

Spark

Stony Brook University
CSE545, Fall 2017

Situations where MapReduce is not efficient

DFS → Map → LocalFS → Network → Reduce → DFS → Map → ...

Situations where MapReduce is not efficient

- Long pipelines sharing data
- Interactive applications
- Streaming applications
- Iterative algorithms (optimization problems)



(Anytime where MapReduce would need to write and read from disk a lot).

Situations where MapReduce is not efficient

- Long pipelines sharing data
- Interactive applications
- Streaming applications
- Iterative algorithms (optimization problems)



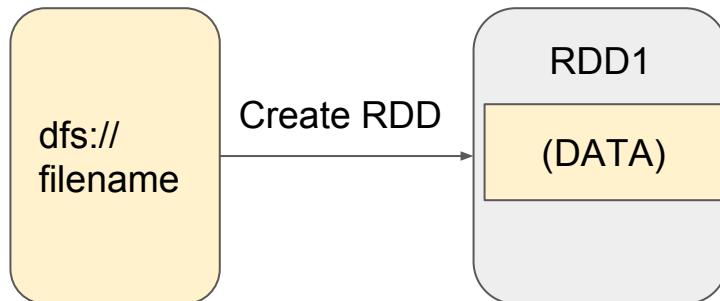
(Anytime where MapReduce would need to write and read from disk a lot).

Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).

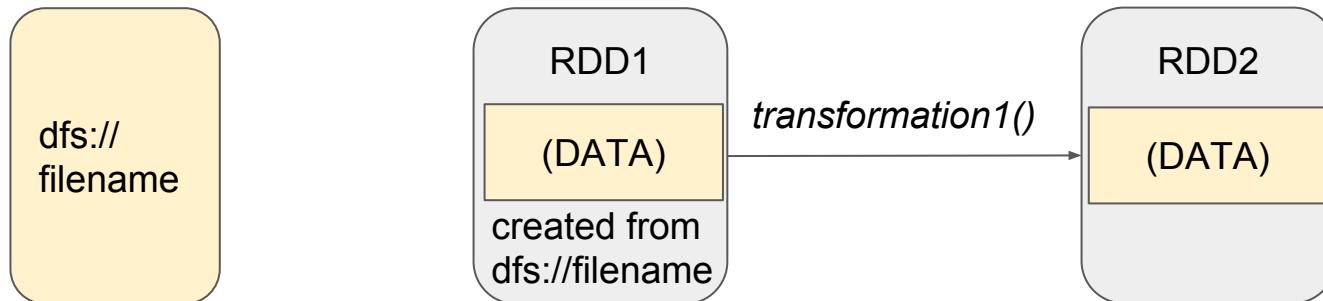
Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



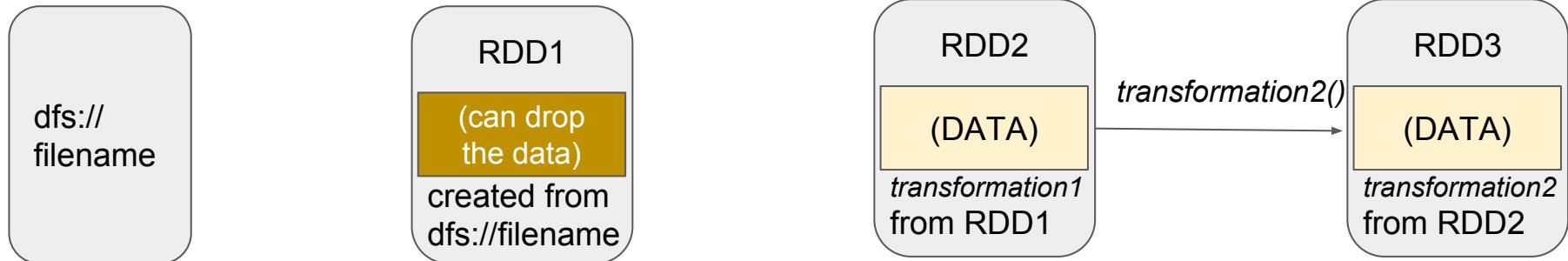
Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



