Pairs and Stripes

Motivation

- Co-occurring terms
- Inverted Index
- Synonym Detection

Synchronization in Streams

- Calculating the mean
  - Example: average spend per customer
- Complex keys that bring data together by the execution framework
- Order inversion

Market Basket Analysis

Where should detergents be placed in the store to maximize their sales? ?

Are window cleaning products purchased when detergents and orange juice are bought together? ?

Is soda typically purchased with bananas?
Does the brand of soda make a difference? ?

How are the demographics of the neighborhood affecting what customers are buying? ?

Co-Occurence Matrix

	OJ	Window Cleaner	Milk	Soda	Detergent
OJ	4	1	1	2	1
Window Cleaner	1	2	1	1	0
Milk	1	1	1	0	0
Soda	2	1	0	3	1
Detergent	1	0	0	1	2

Algorithm Design: Running Example

Term co-occurrence matrix for a text collection

- M = N × N matrix (N = vocabulary size)
- M_{ij}: number of times i and j co-occur in the same context (for concreteness, let's say context = sentence)
  - Corpus: ADCEADEBACED
- Using these context vectors, you can get co-occurrences of D and E, D[E] = 4

Context vectors:

	A	B	C	D	E
A	0	1	3	2	3
B	1	0	1	0	1
C	3	1	0	2	2
D	2	0	2	0	4
E	3	1	2	4	0

Algorithm Design: Running Example (cont.)

- M = N × N matrix (N = vocabulary size)
- M_{ij}: number of times i and j co-occur in the same 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

Large Counting Problems

- Term co-occurrence matrix for text collection = specific instance of a large counting problem
  - A large event space (number of terms)
  - A large number of observations (the collection itself)
  - Goal: keep track of interesting statistics about the events
- Combinatorial event space and large number of observations
- How do we aggregate partial counts efficiently?

Co-Occurrence Matrix: First Try "Pairs"

Each pair corresponds to a cell in the word co-occurrence matrix. This algorithm illustrates the use of complex keys in order to coordinate distributed computations.

	A	B	C	D	E
A	0	1	3	2	3
B	1	0	1	0	1
C	3	1	0	2	2
D	2	0	2	0	4
E	3	1	2	4	0

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For all pairs, emit (a, b) → count

Co-Occurrence Matrix: First Try "Pairs" (cont.)

E.g.,

Doc1	ABCAC
EMITS	
A,B	1
A,C	1
B,A	1
B,C	1
Etc.	

- Reducers sum up counts associated with these pairs.
- Use combiners!

Pairs: Pseudo-Code

```
def emitPairs(row):
        words = row.split(“ ”)
        for all word in words:
                for all neighbor of word:
                        emit ((word,neighbor), 1) # emit count for co-
occurrences

rdd.flatMap(emitPairs)\
   .reduceByKey(lambda a,b: a+b) # sum co-occurrence counts
```

Each pair corresponds to a cell in the word co-occurrence matrix. This algorithm illustrates the use of complex keys in order to coordinate distributed computations.

"Pairs" Analysis

- Advantages
  - Easy to implement, easy to understand
- Disadvantages
  - Lots of pairs to sort and shuffle around (upper bound?)
- How can we overcome this shuffle communication overhead?

Another Try: "Stripes"

- Idea: Group together pairs into an associative array

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

a → [b: 1, c: 2, d: 5, e: 3, f: 2}

- Each mapper takes a sentence.
  - Generates all co-occurring term pairs
  - For each term, emits 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}

