Algorithms for Data Science
Course Unit 2

The Map-Reduce Framework

Spyros Kontogiannis & Christos Zaroliagis

Slides based on those in: http://www.mmds.org
Prime course focus: large scale computing for data analysis & mining

Challenges
- How to distribute computation?
- Distributed/parallel programming is hard

Map-reduce addresses all of the above
- Google’s computational/data manipulation model
- Elegant way to work with big data
Single Node Architecture

Machine Learning, Statistics

“Classical” Data Mining
Motivation: Google Example

- 20+ billion web pages x 20KB = 400+ TB
- 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web
- ~1,000 hard drives to store the web
- Takes even more to do something useful with the data!
- Today, a standard architecture for such problems is emerging
  - Cluster of commodity Linux nodes
  - Commodity network (ethernet) to connect them
Cluster Architecture

2-10 Gbps backbone between racks

1 Gbps between any pair of nodes in a rack

Each rack contains 16-64 nodes

In 2011 it was guestimated that Google had 1M machines, [http://bit.ly/Shh0RO](http://bit.ly/Shh0RO)
Large-scale computing for data mining/analysis problems on commodity hardware

Challenges:

- How do you distribute computation?
- How can we make it easy to write distributed programs?
- Machines fail
  - One server may stay up 3 years (1,000 days)
  - If you have 1,000 servers, expect to lose 1/day
  - People estimated Google had ~1M machines in 2011
    - 1,000 machines fail every day!
Idea and Solution

- **Issue:** Copying data over a network takes time
- **Idea**
  - Bring computation close to the data
  - Store files multiple times for reliability
- **Map-reduce addresses these problems**
  - Google’s computational/data manipulation model
  - Elegant way to work with big data
  - **Storage Infrastructure – File system**
    - Google: GFS. Hadoop: HDFS
  - **Programming model**
    - Map-Reduce
Problem
- If nodes fail, how to store data persistently?

Answer
- Distributed File System
  - Provides global file namespace
  - Google GFS; Hadoop HDFS;

Typical usage pattern
- Huge files (100s of GB to TB)
- Data is rarely updated in place
- Reads and appends are common
Distributed File System

- **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 Node in Hadoop’s HDFS
  - Stores metadata about where files are stored
  - Might be replicated

- **Client library for file access**
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data
Reliable distributed file system
Data kept in “chunks” spread across machines
Each chunk replicated on different machines
  - Seamless recovery from disk or machine failure

Bring computation directly to the data!
Chunk servers also serve as compute servers
Warm-up task
- We have a huge text document
- Count the number of times each distinct word appears in the file

Sample application
- Analyze web server logs to find popular URLs
Task: Word Count

Case 1:
- File too large for memory, but all <word, count> pairs fit in memory

Case 2:
- Count occurrences of words:
  - `words(doc.txt) | sort | uniq -c`
    - where `words` takes a file and outputs the words in it, one per a line
- Case 2 captures the essence of MapReduce
  - Great thing is that it is naturally parallelizable
MapReduce: Overview

- Sequentially read a lot of data
- **Map:**
  - Extract something you care about
- **Group by key:** Sort and Shuffle
- **Reduce:**
  - Aggregate, summarize, filter or transform
- Write the result

Outline stays the same, **Map** and **Reduce** change to fit the problem
MapReduce: The Map Step

Input key-value pairs

Intermediate key-value pairs

…

…
**MapReduce: The Reduce Step**

Intermediate key-value pairs

Key-value groups

Output key-value pairs

Group by key

reduce

reduce

...
More Specifically

- **Input:** a set of key-value pairs
- Programmer specifies two methods:
  - **Map**(*k, v*) → <*k’, v’>*
    - Takes a key-value pair and outputs a set of key-value pairs
      - E.g., key is the filename, value is a single line in the file
      - There is one Map call for every (*k,v*) pair
  - **Reduce**(*k’, <v’>*) → <*k’, v”>*
    - All values *v’* with same key *k’* are reduced together and processed in *v’* order
    - There is one Reduce function call per unique key *k’*
### MapReduce: Word Counting

<table>
<thead>
<tr>
<th>Provided by the programmer</th>
<th>Provided by the programmer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MAP:</strong> Read input and produces a set of key-value pairs</td>
<td><strong>Group by key:</strong> Collect all pairs with same key</td>
</tr>
<tr>
<td><strong>Reduce:</strong> Collect all values belonging to the key and output</td>
<td></td>
</tr>
</tbody>
</table>