Spark's Big Idea

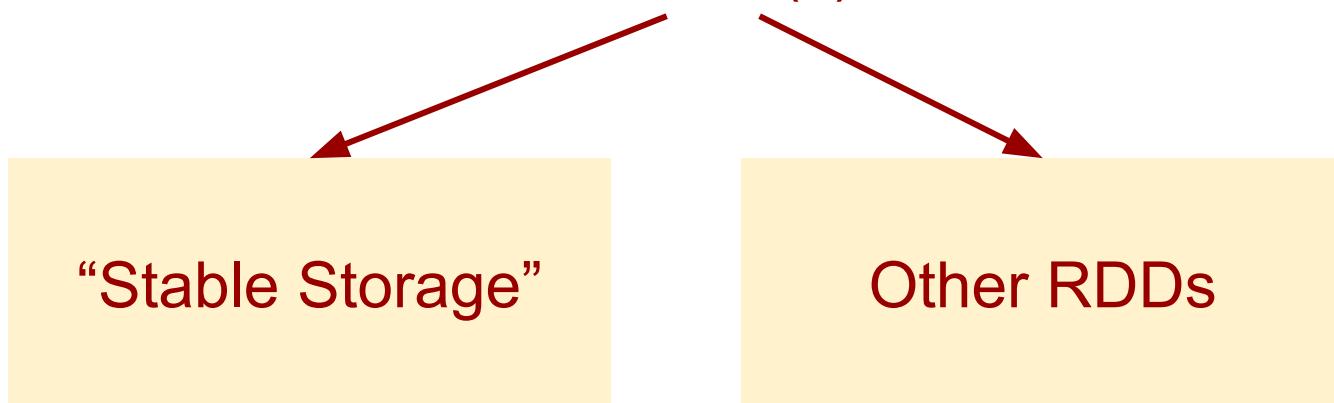
Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).

- Enables rebuilding datasets on the fly.
- Intermediate datasets not stored on disk
(and only in memory if needed and enough space)

➡ Faster communication and I/O

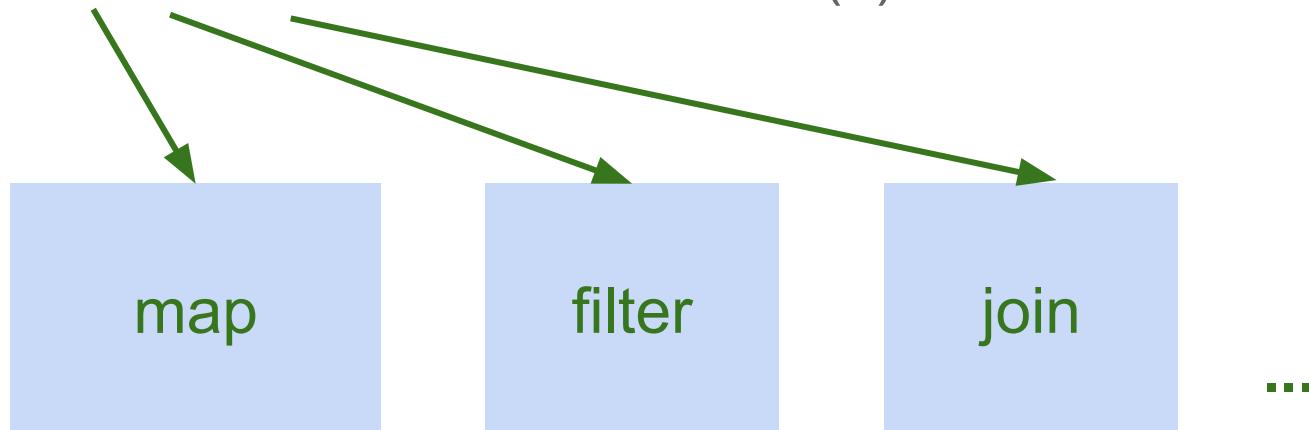
The Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



The Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



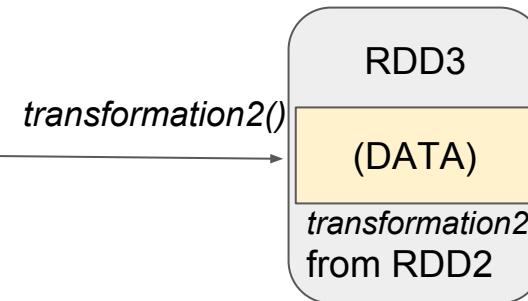
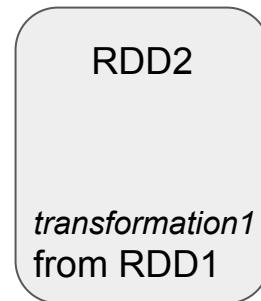
Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



