Smallest font

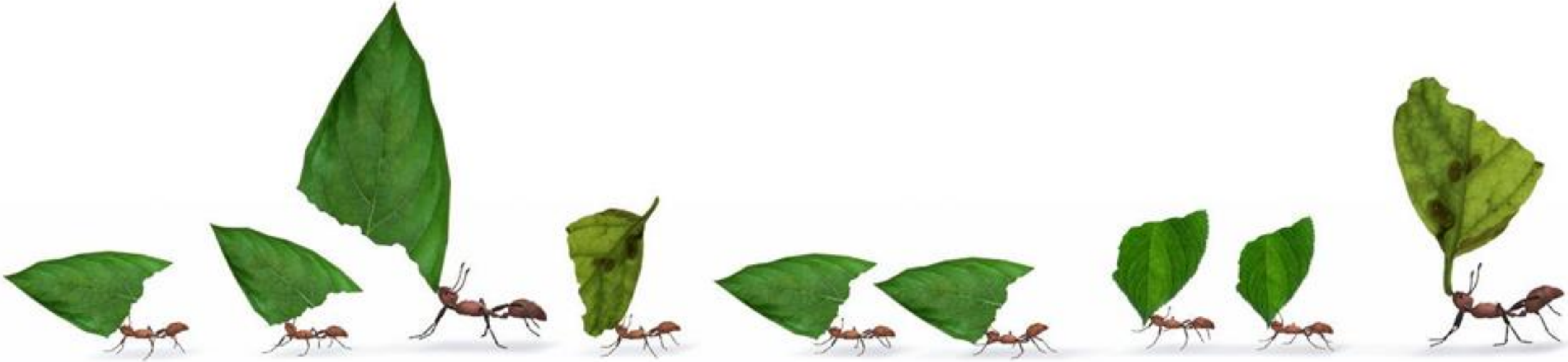
Please turn off and put away your cell phone

Calibration slide

These slides are meant to help with note-taking
They are no substitute for lecture attendance

Smallest font

Big Data



This week

1. Finishing up Big Data infrastructure

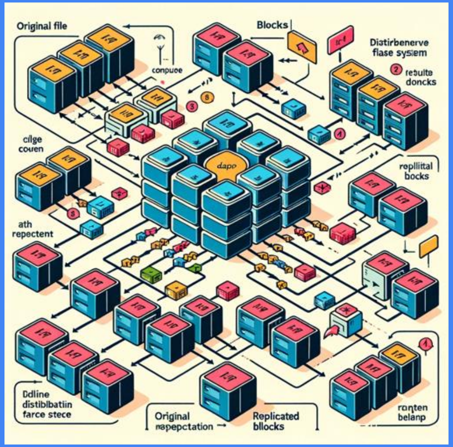
2. Introducing Spark

NYU | Center for Data Science

DS-GA 1004: Big Data

1) Big Data Infrastructure

Taking a closer look at the Hadoop framework

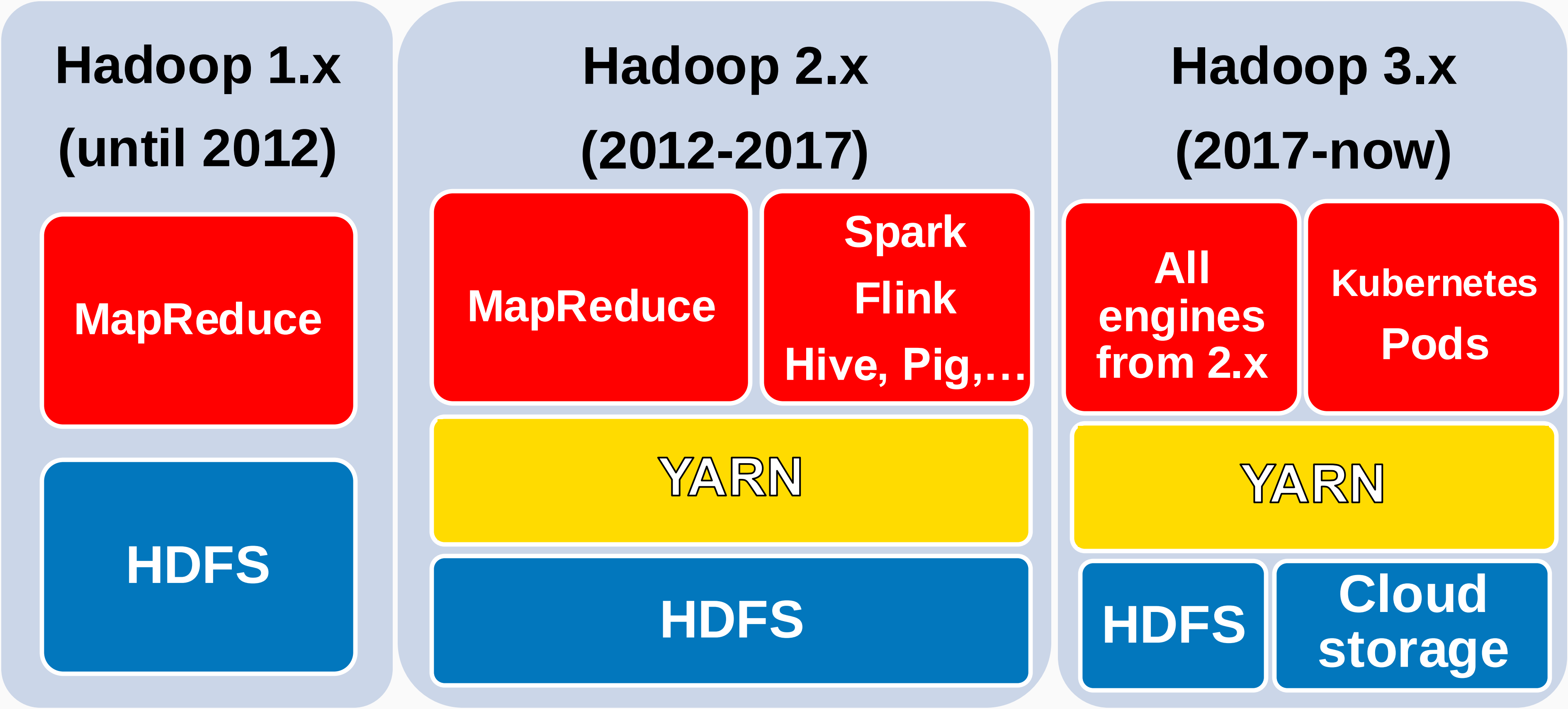


MapReduce
Processing engine

YARN
Resource manager

HDFS
Storage layer

YARN was added to the Hadoop framework in version 2.0



YARN has effectively become the operating system of Hadoop

Yarn components:

Resource manager

- Resource Scheduler
- Application Manager

Node manager		Node manager		Node manager	
Container	Application Master	Container	Application Master	Container	Application Master
Data node		Data node		Data node	

Actual footage of the interaction between resource manager and the application masters

Cluster resources: Some terminology

=

NOT a docker container

Container

Storage space

Processing cores

Memory (RAM)

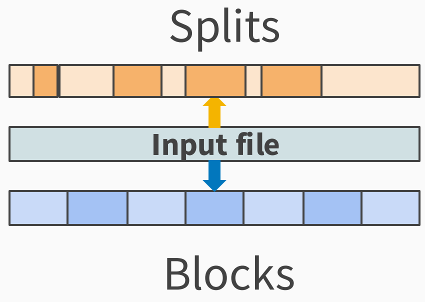
Why should you care about implementation details?

