Clustering
Lecture 8: MapReduce

Jing Gao
SUNY Buffalo
Outline

• Basics
  – Motivation, definition, evaluation

• Methods
  – Partitional
  – Hierarchical
  – Density-based
  – Mixture model
  – Spectral methods

• Advanced topics
  – Clustering ensemble
  – Clustering in MapReduce
  – Semi-supervised clustering, subspace clustering, co-clustering, etc.
Big Data EveryWhere

- Lots of data is being collected and warehoused
  - Web data, e-commerce
  - purchases at department/grocery stores
  - Bank/Credit Card transactions
  - Social Network
Divide and Conquer

"Work"

$w_1 \quad w_2 \quad w_3$

"worker"

$r_1 \quad r_2 \quad r_3$

"Result"

Partition

Combine
Distributed Grep

Very big data

Split data → grep → matches
Split data → grep → matches
Split data → grep → matches
Split data → grep → matches

→ cat → All matches
Distributed Word Count

Very big data

Split data $\rightarrow$ count $\rightarrow$ count $\rightarrow$ count $\rightarrow$ merge $\rightarrow$ merged count

Split data $\rightarrow$ count $\rightarrow$ count

Split data $\rightarrow$ count $\rightarrow$ count

Split data $\rightarrow$ count $\rightarrow$ count
Parallelization Challenges

• How do we assign work units to workers?
• What if we have more work units than workers?
• What if workers need to share partial results?
• How do we aggregate partial results?
• How do we know all the workers have finished?
• What if workers die?
Common Theme?

• **Parallelization problems arise from**
  – Communication between workers (e.g., to exchange state)
  – Access to shared resources (e.g., data)

• **Thus, we need a synchronization mechanism**
Managing Multiple Workers

• **Difficult because**
  – We don’t know the order in which workers run
  – We don’t know when workers interrupt each other
  – We don’t know the order in which workers access shared data

• **Thus, we need**
  – Semaphores (lock, unlock)
  – Conditional variables (wait, notify, broadcast)
  – Barriers

• **Still, lots of problems**
  – Deadlock, race conditions, ...

• **Moral of the story: be careful!**
Concurrency Challenge

• Concurrency is difficult to reason about
• Concurrency is even more difficult to reason about
  – At the scale of datacenters (even across datacenters)
  – In the presence of failures
  – In terms of multiple interacting services
• Not to mention debugging...
• The reality:
  – Lots of one-off solutions, custom code
  – Write you own dedicated library, then program with it
  – Burden on the programmer to explicitly manage everything
What’s the point?

• Right level of abstraction
  – multi-core/cluster environment

• Hide system-level details from the developers
  – No more race conditions, lock contention, etc.

• Separating the what from how
  – Developer specifies the computation that needs to be performed
  – Execution framework ("runtime") handles actual execution
MapReduce

• **Key properties**
  – Google has used successfully in processing its “big-data” sets (~ 20000 peta bytes per day)
  – Users specify the computation in terms of a *map* and a *reduce* function
  – Underlying runtime system automatically parallelizes the computation across large-scale clusters of machines
  – Underlying system also handles machine failures, efficient communications, and performance issues
MapReduce can refer to...

- The programming model
- The execution framework (aka “runtime”)
- The specific implementation

Usage is usually clear from context!
**Typical Large-Data Problem**

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

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

• Programmers specify two functions:
  \[
  \text{map} \ (k, \ v) \rightarrow [(k', \ v')] \\
  \text{reduce} \ (k', \ [v']) \rightarrow [(k', \ v')] 
  \]
  – All values with the same key are sent to the same reducer

• The execution framework handles everything else...
“Everything Else”

• The execution framework
  – Scheduling: assigns workers to map and reduce tasks
  – “Data distribution”: moves processes to data
  – Synchronization: gathers, sorts, and shuffles intermediate data
  – Errors and faults: detects worker failures and restarts

• Limited control over data and execution flow
  – All algorithms must expressed in m, r, c, p

• You don’t know:
  – Where mappers and reducers run
  – When a mapper or reducer begins or finishes
  – Which input a particular mapper is processing
  – Which intermediate key a particular reducer is processing
Architecture Overview

Master node

Job tracker

Slave node 1
Task tracker

Slave node 2
Task tracker

Slave node N
Task tracker

Workers

Workers

Workers

user
MapReduce Implementations

- **Google MapReduce**
  - Not available outside Google

- **Hadoop**
  - An open-source implementation in Java
  - Development led by Yahoo, used in production
  - Now an Apache project
  - Rapidly expanding software ecosystem

- **Custom research implementations**
  - For GPUs, cell processors, etc.
Who uses Hadoop?

