NYU | Center for Data Science

Week 05.1: Spark and RDDs

DS-GA 1004: Big Data

Map-Reduce... too low-level?

- Map-Reduce is great for one-time jobs with simple dependencies
- What if you want interactive or iterative procedures?
  - **Data exploration**
  - **Complex queries** with multiple joins and aggregations
  - **Optimization** and machine learning

Imagine doing gradient descent with MR

- $\mathbf{min}_{w} \sum_{n} f(x_{n} ; w)$
- Initialize w

Imagine doing gradient descent with MR

- $\min_{w} \sum_n f(x_n ; w)$

- Initialize w
- **Repeat** until convergence:
  - **mapper**: $x_n \to g_n = \nabla_w f(x_n ; w)$ // N map jobs, compute gradients
    emit $(1, g_n)$

Imagine doing gradient descent with MR

- $\mathbf{min}_w \sum_n f(x_n ; w)$

- Initialize w
- **Repeat** until convergence:
  - **mapper**: $x_n \rightarrow g_n = \nabla_w f(x_n ; w)$ // N map jobs, compute gradients
    emit $(1, g_n)$
  - **reducer**: $\{(1, g_n)\} \rightarrow G = \sum_n g_n$ // 1 reduce job, accumulate gradients
    emit G
  - $w \leftarrow w - G$

Imagine doing gradient descent with MR

- **min**$_w$ $\sum_n f(x_n ; w)$

- Initialize w
- **Repeat** until convergence:
  - **mapper**: $x_n \rightarrow g_n = \nabla_w f(x_n ; w)$ // N map jobs, compute gradients
    emit $(1, g_n)$
  - **reducer**: $\{(1, g_n)\} \rightarrow G = \sum_n g_n$ // 1 reduce job, accumulate gradients
    emit G
  - $w \leftarrow w - G$

Each gradient step involves a full map-reduce!

And we don't even care about the previous iterations after they're done...

Imagine doing gradient descent with MR

- **min**$_w$ $\sum_n f(x_n ; w)$

- Initialize w
- **Repeat** until convergence:
  - **mapper**: $x_n \rightarrow g_n = \nabla_w f(x_n ; w)$ // N map jobs, compute gradients
    emit $(1, g_n)$
  - **reducer**: $\{(1, g_n)\} \rightarrow G = \sum_n g_n$ // 1 reduce job, accumula
    emit G
  - $w \leftarrow w - G$

Each gradient step involves a full map-reduce!

And we don't even care about the previous iterations after they're done...

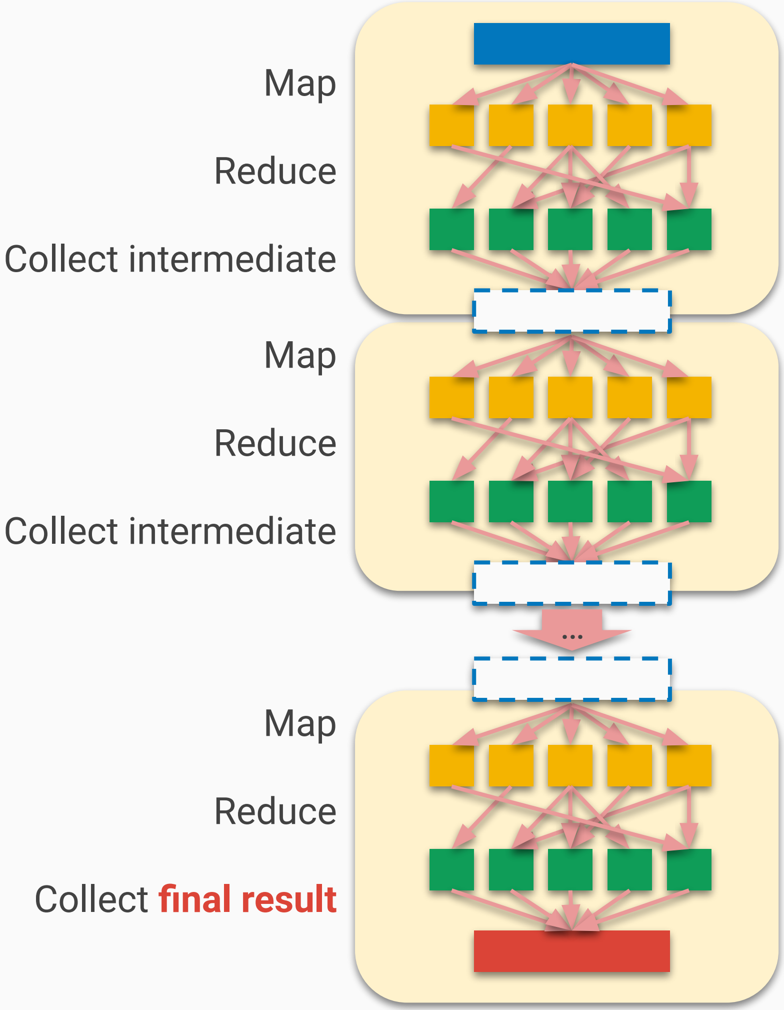
Reducer can't start until all mappers have finished
⇒ high latency