Key: Cleverly constructed data structure brings together partial results

Stripes: Pseudo-Code

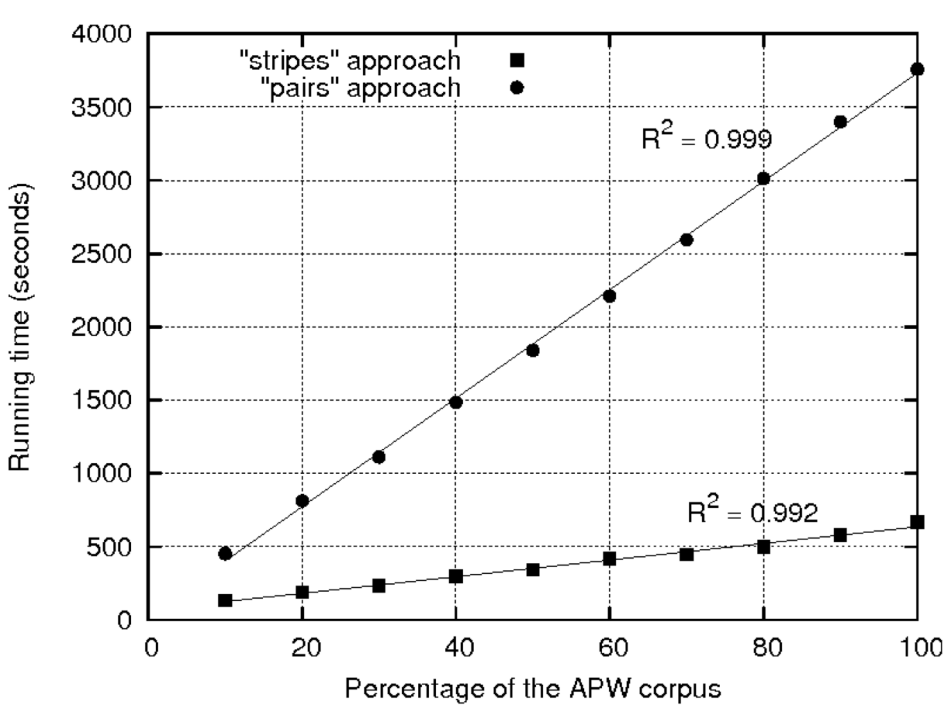
```
a → {b:1, c:2, d:5, e:3, f: 2}

def emitStripes(row):
        words = row.split(“ ”)
        for all word in words:
             H ← new ASSOCIATIVE_ARRAY
             for all neighbor of word:
                H{word} ← H{word} + 1
             emit (word, H) # emit stripe

def sumStripes(a,b):
        emit sum(a,b) # elementwise sum of {}

rdd.flatMap(emitStripes)\
   .reduceByKey(sumStripes)
```

"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 more heavyweight
  - Fundamental limitation in terms of size of event space

Performance

- Which is faster, stripes or pairs?
- Stripes has a bigger value per key—watch out for memory usage!
- Pairs has more partition and sort overhead.



Lin, J.J., & Dyer, C. *Data-intensive text processing with MapReduce*

The End

Relative Frequencies Revisited

Motivation

In this section, we'll take another look at relative frequencies, but this time for co-occurring terms.

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

f(B | A): Stripes

REDUCER A → {Count (A; B)}
a → {b_1 :2, b_2 :12, b_3 :5 b_4 :1}
ab_1 : 2/20
ab_2 : 12/20
ab_3 : 5/20
ab_4 : 1/20

2
+12
+5
+1
=20

- One loop over all value terms (co-occurring words) to compute (a, *)
- Another pass to directly compute the relative frequency f(B | A)

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

Next up: Pairs approach

- Mappers emit word pairs with count 1
- Reducers aggregate counts for each pair

key	value
A,B	1
A,C	1
B,A	1
B,C	1

Need to Reconstruct the List of Co-occurring Terms With the Terms of Interest

- Synchronization
  - Need to reconstruct the list of co-occurring terms with the term of interest
- Fortunately, as in the mapper, the reducer can preserve state across multiple keys
- Inside the reducer, we can buffer in memory all the words that co-occur with **wi** and their counts, in essence building the associative array in the stripes approach

