بسم الله الرحمن الرحيم

CMPS 653
Big Data Analytics

Session 4:
MapReduce & Unstructured Data (Text)

Some slides are adopted from Jimmy Lin's by permission

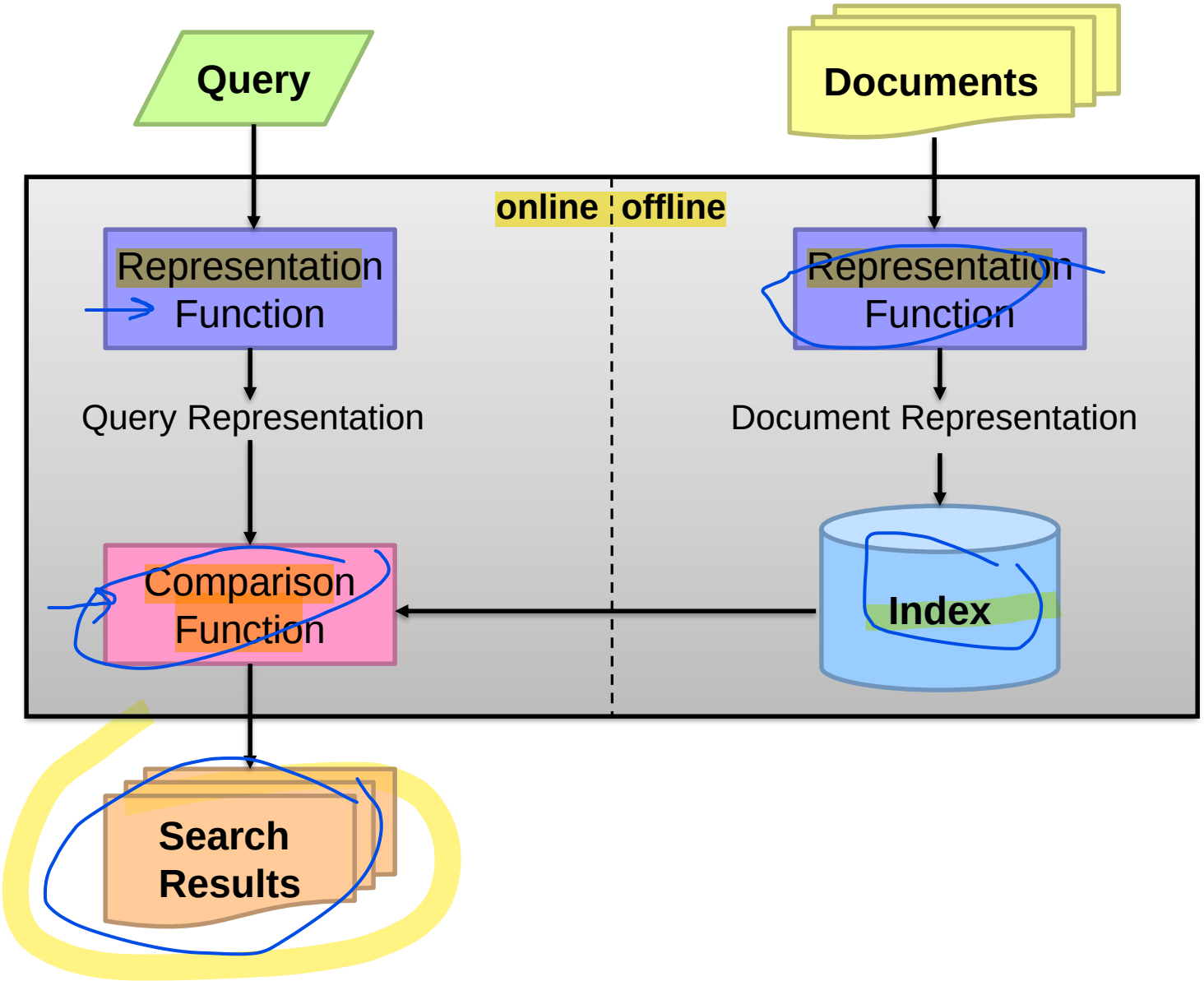
Tamer Elsayed

Source: Google

INFORMATION RETRIEVAL (IR) IN A NUTSHELL!

Information Retrieval is all about …

Learn IR in 2 Minutes ☺



Source: Google

Inverted Index: Boolean

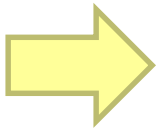
Doc 1: one fish, two fish

Doc 2: red fish, blue fish

Doc 3: red cat in the hat

Doc 4: green eggs and ham

	1	2	3	4
blue		1		
cat			1	
egg				1
fish	1	1		
green				1
ham				1
hat			1	
one	1			
red		1	1	
two	1			



