17-400/17-700 Machine Learning and Data Science at Scale

Wide vs Narrow Dependencies

Carnegie Mellon University
School of Computer Science

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

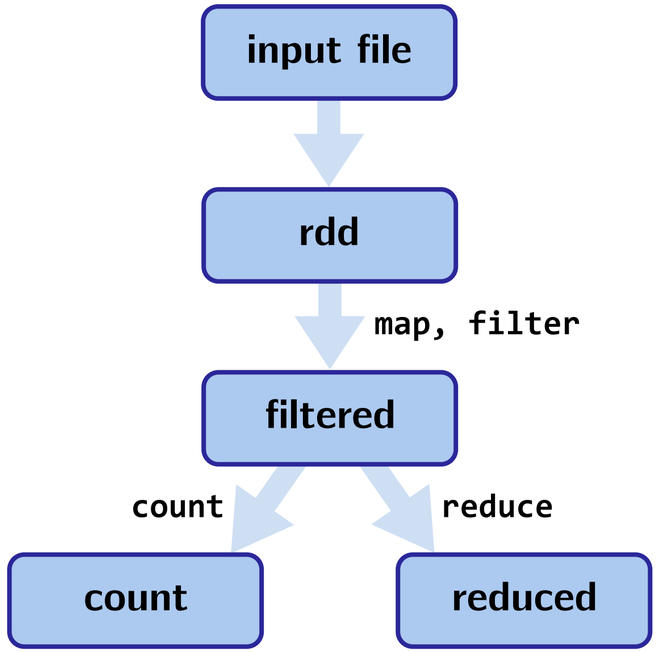
Computations on RDDs are represented as a **lineage graph**; a Directed Acyclic Graph (DAG) representing the computations done on the RDD.

