中山大學
SUN YAT-SEN UNIVERSITY

Lecture 7
MapReduce Algorithms

SE-808 Cloud Application Development (supported by Google)
http://my.ss.sysu.edu.cn/courses/cloud/

School of Software, Sun Yat-sen University

Outline

- Basic MapReduce Algorithms
  - Sort/Search
- MapReduce algorithm design
  - Managing dependencies
  - Coordinating mappers and reducers
- Case study #1: inverted index
- Case study #2: pairwise similarity comparison

Part of the slides are adapted from Jimmy Lin's cloud course slides: http://www.umiacs.umd.edu/~jimmylin/cloud-2010-Spring/index.html

MapReduce Jobs

- Tend to be very short, code-wise
  - IdentityReducer is very common
- “Utility”jobs can be composed
- Represent a data flow, more so than a procedure

Sort: Inputs

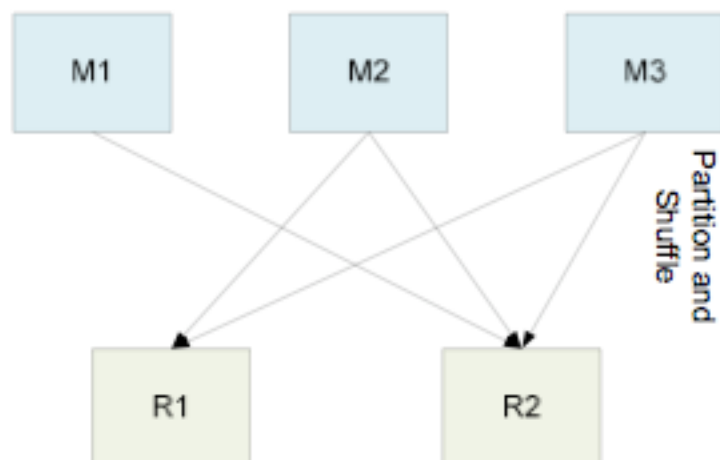
- A set of files, one value per line.
- Mapper key is file name, line number.
- Mapper value is the contents of the line.

Sort Algorithm

- Takes advantage of reducer properties:
  - (key, value) pairs are processed in order by key;
  - reducers are themselves ordered •
- Mapper
  - Identity function for value (k, v) →(v, _)
- Reducer
  - Identity function (k’, _) -> (k’, “”)

Sort: The Trick

- (key, value) pairs from mappers are sent to a particular reducer based on hash(key)
- Must pick the hash function for your data such that
  - k< k=> hash(k) < hash(k)



Final Thoughts on Sort

- Used as a test of Hadoop's raw speed
- Essentially "IO drag race"
- Highlights utility of GFS

- Yahoo: Hadoop Sorts a Petabyte in 16.25 Hours and a Terabyte in 62 Seconds, 3800 nodes, May 2009.

Search: Inputs

- A set of files containing lines of text
  - A search pattern to find
  - Mapper key is file name, line number
  - Mapper value is the contents of the line
- Search pattern sent as special parameter

Search Algorithm

- Mapper:

  Given (filename, some text) and “pattern”, if “text” matches “pattern” output (filename, _)

- Reducer:

  Identity function

Exercise 1: Graph reversal

- Given a directed graph as an adjacency list:

  src1: dest11, dest12, …
  src2: dest21, dest22, …

- Construct the graph in which all the links are reversed

Exercise 2: Frequent Pairs

- Given a large set of market baskets, find all frequent pairs
  - Frequent pairs are item pairs with counts greater than a given threshold

MapReduce Algorithm Design

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 all these slides, a key-value pair denoted 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

