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

Paper by Matei Zaharia et al

Presented by Jiaxing Yang
Spark

- A unified analytics engine for general purpose data processing
  - Iterative Machine Learning
  - Page-rank computation
  - Data mining
  - Etc.
- The original team of the Spark project founded Databricks in 2013
- RDD is a core concept in Spark
Motivation

- Cluster computing frameworks are widely adopted for large-scale data analytics
- MapReduce and Dryad lack abstractions for levering distributed memories
  - Shared data through disks are inefficient
  - Bad for tasks needs to reuse data
- Distributed shared memory, key-value stores, databases, and etc offer fine-grained shared state update interfaces
  - Fault tolerance requires replication -- expensive for data intensive tasks
- **Need an efficient, fault-tolerant method for data sharing through memory**
RDD Abstraction

RDD is a read-only, partitioned collection of records:

- Read-only: RDDs are immutable once generated
- Partitioned: An RDD consists of multiple partitions
  - Partitions can be stored by different machines
RDD Abstraction

RDDs can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs

- Such deterministic operations are called *transformations*
  - Coarse-grained operations
  - *map, filter, join, and etc.*
- An RDD has enough information called *lineage* to compute itself from stable data
Spark Programming Interface

Spark exposes RDDs through a set of APIs:

- Define RDDs with *transformations*
- Use data by *actions* on defined RDDs
  - *count, collect, ...*
- Execute *transformations* only when being used in an action
  - To pipeline *transformations*
Spark Programming Interface

To use Spark:

- Developers write a driver program
  - Connect to workers
  - Track RDDs’ lineage
  - Assume no failure
    - Actually easy to be replicated
Spark Programming Interface

Example: Console Log Mining

1. lines = spark.textFile("hdfs://...")
2. errors = lines.filter(_.startsWith("ERROR"))
3. errors.persist()
4. errors.count()
5. // Count errors mentioning MySQL:
6. errors.filter(_.contains("MySQL")).count()
7. // Return the time fields of errors mentioning
8. // HDFS as an array (assuming time is field
9. // number 3 in a tab-separated format):
10. errors.filter(_.contains("HDFS")).map(_.split(\'\t\')(3)).collect()
## Spark Programming Interface

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

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

- Coarse-grained transformations allow efficient fault tolerance
  - No checkpoint required
  - Only lost partitions need recomputation
- RDDs’ immutable nature enable slow nodes mitigation
  - Run backup tasks
  - Hard to do with DSM because copies of the task will access the same addresses and can interfere with each other

<table>
<thead>
<tr>
<th>Aspect</th>
<th>RDDs</th>
<th>Distr. Shared Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>Coarse- or fine-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Writes</td>
<td>Coarse-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Consistency</td>
<td>Trivial (immutable)</td>
<td>Up to app / runtime</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fine-grained and low-overhead using lineage</td>
<td>Requires checkpoints and program rollback</td>
</tr>
<tr>
<td>Straggler mitigation</td>
<td>Possible using backup tasks</td>
<td>Difficult</td>
</tr>
<tr>
<td>Work placement</td>
<td>Automatic based on data locality</td>
<td>Up to app (runtimes aim for transparency)</td>
</tr>
<tr>
<td>Behavior if not enough RAM</td>
<td>Similar to existing data flow systems</td>
<td>Poor performance (swapping?)</td>
</tr>
</tbody>
</table>
Applications Not Suitable for RDDs

- Applications requiring fine-grained updates on shared state.
  - Web crawler
Representing RDDs

Spark proposes representing RDDs through an interface with:

- A set of partitions
- A set of dependencies on parent RDDs
- A function for computing the dataset from the parents
- Metadata about partitioning scheme and data placement

<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($p$, parentIterators)</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>
Representing RDDs

Dependencies between RDDs

- Narrow dependencies
  - One parent partition is used by at most one child partition
  - *map*, *filter*, etc.

- Wide dependencies
  - One parent partition is used by multiple child partitions
  - *join*, etc.
Representing RDDs

Narrow dependencies are preferred:

- Narrow dependencies:
  - Pipelined execution on one cluster node to compute the parent partition
  - Only the lost parent partitions need to be recomputed
- Wide dependencies:
  - All parent partitions are required to be available and to be shuffled across nodes
  - A node failure may cause a complete re-execution
Spark Implementation

- ~14000 lines of Scala when paper published
  - Job scheduler
  - Spark interpreter
  - Memory management
  - Support for checkpointing
Job Scheduling

- Build a DAG of stages from RDD’s lineage graph
  - Boundaries of stages are
    - wide dependencies
    - already computed partitions
- Assign tasks based on locality
  - Send a task to where the data are
- For wide dependencies, materialize intermediate records
  - Easier fault recovery
Memory Management

Three options for storage of persistent RDDs:

- In-memory, deserialized objects
  - Fastest performance
- In-memory, serialized data
  - Memory efficiency
- On-disk
  - Limited memory
- LRU for partition eviction
  - Evict a partition of the least recently used RDD
    - Unless is the same RDD of the newly computed partition
Support for Checkpointing

- Checkpointing is helpful for RDDs with very long lineage graphs and wide dependencies
  - Hard to recompute
- For RDDs with short lineage graphs and narrow dependencies, checkpointing may never be worthwhile
  - Efficient to recompute
  - Disk I/O and usage may be too expensive
- RDDs are read-only
  - Be written out in the background
Evaluation

- Spark outperforms Hadoop by up to 20x in iterative machine learning and graph applications
  - Speedup comes from avoiding I/O and deserialization
- Customized applications perform and scale well with Spark
- Spark recover quickly when nodes fail
- Spark can be used to query 1-TB data in 5-7s
Iterative Machine Learning Applications

(a) Logistic Regression

(b) K-Means
Iterative Machine Learning Applications

- Minimum overhead of the Hadoop software stack
  - ~25s minimum overhead
- Overhead of HDFS while serving data
  - Memory copies, checksum
- Deserialization cost to convert binary records to usable in-memory Java objects
Page Rank

**Graph:**
- **x-axis:** Number of machines
- **y-axis:** Iteration time (s)
- **Legend:**
  - Hadoop
  - Basic Spark
  - Spark + Controlled Partitioning

- At 30 machines:
  - Hadoop: 171 seconds
  - Basic Spark: 72 seconds
  - Spark + Controlled Partitioning: 23 seconds

- At 60 machines:
  - Hadoop: 80 seconds
  - Basic Spark: 28 seconds
  - Spark + Controlled Partitioning: 14 seconds
Fault Recovery

![Bar Chart]

- Iteration time (s)
- Iteration
- No Failure
- Failure in the 6th Iteration

Values for each iteration:
- Iteration 1: 119
- Iteration 2: 57
- Iteration 3: 56
- Iteration 4: 58
- Iteration 5: 58
- Iteration 6: 81
- Iteration 7: 57
- Iteration 8: 59
- Iteration 9: 57
- Iteration 10: 59
Behavior with Insufficient Memory

![Graph showing iteration time (s) vs. percent of dataset in memory]

- 0%: 68.8
- 25%: 58.1
- 50%: 40.7
- 75%: 29.7
- 100%: 11.5
Expressing Existing Programming Models

- MapReduce
- DryadLINQ
- SQL
- Pregel
- Iterative MapReduce
- Batched Stream Processing

Apply the same operation to many records
Conclusion

- Resilient distributed dataset (RDD)
  - Efficient, general-purpose, fault-tolerant data abstraction
  - Can express a wide range of parallel applications
  - Use coarse-grained transformations to recover from faults
  - Implemented in Spark that outperforms Hadoop
The End

Thank You!