Good example: The interplay between HDFS and Map-Reduce

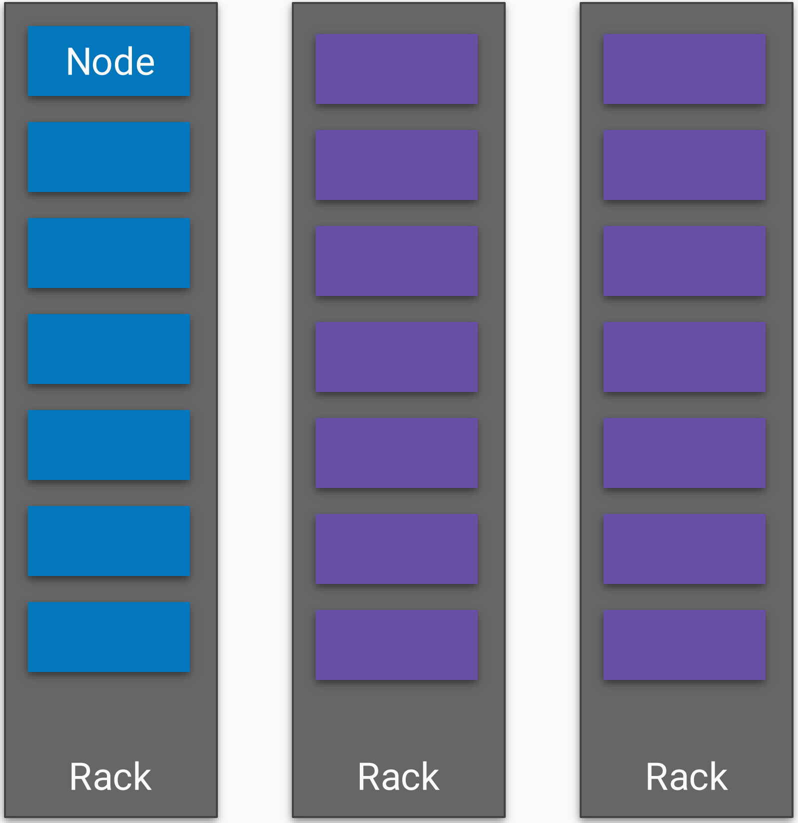
- HDFS shares **blocks** over data nodes
- Map-Reduce shares **jobs** over compute nodes
- What would be the case in an ideal world?
- If these were happening on the *same* node
  - For big data, bringing **compute ⇒ data** is cheaper than the other way around!

So job scheduling and input splits can be coordinated / optimized

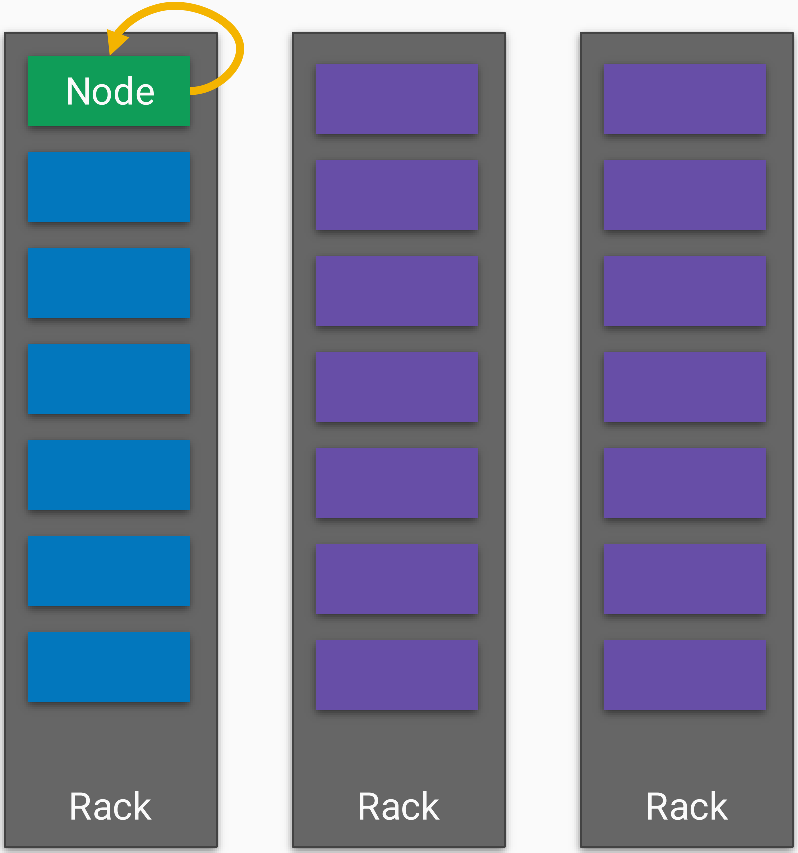
- A typical map-reduce job runs over one large data file
  - Each file contains a large number of (independent) records
- MapReduce divides the input into **splits**
- Each **split** maps onto one or more **blocks**
  - Optimal: Assign work such that processing of a **split** is done on a machine with its **blocks**
- HDFS exposes block layout to the job scheduler to make this possible



Cluster organization: Network topology and interconnections between machines



Where to execute a job?

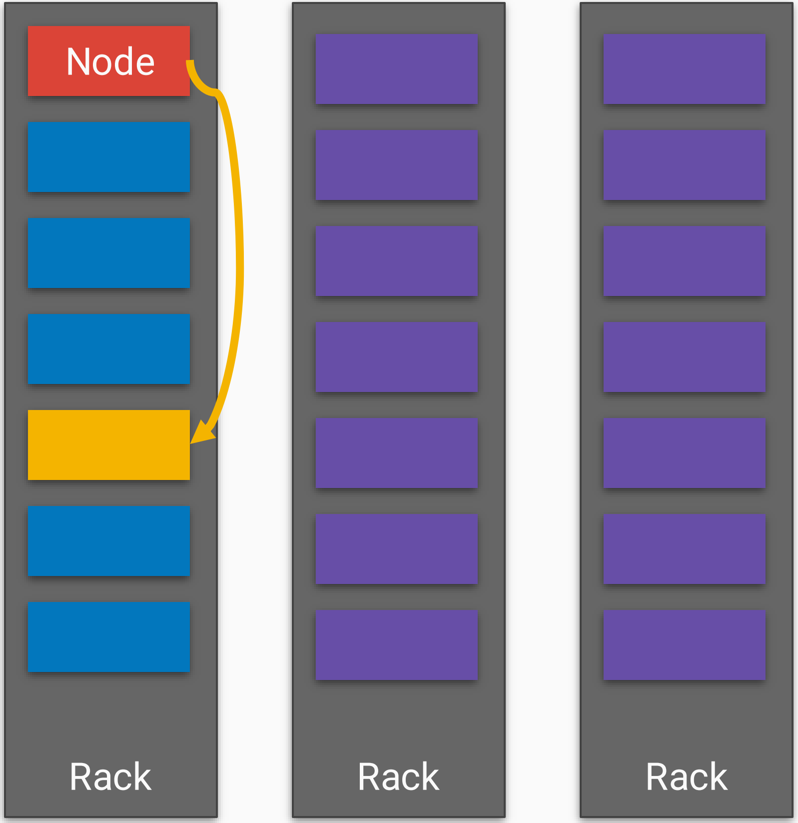


Best case:

Execute on a node that stores the block(s) we need

https://commons.wikimedia.org/wiki/File:CERN_Server.jpg
Florian Hirzinger – www.fh-ap.com, CC BY-SA 3.0 via Wikimedia Commons

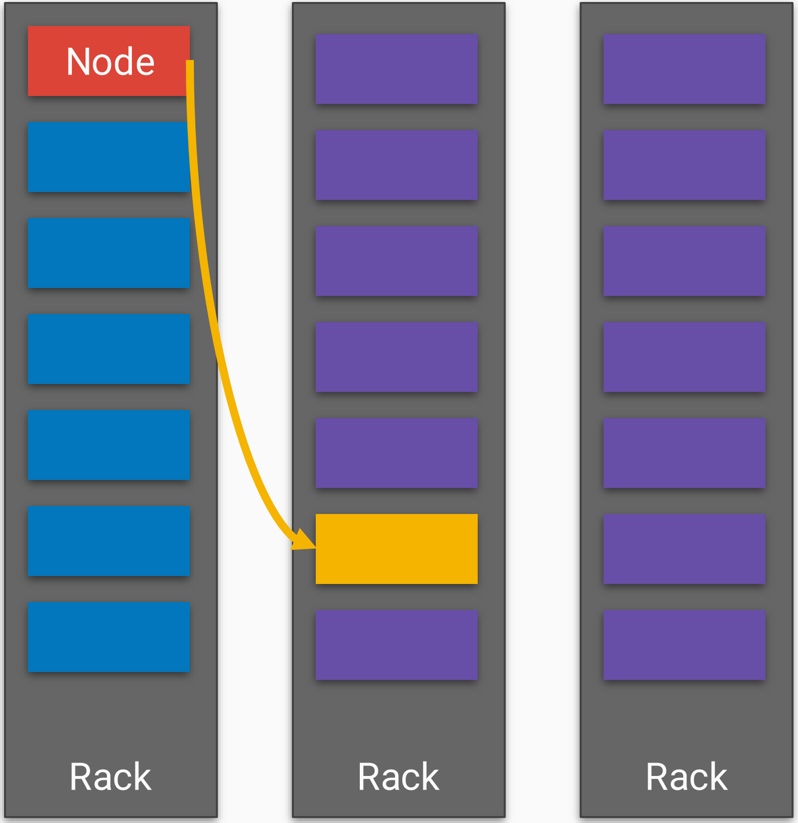
Where to execute a job?



Okay case:

Execute on a different node in the same rack
Within-rack communication is relatively fast

Where to execute a job?



Worst case:

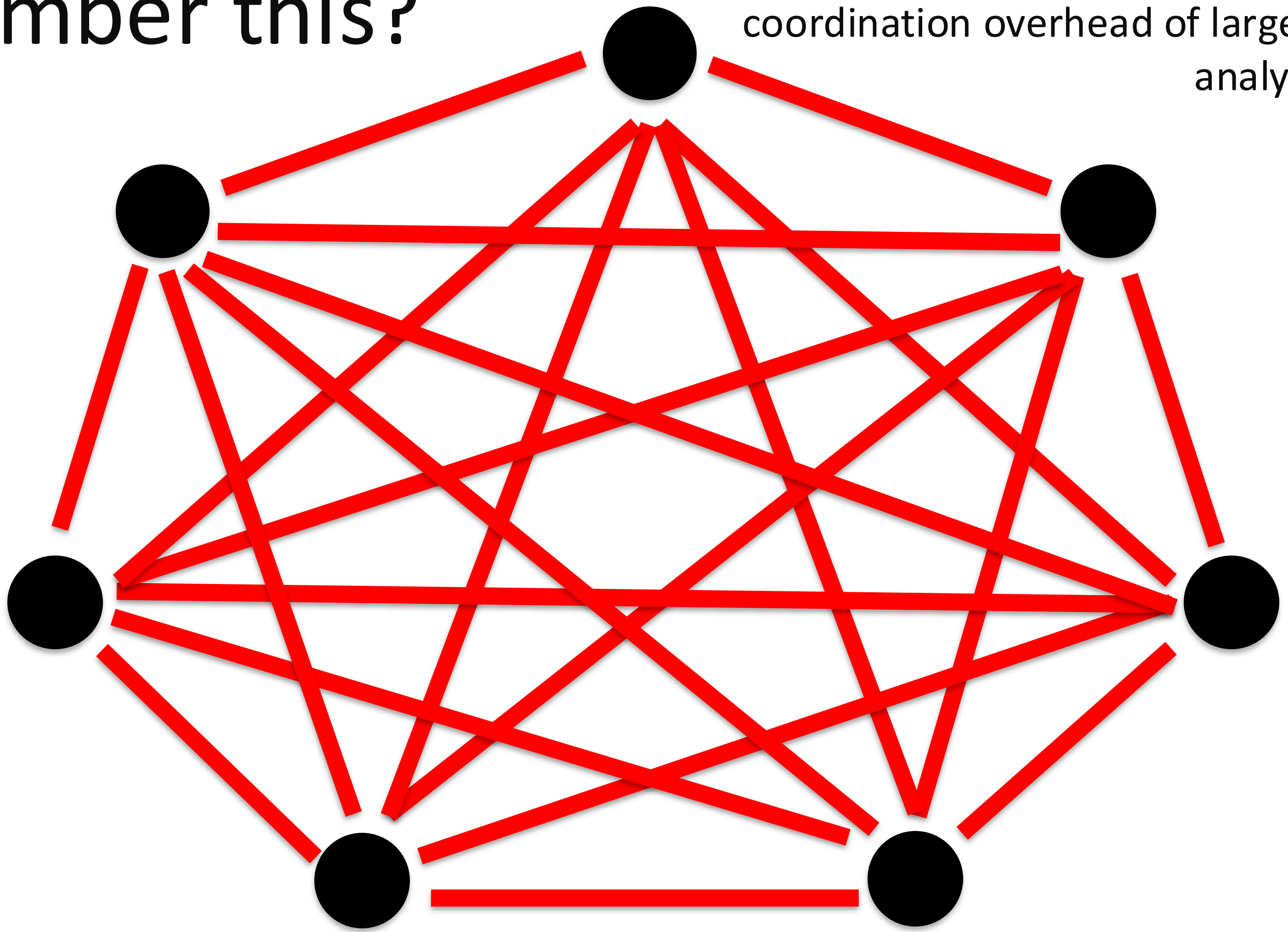
Execute on a node in a different rack
Between-rack communication is slow

Tied in with jobs: Replication factors

- Distribution of blocks to data nodes is not random.
- If we copy a block to multiple nodes, scheduling becomes easier
  - We're more likely to find a free worker that has the data we need for a given job
- HDFS lets you set the replication factor for each file
  - Replication isn't free: cost is a multiple of data size
- Default setup: 3x replication
  - If possible, 2 nodes in one rack, +1 in a separate rack
  - This protects against both **node failure** and **rack failure**

Remember this?

