MapReduce

CMPSC/DS 410

Reading

- Data Intensive Text Processing with MapReduce, Ch 2.
- Jeffrey Dean and Sanjay Ghemawat. **MapReduce: Simplified Data Processing on Large Clusters**. *Communications of the ACM* January 2008, Vol 52. No.1.

Brief History

- Invented at Google to process large data.
- Open source implementation developed with help from Yahoo!
  - Named Hadoop
  - Not an acronym for anything
  - Named after toy elephant

  - Because why name it after something descriptive?
- Now an Apache project

The Problem

- Large textfile
- Need to write wordcount
  - For each word, count the number of times it occurs
  - File might be too big to fit in memory
  - Number of words might be too big to fit in memory
- Applications:
  - Find word frequencies of English words in news stories
  - Find word frequencies of in Tweets

Small data solution

```
def wordcount(filename):
    counter = {}
    with open(filename) as infile:
        for line in infile:
            words = line.split()
            for w in words:
                counter[w] = counter.get(w, 0) + 1
    return counter
```


- Problem 1: file might not fit on your machine
- Problem 2: counter might not fit
- Counter is an associative array.
  - key, value pairs
- How would it be split across multiple machines?
  - Each machine is responsible for a set of keys.

Motivation

Parallel Reads

```
Node j (mapper):
    Read file shards on Node j.
    For each line encountered:
        words = line.split()
        for w in words:
            send message (w,1) to reducer
Reducer Node (does the counting/aggregation):
    Receive messages, increment word counts
```


- Problem 1: Data not all on one machine.
  - Each shard is 64 MB worth of text lines.
  - Multiple nodes read separate file shards in parallel.
- One node (reducer) stores the dictionary.
- Each mapper sends (many) messages to reducer node.
  - Message key: "word", message value: "1"
  - Reducer interprets message to mean "increment count of word by 1"

Motivation

Parallel Reads

Parallel Aggregation

```
Node j (mapper):
  Read file shards on Node j.
  For each line encountered:
    words = line.split()
    for w in words:
      determine reducer node  w
      send message (w,1) to appropriate reducer node
Reducer Node i:
  Receive messages, increment word counts
  At then end, save output to HDFS (as a file shard)
```


- Problem 2: Words cannot fit on one machine.
- Multiple reducers.
  - Each reducer is associated with set of words
    - E.g., Reducer 0 gets words where `python hash(words) % 6 == 0`
    - Reducer 1 gets words where `python hash(words) % 6 == 1`
    - In this specific example there are only 6 reducers (why?)
  - Mapper must send (w, 1) message to appropriate reducer
- Output of reducers cannot fit on one machine.
  - Output is sharded file, shard i produced by reducer i

MapReduce and Hadoop

- You define a:
  - mapper: what key/value messages to create from each line
  - partitioner: decide who is responsible for receiving each key
  - reducer: what to do with the messages received at a node
  - sorter: what order should the messages be received in
  - combiner: an optimization suggestion
- Hadoop:
  - Written in Java
  - MRJob: Convenient Python interface
    - Penalty for convenience:
    - Slower: uses hadoop streaming interface (communicates with java code through stdin and stdout
    - Less control of useful components (e.g., paritioner)
    - Benefits: shorter code

Mappers

- **mapper** function
  - inputs a key,value pair.
    - E.g., key = line number, value = line
    - E.g., key = null, value = line
  - outputs 0, 1, or more key value pairs.
- What Hadoop does automatically:
  - Finds Mapper Nodes that have file shards or can get them quickly.
  - Each Mapper Node reads its file shards
  - For every line, it calls the mapper function on it
  - Then it saves the results to local disk (not HDFS)
    - Messages for reducer 1 are saved in 1 file.
      - Sorted by keys assigned to that reducer
    - Messages for reducer 2 are saved in another.
      - Sorted by keys assigned to that reducer
  - All you need to do is write map function

```
from mrjob.job import MRJob

class WordCount(MRJob):
    def mapper(self, key, line):
        words = line.split()
        for w in words:
            yield (w, 1)
    def reducer(self, key, values):
        #TODO
```