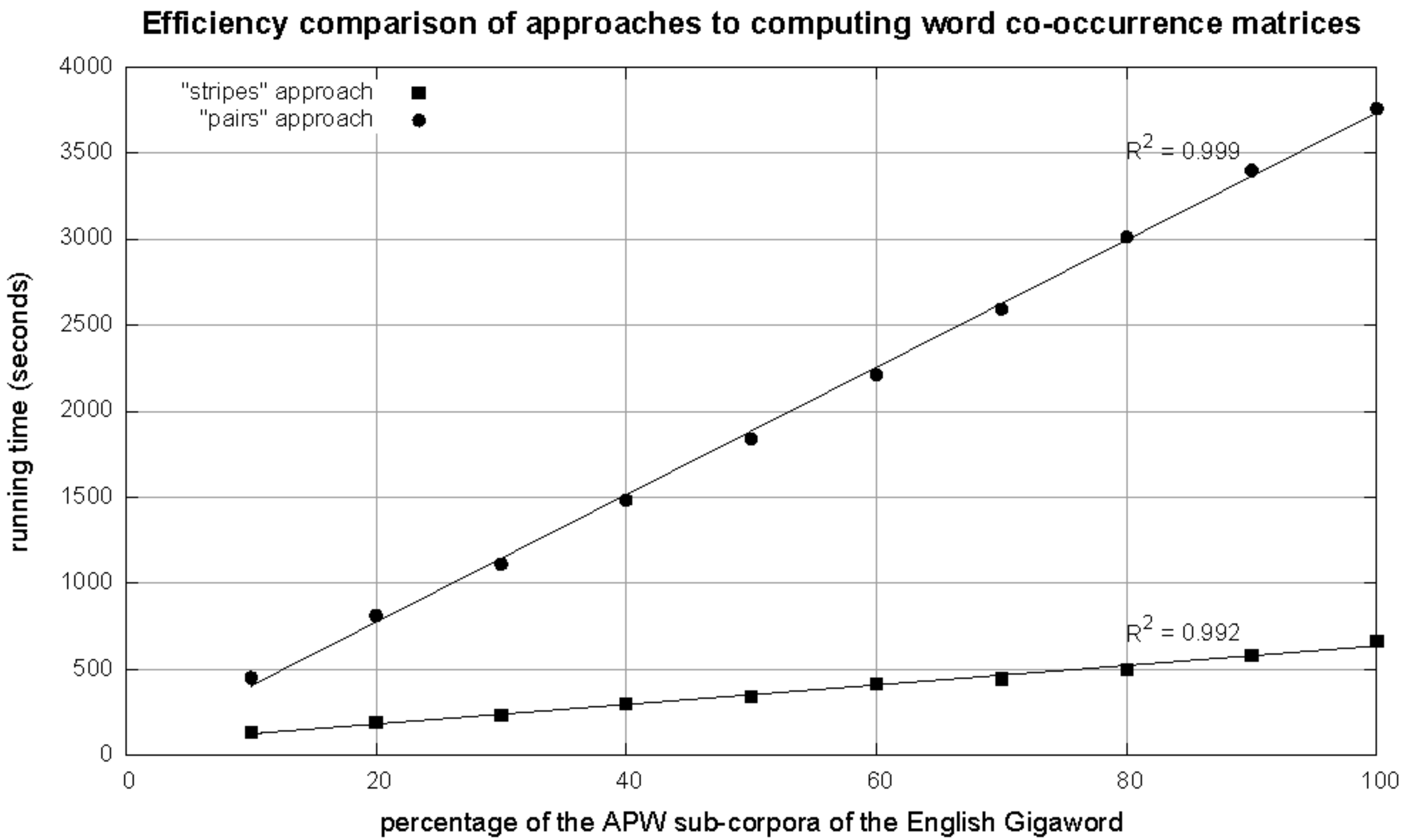
(a, b) → 1
(a, c) → 2
(a, d) → 5 a → { b: 1, c: 2, d: 5, e: 3, f: 2 }
(a, e) → 3
(a, f) → 2

- 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

	a → { b: 1, d: 5, e: 3 }
+	a → { b: 1, c: 2, d: 2, f: 2 }
	a → { b: 2, c: 2, d: 7, e: 3, f: 2 }

“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)

Conditional Probabilities

- How do we compute conditional probabilities 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, *) → 32 Reducer holds this value in memory

(a, b_1) → 3	(a, b_1) → 3 / 32
(a, b_2) → 12	(a, b_2) → 12 / 32
(a, b_3) → 7	(a, b_3) → 7 / 32
(a, b_4) → 1	(a, b_4) → 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"

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

- Easy!
  - One pass to compute (a, *)
  - Another pass to directly compute P(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
- 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?

