MapReduce: A Crash Course

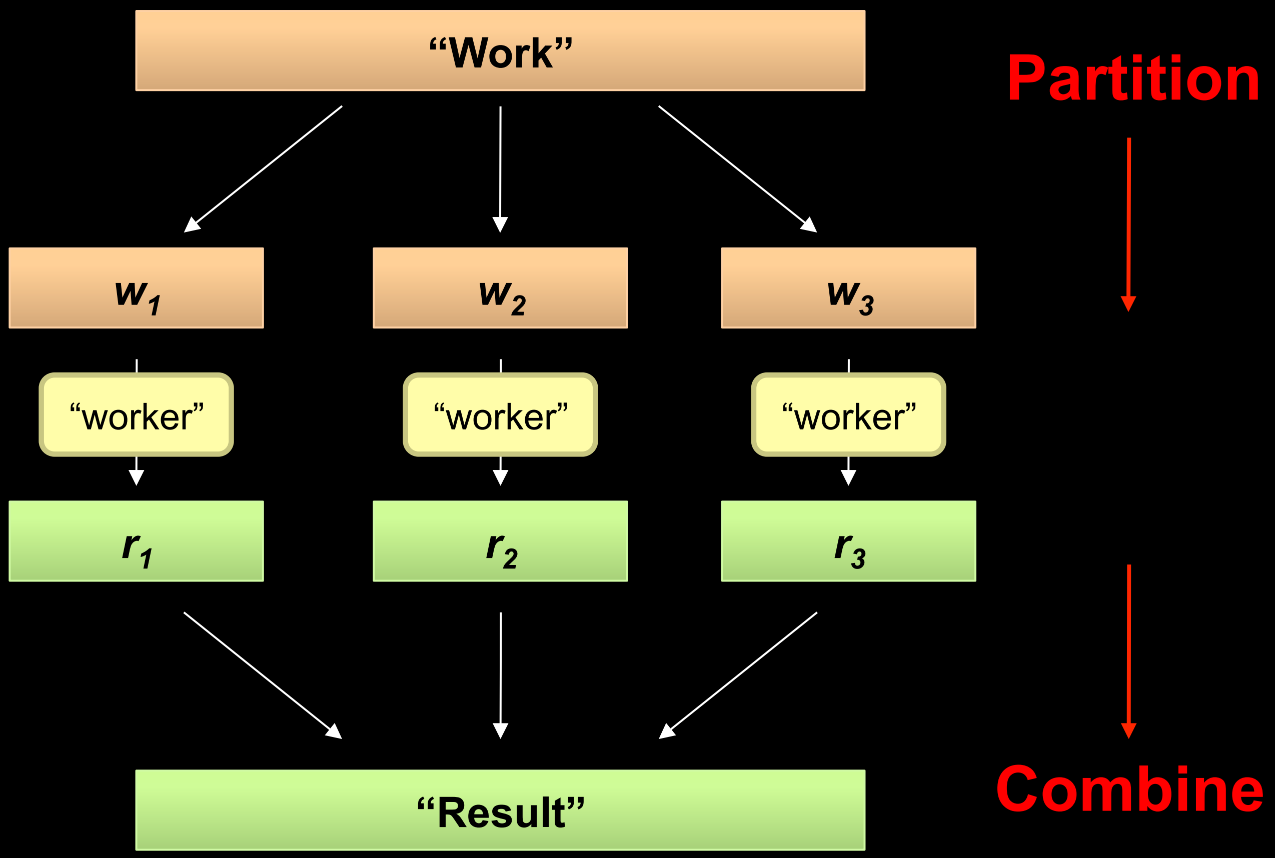
Chris Dyer
Language Technologies Institute
Carnegie Mellon University

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See http://creativecommons.org/licenses/by-nc-sa/3.0/us/ for details

Outline

- What is MapReduce?
- The MapReduce programming model
- Application: inverted indexing