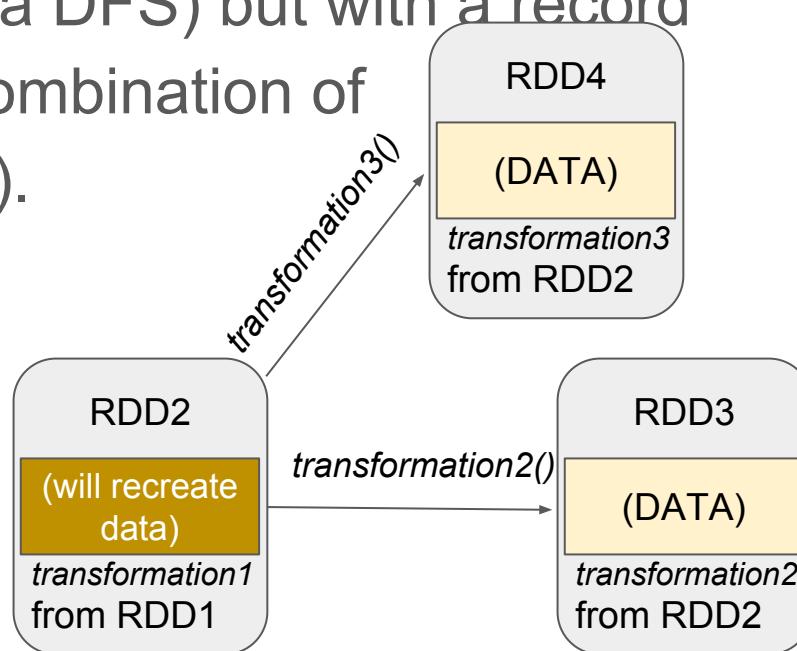
Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



Spark's Big Idea

Resilient Distributed Datasets (RDDs) -- Read-only partitioned collection of records (like a DFS) but with a record of how the dataset was created as combination of *transformations* from other dataset(s).



Original Transformations: RDD to RDD

Transformations

<i>map(f : T ⇒ U)</i>	: RDD[T] ⇒ RDD[U]
<i>filter(f : T ⇒ Bool)</i>	: RDD[T] ⇒ RDD[T]
<i>flatMap(f : T ⇒ Seq[U])</i>	: RDD[T] ⇒ RDD[U]
<i>sample(fraction : Float)</i>	: RDD[T] ⇒ RDD[T] (Deterministic sampling)
<i>groupByKey()</i>	: RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
<i>reduceByKey(f : (V, V) ⇒ V)</i>	: RDD[(K, V)] ⇒ RDD[(K, V)]
<i>union()</i>	: (RDD[T], RDD[T]) ⇒ RDD[T]
<i>join()</i>	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
<i>cogroup()</i>	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
<i>crossProduct()</i>	: (RDD[T], RDD[U]) ⇒ RDD[(T, U)]
<i>mapValues(f : V ⇒ W)</i>	: RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
<i>sort(c : Comparator[K])</i>	: RDD[(K, V)] ⇒ RDD[(K, V)]
<i>partitionBy(p : Partitioner[K])</i>	: RDD[(K, V)] ⇒ RDD[(K, V)]

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Original Transformations: RDD to RDD

Transformations

$map(f : T \Rightarrow U)$: $RDD[T] \Rightarrow RDD[U]$
$filter(f : T \Rightarrow Bool)$: $RDD[T] \Rightarrow RDD[T]$
$flatMap(f : T \Rightarrow Seq[U])$: $RDD[T] \Rightarrow RDD[U]$
$sample(fraction : Float)$: $RDD[T] \Rightarrow RDD[T]$ (Deterministic sampling)
$groupByKey()$: $RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$
$reduceByKey(f : (V, V) \Rightarrow V)$: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$
$union()$: $(RDD[T], RDD[T]) \Rightarrow RDD[T]$
$join()$: $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$
$cogroup()$: $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]$
$crossProduct()$: $(RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$
$mapValues(f : V \Rightarrow W)$: $RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (Preserves partitioning)
$sort(c : Comparator[K])$: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$
$partitionBy(p : Partitioner[K])$: $RDD[(K, V)] \Rightarrow RDD[(K, V)]$

