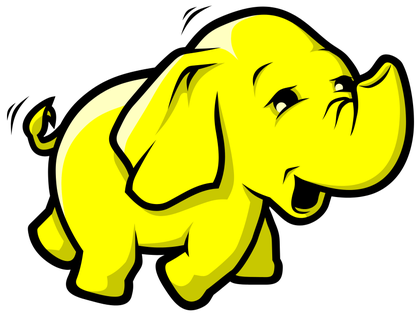
UNIVERSITY of VIRGINIA

CS4740
CLOUD COMPUTING

MapReduce

Prof. Chang Lou, UVA CS, Fall 2025

LAB 1 IS RELEASED



- **Lab1: MapReduce**
  - Deadline: 9/29 23:59:59 EST
  - A lot of interesting questions to answer in your lab 1 implementation
    - How should workers fetch tasks from the coordinator? How to divide and merge data after each phase? What if some workers finish faster? What if workers crash and start again? ...
- **My advice: Start the lab 1 immediately**
  - Don't wait for lectures: they only cover high-level designs
  - You'll learn much more from paper and hands-on experience
- Olsson 005, 9/8, **Lab 1 Overview Session**, 5:00 PM. We **strongly suggest** you to attend.
  - In summary: start lab 1 **today**!

AGENDA

- **MapReduce**
  - Motivation
  - How it works: Word count example
  - Additional challenges and how to deal with

GAME: BIRTHDAY SEARCH

- Goal: find the youngest student for each month (jan 30, feb 28, mar..)!

- Rule 1: Select seven students acting as distributed nodes (1 coordinator + 6 workers).
- Rule 2: Once starts, only two students may engage in a conversation at one time (no group chat).
- Rule 3: Students may record on paper, and paper (except birthday tag) can be passed along as needed.
- Winner bonus: 2pts in the final, Time limit: 3 mins

GAME: BIRTHDAY SEARCH

— Jan 29

— Feb 28

— Mar 31

— Apr 30

— May 30

— Jun 27

— Jul 30

— Aug 28

— Sep 25

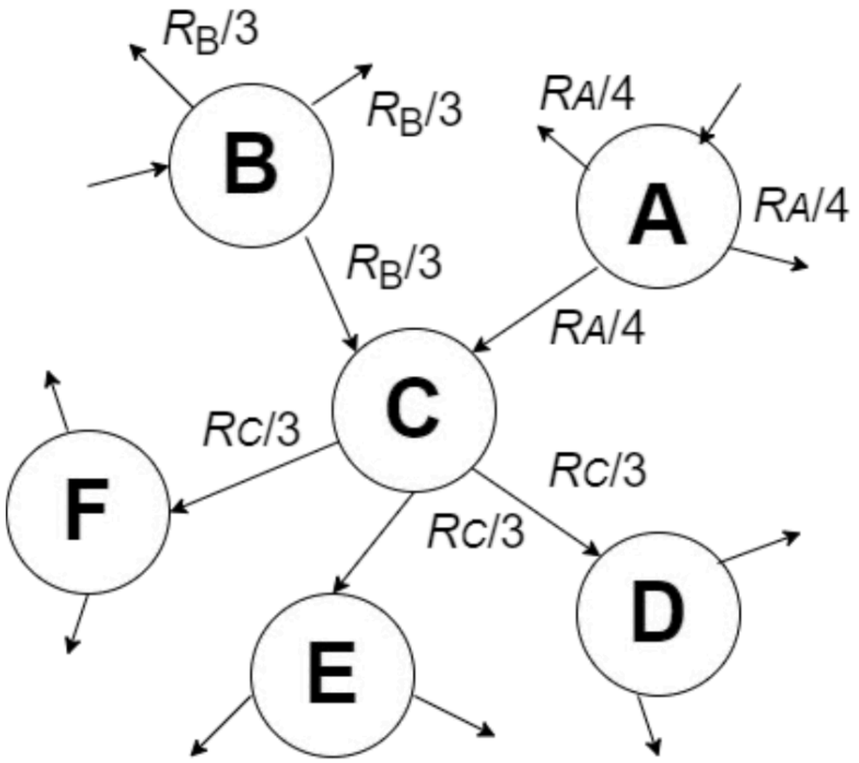
— Oct 28

— Nov 27

— Dec 29

LARGE-SCALE ANALYTICS

- Compute the frequency of words in a corpus of documents.
- Count how many times users have clicked on each of a (large) set of ads.
- PageRank: Compute the “importance” of a web page based on the “importances” of the pages that link to it.
- ....