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(...)
```

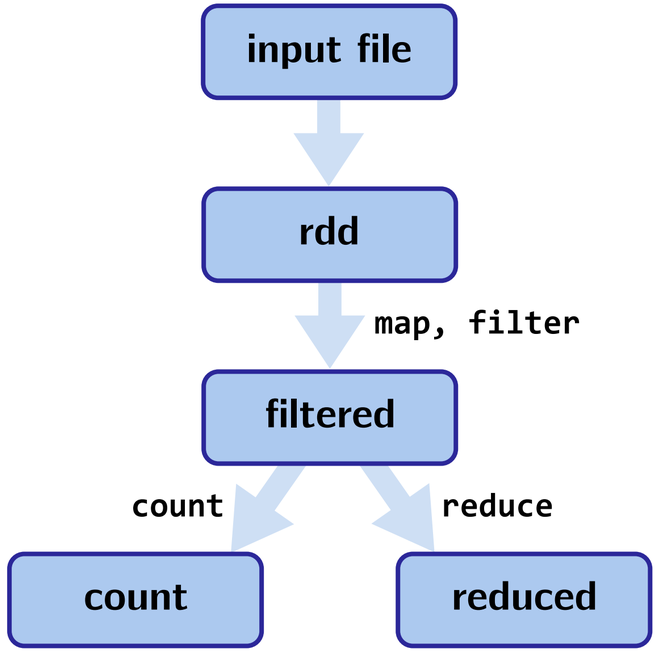


Lineages

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

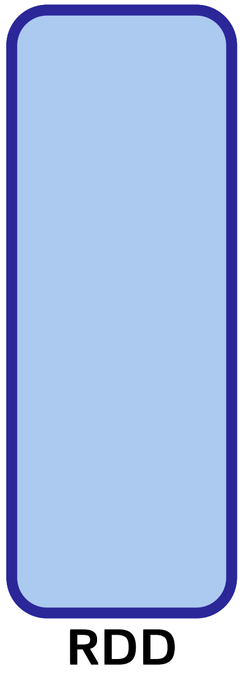
In fact, this is the representation/DAG is what Spark analyzes to do optimizations.

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:

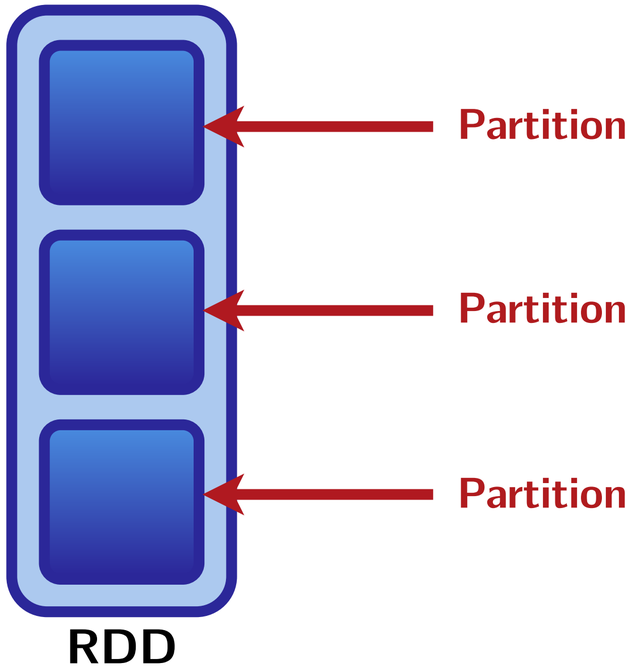


RDD

How are RDDs represented?

RDDs are made up of 2 important parts.

(but are made up of 4 parts in total)



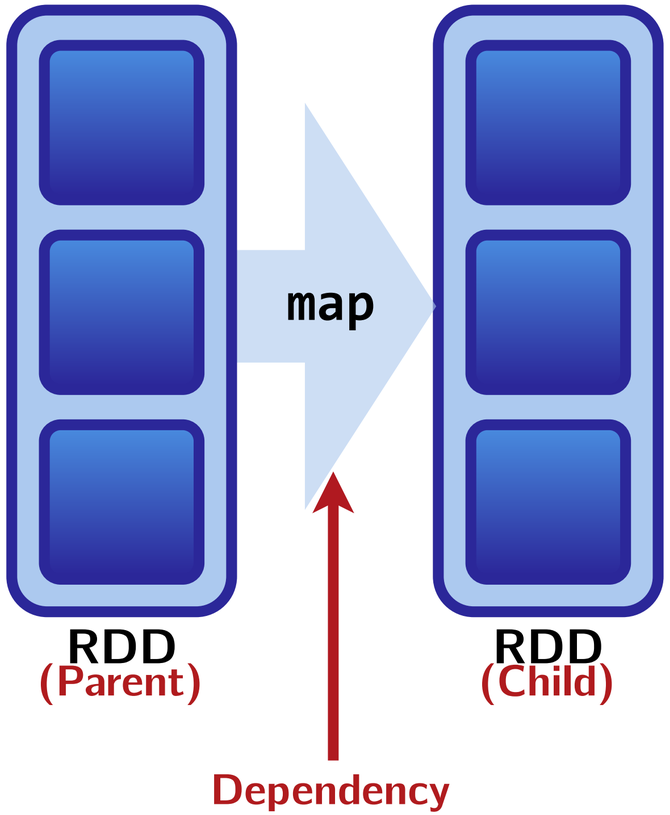
RDDs are represented as:

- **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)



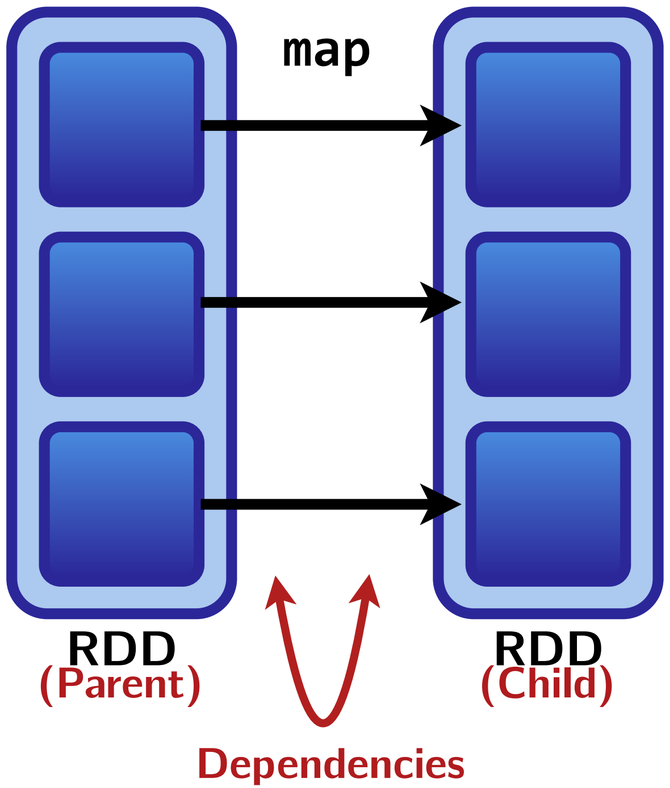
RDDs are represented as:

- **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)



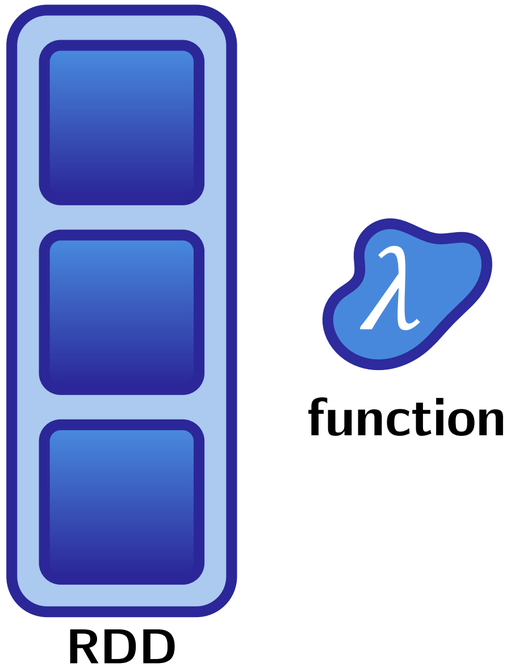
RDDs are represented as:

- **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**

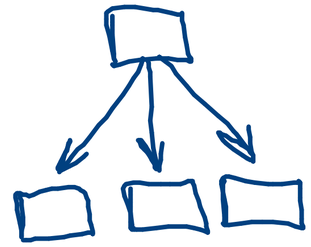
Narrow Dependencies vs Wide Dependences

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

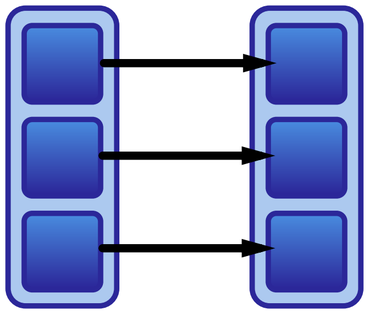
Narrow dependencies:

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.

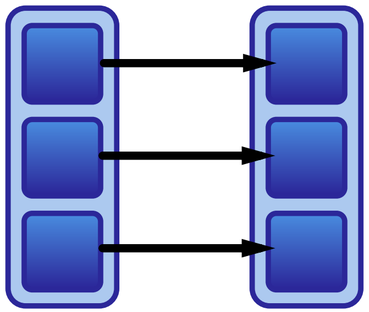


`map, filter`

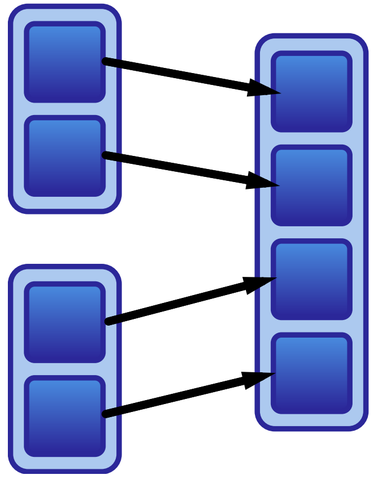
Narrow Dependencies vs Wide Dependences, Visually

Narrow dependencies:

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