### Big document

<table>
<thead>
<tr>
<th>(key, value)</th>
<th>(key, value)</th>
<th>(key, value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(The, 1)</td>
<td>(crew, 1)</td>
<td>(crew, 1)</td>
</tr>
<tr>
<td>(of, 1)</td>
<td>(crew, 1)</td>
<td>(space, 1)</td>
</tr>
<tr>
<td>(the, 1)</td>
<td>(space, 1)</td>
<td>(the, 1)</td>
</tr>
<tr>
<td>(space, 1)</td>
<td>(the, 1)</td>
<td>(the, 1)</td>
</tr>
<tr>
<td>(shuttle, 1)</td>
<td>(Endeavor, 1)</td>
<td>(shuttle, 1)</td>
</tr>
<tr>
<td>(Endeavor, 1)</td>
<td>(recently, 1)</td>
<td>(recently, 1)</td>
</tr>
<tr>
<td>(recently, 1)</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

**The work we're doing now -- the robotics we're doing -- is what we're going to need ...**

**The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/machine partnership. "The work we're doing now -- the robotics we're doing -- is what we're going to need ..."**
map(key, value):
// key: document name; value: text of the document
   for each word w in value:
      emit(w, 1)

reduce(key, values):
// key: a word; values: list of counts (integers)
   result = 0
   for each count v in values:
      result += v
   emit(key, result)
Map-Reduce: Environment

Map-Reduce environment takes care of

- Partitioning the input data
- Scheduling the program’s execution across a set of machines
- Performing the group by key step
- Handling machine failures
- Managing required inter-machine communication
Map-Reduce: A diagram

**MAP:**
Read input and produces a set of key-value pairs

**Group by key:**
Collect all pairs with the same key (Hash merge, Shuffle, Sort, Partition)

**Reduce:**
Collect all values belonging to the key and output
Map-Reduce

- Programmer specifies:
  - Map and Reduce and input files
- Workflow:
  - Read inputs as a set of key-value-pairs
  - Map transforms input kv-pairs into a new set of k'v'-pairs
  - Sorts & Shuffles the k'v'-pairs to output nodes
  - All k’v’-pairs with a given k’ are sent to the same reduce
  - Reduce processes all k'v'-pairs grouped by key into new k''v''-pairs
  - Write the resulting pairs to files
- All phases are distributed with many tasks doing the work
Input and final output are stored on a distributed file system (FS):
- Scheduler tries to schedule map tasks “close” to physical storage location of input data

Intermediate results are stored on local FS of Map and Reduce workers

Output is often input to another MapReduce task
All phases are distributed with many tasks doing the work
- **Master node takes care of coordination**
  - **Task status:** (idle, in-progress, completed)
  - **Idle tasks** get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its $R$ intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master pings workers periodically to detect failures
- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- **Master failure**
  - MapReduce task is aborted and client is notified
How many Map and Reduce jobs?

- $M$ map tasks, $R$ reduce tasks
- **Rule of a thumb:**
  - Make $M >>$ the number of nodes in the cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds up recovery from worker failures
- **Usually** $R << M$
  - Because output is spread across $R$ files
- **Fine granularity tasks:** map tasks >> machines
  - Minimizes time for fault recovery
  - Can do pipeline shuffling with map execution
  - Better dynamic load balancing

<table>
<thead>
<tr>
<th>Process</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Program</td>
<td>MapReduce()</td>
</tr>
<tr>
<td>Master</td>
<td>... wait ...</td>
</tr>
<tr>
<td>Worker 1</td>
<td>Assign tasks to worker machines...</td>
</tr>
<tr>
<td>Worker 2</td>
<td>Map 1</td>
</tr>
<tr>
<td>Worker 3</td>
<td>Read 1.1</td>
</tr>
<tr>
<td>Worker 4</td>
<td>Read 2.1</td>
</tr>
</tbody>
</table>
Refinements: Backup Tasks

- **Problem**
  - Slow workers significantly lengthen the job completion time:
    - Other jobs on the machine
    - Bad disks
    - Weird things

- **Solution**
  - Near end of phase, spawn backup copies of tasks
    - Whichever one finishes first “wins”

