CMSC5741 Big Data Tech. & Apps.

# Lecture 2: MapReduce and Frequent Itemsets

Prof. Michael R. Lyu
Computer Science & Engineering Dept.
The Chinese University of Hong Kong

## Outline

- Introduction
- The Hadoop Distributed File System (HDFS)
- MapReduce
- Hadoop
- Hadoop Streaming
- Problems Suited for MapReduce
- TensorFlow
- Frequent Itemsets
- Conclusion

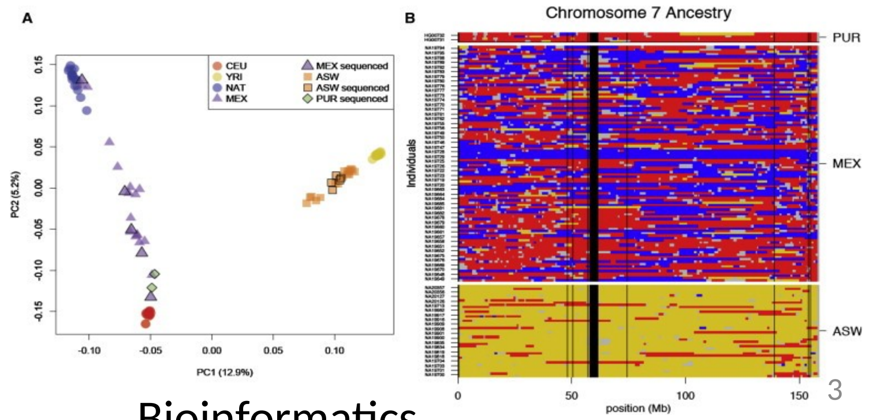
# Introduction

- Much of the course will be devoted to learning with big data

Facebook

Twitter

Netflix



Bioinformatics

# Introduction

- Challenges:
  - How to distribute computation?
  - Distributed/parallel programming is hard
- MapReduce addresses all of the above
  - Google's computational/data manipulation model
  - Elegant way to work with Big Data

## Motivation: Data Volume Now

The scale of data today and tomorrow:

- 2008: Google processes 20 PB a day
- 2009: Facebook has 2.5 PB user data + 15 TB/day
- 2009: eBay has 6.5 PB user data + 50 TB/day
- 2013: Estimated size of digital world is 4.4 ZB
- 2016: 2.5 exabytes (EB) created everyday
- 2017: Google holds 10-15 exabytes of data
- By 2020: 44ZB ($10^{21}$) will be produced (5.2 TB for every person)

# 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 (mainly text data)
- Takes even more resources to do useful work with the data
- Today, a standard architecture for such problem is emerging:
  - Cluster of commodity Linux nodes
  - Commodity network (ethernet) to connect them
  - It was estimated that Google had over 2.5M machines in 16 data centers worldwide (one center includes 9,941 miles of cable)

# Large-scale Computing

- Large-scale computing for data mining problems on commodity hardware
- Challenges:

1. How do you distribute computation?

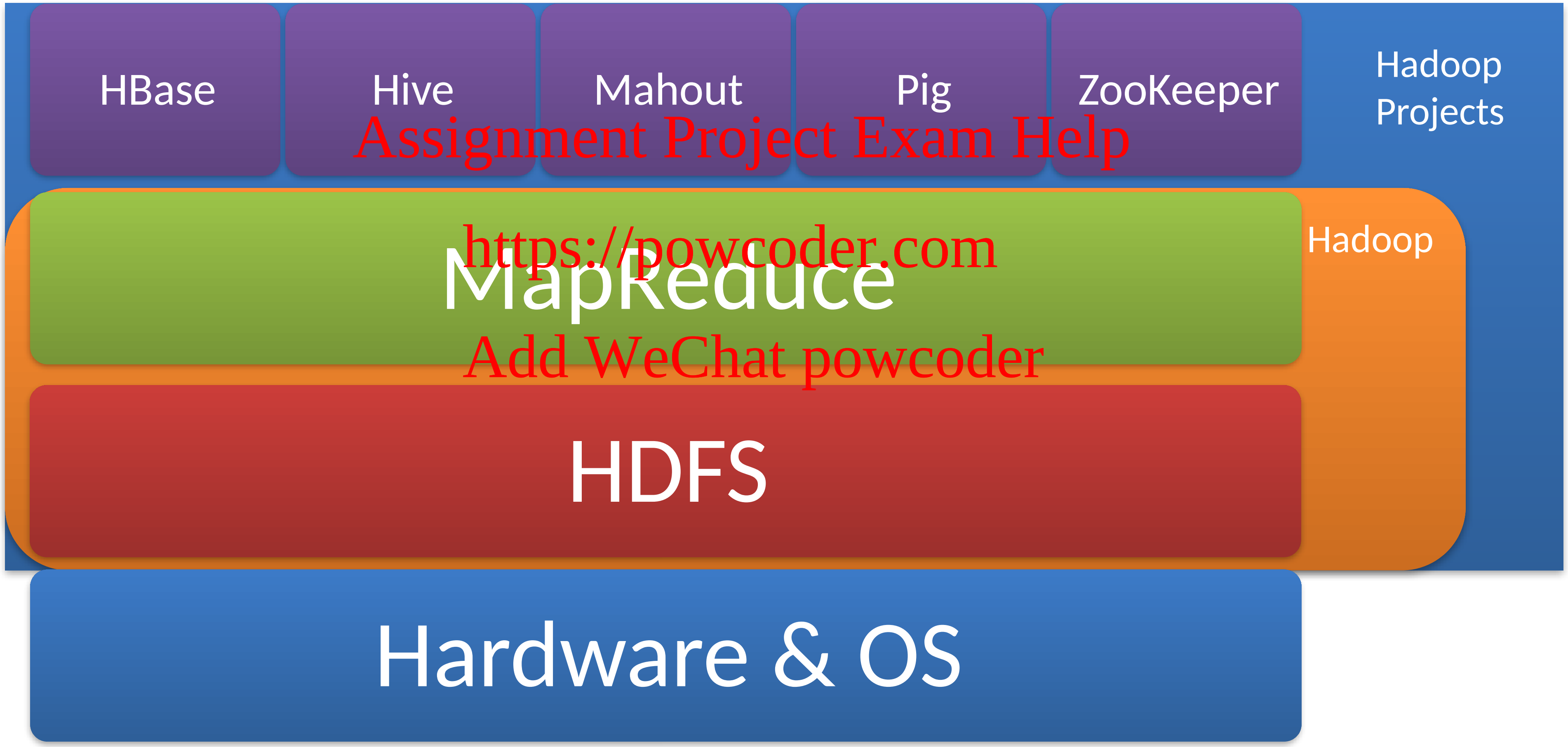
2. How can we make it easy to write distributed programs?

3. How can you handle machine failures?
   - One server may stay up 3 years (1,000 days)
   - If you have 1,000 servers, expect to lose 1/day
   - It is estimated that Google had 2.5M machines in 2016
     – 2,500 machine fails 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
- MapReduce 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

# Relationship



## Outline

- Introduction
- The Hadoop Distributed File System (HDFS)
- MapReduce
- Hadoop
- Hadoop Streaming
- Problems Suited for MapReduce
- TensorFlow
- Frequent Itemsets
- Conclusion

## The Hadoop Distributed File System (HDFS)

- With hundreds of machines at hand, failure is the norm rather than exception
- Traditional file storage system cannot cope with the scale and failure faced by large clusters
- The Hadoop Distributed File System (HDFS) is a natural solution to this problem
  - Distributed File System
  - Provides global file namespace
  - Replica to ensure data recovery

## The Hadoop Distributed File System (HDFS)

- A HDFS instance may consist of thousands of server machines, each storing part of the file system's data.
- Since we have huge number of components, and each component has non-trivial probability of failure, it means that there is always some component that is non-functional.
- Detection of faults and quick, automatic recovery from them are a core architectural goal of HDFS.