`map, filter`

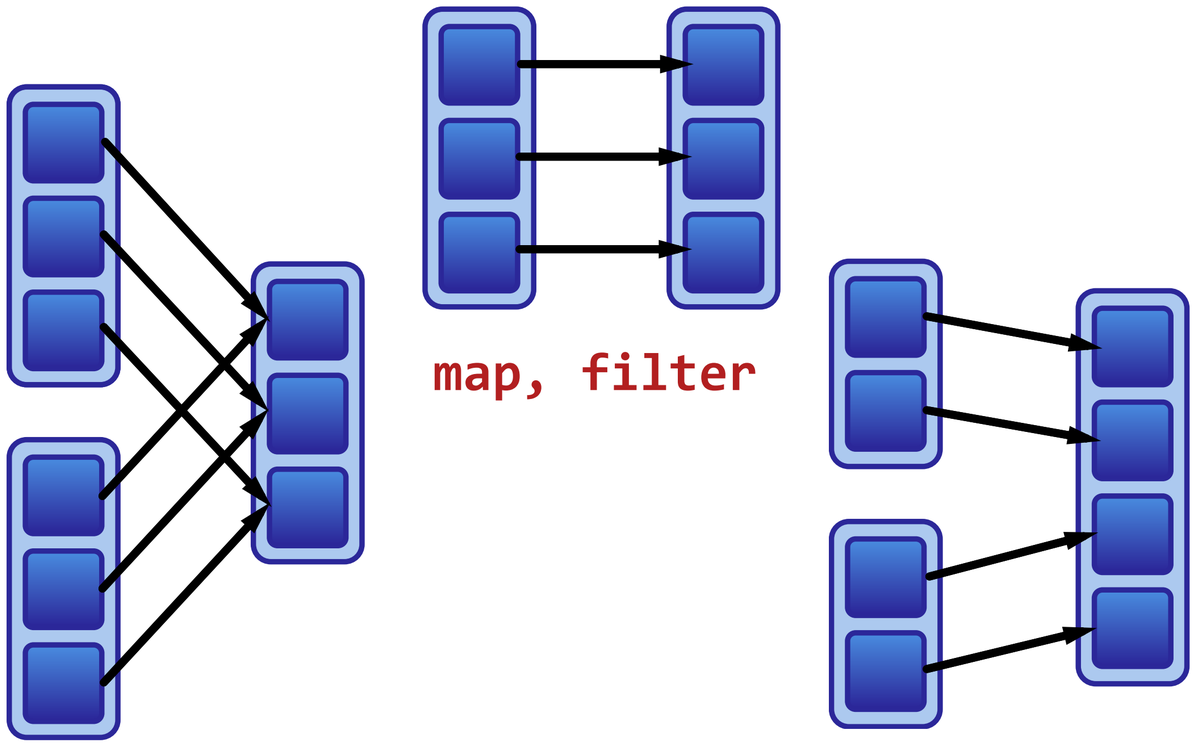


`union`

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`

`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.

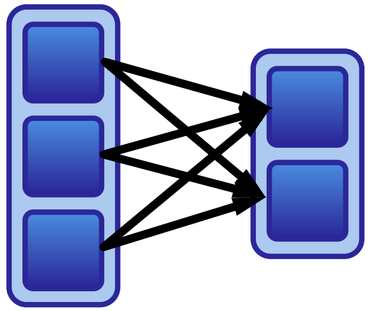
`join` *with co-partitioned inputs*

`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.

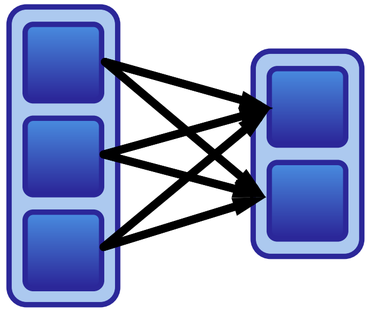
`map, filter`

`union`

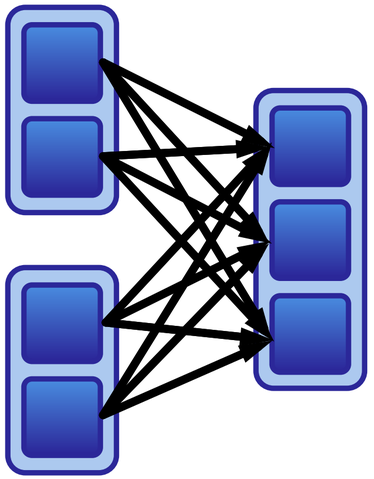
`join` *with co-partitioned inputs*

Wide dependencies:

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



`groupByKey`



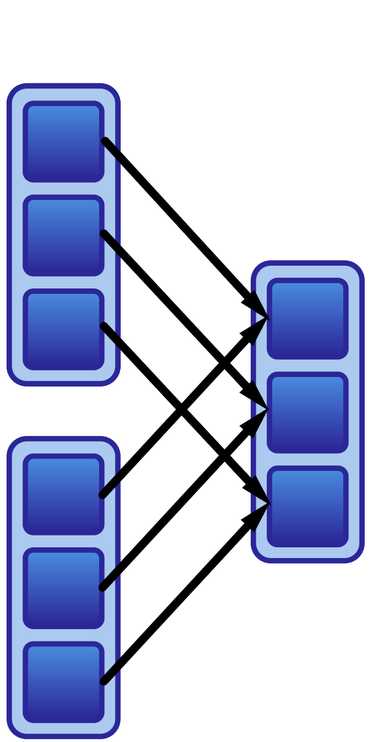
`join`

with inputs not co-partitioned

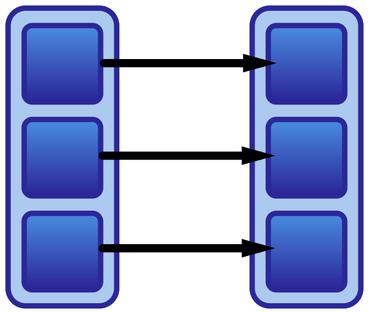
Narrow Dependencies vs Wide Dependences, Visually

Narrow dependencies:

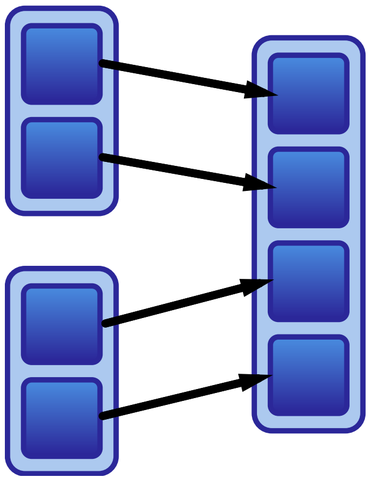
Each partition of the parent RDD is used by at most one partition of the child RDD.



join *with co-partitioned inputs*



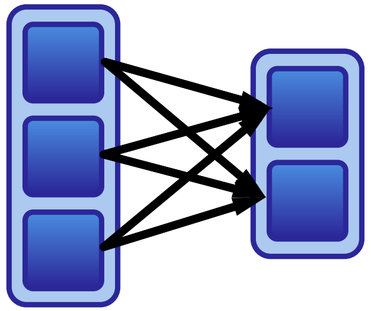
map, filter



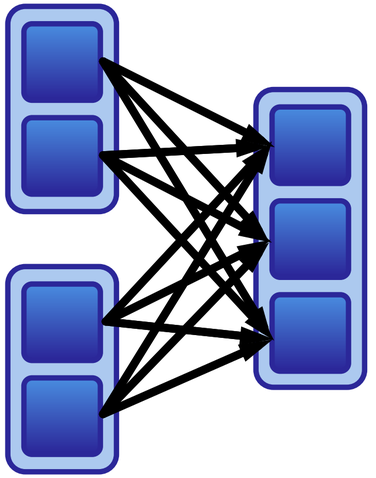
union

Wide dependencies:

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



groupByKey



join *with inputs not co-partitioned*

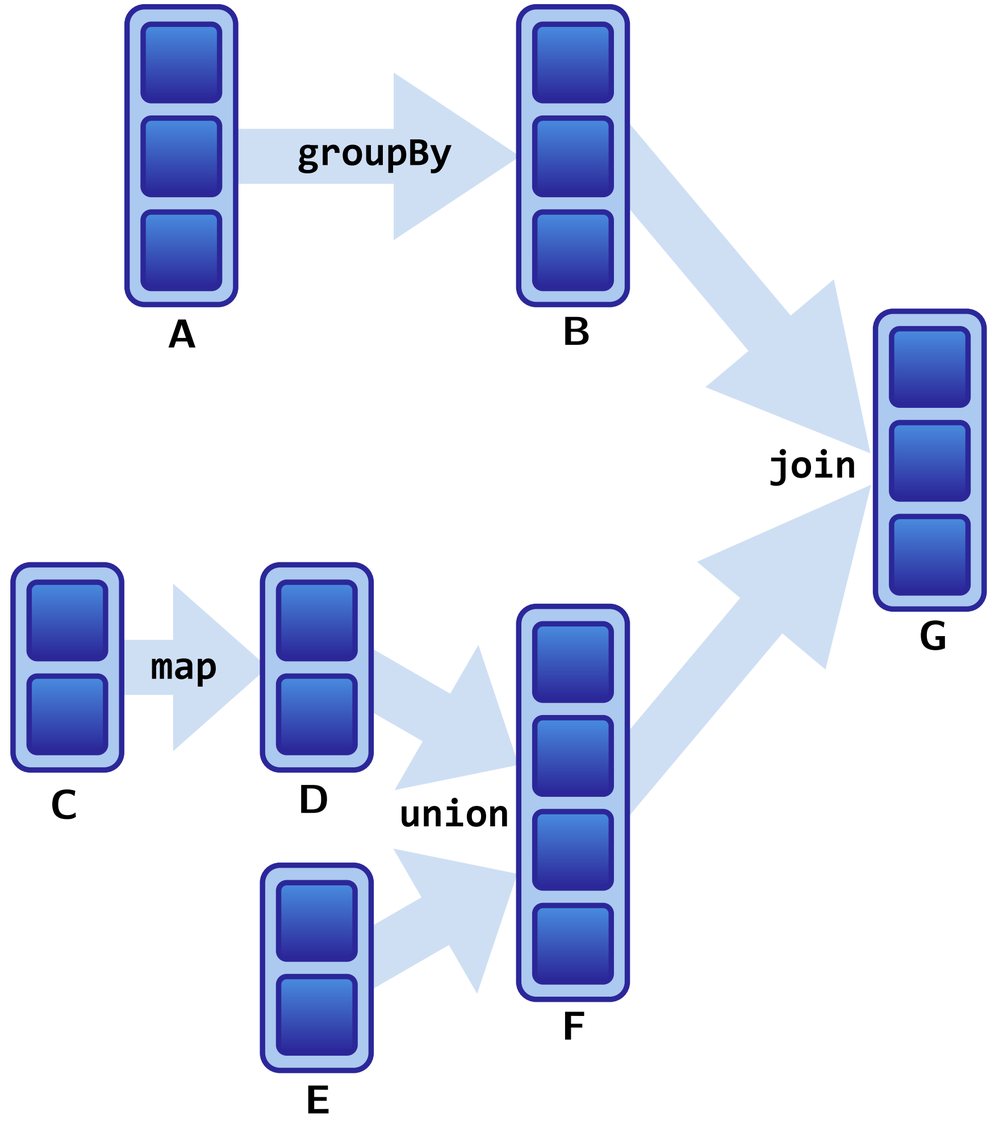
Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example
program and its dependencies.

Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