Custom Partitioner:
To sync word counts for word of interest

hash("**dog,aardvark**") % nPartitions

≠

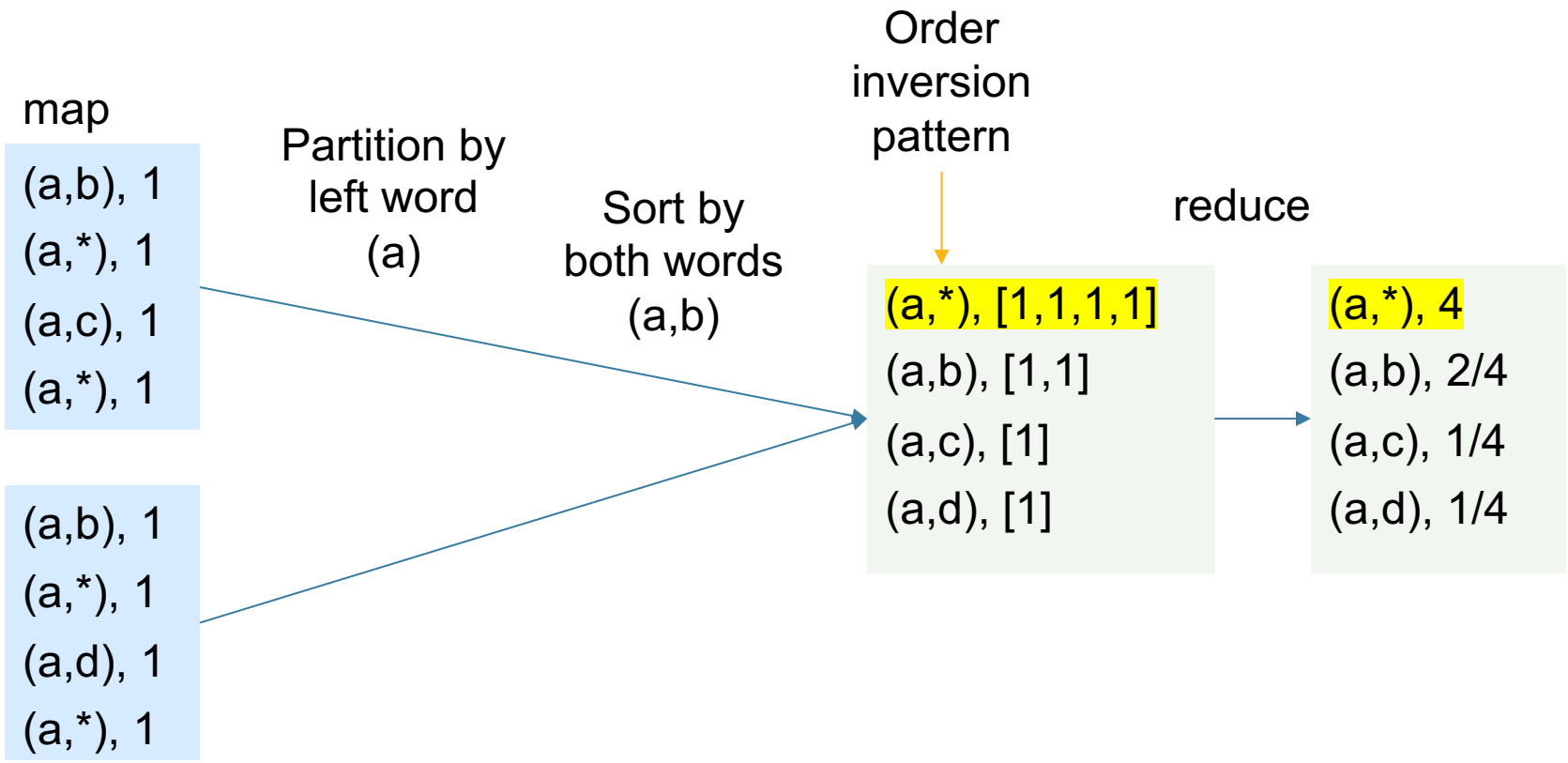
hash("**dog,zebra**") % nPartitions

Disadvantages

- In-memory buffer on the reducer side
- Potential to run out of memory

- Order-inversion pattern to the rescue again!