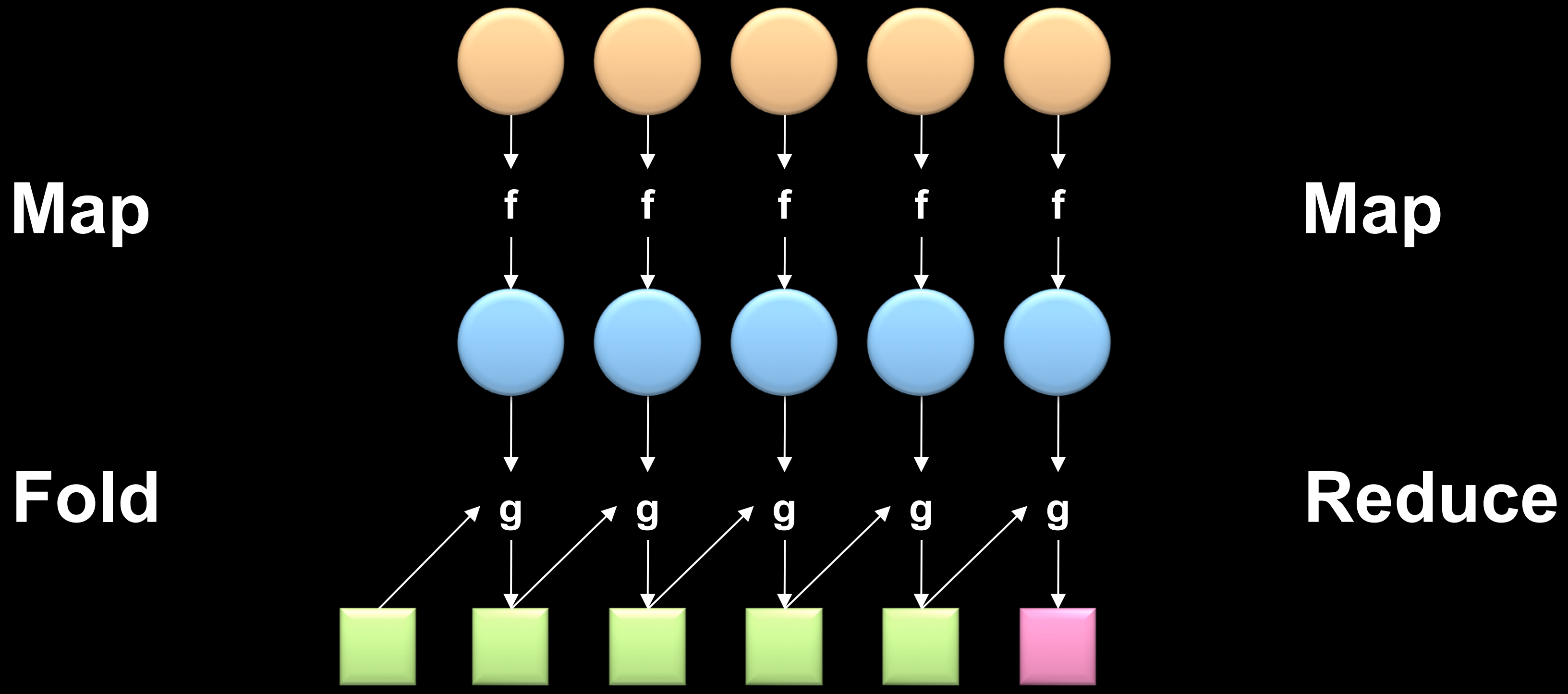
Divide and Conquer



Typical Problem

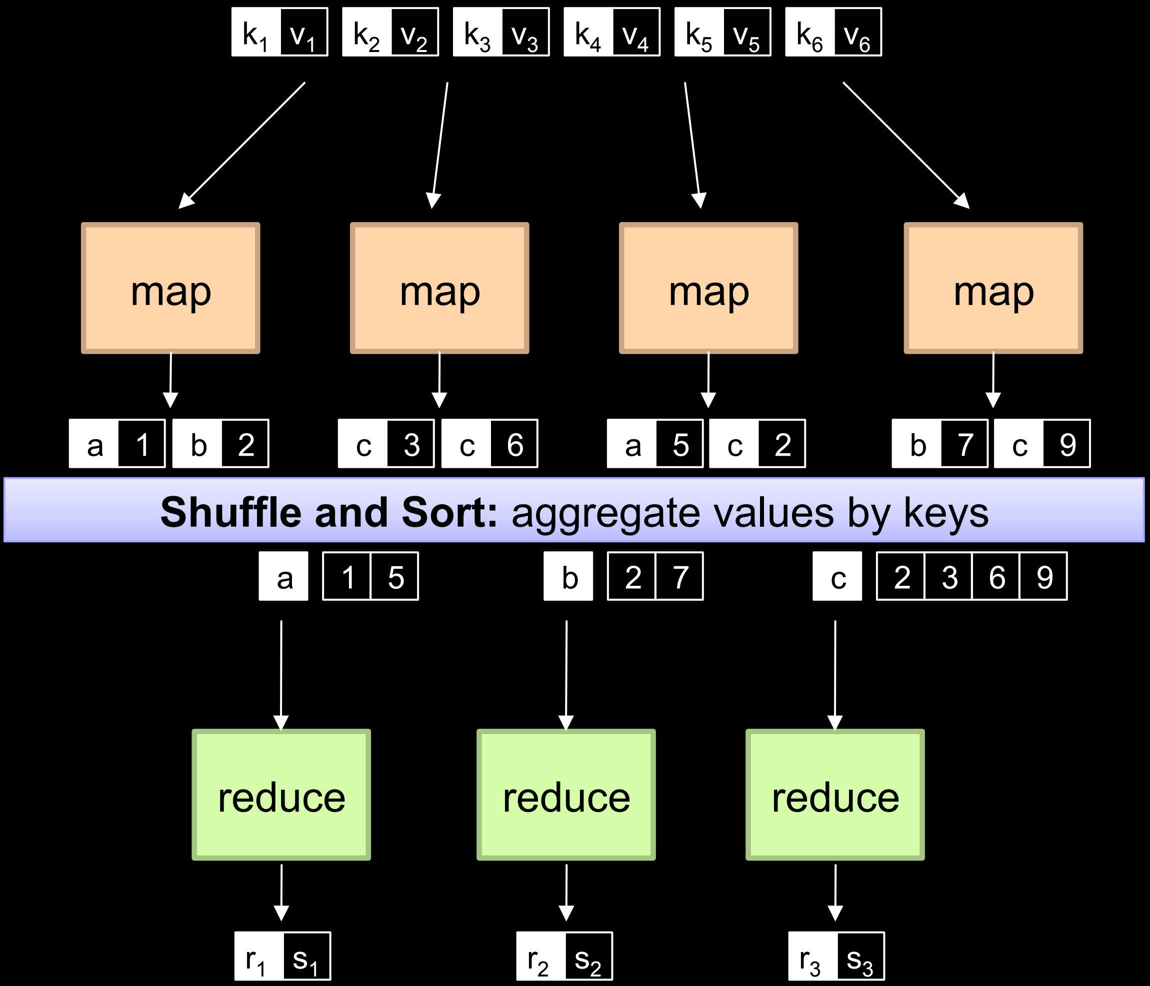
- Iterate over a large number of records
- Map Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results Reduce
- Generate final output