So why we need MapReduce?

OPTION 1: SQL

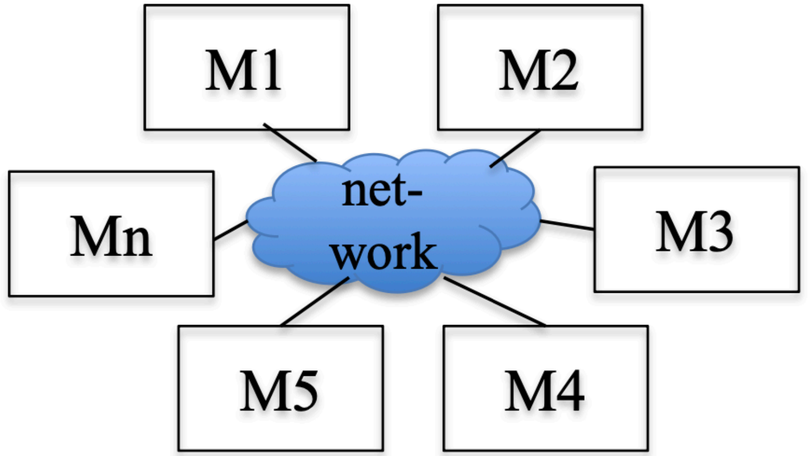
- Before MapReduce, analytics mostly done in SQL, or manually.
- Example: Count word appearances in a corpus of documents.
- With SQL, the rough query might be:

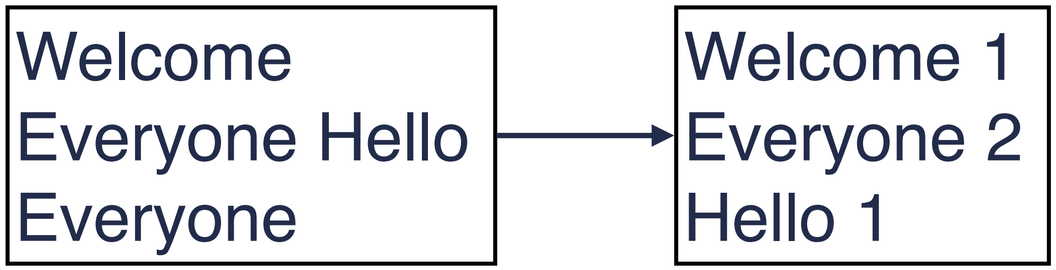
```
SELECT COUNT(*) FROM (
    SELECT UNNEST(string_to_array(doc_content, ' ')) as word
    FROM Corpus )
 GROUP BY word
```

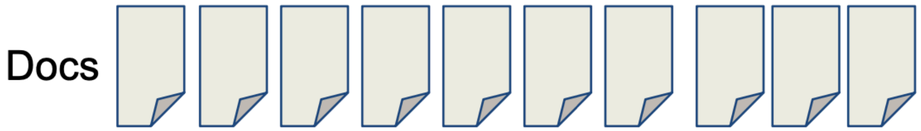
- Very expressive, convenient to program
- But no one knew how to **scale** SQL execution!

OPTION 2: MANUAL

- Example: Count word appearances in a corpus of documents.
  - In real worlds, you are given a huge dataset (e.g., Wikipedia dump or all of Shakespeare's works)