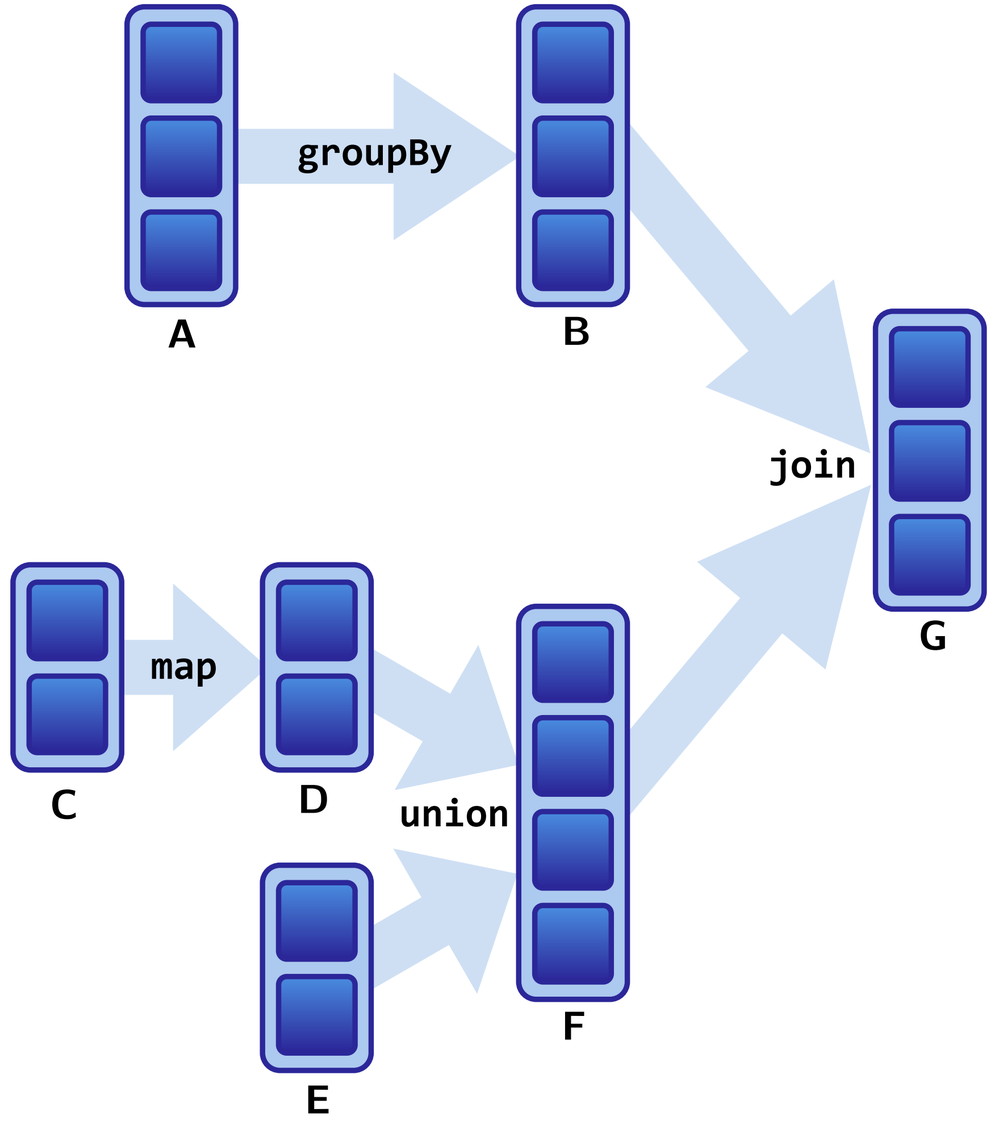
Conceptually assuming the DAG:



Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

Conceptually assuming the DAG:

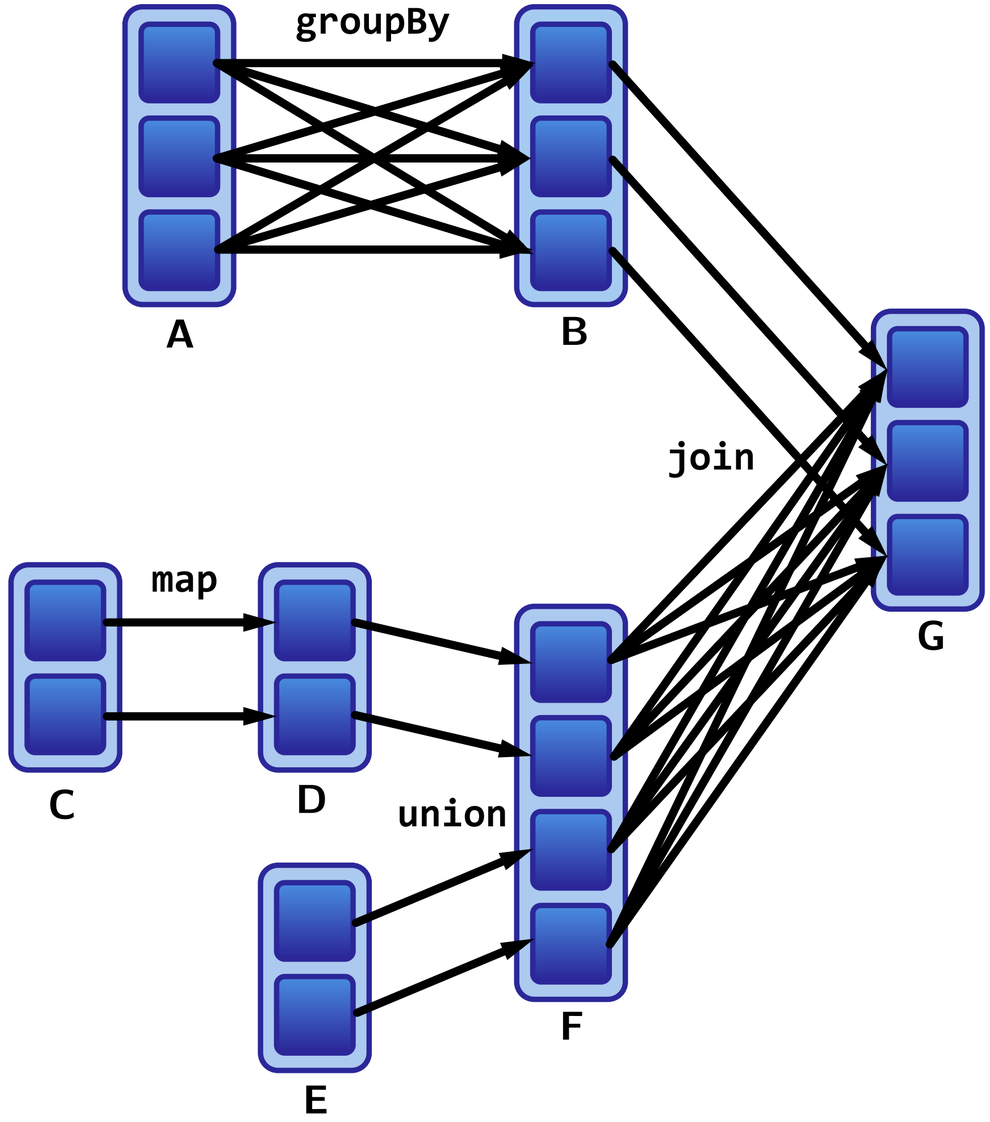


What do the dependencies look like?

Which dependencies are wide, and which are narrow?

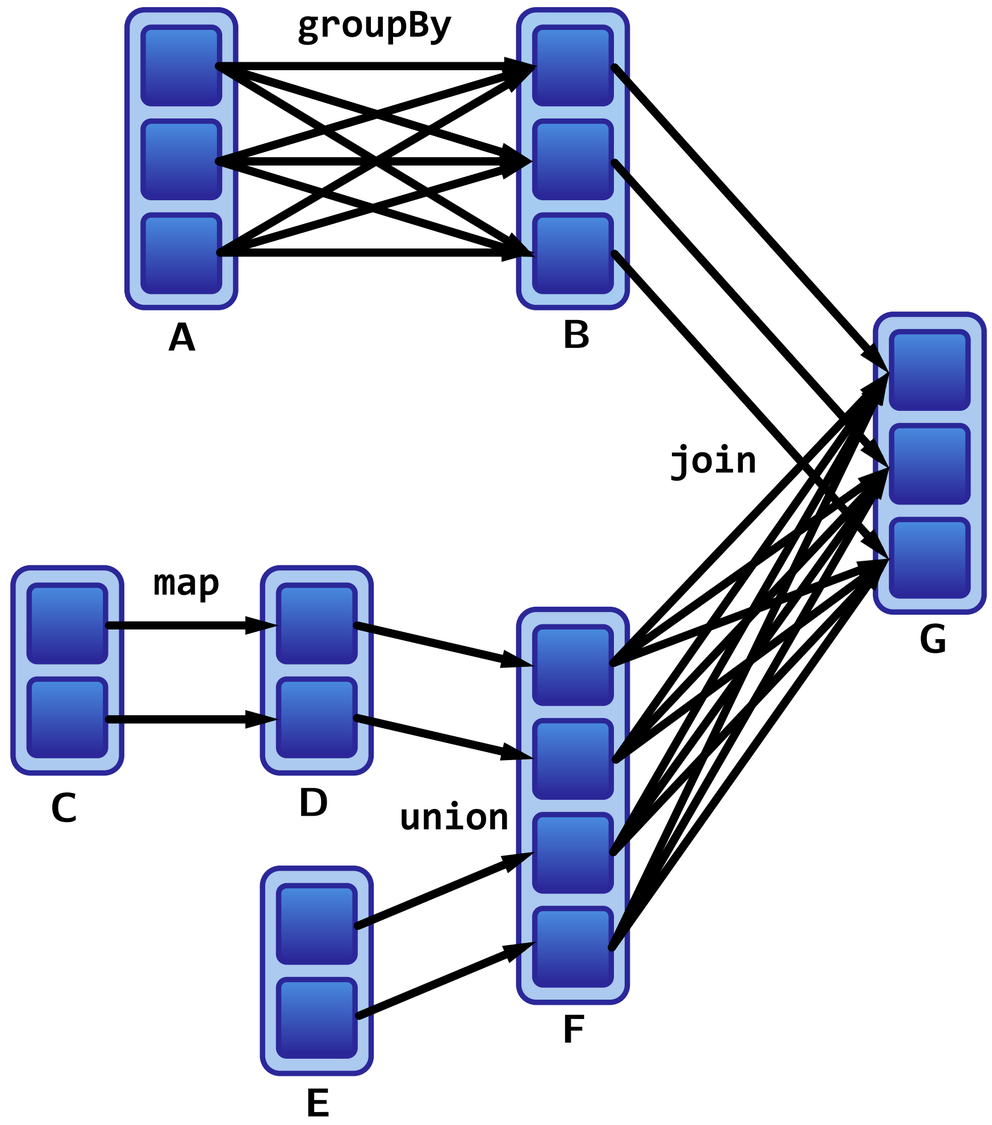
Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.



Narrow Dependencies vs Wide Dependences, Visually

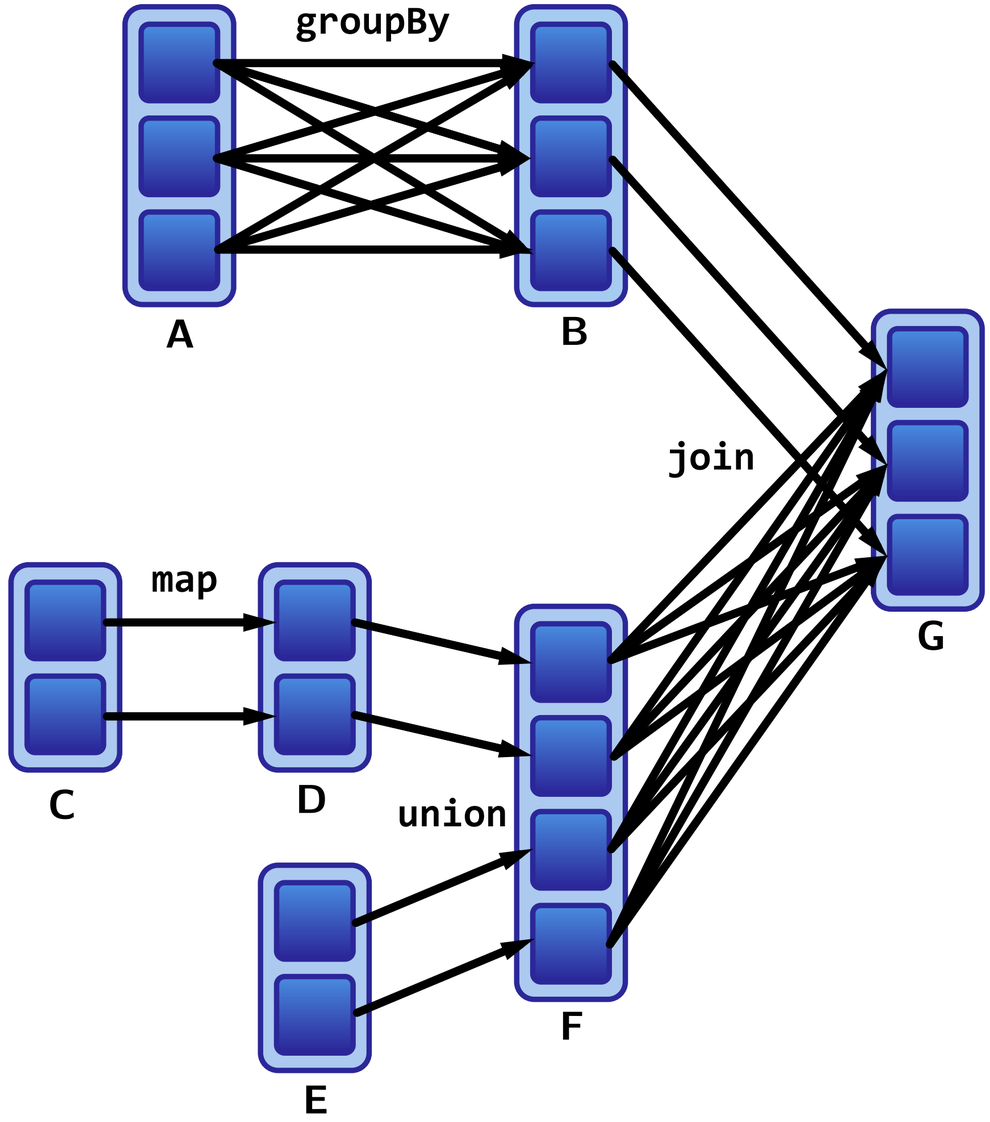
Let's visualize an example program and its dependencies.



Which dependencies are wide, and which are narrow?

Narrow Dependencies vs Wide Dependences, Visually

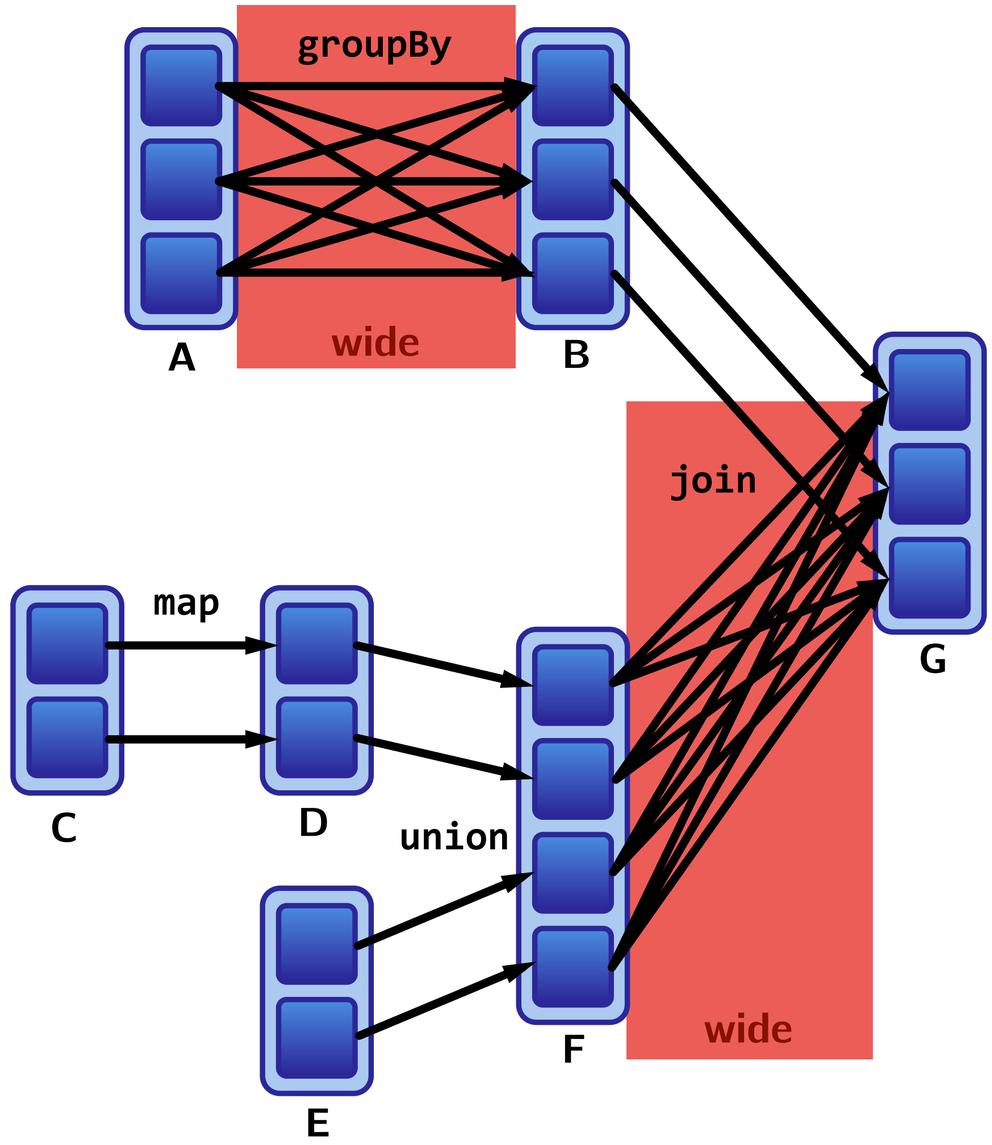
Let's visualize an example program and its dependencies.



Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

Wide transformations:
`groupBy, join`

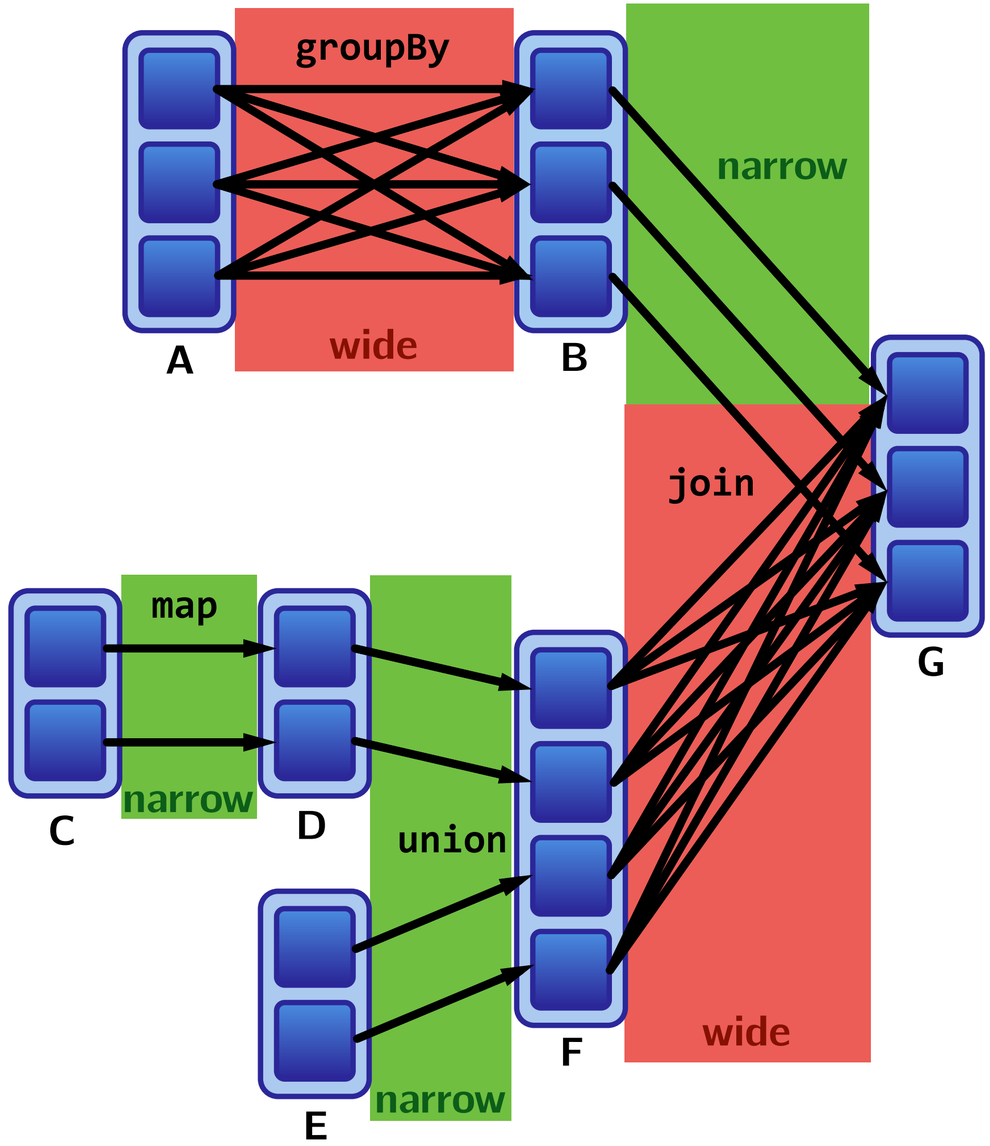


Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

Wide transformations:
`groupBy, join`

Narrow transformations:
`map, union, join`

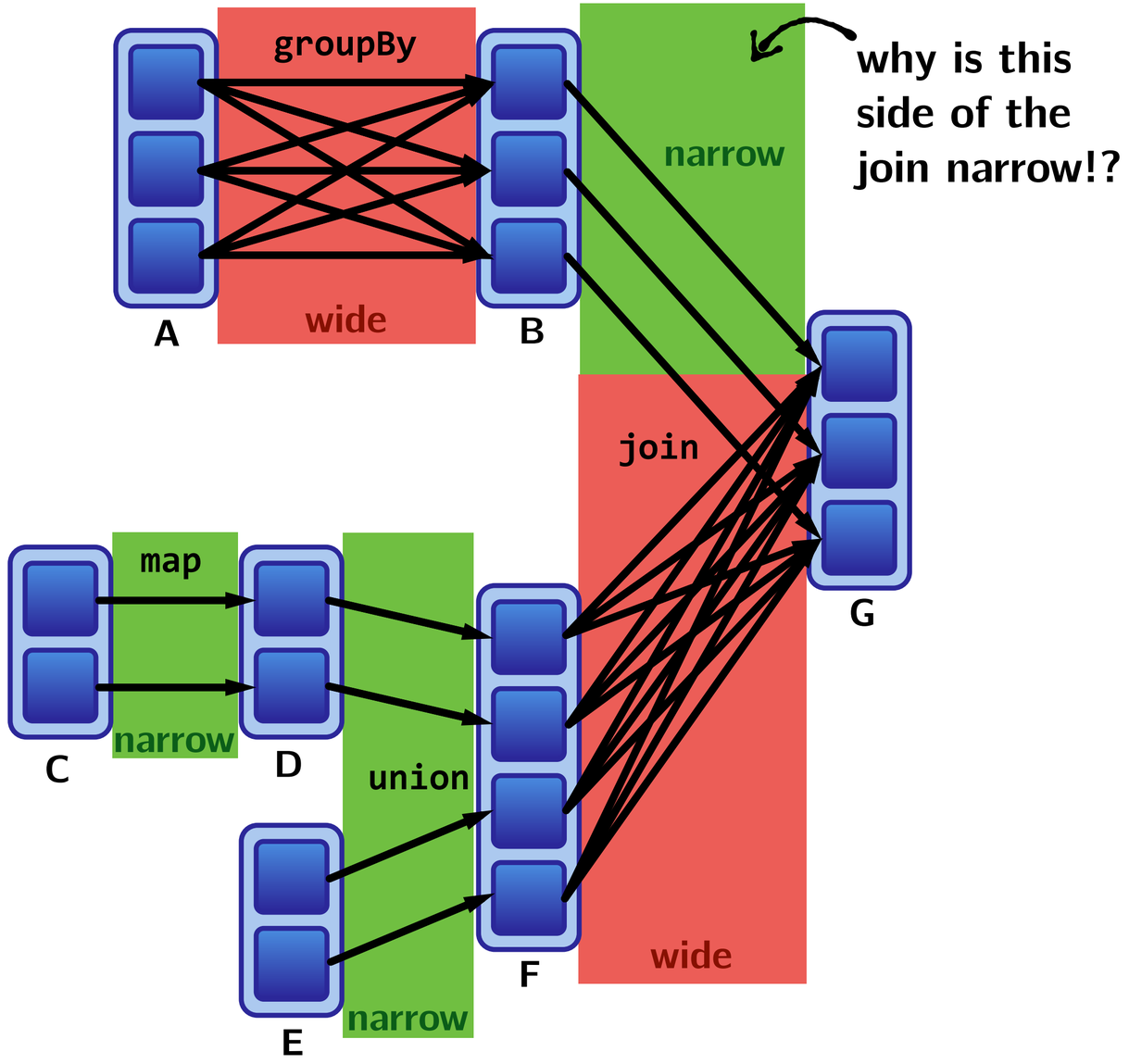


Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

Wide transformations:
`groupBy, join`

Narrow transformations:
`map, union, join`

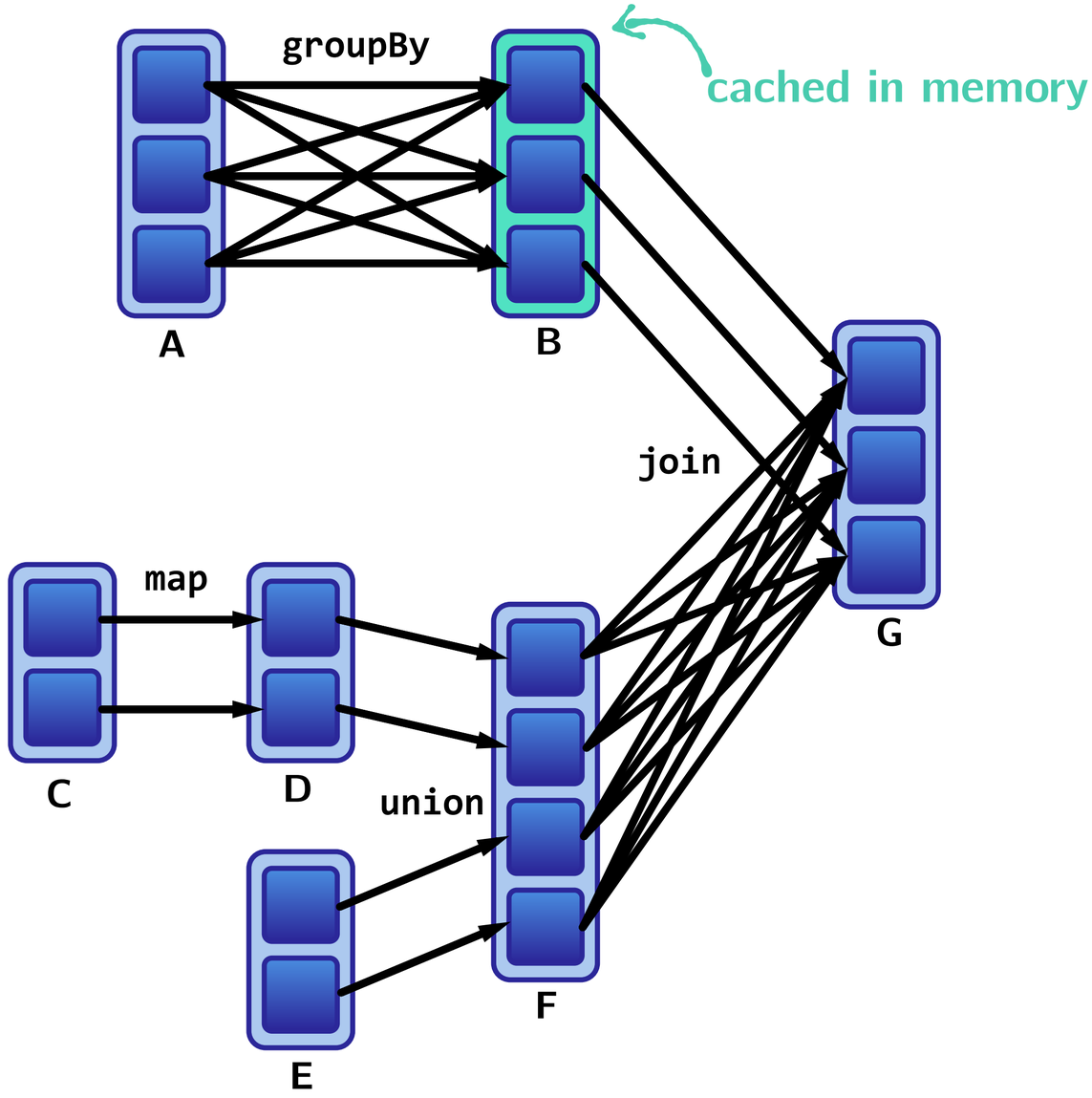


Narrow Dependencies vs Wide Dependences, Visually

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 []
```