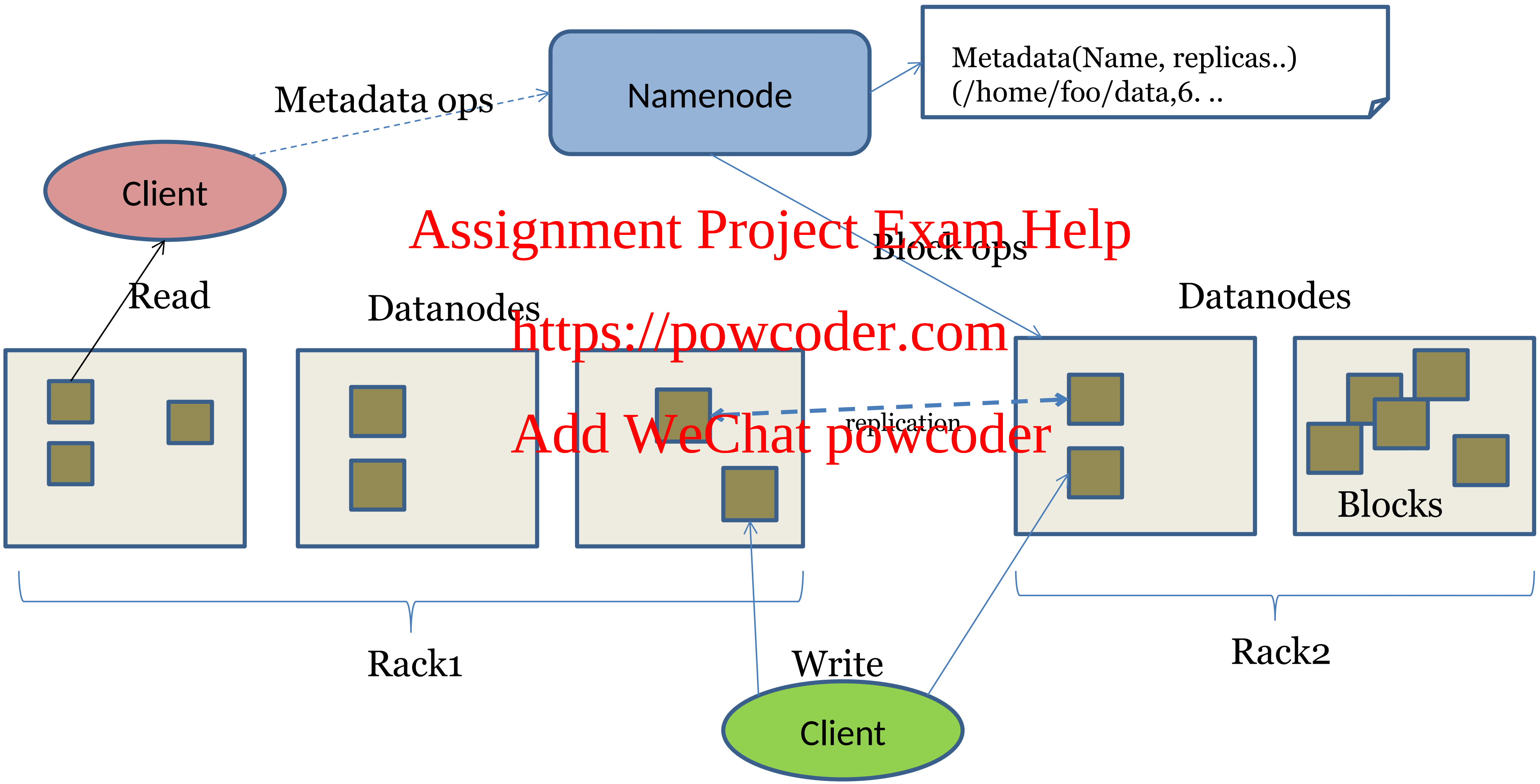
# Data Characteristics

- Streaming data access
- Batch processing rather than interactive user access
- Write-once-read-many: a file, once created, written and closed, need not be changed
- This assumption simplifies coherency in concurrent accesses

# HDFS Architecture

- Master/slave architecture
  - Master: NameNode
  - Slave: DataNode
- HDFS exposes a file system namespace (NameNode) and allows user data to be stored in files.
- A file is split into one or more blocks and set of blocks are stored in DataNodes.

# HDFS Architecture



## File System Namespace

- Namenode maintains the file system.
  - Hierarchical file system with directories and files.
  - Create, remove, move, rename, etc.
  - Any meta information changes to the file system is recorded by the Namenode.
- An application can specify the number of replicas of the file needed: replication factor of the file. This information is stored in the Namenode.

# Data Replication

- HDFS is designed to store very large files across machines in a large cluster.
  - Each file is a sequence of blocks.
  - All blocks in the file except the last are of the same size.
- Blocks are replicated for fault tolerance.
- Block size and replicas are configurable per file.
- The NameNode receives a Heartbeat and a BlockReport from each DataNode in the cluster.
- BlockReport contains all the blocks on a DataNode.

## Replica Selection

- Replica selection for read operation: HDFS tries to minimize the bandwidth consumption and latency.
- If there is a replica on the Reader node then that is preferred.
- HDFS cluster may span multiple data centers: replica in the local data center is preferred over the remote one.

## Safemode Startup

- Each DataNode checks in with Heartbeat and BlockReport.
- NameNode verifies that each block has acceptable number of replicas.
- After a configurable percentage of safely replicated blocks check in with the NameNode, NameNode exits Safemode.
- It then makes the list of blocks that need to be replicated.
- NameNode then proceeds to replicate these blocks to other DataNodes.

# Filesystem Metadata

- The HDFS namespace is stored by NameNode.
- NameNode uses a transaction log called the EditLog to record every change that occurs to the filesystem meta data.
  - For example, creating a new file
  - Change replication factor of a file
  - EditLog is stored in the NameNode's local filesystem

## NameNode

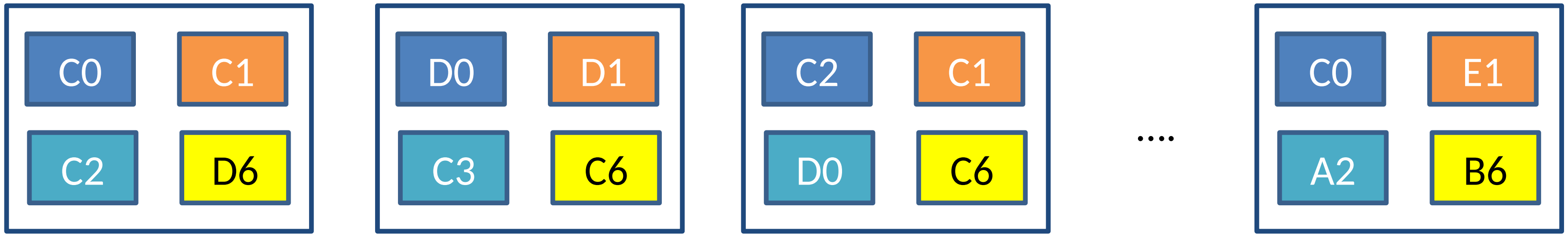
- Keeps image of entire file system namespace.
- When the Namenode starts up
  - Gets the FsImage and Editlog.
  - Update FsImage with EditLog information.
  - Stores a copy of the FsImage as a checkpoint.
- In case of crash
  - Last checkpoint is recovered.