blue	2
cat	3
egg	4
fish	1 → 2
green	4
ham	4
hat	3
one	1
red	2 → 3
two	1

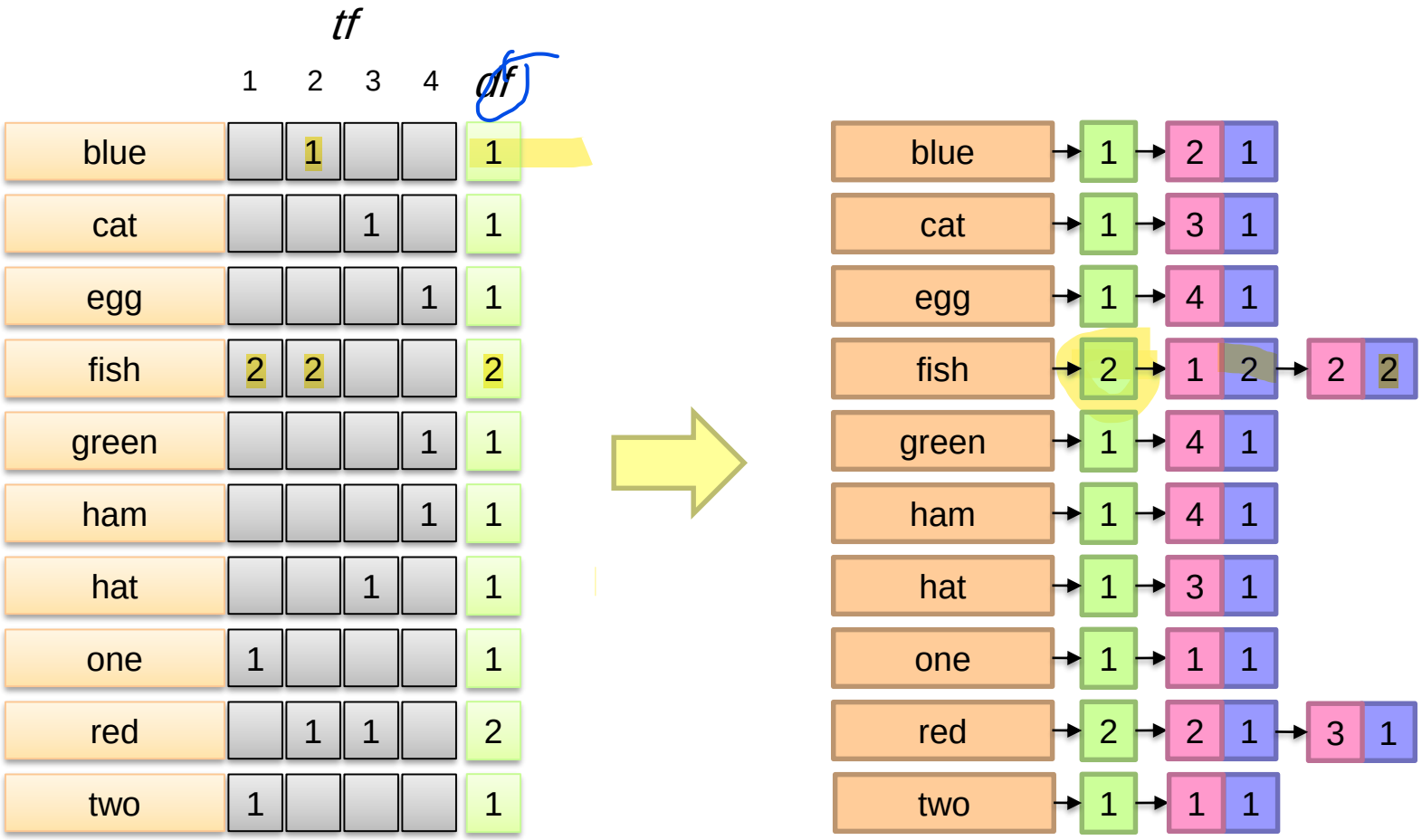
Inverted Index: TF & DF

Doc 1 one fish, two fish

Doc 2 red fish, blue fish

Doc 3 red cat in the hat

Doc 4 green eggs and ham



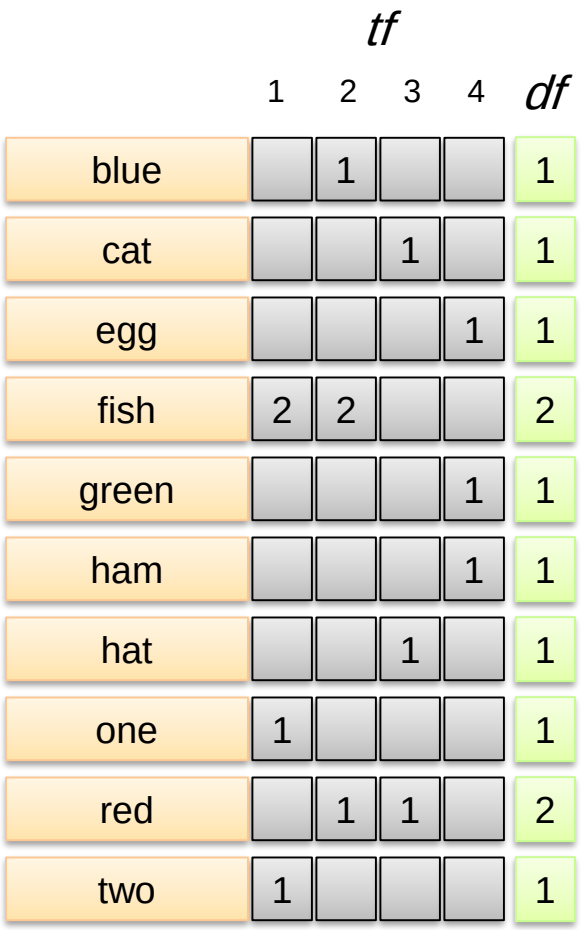
Inverted Index: Positions

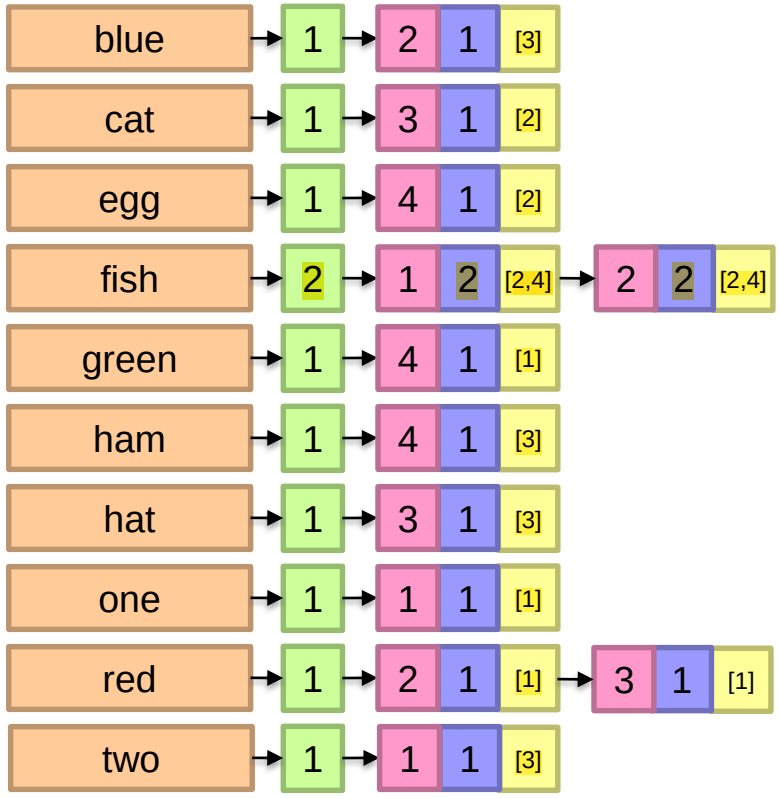
Doc 1 one fish, two fish

Doc 2 red fish, blue fish

Doc 3 red cat in the hat

Doc 4 green eggs and ham





Indexing with MapReduce

Source: Facebook

Inverted Indexing

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
red cat in the hat

Map

	Doc 1	Doc 2	Doc 3
one	1 1		
two	1 1		
fish	1 2		
red		2 1	3 1
blue		2 1	
fish		2 2	
cat			3 1
hat			3 1