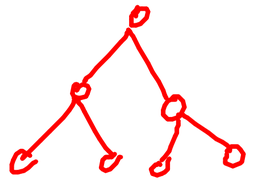
Lineages and Fault Tolerance

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

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:

Recover from failures by recomputing lost partitions from lineage graphs.

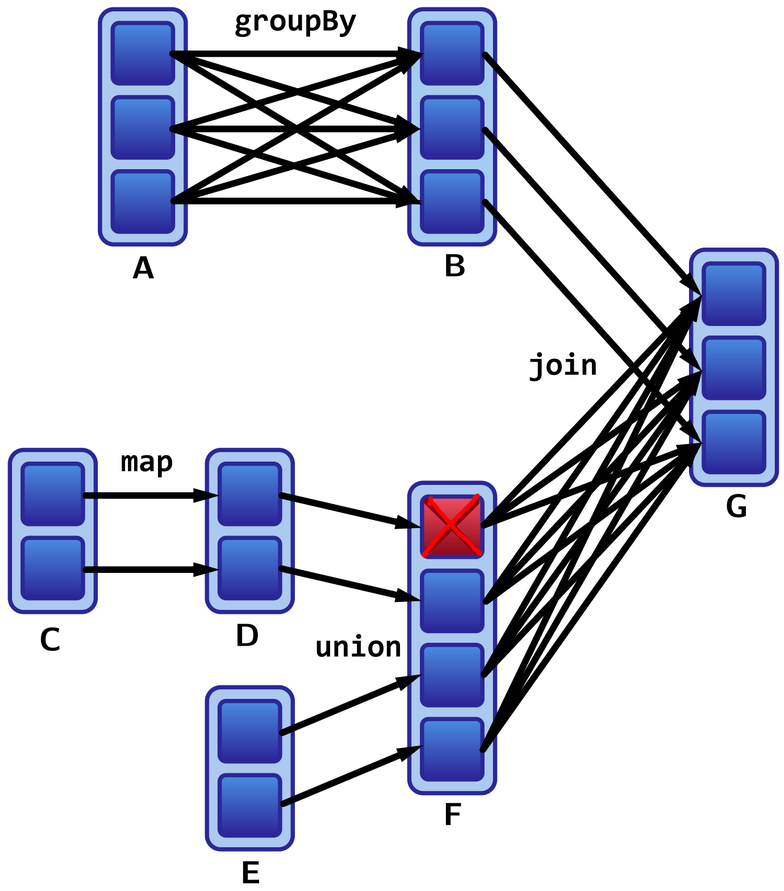
Fault tolerance w/out having to checkpoint + write data to disk!

in-memory + fault tolerant!

Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

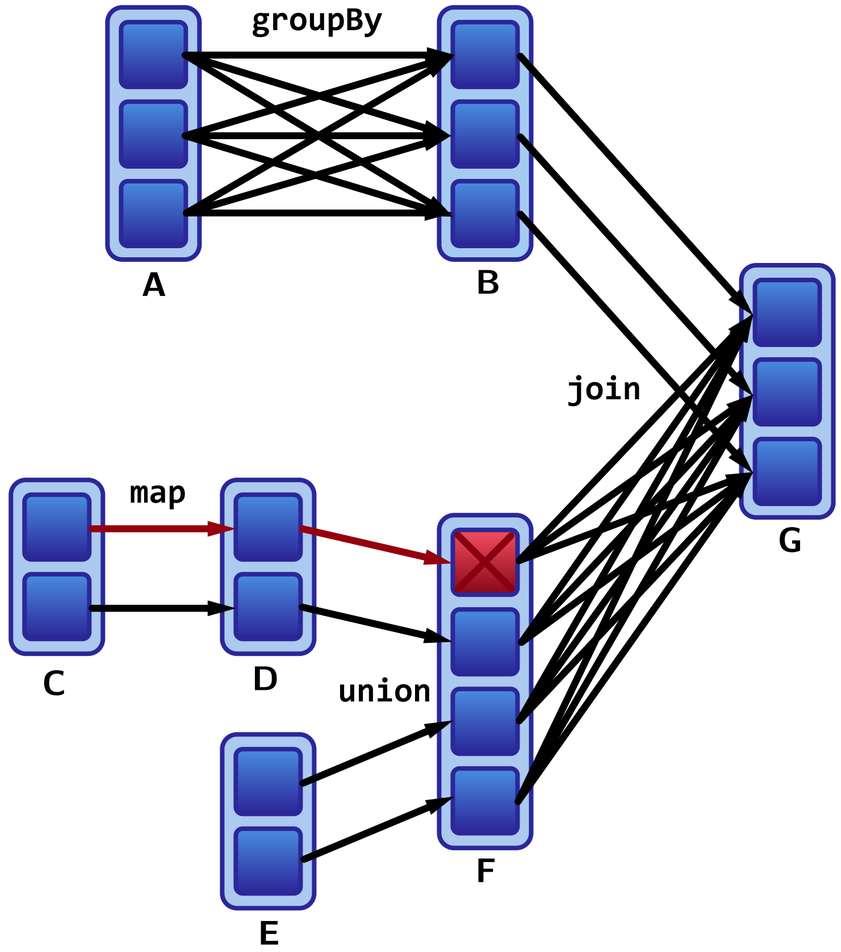
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

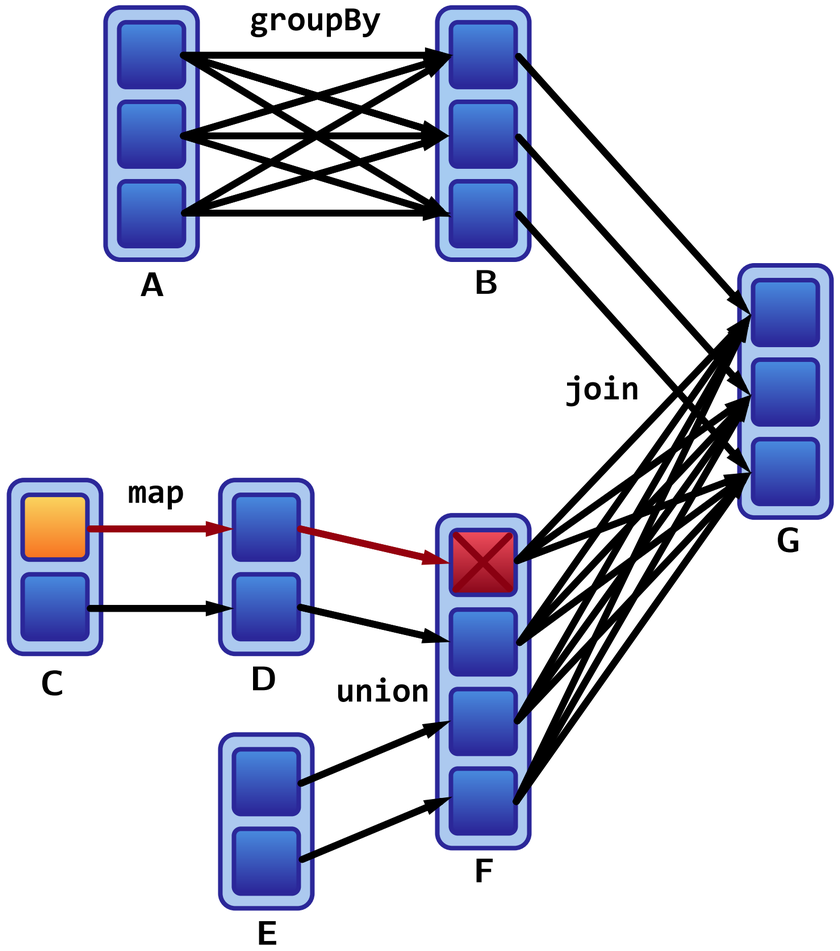
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

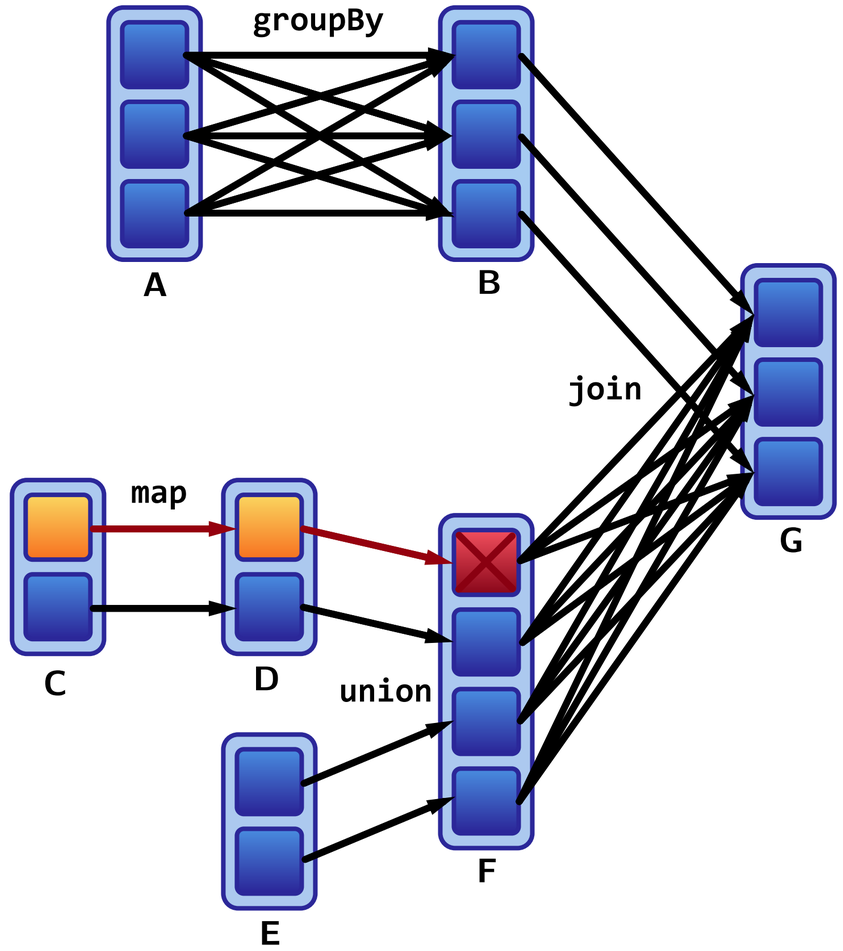
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

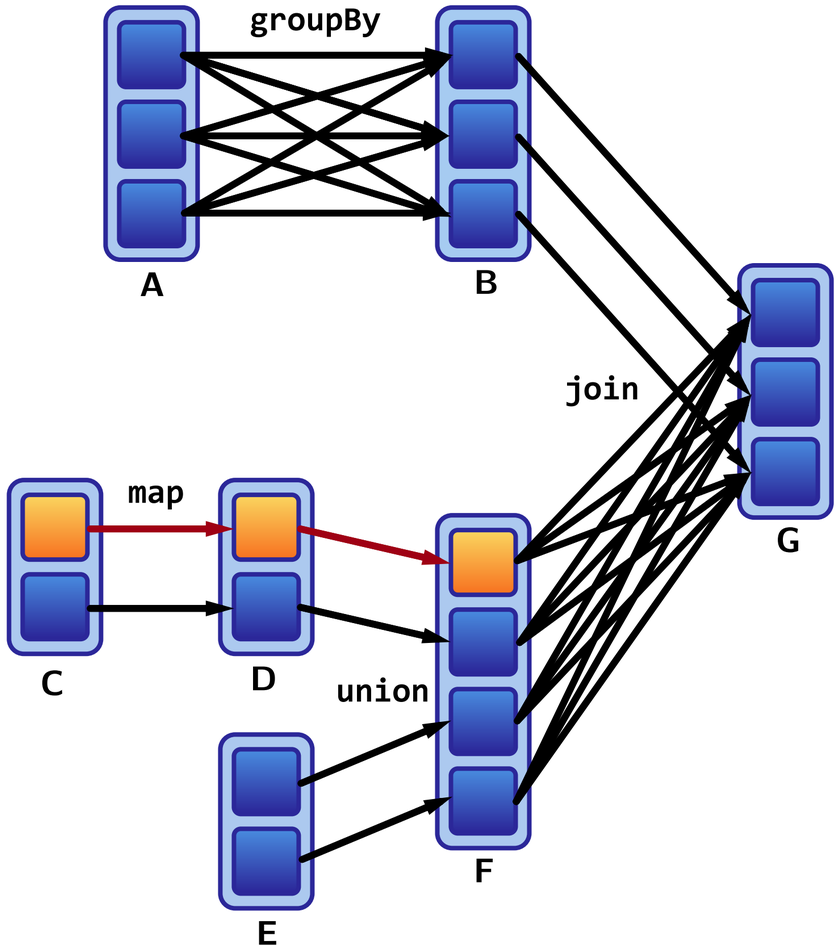
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