Partitioners

- Mapper creates key,value pairs
  - e.g., ("it", 1), ("was", 1), ("dark", 1), ("rainy", 1), ("night", 1)
  - keys are the first part of tuple (words)
  - values are the second part (the numbers)
- (key, value) pairs are first saved to file then sent to **reducers** nodes
  - you specify how many reducers
- Partitioner:
  - function that takes a key and outputs a number
    - e.g., partition("was") = 2
    - so ("was", 1) is sent to reducer 2
  - if not specified, Hadoop will use a default
  - custom partitioner poorly supported by MRJob

```
from mrjob.job import MRJob

class WordCount(MRJob):
    def mapper(self, key, line):
        words = line.split()
        for w in words:
            yield w, 1

    def reducer(self, key, values):
        #####
```

- What Hadoop does automatically:
  - Reads output from each mapper
  - Collects key, value pairs for each reducer
    - Separate file for each reducer
      - Each file sorted by keys for that reducer
  - Sends (key, value) to appropriate reducer
  - Ensures reducers receive their keys in sorted order
  - "Shuffle and sort" phase
  - you can also specify sort order

Reducers

- Each reducer is responsible for set of keys
- **reduce** function
  - inputs a key, and list of values
    - values from all messages with that key
  - outputs a key/value pair
- What Hadoop does automatically:
  - Collects incoming key/value pairs
    - Gets these files from multiple mappers
    - Merges them to maintain sorted order
  - Groups by key to create key, value_list
  - Calls reducer(key, value_list) on each one
  - Saves result to HDFS

```
from mrjob.job import MRJob

class WordCount(MRJob):
    def mapper(self, key, line):
        words = line.split()
        for w in words:
            yield (w, 1)
    def reducer(self, key, values):
        yield (key, sum(values))

if __name__ == '__main__':
    WordCount.run()
```

Mapper Node 1 input shards:

That is that

to be or
not to be

Mapper Node 2 input shards:

That is big

Mapper 1's function outputs:

(That, 1)
(is, 1)
(that, 1)
(to, 1)
(be, 1)
(or, 1)
(not, 1)
(to, 1)
(be, 1)

Mapper 2's function outputs:

(That, 1)
(is, 1)
(big, 1)

Mapper Node 1 outputs:

Sorted file for Reducer 1:

(be, 1)
(be, 1)
(not, 1)
(that, 1)
(That, 1)

Sorted file for Reducer 2

(is, 1)
(or, 1)
(to, 1)
(to, 1)

Mapper Node 2 outputs:

Sorted file for Reducer 1

(big, 1)
(That, 1)

Sorted file for Reducer 2:

(is, 1)

- Note: reducer input is locally sorted, not globally sorted.
- Reducer gets inputs from multiple mappers
  - Merges inputs to maintain sorted order
  - Collects valus with same key

Reducer Node 1 input

be, [1, 1]
big, [1]
not, [1]
that, [1]
That, [1, 1]

Reducer Node 2 input

is, [1, 1]
or, [1]
to, [1, 1]

Reducer Node 1 input

```
be, [1, 1]
big, [1]
not, [1]
that, [1]
That, [1, 1]
```

Reducer Node 1 output (shard 1):

```
be 2
big 1
not 1
that 1
That 2
```


Reducer Node 2 input

```
is, [1, 1]
or, [1]
to, [1, 1]
```


Reducer Node 2 output (shard 2):

```
is 2
or 1
to 1
```

Programming Considerations

- Native language of Hadoop is Java
  - Writing jobs in java provides type safety
  - Writing jobs in java provides more control (e.g., custom partitioner)
- Need to worry about garbage collection.
  - Object churn: occurs when much of time is spent creating/garbage collecting objects.
  - Minimize churn by object re-use.
    - See WordCount2.java
    - The Key is created once per mapper (Text word = new Text();)
    - Then reused inside map function (word.set(token))
- Hadoop supports **counters** for maintaining job statistics
  - You control when to increment/decrement counters
  - Counters work in a distributed setting because increment/decrement are associative
    - increment/decrement messages can arrive out of order
    - but order does not matter to final result
    - counter only useful when job has finished