Shuffle and Sort: aggregate values by keys

Reduce

cat	3 1	
fish	1 2	2 2
one	1 1	
red	2 1	3 1
blue	2 1	
hat	3 1	
two	1 1	

MapReduce: Index Construction

- Map over all documents
  - Emit *term* as key, (*docno*, *tf)* as value
  - Emit other information as necessary (e.g., term position)
- Sort/shuffle: group postings by term
- Reduce
  - Gather and sort the postings (e.g., by *docno* or *tf*)
  - Write postings to disk
- MapReduce does all the heavy lifting!

Inverted Indexing: Pseudo-Code

```
1: class Mapper
2:     method Map(docid n, doc d)
3:         H ← new AssociativeArray                    ▷ histogram to hold term frequencies
4:         for all term t ∈ doc d do                   ▷ processes the doc, e.g., tokenization and stopword removal
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting ⟨n, H{t}⟩)         ▷ emits individual postings

1: class Reducer
2:     method Reduce(term t, postings [⟨n1, f1⟩ ...])
3:         P ← new List
4:         for all ⟨n, f⟩ ∈ postings [⟨n1, f1⟩ ...] do
5:             P.Append(⟨n, f⟩)                        ▷ appends postings unsorted
6:         P.Sort()                                    ▷ sorts for compression
7:         Emit(term t, postingsList P)
```

Positional Indexes

Map

Doc 1
one fish, two fish

one	1	1	[1]
two	1	1	[3]
fish	1	2	[2,4]

Doc 2
red fish, blue fish

red	2	1	[1]
blue	2	1	[3]
fish	2	2	[2,4]

Doc 3
red cat in the hat

red	3	1	[1]
cat	3	1	[2]
hat	3	1	[3]

Shuffle and Sort: aggregate values by keys

Reduce

cat	3	1	[2]			
fish	1	2	[2,4]	2	2	[2,4]
one	1	1	[1]			
red	2	1	[1]	3	1	[1]

blue	2	1	[3]
hat	3	1	[3]
two	1	1	[3]

What's the Problem Here?!

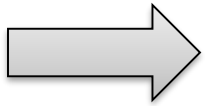
```
1: class Mapper
2:     method Map(docid n, doc d)
3:         H ← new AssociativeArray                ▷ histogram to hold term frequencies
4:         for all term t ∈ doc d do      ▷ processes the doc, e.g., tokenization and stopword removal
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting ⟨n, H{t}⟩)                ▷ emits individual postings

1: class Reducer
2:     method Reduce(term t, postings [⟨n1, f1⟩ ...])
3:         P ← new List
4:         for all ⟨n, f⟩ ∈ postings [⟨n1, f1⟩ ...] do
5:             P.Append(⟨n, f⟩)                          ▷ appends postings unsorted
6:         P.Sort()                                      ▷ sorts for compression
7:         Emit(term t, postingsList P)
```

Scalability Bottleneck!

- Reducers must buffer all postings assoc. with key (to sort)
- What if we run out of memory to buffer postings?

Another Try...

(key)	(values)		
fish	1	2	[2,4]
	34	1	[23]
	21	3	[1,8,22]
	35	2	[8,41]
	80	3	[2,9,76]
	9	1	[9]



(keys)		(values)
fish	1	[2,4]
fish	9	[9]
fish	21	[1,8,22]
fish	34	[23]
fish	35	[8,41]
fish	80	[2,9,76]

How is this different?

Let the framework do the sorting

Where have we seen this before?

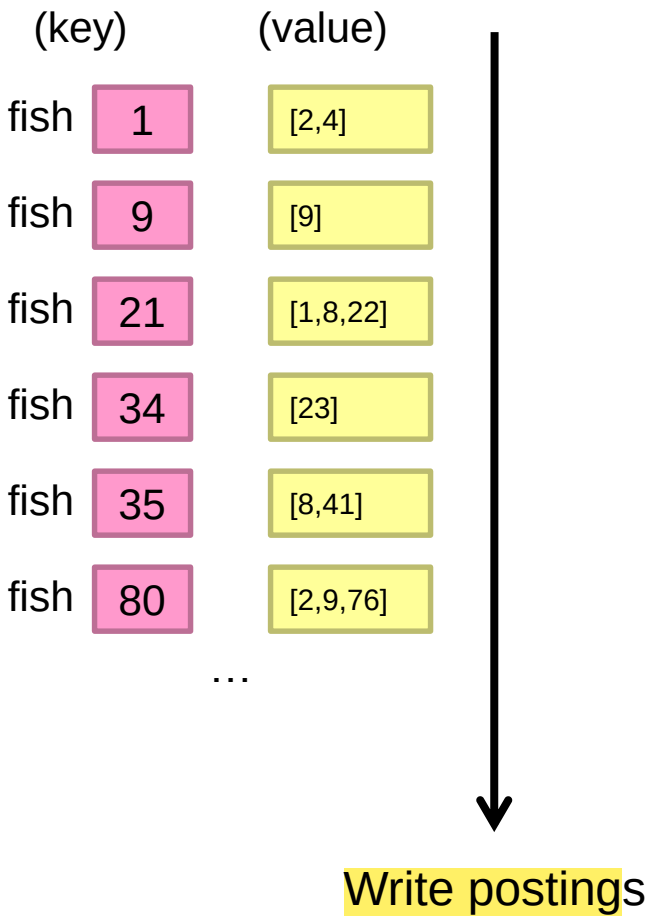
Inverted Indexing: Pseudo-Code

```
1: class Mapper
2:     method Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do                ▷ builds a histogram of term frequencies
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(tuple ⟨t, n⟩, tf H{t})          ▷ emits individual postings, with a tuple as the key

1: class Partitioner
2:     method Partition(tuple ⟨t, n⟩, tf f)
3:         return Hash(t) mod NumOfReducers         ▷ keys of same term are sent to same reducer
```