Complex pipelines

Some computations can be decomposed into a sequence of MR jobs

But this isn’t always the easiest or most natural way to do it!

What if you want to rapidly iterate?

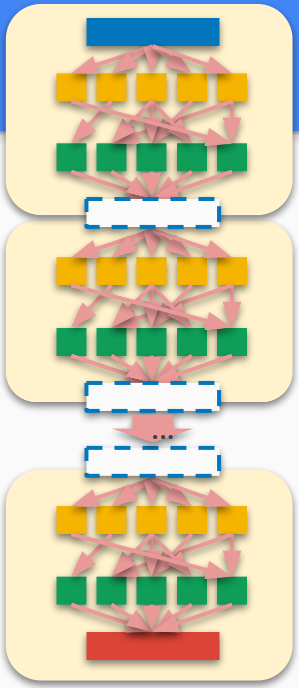


Resilient distributed datasets (RDDs)

[Zaharia et al., 2012]

Reusing data

- Complex computations usually have many **intermediate steps**
- Map-Reduce paradigm favors the following pattern:
  - Compute each step
  - Store intermediate results
  - Move on to the next step
- This can be **wasteful** and **awkward** to implement



Resilient distributed datasets (RDDs)

- RDD:
  - **Data source**
  - Lineage graph of **transformations** to apply to **data**
  - + interfaces for data **partitioning** and **iteration**

- Think of this as **deferred computation**
  - Nothing is **computed** until you ask for it
  - Nothing is **saved** until you say so
  - This makes optimization possible

Resilient distributed datasets (RDDs)

- RDD:
  - **Data source**
  - Lineage graph of **transformations** to apply to **data**
  - + interfaces for data **partitioning** and **iteration**

- Think of this as **deferred computation**
  - Nothing is **computed** until you ask for it
  - Nothing is **saved** until you say so
  - This makes optimization possible

Some notation:

RDD[T] denotes an RDD with data of type T, e.g.

- RDD[String]
- RDD[Tuple(String, Float)]

