Advanced Big Data Systems

Guido Salvaneschi
Different data processing goals

• Low latency (interactive) queries on historical data: enable faster decisions
  • E.g., identify why a site is slow and fix it

• Low latency queries on live data (streaming): enable decisions on real-time data
  • E.g., detect & block worms in real-time (a worm may infect 1mil hosts in 1.3sec)

• Sophisticated data processing: enable “better” decisions
  • E.g., anomaly detection, trend analysis
Goals

- **Easy** to combine *batch*, *streaming*, and *interactive* computations
- **Easy** to develop *sophisticated* algorithms
- **Compatible** with existing open source ecosystem (Hadoop/HDFS)
Memory use

- Aggressive use of memory can be a solution

- Memory transfer rates >> disk or even SSDs
  - Gap is growing especially w.r.t. disk

- Many datasets already fit into memory
  - The inputs of over 90% of jobs in Facebook, Yahoo!, and Bing clusters fit into memory
  - E.g., 1TB = 1 billion records @ 1 KB each

- Memory density (still) grows with Moore’s law
  - RAM/SSD hybrid memories at horizon
Spark

- Project start - UC Berkeley, 2009

- In February 2014, Spark became a Top-Level Apache Project

- Open source (mostly Scala code)

Pros and Cons of MapReduce

Greatly simplifies “big data” analysis on large, unreliable clusters
  • Simple interface: map and reduce
  • Hides details of parallelism, data partition, fault-tolerance, load-balancing...

• Problems
  • cannot support complex (iterative) applications efficiently
  • cannot support interactive applications efficiently

• Root cause
  • Inefficient data sharing

• Hardware had advanced since Hadoop started.
  • Very large RAMs, Faster networks (10Gb+).
  • Bandwidth to disk not keeping up

In MapReduce, the only way to share data across jobs is stable storage -> slow!
Limitations of MapReduce

Slow due to replication and disk I/O, but necessary for fault tolerance.
Goal: In-Memory Data Sharing
Challenges

10-100x faster than network/disk, but how to achieve fault-tolerance **efficiently**?

- Data replication?
- Log fine-grained updates to mutable states?

- Network bandwidth is scarce resource
- Disk I/O is slow
- Costly for data-intensive apps
Observation

Coarse-grained operation:
In many distributed computing, **same** operation is applied to multiple data items in parallel.
RDD (Resilient Distributed Datasets)

• Restricted form of distributed shared memory
  • immutable, partitioned collection of records
  • can only be built through coarse-grained deterministic transformations (map, filter, join...)

• Efficient fault-tolerance using lineage
  • Log coarse-grained operations instead of fine-grained data updates
  • An RDD has enough information about how it’s derived from other dataset
  • Recompute lost partitions on failure
Fault-tolerance
Spark and RDDs

• Implements Resilient Distributed Datasets (RDDs)

• Operations on RDDs
  • **Transformations**: defines new dataset based on previous ones
  • **Actions**: starts a job to execute on cluster

• Well-designed interface to represent RDDs
  • Makes it very easy to implement transformations
  • Most Spark transformation implementation < 20 LoC

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>partitions()</td>
<td>Return a list of Partition objects</td>
</tr>
<tr>
<td>preferredLocations(p)</td>
<td>List nodes where partition p can be accessed faster due to data locality</td>
</tr>
<tr>
<td>dependencies()</td>
<td>Return a list of dependencies</td>
</tr>
<tr>
<td>iterator(p, parentlers)</td>
<td>Compute the elements of partition p given iterators for its parent partitions</td>
</tr>
<tr>
<td>partitioner()</td>
<td>Return metadata specifying whether the RDD is hash/range partitioned</td>
</tr>
</tbody>
</table>