It’s about the necessary communication / coordination overhead of large scale (data analysis) projects



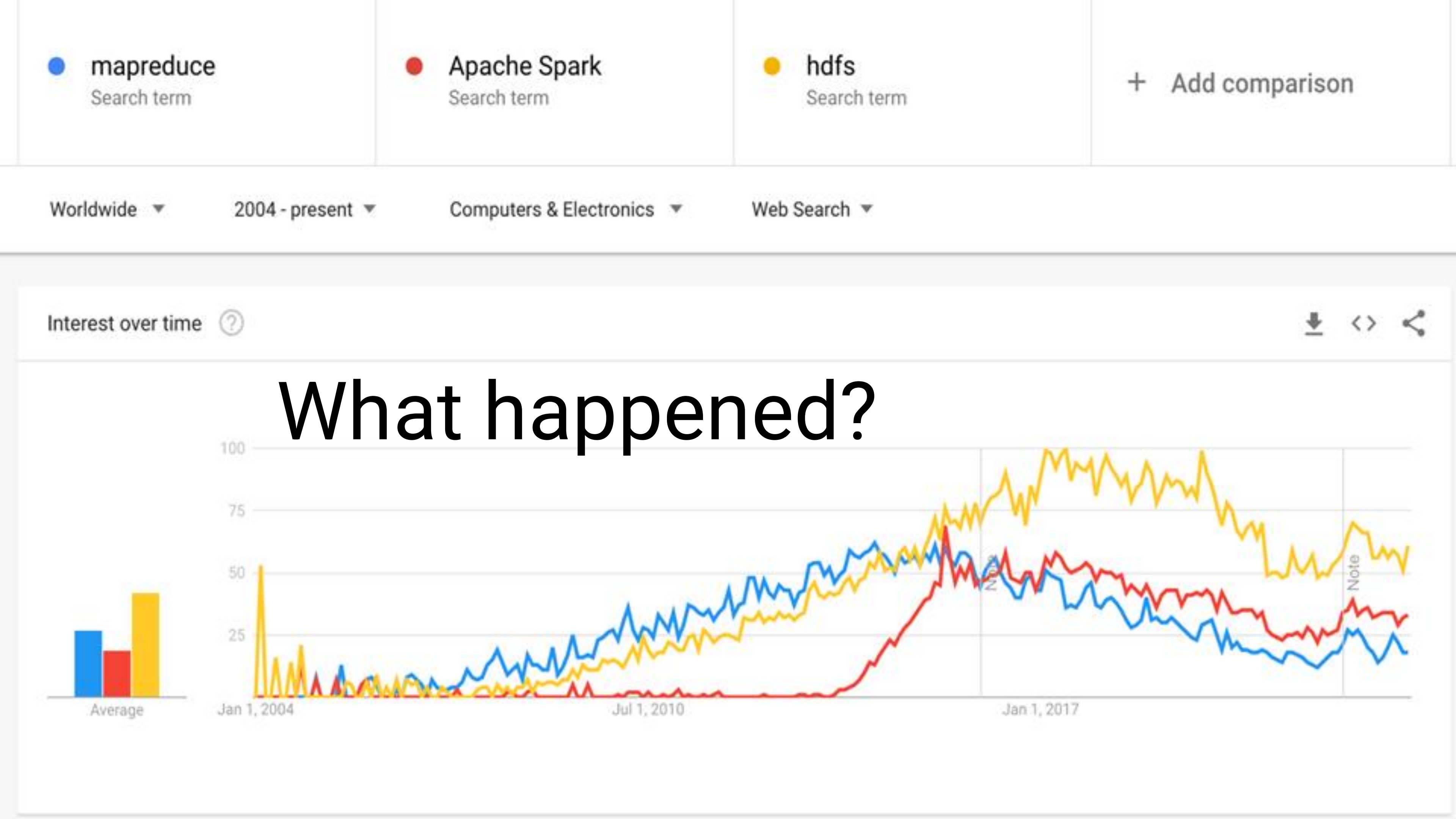
The name of the game: Parallelizing large and complex data analysis tasks

- Everything we have done so far was in the service of doing so.
- Mostly by imposing restrictions.
- Restricting valid data structures (e.g. schemas)
- Restricting valid processing operations (e.g. MapReduce)
- Restricting storage modes (e.g. HDFS)

Is this too restrictive?

2) Spark

What happened?




What do they think about mapReduce and why did they create Spark?

Good:

- Scalability (allowing for parallel processing of “big data”)
- Fault tolerance
- So it works with off-the-shelf hardware (reliable despite unreliable components)
- Takes care of most of the “plumbing” (e.g. scheduling, load-balancing)

The key issue

- The “Stonebraker” criticism notwithstanding (“it’s a bad database”):
- mapReduce is built on an “acyclic data flow model”

What do they mean by that?
Is Map-Reduce... too low-level?

- Map-Reduce is great for one-time jobs with simple dependencies, just on big data. Fine for search, not for Data Science / Machine Learning:
- What if you want interactive or iterative procedures?
  - **Data exploration (EDA)**
  - **Complex queries** with multiple joins and aggregations
  - **Optimization** and machine learning

Reminder from IDS: Gradient descent algorithm, on extremely *small* data

x	y
2	10

x: Cans of red bull
y: Price in $
β: Price per can

For terminological reasons, ŷ is called "a" and β is typically called "w" in this framework (because it is often used in the context of neural networks)

Step size = "learning rate":

$a_0 = 18 \quad C_0 = 64$

$C = (18-10)^2$

$a = x\,w$

$lR = 0.1$

$C = (a - y)^2$

$w_0 = 9$ (initialized randomly)

$$\frac{\partial C}{\partial w} = \frac{\partial a}{\partial w}\frac{\partial C}{\partial a}$$

$$\frac{\partial C}{\partial a} = 2(a - y) = 2a - 20 = 2(xw) - 20 = 4w - 20$$

$$\frac{\partial a}{\partial w} = x = 2 \qquad \frac{\partial C}{\partial w} = 8w - 40$$