Custom Partitioner:
To sync word counts for word of interest



Spark Code Example

```
def makePairs(row):
    words = row.split(‘ ')
    for w1, w2 in combinations(words, 2):
        yield((w1,"*"),1)
        yield((w1,w2),1)

def partitionByWord(x):
    return hash(x[0][0])

def calcRelFreq(row):
    seq = sorted(seq, key=lambda tup: (tup[0][0], tup[0][1]))
    currPair, currWord, = None, None
    pairTotal, wordTotal = 0, 0
    for r in list(row):
        w1, w2 = r[0][0], r[0][1]
        if w2 == "*":
            if w1 != currWord:
                wordTotal = 0
                currWord = w1
            wordTotal += r[1]
        else:
            pairTotal += r[1]

            if currPair != r[0]:
                yield(w1+" - "+w2, pairTotal/wordTotal)
                pairTotal = 0
                currPair = r[0]
```

```
RDD = DATA.flatMap(makePairs)\
          .reduceByKey(add,
             partitionFunc=partitionByWord)\
          .mapPartitions(calcRelFreq, True)

RDD.glom().collect()
```

RESULT:

```
[
    [
    ('bear - pig', 0.5),
    ('bear - zebra', 0.5),
    ('dog - aardvark', 0.33),
    ('dog - banana', 0.33),
    ('dog - pig', 0.33),
    ('pig - banana', 1.0)
    ], [
    ('aardvark - banana', 0.5),
    ('aardvark - pig', 0.5),
    ('zebra - pig', 1.0)
    ]
].
```

```
DATA = sc.parallelize([‘dog aardvark pig banana’,
                       ‘bear zebra pig’])
```

Summary

Computing Relative frequencies with the order-inversion pattern requires:

- Emitting a special key-value pair for each co-occurring word pair in the mapper to capture its contribution to the marginal.
- Controlling the sort order of the intermediate key so that the key-value pairs representing the marginal contributions are processed by the reducer before any of the pairs representing the joint word co-occurrence counts.
- Defining a custom partitioner to ensure that all pairs with the same left word are shuffled to the same reducer.
- Preserving state across multiple keys in the reducer to first compute the marginal based on the special key-value pairs and then dividing the joint counts by the marginals to arrive at the relative frequencies.

The End

Inverted Index

Definitions

- **Inverted index:** Given a term provides access to the list of documents that contain the term, an inverted index consists of postings lists, one associated with each term that appears in the collection.
- **Postings:** A postings list is comprised of individual postings, each of which consists of a document ID and a payload|information about occurrences of the term in the document. The simplest payload is …nothing! The most common payload, however, is term frequency (tf), or the number of times the term occurs in the document.

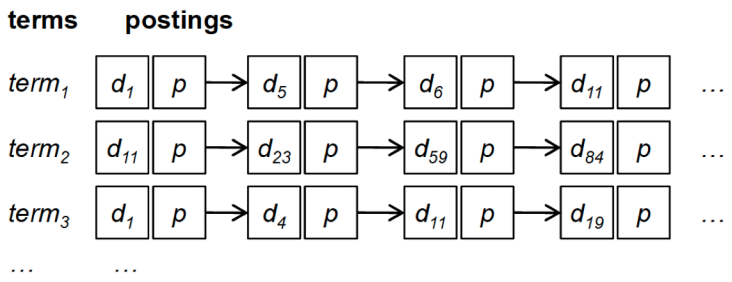


Figure 4.1: Simple illustration of an inverted index. Each term is associated with a list of postings. Each posting is comprised of a document id and a payload, denoted by p in this case. An inverted index provides quick access to documents ids that contain a term.