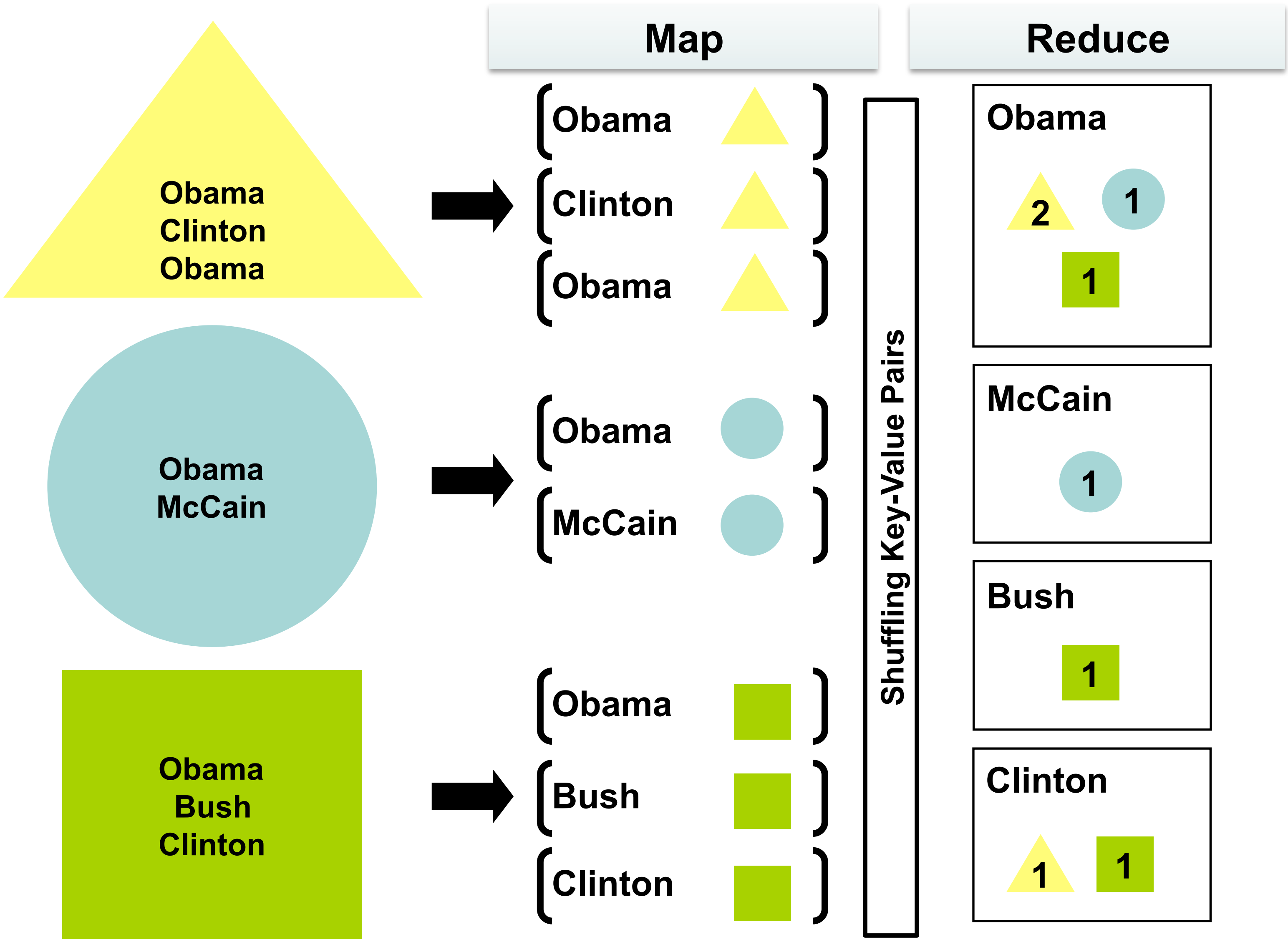
Case study #1: Inverted Index

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

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

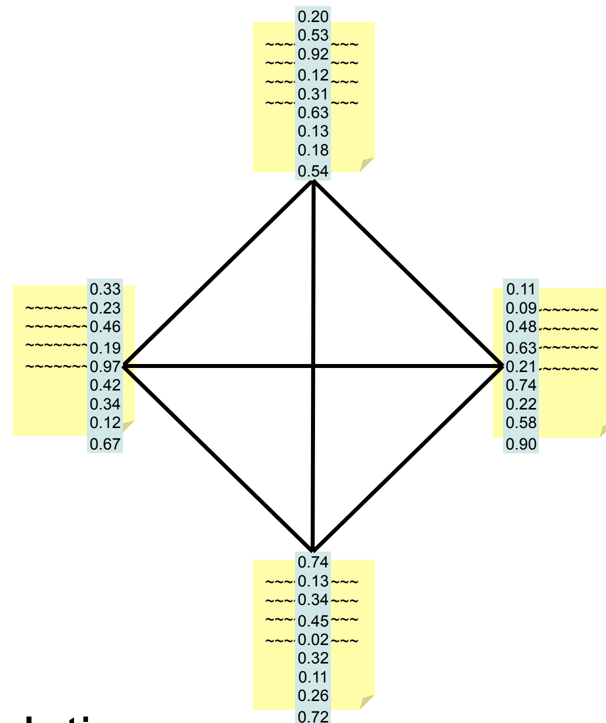
Map Reduce Algorithm: Inverted Index



Questions?

Case study #2:
pairwise similarity comparison