- Amazon/A9
- Facebook
- Google
- IBM
- Joost
- Last.fm
- New York Times
- PowerSet
- Veoh
- Yahoo!
- ......

How do we get data to the workers?

What's the problem here?
Distributed File System

• **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**
  – GFS (Google File System) for Google’s MapReduce
  – HDFS (Hadoop Distributed File System) for Hadoop
Distributed File System Design

• Chunk Servers
  – File is split into contiguous chunks
  – Typically each chunk is 16-64MB
  – Each chunk replicated (usually 2x or 3x)
  – Try to keep replicas in different racks

• Master node
  – a.k.a. Name Nodes in HDFS
  – Stores metadata
  – Might be replicated

• Client library for file access
  – Talks to master to find chunk servers
  – Connects directly to chunk servers to access data
Hadoop HDFS

NameNode:
Stores metadata only

METADATA:
/user/aaron/foo → 1, 2, 4
/user/aaron/bar → 3, 5

DataNodes: Store blocks from files
Hadoop Cluster Architecture

From Jimmy Lin’s slides
Map+Reduce

• Map:
  – Accepts *input* key/value pair
  – Emits *intermediate* key/value pair

• Reduce:
  – Accepts *intermediate* key/value* pair
  – Emits *output* key/value pair

Very big data

**Result**
The Map Step

Input key-value pairs

Intermediate key-value pairs

map

...
The Reduce Step

Intermediate key-value pairs

Key-value groups

Output key-value pairs

reduce

reduce

reduce

...
MapReduce

• Input: a set of key/value pairs
• User supplies two functions:
  – map(k,v) $\rightarrow$ list(k1,v1)
  – reduce(k1, list(v1)) $\rightarrow$ (k1,v2)
• (k1,v1) is an intermediate key/value pair
• Output is the set of (k1,v2) pairs
Word Count

• We have a large collection of documents
• Count the number of times each distinct word appears in the collection of documents
Word Count Execution

Input: the quick brown fox, the fox ate the mouse, how now brown cow

Map:
- the, 1
- brown, 1
- fox, 1
- the, 1
- quick, 1
- how, 1
- now, 1
- brown, 1
- ate, 1
- mouse, 1
- cow, 1

Reduce:
- brown, 2
- fox, 2
- how, 1
- now, 1
- the, 3
- ate, 1
- cow, 1
- mouse, 1
- quick, 1
Word Count using MapReduce

map(key, value):
   // key: document name; value: text of document
   for each word w in value:
      emit(w, 1)

reduce(key, values):
   // key: a word; value: an iterator over counts
   result = 0
   for each count v in values:
      result += v
   emit(result)
Combiners

- Often a map task will produce many pairs of the form (k,v1), (k,v2), ... for the same key k
  - E.g., popular words in Word Count
- Can save network time by pre-aggregating at mapper
- For associative ops. like sum, count, max
- Decreases size of intermediate data
- Example: local counting for Word Count:

```python
def combiner(key, values):
    output(key, sum(values))
```
Word Count with Combiner

Input  Map & Combine    Shuffle & Sort    Reduce    Output

the quick brown fox

the fox ate the mouse

how now brown cow

the, 1
brown, 1
fox, 1

the, 2
fox, 1

how, 1
now, 1
brown, 1

ate, 1
mouse, 1

quick, 1

brown, 2
fox, 2
how, 1
now, 1
the, 3

ate, 1
cow, 1
mouse, 1
quick, 1
Partition Function

• Inputs to map tasks are created by contiguous splits of input file
• For reduce, we need to ensure that records with the same intermediate key end up at the same worker
• System uses a default partition function e.g., hash(key) mod R
• Sometimes useful to override
  – Balance the loads
  – Specific requirement on which key value pairs should be in the same output files
Shuffle and Sort: aggregate values by keys

