Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Ashvin Goel
Electrical and Computer Engineering
University of Toronto

ECE1724

Authors: Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, Ion Stoica
Background

• MapReduce greatly simplified “big data” analysis on large, unreliable clusters

• But as soon as it got popular, users wanted more
  • More complex, iterative multi-stage applications
    • E.g., graph processing, machine learning
  • More interactive ad-hoc queries

• Why not use MapReduce?
  • Iterative and interactive queries require jobs to share data efficiently
  • With MapReduce, the only way to share data across jobs is through disks, which is slow
MapReduce Example

Iterative analysis

Interactive queries

Slow due to disk I/O and replication, but necessary for fault tolerance
Goal: Use Memory to Share Data

Iterative analysis

Interactive queries

10-100× faster than network/disk, but what about fault-tolerance?
Challenge With In-Memory Analytics

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

• Existing storage abstractions are based on fine-grained updates to mutable state
  • E.g., Databases, distributed shared memory, etc.
  • Require replicating data/logs for fault tolerance
    • Costly for data-intensive apps
    • 10-100x slower than memory writes
Solution: Resilient Distributed Datasets (RDDs)

- RDDs are **immutable, partitioned** collections of records
- Support **coarse-grained, deterministic, data-parallel** transformations (map, filter, reduce, join, groupby, ...)

A restricted form of distributed memory abstraction that enables efficient fault-tolerance
  - During normal operation, log transformations (input logging)
  - On failure, re-execute the deterministic transformations needed to recover lost partitions of RDDs
Generality of RDDs

• RDDs can express many parallel algorithms that apply the same operation to many items

• Unify many current programming models
  • Data flow models: MapReduce, Dryad, SQL, ...
  • Specialized models for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), ...

• Support new applications beyond these models
Spark Programming Interface

- Operations on RDDs
  - Transformations - create new RDDs
  - Actions - compute and output results
- Programmers can control partitioning
  - How data in RDD is partitioned across nodes
- Programmers can control persistence
  - Whether partitions are stored in RAM, disk, etc.
Example: Log Mining

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

```scala
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('t')(2))
messages.persist()

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

Results: scaled to 1 TB of data in 5-7 seconds (vs 170 s for on-disk data)
Architecture

- Spark app runs a driver program (master) and one or more executor programs on worker nodes
- Executors access input data blocks, perform data transformations on data partitions, and store outputs
- A cluster manager allocates resources (e.g., worker nodes) to different Spark apps
Implementation

- RDD nodes are grouped into stages
  - Stages are connected by shuffle-type operations (e.g., groupBy, reduce)

- Within each stage, transformations are:
  - Partition-aware
    - Avoids shuffles
  - Pipelined
    - Provides better locality
Tracking Lineage

- RDDs track their lineage, i.e., the graph of transformations that built them

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('t')(2))
```
Failure Recovery with Lineage

• Tracking lineage enables **selectively recovering data partitions on failure**

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('t')(2))
```

Diagram: HadoopRDD → filter → FilteredRDD → map → MappedRDD
Failure Recovery with Lineage

• Tracking lineage enables selectively recovering data partitions on failure

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR")).collect()
messages = errors.map(_.split('t')(2))
```

Need to re-execute filter and map on these partitions
Fault Recovery Results

![Fault Recovery Results](image)

Iteration time (s)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>119</td>
</tr>
<tr>
<td>2</td>
<td>57</td>
</tr>
<tr>
<td>3</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>58</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
</tr>
<tr>
<td>6</td>
<td>81</td>
</tr>
<tr>
<td>7</td>
<td>57</td>
</tr>
<tr>
<td>8</td>
<td>59</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
</tr>
<tr>
<td>10</td>
<td>59</td>
</tr>
</tbody>
</table>

Failure happens at iteration 6.
Example: PageRank

- Start each page with a rank of 1
- On each iteration, update each page’s rank to:

\[ \sum_{i \in \text{neighbors}} \frac{\text{rank}_i}{|\text{neighbors}_i|} \]

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

for (i <- 1 to ITERATIONS) {
    // map operation
    contribs = links.join(ranks).flatMap {
        (url, (neighbors, rank)) =>
            neighbors.map(neighbor => (neighbor, rank / links.size))
    }
    // shuffle operation
    ranks = contribs.reduceByKey(_ + _)
}
```
Example: PageRank

// input: RDD of (url, neighbors) pairs
links = { 
  (url1, (url2, url3, url4)),  
  (url2, (url1, url4)),  
  (url3, (url4)),  
  (url4, ()),  
}\n
// output at start of iteration: RDD of (url, rank) pairs
ranks = { 
  (url1, R1),  
  (url2, R2),  
  (url3, R3),  
  (url4, R4)  
}\n
// contributions of neighbors
contribs = { 
  (url2, R1/3), (url3, R1/3), (url4, R1/3),  
  (url3, R2/2), (url4, R2/2),  
  (url4, R3/1)  
}\n
// shuffle operation, output at end of iteration
ranks = { 
  (url1, R1/3), (url3, R1/3+R2/2), (url4, R1/3+R2/2+R3/1)  
}
Optimizing Placement

- links & ranks are repeatedly joined
- Can co-partition them to avoid shuffles
  - E.g., hash both on URL
  - Can also use app knowledge, e.g., hash on DNS name
PageRank, Optimized Placement

// input: RDD of (url, neighbors) pairs
links = {((url1, (url2, url3, url4)),
          (url2, (url3, url4)),
          (url3, (url4)),
          (url4, ()))}

// output at start of iteration: RDD of (url, rank) pairs
ranks = {(url1, R1),
         (url2, R2),
         (url3, R3),
         (url4, R4)}

// contributions of neighbors
contribs = {(url2, R1/3), (url3, R1/3), (url4, R1/3),
            (url3, R2/2), (url4, R2/2),
            (url4, R3/1)}

// shuffle operation, output at end of iteration
ranks = {(url2, R1/3), (url3, R1/3+R2/2), (url4, R1/3+R2/2+R3/1)}
PageRank Performance

Time per iteration (s)

Hadoop
Basic Spark
Spark + Controlled Partitioning

171
72
23
Conclusion

- RDDs offer a simple and efficient programming model for a broad range of applications
- Leverage the coarse-grained nature of many parallel algorithms for low-overhead recovery
Discussion
Q1

• Why does Spark require using immutable data structures and deterministic transformations?
Why does the paper argue that Spark has minimal cost when nothing fails? Is this correct?
Q3

• What are the types of applications for which Spark is suitable?
Q4

- What are the types of applications for which Spark is not suitable?
Q5

• What problems can arise when transformations cause skew? How can these problems be handled?