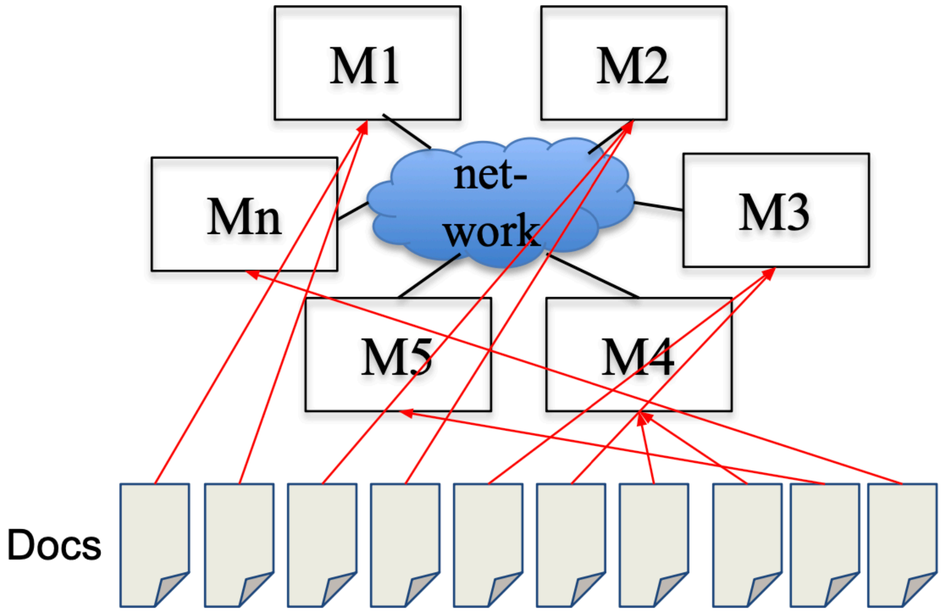






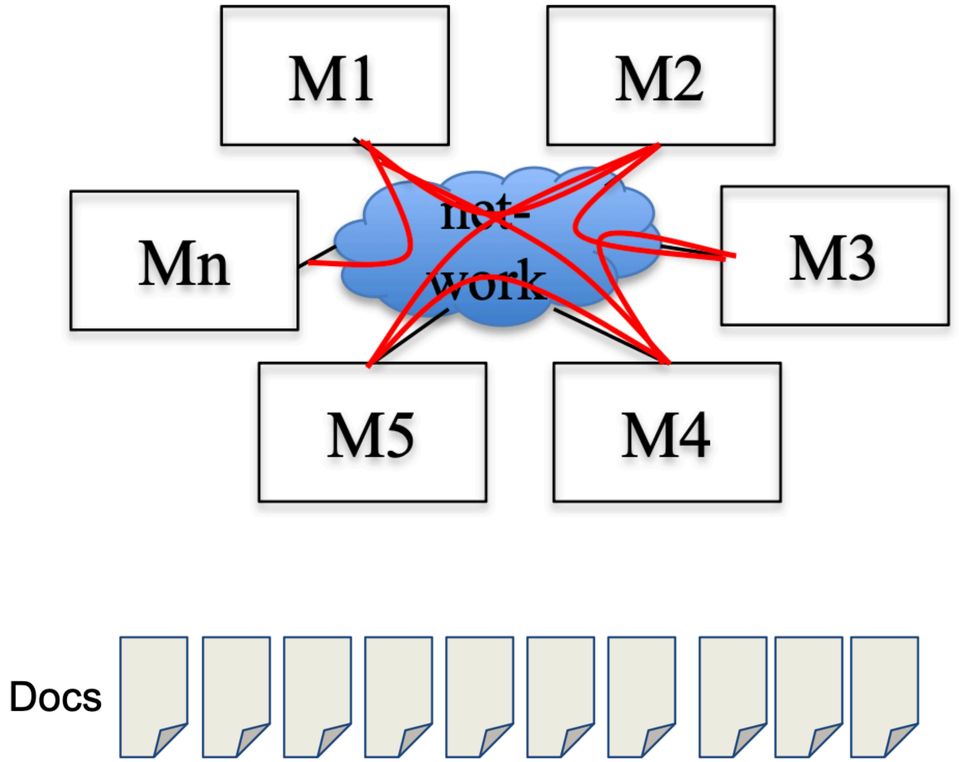
OPTION 2: MANUAL

- Example: Count word appearances in a corpus of documents.
- Phase 1: Assign documents to different machines/nodes.
  - Each computes a dictionary: {word: local_freq}.