Original Actions: RDD to Value, Object, or Storage

Actions

$count()$: $RDD[T] \Rightarrow Long$
$collect()$: $RDD[T] \Rightarrow Seq[T]$
$reduce(f : (T, T) \Rightarrow T)$: $RDD[T] \Rightarrow T$
$lookup(k : K)$: $RDD[(K, V)] \Rightarrow Seq[V]$ (On hash/range partitioned RDDs)
$save(path : String)$: Outputs RDD to a storage system, e.g., HDFS

Current Transformations and Actions

<http://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations>

common transformations: *filter, map, flatMap, reduceByKey, groupByKey*

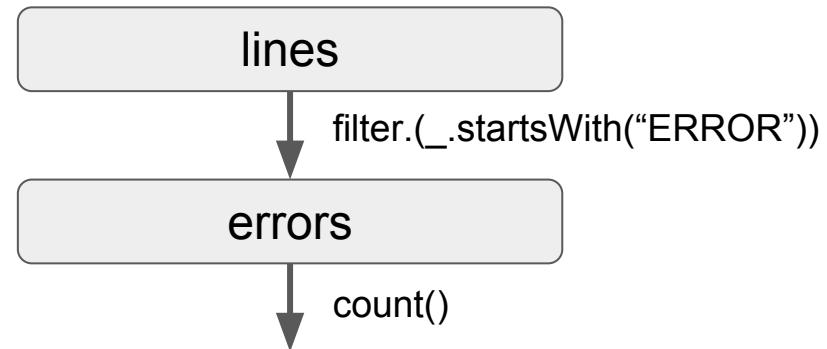
<http://spark.apache.org/docs/latest/rdd-programming-guide.html#actions>

common actions: *collect, count, take*

An Example

Count errors in a log file:

TYPE MESSAGE TIME



Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.”. NSDI 2012. April 2012.

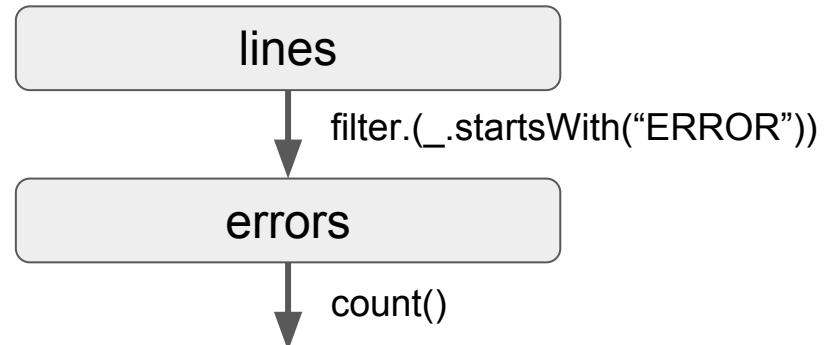
An Example

Count errors in a log file:

TYPE MESSAGE TIME

Pseudocode:

```
lines = sc.textFile("dfs:...")  
errors =  
    lines.filter(_.startswith("ERROR"))  
errors.count
```



Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.". NSDI 2012. April 2012.

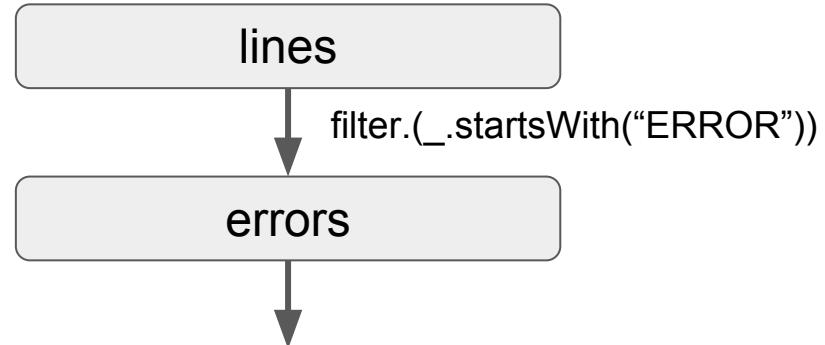
An Example

Collect times of hdfs-related errors

TYPE	MESSAGE	TIME
------	---------	------

```
: Pseudocode:
```

```
: lines = sc.textFile("dfs:...")  
: errors =  
:   lines.filter(_.startsWith("ERROR"))  
: errors.persist  
: errors.count  
:  
...
```



Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.". NSDI 2012. April 2012.

