **lineage graph**; a Directed Acyclic Graph (DAG) representing the computations done on the RDD.

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(...)
```

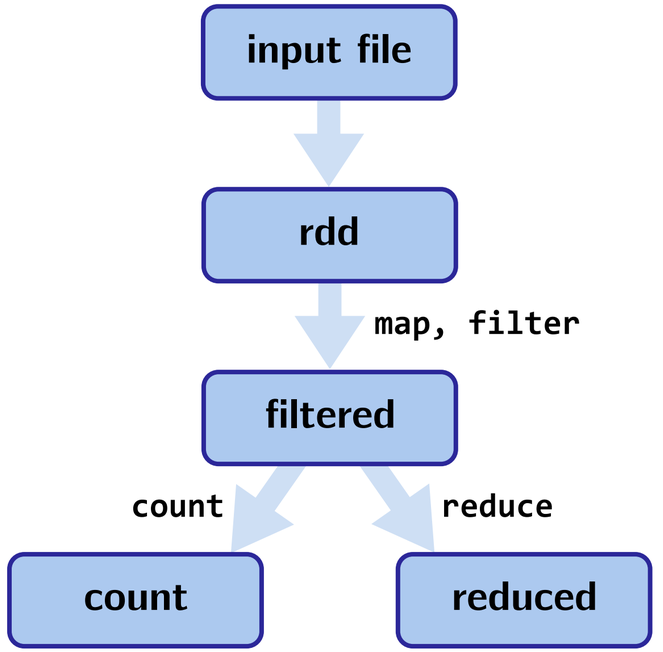


Lineages

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

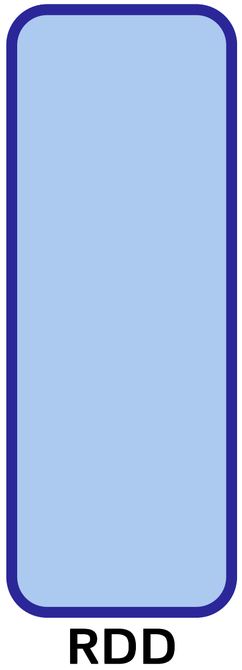
In fact, this is the representation/DAG is what Spark analyzes to do optimizations.

How are RDDs represented?

RDDs are made up of 2 important parts.

(but are made up of 4 parts in total)

RDDs are represented as:

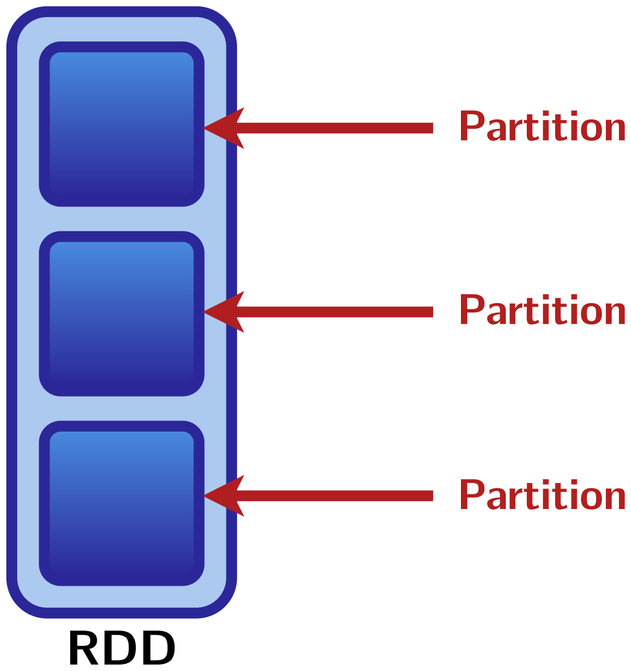


RDD

How are RDDs represented?

RDDs are made up of 2 important parts.

(but are made up of 4 parts in total)



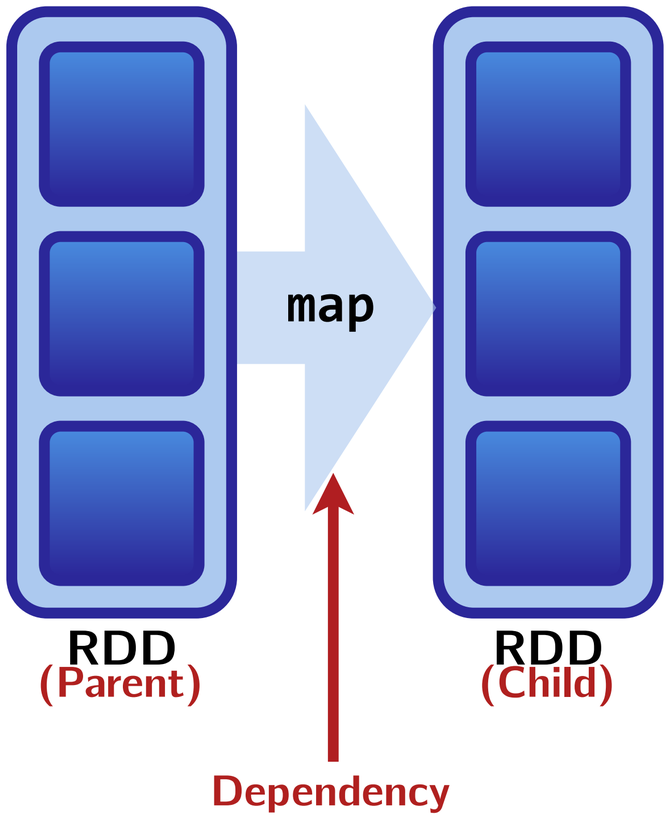
RDDs are represented as:

- **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)



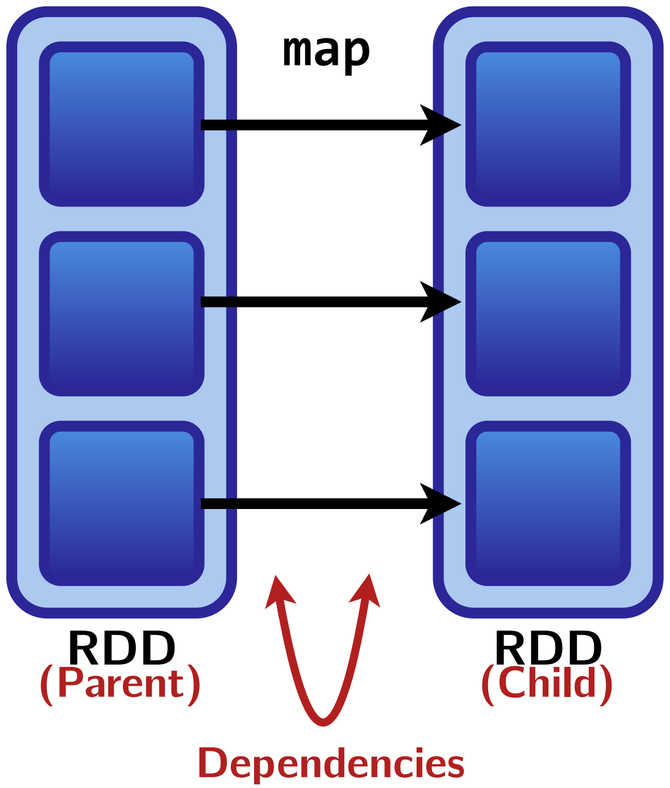
RDDs are represented as:

- **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)



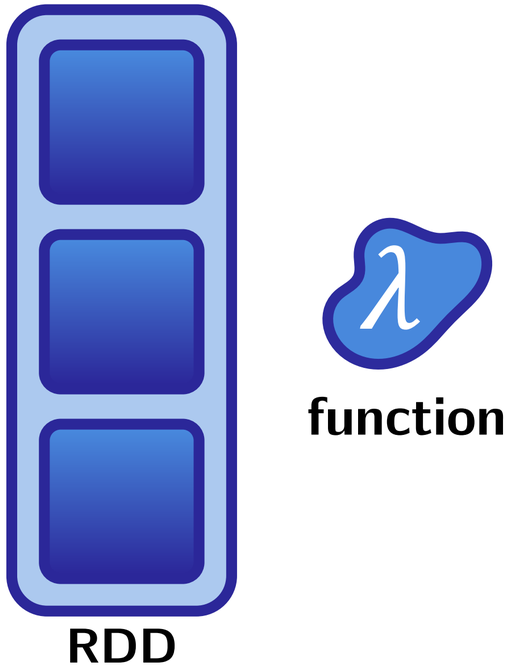
RDDs are represented as:

- **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)



function

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**

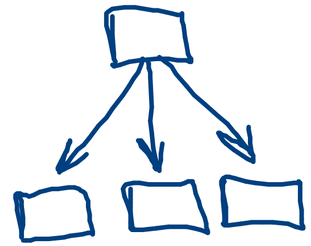
Narrow Dependencies vs Wide Dependences

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

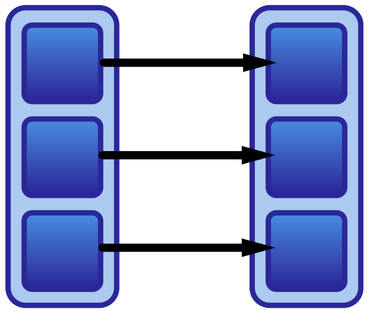
Narrow dependencies:

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.