# DataNode

- A DataNode stores data in files in its local file system.
  - Each block of HDFS is a separate file.
  - These files are placed in different directories.
  - Creation of new directory is determined by heuristics.
- When the filesystem starts up:
  - Generates Blockreport.
  - Sends this report to NameNode.

# HDFS Summary

- Reliable distributed file system
- Data kept in "chunks" spread across machines
- Each chunk replicated on different machine and racks
  - Seamless recovery from disk or machine failure



# Outline

- Introduction
- The Hadoop Distributed File System (HDFS)
- MapReduce
- Hadoop
- Hadoop Streaming
- Problems Suited for MapReduce
- TensorFlow
- Frequent Itemsets
- Conclusion

# MapReduce

- 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

- Using Unix tool chain, we can count the occurrences of words:
  - `words (doc.txt) | sort | uniq -c`
    - Where `words` takes a file and outputs the words in it, one per line
- This way of counting captures the essence of MapReduce
  - Mapper (done by words)
  - Group by keys and sort (done by sort)
  - Reducer (done by uniq)
  - Hadoop handles the partition and parallelization

## MapReduce: Overview

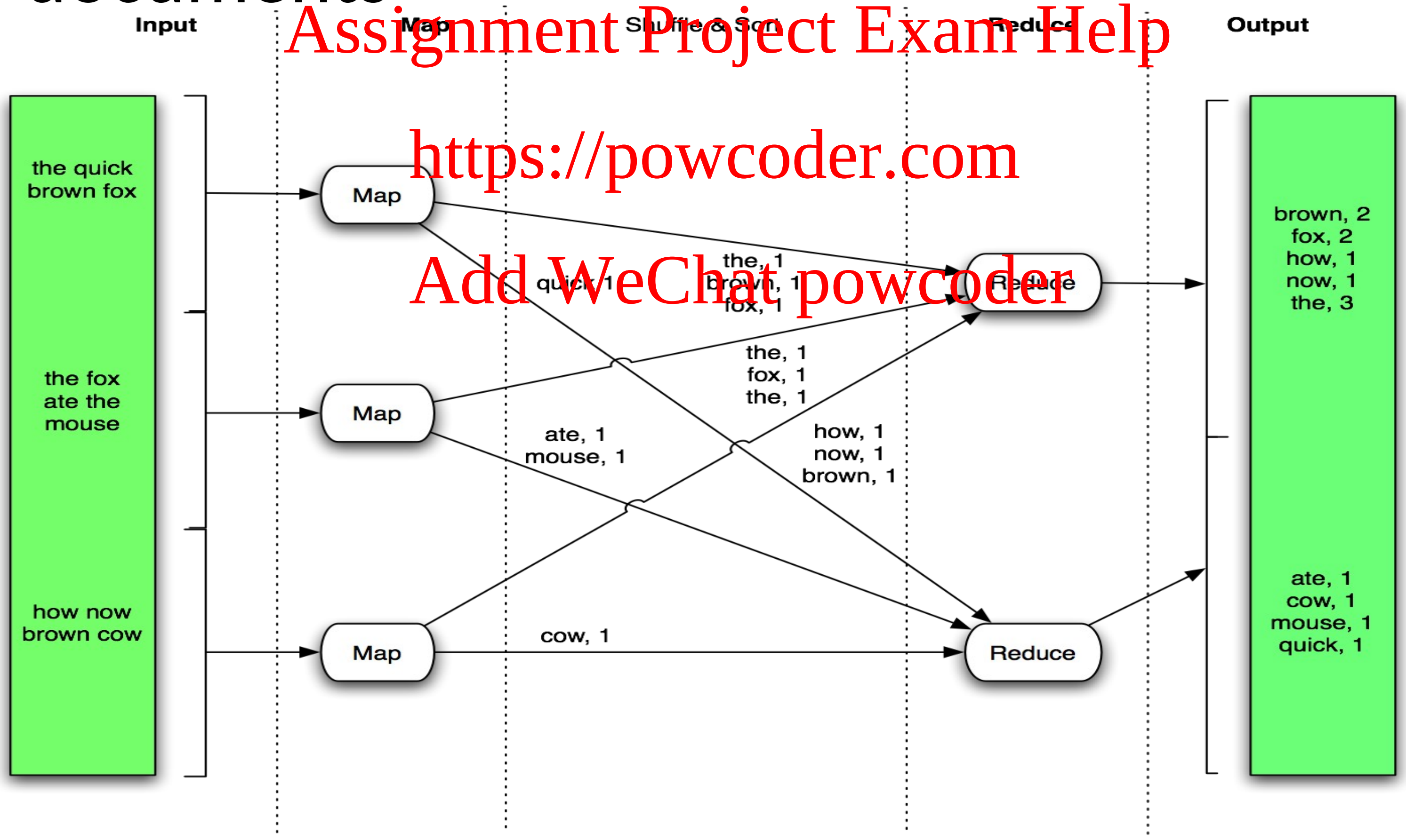
- Sequentially read a lot of data
- Map:
  - Extract something you care about
- Group by key: Sort and Shuffle
- Reduce:
  - Aggregate, summarize, filter or transform
- Write the result

# MapReduce

- Input: a set of key-value paris
- Programmer must specifies two methods:
  - Map(k,v) -> <k’, v’>
    - Takes a key-value pair and outputs a set of key-value pairs
    - There is one Map call for every (k,v) pair
  - Reduce (k’, <v’>) -> <k’, v’’>
    - All values v’ with the same key k’ are reduced together and processed in v’ order
    - There is one Reduce function call per unique key k’