Pairwise Document Similarity



- Applications:
  - Clustering
  - Cross-document coreference resolution
  - “more-like-that” queries

Problem Description

- Consider similarity functions of the form:

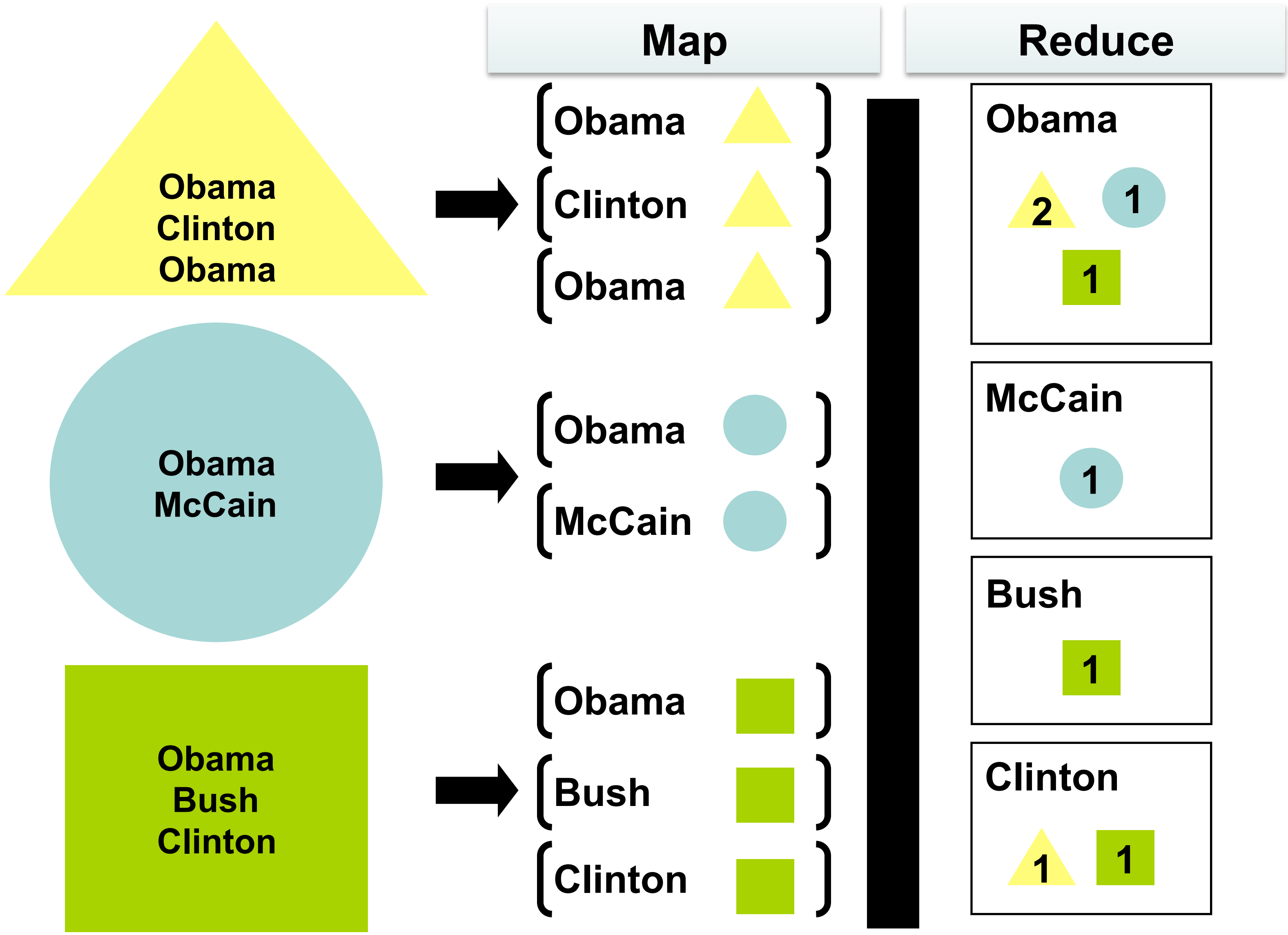
$$sim(d_i, d_j) = \sum_{t \in V} w_{t,d_i} w_{t,d_j}$$

