Data-Intensive Distributed Computing
CS 451/651 431/631 (Winter 2018)

Mix of slides from:
- Reza Zadeh http://reza-zadeh.com
- Jimmy Lin’s course at UWaterloo: http://lintool.github.io/bigdata-2018w/

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See http://creativecommons.org/licenses/by-nc-sa/3.0/us/ for details
So far in 416

Focused on distributed coordination

- Distributed algorithms: consensus, atomic commitment, mutual exclusion,…

- Distributed systems: CDN, DFS, BT, BChains, Chord, ..

What about programmability?
So far in 416

Focused on distributed coordination

- Distributed algorithms: consensus, atomic commitment, mutual exclusion,…

- Distributed systems: CDN, DFS, BT, BChains, Chord, ..

What about programmability?

Well, there is RPC. What, is that not enough?
Reality check

Data growing faster than processing speeds

Only solution is to parallelize on large clusters

» Widely use in both enterprises and web industry
Reality check

Data growing faster than processing speeds

Only solution is to parallelize on large clusters
  » Widely use in both enterprises and web industry

How do we let regular (non 416) developers program these things?
Why use a cluster,
Distributed compute:
Convex Optimization
Matrix Factorization
Machine Learning
Neural Networks

The Bootstrap
Numerical Linear Algebra
Large Graph analysis
Streaming and online
algorithms

Hadoop: 10K nodes, 150K cores, 150 PB (4/2014)

19 Hadoop clusters: 600 PB, 40k servers (9/2015)

Processes 20 PB a day (2008)
Crawls 20B web pages a day (2012)
Search index is 100+ PB (5/2014)
Bigtable serves 2+ EB, 600M QPS (5/2014)

Bigtable serves 2+ EB, 600M QPS (5/2014)

300 PB data in Hive + 600 TB/day (4/2014)

640K ought to be enough for anybody.

150 PB on 50k+ servers running 15k apps (6/2011)

JPMorganChase

150 PB data in Hive + 600 TB/day (4/2014)

300 PB data in Hive + 600 TB/day (4/2014)

LHC: ~15 PB a year

LSST: 6-10 PB a year (~2020)

SKA: 0.3 – 1.5 EB per year (~2020)

How much data?
The datacenter is the computer!
Traditional Dist. computing

Message-passing between nodes: RPC, MPI, ...

Very difficult to do at scale:
» How to split problem across nodes?
  • Must consider network & data locality
» How to deal with failures? (inevitable at scale)
» Even worse: stragglers (node not failed, but slow)
» Heterogeneity of nodes, their locations, complex env
» Have to write programs for each machine
Traditional Dist. computing

Message-passing between nodes: RPC, MPI, ...

Very difficult to do at scale:
  - How to split problem across nodes?
    - Must consider network & data locality
  - How to deal with failures? (inevitable at scale)
    - Even worse: stragglers (node not failed, but slow)
  - Even worse: stragglers (node not failed, but slow)
    - Heterogeneity of nodes, their locations, complex env
  - Heterogeneity of nodes, their locations, complex env
  - Have to write programs for each machine

Rarely used in commodity datacenters
Traditional Dist. computing

Message-passing between nodes: RPC, MPI, …

Very difficult to do at scale:
- How to split problem across nodes?
- Must consider network & data locality
- How to deal with failures? (inevitable at scale)
- Even worse: stragglers (node not failed, but slow)
- Heterogeneity of nodes, their locations, complex env

Rarely used in commodity datacenters

Key question: how do we let developers leverage distribution without having them build a distributed system per use case?
The datacenter *is* the computer!

It’s all about the right level of abstraction
Moving beyond the von Neumann architecture
What’s the “instruction set” of the datacenter computer?

Hide system-level details from the developers
No more race conditions, lock contention, etc.
No need to explicitly worry about reliability, fault tolerance, etc.

Separating the *what* from the *how*
Developer specifies the computation that needs to be performed
Execution framework (“runtime”) handles actual execution

MapReduce is the first instantiation of this idea… but not the last!
Data Flow Models

Restrict the programming interface so that the system can do more automatically

Express jobs as graphs of high-level operators
  » System picks how to split each operator into tasks and where to run each task
  » Re-run parts for fault recovery

Best example: MapReduce
Why Use a Data Flow Engine?