- **Effect**
  - Dramatically shortens job completion time
Often a Map task will produce many pairs of the form \((k, v_1), (k, v_2), \ldots\) for the same key \(k\)
- E.g., popular words in the word count example

Can save network time by pre-aggregating values in the mapper:
- \(\text{combine}(k, \text{list}(v_1)) \rightarrow v_2\)
- Combiner is usually same as the reduce function

Works only if reduce function is commutative and associative
Back to our word counting example:

- Combiner combines the values of all keys of a single mapper (single machine):

  - Much less data needs to be copied and shuffled!
Want to control how keys get partitioned

- Inputs to map tasks are created by contiguous splits of input file
- Reduce needs to ensure that records with the same intermediate key end up at the same worker

System uses a default partition function

- hash(key) mod $R$

Sometimes useful to override the hash function:

- E.g., hash(hostname(URL)) mod $R$ ensures URLs from a host end up in the same output file
Problems Suited for Map-Reduce
Example: Matrix-Vector Multiplication

- $n \times n$ matrix $M$; vector $\mathbf{v}$; compute $\mathbf{x} = M \cdot \mathbf{v}^T$

$$x_i = \sum_{j=1}^{n} m_{ij}v_j$$

- $\mathbf{v}$ fits in main memory of a Map worker
- **Map** task/function: operates on a chunk of $M$, producing pairs $(i, m_{ij} \cdot v_i)$
- **Reduce** task/function: sum of all values associated with $i \rightarrow (i, x_i)$
Example: Matrix-Vector Multiplication

- \( n \times n \) matrix \( M \); vector \( \mathbf{v} \); compute \( \mathbf{x} = M \cdot \mathbf{v}^T \)
- \( \mathbf{v} \) does not fit in main memory of a Map worker
- Split \( M (\mathbf{v}) \) into equal-width (-height) stripes

- \( M \)'s \( i \)-th stripe multiplies only components from \( \mathbf{v} \)'s \( i \)-th stripe
- **Map task**: chunk from an \( M \)'s stripe & corresponding \( \mathbf{v} \)'s stripe
Example: Host size

- Suppose we have a large web corpus
- Look at the metadata file
  - Lines of the form: (URL, size, date, ...)
- For each host, find the total number of bytes
  - That is, the sum of the page sizes for all URLs from that particular host

- Other examples:
  - Link analysis and graph processing
  - Machine Learning algorithms
Statistical machine translation:
- Need to count number of times every 5-word sequence occurs in a large corpus of documents

Very easy with MapReduce:
- Map:
  - Extract (5-word sequence, count) from document
- Reduce:
  - Combine the counts
Example: Join By Map-Reduce

- Compute the natural join $R(A,B) \bowtie S(B,C)$
- $R$ and $S$ are each stored in files
- Tuples are pairs $(a,b)$ or $(b,c)$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$b_2$</td>
</tr>
<tr>
<td>$a_4$</td>
<td>$b_3$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_2$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>$b_3$</td>
<td>$c_3$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_3$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>$a_4$</td>
<td>$c_3$</td>
</tr>
</tbody>
</table>

$R \bowtie S$
Each tuple \((a,b)\) in \(R\) is mapped to \((b,(R,a))\)
key = \(b\), value = \((R,a)\)
- **Note:** “\(R\)” in the value is just a bit that means “this value represents a tuple in \(R\), not \(S\).”

Each tuple \((b,c)\) in \(S\) is mapped to \((b,(S,c))\)
key = \(b\), value = \((S,c)\)

After grouping by keys, each reducer gets a key-list that looks like
\((b, [(R,a_1), (R,a_2),\ldots, (S,c_1), (S,c_2),\ldots])\)
For each pair \((R,a)\) and \((S,c)\) on the list for key \(b\), emit \((a,b,c)\)

- This process can produce a quadratic number of outputs as a function of the list length
- Efficient as long as you don’t have too many tuples with a common shared value.
Map-Reduce Join

- Use a hash function $h$ from B-values to $1...k$
- A Map process turns
  - Each input tuple $R(a,b)$ into key-value pair $(b,(a,R))$
  - Each input tuple $S(b,c)$ into $(b,(c,S))$

- Map processes send each key-value pair with key $b$ to Reduce process $h(b)$
  - Hadoop does this automatically; just tell it what $k$ is.
- Each Reduce process matches all the pairs $(b,(a,R))$ with all $(b,(c,S))$ and outputs $(a,b,c)$
Two-Pass Matrix Multiplication