  - But, actually…

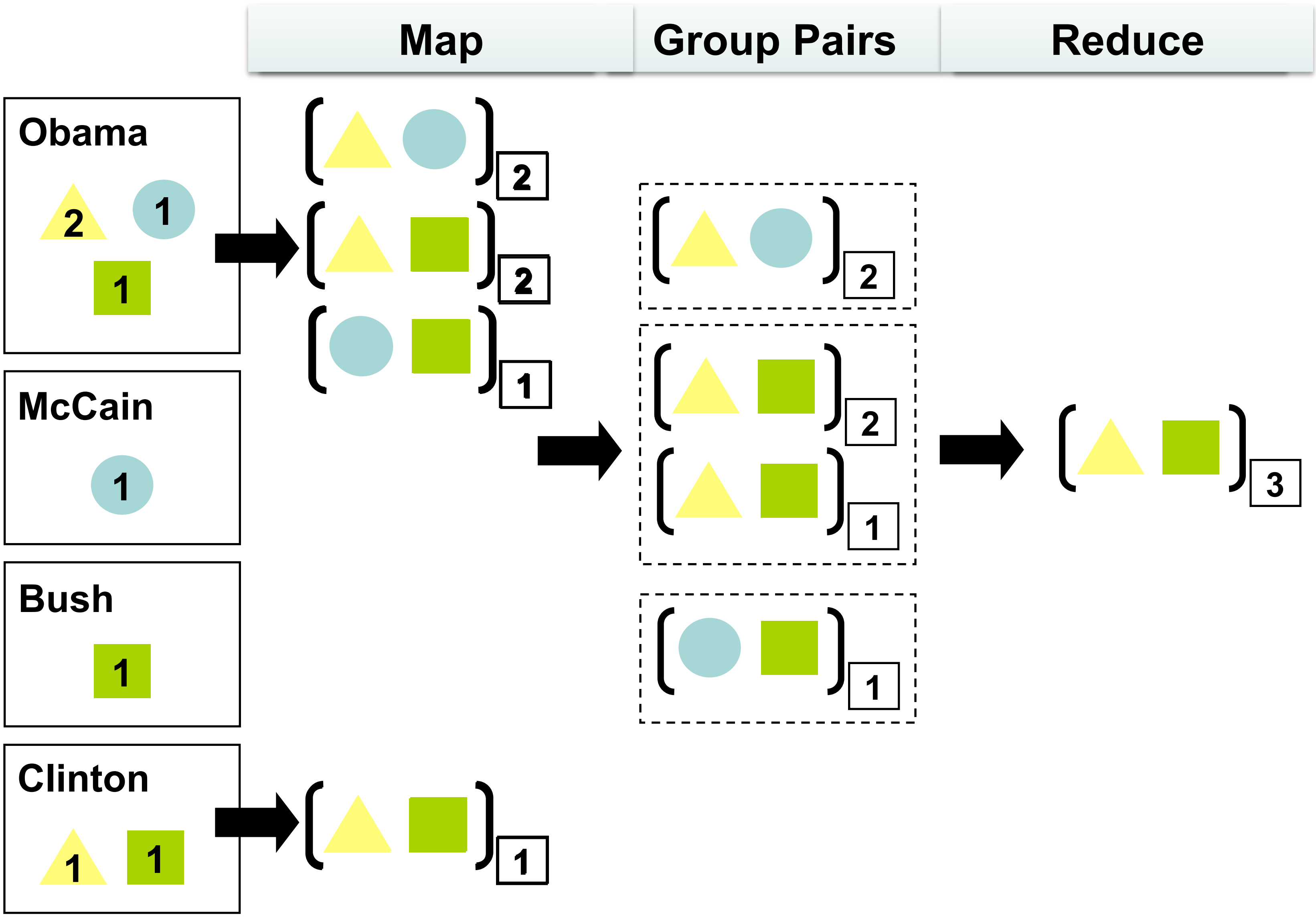
$$sim(d_i, d_j) = \sum_{t \in d_i \cap d_j} w_{t,d_i} w_{t,d_j}$$