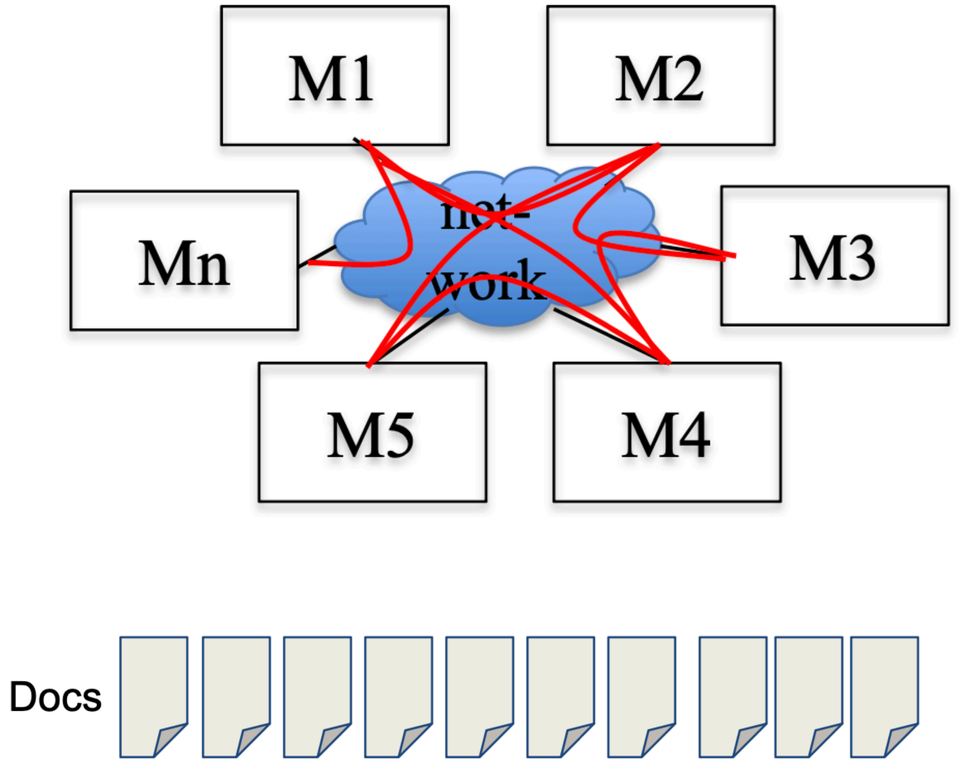
OPTION 2: MANUAL

- Example: Count word appearances in a corpus of documents.
- Phase 1: Assign documents to different machines/nodes.
  - Each computes a dictionary: {word: local_freq}.
- Phase 2: Nodes exchange dictionaries (how?) to aggregate local_freq's.
  - But how to make this scale??