Inverted Indexing: Pseudo-Code

```
1: class Mapper
2:     method Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do                ▷ builds a histogram of term frequencies
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(tuple ⟨t, n⟩, tf H{t})          ▷ emits individual postings, with a tuple as the key

1: class Partitioner
2:     method Partition(tuple ⟨t, n⟩, tf f)
3:         return Hash(t) mod NumOfReducers         ▷ keys of same term are sent to same reducer

1: class Reducer
2:     method Initialize
3:         t_prev ← ∅
4:         P ← new PostingsList
5:     method Reduce(tuple ⟨t, n⟩, tf [f])
6:         if t ≠ t_prev ∧ t_prev ≠ ∅ then
7:             Emit(term t, postings P)             ▷ emits postings list of term t_prev
8:             P.Reset()
9:         P.Append(⟨n, f⟩)                         ▷ appends postings in sorted order
10:        t_prev ← t
11:    method Close
12:        Emit(term t, postings P)                 ▷ emits last postings list from this reducer
```

Chicken and Egg?



We' need *df* for compression of the postings list

But we don't know the *df* until we've seen all postings!

Sound familiar?

Getting the *df*

- In the mapper:
  - Emit "special" key-value pairs to keep track of *df*
- In the reducer:
  - Make sure "special" key-value pairs come first: process them to determine *df*
- Remember: proper partitioning!

Getting the *df*: Modified Mapper

Doc 1
one fish, two fish

Input document…

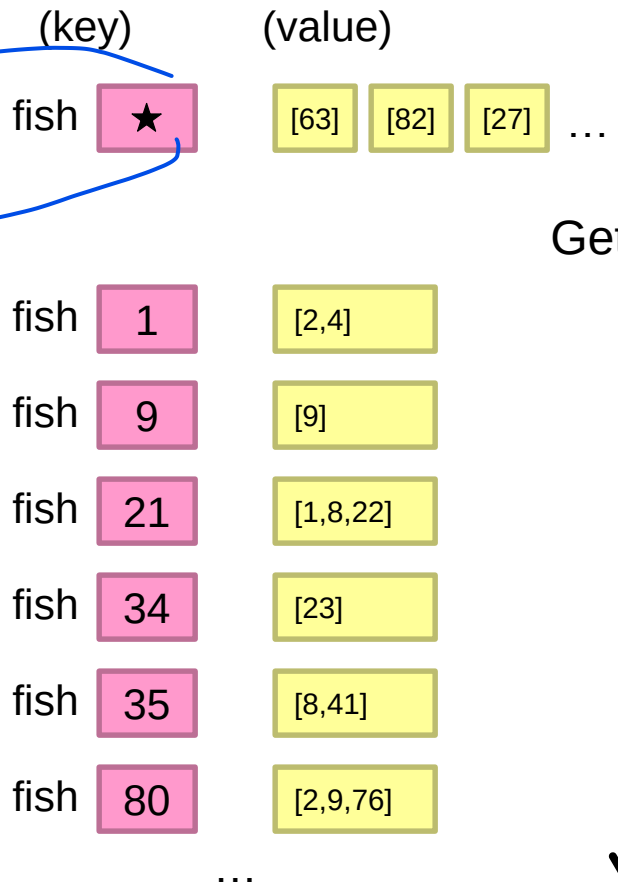
(key)		(value)
fish	1	[2,4]
one	1	[1]
two	1	[3]

Emit normal key-value pairs…

fish	★	[1]
one	★	[1]
two	★	[1]

Emit "special" key-value pairs to keep track of *df*…

Getting the *df*: Modified Reducer



First, compute the *df* by summing contributions from all "special" key-value pair…