Key idea: functional abstraction for these two operations


Map
f
f
f
f
f
Map
Fold
g
g
g
g
g
Reduce

MapReduce

- Programmers specify two functions:
  - **map** (k, v) → <k’, v’>*
  - **reduce** (k’, v’) → <k’, v’>*
    - All values with the same key are reduced together
- Usually, programmers also specify:
  - **partition** (k’, number of partitions) → partition for k’
    - Often a simple hash of the key, e.g. hash(k’) mod n
    - Allows reduce operations for different keys in parallel
  - **combine**(k’,v’) → <k’,v’>
    - “Mini-reducers” that run in memory after the map phase
    - Optimizes to reduce network traffic & disk writes
- Implementations:
  - Google has a proprietary implementation in C++
  - Hadoop is an open source implementation in Java


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
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 9
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 9
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

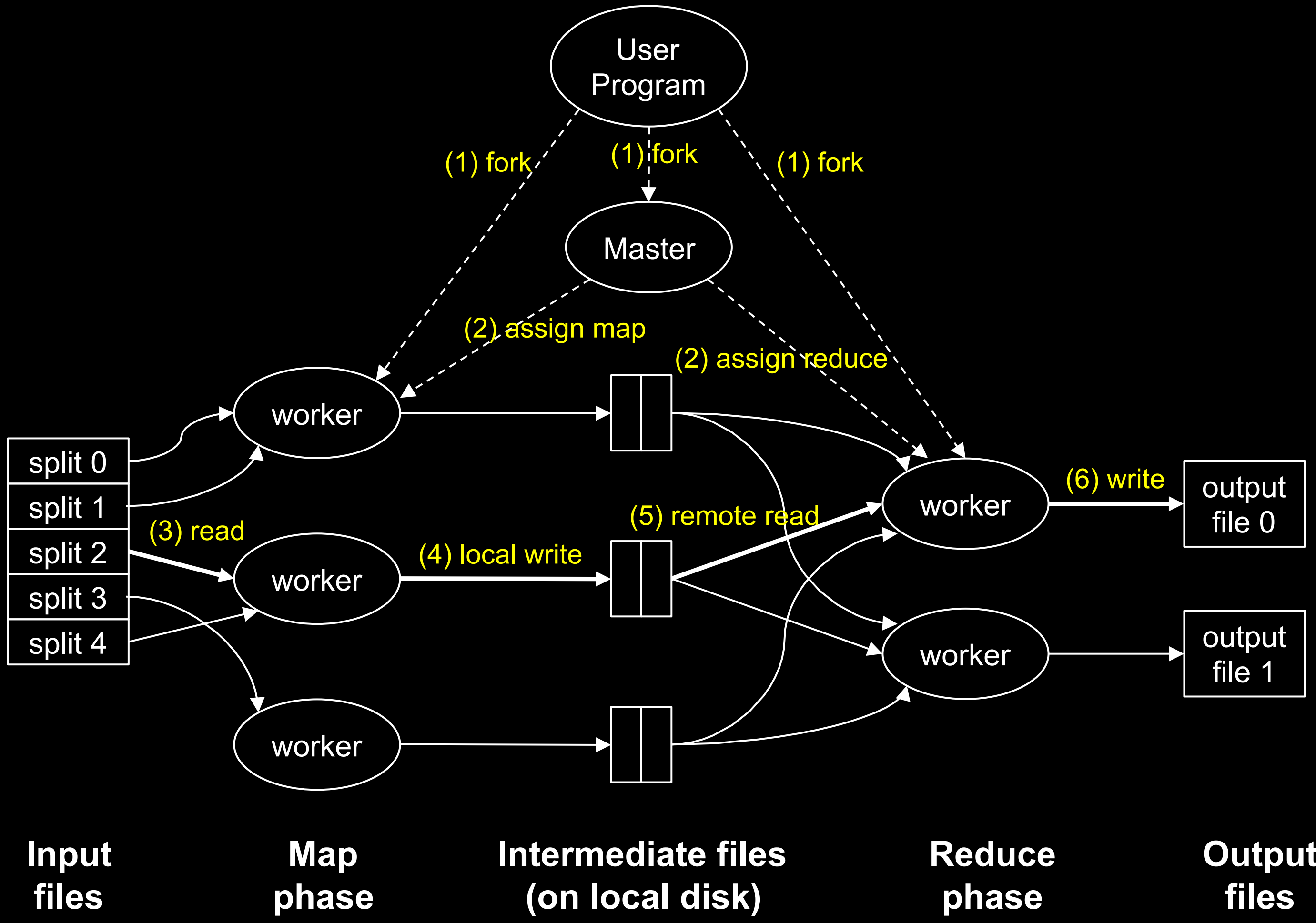
MapReduce Runtime

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves the process to the data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed FS (later)