# MapReduce: Word Count Example

- Now that one document changes to a large corpus of documents



# MapReduce: Word Count Example

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

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

## In-class Practice

- Go to practice

## MapReduce: Environment

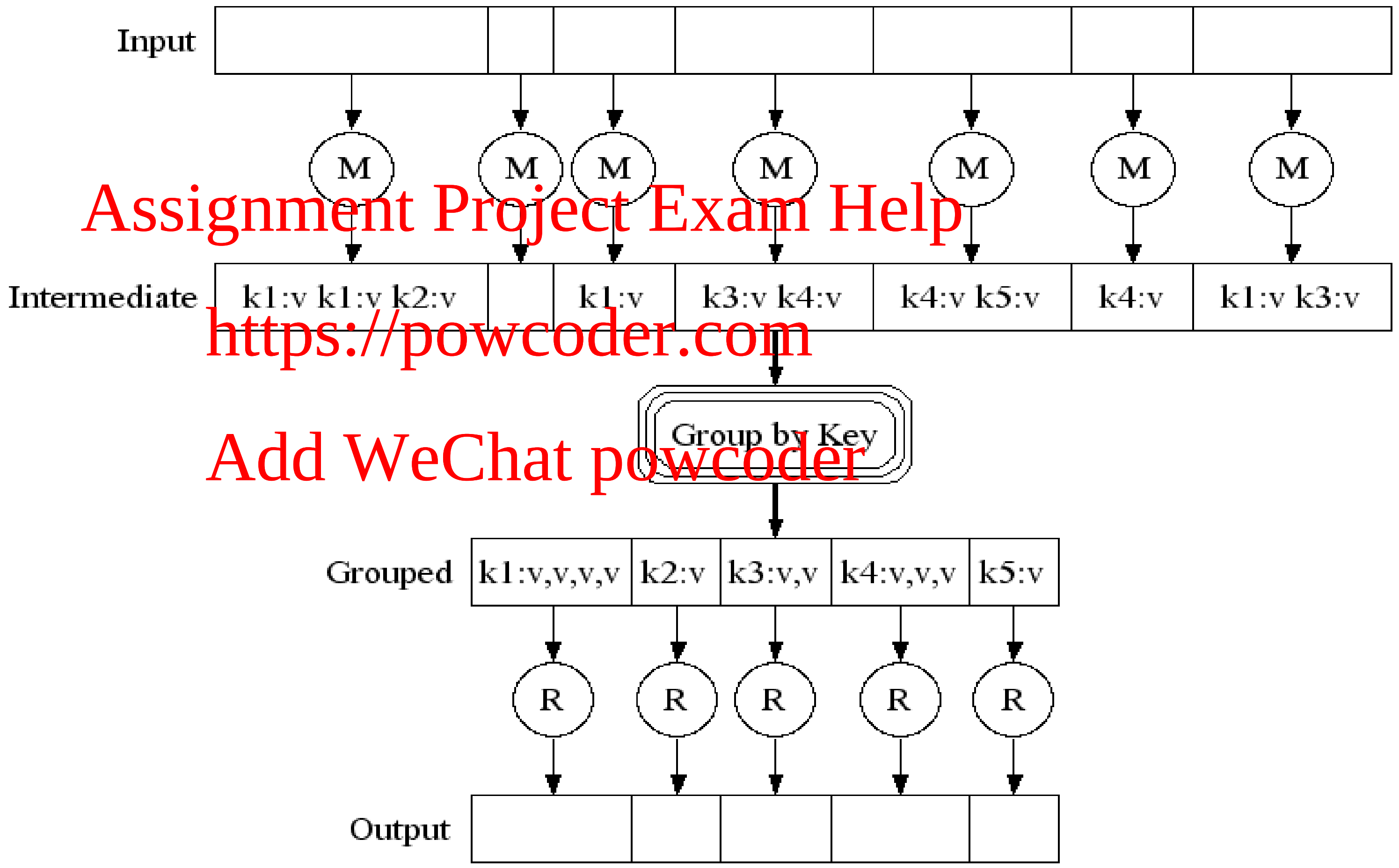
- MapReduce environment takes care of:
  - Partitioning the input data
  - Scheduling the program’s execution across a set of machines
  - Performing the “group by key” step
  - Handling machine failures
  - Managing required inter-machine communication

# MapReduce

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

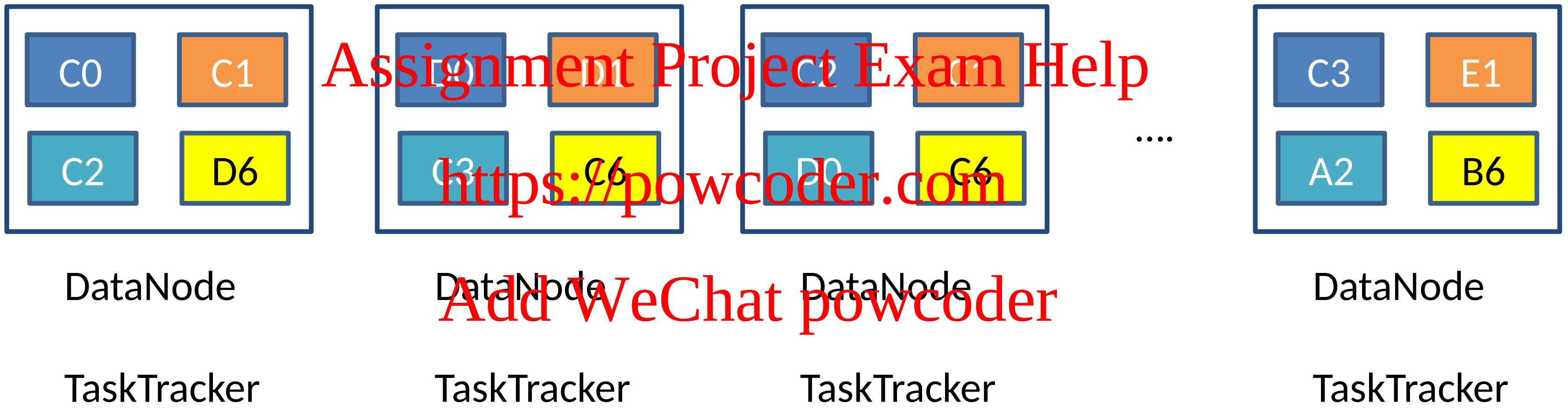
Group by key:
Collect all pairs with the same key

Reduce:
Collect all values belonging to the key and output



# MapReduce

- Move computation to the data



Bring computation directly to the data!

DataNode also serve as compute servers

## Data Flow

- 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

# Coordination: Master

- 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

## Dealing with 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 much larger than the number of nodes in the cluster
  - One chunk per map is common
  - Improves dynamic load balancing and speeds up recovery from worker failures
- Usually R is smaller than M
  - Output is spread across R files

## Refinement: 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 the word count example
- Can save network time by pre-aggregating values in the mapper:
  - Combine(k, list(v)) -> v2
  - Combiner is usually the same as the reduce function
- Works only if reduce function is commutative and associative

# Refinement: Partition Function

- 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

## Outline

- Introduction
- The Hadoop Distributed File System (HDFS)
- MapReduce
- Hadoop
- Hadoop Streaming
- Problems Suited for MapReduce
- TensorFlow
- Frequent Itemsets
- Conclusion

## Hadoop

- Hadoop is an open source implementation of MapReduce framework
  - Hadoop Distributed File System (HDFS) as storage
  - Hadoop handles the task split, task distribution, task monitoring and failure recovery
  - All you need to do is to write two Java classes
    - Mapper
    - Reducer

# Hadoop

- Follow the MapReduce architecture, the Hadoop has a master/slave design

| | master | slave |
|---|---|---|
| MapReduce | jobtracker | tasktracker |
| HDFS | namenode | datanode |

## Word Count in Hadoop

- Mapper

```
public static class MapClass extends MapReduceBase
implements Mapper {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(WritableComparable key, Writable value,
  OutputCollector output, Reporter reporter)
  throws IOException {
    String line = ((Text)value).toString();
    StringTokenizer itr = new StringTokenizer(line);
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      output.collect(word, one);
    }
  }
}
```

## Word Count in Hadoop

- Reducer

```
public static class Reduce extends MapReduceBase implements
Reducer {
  public void reduce(WritableComparable key, Iterator
  values, OutputCollector output, Reporter reporter)
  throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += ((IntWritable) values.next()).get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```

## Word Count in Hadoop

- Main