RDD example: log processing

Spark code

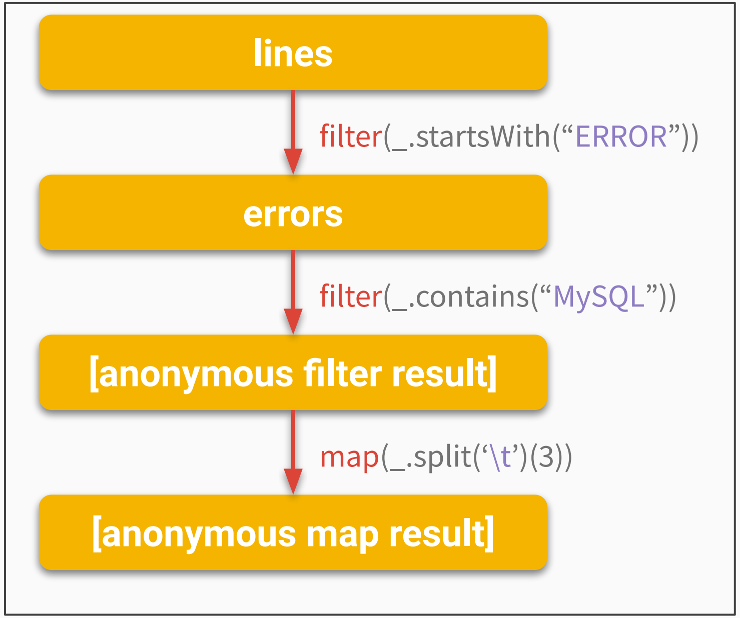
```
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))

errors.filter(_.contains("MySQL"))
      .map(_.split('\t')(3))
      .collect()
```

Legend:
Data **RDD** **Transformation** **Action**

Lineage graph



Adapted from [Zaharia et al., 2012]

RDD example: log processing

Spark code

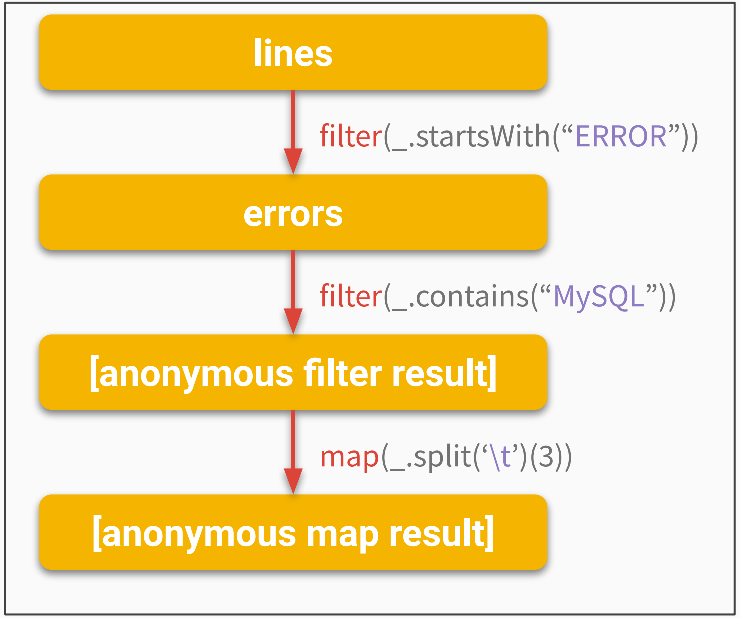
```
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))

errors.filter(_.contains("MySQL"))
      .map(_.split('\t')(3))
      .collect()
```

No computation happens until you take an **action!**

Legend:
Data **RDD** **Transformation** **Action**



Adapted from [Zaharia et al., 2012]

Transformations

Transformations turn one or more RDDs into a new RDD

Transformations are cheap to construct because they don't actually do the computation

Building an RDD is like **writing** (not ***running***) a map-reduce script or a SQL query

- Examples:
  - **map**(**function** T → U) ⇒ RDD[T] → RDD[U]
  - **filter**(**function** T → **Boolean**) ⇒ RDD[T] → RDD[T]
  - **union**() ⇒ (RDD[T], RDD[T]) → RDD[T]

```
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))

errors.filter(_.contains("MySQL"))
      .map(_.split('\t')(3))
      .collect()
```