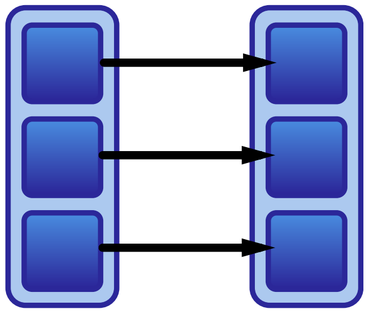


`map, filter`

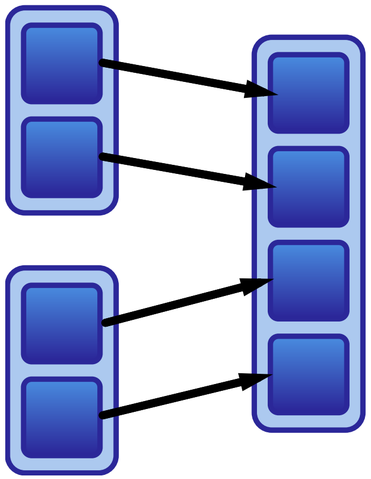
Narrow Dependencies vs Wide Dependences, Visually

Narrow dependencies:

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



`map, filter`

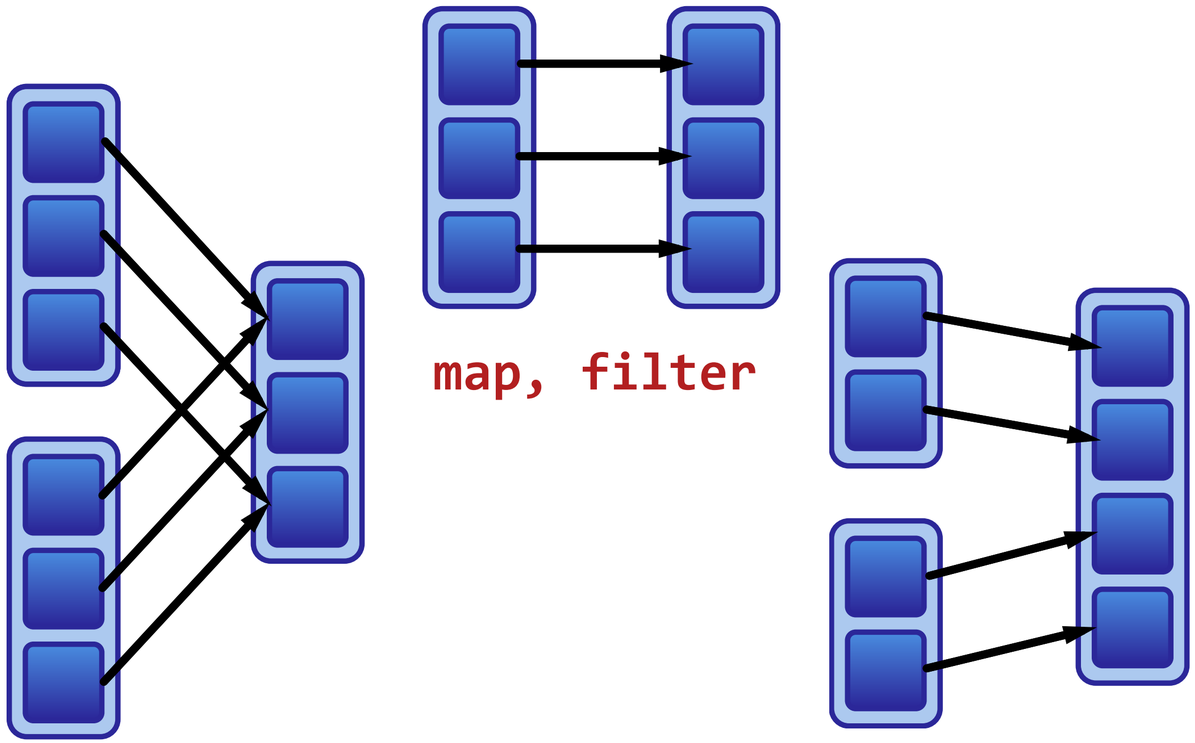


`union`

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`

`union`

`join` *with*
co-partitioned inputs

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.

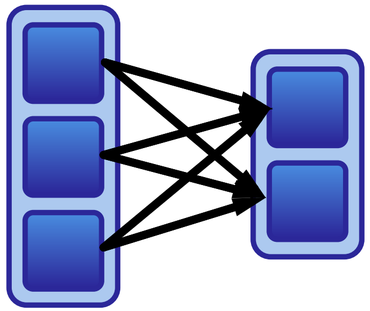
`join` *with co-partitioned inputs*

`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.

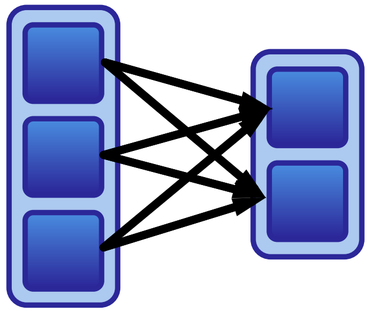
`join` *with co-partitioned inputs*

`map, filter`

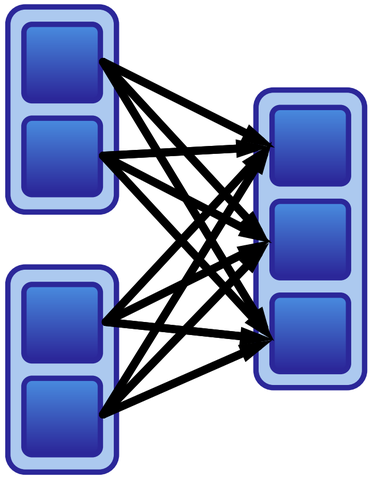
`union`

Wide dependencies:

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



`groupByKey`



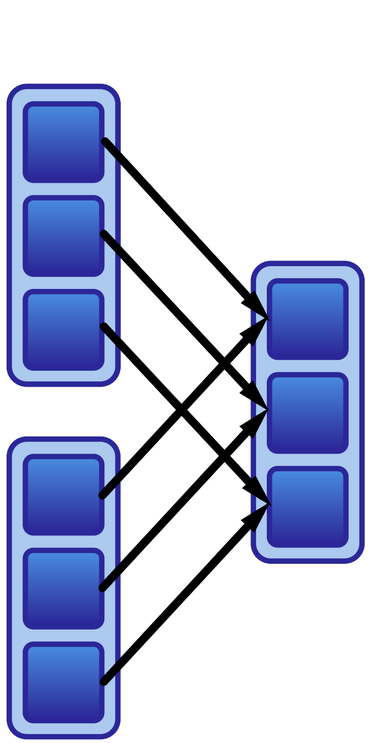
`join`

with inputs not co-partitioned

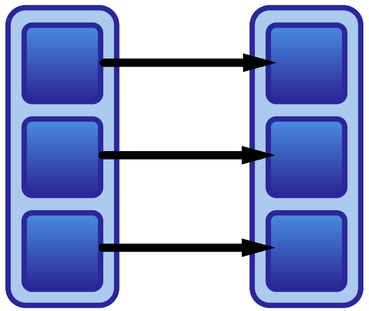
Narrow Dependencies vs Wide Dependences, Visually

Narrow dependencies:

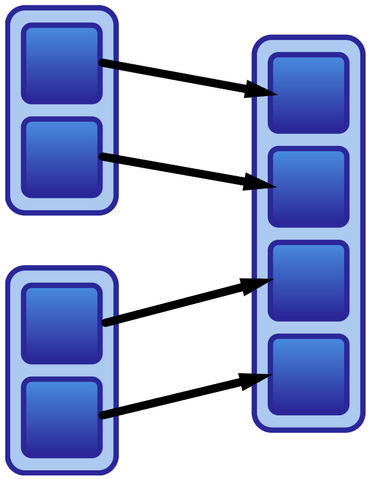
Each partition of the parent RDD is used by at most one partition of the child RDD.