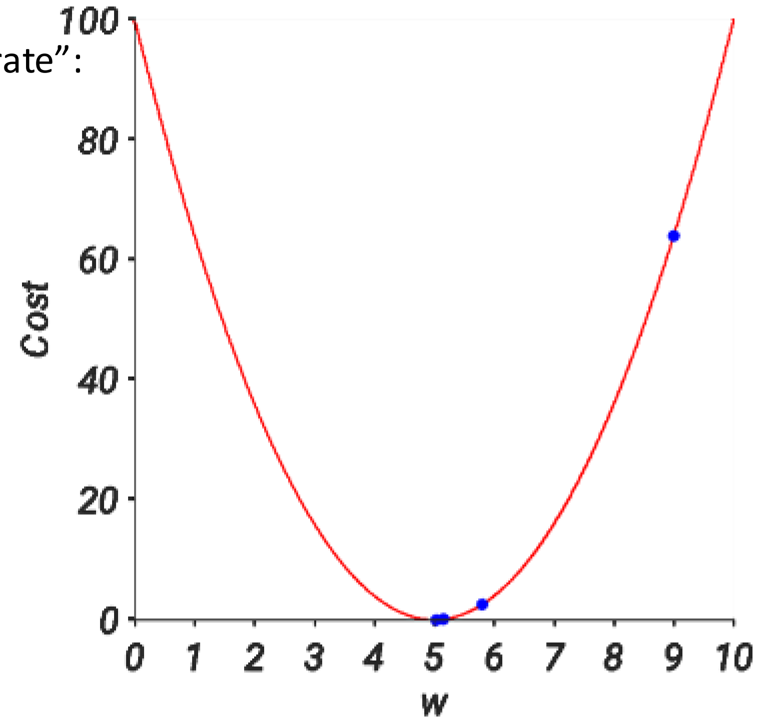
Gradient descent is iterative:

$$w_1 = w_0 - (lR\,\frac{\partial C}{\partial w})$$

$w_1 = 9 - (0.1 * (8*9 - 40)) = 5.8 \qquad a_1=11.6 \quad C_1=2.56$

$w_2 = 5.8 - (0.1 *(8*5.8 - 40)) = 5.16 \qquad a_2=10.32 \quad C_2=0.1$

$w_3 = 5.16 - (0.1 * (8*5.16 - 40)) = 5.03 \qquad a_3=10.06 \quad C_3=0.004$



w ≈ 5.03, with some extra steps

Imagine implementing gradient descent on Big Data with MapReduce

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

Imagine implementing gradient descent on Big Data with MapReduce

- **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 $(1, g_n)$

Imagine implementing gradient descent on Big Data with MapReduce

- **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 implementing gradient descent on Big Data with MapReduce

- $\min_w \sum_n f(x_n ; w)$

- 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 implementing gradient descent on Big Data with MapReduce

- **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...

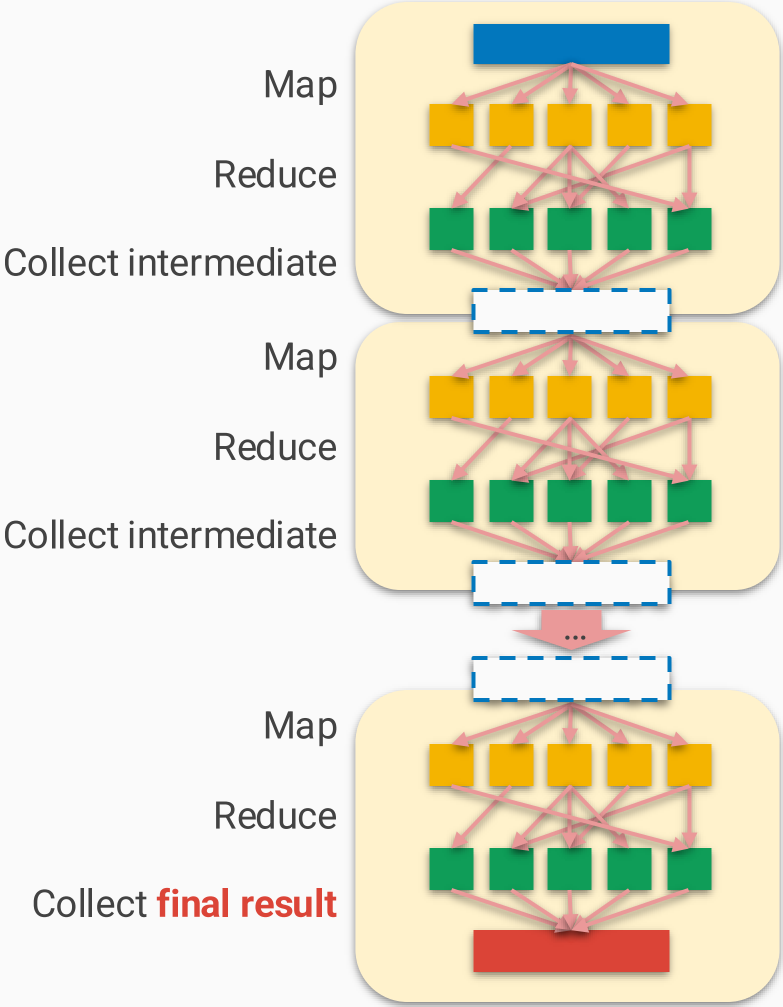
Reducer can't start until all mappers have finished $\Rightarrow$ high latency

Complex pipelines

Computations can be decomposed into a sequence of MapReduce jobs

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

What if you want to rapidly iterate?



Upshot: Many/most commonly used machine learning methods rely on iterative algorithms (e.g. maximum likelihood estimation, gradient descent, kMeans, E/M algorithm, etc.)

Whereas it is possible to implement these with mapReduce, it is clunky and slow:

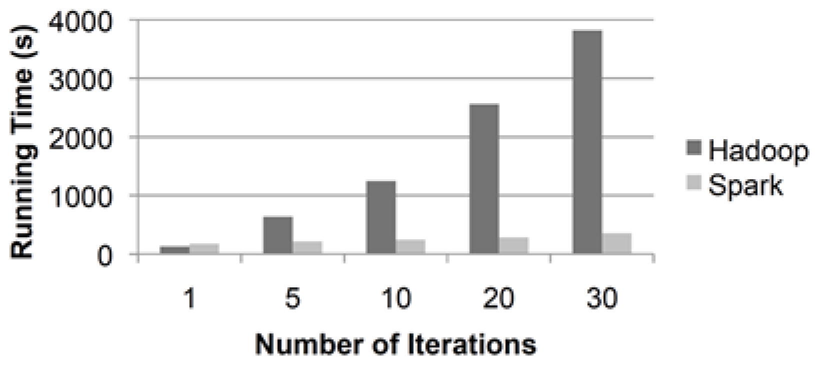


Figure 2: Logistic regression performance in Hadoop and Spark.

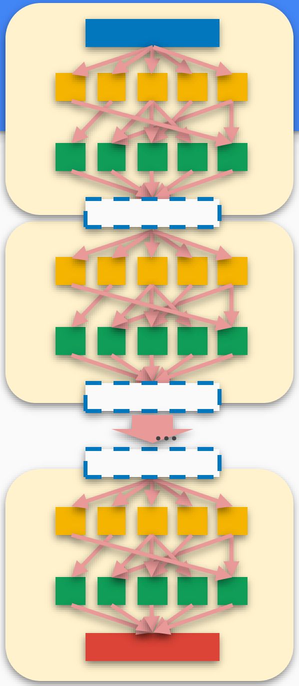
10x+ speedup for logistic regression

The proposed solution rests on a more flexible data structure:

Resilient distributed datasets (RDDs)

[Zaharia et al., 2012]

The key idea: 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**
- A key concept: **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:
  - Linked to a **data source**
  - Lineage graph of **transformations** to apply to **data**
  - + interfaces for data **partitioning** and **iteration**
  - Immutable

- 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 some data of type T, e.g.

