CIS 833 – Information Retrieval and Text Mining

Lecture 4

Introduction to MapReduce

September 3, 2015

Credits for slides: Lin, Hofmann, Mihalcea, Mobasher, Mooney, Schutze.

Required Reading

- MapReduce & Hadoop
  - **Data-Intensive Text Processing with MapReduce** (Lin and Dyer) http://www.umiacs.umd.edu/~jimmylin/book.html
    - Chapter 2, MapReduce
    - Chapter 3, MapReduce Algorithm Design
  - **The Google File System** by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung from Google Labs

- Next:
  - "Information Retrieval" textbook
    - Chapter 1: Boolean Retrieval
    - Chapter 2: Term Vocabulary and Posting Lists
    - Chapter 4: Index Construction

Distributed File System

- Don't move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

GFS: Assumptions

- Commodity hardware over "exotic" hardware
  - Scale "out", not "up"
- High component failure rates
  - Inexpensive commodity components fail all the time
- "Modest" number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Sometimes concurrently
- Large streaming reads over random access
  - High sustained throughput/bandwidth over low latency

GFS: Design Decisions

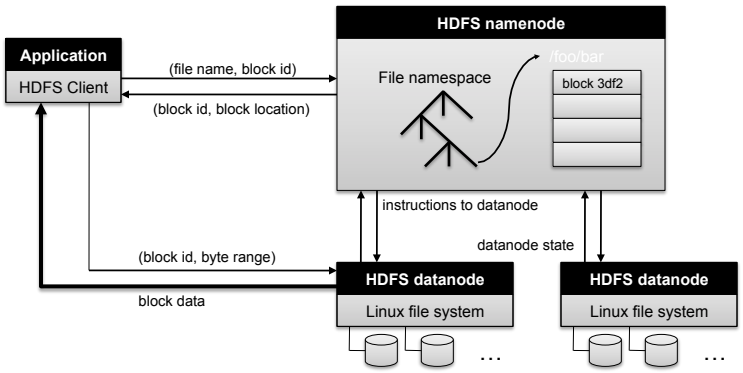
- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify the API
  - Push some of the issues onto the client (e.g., data layout)

HDFS = GFS clone (same basic ideas)

From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Functional differences:
  - No file appends in HDFS (planned feature)
  - HDFS performance is (likely) slower

HDFS Architecture

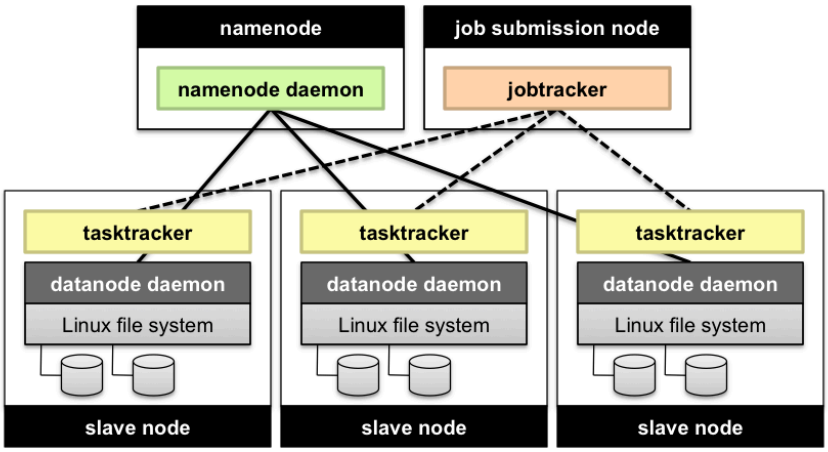


Adapted from (Ghemawat et al., SOSP 2003)