- Multiply matrix \( M = [m_{ij}] \) by \( N = [n_{jk}] \)
  - **Want**: \( P = [p_{ik}] \), where \( p_{ik} = \sum_j m_{ij} \cdot n_{jk} \)
- View \( M \) as collection of tuples \((i, j, m_{ij})\)
- View \( N \) as collection of tuples \((j, k, n_{ij})\)
- First pass \( \approx \) relational join
  - Computes each \( m_{ij} \cdot n_{jk} \)
- Second pass \( \approx \) group + aggregate operation
  - Computes the sum over \( j \)
- Typically, large relations are *sparse* (mostly 0’s)
- Assume a nonzero \( m_{ij} \) is really a tuple of a relation \((i, j, m_{ij})\); similarly for \( n_{jk} \)
  - 0 elements are not represented at all
The Map and Reduce Functions

- **The Map function**: \((i,j,m_{ij}) \rightarrow \text{key} = j, \text{value} = (M,i,m_{ij}); (j,k,n_{jk}) \rightarrow \text{key} = j, \text{value} = (N,k,n_{jk})\)
  - As for join, \(M\) and \(N\) here are bits indicating which relation the value comes from.

- **The Reduce function**: for key \(j\), pair each \((M,i,m_{ij})\) on its list with each \((N,k,n_{jk})\) and produce key = \((i,k)\), value = \(m_{ij} \cdot n_{jk}\)
The Second Pass

- **The Map function**: The identity function.
- Result is that each key \((i,k)\) is paired with the list of products \(m_{ij} \cdot n_{jk}\) for all \(j\)
- **The Reduce function**: sum all the elements on the list, and produce key = \((i,k)\), value = that sum
  - I.e., each output element \(((i,k),s)\) says that the element \(p_{ik}\) of the product matrix \(P\) is \(s\)
Single-Pass Matrix Multiplication

- We can use a single pass if:
  1. Keys (reducers) correspond to output elements (i,k)
  2. Map sends input elements to more than one reducer

- The Map function: \( m_{ij} \rightarrow \) for all \( k \): key = (i,k), value = (M,j,m_{ij}); \( n_{jk} \rightarrow \) for all \( i \): key = (i,k), value = (N,j,n_{jk})

- The Reduce function: for each \( (M,j,m_{ij}) \) on the list
  for key (i,k) find the \( (N,j,n_{jk}) \) with the same j;
  Multiply \( m_{ij} \) by \( n_{jk} \) and then sum the products
Extensions of MapReduce
MapReduce uses two ranks of tasks: one for Map the second for Reduce
  - Data flows from the first rank to the second

Generalize in two ways:
  1. Allow any number of ranks
  2. Allow functions other than Map and Reduce

As long as data flow is in one direction only, we can have the blocking property and allow recovery of tasks rather than whole jobs
Spark

- The most popular implementation of a dataflow system
- Data passed from one rank of processes to the next forms a *Resilient Distributed Dataset* (RDD)
  - Elements of an RDD are like tuples of a relation
    - Generalizes (key-value) pairs
  - Built-in operations include Map, Reduce, and also group-aggregate using any components of the RDD type.
    - The thing Hadoop does behind-the-scenes
Example: 2-Pass Matrix Multiplication

- The 2-pass MapReduce algorithm had a second Map function that didn’t really do anything.
- We could think of it as a five-rank data-flow algorithm of the form Map-GA-Reduce-GA-Reduce, where the RDD types are:
  1. \((j, (M,i,m_{ij}))\) and \((j, (N,k,n_{jk}))\)
  2. \(j\) with list of \((M,i,m_{ij})\)’s and \((N,k,n_{jk})\)’s
  3. \(((i,k), m_{ij} \cdot n_{jk})\)
  4. \((i,k)\) with list of \(m_{ij} \cdot n_{jk}\)’s
  5. \(((i,k), p_{ik})\)
The Graph Model

- Computation is a recursion on some graph
- Graph nodes send messages to one another
  - Messages bunched into *supersteps*, where each graph node processes all data received
  - Sending individual messages would result in far too much overhead
- **Checkpoint** all compute nodes after some fixed number of supersteps
  - Note blocking property fails to hold
- On failure, roll all tasks back to previous checkpoint
Is this the shortest path from node M I know about? If so ...