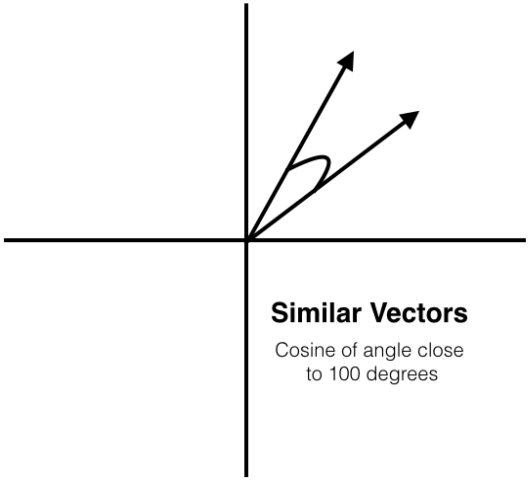
Applications

- Information retrieval (ie. web search)
- Document similarity
- Pairwise similarity

Similarity measures

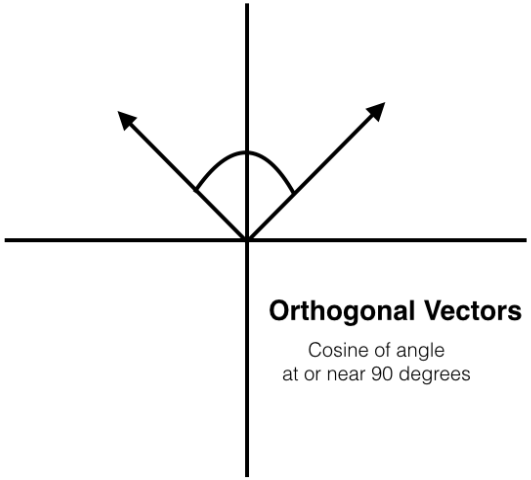
- Earth mover's distance
- Cosine Similarity
- Euclidean Distance
- Jaccard
- Overlap
- Dice
- Etc.

Cosine Similarity



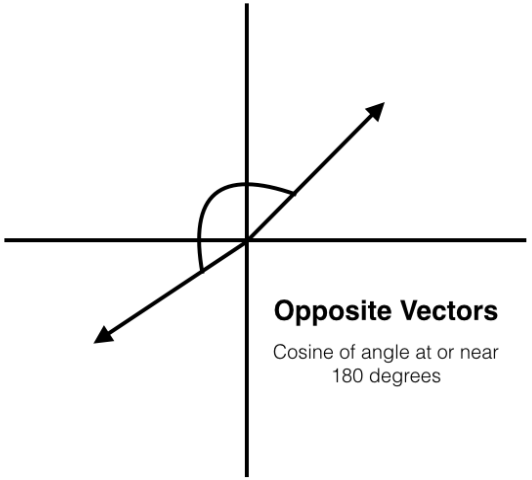
Similar Vectors

Cosine of angle close to 100 degrees



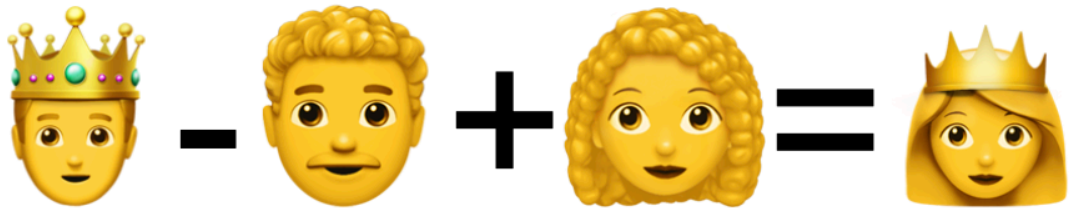
Orthogonal Vectors

Cosine of angle at or near 90 degrees



Opposite Vectors

Cosine of angle at or near 180 degrees



Words in context

"One should wear business attire to work."

"One should wear business clothing to work."

attire ~= clothing

Windowing

"One should wear business attire to work."