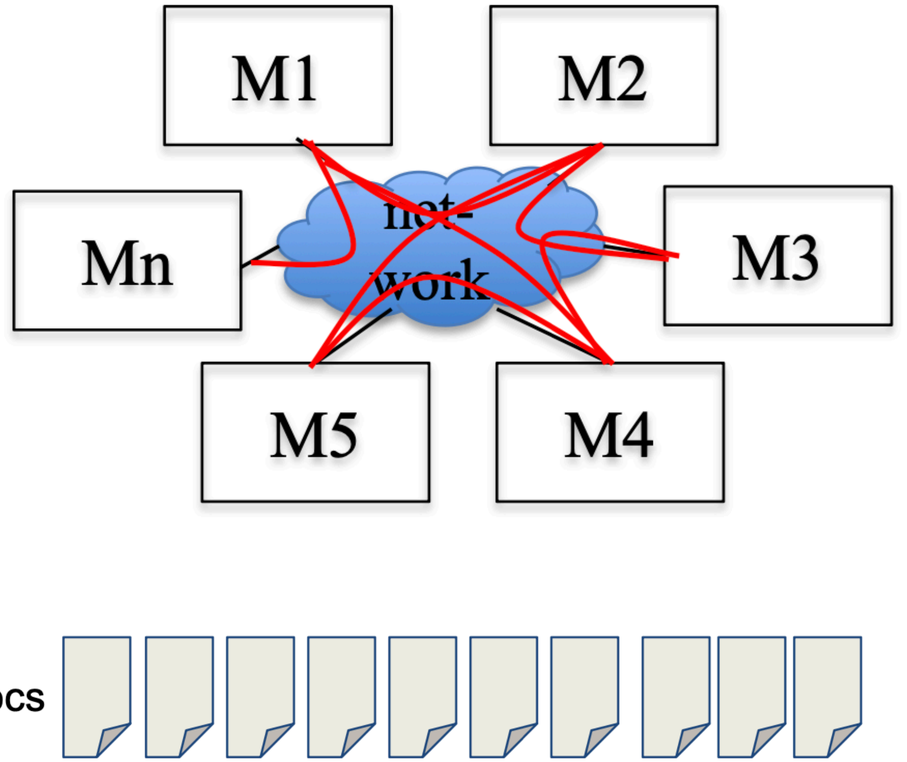
OPTION 2: MANUAL

- Phase 2, Option a: Send all {word: local_freq} dictionaries to one node, who aggregates.
  - But what if it's too much data for one node?
- Phase 2, Option b: Each node sends (word, local_freq) to a designated node, e.g., node with ID hash(word) % N.



OPTION 2: CHALLENGES

- How to generalize to other applications?
- How to deal with failures?
- How to deal with slow nodes?
- How to deal with load balancing (some docs are very large, others small)?
- How to deal with skew (some words are very frequent, so nodes designated to aggregate them will be pounded)?



They are common challenges for large-scale analytics!

Can we have both easy-to-program models and awesome scalability?

ANSWER: MAPREDUCE

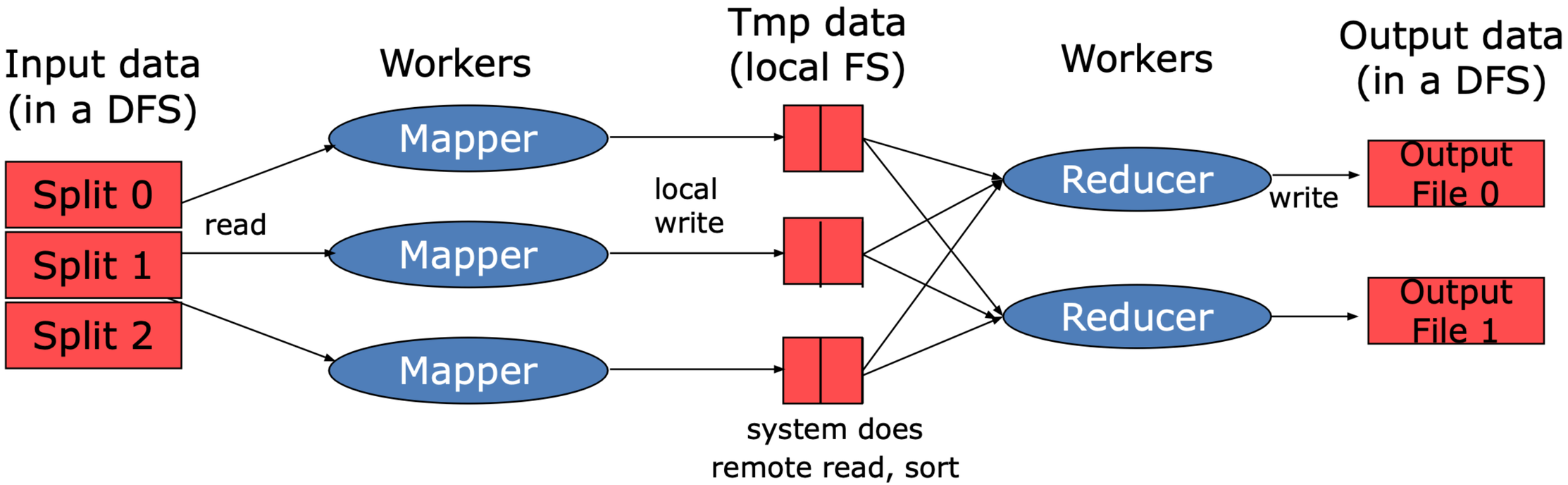
- **Parallelizable** programming model:
  - Applies to a broad class of analytics applications.
  - Isn't as expressive as SQL but it is easier to scale.
  - Consists of three phases, each intrinsically parallelizable:
    - **Map**: processes input elements independently to emit relevant (key, value) pairs from each.
    - Transparently, the runtime system groups all the values for each key together: (key, [list of values]), called "shuffle".
    - **Reduce**: aggregates all the values for each key to emit a global value for each key.
- Scalable, efficient, fault tolerant runtime system (discuss in later slides).
- Technical paper: please read it!