join *with co-partitioned inputs*



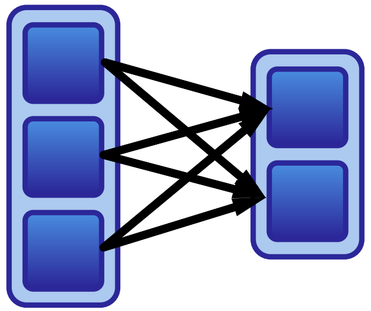
map, filter



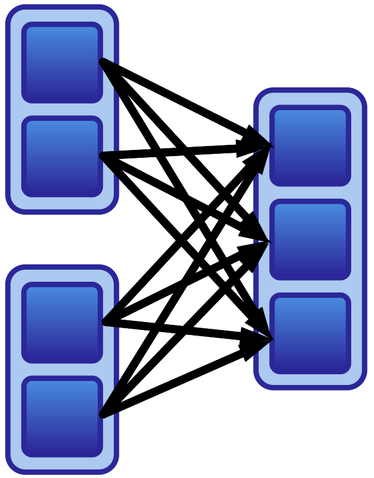
union

Wide dependencies:

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



groupByKey



join *with inputs not co-partitioned*

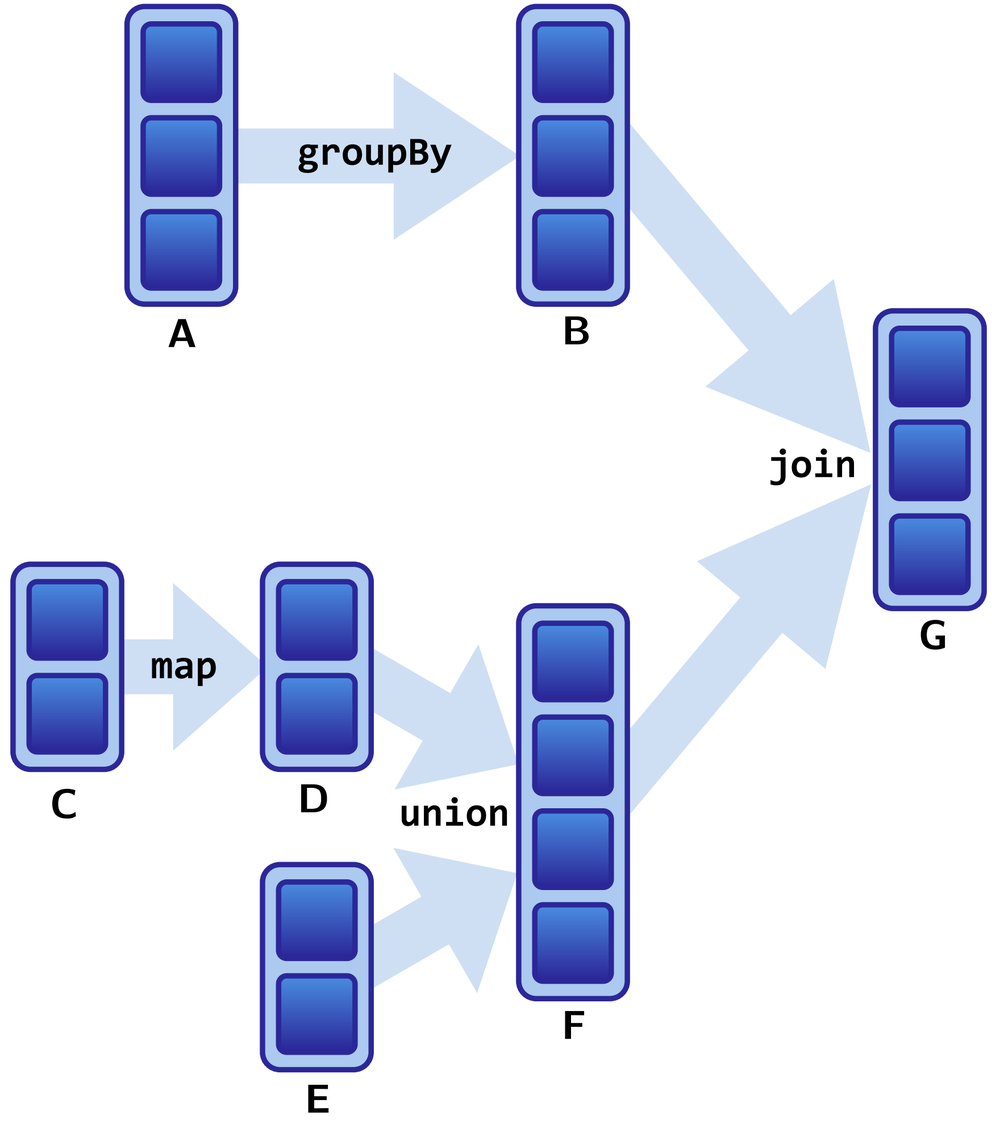
Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example
program and its dependencies.

Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

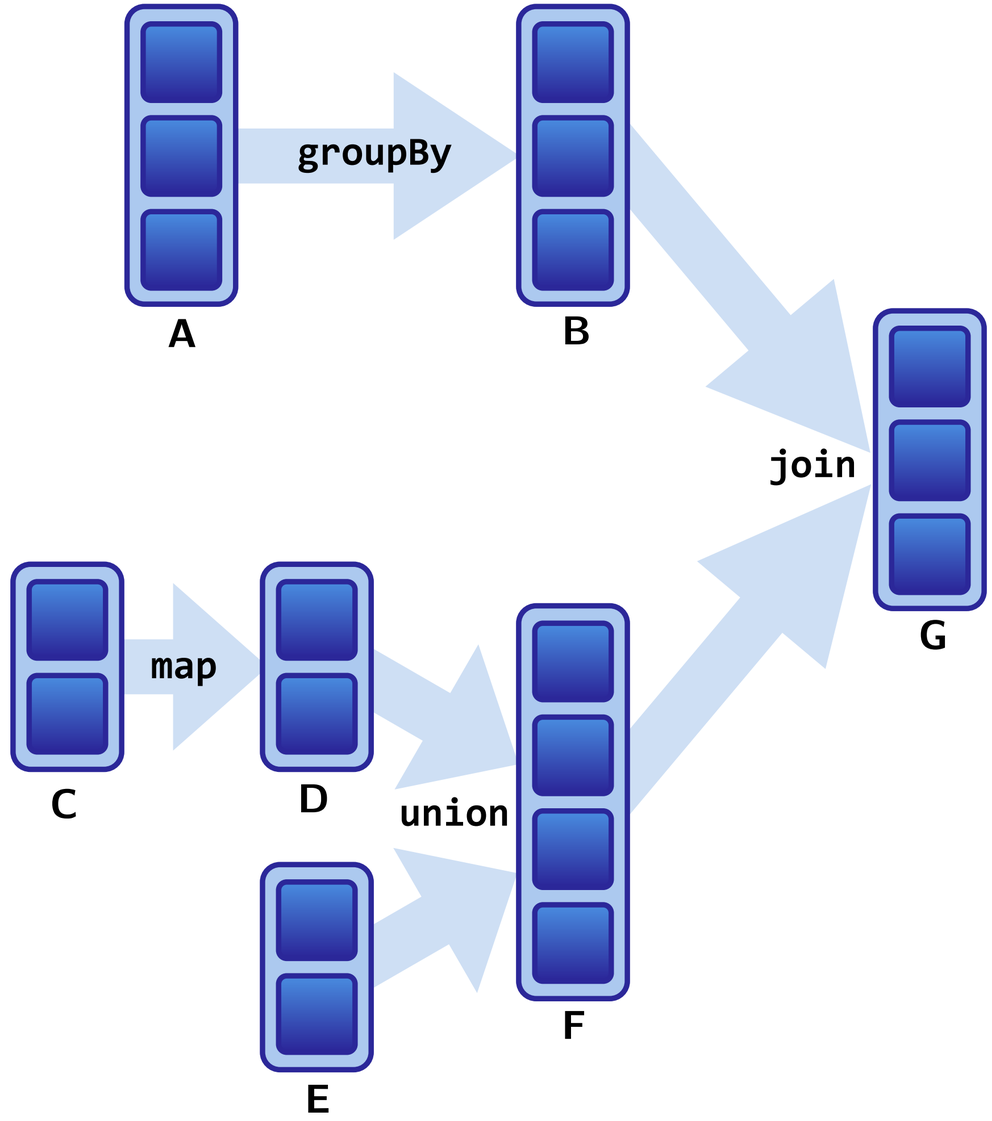
Conceptually assuming the DAG:



Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

Conceptually assuming the DAG:

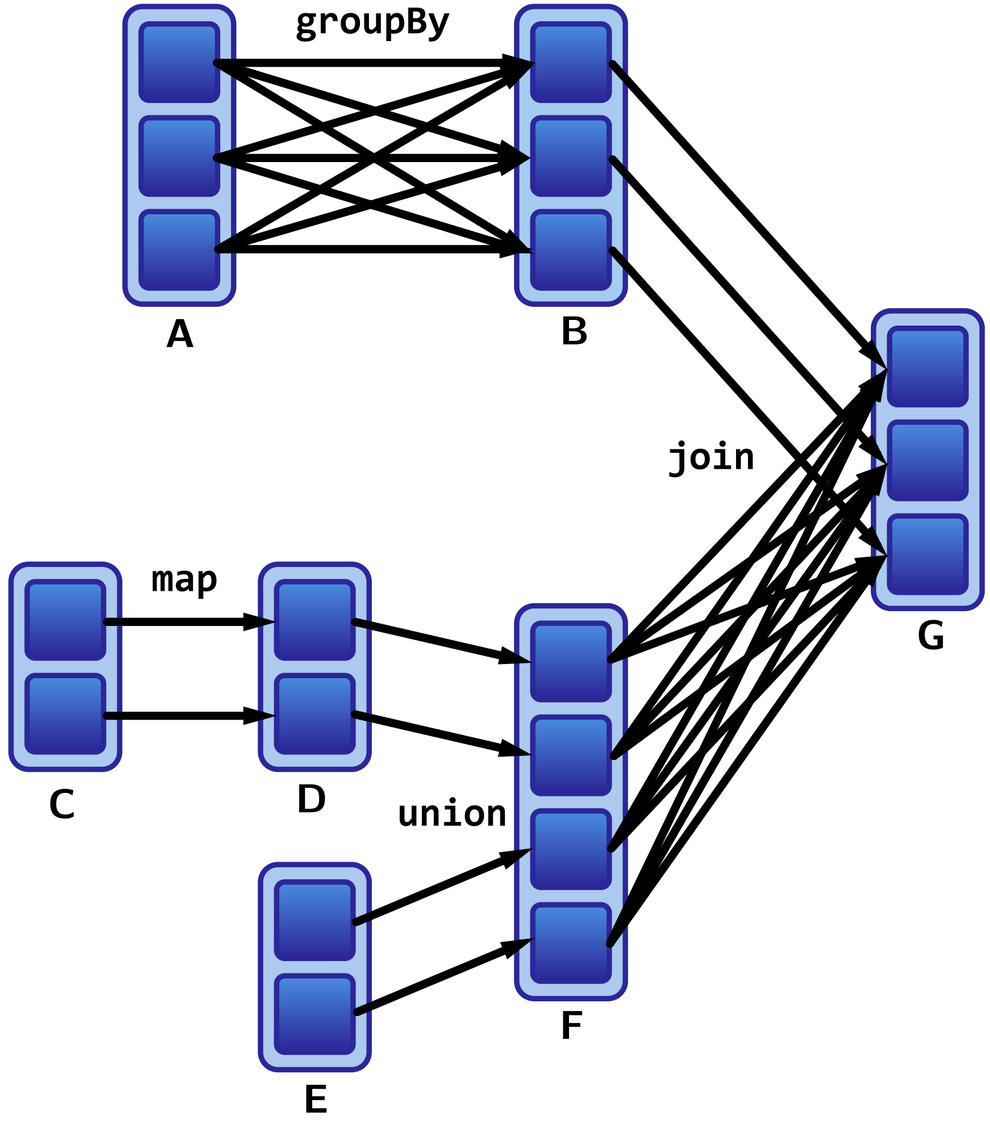


What do the dependencies look like?

Which dependencies are wide, and which are narrow?

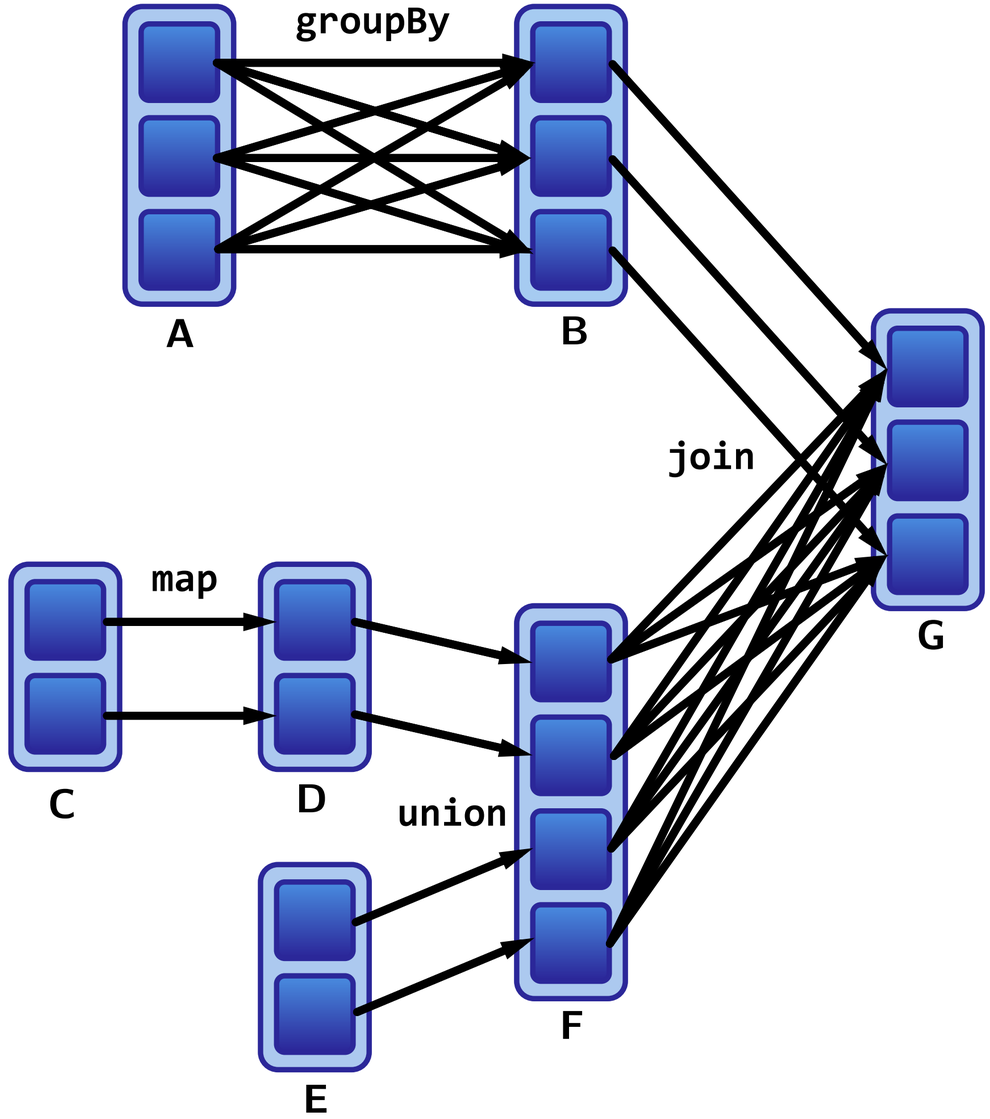
Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.



