Situations where MapReduce is not efficient

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

- Long pipelines sharing data
- Interactive applications
- Streaming applications
- Iterative algorithms (optimization problems)
Situations where MapReduce is not efficient

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

DFS ➔ Map ➔ LocalFS ➔ Network ➔ Reduce ➔ DFS ➔ Map ➔ ...

(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)

DFS ➔ Map ➔ LocalFS ➔ Network ➔ Reduce ➔ DFS ➔ Map ➔ ...

(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).

dfs://filename

RDD1
(can drop the data)
created from dfs://filename

RDD2
(DATA)
transformation1 from RDD1

RDD3
(DATA)
transformation2 from RDD2
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).

“Stable Storage”

Other RDDs
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).

- RDD1
  - created from dfs://filename

- RDD2
  - (DATA)
  - transformation1 from RDD1

- RDD3
  - (DATA)
  - transformation2 from RDD2

- RDD4
  - (DATA)
  - transformation3 from RDD2

  transformation2()

  transformation3()
### Original Transformations: RDD to RDD

<table>
<thead>
<tr>
<th>Transformations</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T → U)</code></td>
<td>$\text{RDD}[T] \Rightarrow \text{RDD}[U]$</td>
</tr>
<tr>
<td><code>filter(f : T → \text{Bool})</code></td>
<td>$\text{RDD}[T] \Rightarrow \text{RDD}[T]$</td>
</tr>
<tr>
<td><code>flatMap(f : T → \text{Seq}[U])</code></td>
<td>$\text{RDD}[T] \Rightarrow \text{RDD}[U]$</td>
</tr>
<tr>
<td><code>sample(fraction : \text{Float})</code></td>
<td>$\text{RDD}[T] \Rightarrow \text{RDD}[T]$ (Deterministic sampling)</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td>$\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, \text{Seq}[V])]$</td>
</tr>
<tr>
<td><code>reduceByKey(f : (V, V) → V)</code></td>
<td>$\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$</td>
</tr>
<tr>
<td><code>union()</code></td>
<td>$\text{RDD}[[T], \text{RDD}[[T]] \Rightarrow \text{RDD}[T]$</td>
</tr>
<tr>
<td><code>join()</code></td>
<td>$\text{RDD}[[K, V], \text{RDD}[[K, W]]] \Rightarrow \text{RDD}[(K, (V, W))]]$</td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td>$\text{RDD}[[K, V], \text{RDD}[[K, W]]] \Rightarrow \text{RDD}[(K, (\text{Seq}[V], \text{Seq}[W]))]$</td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td>$\text{RDD}[[T], \text{RDD}[[U]] \Rightarrow \text{RDD}[(T, U)]$</td>
</tr>
<tr>
<td><code>mapValues(f : V → W)</code></td>
<td>$\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, W)]$ (Preserves partitioning)</td>
</tr>
<tr>
<td><code>sort(c : \text{Comparator}[K])</code></td>
<td>$\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$</td>
</tr>
<tr>
<td><code>partitionBy(p : \text{Partitioner}[K])</code></td>
<td>$\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Transformations</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>map($f : T \rightarrow U$)</td>
<td>$\text{RDD}[T] \rightarrow \text{RDD}[U]$</td>
</tr>
<tr>
<td>filter($f : T \rightarrow \text{Bool}$)</td>
<td>$\text{RDD}[T] \rightarrow \text{RDD}[T]$</td>
</tr>
<tr>
<td>flatMap($f : T \rightarrow \text{Seq}[U]$)</td>
<td>$\text{RDD}[T] \rightarrow \text{RDD}[U]$</td>
</tr>
<tr>
<td>sample($\text{fraction} : \text{Float}$)</td>
<td>$\text{RDD}[T] \rightarrow \text{RDD}[T]$ (Deterministic sampling)</td>
</tr>
<tr>
<td>groupByKey()</td>
<td>$\text{RDD}[(K, V)] \rightarrow \text{RDD}[(K, \text{Seq}[V])]$</td>
</tr>
<tr>
<td>reduceByKey($f : (V, V) \rightarrow V$)</td>
<td>$\text{RDD}[(K, V)] \rightarrow \text{RDD}[(K, V)]$</td>
</tr>
<tr>
<td>union()</td>
<td>$(\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \rightarrow \text{RDD}[(K, (V, W))]$</td>
</tr>
<tr>
<td>join()</td>
<td>$(\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \rightarrow \text{RDD}[(K, (V, W))]$</td>
</tr>
<tr>
<td>cogroup()</td>
<td>$(\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \rightarrow \text{RDD}[(K, (\text{Seq}[V], \text{Seq}[W]))]$</td>
</tr>
<tr>
<td>crossProduct()</td>
<td>$(\text{RDD}[T], \text{RDD}[U]) \rightarrow \text{RDD}[(T, U)]$</td>
</tr>
<tr>
<td>mapValues($f : V \rightarrow W$)</td>
<td>$\text{RDD}[(K, V)] \rightarrow \text{RDD}[(K, W)]$ (Preserves partitioning)</td>
</tr>
<tr>
<td>sort($c : \text{Comparator}[K]$)</td>
<td>$\text{RDD}[(K, V)] \rightarrow \text{RDD}[(K, V)]$</td>
</tr>
<tr>
<td>partitionBy($p : \text{Partitioner}[K]$)</td>
<td>$\text{RDD}[(K, V)] \rightarrow \text{RDD}[(K, V)]$</td>
</tr>
</tbody>
</table>

