Fault Tolerant Distributed Main Memory Systems

CompSci 590.04
Instructor: Ashwin Machanavajjhala
Recap: Map Reduce

\[ \text{map}(k_1, v_1) \rightarrow \text{list}(k_2, v_2) \]

\[ \text{reduce}(k_2, \text{list}(v_1)) \rightarrow \text{list}(k_3, v_3) \]
Recap: Map Reduce

Programming Model

- Simple model
- Programmer only describes the logic

Distributed System

- Works on commodity hardware
- Scales to thousands of machines
- Ship code to the data, rather than ship data to code
- Hides all the hard systems problems from the programmer
  - Machine failures
  - Data placement
  - ...

Lecture 16 : 590.04 Fall 15
Recap: Map Reduce

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
Recap: Map Reduce

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

Thus arose many *specialized* frameworks for parallel processing
Recap: Pregel

Figure 2: Maximum Value Example. Dotted lines are messages. Shaded vertices have voted to halt.
GraphLab

Data Graph

Shared Data Table

Update Functions and Scopes

Scheduling

Lecture 16: 590.04 Fall 15
Problem with specialized frameworks

• Running multi-stage workflows is hard
  – Extract a mentions of celebrities from news articles
  – Construct a co-reference graph of celebrities (based on cooccurrence in the same article)
  – Analyze this graph (say connected components / page rank)

• Graph processing on Map Reduce is slow.

• The input does not have a graph abstraction. Map Reduce is a good candidate to construct the graph in the first place.
Root Cause Analysis

- Why do graph processing algorithms and iterative computation do poorly on Map Reduce?

- There is usually some (large) input that does not change across iterations. Map reduce unnecessarily keeps writing to and reading from disk.
Examples

• Page Rank
  Links in the graph do not change, only the rank of each node changes.

• Logistic Regression
  The original set of points do not change, only the model needs to be updated

• Connected components / K-means clustering
  The graph/dataset does not change, only the labels on the nodes/points changes.
Examples

• Page Rank
  Links in the graph do not change, only the rank of each node changes.

• Logistic Regression
  The original set of points do not change, only the model needs to be updated

• Connected components / K-means clustering
  The graph/dataset does not change, only the labels on the nodes/points changes.
Examples

• Page Rank
  Links in the graph do not change, only the rank of each node changes.

• Logistic Regression
  The original set of points do not change, only the model needs to be updated.

• Connected components / K-means clustering
  The graph/dataset does not change, only the labels on the nodes/points changes.
Idea: Load the “immutable” part into memory

• Twitter follows graph: 26GB uncompressed

• Can be stored in memory using 7 off the shelf machines each having 4 GB memory each.
Idea: Load the “immutable” part into memory

- Twitter follows graph: 26GB uncompressed

- Can be stored in memory using 7 off the shelf machines each having 4 GB memory each.

- Problem: Fault Tolerance!
Fault Tolerant Distributed Memory

• Solution 1: Global Checkpointing

• E.g., Piccolo (http://piccolo.news.cs.nyu.edu/)

• Problem: need to redo a lot of computation. (In Map Reduce: need to only to redo a Mapper or Reducer)
Fault Tolerant Distributed Memory

- Solution 2: Replication (e.g., RAMCloud)
RAMCloud

• Log Structured Storage
• Each master maintains in memory
  – An append only log
  – Hash Table (object id, location on the log)

• Every write becomes an append on the log
  – Plus a write to the hash table

• Log is divided into log segments
Durable Writes

- Write to the head of log (in master’s memory)
- Write to hash table (in master’s memory)
- Replication to 3 other backups
  - They each write to the backup log in memory and return
- Master returns as soon as ACK is received from replicas.

- Backups write to disk when the log segment becomes full.
Fault Tolerant Distributed Memory

• Solution 2: Replication

• Log Structured Storage (e.g., RAMCloud) + Replication

• Problem:
  – Every write triggers replication across nodes, which can become expensive.
  – Log needs constant maintenance and garbage cleaning.
Fault Tolerant Distributed Memory

• Moreover, existing solutions (Piccolo, RAMCloud, memcached) assume that objects in memory can be read as well as written.

• But, in most applications we only need objects in memory that are read (and hence immutable).
Fault Tolerant Distributed Memory

• Solution 3: Resilient Distributed Datasets

Restricted form of distributed shared memory
• Data in memory is immutable
• Partitioned collection of records
• Can only be built through coarse grained deterministic transformations (map, filter, join, etc)

Fault Tolerance through lineage
• Maintain a small log of operations
• Recompute lost partitions when failures occur
Example: Log Mining

```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
```

Original File

This is the RDD that is stored

First action triggers RDD computation and load into memory
RDD Fault Tolerance

- RDDs track the graph of operations used to construct them, called *lineage*.
- Lineage is used to rebuild data lost due to failures

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

- HadoopRDD
- FilteredRDD
- MappedRDD
RDD Fault Tolerance

- The larger the lineage, more computation is needed, and thus recovery from failure will be longer.

- Therefore, RDDs only allow operations that touch a large number of records at the same time.

<table>
<thead>
<tr>
<th>Transformations (define a new RDD)</th>
<th>map</th>
<th>filter</th>
<th>sample</th>
<th>groupByKey</th>
<th>reduceByKey</th>
<th>sortByKey</th>
<th>flatMap</th>
<th>union</th>
<th>join</th>
<th>cogroup</th>
<th>cross</th>
<th>mapValues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actions (return a result to driver program)</td>
<td>collect</td>
<td>reduce</td>
<td>count</td>
<td>save</td>
<td>lookupKey</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RDD Fault Tolerance

• The larger the lineage, more computation is needed, and thus recovery from failure will be longer.

• Therefore, RDDs only allow operations that touch a large number of records at the same time.
  - Great for batch operations
  - Not so good for random access or asynchronous algorithms.
Iterative Computation

• 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
}
```
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)
}
Transformations and Lineage Graphs

Narrow Dependencies:
- map, filter
- union

Wide Dependencies:
- join with inputs co-partitioned
- groupByKey
- join with inputs not co-partitioned

User can specify how data is partitioned to ensure narrow dependencies
it additionally takes into account which partitions of prescribed in Section 4.

Spark's scheduler uses our representation of RDDs, de-

5.1 Job Scheduling

5.2), memory man-

§ 5.4).

Our scheduler assigns tasks to machines based on data

stage's parents are still available. If some stages

to the parent's records in its

random access to elements of hash-partitioned RDDs by

our Spark

materialize intermediate records on the nodes

like MapReduce materializes map outputs.

holding parent partitions to simplify fault recovery, much

tolerate scheduler failures, though replicating the RDD

circuit the computation of a parent RDD. The scheduler

are the shuffle operations required for wide dependen-

dependencies as possible. The boundaries of the stages

contains as many pipelined transformations with narrow

Stages

A: B:

Stage 1

groupBy

Stage 2

map

Stage 3

union

join

Stage 1

Stage 2

Stage 3

Can pipeline execution as long as dependencies are narrow
Summary

• Map Reduce requires writing to disk for fault tolerance
• Not good for iterative computation.

RDD: Restricted form of distributed shared memory
• Data in memory is immutable
• Partitioned collection of records
• Can only be built through coarse grained deterministic transformations (map, filter, join, etc)

Fault Tolerance through lineage
• Maintain a small log of operations
• Recompute lost partitions when failures occur