```
public static void main(String[] args) throws IOException {
  //checking goes here
  JobConf conf = new JobConf();

  conf.setOutputKeyClass(Text.class);
  conf.setOutputValueClass(IntWritable.class);

  conf.setMapperClass(MapClass.class);
  conf.setCombinerClass(Reduce.class);
  conf.setReducerClass(Reduce.class);

  conf.setInputPath(new Path(args[0]));
  conf.setOutputPath(new Path(args[1]));

  JobClient.runJob(conf);
}
```

# Outline

- Introduction
- The Hadoop Distributed File System (HDFS)
- MapReduce
- Hadoop
- Hadoop Streaming
- Problems Suited for MapReduce
- TensorFlow
- Frequent Itemsets
- Conclusion

## Hadoop Streaming

- To enjoy the convenience brought by Hadoop, one has to implement mapper and reducer in Java
  - Hadoop defines a lot of data types and complex class hierarchy
  - There is a learning curve
- Hadoop streaming allows you to use any language to write the mapper and reducer

## Hadoop Streaming

- Using Hadoop Streaming, you need to write
  - Mapper
    - Read input from standard input (STDIN)
    - Write map result to standard output (STDOUT)
      - Key value are separated using tab
  - Group by key
    - Done by Hadoop
  - Reducer
    - Read input (Mapper's output) from standard input (STDIN)
    - Write output (Final result) to standard output (STDOUT)

## Hadoop Streaming

- Allows you to start writing MapReduce application that can be readily deployed without having to learn Hadoop class structure and data types
- Speed up development
- Utilize rich features and handy libraries from other languages (Python, Ruby)
- Efficiency critical application can be implemented in efficient language (C, C++)

## Hadoop Streaming: Word Count Mapper

```
#!/usr/bin/env python

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

## Hadoop Streaming: Word Count Reducer

```
#!/usr/bin/env python
from operator import itemgetter
import sys
current_word = None
current_count = 0
word = None
for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    try:
        count = int(count)
    except ValueError:
        continue
    if current_word == word:
        current_count += count
    else:
        if current_word:
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

## Hadoop Streaming: How to Run?

- To run the sample code

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar \
-input inputPathonHDFS \
-output outputPathonHDFS \
-file pathToMapper.py \
-mapper mapper.py \
-file pathToReducer.py \
-reducer reducer.py
```

- -file caches the argument to every tasktracker
- The above command distribute the mapper.py and reducer.py to every tasktracker

# Hadoop Streaming: Word Count

```
#!/usr/bin/env python
"""A more advanced Mapper, using Python iterators and generators."""

import sys
def read_input(file):
    for line in file:
        yield line.split()
def main(separator='\t'):
    # input comes from STDIN (standard input)
    data = read_input(sys.stdin)
    for words in data:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        for word in words:
            print '%s%s%d' % (word, separator, 1)
if __name__ == "__main__":
    main()
```

# Hadoop Streaming: Word Count

```
#!/usr/bin/env python
"""A more advanced Reducer, using Python iterators and generators."""

from itertools import groupby
from operator import itemgetter
import sys
def read_mapper_output(file, separator='\t'):
    for line in file:
        yield line.rstrip().split(separator, 1)
def main(separator='\t'):
    # input comes from STDIN (standard input)
    data = read_mapper_output(sys.stdin, separator=separator)
    # groupby groups multiple word-count pairs by word,
    # and creates an iterator that returns consecutive keys and their group:
    #   current_word - string containing a word (the key)
    #   group - iterator yielding all ["&lt;current_word&gt;", "&lt;count&gt;"] items
    for current_word, group in groupby(data, itemgetter(0)):
        try:
            total_count = sum(int(count) for current_word, count in group)
            print "%s%s%d" % (current_word, separator, total_count)
        except ValueError:
            # count was not a number, so silently discard this item
            pass
if __name__ == "__main__":
    main()
```

## Demo

- Given a list of academic paper authors and their papers, we try to output:
  - The most used non-trivial words in the title for each author
- We use a python based MapReduce framework implementation called mincemeat.

## Demo

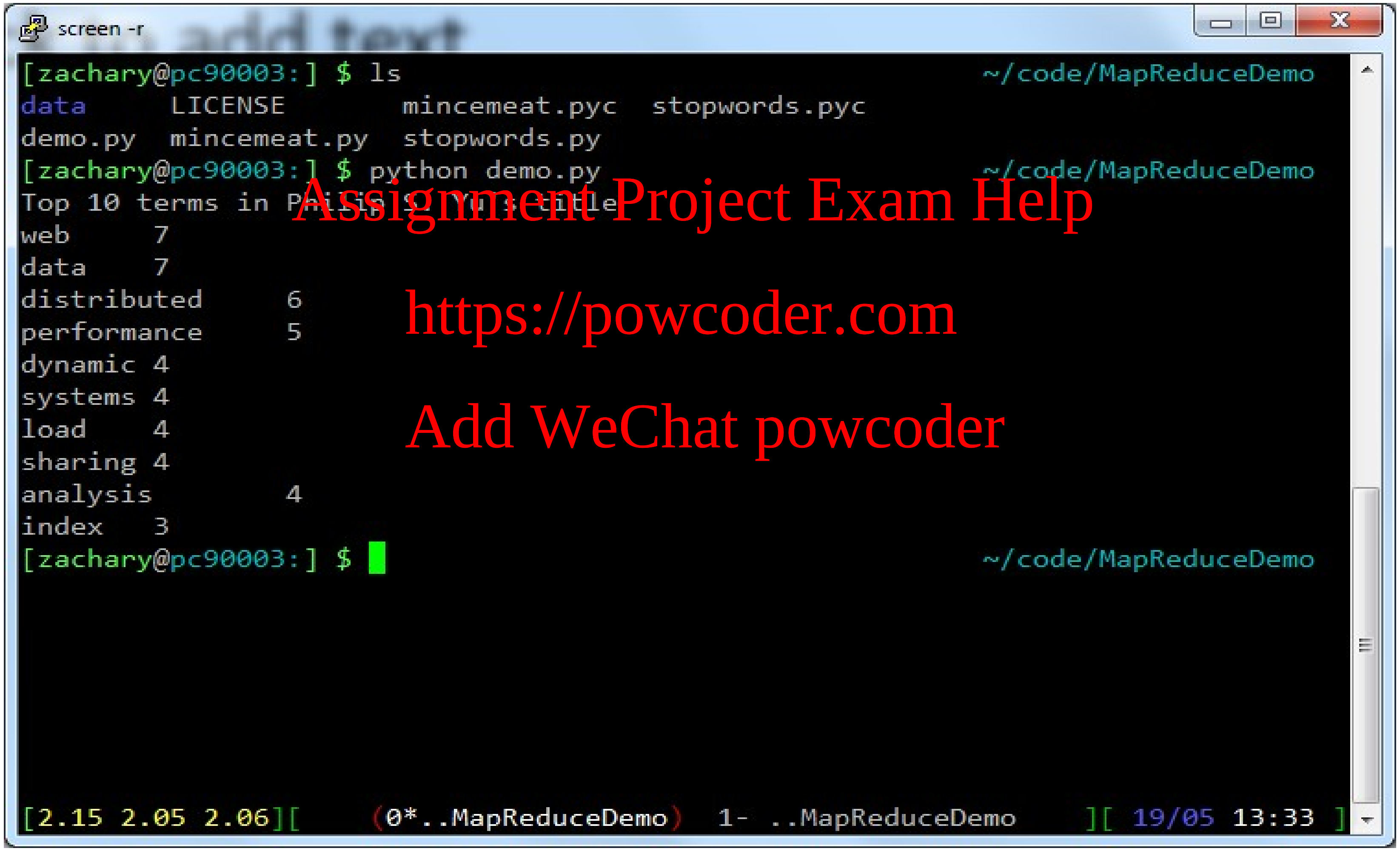
- Input Data
  - Books, Ph.D. Thesis, web pages, academic papers
  - Input format
    - Publication type
    - Affiliation
    - Abbreviation code
    - Authors
    - Title

```
tr/gte/TM-0014-06-88-165:::Frank Manola:::Distributed Object
Management Technology.
tr/ibm/IWBS94:::Christoph Beierle::Udo Pletat:::The Algebra of
Feature Graph Specifications
```