Actions

Actions are what execute computation defined by an RDD

Results of actions are not RDDs

- Examples:
  - **count**() ⇒ RDD[T] → Integer
  - **collect**() ⇒ RDD[T] → Sequence[T]
  - **reduce**(**function** (T, T) → T) ⇒ RDD[T] → T
  - **save**(**path**) ⇒ Save RDD to file system or HDFS

```
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))

errors.filter(_.contains("MySQL"))
      .map(_.split('\t')(3))
      .collect()
```

Work backwards from actions

```
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))

errors.filter(_.contains("MySQL"))
      .map(_.split('\t')(3))
      .collect()
```

1. **collect**() depends on **map**()
2. **map**() depends on **filter**(MySQL)
3. **filter**(MySQL) depends on **filter**(ERROR)
4. **filter**(ERROR) depends on **lines**

Work backwards from actions

```
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))

errors.filter(_.contains("MySQL"))
      .map(_.split('\t')(3))
      .collect()
```

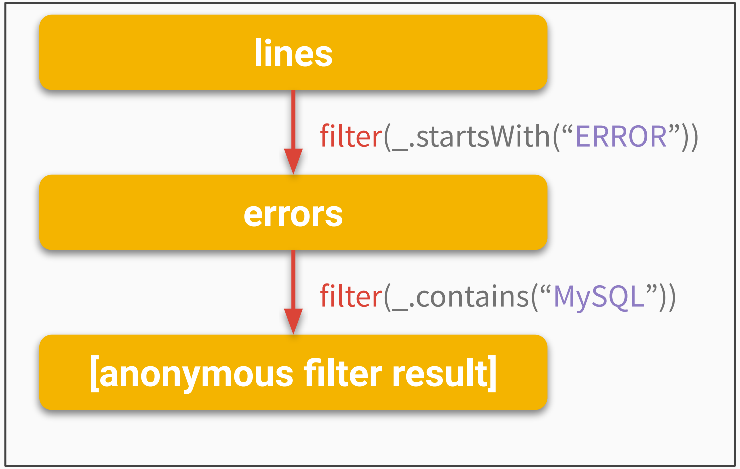
Any previously computed RDDs can be **cached** and reused!

Any lost / corrupted RDDs can be rebuilt from scratch by tracing the **lineage**!

1. **collect**() depends on **map**()
2. **map**() depends on **filter**(MySQL)
3. **filter**(MySQL) depends on **filter**(ERROR)
4. **filter**(ERROR) depends on **lines**

Pipelines

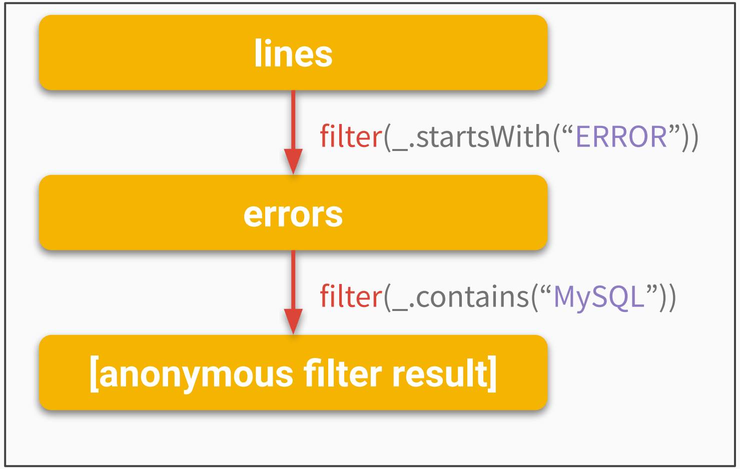
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK Status OK ERROR: Rampaging T-Rex Status OK ERROR: MySQL failure Status OK ERROR: Utahraptor ate my lunch		