Narrow Dependencies vs Wide Dependences, Visually

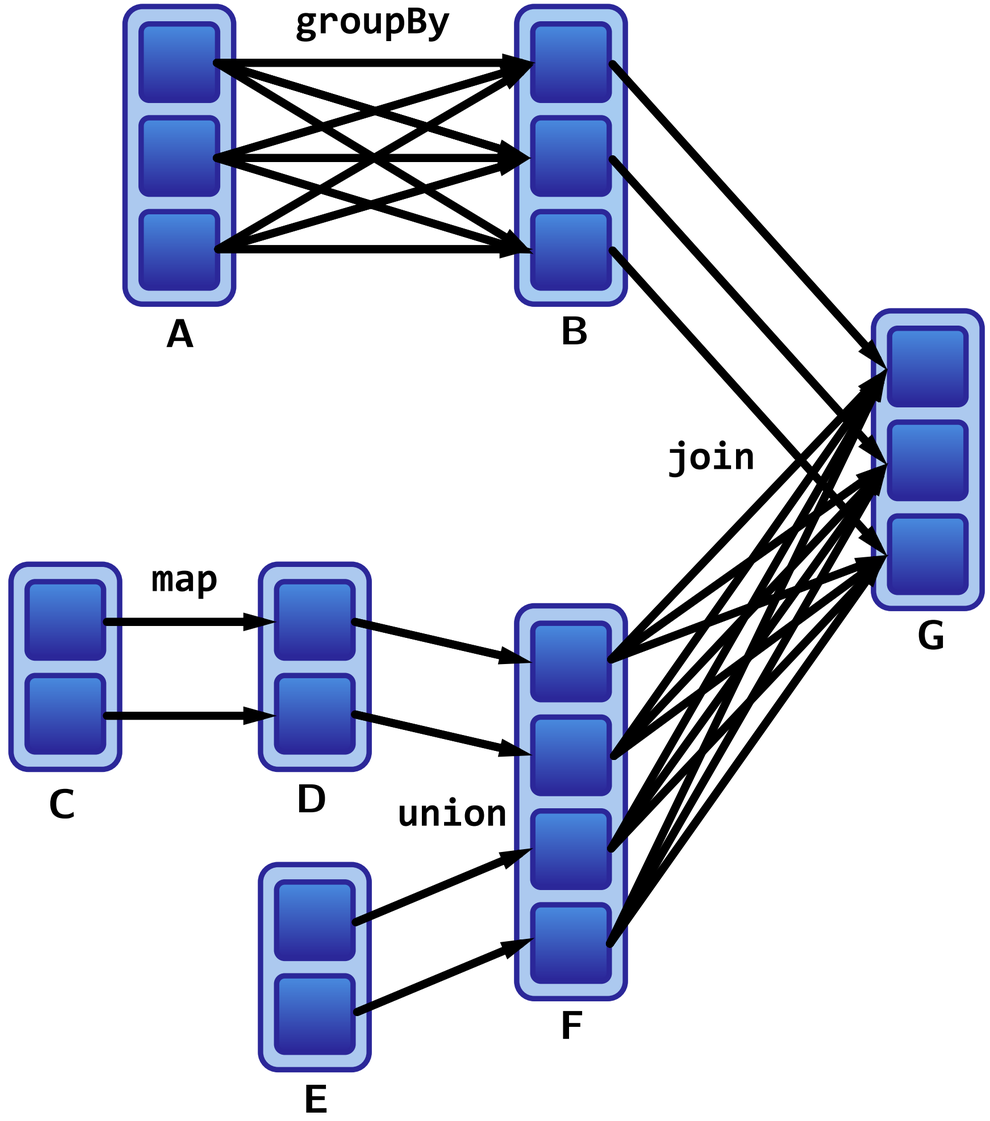
Let's visualize an example program and its dependencies.



Which dependencies are wide, and which are narrow?

Narrow Dependencies vs Wide Dependences, Visually

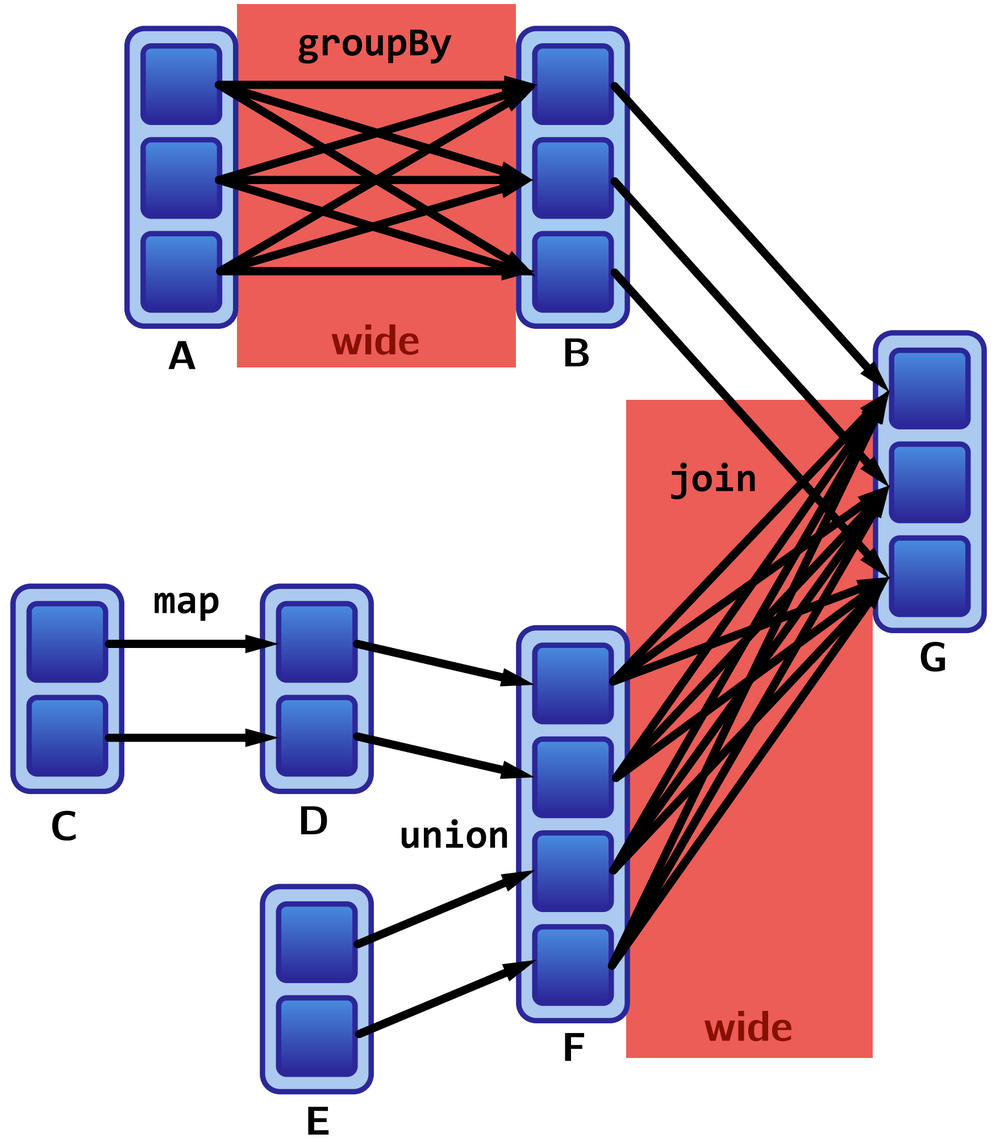
Let's visualize an example program and its dependencies.



Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

Wide transformations:
`groupBy, join`

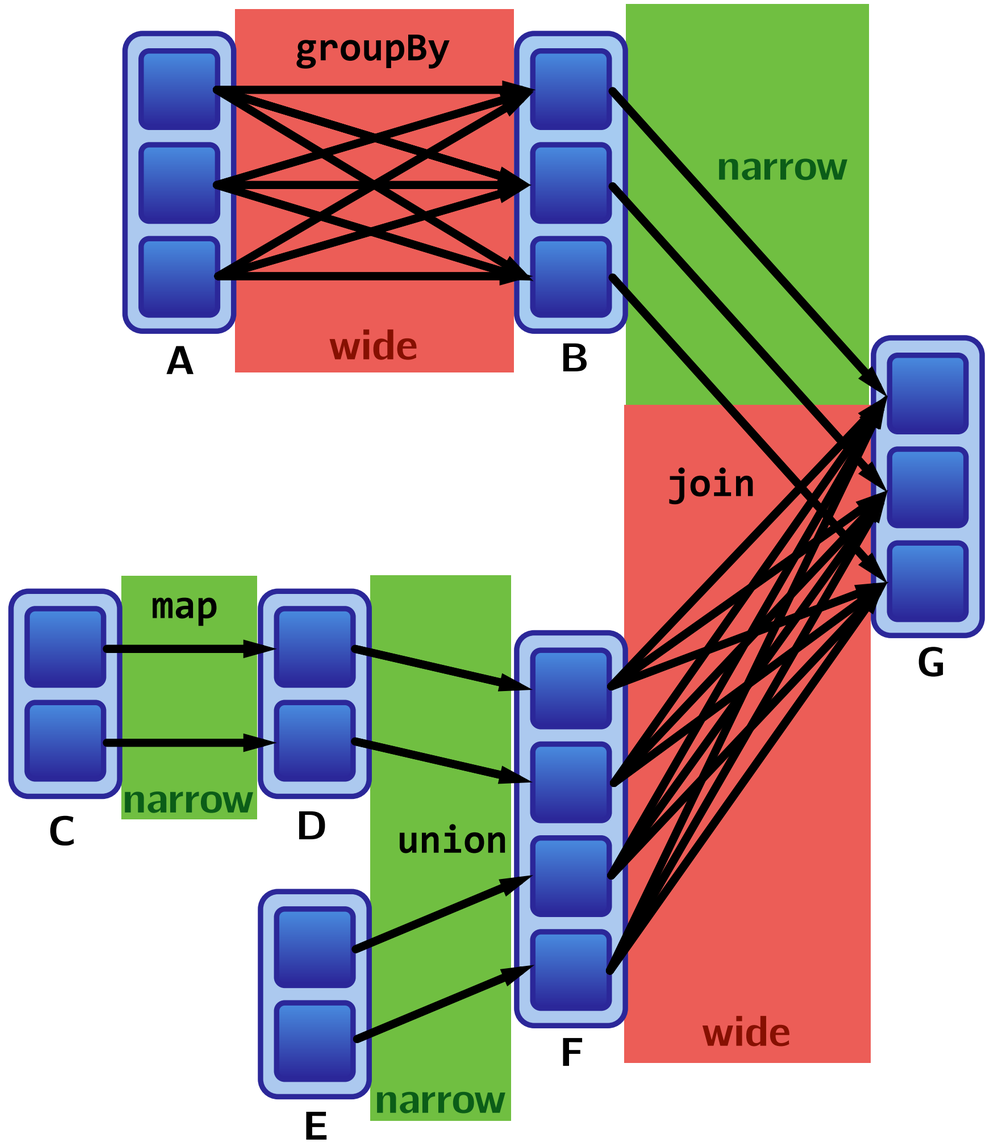


Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

Wide transformations:
`groupBy, join`

Narrow transformations:
`map, union, join`

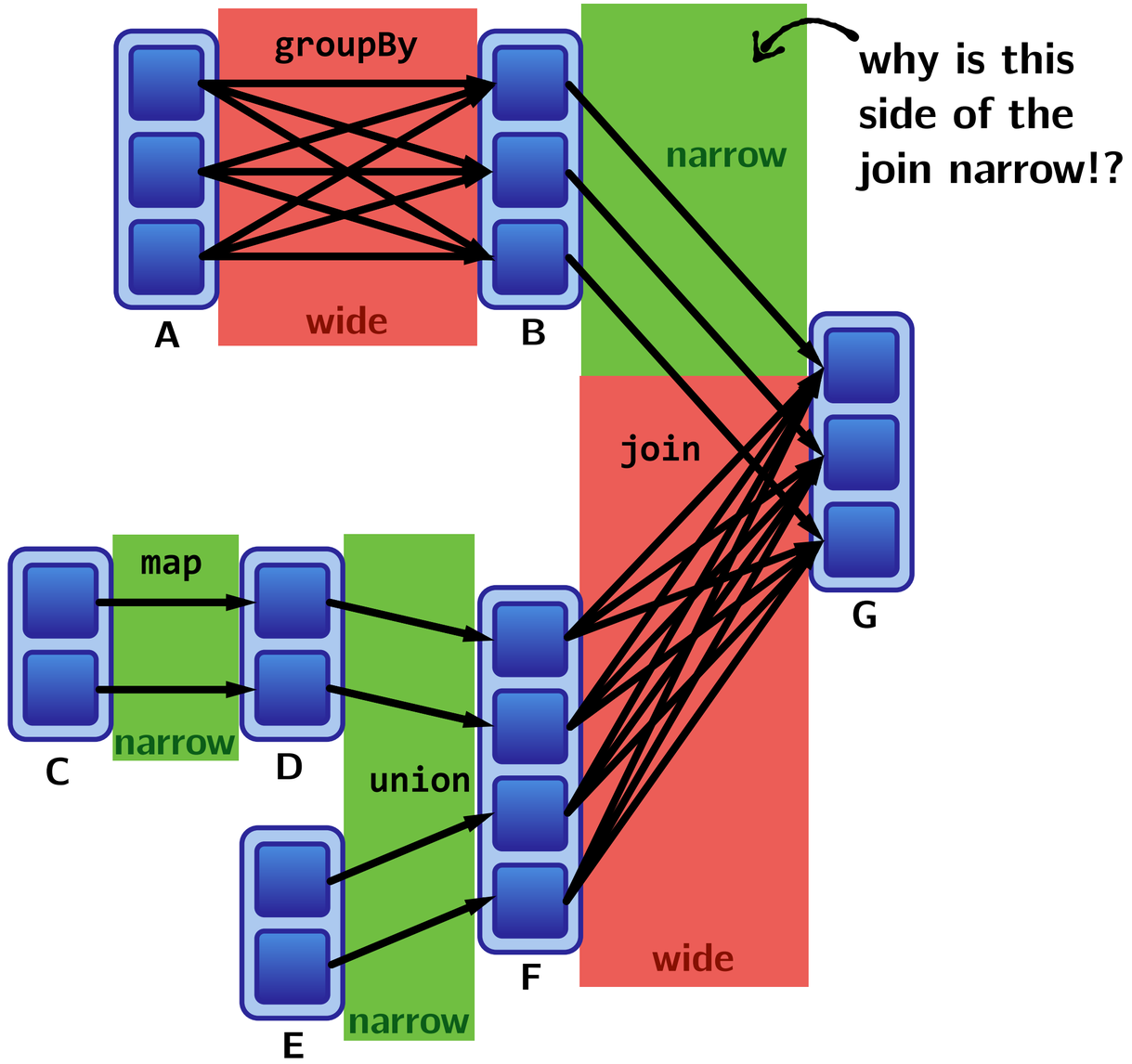


Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

Wide transformations:
`groupBy, join`

Narrow transformations:
`map, union, join`

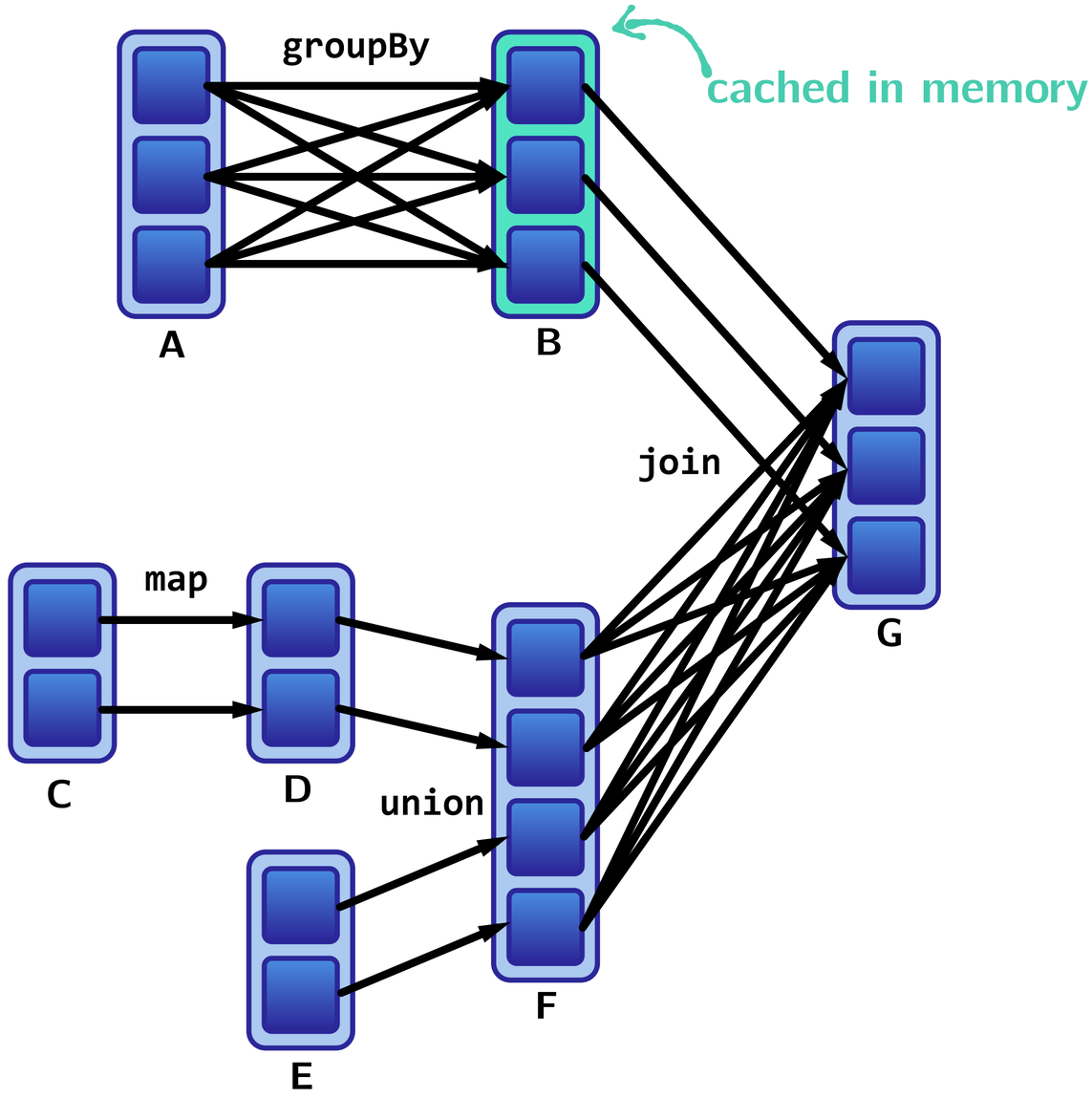


Narrow Dependencies vs Wide Dependences, Visually

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 []
```

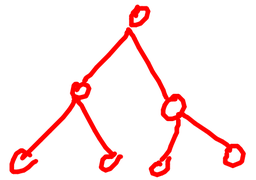
Lineages and Fault Tolerance

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.