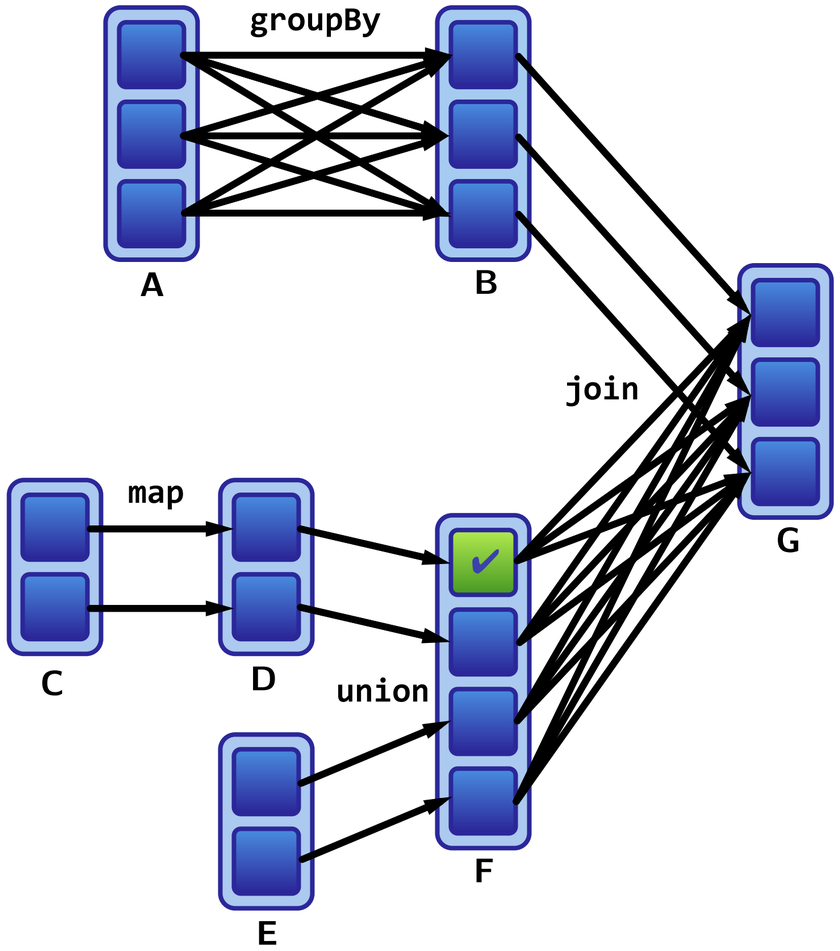
Let's assume one of our partitions from our previous example fails.



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

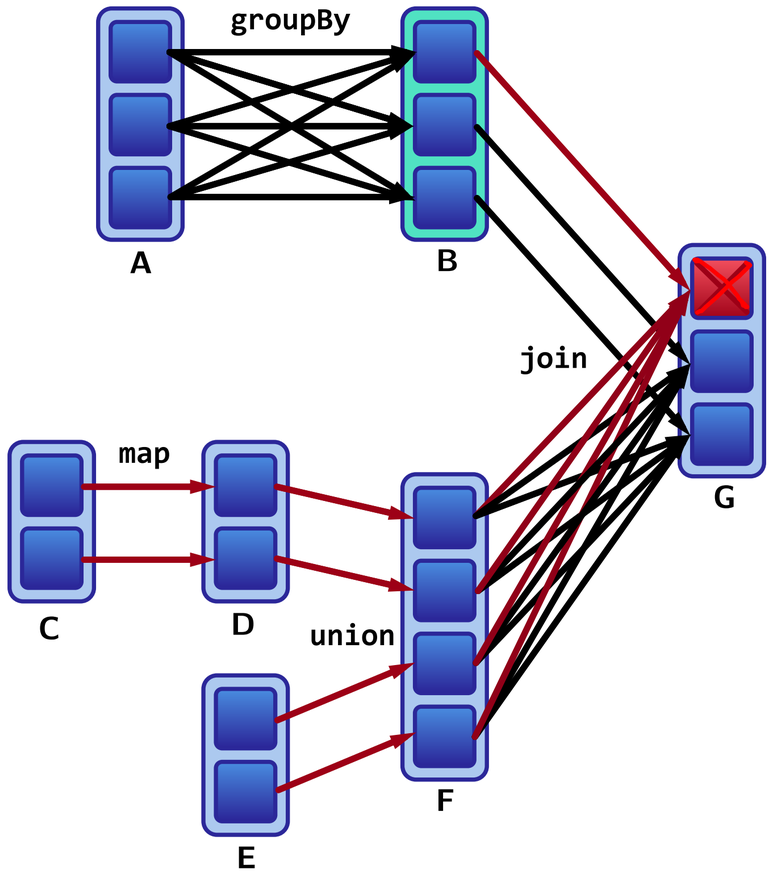
Lineages graphs are the key to fault tolerance in Spark.

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!



17-400/17-700 Machine Learning and Data Science at Scale

Dataframes & Optimizations

Carnegie Mellon University
School of Computer Science

EXAMPLE:

Selecting Scholarship Recipients

Let's imagine that we are an organization, CodeAward, offering scholarships to programmers who have overcome adversity. Let's say we have two datasets with the following schemas:

demographic:
- id (int)
- age (int)
- codingBootcamp (boolean)
- country (string)
- isMinority (boolean)
- isVeteran (Boolean)

finances:
- id (int)
- hasDebt (boolean)
- hasFinancialDependents (boolean)
- hasStudentLoans (boolean)
- income (int)

Let's assume we have the following two RDDs representing potential awardees:

```
demographics = sc.textfile(...) # Pair RDD, (id, tuple representing demographic)
finances = sc.textfile(...)# Pair RDD, (id, tuple representing finances)
```

EXAMPLE:

Selecting Scholarship Recipients

Our data sets include students from many countries, with many life and financial backgrounds. Now, let's imagine that our goal is to tally up and select students for a specific scholarship.

As an example, Let's count:

- Swiss students
- who have debt & financial dependents

How might we implement this Spark program?

Remember that we have the following RDDs:

```
demographics = sc.textfile(...) # Pair RDD, (id, tuple representing demographic)
finances = sc.textfile(...)# Pair RDD, (id, tuple representing finances)
```

EXAMPLE:

Selecting Scholarship Recipients

Possibility 1:

```
demographics.\
  join(finances).\
  filter(lambda p: p[1][0][2] == "Switzerland" and p[1][1][1] and p[1][1][2]).\
  count()
```

Steps:

1. Inner join first
2. Filter to select people in Switzerland
3. Filter to select people with debt & financial dependents

EXAMPLE:

Selecting Scholarship Recipients

Possibility 2:

```
filtered = finances.filter(lambda p: p[1][1] and p[1][2])

demographics.filter(lambda p: p[1][2] == "Switzerland").\
             join(filtered).\
             count()
```

Steps:

1. Filter down the dataset first (look at only people with debt & financial dependents)
2. Filter to select people in Switzerland (look at only people in Switzerland)
3. Inner join on smaller, filtered down dataset

EXAMPLE:

Selecting Scholarship Recipients

Possibility 3:

```
cartesian = demographics.cartesian(finances)

cartesian.filter(lambda p: p[0][0] == p[1][0]).\
          filter(lambda p: p[0][1][2] == "Switzerland" and p[1][1][1] and p[1][1][2]).\
          count()
```

Steps:

1. Cartesian product on both datasets
2. Filter to select resulting of cartesian with same IDs
3. Filter to select people in Switzerland who have debt and financial dependents

EXAMPLE:

Selecting Scholarship Recipients

For each implementation, the end result is the same, **but the time it takes to execute the job is vastly different**.

Possibility 1

```
> ds.join(fs)
    .filter(p => p._2._
    .count
```

▸ (1) Spark Jobs

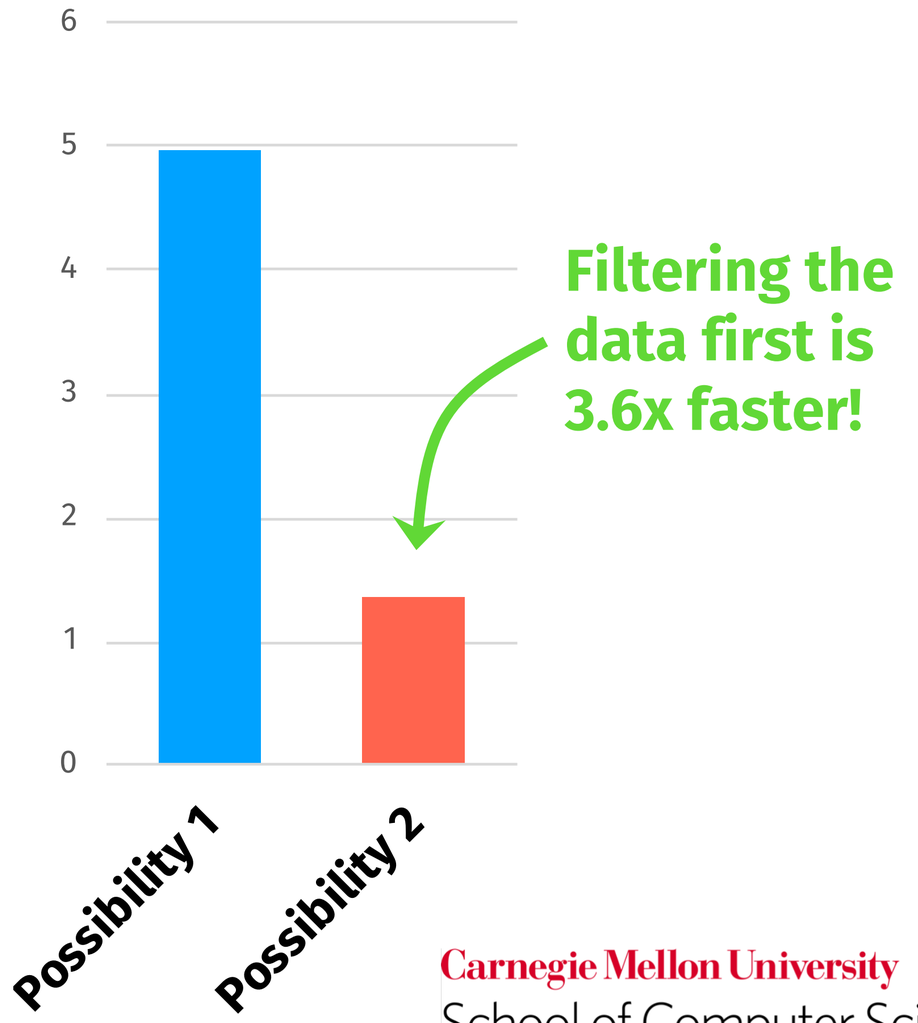
```
res0: Long = 10
Command took 4.97 seconds -
```

Possibility 2

```
> val fsi = fs.filter(
  ds.filter(p => p._2.
    .join(fsi)
    .count
```