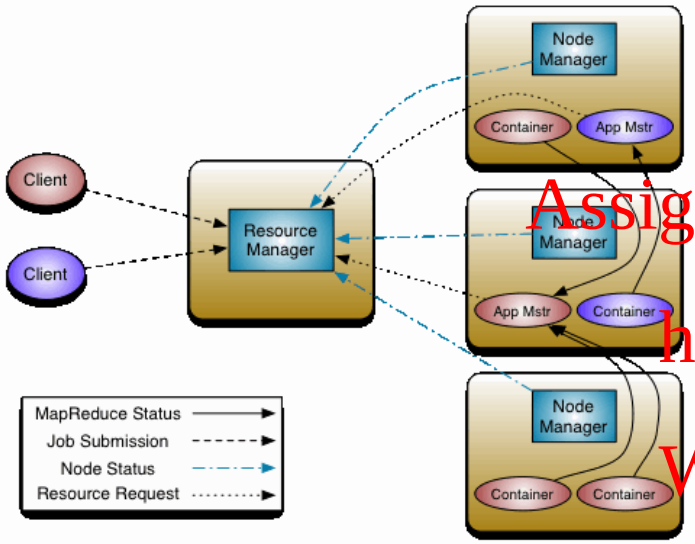
Hadoop Streaming

- Hadoop supports other languages (C, R, Python, etc.)
- Hadoop streaming
  - mapper and reducer written in language of your choice
  - key/value inputs sent to STDIN
  - key/value outputs read in STDOUT
  - tabs, spaces, non-ascii characters and language encodings are important (can cause crashes).
- Slower since requires communicating with/running code outside JVM
- mrjob provides an python interface to jobs using hadoop streaming.

Gotchas

- Multiple mappers run on different nodes.
  - Changing a class variable in one mapper does not affect other nodes!
  - Mappers run independently from each other
    - Why is this good for performance?
    - Why is this good for correctness and reduces potential for bugs?
- Reducers run on different nodes.
- Reducers and mappers only communicate in one way.
  - key,value pairs emitted by mapper are sent to some reducer
  - changing/setting a value in a mapper does not affect reducer

YARN Architecture

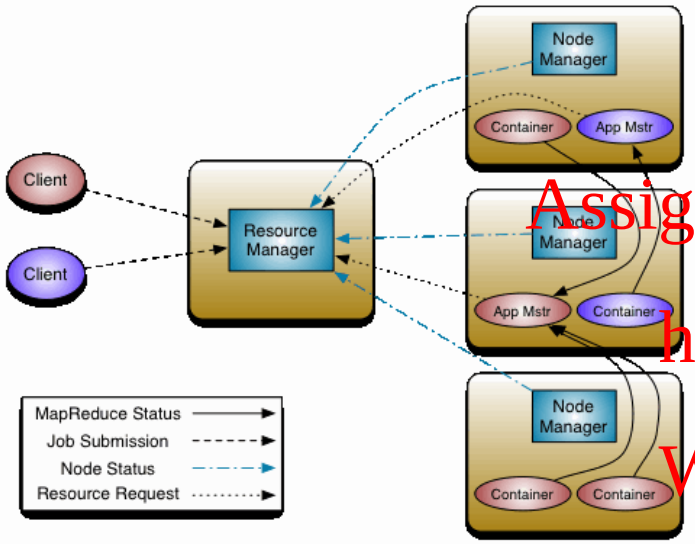


- Yet Another Resource Negotiator
- Comes with Hadoop 2.0
- Allows Hadoop, Spark, etc. to coexist on cluster.
- Components:
  - Container
  - Node manager
  - Resource manager
  - App Master