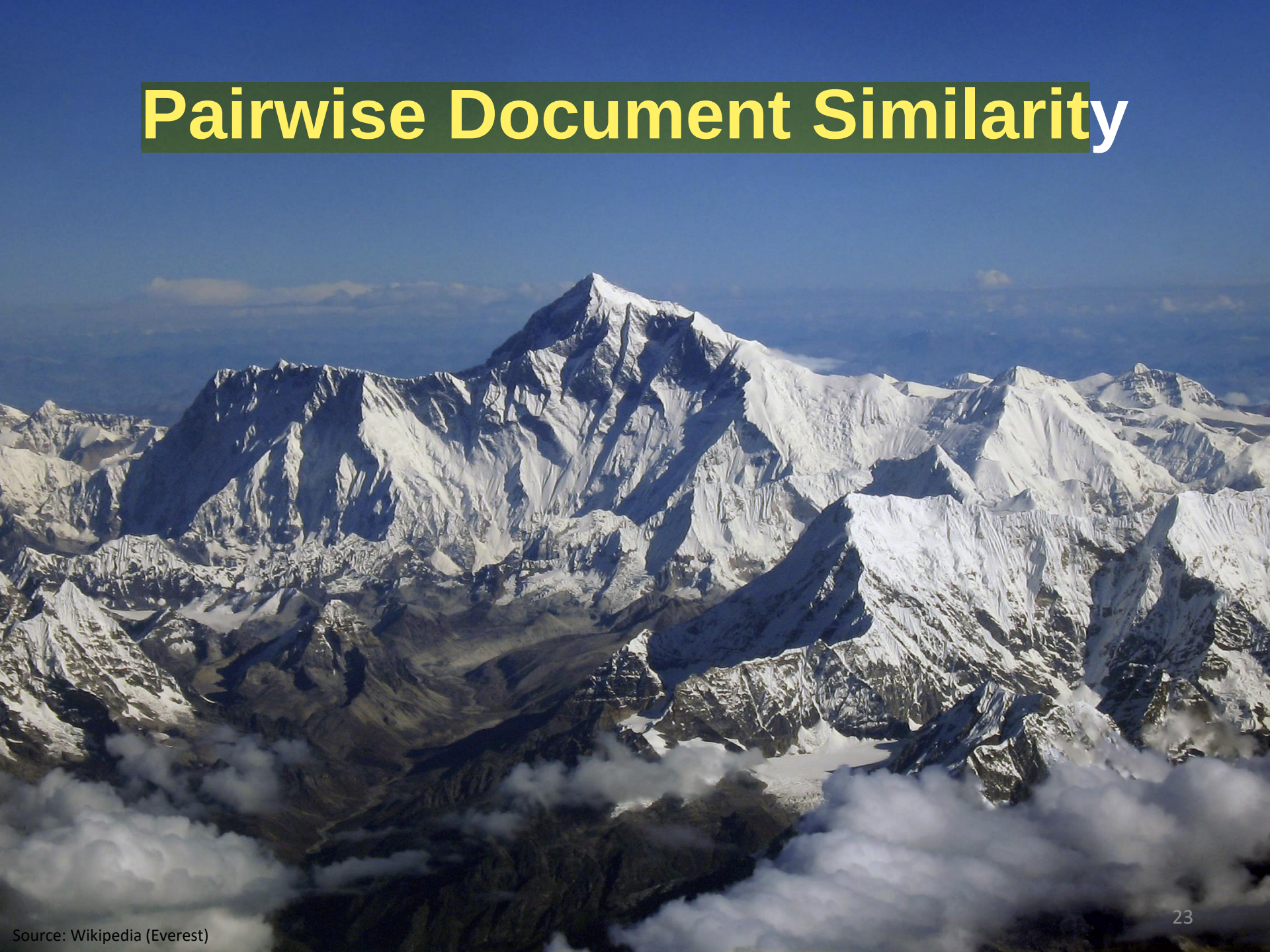
Get *df*…

Important: properly define sort order to make sure "special" key-value pairs come first!

Write postings

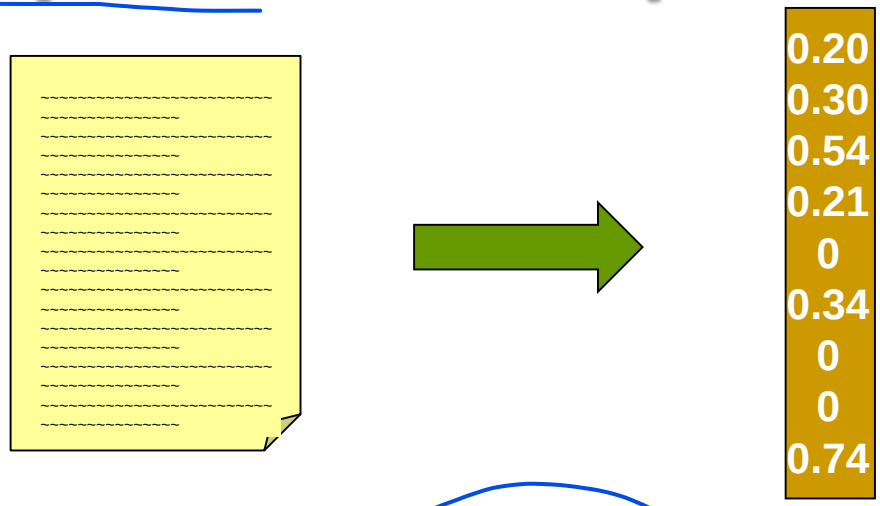
Where have we seen this before?

Pairwise Document Similarity



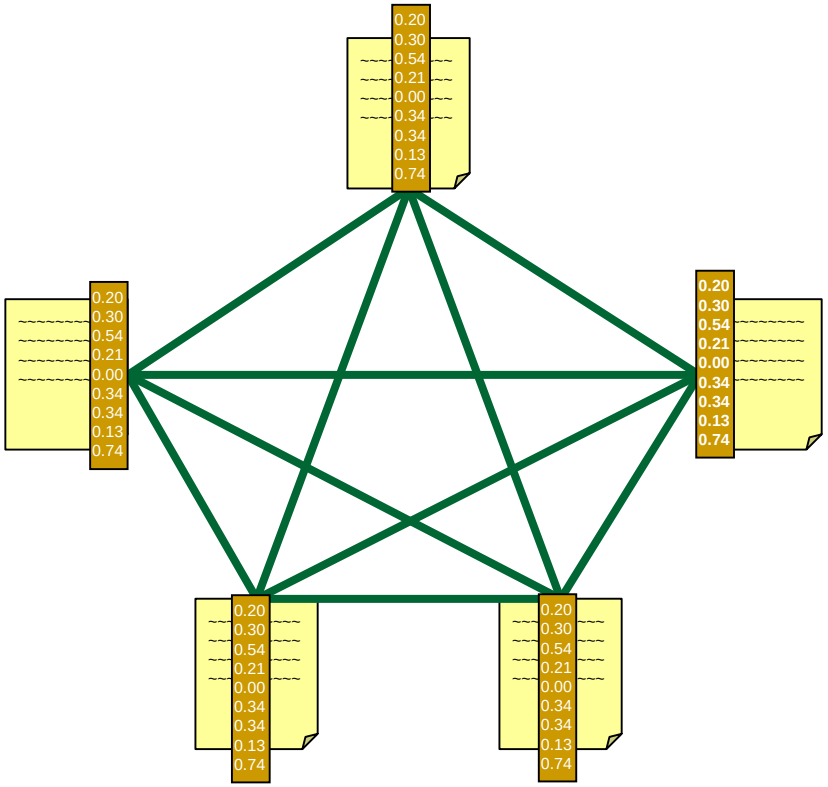
Source: Wikipedia (Everest)

Vector Space Model (VSM)



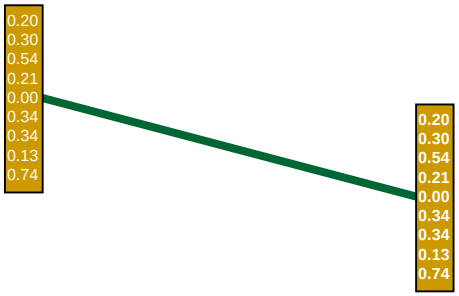
- Represent documents as vectors
- Each term has a weight (0 if not exist)
  - term weights indicate importance
- Sparse
- [Generally: feature vector]

Abstract Problem



- Applications:
  - Near-Duplicate Detection
  - Clustering
  - Friends recommendation in Facebook/Twitter
  - “more-like-that” queries

Similarity in VSM



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

- **Dot product** between vectors
- Represent a large class of similarity metrics (e.g., cosine similarity, BM25, etc.)