Table 3: Interface used to represent RDDs in Spark.
Simple Yet Powerful

WordCount Implementation: Hadoop vs. Spark

```java
public class WordCount {
    public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable>
    {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context
        ) throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }

        public static class IntSumReducer
            extends Reducer<Text, IntWritable, Text, IntWritable>
        {
            private IntWritable result = new IntWritable();

            public void reduce(Text key, Iterable<IntWritable> values,
            Context context
            ) throws IOException, InterruptedException {
                int sum = 0;
                for (IntWritable val : values) {
                    sum += val.get();
                }
                result.set(sum);
                context.write(key, result);
            }

            public static void main(String[] args) throws Exception {
                Configuration conf = new Configuration();
                Job job = Job.getInstance(conf, "word count");
                job.setJarByClass(WordCount.class);
                job.setMapperClass(TokenizerMapper.class);
                job.setCombinerClass(IntSumReducer.class);
                job.setReducerClass(IntSumReducer.class);
                job.setOutputKeyClass(Text.class);
                job.setOutputValueClass(IntWritable.class);
                FileInputFormat.addInputPath(job, new Path("input"));
                FileOutputFormat.setOutputPath(job, new Path("output"));
                System.exit(job.waitForCompletion(true) ? 0 : 1);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        val textfile = sc.textFile("hdfs://...");
        val counts = textfile.flatMap(line => line.split "").map(word => (word, 1)).reduceByKey(_ + _)
        counts.saveAsTextFile("hdfs://...");
    }
}
```
Spark

• Fast, expressive cluster computing system compatible with Apache Hadoop
  • Works with any Hadoop-supported storage system (HDFS, S3, Avro, ...)

• Improves efficiency through:
  • In-memory computing primitives
  • General computation graphs

• Improves usability through:
  • Rich APIs in Java, Scala, Python
  • Interactive shell

  Up to 100× faster

  Often 2-10× less code
More on RDDs

Work with distributed collections as you would with local ones

• Resilient distributed datasets (RDDs)
  • Immutable collections of objects spread across a cluster
  • Built through parallel transformations (map, filter, etc)
  • Automatically rebuilt on failure
  • Controllable persistence (e.g., caching in RAM)
    • Different storage levels available, fallback to disk possible

• Operations
  • Transformations (e.g. map, filter, groupBy, join)
    • Lazy operations to build RDDs from other RDDs
  • Actions (e.g. count, collect, save)
    • Return a result or write it to storage
Workflow with RDDs

- Create an RDD from a data source: `<list>`
- Apply transformations to an RDD: map, filter
- Apply actions to an RDD: collect, count

```
distFile = sc.textFile("...", 4)
```
- RDD distributed in 4 partitions
- Elements are lines of input
- *Lazy evaluation* means no execution happens now
Example: Mining Console Logs

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

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split('t')[2])
messages.cache()

messages.filter(lambda s: "foo" in s).count()
messages.filter(lambda s: "bar" in s).count()
...
```

Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)
Partitions

• Programmer specifies number of partitions for an RDD
  • Default value used if unspecified
  • *more partitions = more parallelism* (If workers are available)
RDD partition-level view

Dataset-level view:

log:

HadoopRDD
path = hdfs://...

errors:

FilteredRDD
func = _.contains(...) shouldCache = true

Partition-level view:

Task 1  Task 2  ...

log: HadoopRDD path = hdfs://...

errors: FilteredRDD func = _.contains(...) shouldCache = true
Job scheduling

```
rdd1.join(rdd2)
groupBy(...)
filter(...)
```

**RDD Objects**

- Build operator DAG

**DAGScheduler**

- Split graph into stages of tasks
- Submit each stage as ready

**TaskScheduler**

- Launch tasks via cluster manager
- Retry failed or straggling tasks

**Worker**

- Execute tasks
- Store and serve blocks

source: https://cwiki.apache.org/confluence/display/SPARK/Spark+Internals
RDD Fault Tolerance

RDDs track the transformations used to build them (their *lineage*) to recompute lost data.

E.g.: `messages = textFile(...).filter(lambda s: s.contains("ERROR")) .map(lambda s: s.split(‘\t’)\[2])`
Fault Recovery Test

![Graph showing iteration time for 10 iterations of k-means on 75 nodes, each iteration contains 400 tasks on 100GB data.]

running time for 10 iterations of k-means on 75 nodes, each iteration contains 400 tasks on 100GB data
Behavior with Less RAM

![Bar chart showing iteration time (s) vs. % of working set in cache.

Cache disabled: 69
25%: 58
50%: 41
75%: 30
Fully cached: 12

% of working set in cache]
Spark in Java and Scala

Java API:

```
JavaRDD<String> lines = spark.textFile(...);

errors = lines.filter(new Function<String, Boolean>() {
    public Boolean call(String s) {
        return s.contains("ERROR");
    }
});

errors.count()
```

Scala API:

```
val lines = spark.textFile(...)

errors = lines.filter(s => s.contains("ERROR"))
// can also write filter(_.contains("ERROR"))

errors.count
```
Creating RDDs

# Turn a local collection into an RDD
sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
sc.textFile("file.txt")
sc.textFile("directory/*.txt")
sc.textFile("hdfs://namenode:9000/path/file")

# Use any existing Hadoop InputFormat
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
Basic Transformations

```python
nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x*x)  # => {1, 4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0)  # => {4}

# Map each element to zero or more others
nums.flatMap(lambda x: range(0, x))  # => {0, 0, 1, 0, 1, 2}
```

Range object (sequence of numbers 0, 1, ..., x-1)
Basic Actions

```python
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Return first K elements
nums.take(2)   # => [1, 2]

# Count number of elements
nums.count()   # => 3

# Merge elements with an associative function
nums.reduce(lambda x, y: x + y) # => 6

# Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
```
Working with Key-Value Pairs

• Spark’s “distributed reduce” transformations act on RDDs of key-value pairs

• Python:  
  ```python
  pair = (a, b)
  pair[0]  # => a
  pair[1]  # => b
  ```

• Scala: 
  ```scala
  val pair = (a, b)
  pair._1 // => a
  pair._2 // => b
  ```

• Java: 
  ```java
  Tuple2 pair = new Tuple2(a, b);  // class scala.Tuple2
  pair._1 // => a
  pair._2 // => b
  ```
Some Key-Value Operations

```python
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])
pets.reduceByKey(lambda x, y: x + y)  
# => {(cat, 3), (dog, 1)}
pets.groupByKey()  
# => {(cat, Seq(1, 2)), (dog, Seq(1)}
pets.sortByKey()  
# => {(cat, 1), (cat, 2), (dog, 1)}

reduceByKey also automatically implements combiners on the map side
```
Example: Word Count

```
val lines = sc.textFile("hamlet.txt")
val counts = lines.flatMap(_._split(" "))
  .map((_, 1))
  .reduceByKey(x + y)
```

```
lines = sc.textFile("hamlet.txt")
counts = lines.flatMap(lambda line: line.split(" "))
  .map(lambda word: (word, 1))
  .reduceByKey(lambda x, y: x + y)
```

[Scala]

[Python]
Multiple Datasets

visits = sc.parallelize([(
    "index.html", "1.2.3.4"),
    ("about.html", "3.4.5.6"),
    ("index.html", "1.3.3.1")])

pageNames = sc.parallelize([(
    "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"))

visits.cogroup(pageNames)
# ("index.html", (Seq("1.2.3.4", "1.3.3.1"), Seq("Home")))
# ("about.html", (Seq("3.4.5.6"), Seq("About")))
Controlling the Level of Parallelism

- All the pair RDD operations take an optional second parameter for the **number of tasks**

```python
words.reduceByKey(lambda x, y: x + y, 5)
words.groupByKey(5)
visits.join(pageViews, 5)
```
Using Local Variables

• External variables you use in a closure will automatically be shipped to the cluster:

```python
query = raw_input("Enter a query:")
pages.filter(lambda x: x.startswith(query)).count()
```

• Some caveats:
  • Each task gets a new copy (updates aren’t sent back)
  • Variable must be Serializable (Java/Scala) or Pickle-able (Python)
  • Don’t use fields of an outer object (ships all of it!)
Closure Mishap Example