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

map

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

map

map

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

Shuffle and Sort: aggregate values by keys

map

map

map

reduce

reduce

reduce

r_1 s_1 r_2 s_2 r_3 s_3
Shuffle and Sort: aggregate values by keys

\[
\begin{array}{cccccccc}
k_1 & v_1 & k_2 & v_2 & k_3 & v_3 & k_4 & v_4 \\
& & k_5 & v_5 & k_6 & v_6 & & \\
\end{array}
\]
How to MapReduce K-means

• Partition \( \{x_1, ..., x_n\} \) into \( K \) clusters
  – \( K \) is predefined

• Initialization
  – Specify the initial cluster centers (centroids)

• Iteration until no change
  – For each object \( x_i \)
    • Calculate the distances between \( x_i \) and the \( K \) centroids
    • (Re)assign \( x_i \) to the cluster whose centroid is the closest to \( x_i \)
  – Update the cluster centroids based on current assignment
K-Means Map/Reduce Design

**Traditional**

**AssignCluster():**
- For each point $p$
  - Assign $p$ the closest $c$

**UpdateCentroids():**
- For each cluster
  - Update cluster center

**Kmeans():**
- While not converge:
  - AssignCluster()
  - UpdateCentroids()
K-Means Map/Reduce Design

Map: assign each p to closest centroids

Reduce: update each centroid with its new location (total, count)

Kmeanslter()
Map(p) // Assign Cluster
- For c in clusters:
  - If dist(p,c)<minDist, then minC=c, minDist = dist(p,c)
  - Emit(minC.id, (p, 1))
Reduce() // Update Centroids
- For all values (p, c):
  - total += p; count += c;
- Emit(key, (total, count))
MapReduce K-means Algorithm

• **Driver**
  - Runs multiple iteration jobs using mapper+combiner+reducer

• **Mapper**
  - Configure: A single file containing cluster centers
  - Input: Input data points
  - Output: (data id, cluster id)

• **Reducer**
  - Input: (data id, cluster id)
  - Output: (cluster id, cluster centroid)

• **Combiner**
  - Input: (data id, cluster id)
  - Output: (cluster id, (partial sum, number of points))
MapReduce Characteristics

- Very large scale data: peta, exa bytes
- Map and Reduce are the main operations: simple code
- There are other supporting operations such as combine and partition
- All the map should be completed before reduce operation starts
- Map and reduce operations are typically performed by the same physical processor
- Number of map tasks and reduce tasks are configurable
- Operations are provisioned near the data
- Commodity hardware and storage
- Runtime takes care of splitting and moving data for operations
- Special distributed file system, such as Hadoop Distributed File System
## MapReducable?

<table>
<thead>
<tr>
<th></th>
<th>One Iteration</th>
<th>Multiple Iterations</th>
<th>Not good for MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clustering</strong></td>
<td>Canopy</td>
<td>KMeans</td>
<td></td>
</tr>
<tr>
<td><strong>Classification</strong></td>
<td>Naïve Bayes, kNN</td>
<td>Gaussian Mixture</td>
<td>SVM</td>
</tr>
<tr>
<td><strong>Graphs</strong></td>
<td></td>
<td>PageRank</td>
<td></td>
</tr>
<tr>
<td><strong>Information Retrieval</strong></td>
<td>Inverted Index</td>
<td>Topic modeling (PLSI, LDA)</td>
<td></td>
</tr>
</tbody>
</table>

- One-iteration algorithms are perfect fits
- Multiple-iteration algorithms are OK fits
  - but **small shared info** have to be synchronized across iterations (typically through filesystem)
- Some algorithms are not good for MapReduce framework
  - Those algorithms typically require **large shared info** with a lot of synchronization.
  - Traditional parallel framework like MPI is better suited for those.
**Development Cycle**

1. Scp data to cluster
2. Move data into HDFS
3. Develop code locally
4. Submit MapReduce job
   4a. Go back to Step 3
5. Move data out of HDFS
6. Scp data from cluster

---

*You*
Take-away Message

• MapReduce programming model
• How to design map, reduce, combiner, partition functions
• Which tasks can be easily MapReduced and which cannot