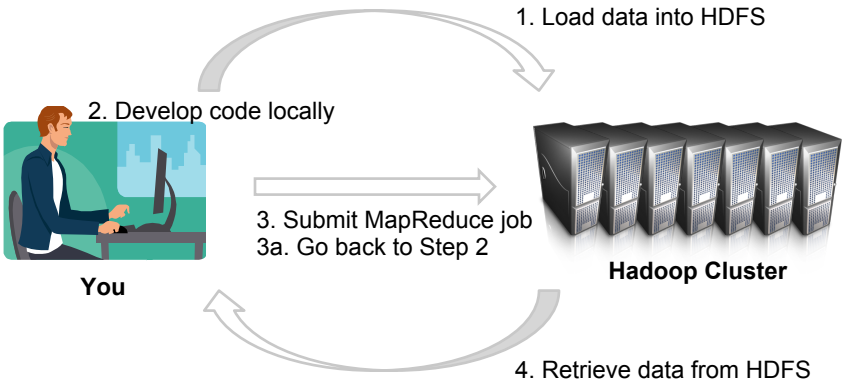
Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

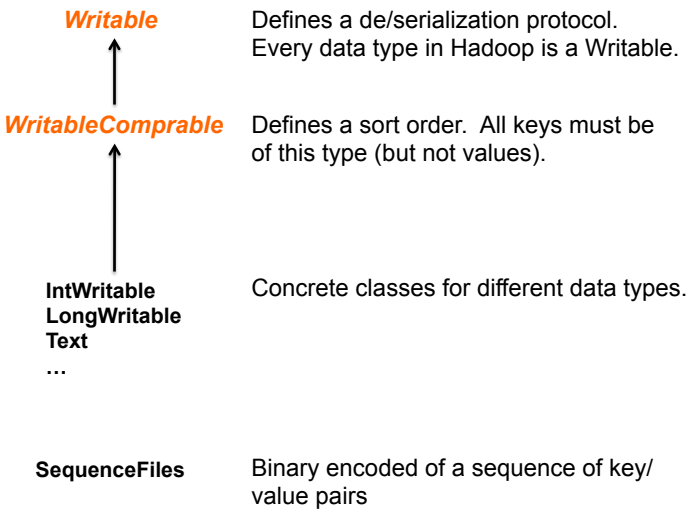
Putting everything together…



Hadoop Workflow



Data Types in Hadoop



Complex Data Types in Hadoop

- How do you implement complex data types?
- The easiest way:
  - Encoded it as Text, e.g., (a, b) = “a:b”
  - Use regular expressions to parse and extract data
  - Works, but pretty hack-ish
- The harder way:
  - Define a custom implementation of WritableComprable
  - Must implement: readFields, write, compareTo
  - Computationally efficient, but slow for rapid prototyping

Mappers

- Java object that implements the Map method.
- A mapper object is initialized for each map task (associated with a particular sequence of key-value pairs called an input split).
- A hook is provided in the API to run programmer-specified code.
- Mappers can read in "side data", providing an opportunity to load state, static data sources, dictionaries, etc. These method calls occur in the context of the same Java object, therefore it is possible to preserve state across multiple input key-value pairs within the same map task.
- The Map method is called on each key-value pair by the execution framework.
- After all key-value pairs in the input split have been processed, the mapper object provides an opportunity to run programmer- specified termination code.

Reducers

- Java object that implements the Reduce method
- A reducer object is initialized for each reduce task.
- The Hadoop API provides hooks for programmer-specified initialization and termination code.
- For each intermediate key in the partition (defined by the partitioner), the execution framework repeatedly calls the Reduce method with an intermediate key and an iterator over all values associated with that key.
- Since this occurs in the context of a single object, it is possible to preserve state across multiple intermediate keys (and associated values) within a single reduce task.

Importance of Local Aggregation

- Ideal scaling characteristics:
  - Twice the data, twice the running time
  - Twice the resources, half the running time
- Why can't we achieve this?
  - Synchronization requires communication
  - Communication kills performance
- Thus… avoid communication!
  - Reduce intermediate data via local aggregation
  - Combiners can help

Combiner Design

- Combiners and reducers share same method signature
  - Sometimes, reducers can serve as combiners
  - Often, not…
- Remember: combiners are optional optimizations
  - Should not affect algorithm correctness
  - May be run 0, 1, or multiple times

Word Count: Baseline

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count s)
```

What's the impact of combiners?

Word Count: Version 1

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1                ▷ Tally counts for entire document
6:         for all term t ∈ H do
7:             Emit(term t, count H{t})
```

Are combiners still needed?

