Spark
Stony Brook University
CSE545, Spring 2019
Situations where MapReduce is not efficient

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

DFS $\rightarrow$ Map $\rightarrow$ LocalFS $\rightarrow$ Network $\rightarrow$ Reduce $\rightarrow$ DFS $\rightarrow$ Map $\rightarrow$ ...
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).
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).

map  filter  join  ...

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).
### Original Transformations: RDD to RDD

<table>
<thead>
<tr>
<th>Transformations</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T ⇒ U)</code></td>
<td><code>RDD[T] ⇒ RDD[U]</code></td>
</tr>
<tr>
<td><code>filter(f : T ⇒ Bool)</code></td>
<td><code>RDD[T] ⇒ RDD[T]</code></td>
</tr>
<tr>
<td><code>flatMap(f : T ⇒ Seq[U])</code></td>
<td><code>RDD[T] ⇒ RDD[U]</code></td>
</tr>
<tr>
<td><code>sample(fraction : Float)</code></td>
<td><code>RDD[T] ⇒ RDD[T]</code> (Deterministic sampling)</td>
</tr>
<tr>
<td><code>sortByKey()</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, Seq[V])]</code></td>
</tr>
<tr>
<td><code>reduceByKey(f : (V, V) ⇒ V)</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
</tr>
<tr>
<td><code>union()</code></td>
<td><code>(RDD[T], RDD[T]) ⇒ RDD[T]</code></td>
</tr>
<tr>
<td><code>join()</code></td>
<td><code>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]</code></td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td><code>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]</code></td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td><code>(RDD[T], RDD[U]) ⇒ RDD[(T, U)]</code></td>
</tr>
<tr>
<td><code>mapValues(f : V ⇒ W)</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, W)]</code> (Preserves partitioning)</td>
</tr>
<tr>
<td><code>sort(c : Comparator[K])</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
</tr>
<tr>
<td><code>partitionBy(p : Partitioner[K])</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T ⇒ U)</code></td>
<td>RDD[T] ⇒ RDD[U]</td>
</tr>
<tr>
<td><code>filter(f : T ⇒ Bool)</code></td>
<td>RDD[T] ⇒ RDD[T]</td>
</tr>
<tr>
<td><code>flatMap(f : T ⇒ Seq[U])</code></td>
<td>RDD[T] ⇒ RDD[U]</td>
</tr>
<tr>
<td><code>sample(fraction : Float)</code></td>
<td>RDD[T] ⇒ RDD[T] (Deterministic sampling)</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, Seq[V])]</td>
</tr>
<tr>
<td><code>reduceByKey(f : (V, V) ⇒ V)</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
</tr>
<tr>
<td><code>union()</code></td>
<td>(RDD[T], RDD[T]) ⇒ RDD[T]</td>
</tr>
<tr>
<td><code>join()</code></td>
<td>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]</td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]</td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td>(RDD[T], RDD[U]) ⇒ RDD[(T, U)]</td>
</tr>
<tr>
<td><code>mapValues(f : V ⇒ W)</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)</td>
</tr>
<tr>
<td><code>sort(c : Comparator[K])</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
</tr>
<tr>
<td><code>partitionBy(p : Partitioner[K])</code></td>
<td>RDD[(K, V)] ⇒ 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>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T ⇒ U)</code></td>
<td>RDD[T] ⇒ RDD[U]</td>
</tr>
<tr>
<td><code>filter(f : T ⇒ Bool)</code></td>
<td>RDD[T] ⇒ RDD[T]</td>
</tr>
<tr>
<td><code>flatMap(f : T ⇒ Seq[U])</code></td>
<td>RDD[T] ⇒ RDD[U]</td>
</tr>
<tr>
<td><code>sample(fraction : Float)</code></td>
<td>RDD[T] ⇒ RDD[T] (Deterministic sampling)</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, Seq[V])]</td>
</tr>
<tr>
<td><code>reduceByKey(f : (V, V) ⇒ V)</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
</tr>
<tr>
<td><code>union()</code></td>
<td>(RDD[T], RDD[T]) ⇒ RDD[T]</td>
</tr>
<tr>
<td><code>join()</code></td>
<td>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]</td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td>(RDD[(K, V), RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W])])</td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td>(RDD[T], RDD[U]) ⇒ RDD[(T, U)]</td>
</tr>
<tr>
<td><code>mapValues(f : V ⇒ W)</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)</td>
</tr>
<tr>
<td><code>sort(c : Comparator[K])</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
</tr>
<tr>
<td><code>partitionBy(p : Partitioner[K])</code></td>
<td>RDD[(K, V)] ⇒ 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>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T ⇒ U)</code></td>
<td><code>RDD[T] ⇒ RDD[U]</code></td>
</tr>
<tr>
<td><code>filter(f : T ⇒ Bool)</code></td>
<td><code>RDD[T] ⇒ RDD[T]</code></td>
</tr>
<tr>
<td><code>flatMap(f : T ⇒ Seq[U])</code></td>
<td><code>RDD[T] ⇒ RDD[U]</code></td>
</tr>
<tr>
<td><code>sample(fraction : Float)</code></td>
<td><code>RDD[T] ⇒ RDD[T]</code> (Deterministic sampling)</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, Seq[V])]</code></td>
</tr>
<tr>
<td><code>reduceByKey(f : (V, V) ⇒ V)</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
</tr>
<tr>
<td><code>union()</code></td>
<td><code>(RDD[T], RDD[T]) ⇒ RDD[T]</code></td>
</tr>
<tr>
<td><code>join()</code></td>
<td><code>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]</code></td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td><code>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]</code></td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td><code>(RDD[T], RDD[U]) ⇒ RDD[(T, U)]</code></td>
</tr>
<tr>
<td><code>mapValues(f : V ⇒ W)</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, W)]</code> (Preserves partitioning)</td>
</tr>
<tr>
<td><code>sort(c : Comparator[K])</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
</tr>
<tr>
<td><code>partitionBy(p : Partitioner[K])</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
</tr>
</tbody>
</table>

