Spark and Resilient Distributed Datasets
Motivation

- MapReduce greatly simplified big data analysis on large, unreliable clusters.

- But as soon as it got popular, users wanted more:
  - Iterative jobs, e.g., machine learning algorithms
  - Interactive analytics
Both iterative and interactive queries need one thing that MapReduce lacks: **Efficient primitives for data sharing.**

- In MapReduce, the only way to share data across jobs is stable storage (disk).
- Replication also makes the system slow, but it is necessary for fault tolerance.
Solution

In memory data processing and sharing
Sharing
Challenge

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

Solution: Resilient Distributed Datasets (RDD)

- A distributed memory abstraction.
- Immutable collections of objects spread across a cluster.
- Lineage among RDDs to enable their re-evaluation in case of cluster node failures.
Resilient Distributed Datasets (RDDs)

- An RDD is divided into a number of partitions, which are atomic pieces of information.

- Partitions of an RDD can be stored on different nodes of a cluster.

Partitions of an RDD can be stored on different nodes of a cluster.
Programming model: Spark

- Based on parallelizable operators.

- Parallelizable operators are higher-order functions that execute user-defined functions in parallel.

- Higher-order functions are RDDs operators.

- There are two types of RDD operators: transformations and actions.
Programming model: Spark

- **Transformations**: lazy operators that create new RDDs.
- **Actions**: launch a computation and return a value to the program or write data to the external storage.

The main programming language is Scala:

- a strongly and statically typed functional-OO language
- compiled over the JVM
- designed at EPFL (Switzerland).
Example (1/2)

- Suppose that a web service is experiencing errors and an operator wants to search terabytes of logs in the Hadoop filesystem (HDFS) to find the cause.

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

- Actions can be used to count errors:

```scala
errors.count()
```

- Or counting errors mentioning MySQL:

```scala
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")) .count()
```
Example (1/2)

Suppose that a web service is experiencing errors and an operator wants to search terabytes of logs in the Hadoop filesystem (HDFS) to find the cause.

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

Actions can be used to count errors:

```scala
errors.count()
```

Or counting errors mentioning MySQL:

```scala
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()
```
Example (1/2)

- Suppose that a web service is experiencing errors and an operator wants to search terabytes of logs in the Hadoop filesystem (HDFS) to find the cause.

  ```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

- Actions can be used to count errors:

  ```scala```
  errors.count()
  ```scala```

- Or counting errors mentioning MySQL:

  ```scala```
  // Count errors mentioning MySQL:
  errors.filter(_.contains("MySQL")).count()
  ```scala```
Fault tolerance via lineage

// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning HDFS as an array (assuming time is field number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
  .map(_.split(' \t' )(3))
  .collect()

the lineage graph enables RDD re-evaluation in case of failure
### RDD transformations and actions

<table>
<thead>
<tr>
<th>Transformations</th>
<th>Result</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>
<thead>
<tr>
<th>Actions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td>RDD[T] ⇒ Long</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>RDD[T] ⇒ Seq[T]</td>
</tr>
<tr>
<td><code>reduce(f : (T, T) ⇒ T)</code></td>
<td>RDD[T] ⇒ T</td>
</tr>
<tr>
<td><code>lookup(k : K)</code></td>
<td>RDD[(K, V)] ⇒ Seq[V] (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>
RDD transformations : Map

- All pairs are independently processed

```scala
// passing each element through a function.
val nums = sc.parallelize(Array(1, 2, 3))
val squares = nums.map(x => x * x) // {1, 4, 9}

// selecting those elements that func returns true.
val even = squares.filter(x => x % 2 == 0) // {4}

// mapping each element to zero or more others.
nums.flatMap(x => Range(0, x, 1)) // {0, 0, 1, 0, 1, 2}
```
RDD transformations: Reduce

- Pairs with identical key are grouped
- Each group is independently processed

val pets = sc.parallelize(Seq(("cat", 1), ("dog", 1), ("cat", 2)))

pets.reduceByKey((x, y) => x + y)
// {(cat, 3), (dog, 1)}

pets.groupByKey()
// {(cat, (1, 2)), (dog, (1))}
RDD transformations: Join

- Equi-join on the key
- Join candidates are independently processed

```scala
val visits = sc.parallelize(Seq(
  ("index.html", "1.2.3.4"),
  ("about.html", "3.4.5.6"),
  ("index.html", "1.3.3.1"))

val pageNames = sc.parallelize(Seq(
  ("index.html", "Home"),
  ("about.html", "About")))

visits.join(pageNames)
  // ("index.html", ("1.2.3.4", "Home"))
  // ("index.html", ("1.3.3.1", "Home"))
  // ("about.html", ("3.4.5.6", "About"))
```
RDD transformations: CoGroup

- Groups each input on key
- Groups with identical keys are processed together

```scala
val visits = sc.parallelize(Seq(("index.html", "1.2.3.4"),
                               ("about.html", "3.4.5.6"),
                               ("index.html", "1.3.3.1")))

val pageNames = sc.parallelize(Seq(("index.html", "Home"),
                                    ("about.html", "About")))

visits.cogroup(pageNames)
// ("index.html", ("1.2.3.4", "1.3.3.1"), ("Home")))
// ("about.html", ("3.4.5.6"), ("About")))
```
PageRank

• The algorithm works on linked N documents (e.g. Web), and iteratively updates a rank for each document by adding up contributions from documents that link to it.

• On each iteration, each document sends a contribution \( c = r/n \) where \( r \) is the current rank of the document and \( n \) is the number of neighbours.

• The rank of each document is then updated according to

\[
\alpha/N + (1 - \alpha) \sum c_i
\]
PageRank

// Load graph as an RDD of (URL, outlinks) pairs
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
    .mapValues(sum => a/N + (1-a)*sum)
PageRank

Figure 9: Iteration times for logistic regression using 256 MB inputs.

Figure 10: Performance of PageRank on Hadoop and Spark.

Figure 11: Iteration times for k-means in presence of a failure.
Conclusion

• MapReduce makes important abstraction step that greatly helps rapid development of efficient and robust Big Data data flows.

• But:
  • we still need some ‘acking’ to ensure good performances
  • problems with iterative analyses
  • MapReduce programming is not easy

• Spark overcomes these limitations at the cost of more RAM needed, and makes a steps further towards ‘The data center is the computer’ scenario.
Merci!