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

RDD components: Implementing deferred computation

- **Transformations**: Operations on RDDs that return a new RDD. They are "**lazy**" (not executed immediately), but the computational steps are recorded in a **lineage graph**. Allows to efficiently create complex data processing pipelines.
- Examples: *map*, *filter*, *join*
- **Actions**: Trigger computation and yield results.
- Examples: *count*, *collect*, *reduce*, *take*, *save*

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**

A math professor, a lumberjack, and a sysadmin are on a camping trip. They're sitting around the campfire when the sysadmin says, "Hey, let's take a look at the log and see what's been going on with the system."

The math professor responds, "Ah, you mean like the logarithmic function? That's a fascinating topic!"

The lumberjack chimes in, "No, no, I think he means the logs I've been cutting down. We could use them to keep the fire going."

The sysadmin shakes his head and says, "No, I mean the system log files. We can use them to troubleshoot any issues with the computer system."

The math professor looks at the lumberjack and says, "Well, I suppose we could use the logs to represent the logarithmic function in a visual way."

The lumberjack looks at the math professor and says, "Wait, what? Logs and logarithms are the same thing?"

The sysadmin laughs and says, "No, no, they're not the same thing at all. I just want to take a look at the log files on the computer system."

The math professor, lumberjack, and sysadmin all look at each other, realizing that they've been talking about completely different things.

The lumberjack shrugs and says, "Well, at least we've got plenty of real logs to keep the fire going!"

RDD example: log processing

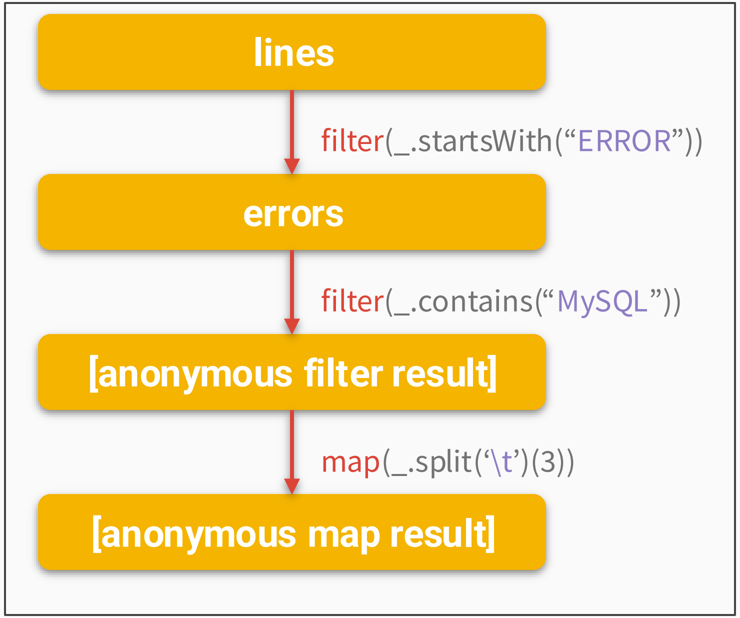
Spark code

```
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**

Lineage graph



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 the computations 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()
```