Better Solution

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

Each term contributes only if appears in $d_i \cap d_j$

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

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

Can we benefit from that?

Algorithm

Algorithm 1 Compute Pairwise Similarity Matrix

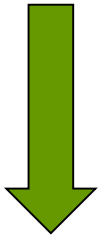
1: $\forall i, j : sim[i, j] \Leftarrow 0$
2: **for all** $t \in V$ **do**
3: $\quad p_t \Leftarrow postings(t)$
4: $\quad$ **for all** $d_i, d_j \in p_t$ **do**
5: $\quad\quad sim[i, j] \Leftarrow sim[i, j] + w_{t,d_i} \cdot w_{t,d_j}$

- Load weights for each term once
- Each term contributes O(df_t^2) partial scores
- Matrix must fit in memory
  - Works for small collections
- Otherwise: disk access optimization

Large Collections!

Goal

scalable and efficient solution for large collections



MapReduce ?!

Algorithm

Algorithm 1 Compute Pairwise Similarity Matrix

1: $\forall i, j : sim[i, j] \Leftarrow 0$
2: **for all** $t \in V$ **do**
3: $\quad p_t \Leftarrow postings(t)$
4: $\quad$ **for all** $d_i, d_j \in p_t$ **do**
5: $\quad\quad sim[i, j] \Leftarrow sim[i, j] + w_{t,d_i} \cdot w_{t,d_j}$

0 Indexing

1 Partial contributions

for each term in parallel

2 Aggregation

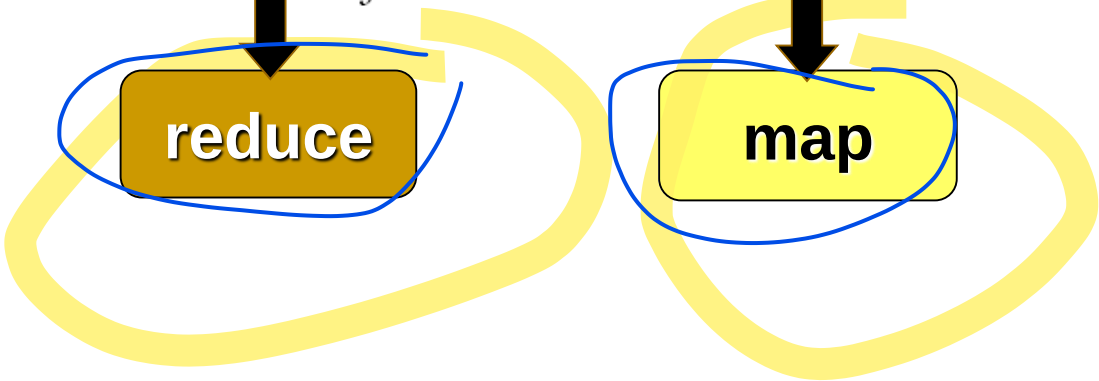
for each doc-pair in parallel

Decomposition

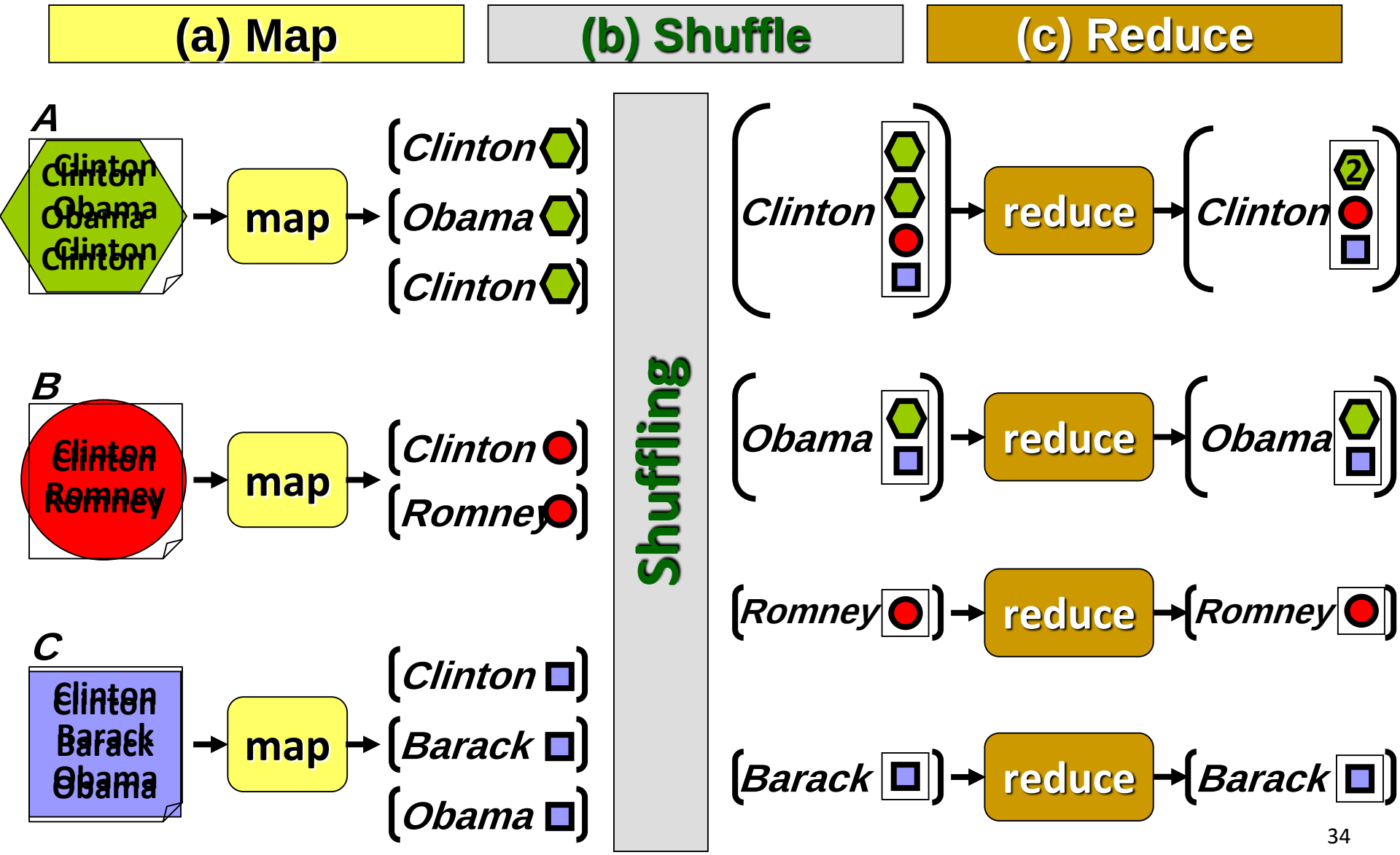
Each term contributes only if appears in $d_i \cap d_j$