An Example

Collect times of hdfs-related errors

TYPE MESSAGE TIME

Pseudocode:

```
lines = sc.textFile("dfs://...")  
errors =  
    lines.filter(_.startswith("ERROR"))  
errors.persist  
errors.count  
...
```

Persistence

Can specify that an RDD “persists” in memory so other queries can use it.

Can specify a priority for persistence; lower priority => moves to disk, if needed, earlier

An Example

Collect times of hdfs-related errors

TYPE MESSAGE TIME

Pseudocode:

```
lines = sc.textFile("dfs://...")  
errors =  
    lines.filter(_.startswith("ERROR"))  
errors.persist  
errors.count  
...
```

Persistence

Can specify that an RDD “persists” in memory so other queries can use it.

Can specify a priority for persistence; lower priority => moves to disk, if needed, earlier

[parameters for persist](#)

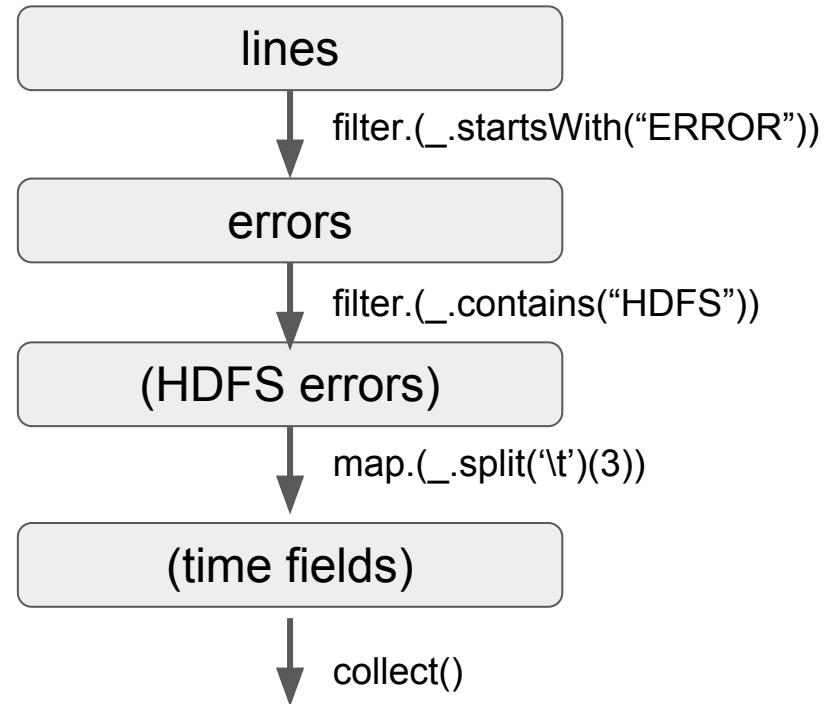
An Example

Collect times of hdfs-related errors

TYPE MESSAGE TIME

Pseudocode:

```
lines = sc.textFile("dfs:...")  
errors =  
    lines.filter(_.startswith("ERROR"))  
errors.persist  
errors.count  
errors.filter(_.contains("HDFS"))  
    .map(_.split('\t')(3))  
    .collect()
```



Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.". NSDI 2012. April 2012.

