Acknowledgements

- Part of the course material is based on slides provided by the following authors
  - P. Michiardi, M. Zaharia, A. Davidson
Introduction - Motivation

- MapReduce greatly simplified “big data” analysis on large, unreliable clusters
- But as soon as it got popular, users wanted more:
  - More complex, multi-stage applications
    (e.g. iterative machine learning & graph processing)
  - More interactive ad-hoc queries

- Response ➔ specialized frameworks for some of these apps
  - E.g. Pregel for graph processing

Motivation - Data point of view

- Complex apps and interactive queries both need one thing that MapReduce lacks:
  - Efficient primitives for data sharing

- In MapReduce, the only way to share data across jobs is stable storage
  - Slow!
Goal

- From the Hadoop approach
  - Slow due to replication and disk I/O, but necessary for fault tolerance

- To an “in-memory” approach

Motivation - System point of view

- Hadoop code base is huge
- Contributions/Extensions to Hadoop are cumbersome
- System/Framework: no unified pipeline
  - Sparse modules
  - Diversity of APIs
  - Higher operational costs
Goals

- Unified pipeline

Summary of the challenges

- **Data**
  - How to design a distributed memory abstraction that is both fault-tolerant and efficient?

- **System**
  - Is it possible to build a unified system that includes library for the most common problems?
Resilient Distributed Datasets (RDD)

What is an RDD

- RDD are partitioned, locality aware, distributed collections
  - RDD are immutable

- RDD are data structures that:
  - Either point to a direct data source (e.g. HDFS)
  - Apply some transformations to its parent RDD(s) to generate new data elements

- Computations on RDDs
  - Represented by lazily evaluated lineage DAGs composed by chained RDDs
RDD Abstraction

- Overall objective
  - Support a wide array of operators (more than just Map and Reduce)
  - Allow arbitrary composition of such operators
- Simplify scheduling
  - Avoid to modify the scheduler for each operator

- The question is: How to capture dependencies in a general way?

RDD Interfaces

- Set of partitions ("splits")
  - Much like in Hadoop MapReduce, each RDD is associated to (input) partitions
- List of dependencies on parent RDDs
  - This is completely new w.r.t. Hadoop MapReduce
- Function to compute a partition given parents
  - This is actually the "user-defined code" we referred to when discussing about the Mapper and Reducer classes in Hadoop
- Optional preferred locations
  - This is to enforce data locality
- Optional partitioning info (Partitioner)
  - This really helps in some "advanced" scenarios in which you want to pay attention to the behavior of the shuffle mechanism
### Examples of RDD

<table>
<thead>
<tr>
<th></th>
<th>Hadoop RDD</th>
<th>Filtered RDD</th>
<th>Joined RDD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Partitions</strong></td>
<td>One per HDFS block</td>
<td>Same as parent RDD</td>
<td>One per reduce task</td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>None</td>
<td>One-to-one on parent</td>
<td>Shuffle on each parent</td>
</tr>
<tr>
<td><strong>Compute (partition)</strong></td>
<td>Read corresponding block</td>
<td>Compute parent and filter it</td>
<td>Read and join shuffled data</td>
</tr>
<tr>
<td><strong>Preferred location</strong></td>
<td>HDFS block location</td>
<td>None (ask parent)</td>
<td>None</td>
</tr>
<tr>
<td><strong>Partitioner</strong></td>
<td>None</td>
<td>None</td>
<td>HashPartitioner (numTask)</td>
</tr>
</tbody>
</table>

#### Dependency types: narrow

- Each partition of the parent RDD is used by at most one partition of the child RDD
- Task can be executed locally and we don’t have to shuffle. (Eg: map, flatMap, filter, sample)
Dependency types: wide

- Multiple child partitions may depend on one partition of the parent RDD

- This means we have to shuffle data unless the parents are hash-partitioned
  - Eg: sortByKey, reduceByKey, groupByKey, cogroupByKey, join, cartesian

Dependency Types: Optimizations

- Benefits of Lazy evaluation:
  The DAG Scheduler optimizes Stages and Tasks before submitting them to the Task Scheduler

- Examples:
  - Pipelining narrow dependencies within a Stage
  - Join plan selection based on partitioning
  - Cache reuse
Operations on RDDs: Transformations