Ease of programming
  » High-level functions instead of message passing

Wide deployment
  » More common than MPI, especially “near” data

Scalability to huge commodity node clusters
  » Even HPC world is now concerned about resilience

Examples: Spark, Pig, Hive, Storm, but initially publicized with MapReduce
Roots in Functional Programming

Simplest data-parallel abstraction
Process a large number of records: “do” something to each

Map

We need something more for sharing partial results across records!
Roots in Functional Programming

Let’s add in aggregation!

Map

Fold

MapReduce = Functional programming + distributed computing!
A Data-Parallel Abstraction

Process a large number of records

Map “Do something” to each

Group intermediate results

“Aggregate” intermediate results

Reduce

Write final results

Key idea: provide a functional abstraction for these two operations
MapReduce

Programmer specifies two functions:

\[
\text{map} \ (k_1, \ v_1) \rightarrow \text{List}[(k_2, \ v_2)] \\
\text{reduce} \ (k_2, \ \text{List}[v_2]) \rightarrow \text{List}[(k_3, \ v_3)]
\]

All values with the same key are sent to the same reducer

The execution framework handles everything else…
group values by key

1. Map 
   - Map 1: a1, b2, c3, c6
   - Map 2: a5, c2
   - Map 3: b7, c8

2. Reduce 
   - Reduce 1: a, 1, 5
   - Reduce 2: b, 2, 7
   - Reduce 3: c, 2, 3, 6, 8

3. Values 
   - r1, s1
   - r2, s2
   - r3, s3
“Hello World” MapReduce: Word Count

def map(key: Long, value: String) = {
    for (word <- tokenize(value)) {
        emit(word, 1)
    }
}

def reduce(key: String, values: Iterable[Int]) = {
    for (value <- values) {
        sum += value
    }
    emit(key, sum)
}
“Hello World” MapReduce: Word Count

Input Files

Apple Orange Mango
Orange Grapes Plum

Apple Plum Mango
Apple Apple Plum

Each line passed to individual mapper instances

Apple Orange Mango
Orange Grapes Plum

Map Key Value Splitting

Apple,1
Orange,1
Mango,1

Apple,1
Grapes,1
Plum,1

Reduce Key Value Pairs

Apple,4
Apple,1
Apple,1
Apple,1

Grapes,1
Grapes,1

Mango,1
Mango,1

Orange,1
Orange,1

Plum,1
Plum,1
Plum,1

Final Output

Apple,4
Grapes,1
Mango,2
Orange,2
Plum,3
MapReduce

Programmer specifies two functions:

\[
\begin{align*}
  \text{map} & \quad (k_1, v_1) \rightarrow \text{List}[(k_2, v_2)] \\
  \text{reduce} & \quad (k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_3, v_3)]
\end{align*}
\]

All values with the same key are sent to the same reducer

The execution framework handles everything else…
What’s “everything else”?
MapReduce “Runtime”

Handles scheduling
Assigns workers to map and reduce tasks

Handles “data distribution”
Moves processes to data

Handles synchronization
Groups intermediate data

Handles errors and faults
Detects worker failures and restarts

Everything happens on top of a distributed FS (HDFS)
MapReduce Implementations

Google has a proprietary implementation in C++
  Bindings in Java, Python

Hadoop provides an open-source implementation in Java
  Development begun by Yahoo, later an Apache project
  Used in production at Facebook, Twitter, LinkedIn, Netflix, …
  Large and expanding software ecosystem
Potential point of confusion: Hadoop is more than MapReduce today

Lots of custom research implementations
Limitations of MapReduce

MapReduce is great at one-pass computation, but inefficient for multi-pass algorithms

No efficient primitives for data sharing
  » State between steps goes to distributed file system
  » Slows down pipeline: replication & disk storage
Logical View

Map:

- $k_1 v_1$
- $k_2 v_2$
- $k_3 v_3$
- $k_4 v_4$
- $k_5 v_5$
- $k_6 v_6$

Map:

- a 1 b 2
- c 3 c 6
- a 5 c 2
- b 7 c 8

Group values by key:

- a 1 5
- b 2 7
- c 2 3 6 8

Reduce:

- $r_1 s_1$
- $r_2 s_2$
- $r_3 s_3$
Physical View

Adapted from (Dean and Ghemawat, OSDI 2004)
Physical View

Adapted from (Dean and Ghemawat, OSDI 2004)
Example: Iterative Apps

Commonly spend 90% of time doing I/O
Example: PageRank

Repeatedly multiply sparse matrix and vector

Requires repeatedly hashing together page adjacency lists and rank vector

Neighbors (id, edges)

Ranks (id, rank)

iteration 1

iteration 2

iteration 3

...
MapReduce -> Spark

While MapReduce is simple, composing multiple M/R stages has a huge I/O cost: network + disk

Spark compute engine:
Extends a PL with data-flow operators and in-memory distributed collection data-structure
» “Resilient distributed datasets” (RDD)
Spark
Answer to “What’s beyond MapReduce?”

Brief history:
Developed at UC Berkeley AMPLab in 2009
Open-sourced in 2010
Became top-level Apache project in February 2014
Commercial support provided by DataBricks
Data Flow Models

- Restrict the programming interface so that the system can do more automatically
- Express jobs as graphs of high-level operators
  - System picks how to split each operator into tasks and where to run each task
  - Re-run parts for fault recovery
- Best example: MapReduce

Spark: more types of graph ops + in-memory datasets

Map-like Operations

\[
\begin{align*}
\text{map} & \quad f: (T) \Rightarrow U \\
\text{filter} & \quad f: (T) \Rightarrow \text{Boolean} \\
\text{flatMap} & \quad f: (T) \Rightarrow \text{TraversableOnce}[U] \\
\text{mapPartitions} & \quad f: (\text{Iterator}[T]) \Rightarrow \text{Iterator}[U]
\end{align*}
\]

(Not meant to be exhaustive)
Reduce-like Operations

- **groupByKey**
  - Input: RDD[(K, V)]
  - Output: RDD[(K, Iterable[V])]

- **reduceByKey**
  - Input: RDD[(K, V)]
  - Function: f: (V, V) ⇒ V
  - Output: RDD[(K, V)]

- **aggregateByKey**
  - Input: RDD[(K, V)]
  - SeqOp: (U, V) ⇒ U
  - CombOp: (U, U) ⇒ U
  - Output: RDD[(K, U)]

(Not meant to be exhaustive)
Sort Operations

- `sort`
- `repartitionAndSortWithinPartitions`

(Not meant to be exhaustive)
Join-like Operations

\[
\text{RDD}[(K, V)] \quad \text{RDD}[(K, W)]
\]

\[
\text{join} \quad \text{cogroup}
\]

\[
\text{RDD}[(K, (V, W))] \\
\text{RDD}[(K, (V, W))]
\]

\[
\text{RDD}[(K, (V, W))] \\
\text{RDD}[(K, (Iterable[V], Iterable[W]))]
\]

(Not meant to be exhaustive)
Join-like Operations

leftOuterJoin

fullOuterJoin

(RDD[(K, V)])  (RDD[(K, W)])

(RDD[(K, (V, Option[W]))])  (RDD[(K, (Option[V], Option[W]))])

(Not meant to be exhaustive)
Set-ish Operations

- `union`
- `intersection`

(Not meant to be exhaustive)
Set-ish Operations

**distinct**

- `RDD[T]` → `RDD[T]`

**cartesian**

- `RDD[T]` → `RDD[(T, U)]`

(Not meant to be exhaustive)
Spark Word Count

```scala
val textFile = sc.textFile(args.input())

textFile
  .flatMap(line => tokenize(line))
  .map(word => (word, 1))
  .reduceByKey((x, y) => x + y)
  .saveAsTextFile(args.output())
```
What’s an **RDD**?
Resilient Distributed Dataset (RDD)

= immutable  = partitioned

» Immutable collections of objects, spread across cluster
» Statically typed: RDD[T] has objects of type T

Wait, so how do you actually do anything?
Developers define *transformations* on RDDs
Framework keeps track of lineage
Transformations are lazy:
Framework keeps track of lineage