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

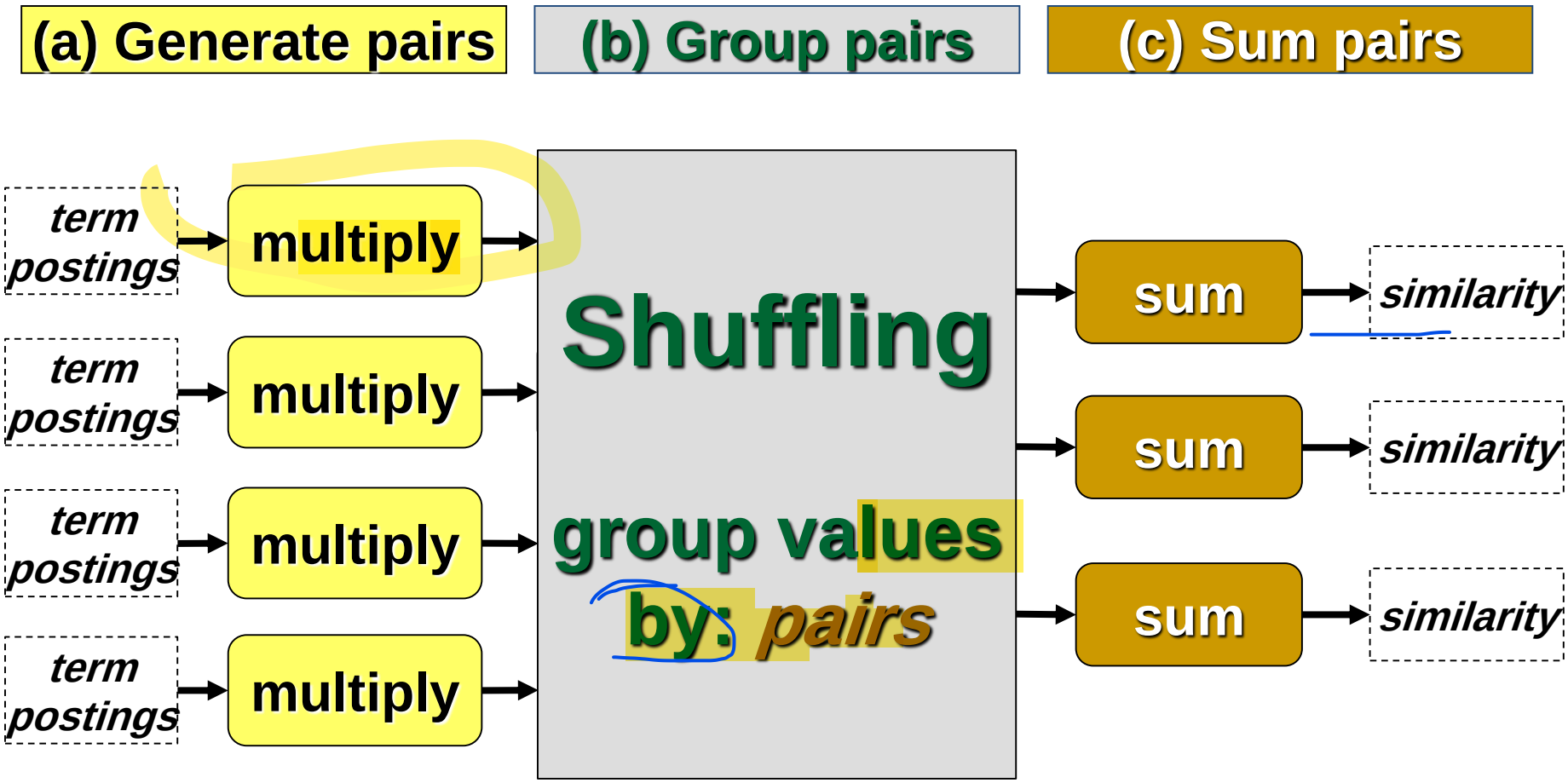
$$sim(d_i, d_j) = \sum_{t \in d_i \cap d_j} term\_contrib(t, d_i, d_j)$$