https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

MAPREDUCE: HOW IT WORKS

- Input: a collection of elements of (key, value) pair type.
- Programmer defines two functions:
  - Map(key, value) → a list of (key', value') pairs
  - Reduce(key, value-list) → output
- Execution
  - Apply Map to each input key-value pair, in parallel for different keys.
  - Sort emitted (key', value') pairs to produce (key' value'-list) pairs.
  - Apply Reduce to each (key', value'-list) pair, in parallel for different keys.
- Output is the union of all Reduce invocations' outputs.

MAPREDUCE: WORKFLOW



Map phase:
extract something you care about from each record

Reduce phase:
aggregate

EXAMPLE: WORD COUNT

- We have a directory, which contains many documents.
- The documents contain words separated by whitespace and punctuation.
- Goal: Count the number of times each distinct word appears across the files in the directory.

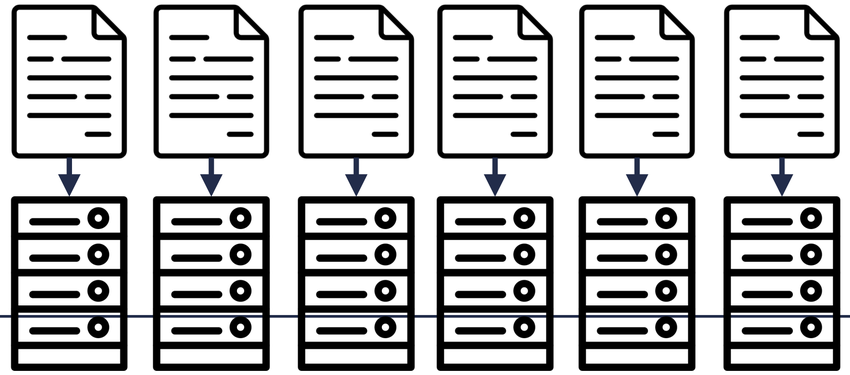
MAP PHASE

- Mapper is given key: document ID; value: document content, say:

(D1,“The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am.”)

- It will emit the following pairs:

<The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store,1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store,1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>



..

INTERMEDIARY PHASE (SHUFFLE)

– Transparently, the runtime sorts emitted (key, value) pairs by key:

<9am, 1>
<at, 1>
<closed, 1>
<in, 1>
<morning, 1>
<opens, 1>
<opens, 1>
<store, 1>
<store,1>
<store, 1>
<store,1>

Reducer 1

<teacher, 1>
<the, 1>
<the, 1>
<the, 1>
<the, 1>
<the 1>
<to, 1>
<went, 1>
<was, 1>
<The, 1>

Reducer 2

REDUCE PHASE

- For each unique key emitted from the Map Phase, function Reduce(key, value-list) is invoked on Reducer 1 or Reducer 2.
- Across their invocations, these Reducers will emit:

Reducer 1

<9am, 1>
<at, 1>
<closed, 1>
<in, 1>
<morning, 1>
<opens, 2>
<store, 4>

Reducer 2

<teacher, 1>
<the, 5>
<to, 1>
<went, 1>
<was, 1>
<The, 1>

Output

WORD COUNT WITH MAPREDUCE

```
Map(key, value): // key: document ID; value: document content
  FOR (each word w IN value)
    emit(w, 1);
```

```
Reduce(key, value-list): // key: a word; value-list: a list of integers
  result = 0;
  FOR (each integer v on value-list)
    result += v;
  emit(key, result);
```

EXERCISE 1

- (MapReduce) You are given a symmetric social network (like Facebook) where a is a friend of b implies that b is also a friend of a. The input is a dataset D (sharded) containing such pairs (a, b) – note that either a or b may be a lexicographically lower name. Pairs appear exactly once and are not repeated. Find the last names of those users whose first name is “Kanye” and who have at least 300 friends.
- Write pseudocode code for Map() and Reduce(). Your pseudocode may assume the presence of appropriate primitives (e.g., “firstname(user_id)”, etc.). The Map function takes as input a tuple (key=a,value=b)

EXERCISE 1 SOLUTION

- M1 (a,b):
  - if (firstname(a)==Kanye) then output (a,b)
  - if (firstname(b)==Kanye) then output (b,a)
  - // note that second if is NOT an else if, so a single M1 function may be output up to 2 KV pairs!
- R1 (x, V):
  - if |V| >= 300 then output (lastname(x), -)

BTW, WHY CALLED MAPREDUCE?

