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

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica

Presented by: Wenjia He
Motivation

MapReduce and Dryad can help users access clusters’ computational resources to analyze large-scale data.

Restriction:

● Problem: Data reuse
  ○ Iterative machine learning and graph algorithms
  ○ Interactive data mining

● Solution: Stable storage instead of leveraging distributed memory
  ○ Overhead due to data replication, disk I/O and serialization
Motivation

Pregel and HaLoop are specialized frameworks for iterative applications.

Restriction:
- Support specific computation patterns
  - E.g. Looping MapReduce steps
- Share data implicitly
Motivation

Example of interactive data mining:

Input Data \[\rightarrow\] Ad-hoc query \[\rightarrow\] Result

Ad-hoc query \[\rightarrow\] Result

Ad-hoc query \[\rightarrow\] Result

Ad-hoc query \[\rightarrow\] Result

Slow!
Motivation

Restrictions of cluster computing frameworks like MapReduce and Dryad

Restrictions of specialized frameworks like Pregel and HaLoop

Distributed memory abstraction:

Resilient Distributed Datasets (RDD)
Challenge

Defining a programming interface to provide fault tolerance efficiently

Interface based on fine-grained updates (e.g. distributed shared memory)

Replicate data or log updates

Interface based on coarse-grained transformations

Log transformations
RDD

• Read-only and partitioned collection of records
• Created from data in stable storage or other RDDs
• Use lineage instead of materialization all the time
  ○ Useful for fault recovery
  ○ Small size in batch analytics
• Control *persistence* and *partitioning*
RDD example: console log mining

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

// 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()
```
### RDD representation

<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($p$, parentIters)</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>
RDD representation example

- HDFS files
  - `partitions()`: one partition for each block of the file
  - `preferredLocations(p)`: nodes that the block is on
  - `iterator(p, parentIter)`: read the block

- Map
  - `partitions()` and `preferredLocations(p)`: same as its parent
  - `iterator(p, parentIter)`: map function
# RDD Advantages

- Compared with DSM

<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>
RDD Advantages

● Express existing programming models
  ○ Including MapReduce, SQL, Pregel, …
● Leverage for debugging
  ○ Reconstruct RDDs and re-run tasks
Spark programming interface

- Provide RDD abstraction through API in Scala
- Driver program connected with workers
- Driver defines RDDs and long-lived workers
  store RDD partitions in RAM
## Spark operations

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

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

```scala
val points = spark.textFile(...)
  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```
Spark example: PageRank

```scala
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)
}
```
Spark implementation: job scheduling

- RDD dependencies:
  - Narrow dependency
  - Wide dependency
- Strength of narrow dependency:
  - Pipelined execution
  - Efficient recovery
Spark implementation: job scheduling

Build a DAG of stages to execute which contains as many pipelined transformations with narrow dependencies as possible.
Spark implementation: interpreter integration

- Class shipping
  Serve each line’s class over HTTP
- Modified code generation
  Reference instance of each line directly
Spark implementation: memory management

- Three kinds of persistent RDD storage
  - In-memory storage as deserialized Java objects
  - In-memory storage as serialized Java objects
  - On-disk storage

- Strategy: LRU eviction policy
  - Except the same RDD as the one with new partition
Spark implementation: checkpointing

- Problem:
  time-consuming when RDD recovery with long lineage chain

- Solution:
  Checkpoint some RDDs to stable storage

- API for checkpointing:
  REPLICA flag to *persist*

- Not worry about consistency due to read-only nature
Evaluation configuration

- Amazon EC2: m1.xlarge EC2 nodes with 4 cores and 15 GB of RAM
- HDFS used for storage with 256 MB blocks
- OS buffer caches cleared for IO cost measurement
Evaluation: machine learning application

- Spark achieves up to 25.3x and 20.7x speedup over Hadoop and HadoopBinMem in later iterations
- Improvement in logistic regression is larger because it is less compute-intensive
Evaluation: machine learning application

- Overhead of Hadoop software stack
  - At least 25s
- Overhead of HDFS while serving data
  - Memory copies and checksum
- Deserialization cost to convert binary records to Java objects
  - HDFS overhead: 2s
  - Text -> binary overhead: 7s
  - Binary -> java objects: 3s
  - Logistic regression computation: 3s
Evaluation: PageRank

- Link graph of ~4 million articles from 54 GB Wikipedia dump
- Spark achieves 2.4x and 7.4x speedup with partition not controlled and controlled
- Results scale linearly with the number of machines
Evaluation: fault recovery

- 400 tasks working on 100 GB data
- Node fail at the start of the 6th iteration
- RDD reconstruction takes around 23s
- RDD reconstruction only affects current iteration
Evaluation: insufficient memory

- Performance degrades gracefully with less space
Evaluation: user applications

- In-memory analytics
  - Video data analytics reports (e.g. computing statistics)
  - Speed up by 40x and less RAM requirement

- Traffic modeling
  - Expectation maximization (EM) algorithm
  - Scale linearly with node number

- Twitter Spam classification
  - Logistic regression with reduceByKey
  - Scaling not close to linear

(a) Traffic modeling
(b) Spam classification
Evaluation: interactive data mining

- Configuration: 100 m2.4xlarge EC2 instances with 8 cores and 68 GB of RAM
- Querying 1 TB of data on Spark takes 5-7s; while querying from disk takes 170s
Conclusion

RDD is an efficient, general-purpose and fault-tolerant distributed memory abstraction due to coarse-grained transformations

<table>
<thead>
<tr>
<th>Characteristics of related work</th>
<th>Characteristics of RDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data flow models (e.g. MapReduce)</td>
<td>Share through stable storage systems</td>
</tr>
<tr>
<td>High-level programming interfaces (e.g. DryadLINQ)</td>
<td>Cannot share data efficiently across queries</td>
</tr>
<tr>
<td>Specialized high-level interface (e.g. Pregel)</td>
<td>Share data implicitly</td>
</tr>
<tr>
<td>Systems that expose shared state (e.g. DSM systems)</td>
<td>Expensive recovery through checkpoints and roll-back</td>
</tr>
</tbody>
</table>
Limitation

- Batch application
- Not tolerate scheduler failures
Questions?