I found a path from node M to you of length L

I found a path from node M to you of length L+5

I found a path from node M to you of length L+3

I found a path from node M to you of length L+6
Some Systems

- **Pregel**: the original, from Google
- **Giraph**: open-source (Apache) Pregel
  - Built on Hadoop
- **GraphX**: a similar front end for Spark
- **GraphLab**: similar system that deals more effectively with nodes of high degree
  - Will split the work for such a graph node among several compute nodes
Cost Measure of MapReduce Algorithms
Cost Measures for Algorithms

- In MapReduce we quantify the cost of an algorithm using:
  1. Communication cost = total I/O of all processes
  2. Elapsed communication cost = max of I/O along any path
  3. (Elapsed) computation cost analogous, but count only running time of processes

Note that here the big-O notation is not the most useful (adding more machines is always an option)
Example: Cost Measures

- For a map-reduce algorithm:
  - **Communication cost** = input file size + 2 × (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
  - **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process.
What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
  - Ignore one or the other

- Total cost tells what you pay in rent from your friendly neighborhood cloud

- Elapsed cost is wall-clock time using parallelism
Cost of Map-Reduce Join

- **Total communication cost**
  \[ O(|R| + |S| + |R \bowtie S|) \]
- **Elapsed communication cost** \( = O(s) \)
  - We’re going to pick \( k \) and the number of Map processes so that the I/O limit \( s \) is respected
  - We put a limit \( s \) on the amount of input or output that any one process can have. **\( s \) could be:**
    - What fits in main memory
    - What fits on local disk
- **With proper indexes, computation cost is linear in the input + output size**
  - So computation cost is like comm. cost
Complexity Theory of MapReduce Algorithms
Data consists of records for 3000 drugs
- List of patients taking, dates, diagnoses
- About 1M of data per drug

Problem is to find drug interactions
- Example: two drugs that when taken together increase the risk of heart attack

Must examine each pair of drugs and compare their data
First attempt (initial plan):
- Key = set of two drugs \{i, j\}
- Value = the record for one of these drugs

Given drug \(i\) and its record \(R_i\), the mapper generates all key-value pairs \(\{(i, j), R_j\}\), where \(j\) is any other drug besides \(i\).

Each reducer receives its key and a list of the two records for that pair: \(\{(i, j), [R_i, R_j]\}\)

**Question for thought**: wouldn’t it be better if the value were a pointer to \(R_i\)?
Example: Three Drugs

Mapper for drug 1
{1, 2}
Drug 1 data
{1, 3}
Drug 1 data

Mapper for drug 2
{1, 2}
Drug 2 data
{2, 3}
Drug 2 data

Mapper for drug 3
{1, 3}
Drug 3 data
{2, 3}
Drug 3 data

Reducer for {1,2}
Reducer for {1,3}
Reducer for {2,3}
Example: Three Drugs

Mapper for drug 1
{1, 2} Drug 1 data
{1, 3} Drug 1 data

Mapper for drug 2
{1, 2} Drug 2 data
{2, 3} Drug 2 data

Mapper for drug 3
{1, 3} Drug 3 data
{2, 3} Drug 3 data

Reducer for {1,2}
Reducer for {1,3}
Reducer for {2,3}
Example: Three Drugs

{1, 2} Drug 1 data Drug 2 data Reducer for {1,2}

{1, 3} Drug 1 data Drug 3 data Reducer for {1,3}

{2, 3} Drug 2 data Drug 3 data Reducer for {2,3}
What Went Wrong?

- 3000 drugs
- times 2999 key-value pairs per drug
- times 1,000,000 bytes per key-value pair
- = **9 TB** communicated over a 1Gb Ethernet
- = 90,000 seconds of network use (25 hours)
The Improved Algorithm

- Group the drugs into 30 groups of 100 drugs each
  - Say $G_1 =$ drugs 1-100, $G_2 =$ drugs 101-200,..., $G_{30} =$ drugs 2901-3000
  - Let $g(i) =$ the number of the group into which drug $i$ goes
The Map Function

- A key is a set of two group numbers
- The mapper for drug $i$ produces 29 key-value pairs
  - Each key is the set containing $g(i)$ and one of the other group numbers
  - The value is a pair consisting of the drug number $i$ and the megabyte-long record for drug $i$
The reducer for pair of groups \( \{m, n\} \) gets that key and a list of 200 drug records – the drugs belonging to groups \( m \) and \( n \)

Its job is to compare each record from group \( m \) with each record from group \( n \)

- **Special case**: also compare records in group \( n \) with each other, if \( m = n+1 \) or if \( n = 30 \) and \( m = 1 \).