“Hello World”: Word Count

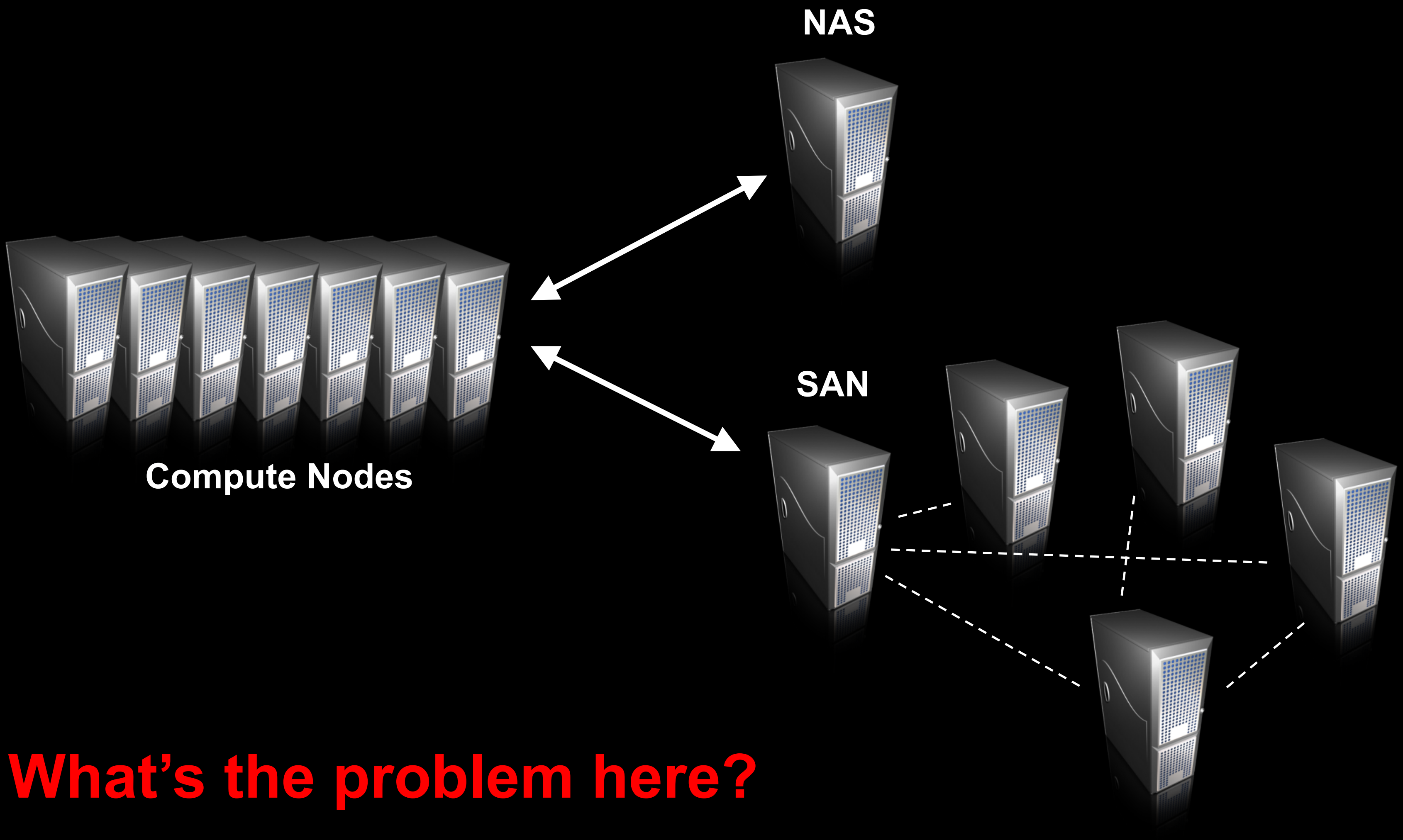
```
Map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_values:
        EmitIntermediate(w, "1");

Reduce(String key, Iterator intermediate_values):
    // key: a word, same for input and output
    // intermediate_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
        Emit(AsString(result));
```



Redrawn from Dean and Ghemawat (OSDI 2004)

How do we get data to the workers?



What's the problem here?

Distributed File System

- Don’t move data to workers… Move workers to the data!
  - Store data on the local disks for 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, disk throughput is good
- A distributed file system is the answer
  - GFS (Google File System)
  - HDFS for Hadoop (= GFS clone)

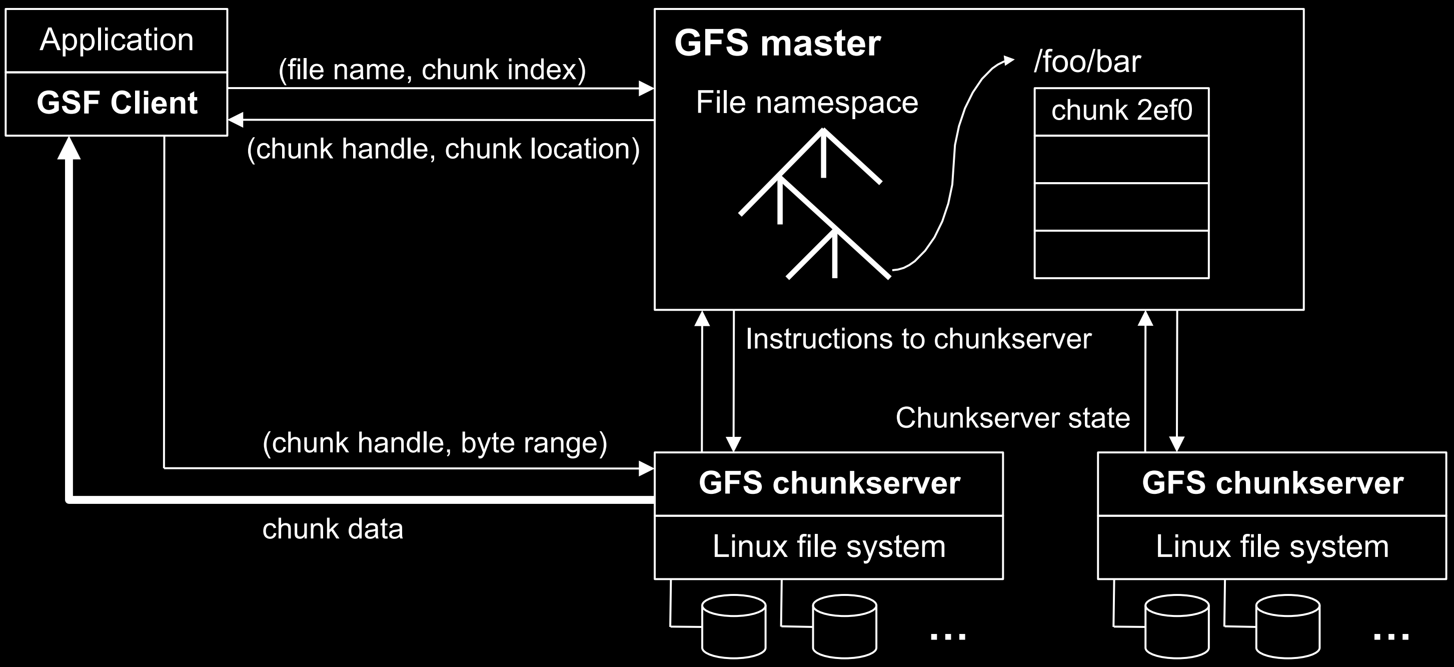
GFS: Assumptions

- Commodity hardware over “exotic” hardware
- High component failure rates
  - Inexpensive commodity components fail all the time
- “Modest” number of HUGE files
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
- High sustained throughput over low latency

GFS slides adapted from material by Dean et al.