Original Actions: RDD to Value, Object, or Storage

<table>
<thead>
<tr>
<th>Actions</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>$\text{RDD}[T] \rightarrow \text{Long}$</td>
</tr>
<tr>
<td>collect()</td>
<td>$\text{RDD}[T] \rightarrow \text{Seq}[T]$</td>
</tr>
<tr>
<td>reduce($f : (T, T) \rightarrow T$)</td>
<td>$\text{RDD}[T] \rightarrow T$</td>
</tr>
<tr>
<td>lookup($k : K$)</td>
<td>$\text{RDD}[(K, V)] \rightarrow \text{Seq}[V]$ (On hash/range partitioned RDDs)</td>
</tr>
<tr>
<td>save(path : String)</td>
<td>Outputs RDD to a storage system, e.g., HDFS</td>
</tr>
</tbody>
</table>

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:

```
<table>
<thead>
<tr>
<th>TYPE</th>
<th>MESSAGE</th>
<th>TIME</th>
</tr>
</thead>
</table>
```

```
lines.filter(_.startsWith("ERROR"))
count()
```

An Example

Count errors in a log file:

<table>
<thead>
<tr>
<th>TYPE</th>
<th>MESSAGE</th>
<th>TIME</th>
</tr>
</thead>
</table>

Pseudocode:

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

An Example

Collect times of hdfs-related errors

<table>
<thead>
<tr>
<th>TYPE</th>
<th>MESSAGE</th>
<th>TIME</th>
</tr>
</thead>
</table>

Pseudocode:

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

An Example

Collect times of hdfs-related errors

<table>
<thead>
<tr>
<th>TYPE</th>
<th>MESSAGE</th>
<th>TIME</th>
</tr>
</thead>
</table>

Pseudocode:

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

Persistance

Can specify that an RDD “persists” in memory so other queries can use it.
Can specify a priority for persistance; lower priority => moves to disk, if needed, earlier

An Example

Collect times of hdfs-related errors

<table>
<thead>
<tr>
<th>TYPE</th>
<th>MESSAGE</th>
<th>TIME</th>
</tr>
</thead>
</table>

Pseudocode:

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

Persistance

Can specify that an RDD “persists” in memory so other queries can use it.
Can specify a priority for persistance; lower priority => moves to disk, if needed, earlier

An Example

Collect times of hdfs-related errors

**TYPE  MESSAGE  TIME**

Pseudocode:

\[
\text{lines} = \text{sc.textFile("dfs:")}
\]

\[
\text{errors} =
\quad \text{lines.filter(_.startsWith("ERROR"))}
\quad \text{errors.persist}
\quad \text{errors.count}
\quad \text{errors.filter(_.contains("HDFS"))}
\quad \quad .\text{map(_.split(\'\t\')(3))}
\quad \quad .\text{collect()}
\]

An Example

Collect times of hdfs-related errors

**TYPE**  **MESSAGE**  **TIME**

Pseudocode:

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

An Example

Word Count

textFile
An Example

Word Count

Scala:

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

Apache Spark Examples
http://spark.apache.org/examples.html
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

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:

```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.