Word Count: Version 2

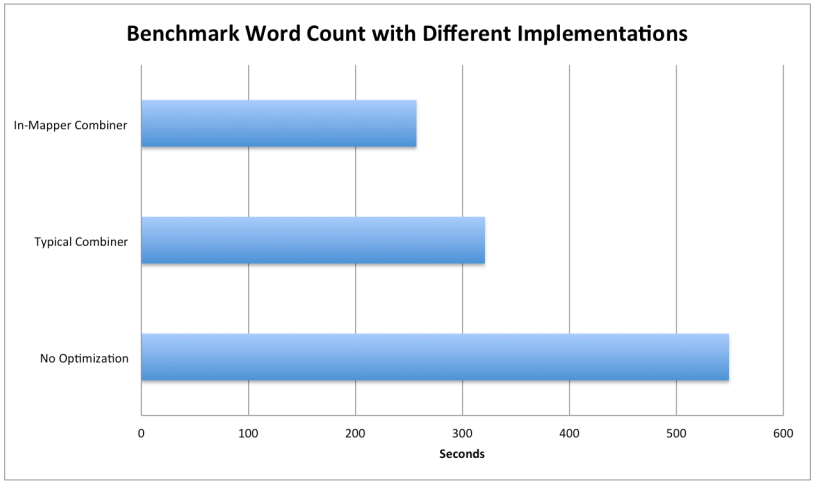
```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1                ▷ Tally counts across documents
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

Key: preserve state across input key-value pairs!

Are combiners still needed?

Design Pattern for Local Aggregation

- "In-mapper combining"
  - Fold the functionality of the combiner into the mapper by preserving state across multiple map calls (corresponding to a split)
- Advantages
  - Speed
  - Why is this faster than actual combiners?
- Disadvantages
  - Explicit memory management required
  - Potential for order-dependent bugs



English wikipedia dataset (the size of the 2 October 2013 dump used was approximately 9.5 GB compressed, 44 GB uncompressed)

https://alpinenow.com/blog/in-mapper-combiner/

Another example

- Example: find the average of all integers associated with the same key

Computing the average: Version 1

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)
1: class Reducer
2:     method Reduce(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Why can't we use reducer as combiner?

Computing the average: Version 2

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)
1: class Combiner
2:     method Combine(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         Emit(string t, pair (sum, cnt))          ▷ Separate sum and count
1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Why doesn't this work?

Computing the average: Version 3

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, pair (r, 1))

1: class Combiner
2:     method Combine(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         Emit(string t, pair (sum, cnt))

1: class Reducer
2:     method Reduce(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, pair (r_avg, cnt))
```

Fixed?

Computing the average: Version 4

```
1: class Mapper
2:     method Initialize
3:         S ← new AssociativeArray
4:         C ← new AssociativeArray
5:     method Map(string t, integer r)
6:         S{t} ← S{t} + r
7:         C{t} ← C{t} + 1
8:     method Close
9:         for all term t ∈ S do
10:            Emit(term t, pair (S{t}, C{t}))
```

Are combiners still needed?

Scalable Hadoop Algorithms: Themes

- Avoid object creation (especially if memory if limited in the cluster you are using)
  - Inherently costly operation
  - Garbage collection
- Create custom objects (especially if bandwidth is limited in the cluster you are using)
- Avoid buffering
  - Limited heap size
  - Works for small datasets, but won't scale!

```
for (int i = 0; i < 10; ++i) {
    StringBuilder buffer = new StringBuilder();
    buffer.append("Text");
    ...
}
```

```
StringBuilder buffer = new StringBuilder();
for (int i = 0; i < 10; ++i) {
    buffer.setLength(0);
    buffer.append("Text");
    ...
}
```

http://blogs.sequoiainc.com/blogs/hadoop-optimizing-mapreduce-jobs