in-memory + fault tolerant!

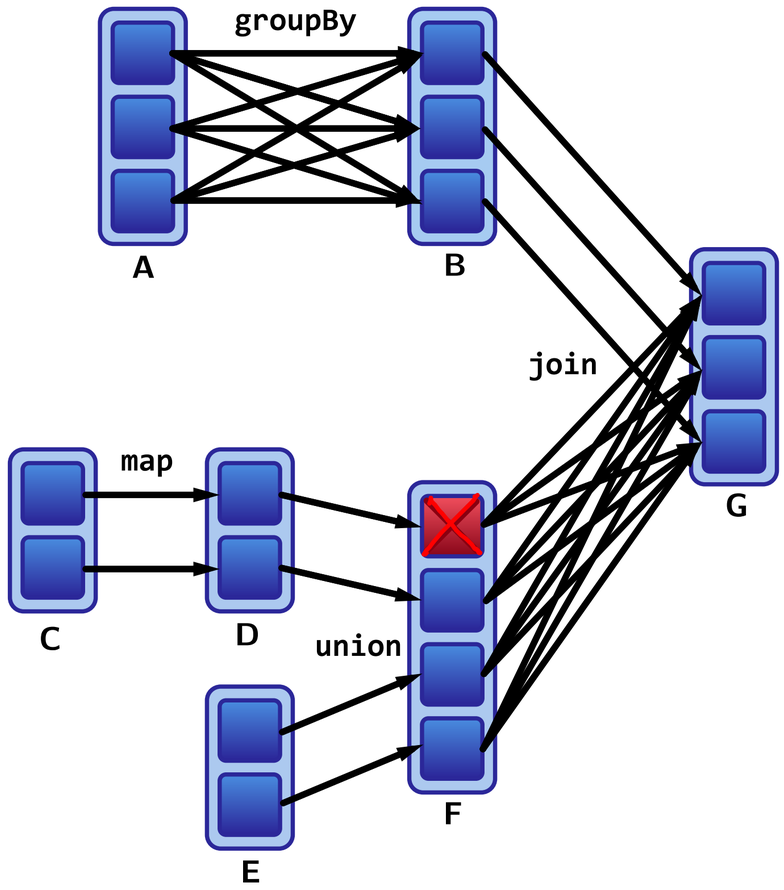
Along with keeping track of dependency information between partitions as well, this allows us to:

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.

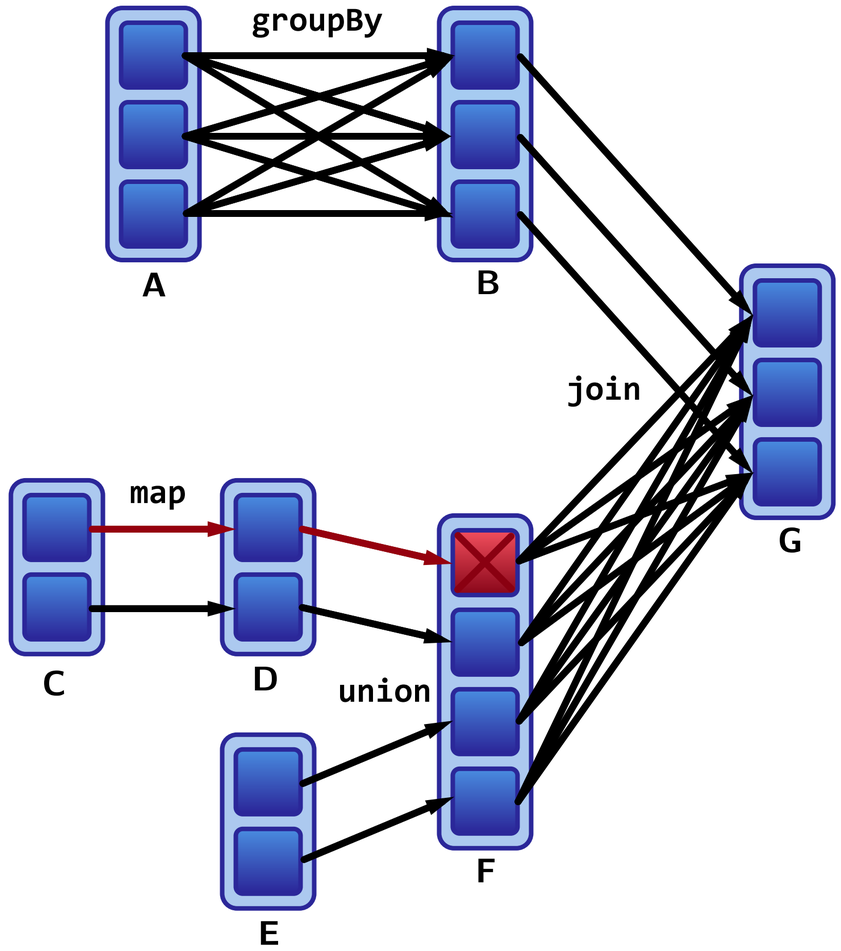
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

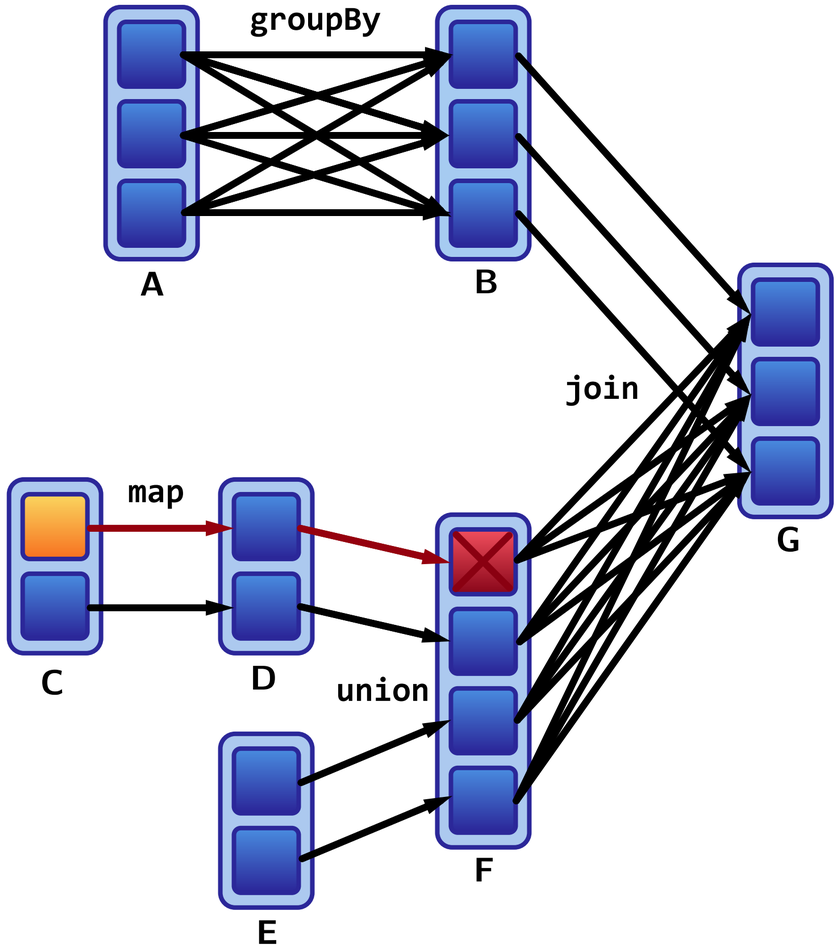