## Demo

- To run the demo:
  - In terminal 1, type:
    - `python demo.py`
    - This set the mapper, reducer and start the main program
  - In terminal 2, type:
    - python mincemeat.py -p changeme 127.0.0.1
    - “changeme” is the authentication password
    - 127.0.0.1 is the server address
    - This starts the Map-Reduce framework

# Demo



```
[zachary@pc90003:] $ ls                                          ~/code/MapReduceDemo
data      LICENSE       mincemeat.pyc  stopwords.pyc
demo.py   mincemeat.py  stopwords.py
[zachary@pc90003:] $ python demo.py                              ~/code/MapReduceDemo
Top 10 terms in Papers' titles
web      7
data     7
distributed      6
performance      5
dynamic  4
systems  4
load     4
sharing  4
analysis         4
index    3
[zachary@pc90003:] $                                             ~/code/MapReduceDemo




[2.15 2.05 2.06][    (0*..MapReduceDemo)  1- ..MapReduceDemo    ][ 19/05 13:33 ]
```

# Outline

- Introduction
- The Hadoop Distributed File System (HDFS)
- MapReduce
- Hadoop
- Hadoop Streaming
- Problems Suited for MapReduce
- TensorFlow
- Frequent Itemsets
- Conclusion

## 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

# Example: Language Model

- 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 MapReduce

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

| A | B |
|---|---|
| a1 | b1 |
| a2 | b1 |
| a3 | b2 |
| a4 | b3 |

X

| B | C |
|---|---|
| b2 | c1 |
| b2 | c2 |
| b3 | c3 |

=

| A | C |
|---|---|
| a3 | c1 |
| a3 | c2 |
| a4 | c3 |

Note – Other relational-algebra operations: Selection, Projection, Union/Interaction/Difference, Grouping/Aggregation

# MapReduce Join

- Use a hash function from B-values to
- A Map process turns:
  - Each input tuple into key-value pair
  - Each input tuple into
- Map processes send each key-value pair with key to Reduce process
  - Hadoop does this automatically; just tell it what is
- Each Reduce process matches all the pairs with all and outputs .

# Cost Measures for Algorithms

- In MapReduce we quantify the cost of an algorithm using
  - Communication cost
    - total I/O of all processes
  - Elapsed communication cost
    - Max of I/O alone any path
  - (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 MapReduce 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 MapReduce Join

- Total communication cost
  - O(|R| + |S| + |RS|)
- Elapsed communication cost = O(*s*), where *s* is the I/O limit
  - 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 communication cost

# Outline

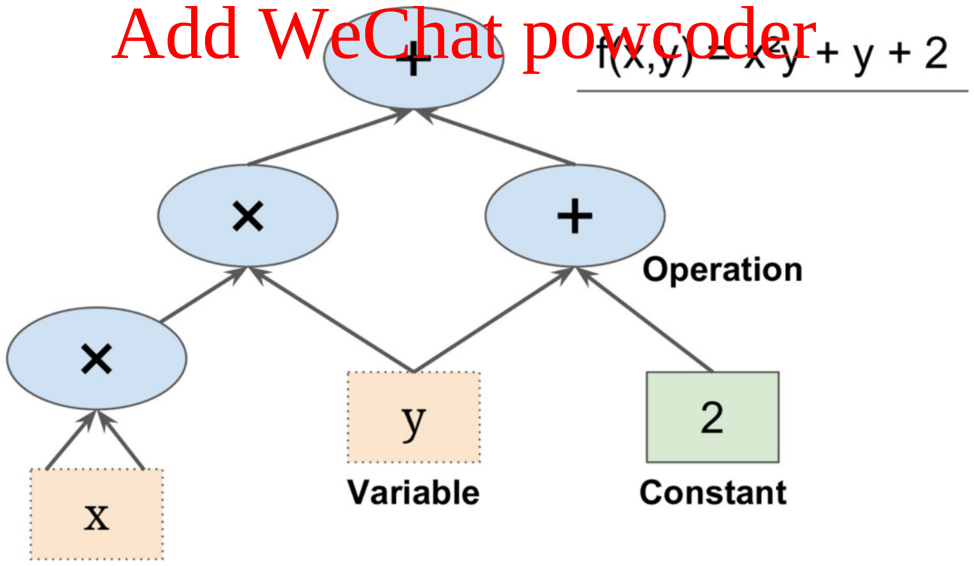
- Introduction
- The Hadoop Distributed File System (HDFS)
- MapReduce
- Hadoop
- Hadoop Streaming
- Problems Suited for MapReduce
- TensorFlow
- Frequent Itemsets
- Conclusion

## TensorFlow

- Interface for expressing machine learning algorithms, and an implementation for executing large-scale algorithms
- Dataflow framework that compiles to native CPU / GPU code
- Drastic reduction in development time
- Visualization (TensorBoard)

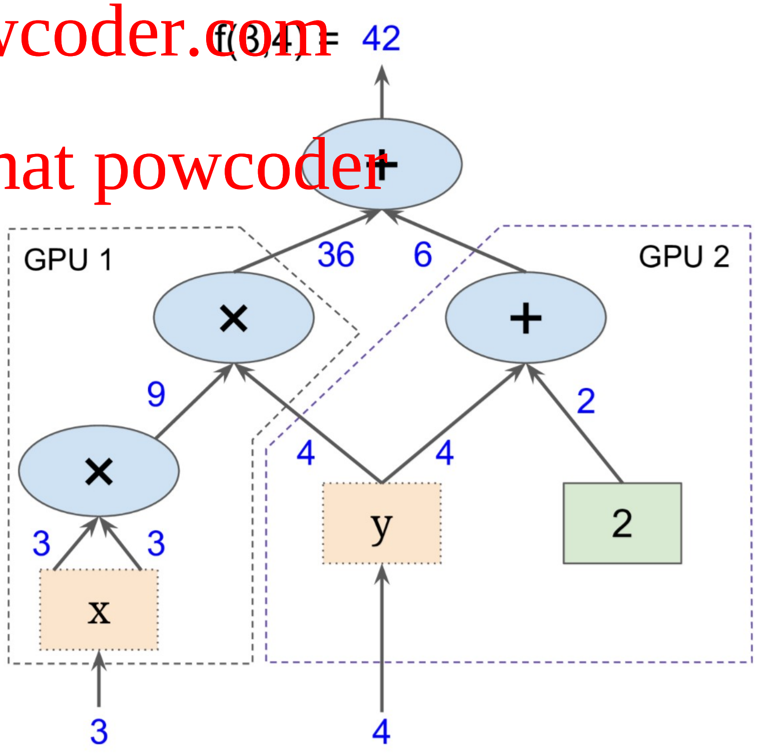
# Programming Model

- Express a numeric computation as a **graph**
  - Graph nodes are **operations** which have any number of inputs and outputs
  - Graph edges are **tensors** which flow between nodes



## Big Data: Distributed Environment

- Portability: deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API



## Outline

- Introduction
- The Hadoop Distributed File System (HDFS)
- MapReduce
- Hadoop
- Hadoop Streaming
- Problems Suited for MapReduce
- TensorFlow
- **Frequent Itemsets**
- Conclusion

# Frequent Itemsets

- Simple question: Find sets of items that appear together “frequently” in baskets
- Support for itemset : Number of baskets containing all items in
  - Often expressed as a fraction of the total number of baskets
- Given a support threshold , then sets of items that appear in at least baskets are called frequent itemsets.