Pipelines

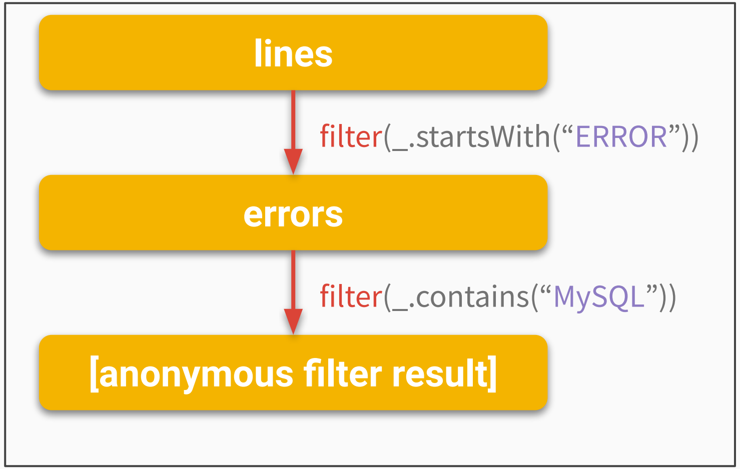
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK **Status OK** ERROR: Rampaging T-Rex Status OK ERROR: MySQL failure Status OK ERROR: Utahraptor ate my lunch		

Pipelines

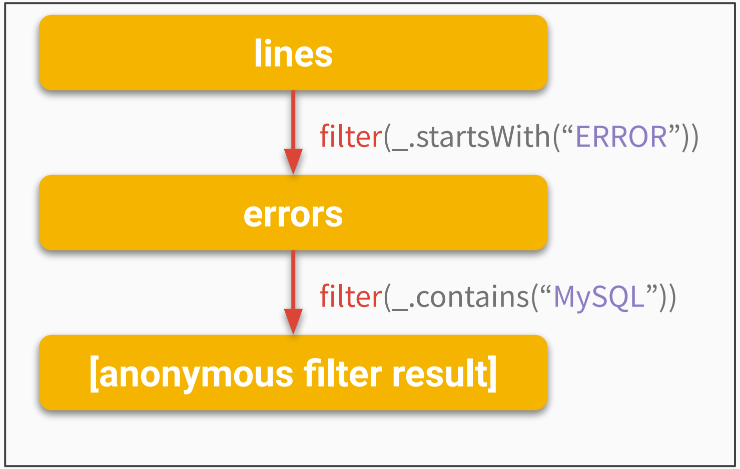
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK Status OK **ERROR: Rampaging T-Rex** Status OK ERROR: MySQL failure Status OK ERROR: Utahraptor ate my lunch	**ERROR: Rampaging T-Rex**	

Pipelines

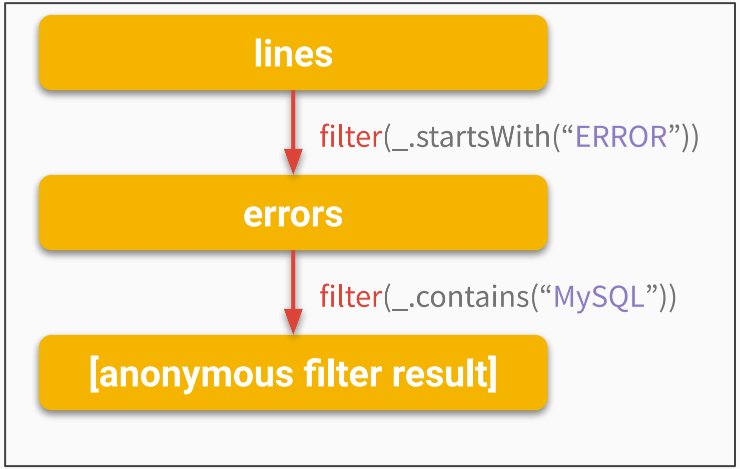
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK Status OK ERROR: Rampaging T-Rex **Status OK** ERROR: MySQL failure Status OK ERROR: Utahraptor ate my lunch	ERROR: Rampaging T-Rex	