```scala
class MyCoolRddApp {
  val param = 3.14
  val log = new Log(...)
  ...

  def work(rdd: RDD[Int]) {
    rdd.map(x => x + param)
    .reduce(...)
  }
}
```

NotSerializableException: MyCoolRddApp (or Log)

How to get around it:

```scala
class MyCoolRddApp {
  ...

  def work(rdd: RDD[Int]) {
    val param_ = param
    rdd.map(x => x + param_)
    .reduce(...)
  }
}
```

References only local variable instead of this.param
Software Components

- Spark runs as a library in your program (one instance per app)
- Runs tasks locally or on a cluster
  - Standalone deploy cluster, Mesos or YARN
- Accesses storage via Hadoop InputFormat API
  - Can use HBase, HDFS, S3, ...

- A Spark program is two programs:
  A driver program and a workers program
- Worker programs run on cluster nodes or in local threads
- RDDs are distributed across workers
Task Scheduler

- Supports general task graphs
- Pipelines functions where possible
- Cache-aware data reuse & locality
- Partitioning-aware to avoid shuffles

```
Task Scheduler

A: = cached partition
B: = RDD

Stage 1
A: map
B: groupBy

Stage 2
C: map
D: filter
E: join

Stage 3
F: = cached partition

= RDD
```
Hadoop Compatibility

• Spark can read/write to any storage system/format that has a plugin for Hadoop!
  • Examples: HDFS, S3, HBase, Cassandra, Avro, SequenceFile
  • Reuses Hadoop’s InputFormat and OutputFormat APIs

• APIs like SparkContext.textFile support filesystems, while SparkContext.hadoopRDD allows passing any Hadoop JobConf to configure an input source
import spark.SparkContext
import spark.SparkContext._

object WordCount {
  def main(args: Array[String]) {
    val sc = new SparkContext("local", "WordCount", args(0), Seq(args(1)))
    val lines = sc.textFile(args(2))
    lines.flatMap(_.split(" "))
      .map(word => (word, 1))
      .reduceByKey(_ + _)
    .saveAsTextFile(args(3))
  }
}
import sys
from pyspark import SparkContext

if __name__ == "__main__":
    sc = SparkContext("local", "WordCount", sys.argv[0], None)
    lines = sc.textFile(sys.argv[1])

    lines.flatMap(lambda s: s.split(" "))
        .map(lambda word: (word, 1))
        .reduceByKey(lambda x, y: x + y)
        .saveAsTextFile(sys.argv[2])
Page Rank

• Give pages ranks (scores) based on links to them
  • Links from many pages ➔ high rank
  • Link from a high-rank page ➔ high rank

• Good example of a more complex algorithm
  • Multiple stages of map & reduce

• Benefits from Spark’s in-memory caching
  • Multiple iterations over the same data
Page Rank

\[ PR(x) = (1 - d) + d \sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)} \]

- Sketch of algorithm:
- Start with seed \( PR_i \) values
- Each page distributes \( PR_i \) “credit” to all pages it links to
- Each target page adds up “credit” from multiple in-bound links to compute \( PR_{i+1} \)
- Iterate until values converge
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page $p$ contribute $\frac{\text{rank}_p}{|\text{neighbors}_p|}$ to its neighbors
3. Set each page’s rank to $0.15 + 0.85 \times \text{contribs}$
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page $p$ contribute $\frac{\text{rank}_p}{|\text{neighbors}_p|}$ to its neighbors
3. Set each page’s rank to $0.15 + 0.85 \times \text{contribs}$
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page $p$ contribute $\frac{\text{rank}_p}{|\text{neighbors}_p|}$ to its neighbors
3. Set each page’s rank to $0.15 + 0.85 \times \text{contribs}$
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page $p$ contribute $\frac{\text{rank}_p}{|\text{neighbors}_p|}$ to its neighbors
3. Set each page’s rank to $0.15 + 0.85 \times \text{contribs}$
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page $p$ contribute $\frac{\text{rank}_p}{|\text{neighbors}_p|}$ to its neighbors
3. Set each page’s rank to $0.15 + 0.85 \times \text{contribs}$
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page $p$ contribute $\frac{\text{rank}_p}{|\text{neighbors}_p|}$ to its neighbors
3. Set each page’s rank to $0.15 + 0.85 \times \text{contribs}$

Final state:
Page Rank: MapReduce (Just an intuition)

One PageRank iteration:

• Input:
  \[(id_1, [score_1^{(t)}, out_{11}, out_{12}, ..]), (id_2, [score_2^{(t)}, out_{21}, out_{22}, ..])] \ldots

• Output:
  \[(id_1, [score_1^{(t+1)}, out_{11}, out_{12}, ..]), (id_2, [score_2^{(t+1)}, out_{21}, out_{22}, ..])] \ldots

Input format must match the output format.
Pseudocode

fun map( key: url, value: [pagerank, outlink_list] )

foreach outlink in outlink_list
    emit( key: outlink, value: pagerank/size(outlink_list) )
emit( key: url, value: outlink_list )

fun reduce( key: url, value: list_pr_or_urls )

outlink_list = []
pagerank = 0
foreach pr_or_urls in list_pr_or_urls
    if is_list( pr_or_urls )
        outlink_list = pr_or_urls
    else
        pagerank += pr_or_urls
pagerank = 0.15 + ( 0.85 * pagerank )
emit( key: url, value: [pagerank, outlink_list] )

(1) Each page distributes $PRI$ “credit” to all pages it links to

(2) Each target page adds up “credit” from multiple in-bound links
The result of each iteration is persisted!

Slow due to replication and disk I/O, but necessary for fault tolerance
Scala Implementation ()

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

for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (neighbors, rank)) =>
      neighbors.map(dest => (dest, rank/neighbors.size))
  }
  ranks = contribs.reduceByKey(_ + _) // Sum all links pointing to each url
    .mapValues(0.15 + 0.85 * _)
}

ranks.saveAsTextFile(...)
```