- Two step solution in MapReduce:
  1. Build inverted index
  2. Compute pairwise similarity from postings

Building the Inverted Index

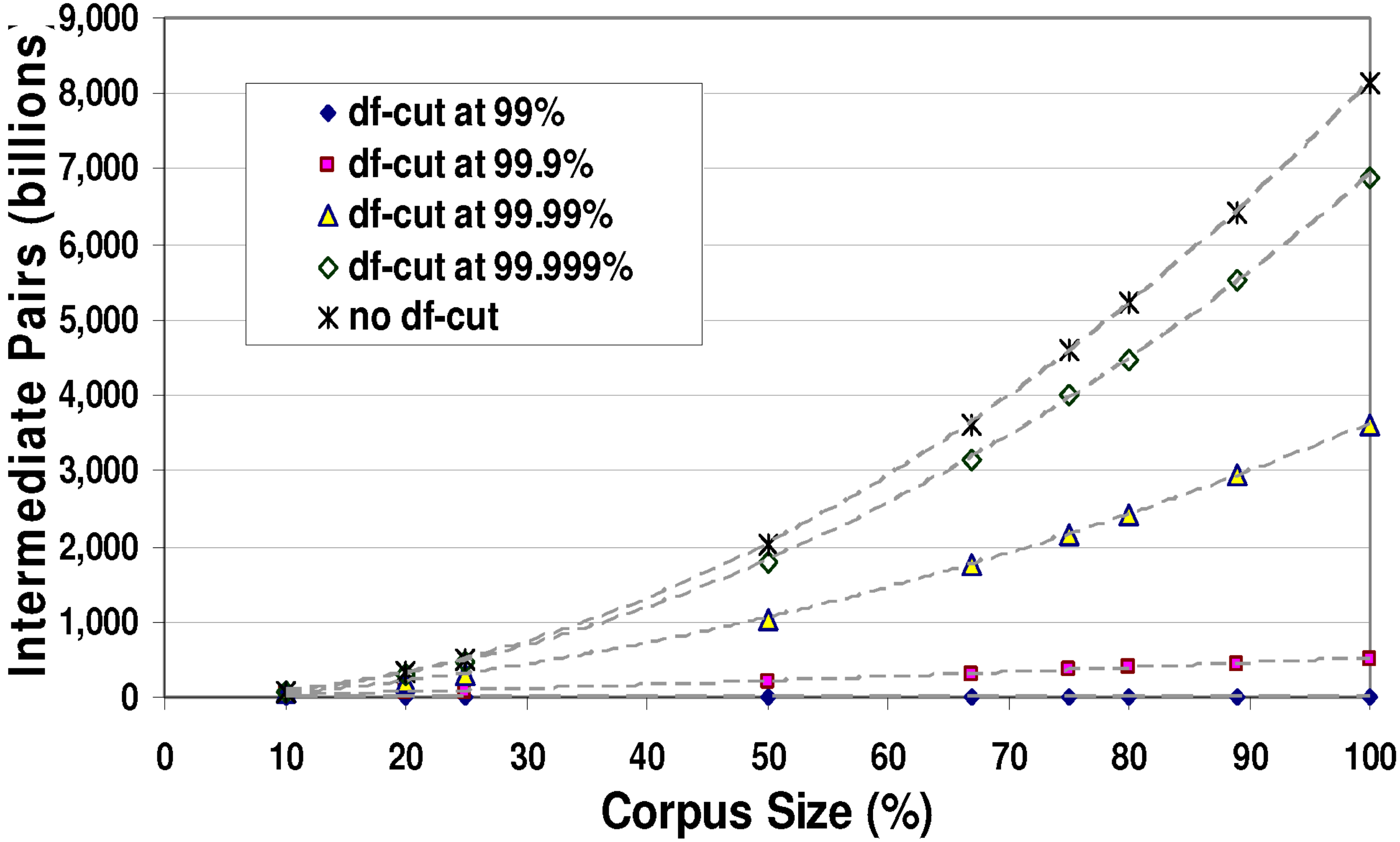


Computing Pairwise Similarity



Analysis

- Main idea: access postings once
  - $O(df^2)$ pairs are generated
  - MapReduce automatically keeps track of partial weights
- Control effectiveness-efficiency tradeoff by dfCut
  - Drop terms with high document frequencies
  - Has large effect since terms follow Zipfian distribution



Data Source: subset of AQUAINT-2 collection of newswire articles; approximately 900k documents.

Exercise 3: TF-IDF

- Term Frequency
  - Inverse Document Frequency
  - Relevant to text processing
  - Common web analysis algorithm

The Algorithm, Formally

$$\mathrm{tf_i} = \frac{n_i}{\sum_k n_k}$$

$$\mathrm{idf_i} = \log \frac{|D|}{|\{d : t_i \in d\}|}$$

$$\mathrm{tfidf} = \mathrm{tf} \cdot \mathrm{idf}$$

- D – the set of documents
- |D| - the number of documents

Information We Need

- Number of times term X appears in a given document
- Number of terms in each document
- Number of documents X appears in
- Total number of documents

Job 1: Word Frequency in Doc

- Mapper