Pipelines

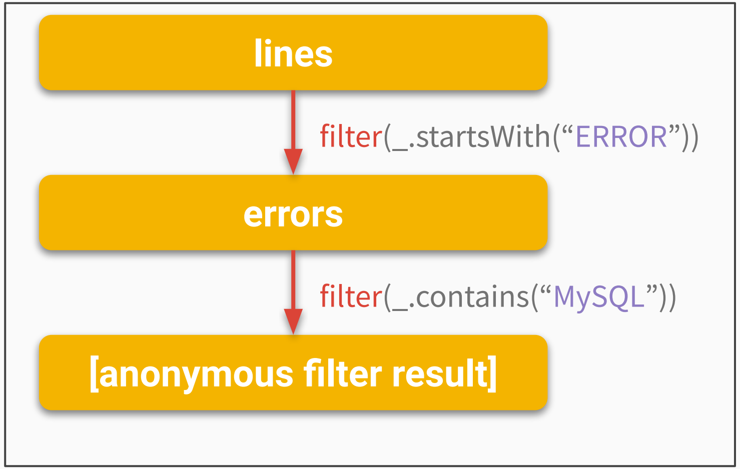
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK Status OK ERROR: Rampaging T-Rex Status OK **ERROR: MySQL failure** Status OK ERROR: Utahraptor ate my lunch	ERROR: Rampaging T-Rex **ERROR: MySQL failure**	**ERROR: MySQL failure**

Pipelines

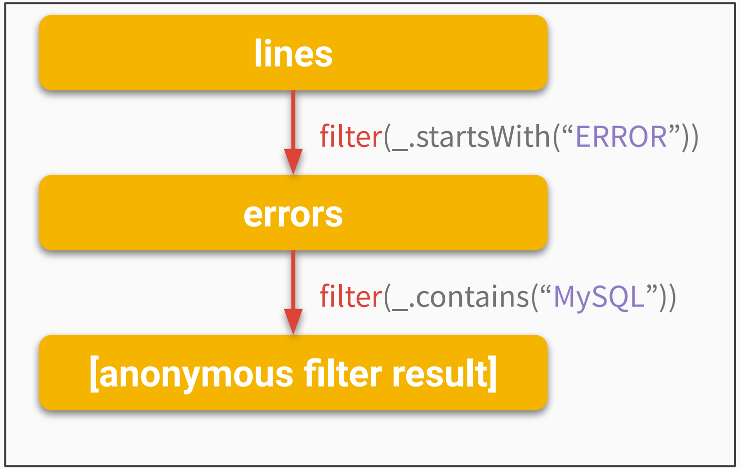
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK	ERROR: Rampaging T-Rex	ERROR: MySQL failure
Status OK	ERROR: MySQL failure	
ERROR: Rampaging T-Rex		
Status OK		
ERROR: MySQL failure		
Status OK		
ERROR: Utahraptor ate my lunch		

Pipelines

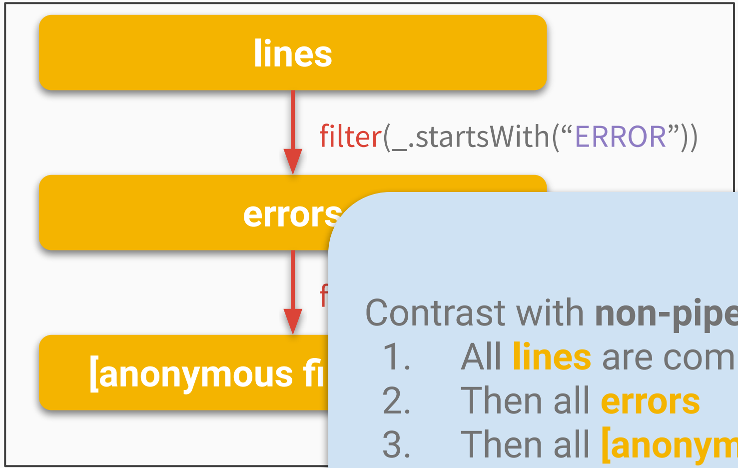
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK Status OK ERROR: Rampaging T-Rex Status OK ERROR: MySQL failure Status OK **ERROR: Utahraptor ate my lunch**	ERROR: Rampaging T-Rex ERROR: MySQL failure **ERROR: Utahraptor ate my lunch**	ERROR: MySQL failure