Actions trigger actual execution

RDD Lifecycle
Spark Word Count

```scala
val textFile = sc.textFile(args.input())
val a = textFile.flatMap(line => line.split(" "))
val b = a.map(word => (word, 1))
val c = b.reduceByKey((x, y) => x + y)
c.saveAsTextFile(args.output())
```
RDDs and Lineage

On HDFS

```
val textFile: RDD[String] = ... .flatMap(line => line.split(" "))
val a: RDD[String] = ... .map(word => (word, 1))
val b: RDD[(String, Int)] = ... .reduceByKey((x, y) => x + y)
val c: RDD[(String, Int)]
```

Remember, transformations are lazy!
RDDs and Lineage

On HDFS

textFile: RDD[String]

a: RDD[String]

b: RDD[(String, Int)]

c: RDD[(String, Int)]

Action!

Remember, transformations are lazy!
RDDs and Optimizations

Lazy evaluation creates **optimization** opportunities

**On HDFS**

```
RDDs don’t need to be materialized!
```

```
Optimize Map->Map
```

```
Optimize
```

```
Action!
```

```
RDDs and Optimizations
```

```
RDDs don’t need to be materialized!
```

```
Optimize Map->Map
```

```
Action!
```
RDDs and Caching

RDDs can be materialized in memory (and on disk)!

On HDFS

```
textFile: RDD[String]
  .flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey((x, y) => x + y)
  .saveAsTextFile(args.output())
```

Spark works even if the RDDs are partially cached!
Resilient Distributed Datasets (RDDs)

» Collections of objects across a cluster with user controlled partitioning & storage (memory, disk, ...)
» Built via parallel transformations (map, filter, ...)
» Only lets you make RDDs such that they can be:

Automatically rebuilt on failure
Spark Architecture
Spark Architecture
Fault Tolerance

RDDS track lineage info to rebuild lost data

```scala
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```
Fault Tolerance

RDDs track *lineage* info to rebuild lost data

```python
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```
Fault Tolerance

RDDs track *lineage* info to rebuild lost data

```python
file.map(lambda rec: (rec.type, 1))
  .reduceByKey(lambda x, y: x + y)
  .filter(lambda (type, count): count > 10)
```
Benefit of a single ecosystem

Same engine performs data extraction, model training and interactive queries

Separate engines

Single (Spark) engine
Example: graph processing
Spark: a general platform

Standard libraries included with Spark

- Spark SQL structured
- Spark Streaming real-time
- GraphX graph
- MLlib machine learning

Spark Core
Spark.ML Library (MLlib)

points = context.sql("select latitude, longitude from tweets")
model = KMeans.train(points, 10)

classification: logistic regression, linear SVM, naïve Bayes, classification tree

regression: generalized linear models (GLMs), regression tree

collaborative filtering: alternating least squares (ALS), non-negative matrix factorization (NMF)

clustering: k-means

decomposition: SVD, PCA

optimization: stochastic gradient descent, L-BFGS
Spark.GraphX

General graph processing library

Build graph using RDDs of nodes and edges

Large library of graph algorithms with composable steps
Spark Streaming

Run a streaming computation as a series of very small, deterministic batch jobs

- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches
Spark Streaming

Run a streaming computation as a series of very small, deterministic batch jobs

- Batch sizes as low as ½ second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system
Spark SQL

// Run SQL statements
val teenagers = context.sql(  
  "SELECT name FROM people WHERE age >= 13 AND age <= 19"
)

// The results of SQL queries are RDDs of Row objects
val names = teenagers.map(t => "Name: " + t(0)).collect()
Spark SQL

Enables loading & querying structured data in Spark

From Hive:
```python
c = HiveContext(sc)
rows = c.sql("select text, year from hivetable")
rows.filter(lambda r: r.year > 2013).collect()
```

From JSON:
```python
c.jsonFile("tweets.json").registerAsTable("tweets")
c.sql("select text, user.name from tweets")
```
Many other data-flow systems

Graph Computations: Pregel, GraphLab

SQL based engines: Hive, Pig, ...

ML engines: TensorFlow

... data-flow an ideal abstract? Who knows.
Take-aways

Data flow engines are important for distributed processing: simplify life for devs!

**MapReduce**: batch processing + distinct map and reduce phases. Inefficient and low level.

**Spark**: RDDs for fault tolerance; ecosystem.