Spark works backwards from actions towards the data source (through transformations)

```
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**
5. **lines** depends on **textfile**

Spark works backwards from actions towards the data source (through transformations)

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

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

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

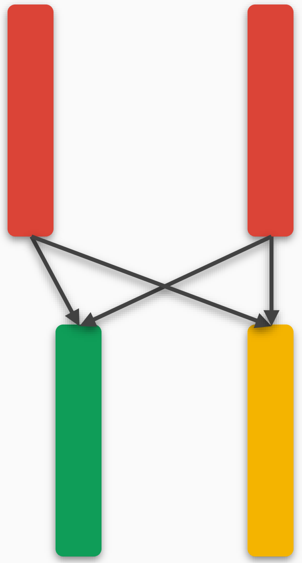
Any previously computed RDDs can be **cached** and reused!

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**
5. **lines** depends on **textfile**

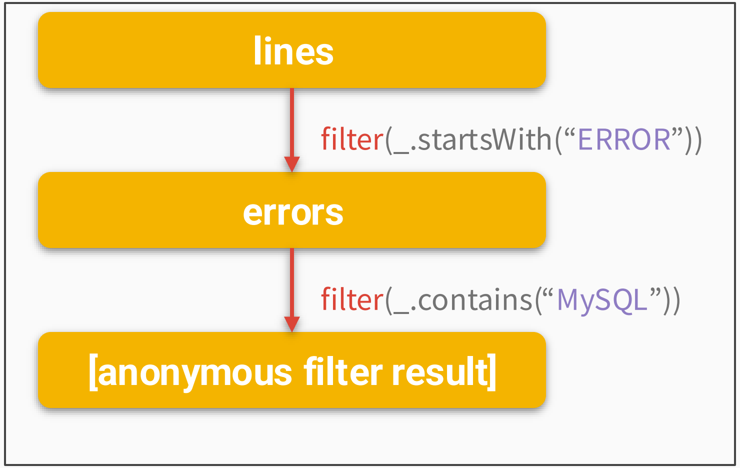
The concept of a lineage graph

- It’s called a “lineage graph”, but it need not be linear!
- **Any RDD** can depend on **multiple parent RDDs**
- Once a **parent RDD** has been computed,
  it can be cached and reused by **multiple descendents**!
- This ability to reuse RDDs is what makes Spark so efficient for iterative algorithms.



Pipelines

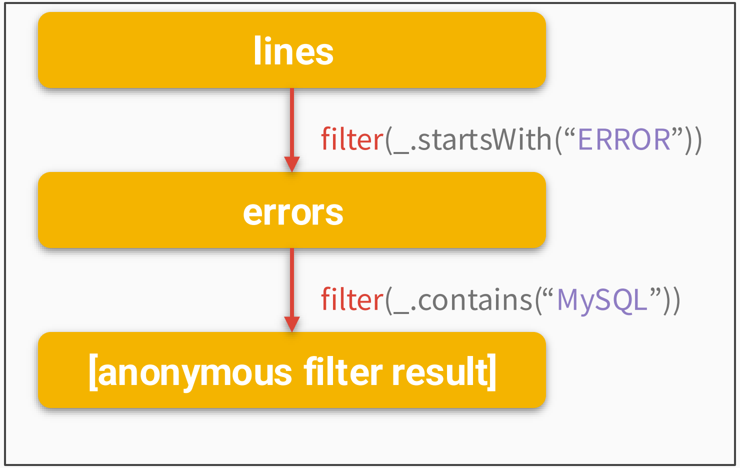
- Lineages can be **pipelined**
- We don’t need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



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

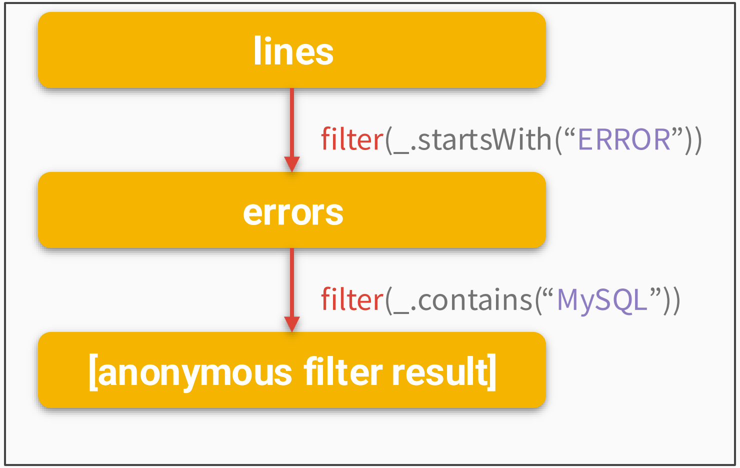
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



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

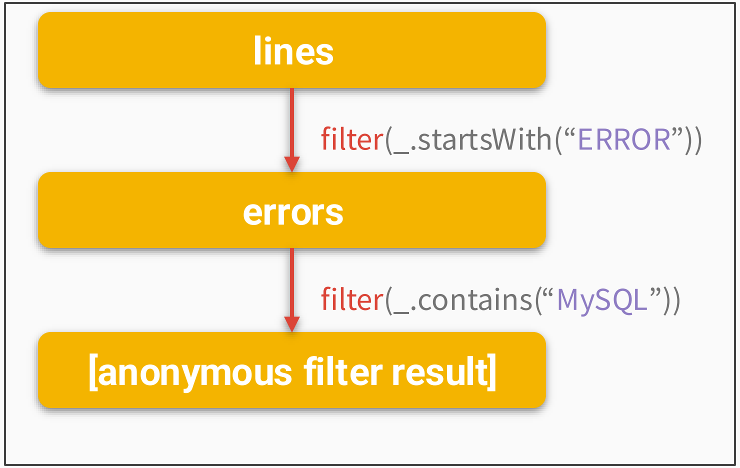
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



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

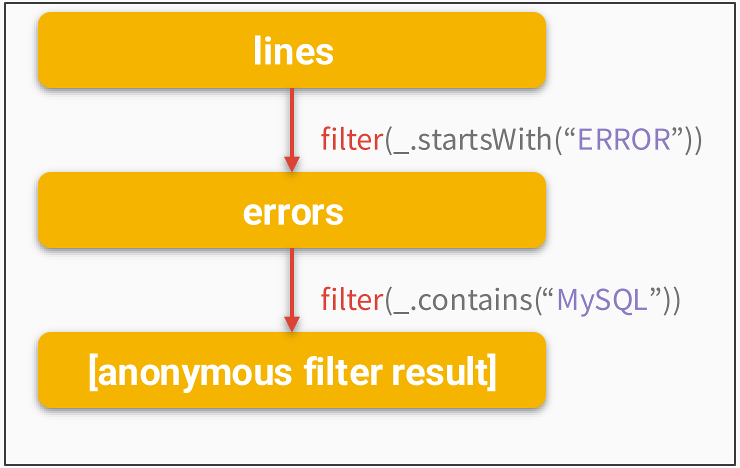
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



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

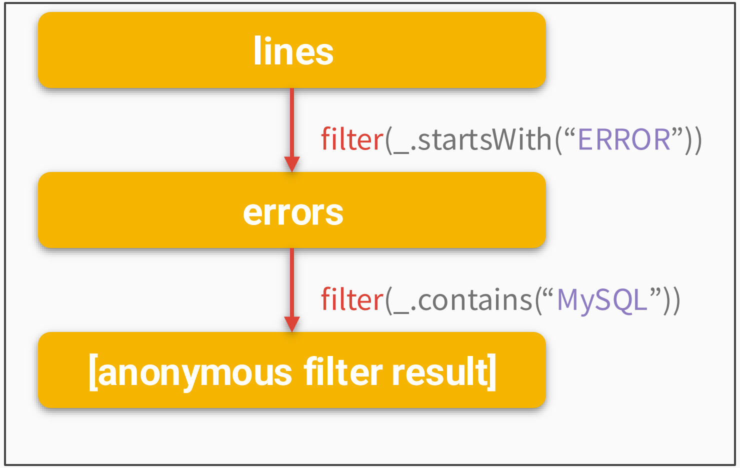
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK	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

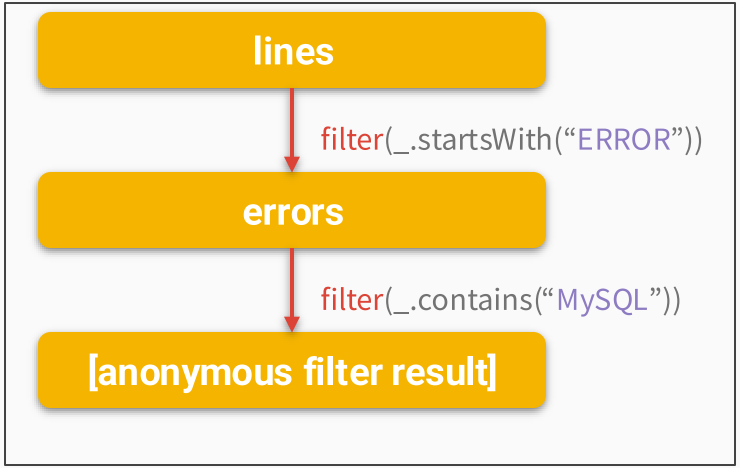
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK	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

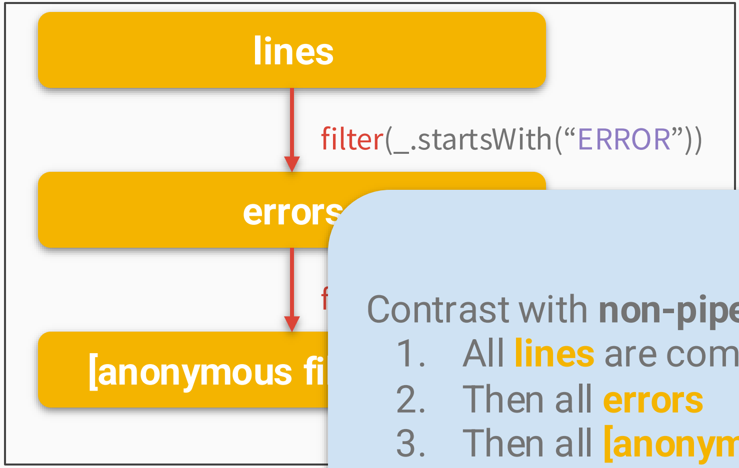
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



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**

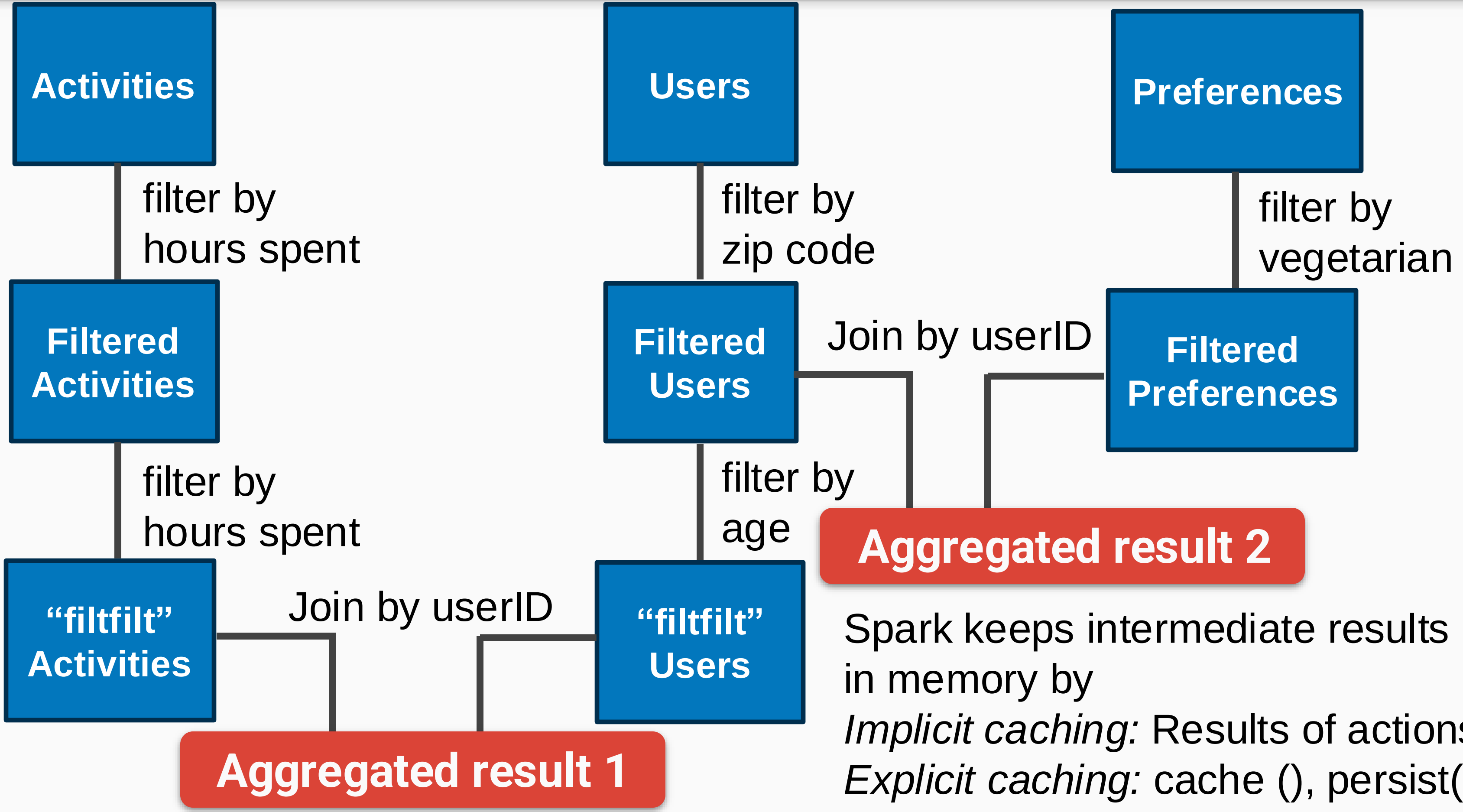


Contrast with **non-pipelined** implementation:

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