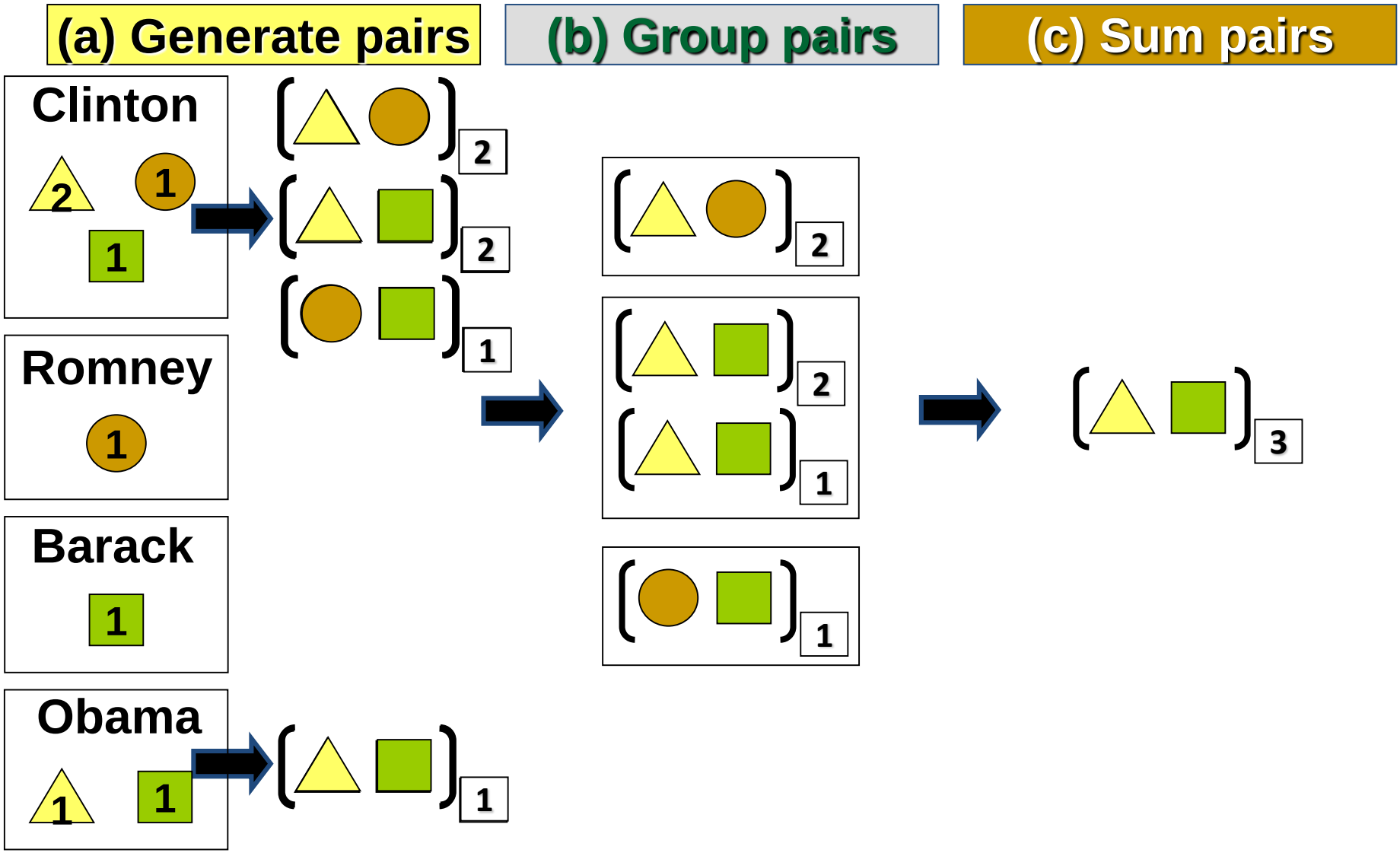
Step 1: Indexing



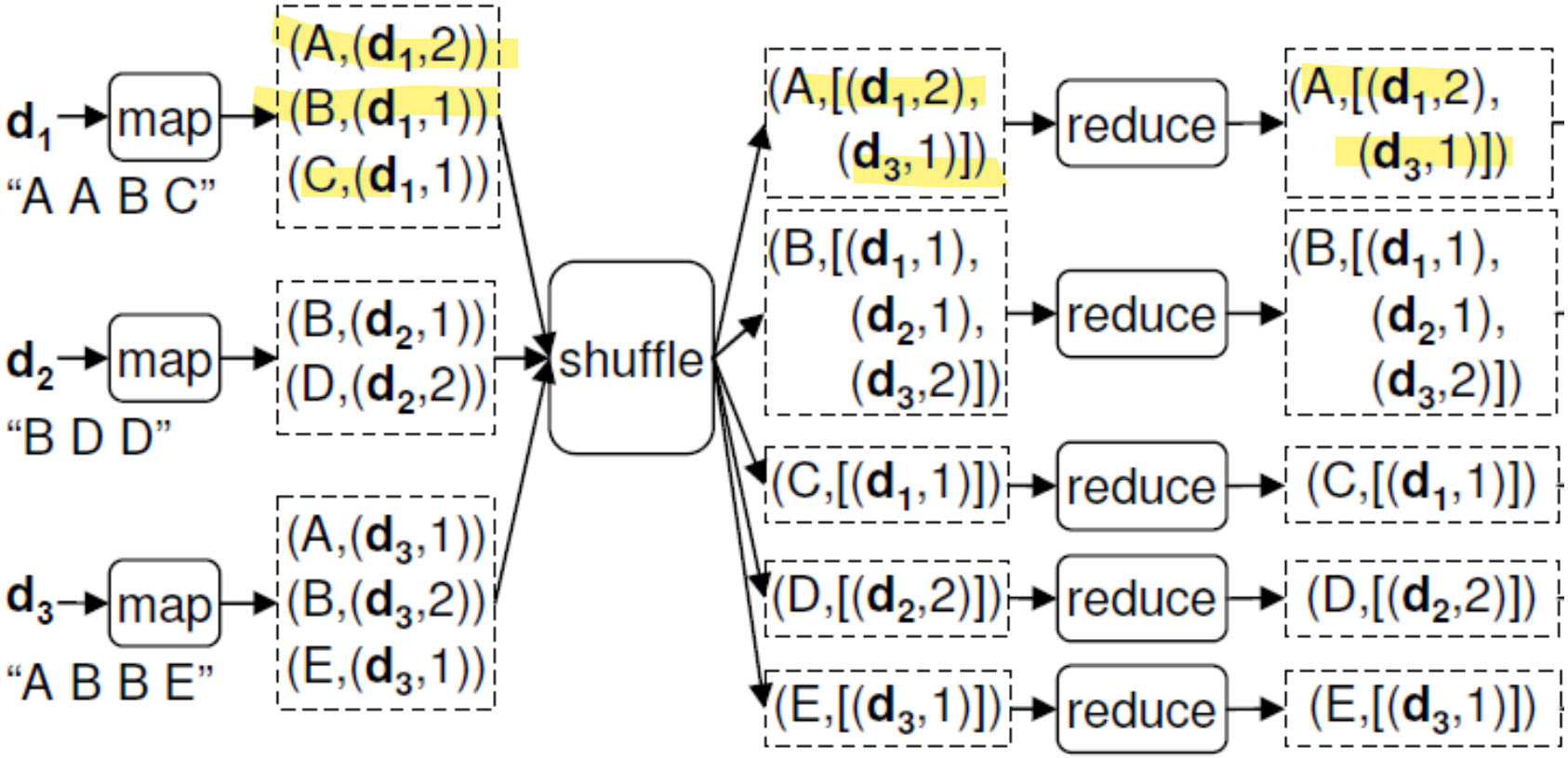
Step 2: Pairwise Similarity



Step 2: Pairwise Similarity



Another Ex.: Indexing Step



Another Ex: Pairwise Sim. Step