An Example

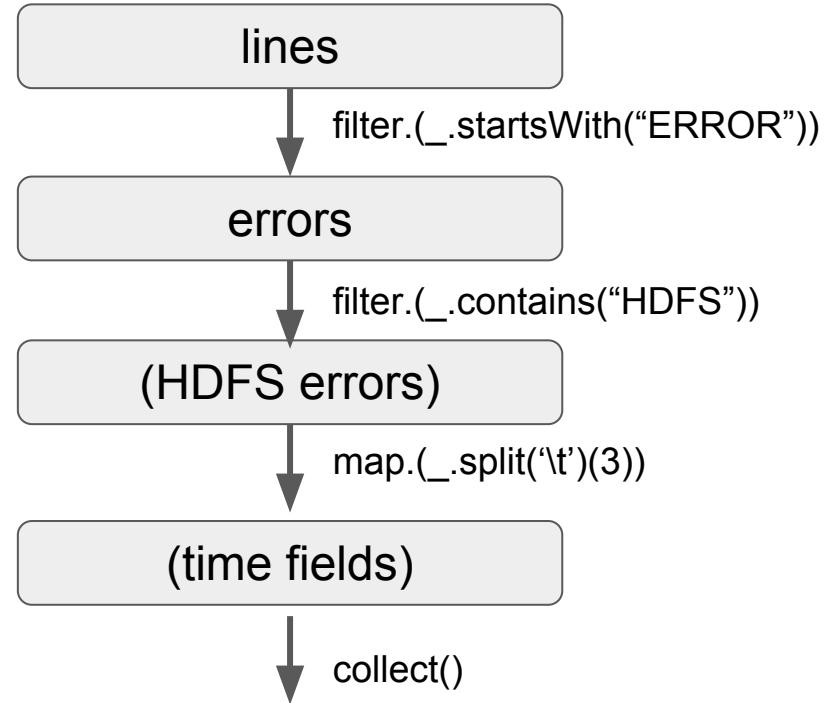
Collect times of hdfs-related errors

TYPE MESSAGE TIME

Pseudocode:

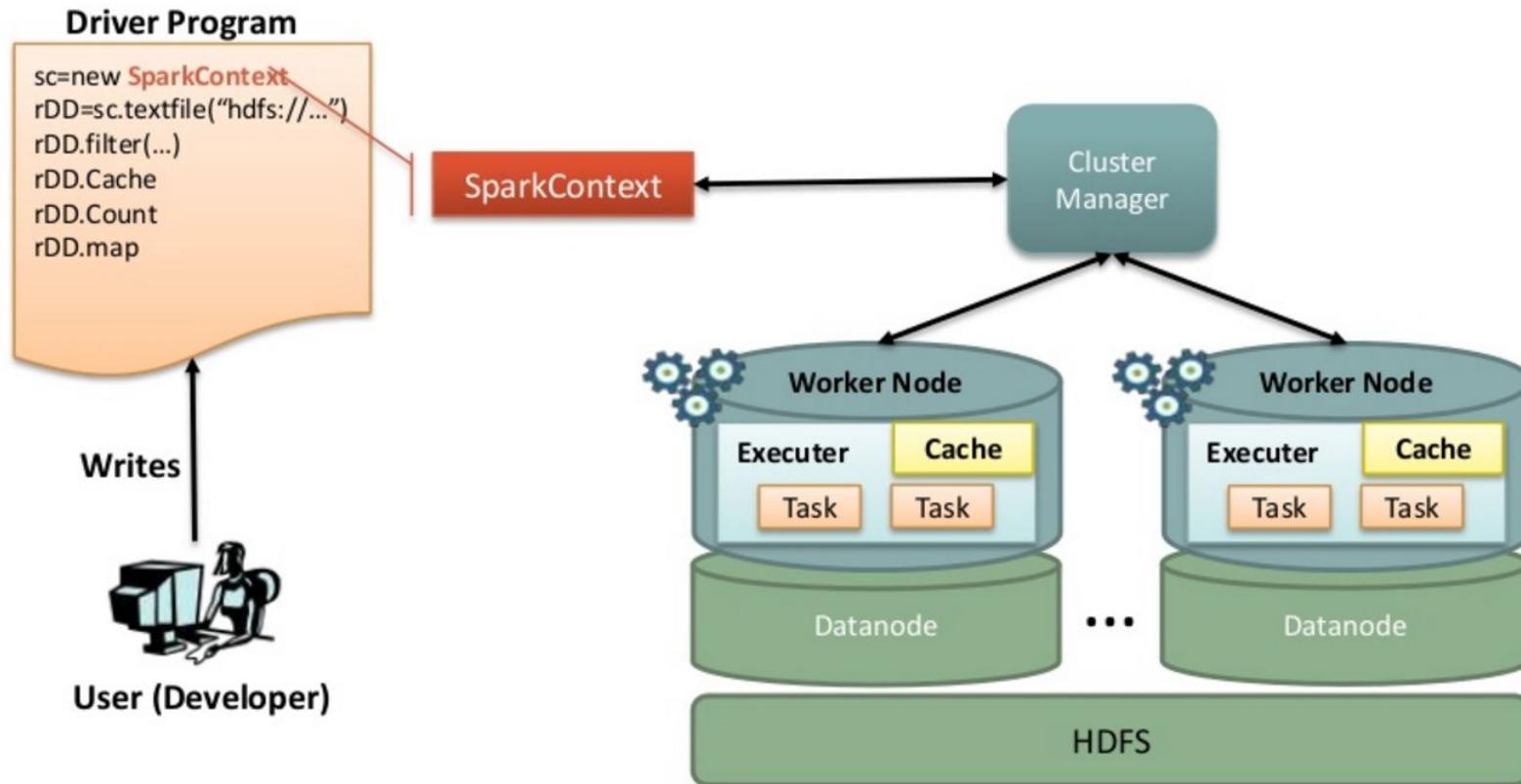
```
lines = sc.textFile("dfs:...")  
errors =  
  lines.filter(_.startswith("ERROR"))  
errors.persist  
errors.count  
errors.filter(_.contains("HDFS"))  
  .map(_split('\t')(3))  
  .collect()
```