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

An example lineage graph of a multi-parent RDD pipeline



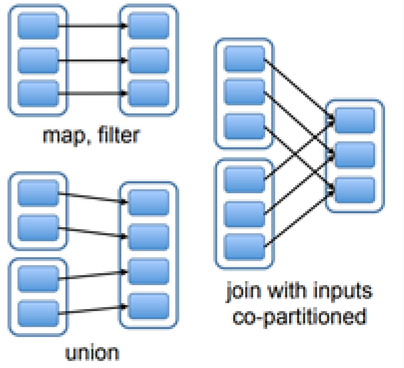
Spark keeps intermediate results in memory by
Implicit caching: Results of actions
Explicit caching: cache (), persist()

Partitions: 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



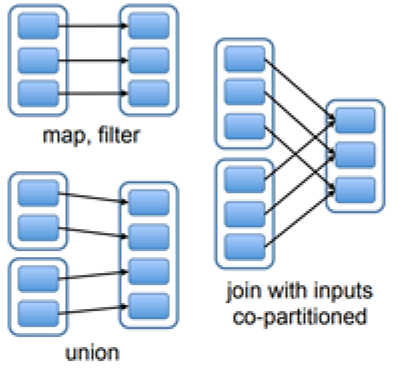
Figures adapted from [Zaharia et al., 2012]

Partitions: Narrow and wide dependencies

Narrow dependencies

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

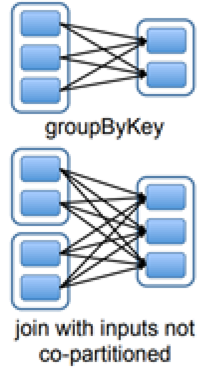
- Low communication
- Localized
- Easy to pipeline
- Easy failure recovery



Wide dependencies

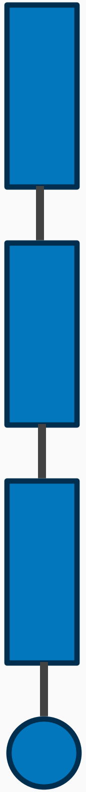
Partition of parent RDD goes to multiple child RDD partitions

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



Figures adapted from [Zaharia et al., 2012]

Example: RDDs and pipelines in Spark



initialRDD: Creates partitions by using, e.g. *parallelize* on the data source

filteredRDD: Filtering the initialRDD, e.g. by *filter* primes,
a narrow dependency transformation

squaredRDD: Squaring the filteredRDD, e.g. by *map,*
a narrow dependency transformation

totalSum: Aggregating the sum by applying *reduce* to the squaredRDD,
a wide dependency action

Example: RDDs & Partitions

initialRDD (Read Data Source) → Partition 1: [1:15] | Partition 2: [16:30]

Filter transformation (by isPrime)

filteredRDD (Narrow) Partition 1: [2, 3, 5, 7, 11, 13] | Partition 2: [17, 19, 23, 29]

Map transformation (square)

squaredRDD (Narrow) Partition 1: [4, 9, 25, 49, 121, 169] | Partition 2: [289, 361, 529, 841]

Aggregate / sum action (reduce)

totalSum (Wide): **2397**

Caution: Much like "bias" in machine learning, "partition" seems to be the favorite word in Big Data, with many different meanings

- In **Hadoop / CAP theorem**: Network partition, disconnected nodes in a network.
- In **Spark**: Data partition, a chunk of data
- For **disks**: Logical division of a hard drive into storage sections
- For **databases**: Distributing large databases into chunks across nodes (sharding, e.g. MongoDB)
- …

RDDs

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

Next week

- Applied Spark
- Column-oriented storage (Parquet)
- Dremel

Q&R