Big Data Technology
Spark: In-memory Distributed Data Processing

Eshcar Hillel
Yahoo!

Ronny Lempel
Outbrain
Roadmap

- Previous class – MR Implementation

- This class – Spark
  - Fault tolerant distributed data processing
  - In-memory computation
  - Generalizes MapReduce
Spark in Hadoop

DATA ACCESS

Batch
- Map
-Reduce
Script
- Pig
Sql
- Hive
NoSql
- HBase
- Accumulo
- Phoenix
Stream
- Storm
Search
- Solr
In-Mem
- Spark
Others
- HAWQ Partners

YARN: Data Operating System

HDFS: Hadoop Distributed File System

DATA MANAGEMENT

https://hortonworks.com/products/data-center/hdp/

CS236620 Big Data Technology
Historical Perspective

- GFS 2003
- MR 2004
- Yahoo Hadoop 2006
- RDD 2012
- Hadoop Open Source 2008-9
- Hadoop 2.0 2013
- Spark 2014
Map-Reduce Critique

MR simplifies big data analysis on large clusters, however:

- “Too primitive”
  - Even the simplest tasks require programming
  - Data is non-structured – hard to manage

- Data flow is extremely rigid
  - Cannot handle iterative applications
  - Cannot handle stream processing
  - No interactive (shell) mode

- Sharing data only through (slow!) stable storage
1. Reading data from HDFS into mappers = disk IO (+ network traffic)
2. Writing mapped files to local disks = disk IO
3. Shuffling files from mappers to reducers = network traffic + disk IO
4. Reading local files into reducers = disk IO
5. Persisting data in HDFS = disk IO + network traffic

Network/disk is 10-100x slower than memory write
Examples of MR Use Cases

Disk I/O promotes fault tolerance, but is slow

With larger and faster RAM and super-fast LANs, can we do better?

Source: Resilient Distributed Datasets talk in NSDI 12
Prior Data Sharing Abstractions

- MR
  - Coarse-grained updates (batch)
  - FT: Replicating data on disk
- Key-value stores (lecture 7)
  - Fine-grained updates (per item)
  - FT: Replicating logs and data on disk
Goal: In-memory Data Sharing

Fast!

FT?
Apache Spark

- Fault tolerant in-memory distributed computing framework for large-scale data processing
- Based on Resilient Distributed Datasets (RDDs), which are collections of fault tolerant, immutable (read-only) partitions
- Supports multiple infrastructures
  - Hadoop, Cassandra, Amazon S3
- Multiple components and algorithms
- Rapid evolution: 1000+ contributors
Resilient Distributed Datasets

- Form of distributed shared memory
  - Immutable (read-only) partitions
  - Built through coarse-grained *transformations*
- Fault tolerance is ensured using *lineage*
  - Remember the transformations that derive each RDD
  - Reconstruct partitions upon failure in the sequence of needed transformations
  - No failures – no cost!
Fault Tolerance using Lineage

Lineage is a replicated log allows to re-compute lost RDDs

CS236620 Big Data Technology
Expressiveness of RDDs

RDD can express many parallel algorithms

- Data flow models: MR, SQL
- Iterative computations: machine learning, graph processing
- Incremental processing
- Interactive ad-hoc queries
How Does It Work?

- A driver program runs the main function
- First creates SparkContext object
  - Defines how to access the cluster
  - Used to create other variables
- Applies transformation and action to create RDDs
- May persist an RDD in memory
Spark Cluster Overview

Source: http://spark.apache.org/docs/latest/cluster-overview.html

CS236620 Big Data Technology
val lines = sc.textFile("hdfs://...")
val err = lines.filter(_.startsWith("ERROR"))
val msgs= err.map(_.split("\t")).map(r => r(1))
msgs.cache()
msgs.filter(_.contains("foo")).count()
msgs.filter(_.contains("bar")).count()
Transformation and Actions

- Transformations are lazy, not computed immediately
  - Map, filter, distinct, reduceByKey
- Transformations build the lineage graph (DAG)
- Allows to optimize the required computations
- Actions are when the DAG is executed
  - Reduce, count, forEach, saveAsText

Triggers a shuffle
Transformations vs. Actions

val lines = sc.textFile("hdfs://...")
val err = lines.filter(_.startsWith("ERROR"))
val msgs = err.map(_.split("\t")).map(...)
msgs.cache()
msgs.filter(_.contains("foo")).count()
msgs.filter(_.contains("bar")).count()
Log Mining Example

Second count only needs to go through step #4, #5

1. Read hdfs block
2. Process
3. Cache data
4. Partial count

Driver

Cache 1
Block 1
Worker

Cache 2
Block 2
Worker

Cache 3
Block 3
Worker

1. Read hdfs block
2. Process
3. Cache data
4. Partial count

5. Final count
RDD Persistence

- Persist an RDD across operations using `persist()` or `cache()`.
- Can use different storage level:
  - Only memory
  - Only disk
  - Spill to disk if cannot fit in memory
  - Replicate across nodes
Example: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page’s rank to
   \[ \sum_{i \in \text{neighbors}} \left( \frac{\text{rank}_i}{|\text{neighbors}_i|} \right) \]

```python
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs
for (i <- 1 to ITERATIONS) {
    contribs = links.join(ranks).flatMap {
        (url, (links, rank)) =>
        links.map(dest => (dest, rank/links.size))
    }
    ranks = contribs.reduceByKey((x,y) => x+y)
    .mapValues(sum => a/N + (1-a)*sum)
}
```

Source: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, NSDI 12
Optimizing Placement

- links and ranks are repeatedly joined
- Can co-partition them to avoid shuffles
## PageRank performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Time per iteration (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>171</td>
</tr>
<tr>
<td>Basic Spark</td>
<td>72 (x2.5)</td>
</tr>
<tr>
<td>Spark + controlled partitioning</td>
<td>23 (x8)</td>
</tr>
</tbody>
</table>
Summary

- Scalable platform for distributed computing
- Simple and efficient programing model for a broad range of applications
- Leverages coarse grained nature of many parallel algorithms for low overhead recovery

- Still some things are not to be solved by Spark …
Next Class

- Key-value store use case: user profile store
Further Reading

- www.spark-project.org