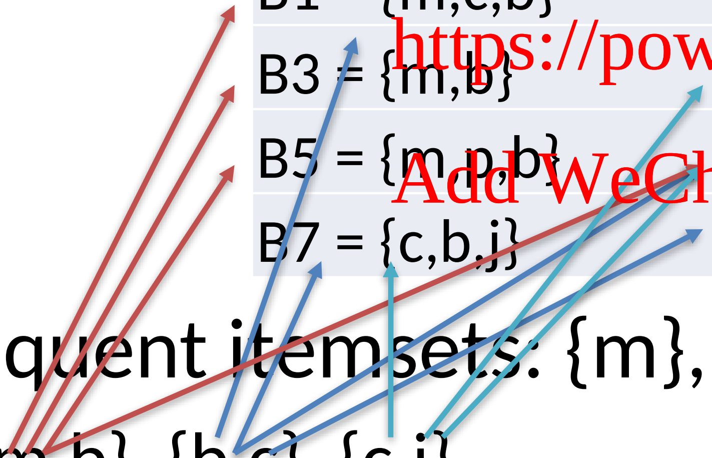
| TID | Items |
|---|---|
| 1 | Bread, Coke, Milk |
| 2 | Beer, Bread |
| 3 | Beer, Coke, Diaper, Milk |
| 4 | Beer, Bread, Diaper, Milk |
| 5 | Coke, Diaper, Milk |

Support of {Beer, Bread} = 2

## Example: Frequent Itemsets

- Items = {milk, coke, pepsi, beer, juice}
- Minimum support = 3 baskets

| | |
|---|---|
| B1 = {m,c,b} | B2 = {m,p,j} |
| B3 = {m,b} | B4 = {c,j} |
| B5 = {m,p,b} | B6 = {m,c,b,j} |
| B7 = {c,b,j} | B8 = {b,c} |

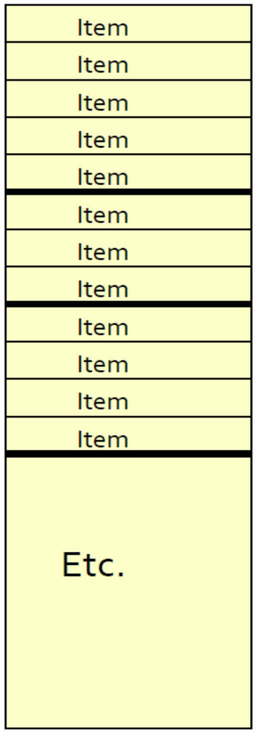


- Frequent itemsets: {m}, {c}, {b}, {j},

  {m,b}, {b,c}, {c,j}

# Itemsets: Computation Model

- Typically, data is kept in flat files rather than in a database system:
  - Stored on disk
  - Stored basket-by-basket
  - Baskets are small but we have many baskets and many items
    - Expand baskets into pairs, triples, etc. as you read baskets
    - Use k nested loops to generate all sets of size k



Items are positive integers, and boundaries between baskets are –1.

## Computation Model

- The true cost of mining disk-resident data is usually the number of disk I/O's
- In practice, association-rule algorithms read the data in passes – all baskets read in turn
- We measure the cost by the number of passes an algorithm makes over the data

## Main-Memory Bottleneck

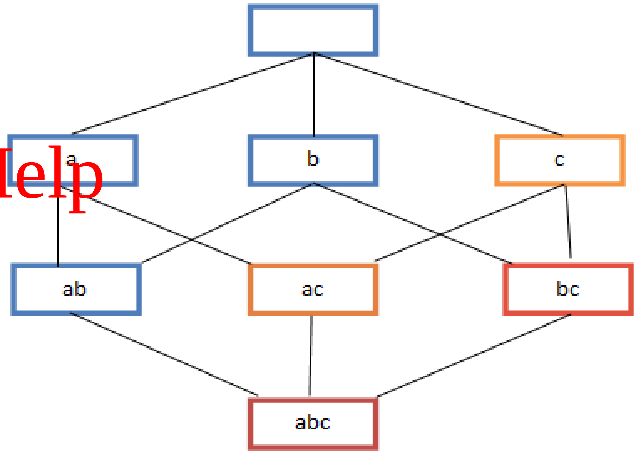
- For many frequent-itemset algorithms, main-memory is the critical resource
  - As we read baskets, we need to count something, e.g., occurrences of pairs of items
  - The number of different things we can count is limited by main memory
  - Swapping counts in/out is a disaster

## Naïve Algorithm to Count Pairs

- Read file once, counting in main memory the occurrences of each pair:
  - From each basket of *n* items, generate its *n(n-1)/2* pairs by two nested loops
- Fails if $(\#items)^2$ exceeds main memory
  - Remember: #items can be 100K (Wal-Mart) or 10B (Web pages)
    - Suppose $10^5$ items, counts are 4-byte integers
    - Number of pairs of items: $10^5(10^5-1)/2 = 5*10^9$
    - Therefore, $2*10^{10}$ (20 gigabytes) of memory needed

# A-Priori Algorithm

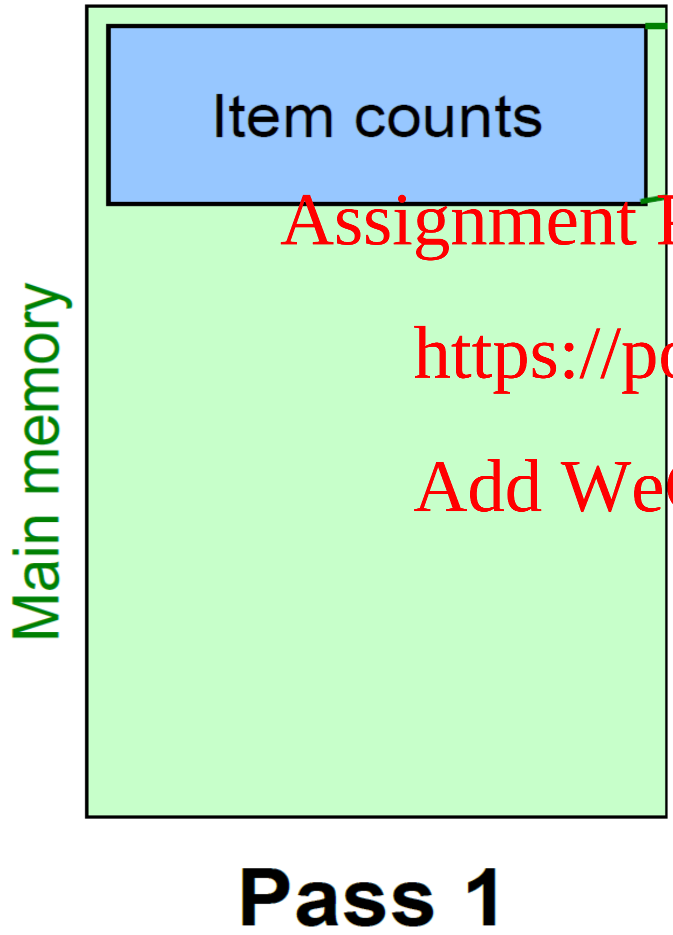
- A two-pass approach called a-priori limits the need for main memory
- Key idea: *monotonicity*
  - If a set of items *I* appears at least *s* times, so does every **subset** *J* of *I*.
- Contrapositive for pairs:
  - If item *i* does not appear in *s* baskets, then no pair including *i* can appear in *s* baskets