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 data sets, streaming reads
- Simplify the API
  - Push some of the issues onto the client



Redrawn from Ghemawat et al. (SOSP 2003)

Master's Responsibilities

- Metadata storage
- Namespace management/locking
- Periodic communication with chunkservers
- Chunk creation, replication, rebalancing
- Garbage collection

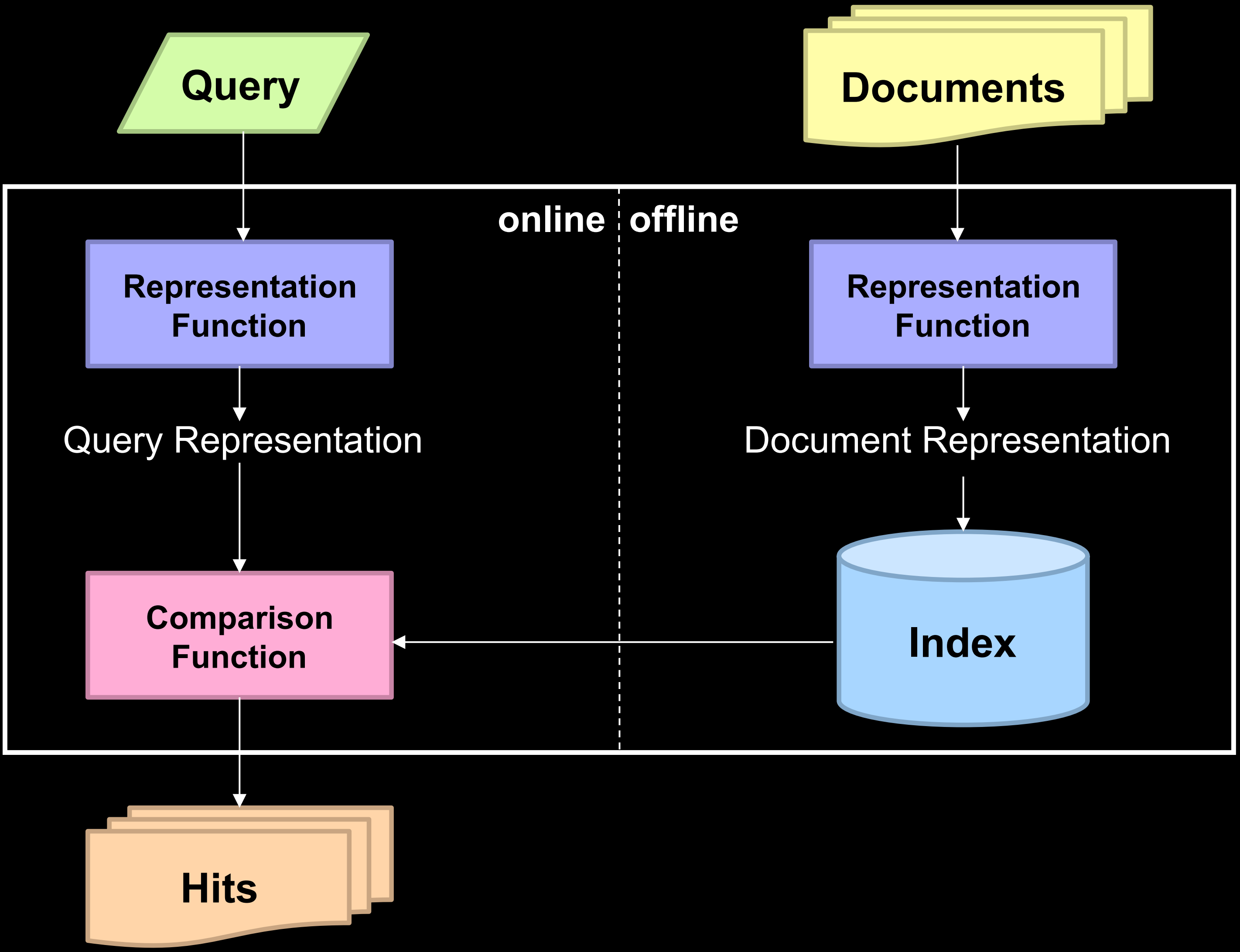
Questions?

MapReduce Application:
Inverted Indexing

Text Retrieval: Topics

- Introduction to information retrieval (IR)
- Inverted indexing with MapReduce

Architecture of IR Systems



How do we represent text?

- "Bag of words"
  - Treat all the words in a document as index terms for that document
  - Assign a weight to each term based on "importance"
  - Disregard order, structure, meaning, etc. of the words
  - Simple, yet effective!
- Assumptions
  - Term occurrence is independent
  - Document relevance is independent
  - "Words" are well-defined

Sample Document

McDonald's slims down spuds

Fast-food chain to reduce certain types of fat in its french fries with new cooking oil.

NEW YORK (CNN/Money) - McDonald's Corp. is cutting the amount of "bad" fat in its french fries nearly in half, the fast-food chain said Tuesday as it moves to make all its fried menu items healthier.

But does that mean the popular shoestring fries won't taste the same? The company says no. "It's a win-win for our customers because they are getting the same great french-fry taste along with an even healthier nutrition profile," said Mike Roberts, president of McDonald's USA.