– Terms are borrowed from Functional Language (e.g., Lisp)

(**map** square ‘(1 2 3 4))
– Output: (1 4 9 16)
[processes each record sequentially and independently]

(**reduce** + ‘(1 4 9 16))
– (+ 16 (+ 9 (+ 4 1)))
– Output: 30
[processes set of all records in batches]

CHAINING MAPREDUCE

- The programming model seems pretty restrictive.
- But quite a few analytics applications can be written with it, especially with a technique called chaining.
- If the output of reducers is (key, value) pairs, then their output can be passed onto other Map/Reduce processes.
- This chaining can support a variety of analytics (though certainly not all types of analytics, e.g., no ML b/c no loops).

EXAMPLE: WORD FREQUENCY

- Suppose instead of word count, we wanted to compute word frequency: the probability that a word would appear in a document.

- This means computing the fraction of times a word appears, out of the total number of words in the corpus.

SOLUTION: CHAIN TWO MAPREDUCE'S

- First Map/Reduce: Word Count (like before)
  - Map: process documents and output <word, 1> pairs.
  - Multiple Reducers: emit <word, word_count> for each word.
- Second MapReduce:
  - Map: process <word, word_count> and output (1, (word, word_count)).
  - 1 Reducer: perform two passes:
    - In first pass, sum up all word_count's to calculate overall_count.
    - In second pass calculate fractions and emit multiple <word, word_count/overall_count>.
- Scalability is not too bad, as first stage's output is a rather small dictionary (maximum # of English words with an integer for each).

A FEW CHALLENGES

- How to improve performance?
- How to deal with failures?
- How to deal with slow nodes?

PERFORMANCE

- A common bottleneck: network
  - input data transferred all over the cluster, how to solve?
- Solution: scheduling policy for **data locality**
  - Asks DFS for locations of replicas of input file blocks.
  - Map tasks are scheduled so DFS input block replica are on the same machine or on the same rack.
- Effect: Thousands of machines read input at local speed.
  - eliminate network bottleneck!

FAULT TOLERANCE

- Failures are the norm in data centers.
  - Worker failure
  - Master failure

- Worker failure solution: **Heartbeats**
  - Master detects if workers failed by periodically pinging them.
  - Re-execute in-progress map/reduce tasks.
- Master failure solution: **Replication**
  - Initially, was single point of failure; Resume from Execution Log. Subsequent versions used replication and consensus.
- Effect: From Google's paper: once, a Map/Reduce job lost 1600 of 1800 machines, but it still finished fine.

STRAGGLERS

- Slow workers or stragglers significantly lengthen completion time.
- Slowest worker can determine the total latency!
  - Other jobs consuming resources on machine.
  - Bad disks with errors transfer data very slowly.
  - This is why many systems measure 99th percentile latency.
- Solution: spawn backup copies of tasks (**redundant execution**).
  - Whichever one finishes first "wins."
  - I.e., treat slow executions as failed executions!

TAKEAWAYS

- Big data requires distributed computation power
- MapReduce: efficient, generalized and fault-tolerant framework for distributed analytics
  - uses **parallelization** + **aggregation** to schedule applications across clusters
  - various designs deal with failure: heartbeats, replication, redundant execution
- Today: OLSSON HALL 005, **Lab 1** Overview Session, 5:00 PM
- Next class: **RPC (remote procedure call)**

ACKNOWLEDGEMENT

THIS COURSE IS DEVELOPED HEAVILY BASED ON COURSE MATERIALS SHARED BY PROF. INDRANIL GUPTA, PROF. ROBERT MORRIS, PROF. MICHAEL FREEDMAN, PROF. KYLE JAMIESON, PROF. WYATT LLOYD AND PROF. ROXANA GEAMBASU. MANY APPRECIATIONS FOR GENEROUSLY SHARING THEIR MATERIALS AND TEACHING INSIGHTS.