  - –Input: docname→ contents
  - –Output: (word, docname) →1
- Reducer –Sums counts for word in document
  - –Outputs: (word, docname) → n
- Combiner is same as Reducer

Job 2: Word Counts For Docs

- Mapper
  - –Input: (word, docname)→ n
  - –Output: docname → (word, n)
- Reducer
  - –Sums frequency of individual n's in same doc
  - –Feeds original data through
  - –Outputs (word, docname)→ (n, N)

Job 3: Word Frequency in Corpus

- Mapper
  - –Input: ((word, docname), (n, N))
  - –Output: (word, (docname, n, N, 1))
- Reducer
  - –Sums counts for word in corpus
  - –Outputs ((word, docname), (n, N, m))

Thoughts on TF-IDF

- Several small jobs add up to full algorithm
- Lots of code reuse possible
  - Stock classes exist for aggregation, identity
- Jobs 3 and 4 can really be done at once in same reducer, saving a write/read cycle
- Very easy to handle medium-large scale, but must take care to ensure flat memory usage for largest scale

Job 4: Calculate TF-IDF

- Mapper
  - –Input: ((word, docname), (n, N, m))
  - –Assume D is known (or, easy MR to find it)
  - –Output ((word, docname), TF*IDF)
- Reducer
  - –Just the identity function

Whatever the good tools you have, algorithms are always the KEY.