Matrix vs Stripes

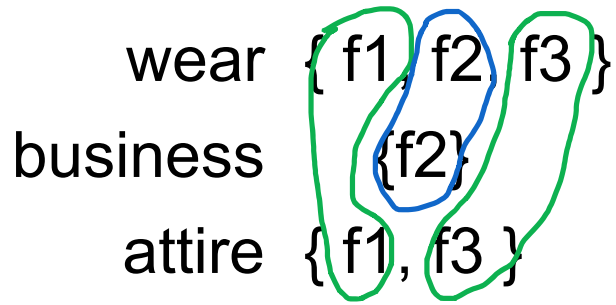
	f1	f2	f3
wear	1	1	1
business	0	1	0
attire	1	0	1

wear	{ f1, f2, f3 }
business	{ f2 }
attire	{ f1, f3 }

Where f1 – fn are the “features”, or in other words, the vocabulary

Computing Similarities

Stripes



Number of features in common

wear + attire : 2

wear + business : 1

attire + business : 0

Computing Similarities

Could we do this “in parallel”?

wear { f1, f2, f3 }	business { f2 }	attire { f1, f3 }
task 1	task 2	task 3

wear + business	wear + attire	attire + business
?	?	?

Computing Similarities

A third data structure…

	f1	f2	f3
wear	1	1	1
business	0	1	0
attire	1	0	1

matrix

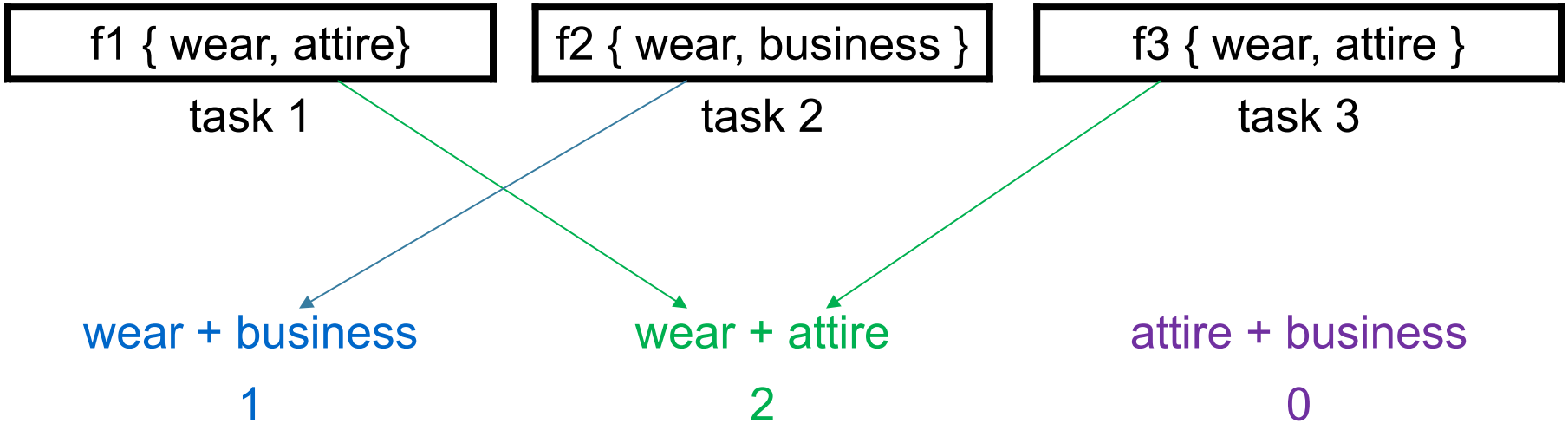
wear	{ f1, f2, f3 }
business	{ f2 }
attire	{ f1, f3 }

stripes

f1	{ wear:3, attire }
f2	{ wear, business }
f3	{ wear, attire }

inverted index

Computing Similarities



The End