- **Transformations**
  - Set of operations on a RDD that define how they should be transformed
  - As in relational algebra, the application of a transformation to an RDD yields a new RDD (because RDDs are immutable)
  - Transformations are lazily evaluated, which allow for optimizations to take place before execution

- **Examples (not exhaustive)**
  - `map(func)`, `flatMap(func)`, `filter(func)`
  - `groupByKey()`
  - `reduceByKey(func)`, `mapValues(func)`, `distinct()`, `sortByKey(func)`
  - `join(other)`, `union(other)`
  - `sample()`

Operations on RDDs: Actions

- **Actions**
  - Apply transformation chains on RDDs, eventually performing some additional operations (e.g., counting)
  - Some actions only store data to an external data source (e.g. HDFS), others fetch data from the RDD (and its transformation chain) upon which the action is applied, and convey it to the driver

- **Examples (not exhaustive)**
  - `reduce(func)`
  - `collect()`, `first()`, `take()`, `foreach(func)`
  - `count()`, `countByKey()`
  - `saveAsTextFile()`
Examples

Word count

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

- “sc” is the SparkContext
  - A SparkContext initializes the application driver, the latter then registers the application to the cluster manager, and gets a list of executors

- Alternative version:

```scala
val counts = textFile.flatMap(_.split(" "))
  .map(_ => 1))
  .reduceByKey(_ + _)```
Common Transformations

```
RDD[String]
- Hello world
- Welcome to Spark
- Hello again

...)
.map(_.split("
"))
.flatMap(line => line.split(" "))
```

```
RDD[Array[String]]
- Hello
- world
- welcome
to
Spark
- Hello
- again

...)
```

```
RDD[String]
- Hello
- Hello
- welcome
to
Spark

...)
```

Common Transformations

```
RDD[String]
- Hello
- Hello
- Hello
- welcome
to
Spark

...)
.map(word => (word, 1))
```

```
RDD[(String, Int)]
- Hello 1
- Hello 1
- Hello 1
- welcome 1
to 1
Spark 1

...)
```
Log mining

- Load error messages from a log into memory, then interactively search for various patterns

```scala
lines = sc.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split("\t"))
messages.persist() // keep in memory

messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
```

Pagerank

- Pagerank defined as usual
  \[ P(n) = \alpha \left( \frac{1}{|G|} \right) + (1 - \alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)} \]
- Simple version, with no sink nodes

```scala
val links = ... // load RDD of (url, neighbors) pairs
var ranks = ... // load RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
    val contribs = links.join(ranks).flatMap {
        (url, (links, rank)) =>
            links.map(dest => (dest, rank/links.size))
    }
    ranks = contribs.reduceByKey(_ + _)
        .mapValues(0.15/G + 0.85 * _)
}
ranks.saveAsTextFile("hdfs://...")
```
PageRank Performance

![PageRank Performance Chart]

Other Iterative Algorithms

![Other Iterative Algorithms Chart]
Overview of the framework

A Very Simple Application Example

```
val sc = new SparkContext("spark://...", "MyJob", home, jars)
val file = sc.textFile("hdfs://...") // This is an RDD
val errors = file.filter(_.contains("ERROR")) // This is an RDD
errors.cache()
errors.count() // This is an action
```
Spark Applications: The Big Picture

- There are two ways to manipulate data in Spark
  - Use the interactive shell
  - Write standalone applications, i.e., driver programs

Spark Components: details
Data Locality

- Data locality principle
  - Same as for Hadoop MapReduce
  - Avoid network I/O, workers should manage local data

- Data locality and caching
  - First run: data not in cache, so use HadoopRDD’s locality prefs (from HDFS)
  - Second run: FilteredRDD is in cache, so use its locations
  - If something falls out of cache, go back to HDFS
Lifetime of a Job in Spark

In Summary

- **Our example Application: a jar file**
  - Creates a SparkContext, which is the core component of the driver
  - Creates an input RDD, from a file in HDFS
  - Manipulates the input RDD by applying a filter transformation
  - Invokes the action count() on the transformed RDD

- **The DAG Scheduler**
  - Gets: RDDs, functions to run on each partition and a listener for results
  - Builds Stages of Tasks objects (code + preferred location)
  - Submits Tasks to the Task Scheduler as ready
  - Resubmits failed Stages

- **The Task Scheduler**
  - Launches Tasks on executors
  - Relaunches failed Tasks
  - Reports to the DAG Scheduler