Resilient Distributed Datasets

Sina Gholamnejad Davani

contents borrowed from:
https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia
Motivation

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

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

• Response: specialized frameworks for some of these apps (e.g. Pregel for graph processing)
Motivation

• Complex apps and interactive queries both need one thing that MapReduce lacks:
  • Efficient primitives for **data sharing**
• In MapReduce, the only way to share data across jobs is stable storage -&gt; slow!
Examples:

- HDFS read
- HDFS write
- HDFS read
- HDFS write

Input

iter. 1

iter. 2

... 

HDFS read

query 1

result 1

query 2

result 2

query 3

result 3

Input

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

10-100× faster than network/disk, but how to get FT?
Challenge:

• Existing storage abstractions have interfaces based on fine-grained updates to mutable state » RAMCloud, databases, distributed mem, Piccolo

• Requires replicating data or logs across nodes for fault tolerance
  • Costly for data-intensive apps
  • 10-100x slower than memory write
Solution: Resilient Distributed Datasets (RDDs)

• Restricted form of distributed shared memory
  • Immutable, partitioned collections of records
  • Can only be built through coarse-grained deterministic transformations (map, filter, join, …)
• Efficient fault recovery using lineage
  • Log one operation to apply to many elements
  • Recompute lost partitions on failure
  • No cost if nothing fails
Recovery

Input → iter. 1 → iter. 2 → ... 

one-time processing 

Input 

query 1 → output 
query 2 → output 
query 3 → output 
...
Tradeoff Space

Granularity of Updates

Fine

Coarse

Network bandwidth

Memory bandwidth

K-V stores, databases, RAMCloud

Best for transactional workloads

HDFS

Best for batch workloads

RDDs

Write Throughput

Low

High
Example: Log Mining

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

```python
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
```

**Result:** scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)
Question 6

• Explain Figure 1 about a lineage graph.

• Linear applying of multiple transformation operations and creating a new RDD in each step
Example: PageRank

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

$$\sum_{i \in \text{neighbors}} \text{rank}_i / |\text{neighbors}_i|$$

```
lights = // 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(_ + _)
}
Optimizing Placement

• Can *co-partition* them (e.g. hash both on URL) to avoid shuffles
PageRank Performance

Time per iteration (s)

Hadoop: 171
Basic Spark: 72
Spark + Controlled Partitioning: 23
Narrow VS Wide Dependencies:

Narrow Dependencies:
- `map, filter`
- `union`

Wide Dependencies:
- `groupByKey`
- `join with inputs not co-partitioned`
Question 1

• “...individual RDDs are immutable...” What does it mean by being “immutable”? What benefits does this property of RDD bring?

  • Not modifiable; low cost fault tolerance
Question 2

• When an RDD is being created (new data are being written into it), can the data in the RDD be read for computing before the RDD is completely created?

  • No, because in that point of time the data set is not completely settled and can not be used in other operations.
Question 3

• “This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its lineage) rather than the actual data.” “To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state.”

Why does using RDD help to provide efficient fault tolerance? Or why does coarse-grained transformation help with the efficiency?

• each data set can be represented by the input data and a set of transformations which is going to apply to many data records. This makes the rebuilding process for a lost data set fairly simple and low cost
Question 4

• What is difference between transformation and action?

  • transformations → apply the same operation to many data items
  • Actions → operations that return a value to the application or export data to a storage system
Question 5

• “In addition, programmers can call a persist method to indicate which RDDs they want to reuse in future operations.” What’s the consequence if a user does not explicitly request persistence of an RDD?

• the RDD would be discarded from the memory
Thank you