// Intermediate values
scala> contributions.collect
Scala> Array[(String, Double)] =
Array((MapR,1.0), (Baidu,0.5), (Blogger,0.5), (Google,0.5), (Baidu,0.5), (MapR,1.0))

// A possible initialization
val links = sc.parallelize(List(("MapR",List("Baidu","Blogger")),("Baidu", List("MapR")),("Blogger",List("Google","Baidu")),("Google", List("MapR"))))
  .partitionBy(new HashPartitioner(4)).persist()
var ranks = links.mapValues(v => 1.0)
```
Python Implementation

```
links = # RDD of (url, neighbors) pairs
ranks = # RDD of (url, rank) pairs

for i in range(NUM_ITERATIONS):
    def compute_contribs(pair):
        [url, [links, rank]] = pair  # split key-value pair
        return [(dest, rank/len(links)) for dest in links]

    contribs = links.join(ranks).flatMap(compute_contribs)
    ranks = contribs.reduceByKey(lambda x, y: x + y)\
        .mapValues(lambda x: 0.15 + 0.85 * x)

    ranks.saveAsTextFile(...)```
PageRank Performance

![PageRank Performance Chart]

- Iteration time (s) vs Number of machines
- Comparison between Hadoop and Spark
- Data points: 171, 23, 80, 14
Other Iterative Algorithms

- **K-Means Clustering**
  - Time per Iteration (s): 4.1 (Hadoop) and 0.96 (Spark)
  - Total Time: 155 seconds

- **Logistic Regression**
  - Time per Iteration (s): 110 seconds
Spark ecosystem

Spark SQL
Spark Streaming
MLlib (machine learning)
GraphX (graph)

Apache Spark
Sources & References

On the problem with the stack of big data analytics

RDDs
• web.eecs.umich.edu/~mosharaf/Slides/EECS582/W16/030916-Qi-Spark.pptx

Spark

Extra: shark