### Original Actions: RDD to Value, Object, or Storage

<table>
<thead>
<tr>
<th>Actions</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td><code>RDD[T] ⇒ Long</code></td>
</tr>
<tr>
<td><code>collect()</code></td>
<td><code>RDD[T] ⇒ Seq[T]</code></td>
</tr>
<tr>
<td><code>reduce(f : (T, T) ⇒ T)</code></td>
<td><code>RDD[T] ⇒ T</code></td>
</tr>
<tr>
<td><code>lookup(k : K)</code></td>
<td><code>RDD[(K, V)] ⇒ Seq[V]</code> (On hash/range partitioned RDDS)</td>
</tr>
<tr>
<td><code>save(path : String)</code></td>
<td>Outputs RDD to a storage system, e.g., HDFS</td>
</tr>
</tbody>
</table>

Current Transformations and Actions

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

common actions: *collect*, *count*, *take*
An Example

Count errors in a log file:

TYPE  MESSAGE  TIME

```
lines
filter(_.startsWith("ERROR"))
errors
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
type = sc.textFile("dfs:...")
errors = type.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 persistence; 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:

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

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

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()

The Spark Programming Model

An Example

Word Count

textFile
Scalable:

```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.*

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

```python
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))
```
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) and word in fwBC.value) > 0)
    .flatMap(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs:...")
```
Accumulators

Write-only objects that keep a running aggregation

Default Accumulator assumes sum function

```python
initialValue = 0
sumAcc = sc.accumulator(initialValue)
rdd.foreach(lambda i: sumAcc.add(i))
print(sumAcc.value)
```
Accumulators

Write-only objects that keep a running aggregation

**Default Accumulator assumes sum function**

**Custom Accumulator:** Inherit (AccumulatorParam) as class and override methods

```python
initialValue = 0
sumAcc = sc.accumulator(initialValue)
rdd.foreach(lambda i: sumAcc.add(i))
print(minAcc.value)

class MinAccum(AccumulatorParam):
    def zero(self, zeroValue = np.inf):#overwrite this
        return zeroValue
    def addInPlace(self, v1, v2):#overwrite this
        return min(v1, v2)
minAcc = sc.accumulator(np.inf, MinAccum())
rdd.foreach(lambda i: minAcc.add(i))
print(minAcc.value)
```
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.
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.