But others are not so sure. McDonald's will not specifically discuss the kind of oil it plans to use, but at least one nutrition expert says playing with the formula could mean a different taste.

Shares of Oak Brook, Ill.-based McDonald's (MCD: down $0.54 to $23.22, Research, Estimates) were lower Tuesday afternoon. It was unclear Tuesday whether competitors Burger King and Wendy's International (WEN: down $0.80 to $34.91, Research, Estimates) would follow suit. Neither company could immediately be reached for comment.

…

"Bag of Words"

16 × said

14 × McDonalds

12 × fat

11 × fries

8 × new

6 × company, french, nutrition

5 × food, oil, percent, reduce, taste, Tuesday

…

Representing Documents

Document 1

The quick brown fox jumped over the lazy dog's back.

Document 2

Now is the time for all good men to come to the aid of their party.

Term	Document 1	Document 2
aid	0	1
all	0	1
back	1	0
brown	1	0
come	0	1
dog	1	0
fox	1	0
good	0	1
jump	1	0
lazy	1	0
men	0	1
now	0	1
over	1	0
party	0	1
quick	1	0
their	0	1
time	0	1

Stopword List
for
is
of
the
to

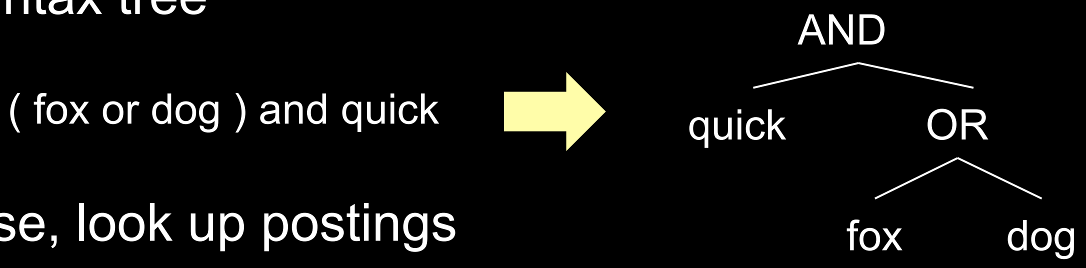
Inverted Index

Term	Doc 1	Doc 2	Doc 3	Doc 4	Doc 5	Doc 6	Doc 7	Doc 8
aid	0	0	0	1	0	0	0	1
all	0	1	0	1	0	1	0	0
back	1	0	1	0	0	0	1	0
brown	1	0	1	0	1	0	1	0
come	0	1	0	1	0	1	0	1
dog	0	0	1	0	1	0	0	0
fox	0	0	1	0	1	0	1	0
good	0	1	0	1	0	1	0	1
jump	0	0	1	0	0	0	0	0
lazy	1	0	1	0	1	0	1	0
men	0	1	0	1	0	0	0	1
now	0	1	0	0	0	1	0	1
over	1	0	1	0	1	0	1	1
party	0	0	0	0	0	1	0	1
quick	1	0	1	0	0	0	0	0
their	1	0	0	0	1	0	1	0
time	0	1	0	1	0	1	0	0

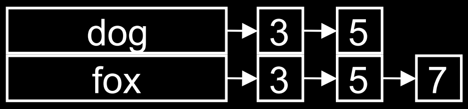
Term	Postings
aid	→ 4 → 8
all	→ 2 → 4 → 6
back	→ 1 → 3 → 7
brown	→ 1 → 3 → 5 → 7
come	→ 2 → 4 → 6 → 8
dog	→ 3 → 5
fox	→ 3 → 5 → 7
good	→ 2 → 4 → 6 → 8
jump	→ 3
lazy	→ 1 → 3 → 5 → 7
men	→ 2 → 4 → 8
now	→ 2 → 6 → 8
over	→ 1 → 3 → 5 → 7 → 8
party	→ 6 → 8
quick	→ 1 → 3
their	→ 1 → 5 → 7
time	→ 2 → 4 → 6

Boolean Retrieval

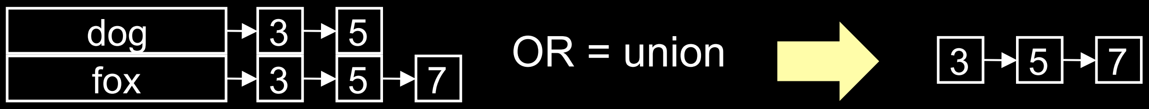
- To execute a Boolean query:
  - Build query syntax tree



  - For each clause, look up postings



  - Traverse postings and apply Boolean operator



- Efficiency analysis
  - Postings traversal is linear (assuming sorted postings)
  - Start with shortest posting first