Notice each pair of records is compared at exactly one reducer, so the total computation is not increased.
The big difference is in the communication requirement
Now, each of 3000 drugs’ 1MB records is replicated 29 times
Communication cost = 87 GB << 9 TB
A Model for Map-Reduce Problems

1. A set of *inputs*
   - Example: the drug records

2. A set of *outputs*
   - Example: one output for each pair of drugs, telling whether a statistically significant interaction was detected

3. A many-many relationship between each output and the inputs needed to compute it
   - Example: The output for the pair of drugs \{i, j\} is related to inputs i and j
Example: Drug Inputs/Outputs

Drug 1

Drug 2

Drug 3

Drug 4

Output 1-2

Output 1-3

Output 1-4

Output 2-3

Output 2-4

Output 3-4
Example: Matrix Multiplication
Reducer Size

- **Reducer size**, denoted $q$: maximum number of inputs that a given reducer can have
  - I.e., the length of the value list.
- Limit might be based on how many inputs can be handled in main memory
- Or: make $q$ low to force lots of parallelism
The average number of key-value pairs created by each mapper is the *replication rate*

- Denoted $r$
- Represents the communication cost per input
Example: Drug Interaction

- Suppose we use $g$ groups and $d$ drugs
- A reducer needs two groups, so $q = 2d/g$
- Each of the $d$ inputs is sent to $g-1$ reducers, or approximately $r = g$
- Replace $g$ by $r$ in $q = 2d/g$ to get $r = 2d/q$

Tradeoff!
The bigger the reducers, the less communication
Upper and Lower Bounds on $r$

- What we did gives an upper bound on $r$ as a function of $q$
- A solid investigation of MapReduce algorithms for a problem includes lower bounds
  - Proofs that you cannot have lower $r$ for a given $q$
A *mapping schema* for a problem and a reducer size $q$ is an abstraction of a MR algorithm.

It assigns inputs to sets of reducers, with two conditions:

1. No reducer is assigned more than $q$ inputs.
2. For every output, there is some reducer that receives all of the inputs associated with that output.
   - Say the reducer *covers* the output.
   - If some output is not covered, we can’t compute that output.
Every MapReduce algorithm has a mapping schema

The requirement that there be a mapping schema is what distinguishes MapReduce algorithms from general parallel algorithms
Example: Drug Interactions

- d drugs, reducer size q
- Each drug has to meet each of the d-1 other drugs at some reducer
- If a drug is sent to a reducer, then at most q-1 other drugs are there
- Thus, each drug is sent to at least \(\lceil (d-1)/(q-1) \rceil\) reducers, and \(r \geq \lceil (d-1)/(q-1) \rceil\)
  - Or approximately \(r \geq \lceil d/q \rceil\)
- Half the \(r\) from the algorithm we described
- Better algorithm gives \(r \leq d/q + 1\), so lower bound is actually tight
Pointers and Further Reading
Implementations

- **Google**
  - Not available outside Google

- **Hadoop**
  - An open-source implementation in Java
  - Uses HDFS for stable storage

- **Aster Data**
  - Cluster-optimized SQL Database that also implements MapReduce
Cloud Computing

- Ability to rent computing by the hour
  - Additional services e.g., persistent storage
- Amazon’s “Elastic Compute Cloud” (EC2)
- Aster Data and Hadoop can both be run on EC2
Reading

- Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: The Google File System
Resources

- Hadoop Wiki
  - Introduction
  - Getting Started
  - Map/Reduce Overview
    - [http://wiki.apache.org/lucene-hadoop/HadoopMapReduce](http://wiki.apache.org/lucene-hadoop/HadoopMapReduce)
    - [http://wiki.apache.org/lucene-hadoop/HadoopMapRedClasses](http://wiki.apache.org/lucene-hadoop/HadoopMapRedClasses)
  - Eclipse Environment
- Javadoc
  - [http://lucene.apache.org/hadoop/docs/api/](http://lucene.apache.org/hadoop/docs/api/)
- Releases from Apache download mirrors
  - http://www.apache.org/dyn/closer.cgi/lucene/hadoop/
- Nightly builds of source
- Source code from subversion