▸ (1) Spark Jobs

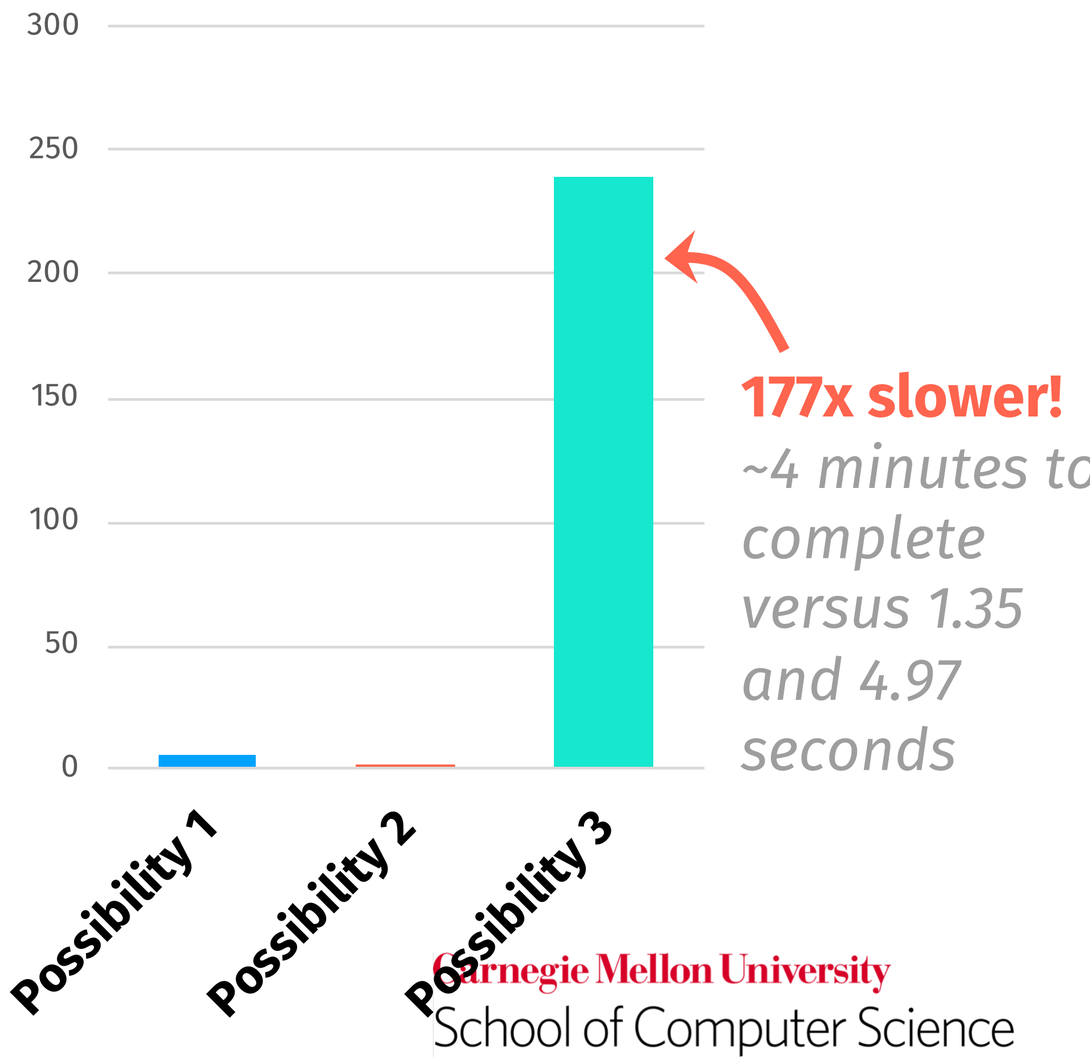
```
fsi: org.apache.spark.
res4: Long = 10
Command took 1.35 seconds
```



EXAMPLE:

Selecting Scholarship Recipients

For each implementation, the end result is the same, **but the time it takes to execute the job is vastly different**.



RECURRING THEME: 

We actually have to think carefully about how our Spark jobs might be executed on the cluster in order to get good performance.

RECURRING THEME:

But wait. Wouldn't it be nice if Spark automatically knew, if we wrote the code in possibility 3, that it could rewrite our code to possibility 2?

might be executed

Given a bit of extra structural information, Spark can do many optimizations for you!

A NOTE ON STRUCTURE

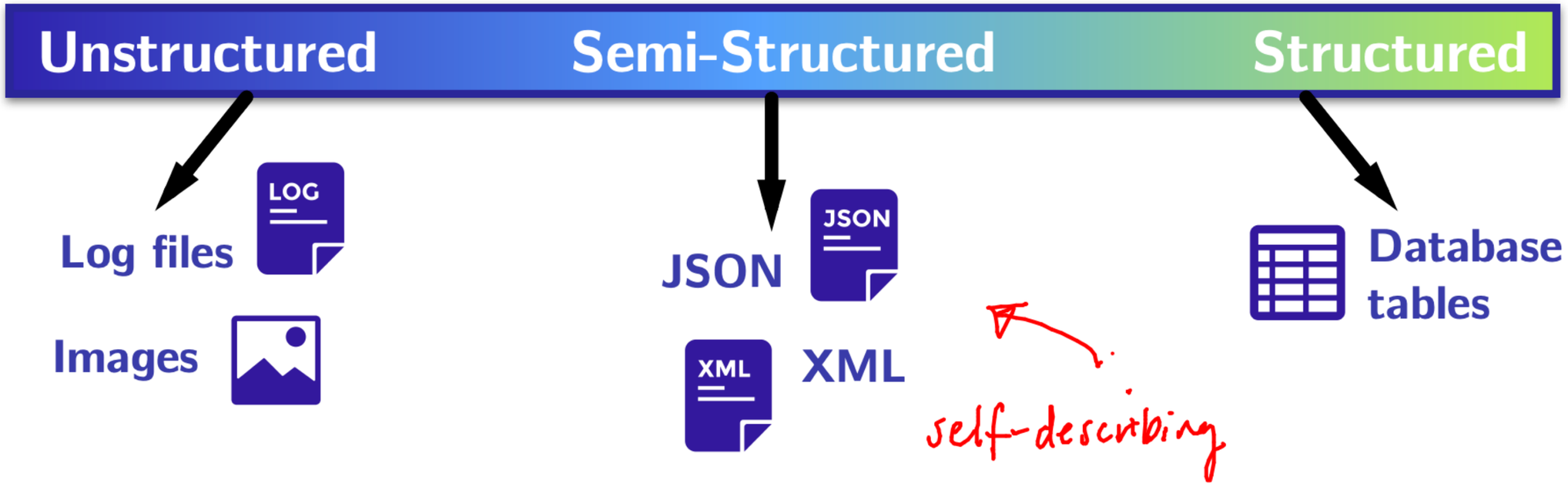
Structured vs Unstructured Data

All data isn't equal, structurally. It falls on a spectrum from unstructured to structured.

A NOTE ON STRUCTURE

Structured vs Unstructured Data

All data isn't equal, structurally. It falls on a spectrum from unstructured to structured.



A NOTE ON STRUCTURE

Structured Data vs RDDs

Spark + regular RDDs don't know anything about the **schema** of the data it's dealing with.

For example, given an RDD containing instances of a class called `Account`
(Assume `Account` *contains fields* `name`, `balance`, *and* `risk`.*)*

Spark/RDDs see:

Blobs of objects we know nothing about, except that they're called `Account`.

Spark can't see inside this object or analyze how it may be used, and to optimize based on that usage. It's opaque.

A database/Hive sees:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

A NOTE ON STRUCTURE

Structured Data vs RDDs

Spark + regular RDDs don't know anything about the **schema** of the data it's dealing with.

For example, given an RDD containing instances of a class called `Account`
(Assume `Account` *contains fields* `name`, `balance`, *and* `risk`.*)*

Spark/RDDs see:

A database/Hive sees:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

Columns of named and typed values.

If Spark could see data this way, it could break up and only select the datatypes it needs to send around the cluster.

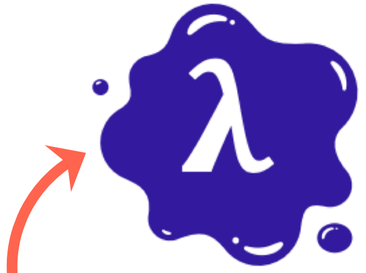
A NOTE ON STRUCTURE

Structured Data vs Computation

The same can be said about ***computation***.

In Spark:

- We do **functional transformations** on data.
- We pass user-defined function literals to higher-order functions like **`map`**, **`flatMap`**, and **`filter`**.



Like the data Spark operates on, function literals too are completely opaque to Spark.

A user can do anything inside of one of these, and all Spark can see is something like:
$anon$1@604f1a67

A NOTE ON STRUCTURE

Structured Data vs Computation

The same can be said about ***computation***.

In Spark:

- We do **functional transformations** on data.
- We pass user-defined function literals to higher-order functions like **`map`**, **`flatMap`**, and **`filter`**.

In a database/Hive:

- We do **declarative transformations** on data.
- Specialized/structured pre-defined operations.

Fixed set of operations, fixed set of types they operate on.

Optimizations the norm!

IN SUMMARY

Structured vs Unstructured

Spark RDDs: *as we know them so far*

Not much structure.

Difficult to aggressively optimize.

Databases/Hive:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

```
SELECT
WHERE
ORDER BY
GROUP BY
COUNT
```

IN SUMMARY

Structured vs Unstructured

Spark RDDs: *as we know them so far*

Not much structure.

Difficult to aggressively optimize.

Databases/Hive:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

```
SELECT
WHERE
ORDER BY
GROUP BY
COUNT
```

Lots of structure.

Lots of optimization opportunities!

Optimizations + Spark?

RDDs operate on unstructured data, and there are few limits on computation; your computations are defined as functions that you've written yourself, on your own data types.

But as we saw, we have to do all the optimization work ourselves!

Wouldn't it be nice if Spark could do some of these optimizations for us?

DataFrames & Spark SQL make such optimizations possible!

We've got to give up some of the freedom, flexibility, and generality of the functional API in order to give Spark more opportunities to optimize though.

Revisiting Our Selecting Scholarship Recipients Example

Our data sets include students from many countries, with many life and financial backgrounds. Now, let's imagine that our goal is to tally up and select students for a specific scholarship.

As an example, Let's count:

- Swiss students
- who have debt & financial dependents

How might we implement this program with the DataFrame API?

```
// Remember, DataFrames available to us:
val demographicsDF = sc.textfile(...).toDF // DataFrame of Demographic
val financesDF = sc.textfile(...).toDF     // DataFrame of Finances
```

Revisiting Our Selecting Scholarship Recipients Example

With DataFrames:

```
demographicsDF.join(financesDF, demographicsDF("ID") === financesDF("ID"), "inner")
        .filter($"HasDebt" && $"HasFinancialDependents")
        .filter($"CountryLive" === "Switzerland")
        .count
```

Revisiting Our Selecting Scholarship Recipients Example

Recall

While for all three of these possible examples, the end result is the same, the time it takes to execute the job is vastly different.

Possibility 1

```
> ds.join(fs)
    .filter(p => p._2._
    .count
```

▸ (1) Spark Jobs

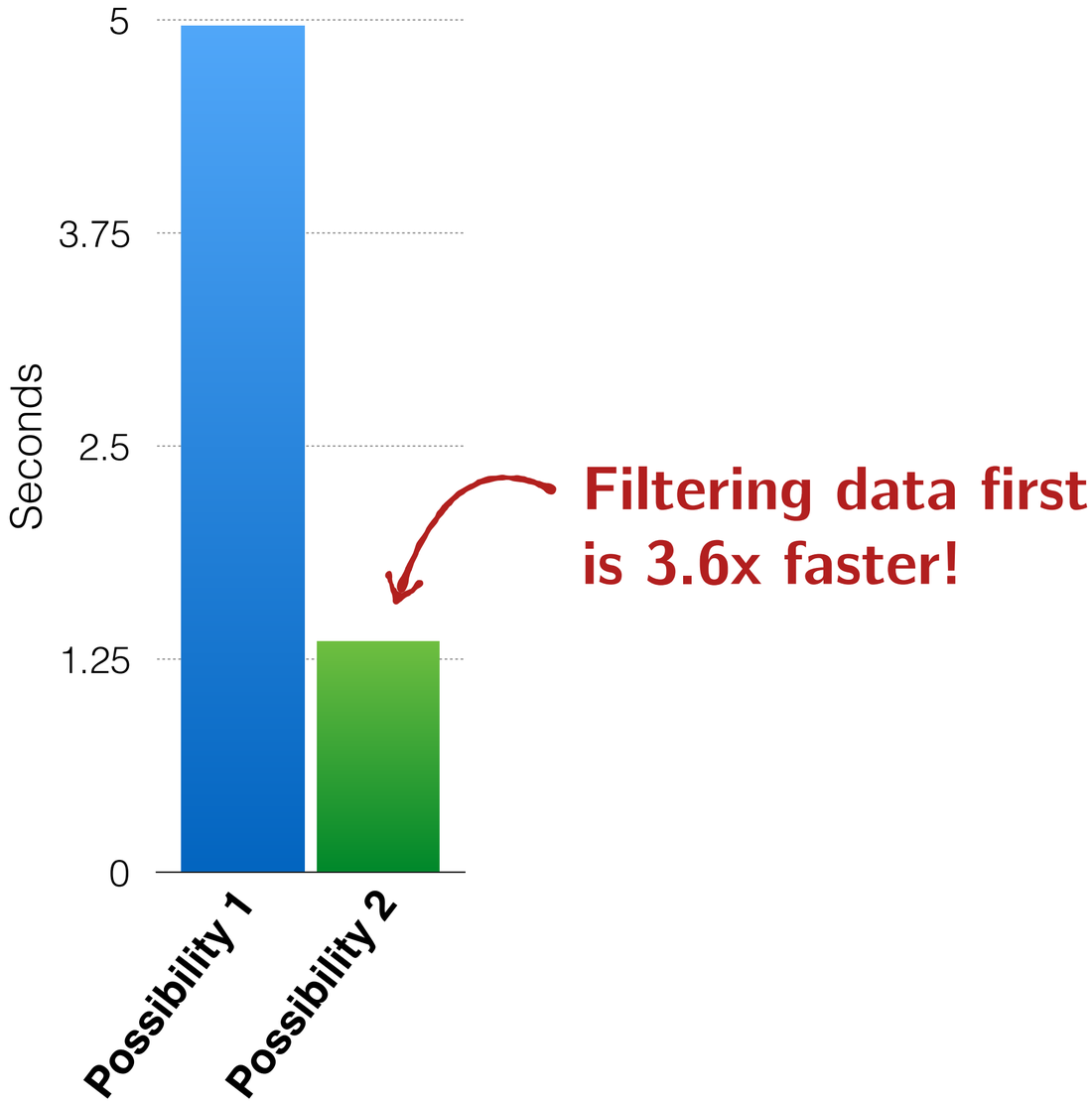
```
res0: Long = 10
Command took 4.97 seconds -
```