Pipelines

- Lineages can be **pipelined**
- We don’t need to wait for all of lines to finish to build errors
- **No need for intermediate storage like in Map-Reduce**



Contrast with **non-pipelined** implementation:

1. All lines are computed
2. Then all errors
3. Then all [anonymous filter]

lines	errors	[anonymous filter]
Status OK Status OK ERROR: Rampaging T-Rex Status OK ERROR: MySQL failure Status OK **ERROR: Utahraptor ate my lunch**	ERROR: Rampaging T-Rex ERROR: MySQL failure **ERROR: Utahraptor ate my lunch**	ERROR: MySQL failure

The RDD interface

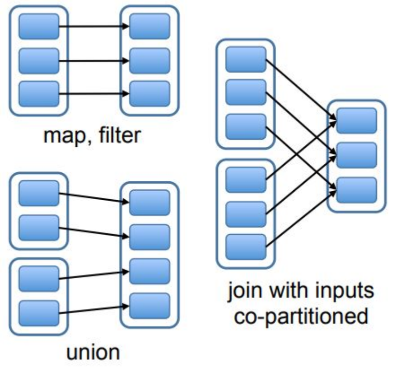
- **partitions**() → Returns a list of *Partitions* (analogous to **splits** in map-reduce)
- **preferredLocations**(*p*) → (HDFS) Nodes where partition *p* can be found
- **dependencies**() → Get the dependencies for this RDD
- **iterator**(*p*, *parentIters*) → Get elements of partition *p*, given iterators for parent partitions
- **partitioner**() → Get metadata about how the RDD is partitioned (e.g., is it a hash or range?)

Narrow and wide dependencies

Narrow dependencies

Partition of parent RDD goes to at most 1 partition of child RDDs

- Low communication
- Localized
- Easy to pipeline
- Easy failure recovery



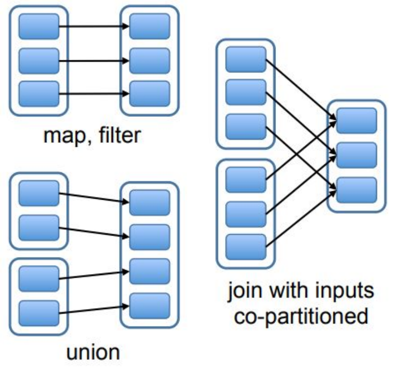
Figures adapted from [Zaharia et al., 2012]

Narrow and wide dependencies

Narrow dependencies

Partition of parent RDD goes to at most 1 partition of child RDDs

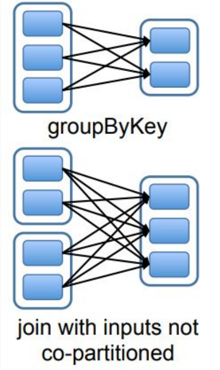
- Low communication
- Localized
- Easy to pipeline
- Easy failure recovery



Wide dependencies

Partition of parent RDD goes to multiple child RDD partitions

- High communication
- High latency
- Difficult to pipeline
- Difficult to recover



Figures adapted from [Zaharia et al., 2012]

RDDs

- Resilient Distributed Datasets (RDDs) and the fundamental data structure of Spark
- Spark uses deferred computation to more efficiently construct complex analyses
    - Transformations and actions!
- RDD partitions are analogous to map-reduce splits, and allow parallel execution