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

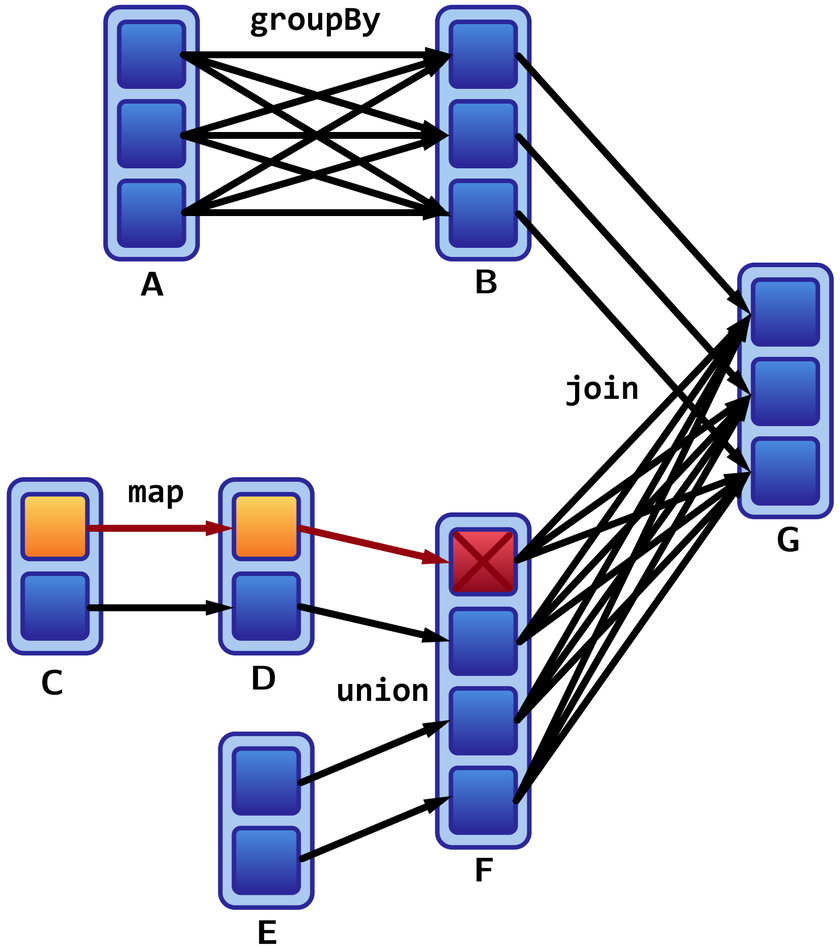
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

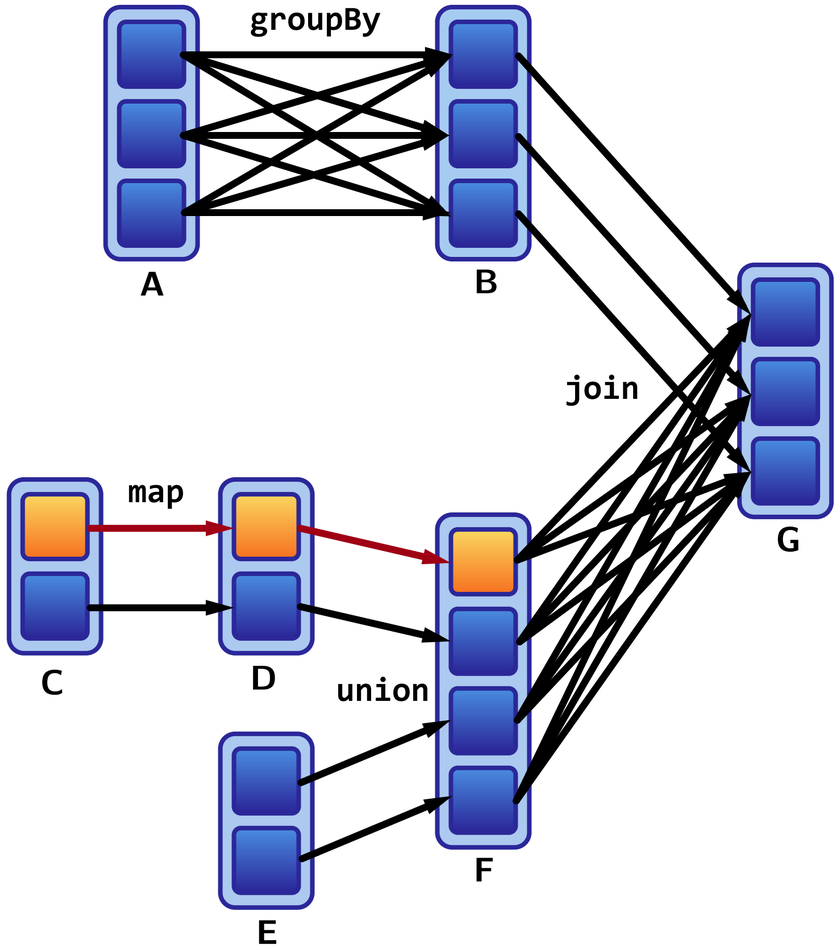
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

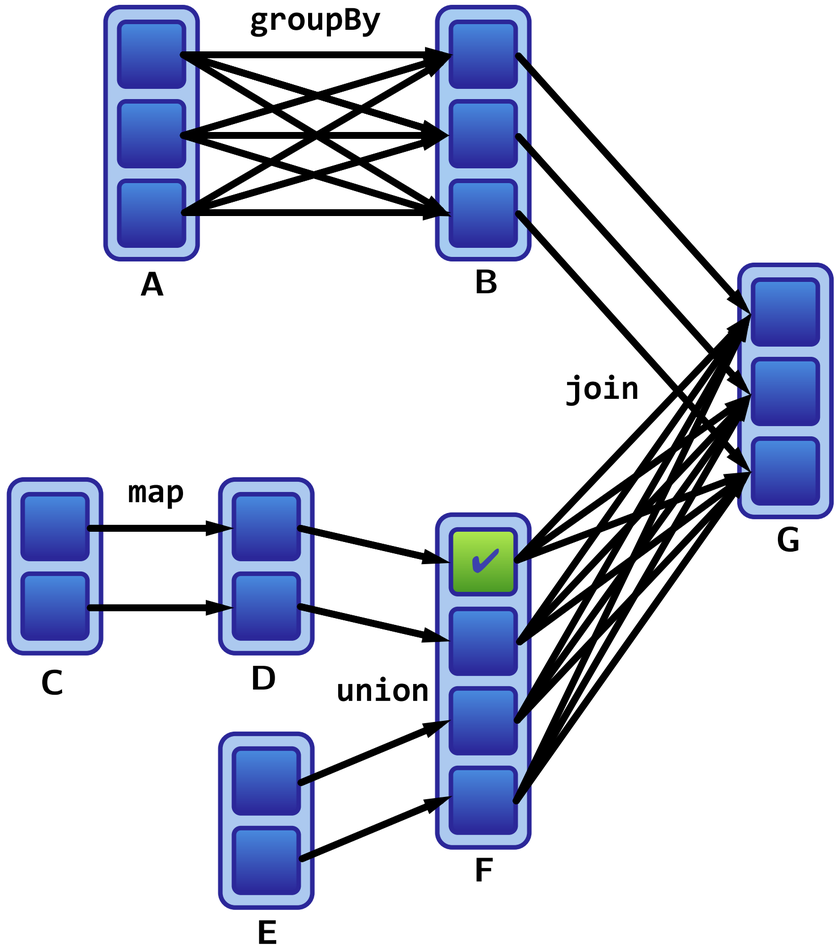
Let's assume one of our partitions from our previous example fails.



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!