Possibility 2

```
> val fsi = fs.filter(
  ds.filter(p => p._2.
    .join(fsi)
    .count
```

▸ (1) Spark Jobs

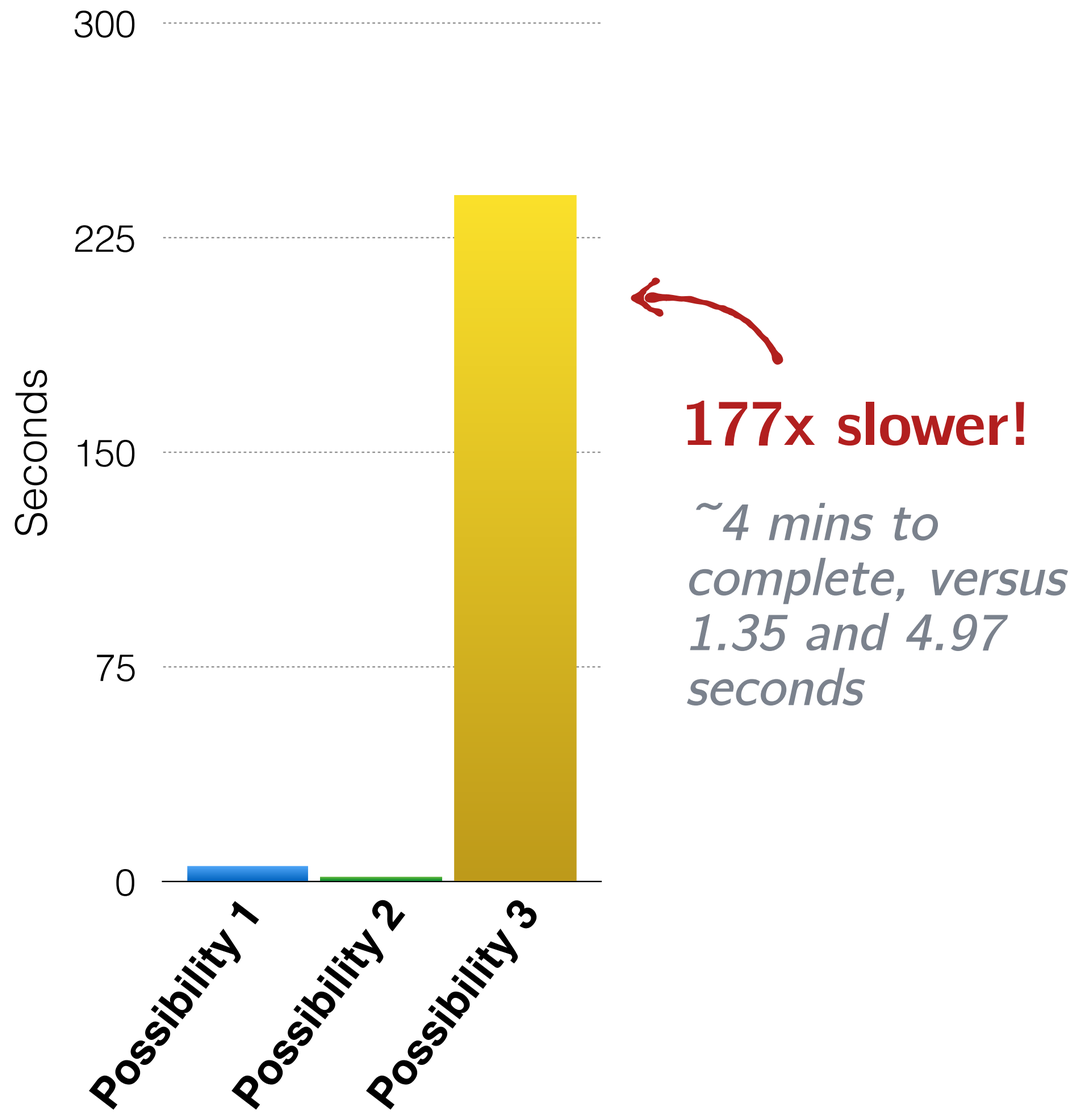
```
fsi: org.apache.spark.
res4: Long = 10
Command took 1.35 seconds
```



Revisiting Our Selecting Scholarship Recipients Example

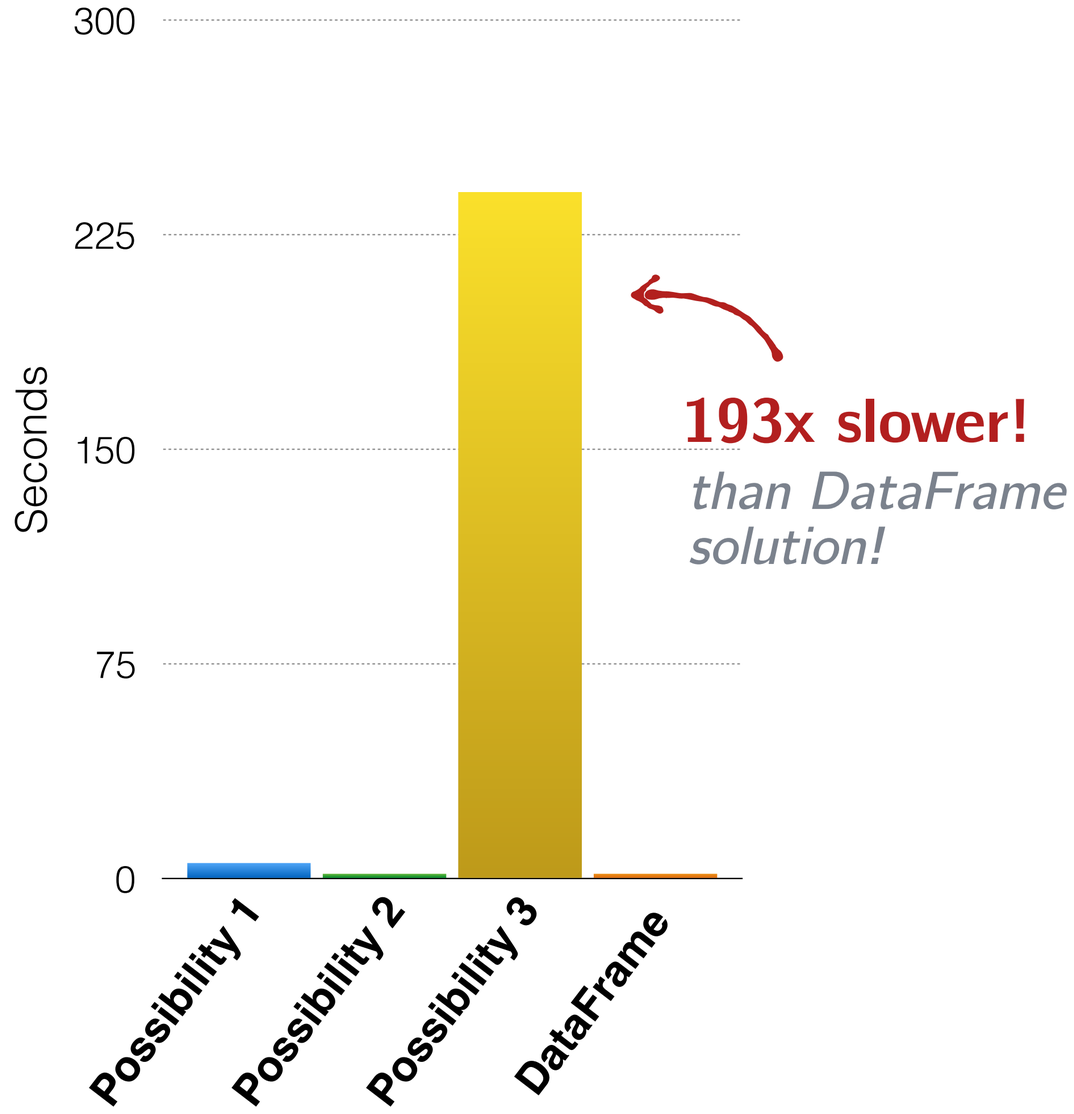
Recall

While for all three of these possible examples, the end result is the same, the time it takes to execute the job is vastly different.



Revisiting Our Selecting Scholarship Recipients Example

Comparing performance between handwritten RDD-based solutions and DataFrame solution...



Revisiting Our Selecting Scholarship Recipients Example

Comparing performance between handwritten RDD-based solutions and DataFrame solution...

Possibility 1

```
> ds.join(fs)
    .filter(p => p._2._
    .count
```

▸ (1) Spark Jobs

```
res0: Long = 10
Command took 4.97 seconds -
```

Possibility 2

```
> val fsi = fs.filter(
  ds.filter(p => p._2.
    .join(fsi)
    .count
```

▸ (1) Spark Jobs

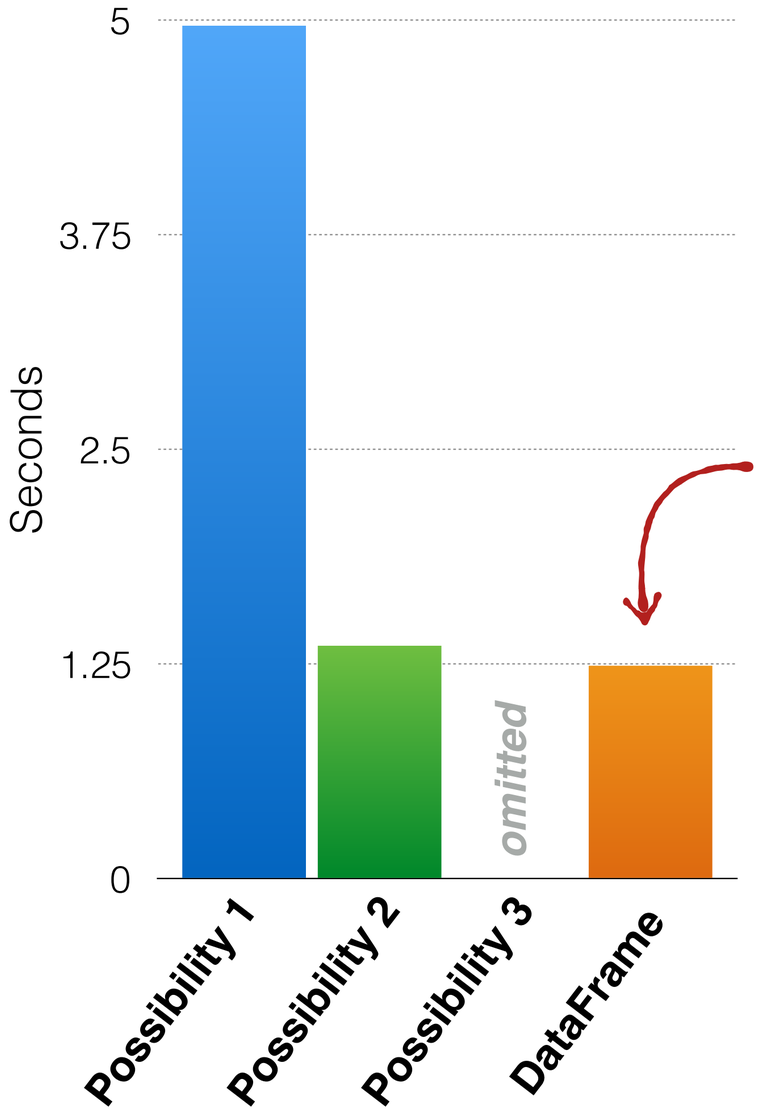
```
fsi: org.apache.spark.
res4: Long = 10
Command took 1.35 seconds
```

DataFrame

```
> demographics.join(fi
                .filter(
                .filter(
                .count
```

▸ (2) Spark Jobs

```
res24: Long = 10
Command took 1.24 seconds
```



4x faster!
than almost same program written for RDDs

Optimizations

How is this possible?

Optimizations

How is this possible?

Recall that Spark SQL comes with two specialized backend components:

- **Catalyst**, query optimizer.
- **Tungsten**, off-heap serializer.

Let's briefly develop some intuition about why structured data and computations enable these two backend components to do so many optimizations for you.