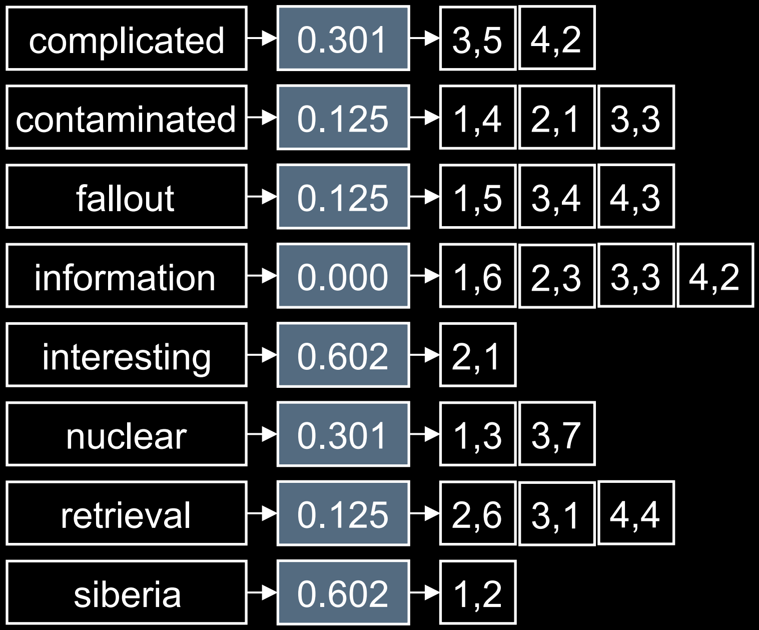
TF.IDF Term Weighting

$$w_{i,j} = \mathrm{tf}_{i,j} \cdot \log \frac{N}{n_i}$$

$w_{i,j}$	**weight assigned to term *i* in document *j***
$\mathrm{tf}_{i,j}$	**number of occurrence of term *i* in document *j***
N	**number of documents in entire collection**
n_i	**number of documents with term *i***

TF.IDF Example

	tf				*idf*
	1	2	3	4	
complicated			5	2	0.301
contaminated	4	1	3		0.125
fallout	5		4	3	0.125
information	6	3	3	2	0.000
interesting		1			0.602
nuclear	3		7		0.301
retrieval		6	1	4	0.125
siberia	2				0.602



complicated	0.301	3,5	4,2		
contaminated	0.125	1,4	2,1	3,3	
fallout	0.125	1,5	3,4	4,3	
information	0.000	1,6	2,3	3,3	4,2
interesting	0.602	2,1			
nuclear	0.301	1,3	3,7		
retrieval	0.125	2,6	3,1	4,4	
siberia	0.602	1,2			

MapReduce it?

- The indexing problem
  - Must be relatively fast, but need not be real time
  - For Web, incremental updates are important
  - Crawling is a challenge itself!
- The retrieval problem
  - Must have sub-second response
  - For Web, only need relatively few results

Indexing: Performance Analysis

- Fundamentally, a large sorting problem
  - Terms usually fit in memory
  - Postings usually don’t
- How is it done on a single machine?
- How large is the inverted index?
  - Size of vocabulary
  - Size of postings

MapReduce: Index Construction

- Map over all documents
  - Emit *term* as key, (*docid*, *tf)* as value
  - Emit other information as necessary (e.g., term position)
- Reduce
  - Trivial: each value represents a posting!
  - Might want to sort the postings (e.g., by *docid* or *tf*)
- MapReduce does all the heavy lifting!

Query Execution

- MapReduce is meant for large-data batch processing
  - Not suitable for lots of real time operations requiring low latency
- The solution: “the secret sauce”
  - Most likely involves document partitioning
  - Lots of system engineering: e.g., caching, load balancing, etc.

Questions?

MapReduce Algorithm Design

(Chapter 3)

Managing Dependencies

- Remember: Mappers run in isolation
  - You have no idea in what order the mappers run
  - You have no idea on what node the mappers run
  - You have no idea when each mapper finishes
- Tools for synchronization:
  - Ability to hold state in reducer across multiple key-value pairs
  - Sorting function for keys
  - Partitioner
  - Cleverly-constructed data structures

Motivating Example

- Term co-occurrence matrix for a text collection
  - M = N x N matrix (N = vocabulary size)
  - M_{ij}: number of times *i* and *j* co-occur in some context (for concreteness, let's say context = sentence)
- Why?
  - Distributional profiles as a way of measuring semantic distance
  - Semantic distance useful for many language processing tasks

"You shall know a word by the company it keeps" (Firth, 1957)

e.g., Mohammad and Hirst (EMNLP, 2006)

MapReduce: Large Counting Problems

- Term co-occurrence matrix for a text collection
  = specific instance of a large counting problem
  - A large event space (number of terms)
  - A large number of events (the collection itself)
  - Goal: keep track of interesting statistics about the events
- Basic approach
  - Mappers generate partial counts
  - Reducers aggregate partial counts

How do we aggregate partial counts efficiently?

First Try: “Pairs”

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For all pairs, emit (a, b) → count
- Reducers sums up counts associated with these pairs
- Use combiners!

Note: in these slides, we donate a key-value pair as k → v

“Pairs” Analysis

- Advantages
  - Easy to implement, easy to understand
- Disadvantages
  - Lots of pairs to sort and shuffle around (upper bound?)

Another Try: "Stripes"