Functional Programming



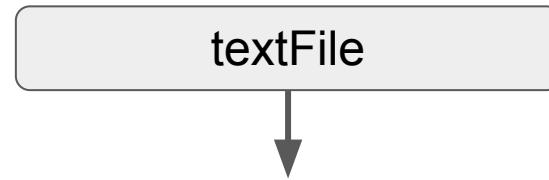
Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.". NSDI 2012. April 2012.

The Spark Programming Model



An Example

Word Count

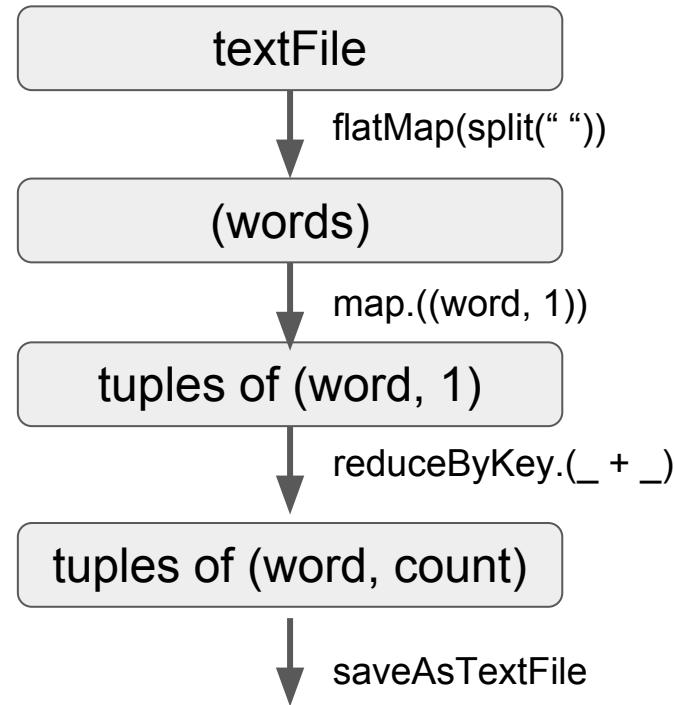


An Example

Word Count

Scala:

```
val textFile =  
  sc.textFile("hdfs://...")  
val counts = textFile  
  .flatMap(line => line.split(" "))  
  .map(word => (word, 1))  
  .reduceByKey(_ + _)  
counts.saveAsTextFile("hdfs://...")
```