# A-Priori Algorithm

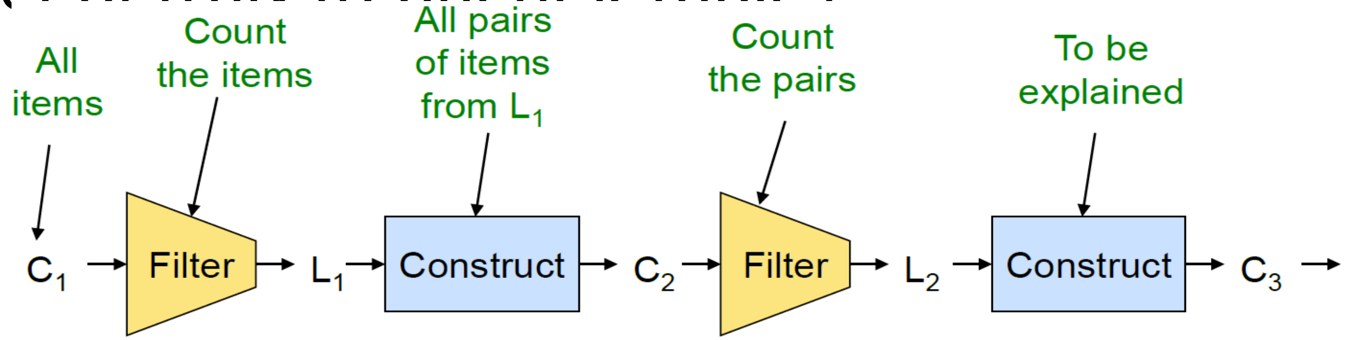
- Pass 1: Read baskets and count in main memory the occurrences of each individual item
  - Requires only memory proportional to #items
- Items that appear at least s times are the frequent items
- Pass 2: Read baskets again and count in main memory only those pairs where both elements are frequent (from Pass 1)
  - Requires memory proportional to square of frequent items only (for counts)
  - Plus a list of the frequent items (so you know what must be counted)

## Main-Memory: Picture of A-Priori



# Frequent Triplets, Etc.

- Now we know how to find frequent pairs, how about frequent triplets and frequent k-tuples?
- For each k, we construct two sets of k-tuples (sets of size k):
- = candidate k-tuples = those that might be frequent sets (support > s) based on information from the pass for *k*−1
- = the set of truly frequent *k*-tuples



# Example

- Hypothetical steps of the A-Priori algorithm
- = { {b} {c} {j} {m} {n} {p} }
- Count the support of itemsets in
- Prune non-frequent: = { b, c, j, m }
- Generate = { {b,c} {b,j} {b,m} {c,j} {c,m} {j,m} }
- Count the support of itemsets in
- Prune non-frequent: = { {b,m} {b,c} {c,m} {c,j} }
- Generate = { {b,c,m} {b,c,j} {b,m,j} {c,m,j} }
- Count the support of itemsets in
- Prune non-frequent: = { {b,c,m} }

# A-Priori for All Frequent Itemsets

- One pass for each *k* (itemset size)
- Needs room in main memory to count each candidate *k*-tuple
- For typical market-basket data and reasonable support (e.g., 1%), *k* = 2 requires the most memory
- Many possible extensions: Lower the support *s* as itemset gets bigger
  - Association rules with intervals:
    - For example: Men over 65 have 2 cars
  - Association rules when items are in a taxonomy
    - Bread, Butter → FruitJam
    - BakedGoods, MilkProduct → PreservedGoods

# Map-Reduce Implementation

- Divide the file in which we want to find frequent itemsets into equal chunks randomly.

- Solve the frequent itemsets problem for the smaller chunk at each node. (Pretend the chunk is the entire dataset)

- Given:
  - Each chunk is fraction  of the whole input file (total  chunks)
  - is the support threshold for the algorithm
  - or  is the threshold as we process a chunk

## Map-Reduce Implementation

- At each node, we can use A-Priori algorithm to solve the smaller problem

- Take the group of all the itemsets that have been found frequent for one or more chunks.

  - Every itemset that is frequent in the whole file is frequent in at least one chunk
  - All the true frequent itemsets are among the candidates

## Map-Reduce Implementation

- We can arrange the aforementioned algorithm in a two-pass Map-Reduce framework
  - First Map-Reduce cycle to produce the candidate itemsets
  - Second Map-Reduce cycle to calculate the true frequent itemsets.

## Map-Reduce Implementation

**First Mapper**

- Run A-Priori algorithm on the chunk using support threshold
- Output the frequent itemsets for that chunk (F, c), where F is the key (itemset) and c is count (or proportion)

**First Reducer**

- Output the candidate itemsets to verify in the second pass
- Given (F,c), discard c and output all candidate itemsets Fs

## Map-Reduce Implementation

**Second Mapper**

- For all the candidate itemsets produced by first reducer, count the frequency in local chunk

**Second Reducer**

- Aggregate the output of second mapper, and sum the count to get the frequency of candidate itemsets in the whole input file
- Filter the itemsets with support smaller than

## Outline

- Introduction
- The Hadoop Distributed File System (HDFS)
- MapReduce
- Hadoop
- Hadoop Streaming
- Problems Suited for MapReduce
- TensorFlow
- Frequent Itemsets
- Conclusion

## Conclusion

- HDFS is a reliable distributed file system
- MapReduce is a distributed computing environment
  - Hadoop is an open-source implementation of MapReduce
  - Hadoop Streaming allow you to use any language to write MapReduce code
- Frequent Itemsets problem can be solved efficiently using its monotonicity property
  - A-Priori algorithm

# One-Slide Takeaway

- HDFS is a distributed file system built with robust in mind
- MapReduce is a convenient paradigm to implement parallel program
- Hadoop is an open source implementation of MapReduce
- Hadoop streaming allows you to use any language to program mapper and reducer
- Monotonicity property enable efficient algorithms for Frequent Itemsets problem

# References

- MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean and Sanjay Ghemawat
- The Hadoop Distributed File System: Architecture and Design by Apache Foundation Inc
- Hadoop in Action, Chuck Lam
- Hadoop File System, ppt by B. Ramamurthy
- Intro & MapReduce, pdf by Jure Leskovec
- Association Rules, pdf by Jure Leskovec
- Writing an Hadoop MapReduce Program in Python, http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/

# In-Class Practice

**Given the following input:**

I spent long spells at sea on all types of vessel; I followed officer training with the Surface Fleet and with the Royal Marines.

1. Write the output of the word count mapper's output for the above input.
2. Write the output of the word count mapper's output after the shuffle process.
3. Write the output of the word count reducer's output for the above input.