- Idea: group together pairs into an associative array

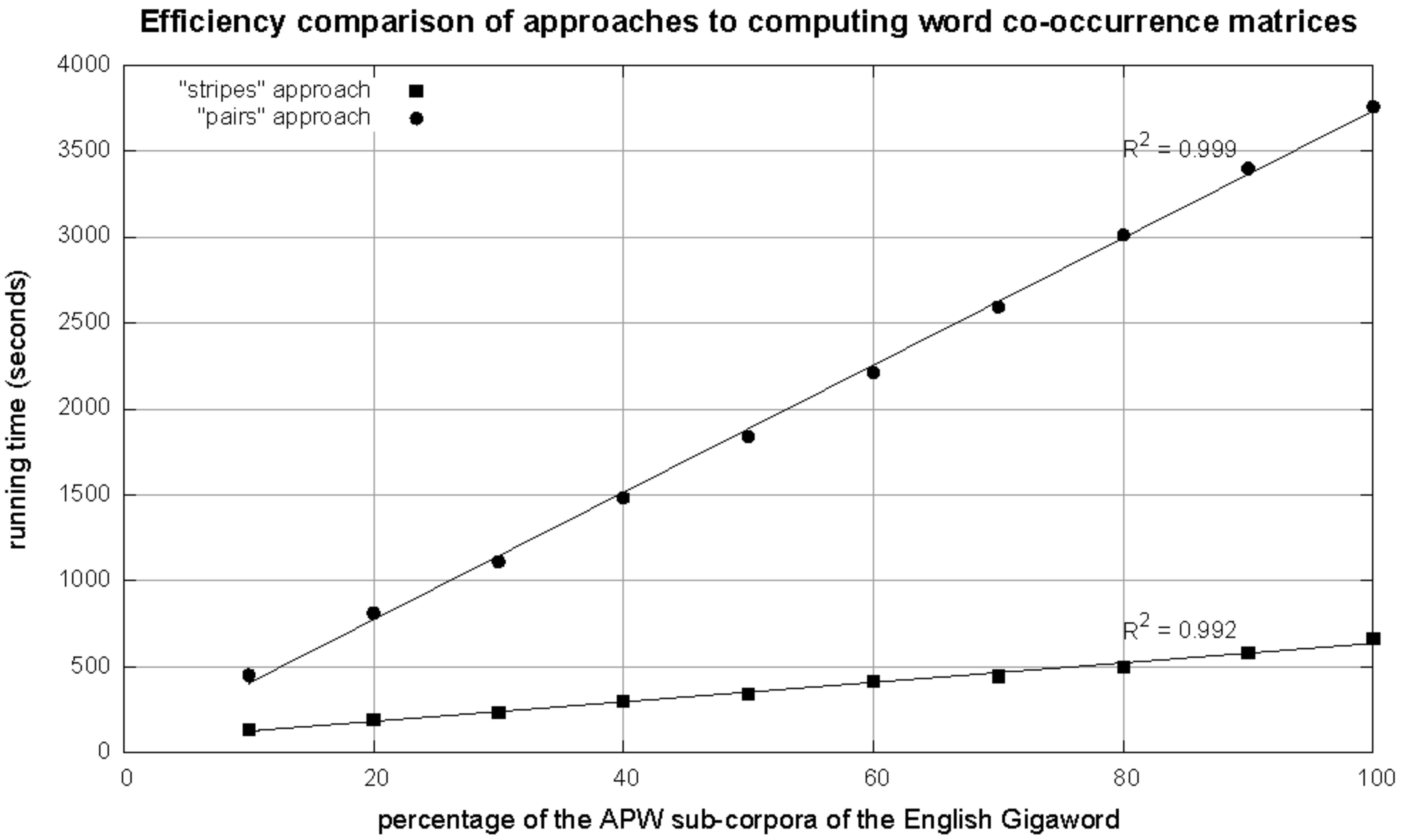
$$\begin{array}{l} (a, b) \rightarrow 1 \\ (a, c) \rightarrow 2 \\ (a, d) \rightarrow 5 \\ (a, e) \rightarrow 3 \\ (a, f) \rightarrow 2 \end{array} \qquad a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$$

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For each term, emit a → { b: count$_b$, c: count$_c$, d: count$_d$ … }
- Reducers perform element-wise sum of associative arrays

$$\begin{array}{rl} & a \rightarrow \{ b: 1, d: 5, e: 3 \} \\ + & a \rightarrow \{ b: 1, c: 2, d: 2, f: 2 \} \\ \hline & a \rightarrow \{ b: 2, c: 2, d: 7, e: 3, f: 2 \} \end{array}$$

“Stripes” Analysis

- Advantages
  - Far less sorting and shuffling of key-value pairs
  - Can make better use of combiners
- Disadvantages
  - More difficult to implement
  - Underlying object is more heavyweight
  - Fundamental limitation in terms of size of event space



Cluster size: 38 cores
Data Source: Associated Press Worldstream (APW) of the English Gigaword Corpus (v3), which contains 2.27 million documents (1.8 GB compressed, 5.7 GB uncompressed)

Relative frequency estimates

- How do we compute relative frequencies from counts?

$$P(B \mid A) = \frac{\text{count}(A,B)}{\text{count}(A)} = \frac{\text{count}(A,B)}{\sum_{B'} \text{count}(A,B')}$$

- Why do we want to do this?
- How do we do this with MapReduce?

P(B|A): “Pairs”

$(a, *) \rightarrow 32$ **Reducer holds this value in memory**

$(a, b_1) \rightarrow 3$	$(a, b_1) \rightarrow 3 / 32$
$(a, b_2) \rightarrow 12$	$(a, b_2) \rightarrow 12 / 32$
$(a, b_3) \rightarrow 7$	$(a, b_3) \rightarrow 7 / 32$
$(a, b_4) \rightarrow 1$	$(a, b_4) \rightarrow 1 / 32$
…	…

- For this to work:
  - Must emit extra (a, *) for every b_n in mapper
  - Must make sure all a’s get sent to same reducer (use partitioner)
  - Must make sure (a, *) comes first (define sort order)
  - Must hold state in reducer across different key-value pairs

P(B|A): "Stripes"

a $\rightarrow$ {b_1:3, b_2 :12, b_3 :7, b_4 :1, … }

- Easy!
  - One pass to compute (a, *)
  - Another pass to directly compute *f*(B|A)

Synchronization in Hadoop

- Approach 1: turn synchronization into an ordering problem
  - Sort keys into correct order of computation
  - Partition key space so that each reducer gets the appropriate set of partial results
  - Hold state in reducer across multiple key-value pairs to perform computation
  - Illustrated by the “pairs” approach
- Approach 2: construct data structures that “bring the pieces together”
  - Each reducer receives all the data it needs to complete the computation
  - Illustrated by the “stripes” approach

Issues and Tradeoffs

- Number of key-value pairs
  - Object creation overhead
  - Time for sorting and shuffling pairs across the network
  - In Hadoop, every object emitted from a mapper is written to disk
- Size of each key-value pair
  - De/serialization overhead
- Combiners make a big difference!
  - RAM vs. disk and network
  - Arrange data to maximize opportunities to aggregate partial results

Questions?

Thank you!