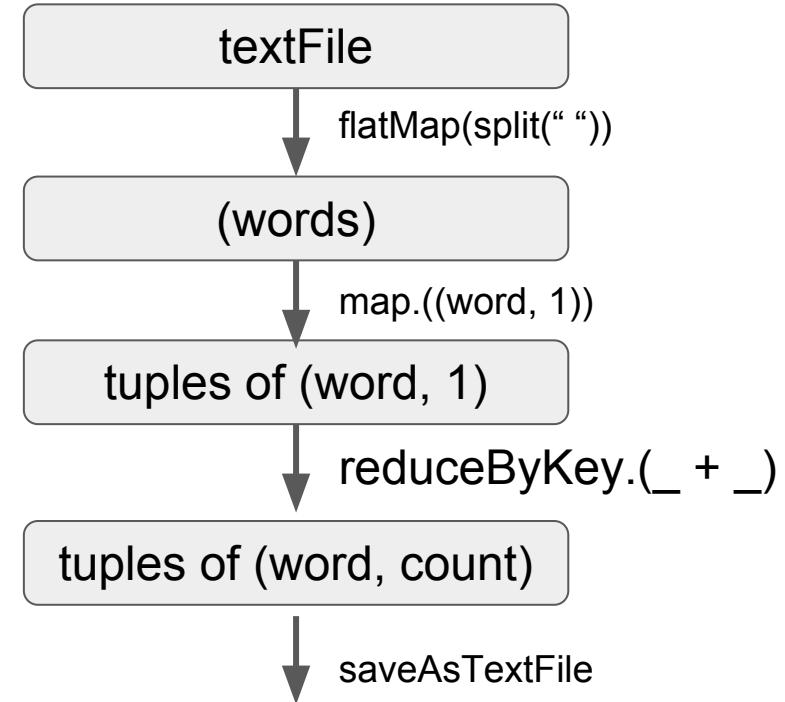


An Example

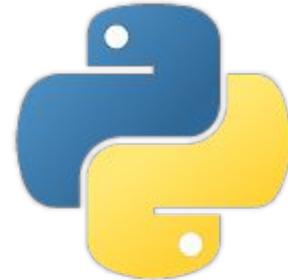
Word Count

Python:

```
textFile = sc.textFile("hdfs://...")  
counts = textFile  
    .flatMap(lambda line: line.split(" "))  
    .map(lambda word: (word, 1))  
    .reduceByKey(lambda a, b: a + b)  
counts.saveAsTextFile("hdfs://...")
```



PySpark Demo



<https://data.worldbank.org/data-catalog/poverty-and-equity-database>

Lazy Evaluation

Spark waits to **load data** and **execute transformations** until necessary -- *lazy*
Spark tries to complete **actions** as immediately as possible -- **eager**

Why?

- Only executes what is necessary to achieve action.
- Can optimize the complete *chain of operations* to reduce communication

Lazy Evaluation

Spark waits to *load data* and *execute transformations* until necessary -- **lazy**
Spark tries to complete actions as quickly as possible -- **eager**

Why?

- Only executes what is necessary to achieve action.
 - Can optimize the complete *chain of operations* to reduce communication

e.g.

```
rdd.map(lambda r: r[1]*r[3]).take(5) #only executes map for five records
```

```
rdd.filter(lambda r: "ERROR" in r[0]).map(lambda r: r[1]*r[3])  
#only passes through the data once
```

Broadcast Variables

Read-only objects can be shared across all nodes.

Broadcast variable is a wrapper: access object with .value

```
: Python:
```

```
: filterWords = ['one', 'two', 'three', 'four', ...]
: fwBC = sc.broadcast(set(filterWords))

: textFile = sc.textFile("hdfs://...")
: counts = textFile
:     .map(lambda line: line.split(" "))
:     .filter(lambda words: len(set(words)) & fwBC.value) > 0)
:     .flatMap(lambda word: (word, 1))
:     .reduceByKey(lambda a, b: a + b)
: counts.saveAsTextFile("hdfs://...")
```

Spark Overview

- RDD provides full recovery by backing up transformations from stable storage rather than backing up the data itself.
- RDDs, which are immutable, can be stored in memory and thus are often much faster.
- Functional programming is used to define transformation and actions on RDDs.
- Still need Hadoop (or some DFS) to hold original or resulting data efficiently and reliably.
- Lazy evaluation enables optimizing chain of operations.
- Memory across Spark cluster should be large enough to hold entire dataset to fully leverage speed.
 - MapReduce may still be more cost-effective for very large data that does not fit in memory.