source:
http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

YARN Architecture

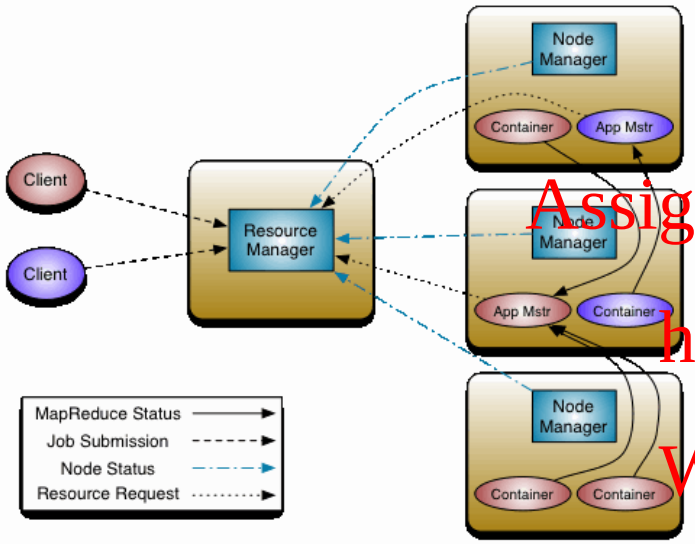
Container



- A share of cpu, memory, etc.
- Analgous to virtual machine

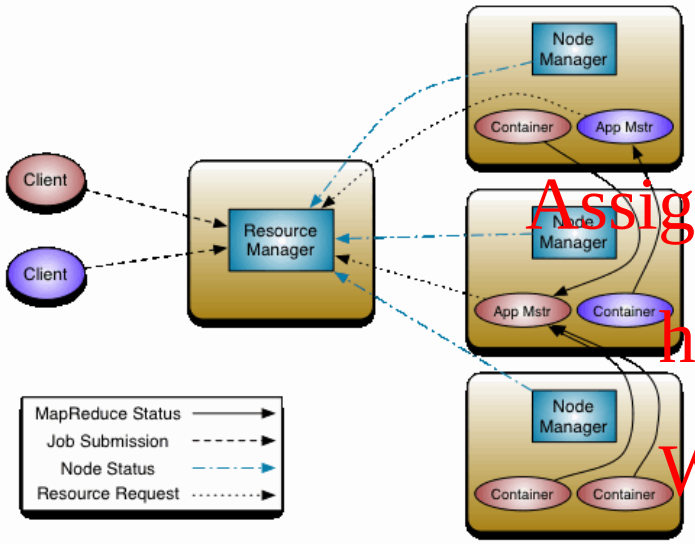
YARN Architecture

Node Manager



- Monitors containers
- Reports usage to Resource Manager

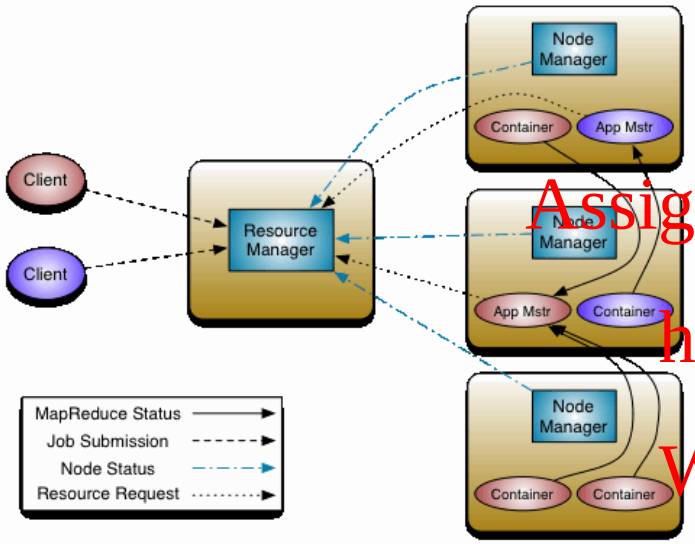
YARN Architecture



Resource Manager

- Scheduler and Applications Manager
- Scheduler maintains queue of waiting jobs
- Applications manager
  - accepts new jobs
  - decides first container for a job

YARN Architecture



App Master

- Starts in first container of your job
- Negotiates additional containers from Resource Manager
  - e.g., I need 10 mappers
  - e.g., I need 12 reducers
- Tracks progress of your job
  - e.g., Mapper 1 failed
    - Find new mappers with replicas of data shards Mapper 1 was using
    - Use those mappers to redo that part of computation
  - e.g., Mapper 2 is too slow, restart Mapper 2 in new container