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

Presenter: Shucheng Zhong
Overview

• A new distributed memory abstraction: Resilient Distributed Datasets (RDDs)
  • In-memory computation on large clusters
  • fault tolerance

• An implementation of RDDs: Spark
Motivation

Cluster computing framework simplified large-scale data analytics (MapReduce, Dryad, etc.)

But, 2 types of applications are handled inefficiently

1. Multi-stage, iterative applications (ML, graph processing, etc.)
2. Interactive data mining

Do these applications have something in common?
Motivation

Iterative applications & Interactive data mining need:

- **Reuse data**

In most frameworks, the only way to reuse data across computations is writing data to an external stable storage system.

Data replication, disk I/O, serialization $\Rightarrow$ large overhead
MapReduce Examples

- Iterative application
- Interactive data mining
In-Memory Data Sharing

- Iterative application

- Interactive data mining
Solution: Resilient Distributed Datasets (RDDs)

RDDs are fault-tolerant, parallel-data structures that let users

• persist intermediate results in memory,

• control RDDs’ partitioning for optimization

• manipulating RDDs using a rich set of operators

Challenge: How to achieve fault-tolerance efficiently?
Fault Tolerance

Existing storage abstractions have interfaces based on fine-grained updates to mutable state (distributed shared memory, databases, Piccolo, etc.)

Achieve fault tolerance by
1. replicate data across machines
2. log updates across machines

Copying large amount of data with limited bandwidth
RDDS’ Fault Tolerance

Restricted form of distributed shared memory
• one RDD is an immutable, partitioned collections of records
• can only be built using coarse-grained deterministic transformations (e.g. map, filter, join)

Efficient fault-tolerance using lineage
• Log transformation operations instead of data
• Recompute lost partition on failure
• No need of replication
RDDs’ Fault Tolerance

RDDs track the graph of transformations that built them (their lineage) to rebuild lost data.
Overview

1. A new distributed memory abstraction: Resilient Distributed Datasets (RDDs)
   - In-memory computation on large clusters
   - Fault tolerance

2. An implementation of RDDs: Spark
Spark Programming Interface

Programmed in Scala and can be used interactively from Scala interpreter

```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
```

Example: Console Log Mining
Spark Programming Interface

Programmed in Scala and can be used interactively from Scala interpreter
Example: PageRank

Start each page with a rank of 1

On each iteration, update each page’s rank to

$$\Sigma_{i \in \text{neighbors}} rank_i / |\text{neighbors}_i|$$

links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs
for (i <- 1 to ITERATIONS) {
    ranks = links.join(ranks).flatMap {
        (url, (links, rank)) =>
        links.map(dest => (dest, rank/links.size))
    }.reduceByKey(_ + _)
Representing RDDs in Spark

Five pieces of information of each RDD:

1. a set of partitions
2. a set of dependencies on parent RDDs
3. a function for computing the dataset
4. metadata about its partitioning scheme
5. metadata about its data placement
Narrow Depend. vs. Wide Depend.

Narrow Dependencies:
• each partition is used at most once
• allow pipelined execution
• easy to fix on failure

Wide Dependencies:
• partitions can be used more than once
• have to wait until all parents available
• Require a complete re-execution on failure
Job Scheduling

- Dryad-like DAGs
- Pipeline functions within a stage
- Assign task based on locality
- Re-run tasks on failure

Example: How Spark computes job stage
Memory Management

Three options to store persistent RDDs:

1. in-memory storage as deserialized Java objects
   fastest performance as JVM accesses RDDs natively

2. in-memory storage serialized data
   slower but more memory-efficient

3. on-disk storage
   use LRU eviction policy at the level of RDDs by default
Evaluation: Performance on ML Apps

Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.
Evaluation: Fault Recovery

Run 10 iterations of k-means on a 75-node cluster.

At 6th iteration, one of the nodes is killed.

Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.
Evaluation: Behavior with Insufficient Memory

Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.
Summary

RDD is an efficient, general-purpose and fault-tolerant abstraction for sharing data in cluster applications.

RDD offers an API based on coarse-grained transformations that lets them recover data efficiently using lineage.

Spark is the implementation of RDDs in Scala.